

Progetto SO - Laboratorio di Sistemi Operativi

A.A. 2019-2020

AMACAp2p

1 Descrizione generale del progetto

1.1 Componenti del progetto

Arcara Alessio	matricola: 0900061028
Borghesi Andrea	matricola: 0000874412
Carchesio Michael	matricola: 0000889083
Crimaldi Alessia	matricola: 0000883306
Tagani Chiara	matricola: 0000893990

- Indirizzo mail del referente: *michael.carchesio@studio.unibo.it*

1.2 Funzionalità implementate e suddivisione del lavoro

Progettazione rete P2P	Borghesi, Carchesio, Tagani
Login nella rete P2P	Borghesi, Carchesio
Chat privata e di gruppo	Borghesi, Carchesio
Persistenza dei messaggi	Tagani
Monitor di rete	Carchesio
Concorrenza	Carchesio
Grafica con Swing	Arcara, Carchesio
Progettazione crittografia	Arcara, Crimaldi
Cifratura asimmetrica	Arcara
Integrità del messaggio e autenticazione	Crimaldi

La documentazione è stata scritta e documentata dall'intero gruppo.

Contenuto della documentazione

1 Descrizione generale del progetto	1
1.1 Componenti del progetto	1
1.2 Funzionalità implementate e suddivisione del lavoro	1
2 Istruzioni per la demo	4
3 Rete P2P decentralizzata	9
3.1 Connettersi alla rete	9
3.2 Avvio di una chat privata	16
3.3 Chat di gruppo	18
3.4 Partecipare ad una chat di gruppo	20
3.5 Gestione delle eccezioni, problemi e concorrenza in PeerB e PeerGroup	22
3.5.1 Gestione login contemporaneo di più peer	22
3.5.2 Gestione errore monitor	22
3.5.3 Gestione riconoscimento di un utente o di un gruppo	23
3.5.4 Concorrenza tra peer	26
3.5.5 Concorrenza in PeerGroup	29
4 Pubblicazione e scrittura dei messaggi	31
4.1 Persistenza dei messaggi e problemi di concorrenza	31
4.2 Creazione file e scrittura messaggi	31
4.3 Gestire le richieste concorrenti: Synchronized	37
5 Gestione del monitor	38
5.1 Introduzione	38
5.2 Struttura ConsoleStampa	38
5.3 Javaservice e struttura dei metodi utilizzati	40
6 Menù di scelta in JavaSwingConsole	43
7 Crittografia	46
7.1 Crittografia a chiave pubblica:	46
7.2 Algoritmo RSA:	46
7.2.1 L'algoritmo RSA non è sicuro:	48
7.2.2 Simplified OAEP:	49
7.2.3 Javaservice:	50
7.3 Algoritmo SHA:	51

7.3.1	Problematiche degli algoritmi di hash	51
7.3.2	Lato sender	54
7.3.3	Lato receiver	55

2 Istruzioni per la demo

Nota: Il programma è stato strutturato su un sistema Unix

Per comunicare tra due peer avviare due terminali differenti e in entrambi inserire il comando "jolie PeerAA.ol". È possibile, se lo si desidera, aprire anche un altro terminale in cui visionare la console tramite il comando "jolie ConsoleStampa.ol".



Figure 1: Monitor per stampa

Nella console di tracciamento, vengono registrate le azioni che succedono nel seguente modo:

"Data e ora, un counter e le azioni"

In un altro terminale, inserire il comando "jolie PeerAA.ol": dopo aver pigiato invio parte il programma e nello schermo appare una finestra che chiede di inserire lo username, come segue:

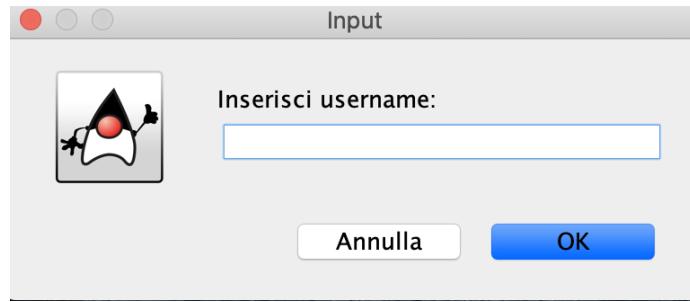


Figure 2: Inserimento username

Una volta inserito lo username, il peer si collega alla rete e può decidere di creare un gruppo o iniziare una chat privata con un altro peer, che deve essere connesso.

Vediamo il caso della **CHAT PRIVATA**.

Aperto un altro terminale, bisogna inserire di nuovo il comando ”jolie PeerAA.ol” per creare un altro peer da collegare alla rete che può iniziare a comunicare con il peer già connesso.

Supponiamo che il primo peer connesso si chiami Alice. Appare una schermata come la seguente:

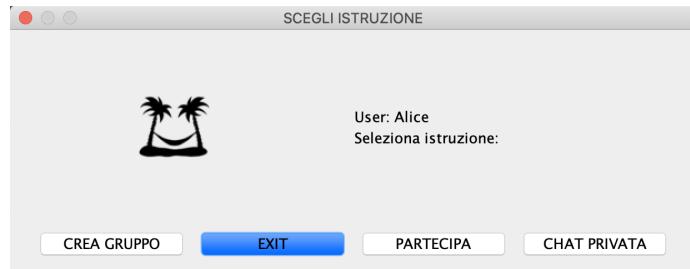


Figure 3: Alice sceglie l'istruzione

Supponiamo che il secondo peer connesso si chiami Bob. Alice decide di iniziare una chat privata con quest'ultimo, quindi nella finestra clicca l'opzione "chat privata" e appare la seguente finestra:

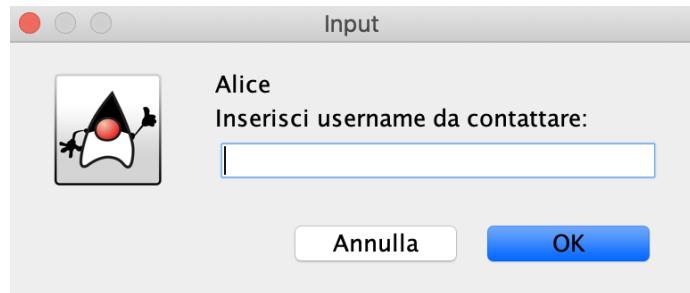


Figure 4: Alice contatta uno user

Una volta inserito lo username da contattare (in questo caso Bob), appare la richiesta di inizio chat a Bob, il quale deve decidere se accettare o meno.

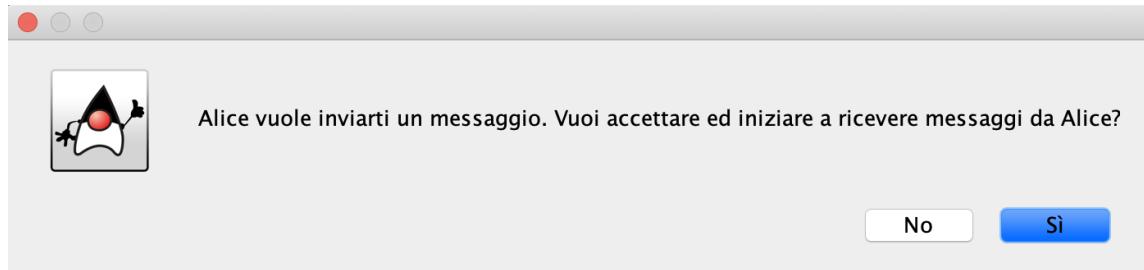


Figure 5: showYesOrNotDialog per l'utente da contattare

Ovviamente questo succede se Bob è connesso alla rete, altrimenti nel terminale verrebbe registrata una riga in cui si dice che l'username ricercato non esiste.

Se Bob accetta la richiesta di Alice, appare una finestra in cui quest'ultima può finalmente mandare dei messaggi al suo partner:

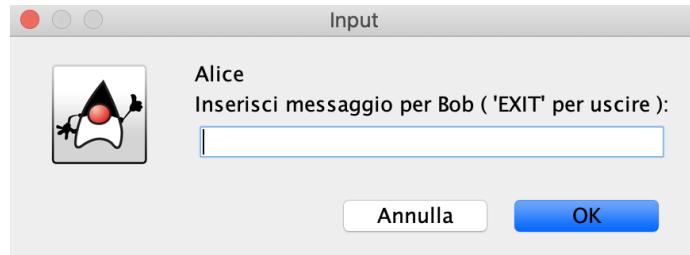


Figure 6: Alice scrive a Bob

Tutto ciò deve essere fatto anche da Bob, altrimenti Alice scriverebbe senza ricevere risposta da Bob, ma solo quest'ultimo potrà visualizzare i messaggi ricevuti da Alice.

Ora vediamo il caso della **CHAT PUBBLICA**.

Apriamo il terminale nella directory desiderata, scriviamo il comando ”jolie PeerAA.ol” e inviamo. Chiamiamo il nostro peer ”Bob” e supponiamo che questo voglia iniziare una chat di gruppo.

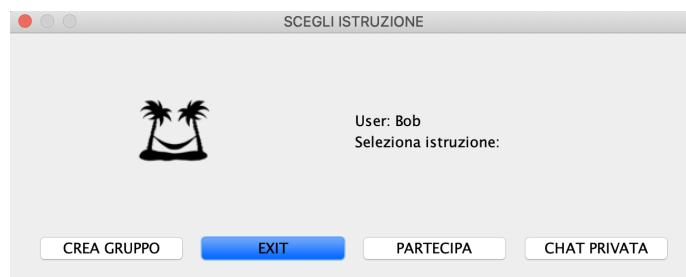


Figure 7: Scelta per Bob

Dopo aver cliccato su tale sezione, Bob deve inserire il nome del gruppo:

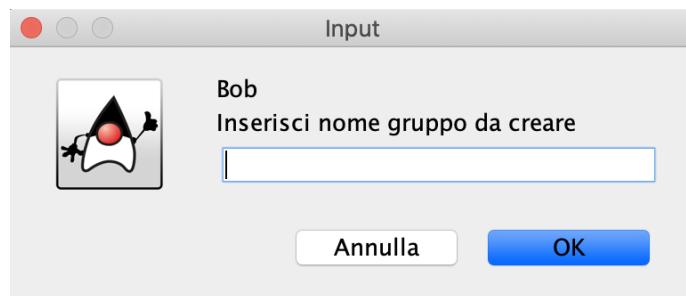


Figure 8: Inserimento nome del gruppo

Una volta inserito il nome del gruppo, Bob può finalmente iniziare a mandare i messaggi. Per creare altri peer che possano accedere alla rete ed entrare nel gruppo, inserire il comando ”jolie peerAA.ol” in un nuovo terminale (bisogna aprirne uno per ogni peer), cliccare su ”PARTECIPA” e inserire il nome del gruppo creato in precedenza: se si sbaglia nome, sul terminale e sulla console di tracciamento appare un messaggio che riferisce che il gruppo ricercato è inesistente.

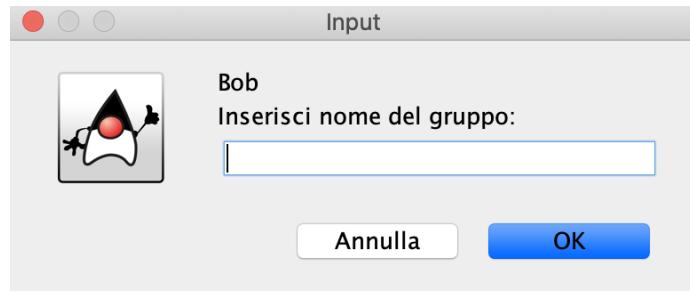


Figure 9: Partecipazione ad un gruppo

Una volta creato il gruppo, il peer può finalmente mandare dei messaggi:

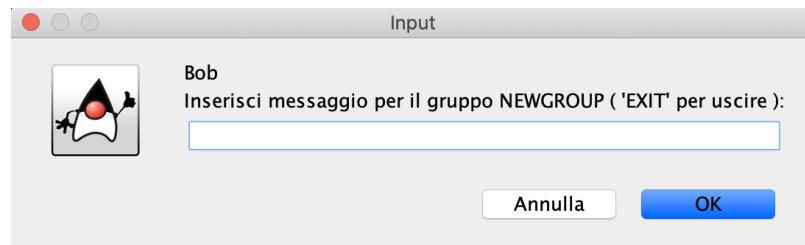


Figure 10: Comunicazione con il gruppo

3 Rete P2P decentralizzata

Dopo una lunga valutazione delle possibili alternative, abbiamo optato per un sistema peer-to-peer totalmente decentralizzato: senza la necessità di un server o di un “super peer” (così come di uno o più servizi di supporto). In questo modo, ogni peer è totalmente indipendente dagli altri e non necessita di alcun supporto per poter rimanere attivo in rete. Questo grande vantaggio ha posto però anche diversi limiti.

Esempio pratico di uno svantaggio: supponiamo che un utente effettui il login con il nome “Mario”. Successivamente, se questo utente andrà offline il nome “Mario” potrà essere utilizzato da un altro utente. Quindi, in una rete come la nostra, non potrà mai esistere una fase di login con username e password, poiché questo richiederebbe un server.

3.1 Connettersi alla rete

Esaminiamo passo per passo come avviene il processo di login di un utente.

Premessa: la nostra rete, per difficoltà logistiche, è stata progettata per funzionare in localhost, sfruttando un sistema di broadcast (spiegato in seguito) che mandi una notifica in uno specifico range di porte (nel codice di seguito riportato abbiamo considerato le porte dalla 10001 alla 10101, tale range può essere modificato a piacimento, ma bisogna tener conto che più ampio è il range di porte a cui inviare una notifica, maggiore sarà il tempo d'attesa; per 100 porte si ottiene un tempo d'attesa molto breve)

PASSO 1: RICERCA PORTA LIBERA PER INPUTPORT DEL SERVIZIO PeerB

Per avviare un peer in rete bisogna digitare da terminale il comando “jolie PeerAA.ol”. Come prima cosa viene ricercata una porta libera tra il range di porte prestabilito attraverso l’outputPort denominata “port”. Il nostro peer si posizionerà nella prima porta libera che trova. Per “posizionarsi” in tale parte, effettua l’embedding dell’eseguibile “PeerBB.ol”, che si occuperà del lato server del nostro peer. In questo modo viene sfruttato il *dynamic embedding* per l’inputPort offerto da Jolie.

Attraverso un ciclo *while*, viene tentato di effettuare un embedding in ogni porta fino a quando non se ne trova una libera. Infatti, se viene effettuato l’embedding su una porta occupata, verrà prodotto un *RuntimeException*, il quale viene gestito

```

//RICERCA PRIMA PORTA LIBERA TRA 10001 E 10101
condition = true
portNum = 10001
while( condition && portNum<10102 ) {
    scope( e ){
        install( RuntimeException  => {
            portNum = portNum + 1
        })
        with( emb ) {
            .filepath = "-C LOCATION=\"\" + "socket://localhost:" + portNum + "\" PeerBB.ol"
            .type = "Jolie"
        };
        loadEmbeddedService@Runtime( emb )()
    }
    num_port = portNum //Assegnazione numero di porta generato
    condition = false
}

```

Figure 11: Ricerca prima porta disponibile

tramite l'incremento del numero di porta per ritentare nella porta successiva. Se tutte le porte nel range sono occupate, verrà stampato un avviso e il programma terminerà la sua esecuzione.

Alla fine di questo passo avremo due servizi attivi: PeerAA, per il lato client, e PeerBB, per il lato server. Questi due servizi insieme costituiscono le due facce del nostro peer.

PASSO 2: BROADCAST E ”DOPPIO SALUTO”

Il passo successivo, in breve, serve per notificare agli altri peer della nostra presenza nella rete e a noi per conoscere gli username dei peer online e le rispettive porte.

- FASE I: BROADCAST

Supponiamo che il nostro peer trovi libera la porta 10010. Come mostrato in figura verrà inviata una notifica tramite il metodo *broadcast* a tutti i processi nel range prestabilito.

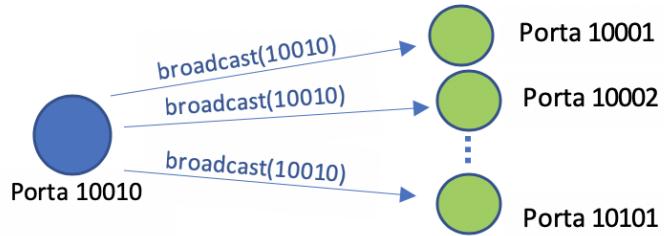


Figure 12: Schema riassuntivo

Sotto viene riportato il codice lato client di quanto descritto sopra. Si noti che viene gestita l’eccezione *ConnectException* (la quale viene racchiusa nell’eccezione *IOException*) dato che, tale errore, viene sollevato quando viene mandato un messaggio di broadcast in una porta vuota o in una porta occupata da un processo diverso da un peer.

```
define broadcastMsg {
    for( i = 1001, i < 10101, i++ ) {
        scope( e ) {
            install( IOException => i = i )
            if( i != user.port ) {
                port.location = "socket://localhost:" + i
                broadcast@port( user.port )
            }
        }
    }
}
```

Figure 13: Codice broadcastMsg

Nell’immagine sotto si osservi l’implementazione del broadcast lato server.

```
//BROADCAST
[broadcast( newuser.port )] {
    out.location = "socket://localhost:" + newuser.port
    hello@out( global.user )
}
```

Figure 14: broadcast lato server

Riassumendo: viene inviato a tutte le porte in cui potrebbe esserci un peer il numero di porta del nostro peer che si sta connettendo alla rete. I peer che ricevono tale notifica, rispondono con il metodo *hello* inviato alla porta ricevuta con il broadcast.

Il metodo *broadcast* è un metodo *OneWay* e non potrebbe essere altrimenti: se fosse una *RequestResponse* il servizio PeerAA rimarrebbe bloccato alla prima porta vuota o in cui è presente un processo diverso da un peer.

- **FASE II: HELLO**

Come visionato nel paragrafo precedente, i peer che ricevono la notifica di broadcast rispondono con il servizio *hello* di cui viene riportato il codice di seguito.

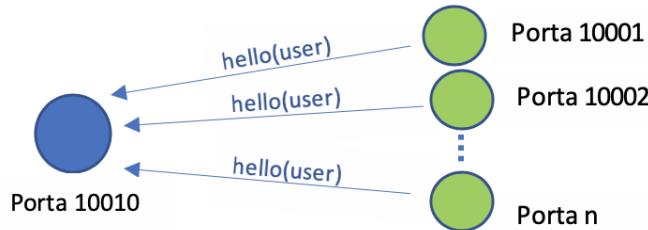


Figure 15: Schema riassuntivo

Anche il servizio *hello* è di tipo *OneWay* (per le stesse ragioni del broadcast) e richiede una variabile di tipo *user* come riportato.



Figure 16: Servizio

Quindi, tramite l'*hello*, i peer che hanno ricevuto la notifica di broadcast mandano al peer che ha inviato tale notifica un “saluto” in cui si presentano, ovvero

inviano il proprio username e la propria porta. A questo punto, il nostro peer sarà a conoscenza di tutti i peer presenti momentaneamente nella rete. Come si può vedere nel codice cerchiato del servizio *hello*, il peer registra le informazioni dei peer da cui riceve questo “saluto” in due array: uno per i nomi e l’altro per le porte.

Se vogliamo conoscere la porta di un peer dal suo username, ci basterà cercare tale username nell’array *peer_names*, salvarci la posizione in cui risiede, e ottenere il valore della porta nell’array *peer_port* nella medesima posizione. Quanto appena descritto è il servizio *searchPeer* che verrà utilizzato proprio per questo scopo.

Ora, il nostro peer conosce gli altri peer in rete, ma questi non conoscono ancora le sue informazioni.

- **FASE III: LOGIN**

La fase successiva è quella di *login*: l’utente dovrà inserire un username che non sia già stato scelto da un altro utente online. Il servizio *login* è di tipo *RequestResponse* poiché l’esecuzione potrà proseguire solo una volta inserito un username valido.

```
//Iscrizione nella rete
port.location = "socket://localhost:" + user.port
login@port(user.port)(user.name)
```

Figure 17: Fase di login

Tramite queste due righe di codice, il nostro PeerAA comunica con il nostro PeerBB, ovvero la parte client del nostro peer comunica con la parte server del medesimo. Questo passaggio, ovvero la comunicazione tra client e server dello stesso peer, è fondamentale nell’architettura della nostra rete e verrà utilizzato diverse volte. Nel caso del metodo *login*, PeerAA comunica a PeerBB il numero di porta occupato e attende invece un username valido che il PeerBB richiederà all’utente, come si può notare nel codice sottostante.

```

login(user.port)(response) [
    //ciclo while che si interrompe solo quando viene inserito un username valido
    condition = true
    while ( condition ) {

        install( TypeMismatch => { ...
            })

        synchronized( lock ) {
            showInputDialog@SwingUI( "Inserisci username: " )( responseUser )
            isOriginal = true
            for ( i = 0, i < #global.peer_names, i++ ) {

                //UpperCase per la verifica degli user
                toUpperCase@StringUtils( string(global.peer_names[i]) )( responsePeer )
                toUpperCase@StringUtils( responseUser )( responseUserUppercase )

                //Acquisisco lunghezza stringa per controllo aggiuntivo
                length@StringUtils( responseUser )( lengthUserWord )

                if( (responsePeer == responseUserUppercase) || (lengthUserWord < 2) ) {
                    isOriginal = false
                }
            }
        }

        if ( isOriginal ) {
            condition = false
            //Richiamo define per sistemare i caratteri
            settaggioCaratteri

            //Genero username completo combinando i caratteri restituiti dalla define
            response = responseUp + responseLower
        } else {
            showMessageDialog@SwingUI("Username già utilizzato o nome troppo corto")()
        }
    }
}

```

Figure 18: Login lato server

Come si può osservare dal codice, un username è valido se è originale e se è più lungo di un carattere. Quindi, se *isOriginal* rimane *true* dopo il ciclo *for*, allora *condition* verrà posto a *false* e si potrà uscire dal ciclo *while*. Altrimenti, verrà visualizzato il messaggio indicato dall'ultima freccia nello screenshot precedente e il ciclo *while* ricomincerà dall'inizio.

Infine, il metodo *login* manda a ogni porta presente nell'array *peer_port* un “saluto” per farsi conoscere dagli altri peer attraverso il servizio *sendHi*.

```

for( i=0, i < #global.peer_port, i++ ) {
    if ( global.peer_port[0] != 0 ) { //controllo che sia stata settata almeno una porta
        out.location = "socket://localhost:" + global.peer_port[i]
        sendHi@out( global.user )
    }
}

```

Figure 19: sendHi lato client

- FASE IV: HI

Tramite il metodo *sendHi*, similmente al metodo *hello*, il peer novizio della rete si “presenta” agli altri peer della rete, ovvero fornisce loro il proprio username e la propria porta.

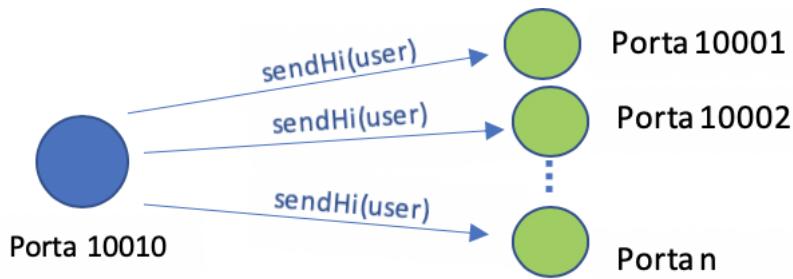


Figure 20: Schema di esempio

Ugualmente al servizio *hello*, anche il metodo *sendHi* è di tipo *OneWay* e richiede una variabile di tipo user. Infatti, avviene più o meno lo stesso procedimento del codice cerchiato nell’immagine relativa all’*hello*.

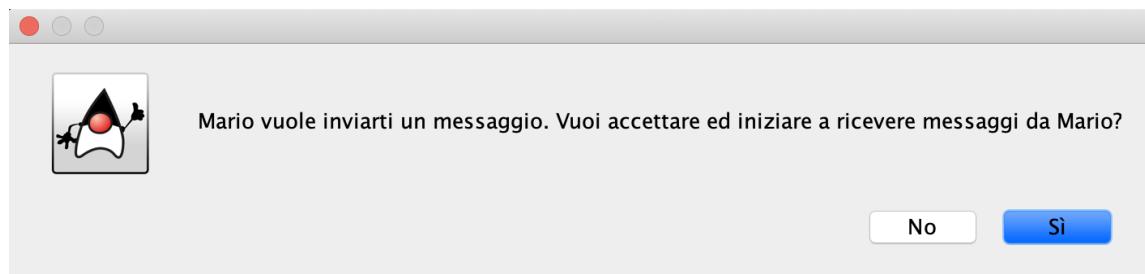
A questo punto, il nostro peer conosce tutti i peer presenti in rete e questi conoscono le informazioni relative al nostro peer. I passi successivi sono la creazione di un file per fornire la persistenza delle conversazioni e la generazione delle chiavi pubbliche e private per la crittografia. Questi due passi verranno descritti in maniera approfondita più avanti.

Dopodiché il nostro peer sarà pronto per iniziare una chat privata, avviare una chat di gruppo oppure partecipare ad un gruppo già esistente.

3.2 Avvio di una chat privata

Una volta effettuato il login, l'utente potrà scegliere tra diverse opzioni tra cui l'avvio di una chat privata. Quando un utente desidera parlare con un altro utente online deve premere il pulsante “CHAT PRIVATA” e in seguito inserire l'username del destinatario. Così facendo, tale destinatario riceverà una richiesta per accettare o rifiutare di ricevere messaggi dal peer che ha provato a contattarlo.

Facciamo un esempio. Supponiamo che un peer di nome Mario provi ad avviare una chat privata con un peer di nome Luigi. L'utente con username Luigi riceverà il seguente messaggio.



Se viene premuto il pulsante “No”, Mario riceverà una notifica in cui viene informato che Luigi ha rifiutato la chat. Se viene premuto il pulsante “Sì”, Mario potrà mandare dei messaggi a Luigi.

Se Luigi vuole rispondere a Mario deve mandare a sua volta una richiesta di chat privata a Mario.

Analizziamo ora il codice nello specifico.

```
showInputDialog@SwingUI( user.name + "\nInserisci username da contattare: " )( dest )

//Restituisce il numero di porta da contattare del destinatario ( 0 se inesistente )
searchPeer@port( dest )( dest_port )

if ( dest_port == 0 ) {
    println@Console( "L'username ricercato non esiste." )(  )
} else {
    startChat
}
```

Figure 21: Codice per avviare chat privata

Quando viene inserito uno username da contattare, per prima cosa si verifica se l'username ricercato è di un peer della rete (ovvero se appartiene a un peer con il quale si è scambiato il “doppio saluto”). Se l'username esiste, allora il servizio *searchPeer* (descritto in precedenza) restituisce il numero di porta associato a tale username, altrimenti restituisce 0.

Il primo passo è mandare una richiesta al destinatario per iniziare a mandargli dei messaggi.

```
//Gestione errore se l'utente abbandona la rete
install( IOException => println@Console( "L'utente è andato offline." )() )

msg.username = user.name
port.location = "socket://localhost:" + dest_port

//Invio richiesta di chat al destinatario
chatRequest@port( user.name )( enter )
```

Figure 22: Codice di appoggio

Per prima viene gestita l'eccezione nel caso in cui l'utente contattato vada offline. Subito dopo, viene utilizzato il servizio *chatRequest*: un servizio di tipo *RequestResponse* che richiede una variabile di tipo string, ovvero il nostro username, e restituisce una variabile di tipo *bool*. Tale variabile sarà *true* se il peer contattato avrà accettato la richiesta, altrimenti sarà *false*. Successivamente con il metodo *richies-taChiavi* il peer si appropria della chiave pubblica del destinatario, in modo da poter criptare i messaggi che gli manda tramite l'algoritmo RSA.

A questo punto si entra in un ciclo *while*, il quale a ogni iterazione richiederà di inserire un messaggio da mandare. Se tale messaggio è “EXIT”, allora il ciclo *while* terminerà e l'utente tornerà al menù principale. Altrimenti, viene eseguita la codifica del messaggio e il messaggio criptato verrà spedito con il servizio *sendString*. Tale servizio, in sintesi, decodifica il messaggio e lo stampa nella console (oltre a scriverlo su file per mantenerne traccia; l'argomento della persistenza verrà trattato più avanti).

Sotto viene riportato uno screenshot di un esempio di chat privata.

```

Per rispondere a Mario avvia una chat con lui/lei.
23/07/2020 10:59:24      Mario : Ciao Luigi
23/07/2020 10:59:28      Mario : Tutto bene?
23/07/2020 10:59:32      Luigi: Ciao Mario
23/07/2020 10:59:43      Luigi: Io sto bene grazie
23/07/2020 10:59:47      Luigi: Te come stai?
23/07/2020 11:00:25      Mario : Mi sento in forza!

```

3.3 Chat di gruppo

Nella nostra rete un gruppo non è altro che un peer a cui ogni partecipante del gruppo manda il messaggio che desidera mandare al gruppo. Sarà questo peer a inoltrare il messaggio a tutti gli altri partecipanti del gruppo.

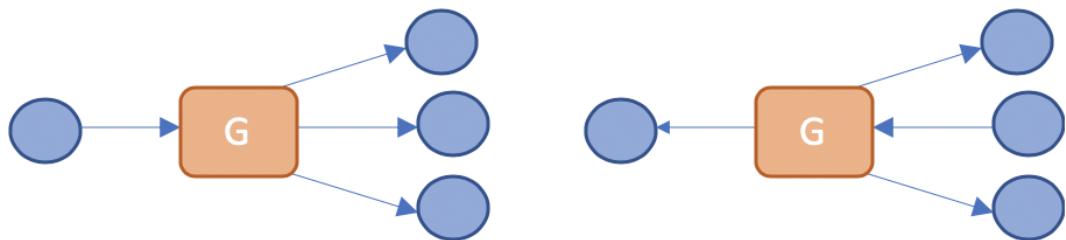


Figure 23: Comunicazione in un gruppo

Nell’immagine sopra si può osservare un esempio di comunicazione in un gruppo. Il PeerGroup (in arancione) riceve un messaggio alla volta dai peer partecipanti e lo rimanda agli altri partecipanti.

Analizziamo ora il procedimento dal punto di vista pratico. Quando un peer crea un gruppo effettua l’embedding del servizio “PeerGroup.ol”. In altre parole, sta creando un altro peer nella rete. Ne consegue che questo nuovo peer (che chiameremo PeerGroup) è dipendente dal peer che lo ha generato: se tale peer (che chiameremo host) andrà offline, anche il PeerGroup andrà offline. I passi per la creazione di un PeerGroup sono pressoché identici a quelli della creazione di un peer normale: la ricerca di una porta libera nel range prestabilito, l’inserimento di un nome valido per il gruppo e l’invio del messaggio di broadcast.

Una volta settata, ancora grazie al embedding dinamico di Jolie, la location dell’inputPort

del PeerGroup e inserito un nome valido del gruppo, si possono inizializzare le variabili con il servizio *setGroup*.

```
setGroup( request )() {
    global.group.name = request.name
    global.group.port = request.port
    global.members[ 0 ] = request.host
}
port.location = "socket://localhost:" + group.port
group.host = user.port
setGroup@port( group )()
```

Figure 24: Servizio setGroup

Le variabili identificative del gruppo sono: il nome, la porta e i membri. Le porte dei membri del gruppo sono memorizzate nell’array *members*, di cui la prima porta è quella dell’host. Infine, si manda la notifica di broadcast esattamente come con un peer normale. Come già visto, i peer della rete al broadcast rispondono con un “saluto” tramite il metodo *hello*, che nel servizio PeerGroup è implementato in maniera semplificata.

```
//metodo per rispondere a un saluto
[hello( peer )] {
    out.location = "socket://localhost:" + peer.port
    sendHi@out( global.group )
}
```

Figure 25: Hello lato server

Il PeerGroup risponderà a sua volta “salutando” il peer da cui ha ricevuto l’*hello* tramite il metodo *sendHi* (è lo stesso visto in precedenza poiché viene ricevuto dal servizio PeerB.ol). Notare che in PeerGroup.ol non è implementato il metodo *sendHi*. Questa scelta è dovuta dal fatto che al PeerGroup non interessa sapere quali peer sono presenti in rete, ma solo quali peer fanno parte del gruppo.

Non appena il gruppo viene creato correttamente, il suo host potrà già inviare messaggi nel gruppo.

Il sistema di messaggistica nel gruppo funziona nello stesso modo della chat privata, se non per due variazioni importanti: la crittografia è diversa (tale diversità

verrà approfondita in seguito) e al posto del servizio *sendString* viene usato il servizio *sendMessage* (implementato in PeerGroup.ol).

```
port.location = "socket://localhost:" + group.port
sendMessage@port( msg )
```

Il servizio *sendMessage* è di tipo *OneWay* e si occupa di inoltrare il messaggio a tutti i membri del gruppo. (Nel codice si può notare che vengono ignorate le porte poste a -1, ovvero le porte che in precedenza erano occupate da un membro del gruppo che in seguito ha abbandonato la chat).

Per inoltrare i messaggi viene utilizzato il metodo *forwardMessage* implementato in PeerB. Tale metodo verifica l'integrità del messaggio ricevuto (rimandiamo anche in questo caso la spiegazione alla sezione sulla crittografia), lo scrive su file e lo stampa a video.

Se viene inserito il messaggio “EXIT”, come nella chat privata, si uscirà e verrà chiamato il metodo *exitGroup*, il quale avverte il PeerGroup che sta uscendo dal gruppo e che, quindi, non dovrà più ricevere messaggi da esso.

Tale metodo setta nell'array *members* la porta del peer che desidera uscire a -1, in modo che venga ignorata dal metodo *forwardMessage*.

3.4 Partecipare ad una chat di gruppo

Per partecipare a una chat di gruppo bisogna premere il pulsante “PARTECIPA” e successivamente inserire il nome esatto del gruppo.

```
//metodo per ricevere messaggi dai peer del gruppo e spedirli a tutti gli altri membri
[sendMessage(msg)] {
    for ( i=0, i < #global.members, i++ ) {
        if( global.members[i] != -1 ) {
            out.location = "socket://localhost:" + global.members[i]
            forwardMessage@out(msg)
        }
    }
}
```

```

if ( responseMessage == "EXIT" ) {
    exitGroup@port( user )()
}

showInputDialog@SwingUI( user.name + "\nInserisci nome del gruppo: " )( responseContact )
group.name = responseContact

//Ricerca porta di gruppo per la comunicazione pubblica .
searchPeer@port( group.name )( group.port )

if ( group.port == 0 ) {
    println@Console( "Il gruppo ricercato non esiste." )()
    scope( exceptionConsole ) {
        install( IOException => println@Console("Errore, console non disponibile!")() )
        press@portaStampaConsole( user.name + " ha ricercato un gruppo " + group.name + " inesistente!" )()
    }
} else {
    port.location = "socket://localhost:" + group.port
    enterGroup@port( user )()
    println@Console( "\nBenvenuto nel gruppo " + group.name + "!\n" )()

    //Inizio la comunicazione con il gruppo .
    startGroupChat
}

```

Figure 26: Servizi sendMessage, enterGroup ed exitGroup

Come si può notare nel codice soprastante, viene sfruttato anche in questo caso il servizio *searchPort* per sapere se il gruppo che stiamo cercando è online. In questo modo otterremo la porta del PeerGroup e potremo utilizzare il metodo *enterGroup*. Tale metodo aggiorna l'array *members* in modo che il metodo *forwardMessage* inoltri i futuri messaggi anche al nuovo partecipante.

```

//metodo per aggiungere nuovi peer al gruppo
[
    enterGroup(peer)() {
        global.members[ #global.members ] = peer.port
    }
]

```

Figure 27: Servizio enterGroup lato server

Infine, anche il nuovo partecipante può cominciare a inviare messaggi al PeerGroup come descritto in precedenza.

3.5 Gestione delle eccezioni, problemi e concorrenza in PeerB e PeerGroup

3.5.1 Gestione login contemporaneo di più peer

Premessa: come descritto in precedenza, nella fase di login il primo passo che viene compiuto è l'assegnazione di una porta libera. In quel momento il peer occupa già una porta, ma non gli è stato ancora assegnato un username. Per default in PeerB, troviamo la variabile global.user.name inizializzata a "undefined".

Se più peer si trovano contemporaneamente in questa situazione potrebbero esserci diversi problemi. Per ovviare a essi sono stati aggiunti i metodi *setPort* e *sendUsername*: il primo permette di settare la variabile *global.user.port* prima che venga effettuato il login, mentre il secondo permette di inviare a un peer le proprie informazioni (porta e username) in modo che questo peer le salvi. Quest'ultimo servizio viene utilizzato in PeerA dopo il metodo login all'interno di un ciclo *while* che verifica la presenza di altri peer con nome "undefined" per "farsi conoscere" da questi tramite esso.

3.5.2 Gestione errore monitor

Nella nostra rete abbiamo utilizzato un'implementazione di Swing (spiegata in seguito), attraverso la quale sfruttando un javaservice riusciamo ad avere un monitor sfruttando delle strutture presenti in Java e, allo stesso tempo, riusciamo ad avere una console dialog tutta nostra per le scelte che gli utenti in rete andranno ad effettuare.

Per quanto riguarda il monitor, inizialmente abbiamo riscontrato dei problemi derivanti dal suo utilizzo: per avviare la rete eravamo costretti ad attivare necessariamente il monitor, il quale registrava tutti i log di rete. Per porre rimedio a tutto ciò abbiamo pensato di creare un peer che di fatto gestiva l'apertura di default di tale console nel momento in cui il primo utente effettuava il login alla rete. Il problema in tutto questo sarebbe stato il seguente: se quello specifico peer avesse abbandonato la comunicazione volontariamente o involontariamente, avrebbe chiuso la console, generando nuovamente un errore che non permetteva all'intero sistema di proseguire nella propria esecuzione.

Esempio: Alice e Bob vogliono comunicare con amici attraverso un gruppo creato da Alice. Dopo aver attivato il monitor e dopo aver effettuato la registrazione

di entrambi alla rete, Alice decide di creare il gruppo online di amici per organizzare un’uscita serale, senza dover scrivere necessariamente a tutti gli invitati. Dopo la creazione del gruppo, Bob cerca di partecipare al gruppo, ma nel momento in cui invia la propria richiesta, accidentalmente il monitor che permette la visualizzazione dei log si arresta non permettendo ad Alice di organizzare la serata. Quale soluzione potremmo adottare per soddisfare la richiesta di Alice?

La soluzione ottimale è stata quella di gestire le eccezioni che venivano a crearsi. In Jolie è presente una funzionalità che permette di catturare degli errori come in Java(try-catch); tale istruzione è denominata **scope()** accompagnata dall’istruzione **install()**. Di seguito il codice utilizzato:

```
scope( exceptionConsole ) {
    install( IOException => println@Console("Errore, console non disponibile!")()
    press@portaStampaConsole( user.name + " si è unito/a alla rete! " + "(" + num_port + ")" )()
}
```

Figure 28: Gestione eccezione monitor

Come si nota dalla figura, l’istruzione **install** gestisce l’errore per avvisare l’utente, omettendo il codice sottostante che precedentemente permetteva l’invio di stringhe al monitor attraverso una `outputPort`.

Replicando, inoltre, tale istruzione su tutti i possibili invii di messaggi al monitor, siamo riusciti a gestire il corretto funzionamento dell’intero sistema senza l’appoggio di una struttura che permettesse di effettuare un controllo.

3.5.3 Gestione riconoscimento di un utente o di un gruppo

Nel nostro sistema inizialmente non era possibile identificare chi era in realtà online: se si trattava di un gruppo o semplicemente di un user.

Per porre rimedio a tutto ciò, abbiamo ideato ed implementato due *define* che ci permetessero di gestire tale problema: **settaggioCaratteri** e **riconoscimentoUser-Group**:

```

define settaggioCaratteri {
    //Salvo la stringa contenente lo username .
    stringa = responseUser

    //Ricavo la lunghezza massima .
    length@StringUtils( responseUser )( length )

    with( stringa ) {
        .begin = 0
        .end = 1
    }

    //Setto la prima lettera Up .
    substring@StringUtils( stringa )( responseFirst )
    toUpperCase@StringUtils( responseFirst )( responseUp )

    with( stringa ) {
        .begin = 1
        .end = length
    }

    //Setto tutte le altre lettere a Lower .
    substring@StringUtils( stringa )( responseSecond )
    toLowerCase@StringUtils( responseSecond )( responseLower )
}

```

(a) settaggioCaratteri

```

define riconoscimentoUserGroup {
    flag = false

    stringaGroupUser = peer.name
    length@StringUtils( stringaGroupUser )( lengthGroupUser )

    with( stringaGroupUser ) {
        .begin = lengthGroupUser - 1
        .end = lengthGroupUser
    }

    substring@StringUtils( stringaGroupUser )( responseLast )
    toUpperCase@StringUtils( responseLast )( responseUpLast )

    if( responseLast == responseUpLast ) {
        flag = true
    }
}

```

(b) riconoscimentoUserGroup

Figure 29: Gestione users-group

Come si nota dalla prima figura a sinistra, come prima cosa ci salviamo in una variabile d'appoggio il nome dell'utente e successivamente trasformiamo tale stringa nel modo che segue: il primo carattere maiuscolo, mentre tutti gli altri in minuscolo.

Nel secondo *define* sulla destra cerchiamo di verificare se l'ultimo carattere è maiuscolo, se si allora si tratta di un gruppo altrimenti di un utente. Ovviamente, viene effettuato allo stesso modo il settaggio dei rispettivi caratteri del gruppo, settandoli tutti in maiuscolo.

In tutto questo, però, sorgeva un problema, diamone una semplice spiegazione fornendoci di un esempio:

Esempio: Entra l'utente **A** e successivamente l'utente **B**, infine entra **Bob** in rete, il quale vede stampato su console *"il gruppo A è online"* e *"il gruppo B è online"*. Ovviamente questo comporta un grave problema dato che A e B non sono gruppi, ma utenti.

Per porre rimedio a tale soluzione, abbiamo inserito un controllo sul numero di lettere per lo username. Di seguito il codice:

```

showInputDialog@SwingUI( "Inserisci username: " )( responseUser )
isOriginal = true

//Acquisisco lunghezza stringa per controllo aggiuntivo
length@StringUtils( responseUser )( lengthUserWord )

//Controllo per verificare se uno username ha inserito un nome troppo corto .
if( lengthUserWord < 2 ){
    isOriginal = false
}

```

Figure 30: Controllo username

In questo caso, non è più possibile creare uno username di lunghezza fissa uguale ad uno. Ovviamente, il problema a questo punto non sorge per il gruppo: se il nome sarà composto da una sola lettera, verrà riconosciuto come gruppo, mentre lo user avrà sempre almeno due lettere, quindi sfruttando i define descritti precedentemente, non sarà più possibile confonderli. Di seguito un esempio sul controllo dello username:

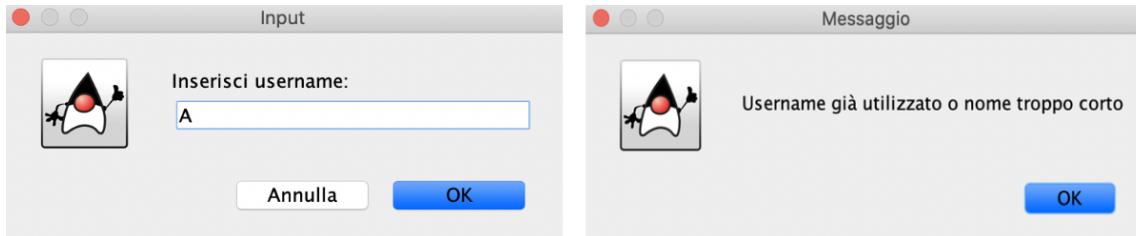


Figure 31: Gestione errore su programma

3.5.4 Concorrenza tra peer

Nella parte riguardante la concorrenza, avevamo pensato all'utilizzo del costrutto semaforo presente in Jolie (semaphore utils), ma abbiamo preferito utilizzare il costrutto di sincronizzazione (synchronized). Tale scelta è stata fatta perché essendo Jolie basato su Java, attraverso la documentazione([link](#)) ci è stato molto utile comprendere il suo funzionamento (molto simile al synchronized presente in Java)

L'altra motivazione che ci ha spinto al non utilizzare i semafori è stata la seguente: nel momento in cui viene creato un nuovo semaforo, necessariamente prima del suo utilizzo bisogna rilasciarlo. Ecco il codice:

```
define p1 {
    acquire@SemaphoreUtils( semaphoreOne )()
    |   |   x = x + 10
    release@SemaphoreUtils( semaphoreOne )()
}

define p2 {
    acquire@SemaphoreUtils( semaphoreOne )()
    |   |   x = x - 10
    release@SemaphoreUtils( semaphoreOne )()
}

main {
    semaphoreOne.name = "semaphoreOne"
    semaphoreOne.permits = 1

    release@SemaphoreUtils( semaphoreOne )();

    for( i = 0, i < 10, i++ ) {
        x = 100;
        { p1 | p2 };
        print@Console( x + " " )();
    }
    println@Console()
}
```

Figure 32: Semafori in Jolie

Come si nota dal codice, vengono creati due processi $P1$ e $P2$, i quali nel *main*,

vengono fatti partire in parallelo, ma come si nota prima della loro esecuzione il semaforo creato, deve essere necessariamente rilasciato.

Concentriamoci ora su PeerB e PeerGroup, dove abbiamo inserito i costrutti di sincronizzazione. Il primo servizio messo a disposizione è *sendUsername*, attraverso il quale (come descritto in precedenza), viene gestito il problema di uno username settato a valore "*undefined*". Cosa potrebbe accadere se due peer contemporaneamente accedessero a tale servizio? Diamone una spiegazione attraverso un esempio.

Esempio: supponiamo di avere due utenti che effettuano l'accesso alla rete senza inserire immediatamente il loro username. Il primo utente effettua una richiesta attraverso login così come il secondo. Successivamente, entrambi inviano i loro username ad un peer che ha ancora settato il proprio ad "*undefined*", il primo utente manda il proprio username e il proprio numero di porta, ma avviene un context switch prima dell'incremento della variabile contatore. Così facendo entra il secondo peer, il quale manda la propria richiesta sovrascrivendo lo username dell'utente precedente e il suo numero di porta. Infine, effettua un incremento così come il primo peer una volta sboccato (**problema di un doppio incremento**).

Per risolvere tale problema è stato necessario inserire un synchronized come segue:

```
[sendUsername(peer)] {
    synchronized( lockUsername ) {
        global.peer_names[ global.count ] = peer.name
        global.peer_port[ global.count ] = peer.port
        global.count = global.count + 1
    }
}
```

Figure 33: sendUsername

Tale soluzione è stata adottata anche in altri servizi come *broadcast*, *hello* e *sendHi* già spiegati nei paragrafi precedenti. Anche in quei casi potevano esserci problemi di concorrenza.

Esempio: ci sono 10 utenti online, Alice decide di aggiungersi alla rete e successivamente utilizzando *broadcast* invia a tutti le proprie informazioni (username e numero di porta). Supponiamo che nello stesso momento in parallelo, Bob mandi le proprie informazioni. Come nella situazione precedente in caso di context switch accidentali, potrebbero esserci problemi di sovrascrizione e doppi incrementi. La stessa situazione accade quando ogni singolo peer presente in rete manda una risposta ad Alice o a Bob.

Di seguito il codice (*hello*) adottato per risolvere tale problematica:

```
[hello( peer )] {

    //RICONOSCIMENTO GRUPPO O USER
    riconoscimentoUserGroup

    if( flag ) {
        println@Console( "Il gruppo " + peer.name + " è online!" )()
    } else {
        println@Console( peer.name + " è online." )()
    }

    //AGGIUNTA SYNCHRONIZED PER RISOLVERE PROBLEMI DI SOVRASCITTURA E DOPPI INCREMENTI
    synchronized( lockHello ) {
        global.peer_names[ global.count ] = peer.name
        global.peer_port[ global.count ] = peer.port
        global.count = global.count + 1
    }
}
```

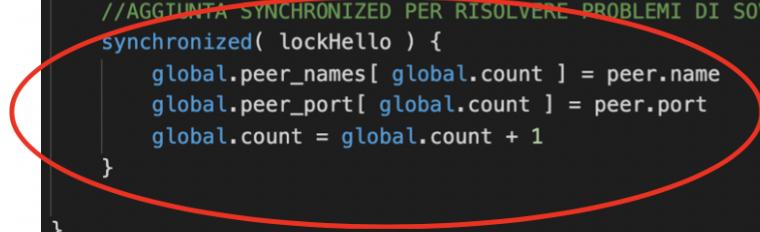


Figure 34: Servizio hello

Ci siamo anche accorti che nel nostro sistema l'utilizzo di synchronized utilizzando un nome uguale per tutti: ad esempio il nome *lock*, poteva comportare dei problemi di stallo. Molte volte la console si bloccava e questo dipendeva dal fatto che un processo precedente, magari, stava sfruttando un servizio. Facciamo un esempio per comprendere meglio:

Esempio: Alice e Bob vogliono comunicare con Mario. Alice entra in rete ed effettua la richiesta di comunicazione con Mario, entra in un metodo synchronized e successivamente avviene un context switch che la fa rimanere bloccata. A questo punto Bob cerca di entrare in rete, ma nel momento in cui entra rimane bloccato perché lo stesso lock che cerca di acquisire viene utilizzato da Alice. Ovviamente,

come avviene in Java, in caso di context switch il lock viene rilasciato attraverso una chiamata *notify* o *notifyAll*, ma in questo caso abbiamo dovuto risolvere in altra maniera; assegnando ad ogni servizio synchronized un nome di lock differente, questo perchè l'accesso al servizio per ogni peer deve essere univoco ed esclusivo.

Altro problema che abbiamo riscontrato è stato il seguente: avviando due console, a volta ci veniva restituito un errore se uno dei due utenti iniziava la comunicazione con l'altro.

Esempio: Alice si registra alla rete e successivamente Bob fa lo stesso, ma nel momento in cui Bob effettua il login, vengono generate nuove chiavi di Alice, la quale possedeva già le proprie. All'inizio pensavamo che ci fosse un problema di concorrenza, questo perchè Bob entrando nel servizio per la generazione delle chiavi andava a sovrascrivere quelle di Alice.

Tale problema, fortunatamente, è stato diagnosticato e risolto nel modo che segue: dopo l'eventuale ricerca di nuovi peer entrati in rete, ma con il proprio username non ancora definito, il numero di porta non veniva settato al peer (o user) corrente, così da generare nuove chiavi per peer già presenti in rete. Quindi, ci è bastato settare il numero di porta al numero di porta esatto del peer.

3.5.5 Concorrenza in PeerGroup

Anche in questo caso, ci siamo accorti come nella chat privata, che potevano sussistere problemi di concorrenza. Anche in questo caso problemi simili in *broadcast*, *hello* e *sendHi* sussistevano. Il metodo di risoluzione adottato è stato sempre lo stesso descritto nel paragrafo precedente.

Altro problema si creava all'interno di due servizi *enterGroup* e *exitGroup*. Diamone una spiegazione avvalendoci di un esempio:

Esempio: Alice vuole unirsi al gruppo *"Viaggio tra amici"* ed effettua la chiamata al servizio per entrare nel gruppo per chat pubblica, allo stesso modo Bob effettua la richiesta. Supponiamo che prima dell'incremento del numero di porta di Alice e quindi del numero di utenti presenti nella chat pubblica (ovviamente per avvisare tutti della sua presenza) avvenga un context switch. Se dovesse arrivare a questo punto Bob sovrascriverebbe il nome di Alice e la sua porta, andando incontro anche

ad un doppio incremento.

Anche in questo caso per risolvere il problema abbiamo utilizzato *synchronized*. Inoltre, come descritto nei paragrafi precedenti, per permettere l'accesso singolo e designare così delle sezioni critiche per i peer, abbiamo usato altri *synchronized* su altri servizi, come ad esempio la scrittura su file(spiegata più avanti).

4 Pubblicazione e scrittura dei messaggi

4.1 Persistenza dei messaggi e problemi di concorrenza

Il nostro sistema di messaggistica istantanea tra nodi di una rete prevede la persistenza dei messaggi inviati tra i vari peer in file creati per ognuno di loro: ogni utente che entra nella rete e partecipa ad una chat, sia pubblica che privata, ha a sua disposizione un file di testo in formato .txt dove vengono salvati i messaggi che manda.

Nel nostro caso, vediamo il messaggio come una richiesta di scrittura su una risorsa condivisa da molti processi. Questo richiama il problema di gestione della concorrenza che si verifica in presenza di più peer che vogliono scrivere sullo stesso file in un determinato momento. Questo potrebbe creare interferenze nel momento di scrittura, perché potrebbe essere possibile che due peer scrivano sulla stessa riga e l'integrità del messaggio può essere compromessa.

Il problema della concorrenza nasce dal fatto che le istruzioni sono scritte in linguaggio Assembly, che è di basso livello e molto simile al linguaggio macchina.

Con questo, anche una semplice istruzione di incremento può essere decomposta in più istruzioni. L'esecuzione quindi non sempre può andare a buon fine perché in ogni momento possono avvenire dei context switch che interrompono l'esecuzione di un processo per farne partire un altro e, quando avviene ciò, possono essere salvati certi valori nei registri del sistema e poi recuperati altri di processi diversi.

Per risolvere questo problema di concorrenza, si possono utilizzare dei costrutti che vengono offerti dal sistema, ovvero i semafori, oppure si usa synchronized.

4.2 Creazione file e scrittura messaggi

La creazione dei file e la scrittura dei messaggi avviene grazie ad operazioni definite nelle interfacce “exec.iol” e “file.iol”, incluse nel deployment. Abbiamo deciso di usare l'interfaccia exec.iol perchè file.iol non offre servizi che permettano di creare file.

```
include "exec.iol"
include "file.iol"
```

Le operazioni che usiamo sono definite come delle request-response: in questo tipo di comunicazione, si manda un messaggio e si aspetta per una risposta, si cattura il

pattern dell’interazione request-response. Infatti nel nostro caso riceve in input una stringa (ovvero il nome del file che è personalizzato per ogni peer) e ritorna void.

exec@Exec()()

La creazione del file viene fatta all’interno di uno scope perché si vuole gestire un errore che può essere generato dalla non presenza del file su cui scrivere.

Distinguiamo i due tipi di chat che possono presentarsi:

CHAT PRIVATA

```
//CHAT PRIVATA
define startChat {
    //START CHATTING
    scope( e ) {

        //Gestione errore se l'utente abbandona la rete
        install( IOException => println@Console( "L'utente è andato offline." ) )

        msg.username = user.name
        port.location = "socket://localhost:" + dest_port

        //Invio richiesta di chat al destinatario
        chatRequest@port( user.name )( enter )

        if ( enter ) {

            with( richiesta ) {
                .filename = "BackupChat/DATABASE_" + user.name + ".txt"
                .content = "\nINIZIO A MANDARE MESSAGGI A " + dest + "\n"
                .append = 1
            }
            writeFile@File( richiesta )()
        }
    }
}
```



La prima cosa che avviene, per quanto riguarda il peer A che si è connesso alla rete, è l’invio della richiesta di chat al peer B: una volta che lui ha accettato, si scrive nel file del peer client la stringa ”INIZIO A MANDARE MESSAGGI A dest”, dove dest assume il nome del peer ricevente. Per fare ciò, uso la shortcut with che crea path ripetitive, quindi inserisco nodi mantenendo lo stesso nome:

.filename è una variabile usata per memorizzare il nome del file, .content indica il contenuto del file e .append permette di scrivere i messaggi in ogni riga.

Prima della scrittura dei messaggi sul file, viene effettuato un controllo, ovvero ci si deve accertare che la lunghezza del messaggio sia minore del limite massimo, che

sarebbe di 64 caratteri: il motivo di questo controllo verrà spiegato successivamente nella parte della crittografia.

```
if( lunghezzaMessaggio < limiteLunghezzaMessaggio ){ //Controllo lunghezza messaggio
    //Scrittura su file
    with( richiesta ) {
        .filename = "BackupChat/DATABASE_" + user.name + ".txt"
        .content = Data + "\t" + user.name + ":" + responseMessage + "\n"
        .append = 1
    }
    writeFile@File( richiesta )() //Scrittura su file
```

Se la condizione nell'if è vera, allora avviene la scrittura dei messaggi su file chiamando l'operazione writeFile che è implementata dal servizio File:

```
writeFile@File( richiesta )() //Scrittura su file
```

Questo è quello che riguarda il peer A, che per noi sarebbe il peer Client, quello che invia. Ora analizziamo il peer B, ovvero il ricevente.

Questo peer riceve la richiesta di chat da parte del peer A e deve rispondere: Se la risposta è positiva, (in questo caso si deve passare la condizione all'interno dell'if, secondo cui responseQuestion è ==0, che sarebbe il valore attribuito al bottone che indica "yes"), nel file del peer in questione viene registrata la stringa "INIZIO A RICEVERE MESSAGGI DA username", dove username sarebbe il nome del peer A.

```
//RICEVI RICHIESTA DI CHAT
[
    chatRequest( username )( response ) {
        showYesNoQuestionDialog@SwingUI( username + " vuole inviarti un messaggio. Vuoi accettare ed iniziare a ricevere messaggi" )
        if( responseQuestion == 0 ) {
            response = true
            println@Console( "Per rispondere a " + username + " avvia una chat con lui/lei." )()

            //Scrittura apice del messaggio
            synchronized( lockPrivateChat ) {
                with( richiesta ) {
                    .filename = "BackupChat/DATABASE_" + global.user.name + ".txt"
                    .content = "\nINIZIO A RICEVERE MESSAGGI DA " + username + "\n"
                    .append = 1
                }
                writeFile@File( richiesta )() //Scrittura su file settato in precedenza
            }
        } else {
            response = false
        }
    }
]
```



Una volta che è stata accettata la richiesta del peer A, inizia lo scambio di messaggi:

nella prossima sezione viene spiegato l'utilizzo della parola chiave "synchronized".

```
//METODO PER SCAMBIARSI MESSAGGI PRIVATI .
[
    sendString( plaintextRequest )( response ) {

        request.message = plaintextRequest.text
        request.publickey1 = global.chiaviPubbliche.publickey1
        request.pub_priv_key = global.chiavePrivata.privatekey
        request.cripto_bit = 1 //Settato ad 1 permette di sfruttare RSA con padding
        Decodifica_RSA@DecryptingServiceOutputPort( request )( plainTextResponse )

        response = "ACK" //Utilizzabile per verificare la corretta ricezione di messaggio

        //Formato settato
        requestFormat.format = "dd/MM/yyyy HH:mm:ss"

        //Regisrazione data ed ora del messaggio
        getCurrentDateTime@Time( requestFormat )( responseDateTime )

        //Stampa messaggio con data, ora e username
        println@Console( responseDateTime + "\t" + plaintextRequest.username + " : " + plainTextResponse.message )()

        //Scrivo nel file
        //Metodo synchronized perchè non ci possono essere scritture contemporanee, questo
        //perchè più peer possono scrivere ad un singolo peer
        synchronized( lockFile ) {
            with( richiesta ) {
                .filename = "BackupChat/DATABASE_" + global.user.name + ".txt"
                .content = responseDateTime + "\t" + plaintextRequest.username + ":" + plainTextResponse.message + "\n"
                .append = 1
            }
            writeFile@File( richiesta )()
        }
    }
]
```



Ogni messaggio scritto sul file viene accompagnato dalla data e ora di invio del messaggio: questo è permesso dal servizio Time il quale implementa l'operazione getCurrentDateTime, anch'essa di tipo RequestResponse perché dà in output un elemento di tipo int, che indica la data e l'ora corrente.

```
getCurrentDateTime@Time()(Data)
```

Vediamo un esempio di output: Esempio: Alice e Bob decidono di collegarsi alla rete e di mandarsi dei messaggi. Mario invia la richiesta di chat a Luigi e quest'ultimo accetta.

Vengono creati quindi i file, nel caso in cui non esistano già (caso in cui i due ragazzi abbiano mandato messaggi anche in precedenza) e si scrivono i messaggi. Si specifica chi ha iniziato a mandare i messaggi e chi li ha ricevuti. Di seguito, vengono mostrati i due file.

INIZIO A RICEVERE MESSAGGI DA Bob	INIZIO A MANDARE MESSAGGI A Alice
24/07/2020 17:08:13 Alice: Ciao Bob	24/07/2020 17:08:13 Alice: Ciao Bob
24/07/2020 17:08:20 Bob: Ciao Alice	24/07/2020 17:08:20 Bob: Ciao Alice
24/07/2020 17:08:24 Alice: Come stai?	24/07/2020 17:08:24 Alice: Come stai?
24/07/2020 17:08:31 Bob: Tutto bene tu?	24/07/2020 17:08:31 Bob: Tutto bene tu?
24/07/2020 17:08:34 Alice: Bene	24/07/2020 17:08:34 Alice: Bene

CHAT PUBBLICA

Per quanto riguarda i gruppi, questi vengono creati da un peer connesso alla rete.

```
//CHAT PUBBLICA
define startGroupChat {

    //inizializzazione persistenza .
    with( richiesta ) {
        .filename = "BackupChat/DATABASE_" + user.name + ".txt"
        .content = "\nINIZIO COMUNICAZIONE CON GRUPPO " + group.name + "\n"
        .append = 1
    }
    writeFile@File( richiesta )()
```

Quando il peer host crea il gruppo, nel suo file viene registrata la stringa "INIZIO COMUNICAZIONE CON GRUPPO group.name", dove group.name assume il nome del gruppo creato in precedenza. Questo dovrà apparire anche nei file di ogni peer che decide di partecipare al gruppo. I messaggi che vengono mandati successivamente nel gruppo vengono scritti nei file di ogni peer nel seguente modo:

```
//Settaggio per scrittura messaggio su file .
synchronized( lockForwardMessage ) {
    with( richiesta ) {
        .filename = "BackupChat/DATABASE_" + global.user.name + ".txt"
        .content = responseDateTime + "\t" + msg.username + ": " + msg.text + "\n"
        .append = 1
    }

    //Scrittura effettiva del messaggio .
    writeFile@File( richiesta )()
}
```

Esempio: Abbiamo tre peer Alice, Bob e Sara che accedono alla rete. Alice crea un gruppo chiamato NEWGROUP a cui partecipano anche gli altri due.

L'esempio di file txt per Alice, Bob e Sara sarà, rispettivamente, come segue:

<pre>INIZIO COMUNICAZIONE CON GRUPPO NEWGROUP 27/07/2020 10:46:12 Alice: Ciao gruppo! 27/07/2020 10:46:22 Bob: Ciao Alice e Sara! 27/07/2020 10:46:40 Sara: Ciao Alice e Bob!</pre>	<pre>INIZIO COMUNICAZIONE CON GRUPPO NEWGROUP 27/07/2020 10:46:12 Alice: Ciao gruppo! 27/07/2020 10:46:22 Bob: Ciao Alice e Sara! 27/07/2020 10:46:40 Sara: Ciao Alice e Bob!</pre>
<pre>INIZIO COMUNICAZIONE CON GRUPPO NEWGROUP 27/07/2020 10:46:12 Alice: Ciao gruppo! 27/07/2020 10:46:22 Bob: Ciao Alice e Sara! 27/07/2020 10:46:40 Sara: Ciao Alice e Bob!</pre>	

4.3 Gestire le richieste concorrenti: Synchronized

L'implementazione di writer deve tenere conto di possibili richieste concorrenti da parte di più servizi publisher per scrivere su una stessa risorsa, che nel nostro caso sarebbe un file: si pensi ad un sistema di chat pubblica, dove più peer possono pensare di mandare un messaggio nello stesso momento.

Nel nostro caso il rischio è quello di incorrere in una eccezione nella richiesta di scrittura sul file. In Jolie, questo risulterebbe in un errore nella classe che gestisce il Thread di scrittura, e la conseguente interruzione del servizio writer.

In questo caso, la gestione delle richieste concorrenti non è adeguata e dovremo ricorrere ad un costrutto specifico del linguaggio Jolie per gestire l'accesso alla risorsa, ovvero **synchronized**, il quale ci permette di vincolare l'accesso alle variabili globali (similmente a Java).

```
//Scrittura apice del messaggio
synchronized( lockPrivateChat ) {
    with( richiesta ) {
        .filename = "BackupChat/DATABASE_" + global.user.name + ".txt"
        .content = "\nINIZIO A RICEVERE MESSAGGI DA " + username + "\n"
        .append = 1
    }
    writeFile@File( richiesta )() //Scrittura su file settato in precedenza
}
```

```
//Settaggio per scrittura messaggio su file .
synchronized( lockForwardMessage ) {
    with( richiesta ) {
        .filename = "BackupChat/DATABASE_" + global.user.name + ".txt"
        .content = responseDateTime + "\t" + msg.username + ":" + msg.text + "\n"
        .append = 1
    }

    //Scrittura effettiva del messaggio .
    writeFile@File( richiesta )()
}
```

```
synchronized( lockFile ) {
    with( richiesta ) {
        .filename = "BackupChat/DATABASE_" + global.user.name + ".txt"
        .content = responseDateTime + "\t" + plaintextRequest.username + ":" + plainTextResponse.message + "\n"
        .append = 1
    }
    writeFile@File( richiesta )()
}
```

5 Gestione del monitor

5.1 Introduzione

Per quanto riguarda la realizzazione ed implementazione del nostro monitor, avevamo pensato inizialmente ad un una struttura basilare, la quale permettesse di offrire all'utente una semplice console (terminale) in grado di far visualizzare le informazioni e i log dei peer presenti in rete.

Tale soluzione, in seguito, ci è sembrata modellabile a nostro piacere sfruttando un javaservice che ci permettesse di aggiungere al nostro progetto un po' di colori: attraverso Adobe Photoshop è stato possibile realizzare un logo che ci rappresentasse appieno; appunto un'amaca. Il passo successivo è stato capire come introdurre questo logo nella nostra console e soprattutto come strutturare il nostro javaservice.

Esempio: supponiamo che ci sia un utente di nome Bob che cerca di effettuare un login alla rete: nel nostro caso ci sarà prima una ricerca sulla prima porta disponibile e successivamente l'inserimento di uno username diverso da quelli già esistenti, come già spiegato nei paragrafi precedenti. Successivamente, se tutto il procedimento di login va a buon fine, verrà inviato un messaggio alla console sfruttando un'outputPort con un numero di porta fisso.

5.2 Struttura ConsoleStampa

Utilizzando una console di questo genere, come specificato nell'esempio che precede, abbiamo sfruttato una location fissa (numero di porta 30000) che permettesse ad ogni peer di inviare tutte le informazioni necessarie: login alla rete, inizio comunicazione con un altro peer disponibile in rete, invio di possibili errori come la creazione di un gruppo già esistente e invio di abbandono chat o rete. Di seguito il codice della inputPort presente nel file “ConsoleStampa.ol”:

```
inputPort portaStampaConsole {
    Location: "socket://localhost:30000"
    Protocol: http
    Interfaces: teniamoTraccia
}
```

(a) inputPort

```
interface teniamoTraccia {
    RequestResponse: press( string )( void )
}
```

(b) Interface

Figure 35: Porta ed interfaccia

Nella inputPort possiamo visualizzare un’interfaccia denominata “teniamoTraccia”. Essa è una RequestResponse (di tipo bloccante che ci permetta, quindi, di ultimare la scrittura delle informazioni e log principali su console). Tale interfaccia si aspetta un tipo stringa come request (la stringa da replicare su console) e come response un tipo void bloccante per ultimare, come già detto la scrittura.

Soffermiamoci ora sulle due parti principali del nostro file: init e main. Nella init abbiamo il primo collegamento al javaservice denominato “JavaSwingConsole”: sfruttando un metodo “aperturaConsole”, riusciamo a far visualizzare a video la console (spiegazione più avanti). Successivamente, troviamo un contatore di tipo global, questo perché ogni qual volta avviene una richiesta di scrittura su monitor, bisogna incrementare tale contatore (inizialmente settato al valore 0).

Spostandoci sul main, possiamo verificare immediatamente il servizio RequestResponse denominato “press”, il quale rimane in attesa di richieste (sfruttando appunto nel file la execution concurrent).



execution{ concurrent }

Figure 36: Esecuzione concorrente

Ovviamente, usufruendo di un contatore, ci siamo accorti di avere alcuni problemi di concorrenza (tale circostanza accade raramente) e per ovviare a questo abbiamo deciso di introdurre il costrutto synchronized, il quale permette un accesso per volta alla scrittura su monitor, assicurando così la mutual exclusion di accesso a sezione critica.

Esempio: Alice, Bob e Pippo vogliono entrare in rete, supponiamo che tutti e tre effettuino la richiesta di entrata contemporaneamente o in un tempo molto ristretto: visualizzeremo su monitor un incremento delle variabile counter, ma non in modo continuo.

Oltre al metodo synchronized, una soluzione che ci sembrava possibile era quella di sfruttare un timer attraverso il servizio messo a disposizione da Jolie chiamato “sleep@Time(millisecondi)()”, il quale generava un thread che permetteva il bloccaggio del sistema, ultimando così la scrittura senza sfruttare il costrutto di sincronizzazione, ma ci è sembrato più corretto sfruttare quest’ultimo in quanto rispecchiava il concetto di gestione ottimale di concorrenza.

```

init {
    //APERTURA CONSOLE .
    aperturaConsole@JavaSwingConsolePort(){}
    global.counter = 0 //Setto un contatore .
}

main {
    [
        press( message )() {
            synchronized( lockConsole ) { //AGGIUNTA DI SYNCHRONIZED PER PROGRESSIONE CONTATORE .
                global.counter = global.counter + 1
                modificaConsole@JavaSwingConsolePort( global.counter + " . " + message )()
            }
        }
    ]
}

```

Figure 37: init e main di ConsoleStampa.ol

Anche in questo caso, come nella init, sfruttiamo un metodo presente nel javaservice denominato modificaConsole, attraverso il quale riusciamo a trascrivere le informazioni necessarie passate attraverso il servizio “press”.

5.3 Javaservice e struttura dei metodi utilizzati

Per collegare Jolie al Java abbiamo sfruttato una outputPort utile per inviare le richieste al javaservice con annesso una interfaccia che contenesse al suo interno due RequestResponse: aperturaConsole e modificaConsole. Infine, abbiamo effettuato un embedded per collegare il javaservice alla outputPort JavaSwingConsolePort. Di seguito il codice:

```

outputPort JavaSwingConsolePort {
    interfaces: ISwing
}

interface ISwing {
    RequestResponse: aperturaConsole( void )( void )
    RequestResponse: modificaConsole( string )( void )
}

embedded {
    Java:
        "blend.JavaSwingConsole" in JavaSwingConsolePort
}

```

Figure 38: connessione al javaservice

Nel javaservice la prima parte che abbiamo implementato è stata quella della “aperturaConsole”, essa ci è servita principalmente per creare il frame sul quale siamo andati a trascrivere i log e le informazioni descritte in precedenza.

Per l'implementazione abbiamo sfruttato la libreria java Swing (`javax.swing.*`), attraverso la quale è stato possibile generare: una sezione che racchiudesse tutte le strutture implementate denominata per l'appunto `JFrame`, un area di testo, `JTextArea`, che ci ha permesso di far visualizzare i messaggi ed infine una `JScrollPane` che racchiudesse al suo interno l'area di testo, che ci ha permesso in seguito la visualizzazione dei log nel momento in cui il contenitore (`JFrame`) raggiungeva la sua massima capienza.

Per inserire, come già accennato in precedenza, l'immagine realizzata abbiamo creato una nuova classe esterna alla `JavaSwingConsole`, la quale estende `JTextArea` e che permette l'aggiunta del nostro logo. Riportiamo di seguito il codice:



```
//CREAZIONE TEXT AREA DELLA CLASSE textAreaWithImage E SETTAGGI .
textArea = new textAreaWithImage( 5, 20 );
textArea.setBackground( new Color(1,1,1, ( float ) 0.01) );
textArea.setEditable( false );

public textAreaWithImage( int a, int b ) {
    super( a, b );

    try {
        immagine = ImageIO.read( new File( "services/src/main/java/blend/logoAmaca.jpeg" ) );
    } catch( IOException exception ) {
        exception.printStackTrace();
    }
}
```

Figure 39: Costruttore e inizializzazione `textAreaWithImage`

Come possiamo osservare, il costruttore della classe prende in input due parametri che rappresentano le dimensioni dell'immagine e nel nostro caso la dimensione dell'area di testo generata assieme al logo creato. L'ultimo metodo, diciamo il più importante che permette la vera e propria scrittura, è denominato “modificaConsole” e ci ha permesso di ricevere come parametro il messaggio di tipo stringa e di trascriverlo successivamente sulla console realizzata. Di seguito il codice per comprendere al meglio tutti i passaggi:

```

public void modificaConsole( Value request ){

    //RICEZIONE TESTO .
    String text = request.strValue();

    //ISTANZA DI CALENDAR .
    Calendar calendario = Calendar.getInstance();

    //SETTAGGIO DATA CON GIORNO, MESE E ANNO .
    SimpleDateFormat data = new SimpleDateFormat("dd/MM/yyyy");

    //SETTAGGIO ORA .
    SimpleDateFormat ora = new SimpleDateFormat("HH:mm:ss");

    //VISUALIZZAZIONE MESSAGGIO SU CONSOLE .
    textArea.append( "(" + data.format( calendario.getTime() ) + ", " + ora.format( calendario.getTime() ) + ") - " + text + "\n");
}

```

Figure 40: Metodo *modificaConsole*

Possiamo notare nella parte cerchiata in rosso, come avviene la scrittura del messaggio passato come parametro attraverso Jolie. Prima di ciò possiamo notare alcune variabili generate attraverso classi che ci hanno permesso la scrittura della data e dell'ora esatta di avvenuta trascrizione di informazioni su console.

Esempio: nel momento in cui un utente, ad esempio Alice, effettua il login alla rete AMACAp2p, sfruttando l'outputPort “portaStampaConsole” invia un messaggio del seguente genere: “Alice si è unito/a alla rete”, alla console (ConsoleStampa.ol) attraverso l'inputPort in ascolto e successivamente entra nel metodo synchronized, incrementa il contatore e invia il messaggio (message), combinato al counter incrementato, al javaservice che, con i metodi precedentemente descritti, trascrive il tutto sul frame generato.

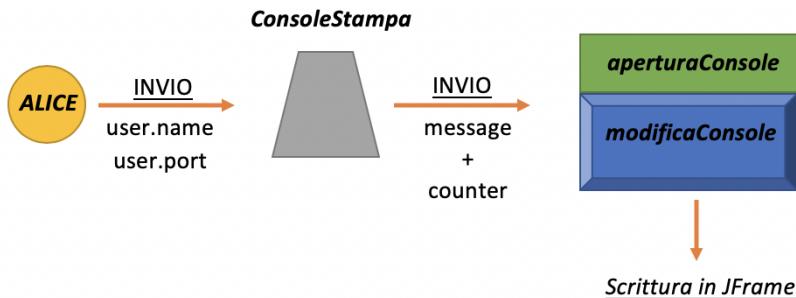


Figure 41: Schema riassuntivo

6 Menù di scelta in Java Swing Console

Dopo un'attenta riflessione, ci siamo accorti che il metodo di input presente in Jolie, il quale sfrutta *in(jvariabile di inserirej)*, comportava dei problemi attraverso l'utilizzo della “*registerForInput@Console()()*”; questo perché la console veniva condivisa per le varie richieste di input causando problemi di concorrenza.

Esempio: uno dei primi problemi di concorrenza è proprio l'utilizzo di variabili condivise, infatti come visto a lezione, operazioni su variabili condivise possono portare a problemi definiti come *perdite di informazioni* o, per meglio dire, *problemi di interferenza*.

Supponiamo ad esempio che ci sia una variabile condivisa denominata *Temp*, alla quale viene assegnato un valore iniziale ad esempio 10 e due processi *P1* e *P2* che effettuano rispettivamente un incremento e un decremento di 10 unità. Alla fine dell'esecuzione (in parallelo) potremmo ottenere dei risultati detti inconsistenti, questo perché ci potrebbe essere una perdita di informazioni. Allo stesso modo, nella console potremmo avere un bloccaggio o il recapito di un valore che non ci aspettavamo.

Per porre rimedio a tutto ciò, una prima soluzione è stata quella di creare un canale di comunicazione che permettesse l'apertura e la chiusura per l'inserimento di un input da console. Di seguito il codice di questa prima idea:

```
//CANALE DI COMUNICAZIONE .
with( service ) {
    .token = ""
}

subscribeSessionListener@Console( service )() //Apro il canale di comunicazione .

in( message )

unsubscribeSessionListener@Console( service )() //Chiudo il canale di comunicazione .
```

Viene aperta una sessione o canale di comunicazione attraverso una stringa, per ricevere qualcosa in input.

Viene disabilitata la sessione o canale di comunicazione con la stessa stringa attraverso la quale tale canale era stato aperto.

Figure 42: Canale di comunicazione

Una soluzione migliore e soprattutto ottimale ci è stata proposta dal Dott. Stefano Pio Zingaro, il quale ci ha consigliato l'utilizzo della SwingUI presente in Jolie e implementata dagli sviluppatori tra cui l'ideatore, il professore Fabrizio Montesi.

Tale soluzione, in effetti, non consente l'utilizzo della console, ma sfrutta una tecnologia resa disponibile a Jolie grazie ad un javaservice già implementato. All'inizio abbiamo sfruttato tale servizio già reso disponibile, ma solo successivamente ci siamo accorti che avevamo bisogno di altro.

Nella nostra struttura **P2P**, dopo l'identificazione del peer noviziato e il successivo login (inserimento del solo username identificativo) viene aperta una nuova finestra sfruttando la “**showInputDialog@SwingUI()**”, essa comportava dei problemi sulla scelta dell'istruzione che il peer voleva eseguire, così abbiamo deciso di realizzare un nostro javaservice che mettesse a disposizione dei pulsanti di scelta.

Sempre nella JavaSwingConsole, possiamo trovare un metodo oltre a quelli già descritti nel paragrafo precedente, che permette di far visualizzare una finestra con dei pulsanti. Riportiamo di seguito il codice della nostra implementazione:

```
public Integer aperturaMenu( Value request ) {  
  
    //Acquisisco valore stringa in ingresso .  
    String stringa = request.strValue();  
  
    //Inizializzo una stringa di bottoni .  
    String[] buttons = { "CHAT PRIVATA", "PARTECIPA", "EXIT", "CREA GRUPPO" };  
  
    //RITORNO IL VALORE INTERO DA CONFRONTARE NEL PeerAA.ol .  
    return JOptionPane.showOptionDialog(null, stringa, "SCEGLI ISTRUZIONE", JOptionPane.WARNING_MESSAGE, 0,  
        new ImageIcon("services/src/main/java/blend/icoAmaca.png"), buttons, buttons[2]);  
}
```

Figure 43: JOptionPane creata da *AMACAp2p*

Come si nota dal metodo esso viene definito come un **Integer**, questo perché ogni pulsante (button) restituirà un valore intero a Jolie. Per realizzare tutto ciò, come si nota dalla parte di codice evidenziata, viene restituito un `JOptionPane.showOptionDialog` accompagnato da: **stringa** che rappresenta il messaggio preso dal nostro lato client Jolie **PeerA.ol** ed inviato direttamente a Java, un `JOptionPane.WARNING_MESSAGE` che permette di far effettuare al peer la scelta, i bottoni inizializzati nella parte di codice superiore ed infine il settaggio dell'icona del nostro team.

Di seguito un semplice schema di chiarimento:

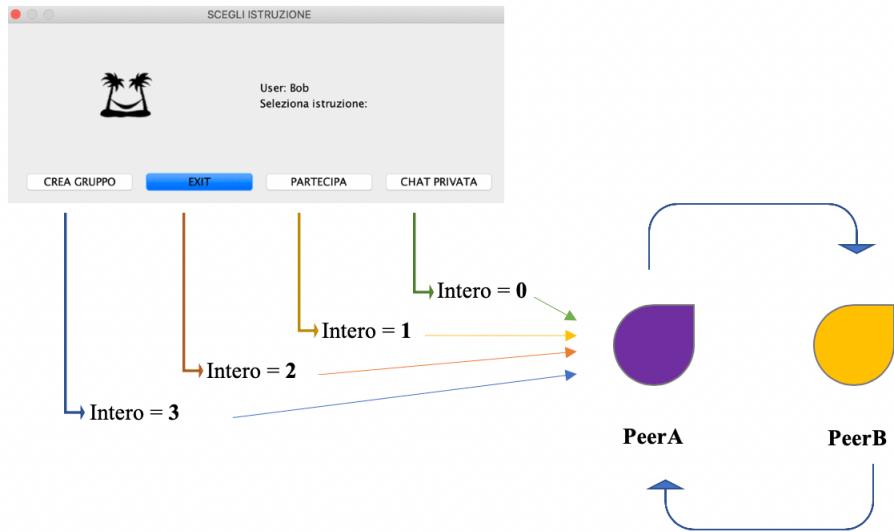
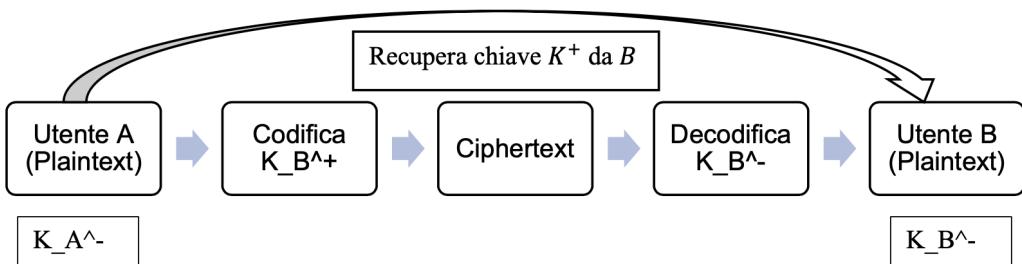


Figure 44: Schema riassuntivo

7 Crittografia

7.1 Crittografia a chiave pubblica:



Se Alice e Bob vogliono che le loro conversazioni rimangano segrete, e che solo loro possano essere in grado di comprendere il contenuto del messaggio trasmesso, allora è necessaria la **crittografia** in modo da rendere illeggibile i messaggi a qualche possibile intruso.

Nel nostro caso viene utilizzato un cifrario a chiave pubblica, dove non esiste una sola chiave segreta per la cifratura e decifratura ma bensì due chiavi, una per la cifratura detta **pubblica** (disponibile a chiunque) e una **privata** per riportare il testo in chiaro.

K_B^+	Chiave pubblica
K_B^-	Chiave privata
$K_B^+(m)$	Codifica ciphertext da Alice via K_B^+
$K_B^-(K_B^+(m))$	Decodifica ciphertext da Bob via K_B^-

Dunque, Alice e Bob scelgono una chiave segreta, ognuno per conto proprio e che non verranno comunicate a nessuno: K_A^-, K_B^- . Tramite una funzione computazionalmente facile, ma quasi impossibile da invertire, Alice e Bob generano la loro propria chiave pubblica che potrà essere fornita a chiunque. Se Alice vuole mandare un messaggio a Bob è sufficiente utilizzare la chiave pubblica di Bob per cifrare il messaggio e spedirglielo. Solo Bob tramite la sua chiave segreta, potrà riconvertire in chiaro e velocemente il messaggio di Alice.

7.2 Algoritmo RSA:

Uno degli algoritmi più famosi che soddisfa i requisiti esposti è l'**algoritmo RSA**, che fa largo uso di operazioni aritmetiche in modulo n . Illustriamolo nei passi seguenti:

Generazione della chiave **pubblica** e **privata**:

- | |
|--|
| 1. Scegliere due numeri primi p, q , tanto più grande sarà il loro valore tanto più difficile risulterà violare RSA. (Consigliato p, q dell'ordine > 1024 bit) |
| 2. Calcolare $n = p * q$ |
| 3. Scegliere un numero $e < n, e \neq 1$ che NON abbia fattori comuni con $F(n) = (p - 1) * (q - 1)$ |
| 4. Trovare un inverso moltiplicativo d tale che $(e * d) \bmod F(n) = 1$
$d = e^{-1} \bmod F(n)$ |

Chiave pubblica: n, e

Chiave privata: n, d

plaintext: m

codifica in ciphertext: $c = m^e \bmod n$
decodifica in plaintext: $m = c^d \bmod n$

Nella realtà i due numeri primi p, q vengono generati casualmente della lunghezza desiderata, ma sorge qui il problema di verificare se questi due numeri casuali siano realmente primi. Il **controllo di primalità** di un numero è la classificazione di quest'ultimo come primo o come composto. Quest'ultima può essere effettuata attraverso un test statistico che si basa sul **Piccolo Teorema di Fermat**:

Se p è un numero primo, con b senza fattori comuni con p , allora $b^p \bmod p = b$

Se un numero supera il Piccolo Teorema di Fermat per almeno 100 basi è 'probabilmente quasi certo' che sia primo.

Esempio funzionamento RSA:

- 1) $p = 5, q = 11$;
- 2) $n = p * q \rightarrow 5 * 11 = 55 \leftarrow$ **prima componente chiave pubblica e privata**
- 3) $F(n) = (p - 1) * (q - 1) = 40$
- 4) $e = 3 \rightarrow MCD(3, 40) = 1 \rightarrow$ nessun fattore in comune \leftarrow **seconda componente chiave pubblica**
- 5) $d = e^{-1} \bmod F(n) = 27 \leftarrow$ **seconda componente chiave privata**

Plaintext: $m = 7$

Chiave pubblica: 55, 3

Chiave privata: 55, 27

Codifica: $7^3 \bmod 55 = 13$

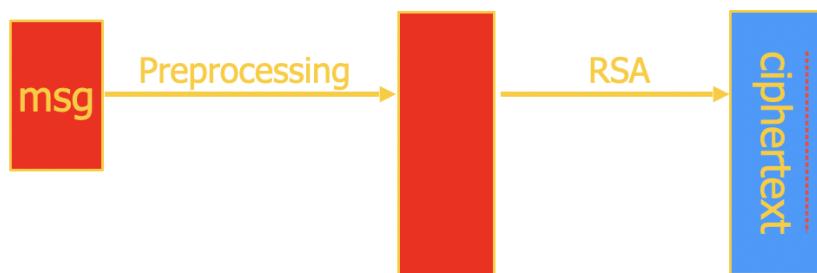
Decodifica: $13^{27} \bmod 55 = 7$

7.2.1 L'algoritmo RSA non è sicuro:

L'algoritmo RSA così descritto non è sicuro, infatti esistono numerosi problemi che rendono completamente inutilizzabile l'RSA nella sua forma standard:

- **Determinismo**, e dunque non semanticamente sicuro, infatti è possibile estrarre informazioni del plaintext, semplicemente dal ciphertext.
- **Lunghezza dei messaggi**, se ad esempio viene scelto un esponente e per la cifratura, qualsiasi messaggio m più piccolo di $\sqrt[e]{N}$ non effettuerà il modulo, permettendo di recuperare il messaggio originale semplicemente effettuando $\sqrt[e]{C}$.
- **Malleabilità**, è possibile trasformare infatti un ciphertext c in un altro, effettuando $c' = c * 2^e \bmod n$. Quando Bob decifrerà il messaggio, otterrà $2m \bmod n$. In altre parole, è possibile effettuare cambiamenti prevedibili al ciphertext.

Per evitare questi problemi, nella nostra implementazione dell'algoritmo RSA si introduce una sorta di sistema di imbottitura, chiamato **padding**.

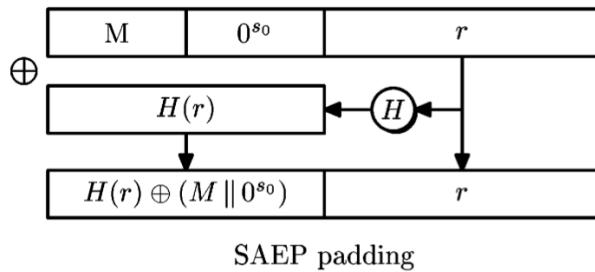


7.2.2 Simplified OAEP:

Il sistema di padding *Simplified OAEP* è una sorta di **rete di Feistel** che utilizza un oracolo random H per effettuare un preprocessing su un messaggio da cifrare asimmetricamente. Questo metodo se usato in combinazione con la funzione botola RSA è considerato semanticamente sicuro.

Soddisfa i due obiettivi seguenti:

- Aggiunge una componente di casualità che converte lo schema di criptaggio deterministico in uno probabilistico.
- La lunghezza del messaggio risultante dall'operazione preliminare di padding sarà fissa di k bit, eliminando il problema della lunghezza minima del messaggio.



$$SAEP(M, r) = ((M \parallel 1 \parallel 0^{s_0}) \oplus H(r)) \parallel r$$

Illustriamo ora il funzionamento per imbottire il messaggio:

- | |
|---|
| 1.al messaggio m trasformato in binario viene aggiunto un 1 per delimitare il messaggio, successivamente vengono aggiunti tanti 0, quanti ne servono per arrivare a lunghezza di 512 bits |
| 2. Viene generata una stringa in binario r che viene passata all'oracolo random (nel nostro caso SHA-512) che ci restituisce una stringa binaria di 512 bits |
| 3. Viene effettuato lo XOR tra $H(r)$ e $(M \parallel 1 \parallel 0^{s_0})$ |
| 4. Al messaggio risultante dallo XOR, viene appeso r al messaggio |

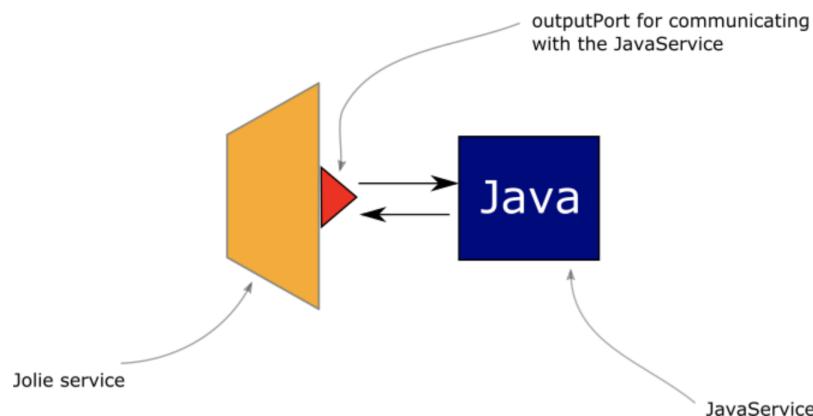
Per riportare il messaggio imbottito all'originale, si dovrà semplicemente prendere r appeso, effettuare l'hash attraverso SHA-512, ottenendo $H(r)$. Verrà successivamente effettuato lo XOR tra $H(r)$ e il messaggio imbottito, ottenendo $(M \parallel 1 \parallel 0^{s_0})$ a cui verrà tolta la concatenazione di 0 e 1.

7.2.3 Javaservice:

Nell'implementare l'algoritmo RSA su Jolie, si è presentato il problema della dimensione massima del tipo *long* di $2^{63} - 1$. Questo è limitante, poiché l'algoritmo RSA per essere considerato sicuro ha bisogno di utilizzare chiavi di ordine molto grande (> 1024 bit).

Nell'esponenziazione di c o m a un d o e molto grande la probabilità di ottenere da Jolie un *infinity* sarebbe certa rendendo inutilizzabile il nostro algoritmo di RSA.

Dunque, abbiamo deciso di utilizzare Java e di implementare tre differenti javaservice per la futura scalabilità della sicurezza: per la generazione delle chiavi, per la cifratura del plaintext e la decifratura del ciphertext.



Queste classi java estendono una classe abstract "JavaServices" che gestisce la conversione automatica tra i tipi di Jolie e i tipi di Java. Inoltre, viene utilizzata la libreria Value per permettere la gestione degli alberi di Jolie in Java.

Ogniqualvolta che un Peer ha bisogno di generare le chiavi, criptare un messaggio o decifrarlo, richiama attraverso una RequestResponse il servizio Java conseguente, che si occupa di tali operazioni.

7.3 Algoritmo SHA:

Lo SHA (Secure Hash Algorithm) è una delle più popolari funzioni di hash di crittografia, che viene utilizzato per la creazione di *"firme digitali"*. Abbiamo implementato l'algoritmo SHA in quanto nella seconda modalità, quella di scrittura su chat pubblica, devono essere garantite sia l'identità del mittente che l'integrità del contenuto dei messaggi. Questo controllo, ovvero il fatto che non ci sia stato nessuno in mezzo alla comunicazione che abbia modificato il messaggio e che la persona che ha inviato il messaggio sia chi dichiara di essere, viene quindi effettuato attraverso l'utilizzo della *"firma digitale"* (un sistema di HASH), che prevede canali pubblici con messaggi firmati. In particolare, l'identità del mittente viene garantita grazie all'utilizzo di una chiave privata per la cifratura e della corrispondente chiave pubblica per la decifratura, entrambe generate e fornite dal lato sender.

La **funzione HASH** $H(m)$ crea un piccolo blocco di dati di dimensione prefissata da un messaggio m , chiamato **valore hash o digest del messaggio**. Il modo di procedere delle funzioni hash è quello di dividere il messaggio in segmenti ed elaborarli in modo da produrre un valore di n bit.

Una funzione *hash* deve avere la seguente proprietà:

dato m , calcolare $m' \neq m$ tale che $H(m) = H(m')$ dev'essere impossibile.

Da ciò deriva il fatto che se $H(m) = H(m')$ allora $m = m'$, ovvero che il messaggio non è stato modificato.

7.3.1 Problematiche degli algoritmi di hash

Molte funzioni hash precedentemente utilizzate come **MD5** o **SHA1** ad oggi sono vulnerabili. Inoltre, una volta compromesse da un attacco crittografico, si ha come risultato una perdita di fiducia da parte degli esperti, e di conseguenza un abbandono della funzione stessa.

Un attacco *hash*, può essere utilizzato per compromettere la sicurezza di un algoritmo di *hash*. Questi attacchi si verificano quando due input diversi producono un output identico. In questo caso è possibile sostituire un input con un altro creando una falla di sicurezza. Dunque, **SHA1** è vulnerabile alle collisioni e per questo motivo si è scelto di utilizzare **SHA2**, che risulta più resistente a quest'ultime.

Le funzioni *hash* sono utili per rilevare eventuali modifiche ai messaggi, ma non sono sufficienti ad autenticarli. Infatti $H(m)$ può essere spedito assieme al messaggio, ma se la funzione H è nota, chiunque potrebbe modificare il messaggio m in un

messaggio m' e computare $H(m')$, producendo quindi un messaggio alterato non più riscontrabile. È per questo motivo che occorre autenticare $H(m)$ con un algoritmo di cifratura.

Sappiamo inoltre che l'algoritmo SHA non può essere implementato in linguaggio di programmazione Jolie. Per questo motivo abbiamo implementato lo SHA-512 in Java e lo abbiamo utilizzato come un Java Service in Jolie. Questo è stato possibile attraverso l'utilizzo del costrutto:

```
embedded {
    Java:
        "blend.ShaAlgorithmService" in ShaAlgorithmServiceOutputPort
}
```

dove “*embedded*” significa appunto farne il caricamento da una sorgente esterna.

Per la sua implementazione, come per l'algoritmo RSA, abbiamo perciò utilizzato il seguente JavaService:

```
public class ShaAlgorithmService extends JavaService {

    private String ByteArrayToBinaryString(byte[] bytes){

        String finalString = "";

        for(int i = 0; i<bytes.length; i++){

            String tempBit = String.format("%8s", Integer.toBinaryString(bytes[i] & 0xFF)).replace(' ', '0');
            finalString = finalString + tempBit;
        }
        return finalString;
    }

    public Value ShaPreprocessingMessage( Value request ) {

        //input
        String s = request.getFirstChild( "message" ).strValue();
        String sb = "";

        try {
            MessageDigest md = MessageDigest.getInstance("SHA-512");
            byte[] data = md.digest(s.getBytes());
            sb = ByteArrayToBinaryString(data);
        } catch(Exception e) {
            System.out.println(e);
        }

        //output
        Value response = Value.create();
        response.getFirstChild("message").setValue(sb);
        return response;
    }
}
```

Metodo *ShaProcessingMessage*:

Creiamo inizialmente due variabili di tipo string: *s*, che conterrà il messaggio da criptare passato in input al metodo; e *sb*, in cui andremo a salvare il messaggio criptato, perciò inizialmente vuota.

```
String s = request.getFirstChild( "message" ).strValue();
String sb = "";
```

Utilizziamo la classe fornita da Java **Messagedigest** integrata per l'hashing SHA-512:

```
MessageDigest md = MessageDigest.getInstance("SHA-512");
```

Creiamo quindi un vettore data di byte che utilizziamo per creare il messaggio codificato attraverso l'uso del metodo **digest()**:

```
byte[] data = md.digest(s.getBytes());
```

Successivamente ci occupiamo di convertire la sequenza di byte ottenuta, attraverso l'utilizzo del metodo '**ByteArrayToBinaryString**' all'interno della classe **ShaAlgorithmService**, che convertirà quindi il valore byte passatogli in input (*data*) nella stringa di binario finale (*sb*).

```
sb = ByteArrayToBinaryString(data);
```

Infine ritorniamo il valore *response* settato con la codifica binaria del messaggio di partenza ottenuta (*sb*).

```
Value response = Value.create();
response.getFirstChild("message").setValue(sb);
return response;
```

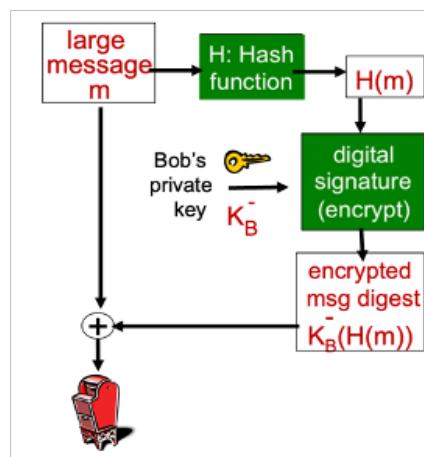
Esempio di **output** di questo algoritmo:

```
Inserisci un messaggio:
messaggio

Message Hash *H(M)* : 0011100110000101000010101110001010100110001001100111100111110
011000101111000101111001110001001100000100011000101001111010011111010010101000000110
1010011011011001010011100101111010000011110000101101001000011011111110000111000111000
01011001100100001100100110011000110011101100001100011101000011000111010000110001101100010001000
0010001101101001111011101011101000100101011010010101110110111101000010010101110101
11100010010001110101011100001100101101100001011100100011111000110100010001101011001
100100000110001110101010
```

7.3.2 Lato sender

Abbiamo utilizzato una funzione HASH, ovvero una funzione "*many-to-one*" poiché non invertibile, con lo scopo di ovviare al problema di tipo computazionale della crittografia a chiave pubblica.



Infatti, con messaggi di una certa lunghezza (come ad esempio contratti), la codifica e la decodifica risultano essere operazioni molto costose. Si è impiegata quindi una funzione che, dato un messaggio di lunghezza variabile, restituisce come risultato un output di lunghezza fissa ed in generale molto minore del messaggio di partenza. Esso costituirà una sorta di *impronta* del messaggio originale. In particolare ci interessano qui delle funzioni hash per le quali sia computazionalmente impossibile, quindi difficile da un punto di vista pratico, trovare due messaggi che danno lo stesso output finale.

Per effettuare l'hash di un messaggio è stato quindi necessario integrare un servizio Java, che si occupa di prendere un valore Jolie (*hash*) contenente il messaggio m, fare il message digest con un algoritmo SHA e restituirlo in un valore Jolie (*hash_response*):

```
hash.message = responseMessage
ShaPreprocessingMessage@ShaAlgorithmServiceOutputPort ( hash ) ( hash_response )
```

Al termine di queste operazioni otterremo quindi $H(m)$, che sarà di dimensione molto più breve rispetto al messaggio di partenza m .

La codifica dell'hash del messaggio $H(m)$ è stata poi effettuata attraverso il metodo Codifica_RSA, che prende in input un valore *codifica*, contenente l'hash del messaggio (*message*), la chiave privata (*publickey1* e *pub_priv_key*) ed un *cripto_bit* inizializzato a 0 per l'esecuzione dell'algoritmo SHA piuttosto che dell'algoritmo RSA nel Servizio Java EncryptingService.

```
codifica.message = hash_response.message
codifica.publickey1 = chiaviPersonalisResponse.publickey1
codifica.pub_priv_key = chiaviPersonalisResponse.privatekey
codifica.cripto_bit = 0

Codifica_RSA@EncryptingServiceOutputPort( codifica )( codifica_response )
```

Il metodo infine restituirà il criptato dell'hash del messaggio con la chiave privata del peer mittente attraverso il valore *codifica_response*.

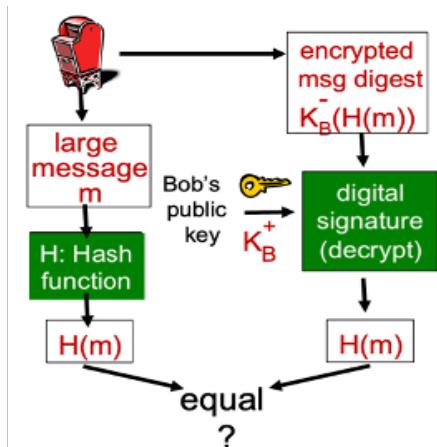
A questo punto il sender si occuperà di inviare al peer ricevente la coppia messaggio (*text*) e criptato dell'hash del messaggio (*message*): [$m, K^-(H(m))$] Sarà inoltre necessario l'invio della chiave pubblica (composta dalle due componenti *publickey1* e *publickey2*).

```
msg.text = responseMessage
msg.message = codifica_response.message
msg.publickey1 = chiaviPersonalisResponse.publickey1
msg.publickey2 = chiaviPersonalisResponse.publickey2

sendMessage@port( msg )
```

7.3.3 Lato receiver

Al momento della decodifica, il receiver deve verificare che le due componenti ricevute siano *congruenti*. Per accertarsi di ciò è necessario quindi effettuare un confronto tra l'hash generato e l'hash acquisito:



- 1) Il peer B, ricevuto il messaggio m e $H(m)$, e conoscendo la chiave pubblica del mittente, potrà riportare in chiaro la funzione **HASH** con la chiave pubblica del mittente per verificarne *l'autenticità*.
- 2) Successivamente, per verificare invece *l'integrità* del messaggio, si effettua la decodifica del message digest applicando la chiave pubblica K^+ ricevuta dal peer mittente e calcolando perciò $K^+(K^-(H(m)))$, da cui si ottiene l'hash del messaggio originale $H(m)$.

Se il confronto tra i due $H(m)$ da esito positivo, allora si ha la certezza dell'identità del firmatario. Se così non fosse, allora il messaggio è stato corrotto nel corso della sua trasmissione.

```
[forwardMessage( msg )] {
    firma.message = msg.message
    firma.publickey1 = msg.publickey1
    firma.pub_priv_key = msg.publickey2
    firma.cripto_bit = 0

    Decodifica_RSA@DecryptingServiceOutputPort( firma )( firma_decodificata )

    plaintext.message = msg.text
    ShaPreprocessingMessage@ShaAlgorithmServiceOutputPort( plaintext )( hash_plaintext )

    if( hash_plaintext.message == firma_decodificata.message ) {
        writeFile@File( richiesta )()
    } else {
        println@Console( "Il messaggio è corrotto." )()
    }
}
```

Abbiamo quindi utilizzato una variabile **firma** contenente l'*hash del messaggio cifrato*, la *chiave pubblica* ed il *cripto bit* a 0. La decodifica viene effettuata con il metodo *Decodifica_RSA* mediante l'uso della chiave pubblica, che restituirà l'hash del messaggio originale (*firma_decodificata*). Mentre il metodo *ShaProcessingMessage* rigenera l'hash a partire dal messaggio originale contenuto nella variabile *plaintext*. Ottenuti hash acquisito e hash generato viene effettuato un confronto tra questi due. Nel caso di successo il messaggio viene salvato su file, mentre in caso contrario, viene

stampato un messaggio d'errore su Console.

Esempio generazione e scambio di chiavi:

Si supponga che Alice desideri comunicare con N utenti in una chat di gruppo, quindi con quello che è un potenziale destinatario che chiamiamo Bob.

Grazie all'utilizzo degli **algoritmi a chiave asimmetrica**, ogni utente può scambiarsi le chiavi in pubblico e necessita di una sola chiave privata, indipendentemente dal numero di persone coinvolte nella comunicazione. Ogniqualvolta che Alice invia un messaggio, come prima cosa il peer A deve richiedere l'invio delle proprie chiavi personali al peer B , che si occupa di generarle. Una volta ottenute le chiavi, ed avendo generato attraverso l'algoritmo SHA l'hash del messaggio, il peer A può criptare quest'ultimo con la chiave privata K^- (composta dalle due componenti n e d segreta). Il peer A di Alice invia poi al peer B di Bob il messaggio \mathbf{m} , il criptato dell'hash del messaggio $K^-(\mathbf{H}(\mathbf{m}))$ e le due componenti della chiave pubblica \mathbf{n} ed \mathbf{e} . Il peer B , che riceve, potrà quindi riportare in chiaro l'hash criptato con la chiave pubblica di Alice, da cui ha ricevuto il messaggio, ed effettuare il confronto tra i due hash per provare l'*autenticità* di Alice e l'*integrità* del messaggio.

