Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica per il Management

Comunicazioni sicure su canali eterogenei: un'analisi sistematica di SSL da Java a Jolie

Relatore:
Chiar.mo Prof.
Davide Sangiorgi

Presentata da:
Davide Zanetti

Correlatore:
Saverio Giallorenzo

Sessione II
Anno Accademico 2014/2015

A tutte le persone che mi sostengono e credono in me,
grazie a voi riesco a rendermi conto che nella vita,
con impegno, sacrificio ed un po' di fortuna,
posso raggiungere qualsiasi traguardo.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Comunicazioni sicure in rete

1.1 Introduzione

Affermare che nella vita di ogni essere umano la comunicazione sia un fatto di fondamentale importanza non è un'esagerazione; si pensi ad esempio all'influenza che il desiderio di comunicare con un sempre più crescente numero di ascoltatori abbia avuto sullo sviluppo tecnologico, soprattutto quello più recente.

In campo informatico, si è partiti negli anni '70 con Arpanet, una piccola rete di computer studiata per lo sviluppo di tecnologie militari, fino ad arrivare ad Internet, una rete mondiale di reti di computer ad accesso pubblico, che rappresenta allo stato attuale il principale mezzo di comunicazione di massa grazie alla sua estrema semplicità di utilizzo ed al suo sviluppo rapidissimo, il quale ha rappresentato una vera e propria rivoluzione tecnologica e socio-culturale dagli inizi deli anni '90 e che tuttora è fonte di ispirazione per tecnologie sempre più avanzate.

Viviamo in un'epoca in cui la presenza della rete è ovunque e la condivisione di risorse e informazioni con altri soggetti è una cosa ormai normalissima. Sia che si legga la posta elettronica, si gestisca un server, si consultino articoli su una pagina web oppure si acquisti della merce tramite il commercio elettronico, si è coinvolti in un qualsiasi processo di comunicazione senza nemmeno rendersene conto.

Fin dalla nascita delle prime reti di computer, però, le preoccupazioni maggiori sono state rivolte verso l'importanza di scambiarsi dati, rispetto alla sicurezza degli stessi, portando così alla costruzione di un'infrastruttura che fondamentalmente è insicura, all'interno della quale le informazioni vengono scambiate in chiaro e qualsiasi malintenzionato, che avesse il controllo di una macchina che è situata in un punto qualsiasi del percorso di comunicazione che connette due terminali, può spiare il traffico ed analizzarlo alla ricerca di informazioni private (quali passwords, numeri di carte di credito, dati personali, ecc.). [1]

Per rendere più chiaro questo particolare si pensi ad una comunicazione in rete da un punto all'altro come un sentiero con numerose fermate, queste ultime rappresentate da altri terminali, sparsi in posizioni imprecisate ed appartenenti a reti diverse da quella di partenza, i quali prendono il nome di gateway. Come specificato in precedenza, Internet rappresenta una rete mondiale di reti di computer; proprio per questo, per collegare tra loro computer situati in reti diverse, che potrebbero non "parlare lo stesso linguaggio", non è sufficiente stendere un semplice cavo tra mittente e ricevente ma è necessario utilizzare un apposito terminale, denominato gateway, il quale è impiegato per permettere a due reti diverse di scambiarsi dati. [2] Questi ultimi quindi percorrono il sentiero ed ogni volta passano attraverso queste fermate, fino a raggiungere la destinazione finale; è quindi possibile immaginare che uno di questi gateway "intermedi" sia controllato da un soggetto interessato ad appropriarsi ed utilizzare i nostri dati, soprattutto se riservati, e che sia in grado di non far trasparire la sua presenza in modo da rendere ad entrambi i soggetti comunicanti la sensazione che le proprie informazioni confidenziali stiano transitando in maniera sicura, anche se si sta utilizzando un mezzo totalmente insicuro.

Con il crescente sviluppo di queste tecnologie ed il conseguente aumento degli utilizzatori, il problema della sicurezza dei dati e della trasmissione degli stessi è passato in primo piano, consentendo la nascita di numerosi servizi, dipendenti dalle funzionalità richieste e dall'applicazione in uso.

In questa tesi ci si occuperà quindi di alcune possibili soluzioni per rendere sicura la trasmissione di dati in rete, in particolare verranno presentati lo stato dell'arte per

realizzare connessioni sicure nel Web, come queste pratiche vengono realizzate utilizzando un linguaggio di programmazione mainstream come Java e uno orientato ai servizi come Jolie.

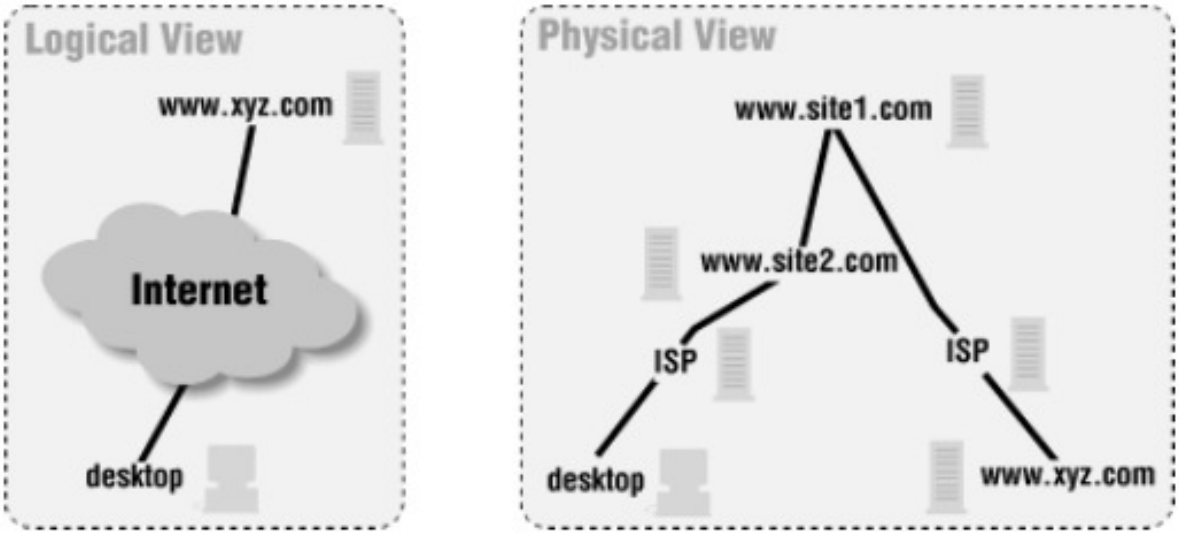

Figura 1.1: Transito dei dati in una rete

1.2 Principali problematiche

Sono sostanzialmente quattro i tipi di minacce che un possibile osservatore malintenzionato può portare allo scambio dei messaggi privati in rete:

- Intercettazione, che altro non è che la lettura del contenuto di un messaggio, possibile sia perché in chiaro sia perché il contenuto è facilmente analizzabile anche se protetto. L'intercettazione può anche essere legale, qualora un magistrato ne decreti l'utilizzo contro organizzazioni sospette;

- Modifica, che riguarda non solo la semplice intercettazione del messaggio, ma anche la modifica dei dati scritti al suo interno. Può essere di tipo criminoso ma anche legale, ad esempio quando si modifica un virus per renderlo inoffensivo;

- Invio sotto falso nome, in questo caso il messaggio viene intercettato e modificato, successivamente, sostituendosi al mittente originale (acquisendone quindi potestà e credenziali), viene recapitato;
- Ripudio della potestà, ovvero quando si nega di aver spedito un messaggio, pur avendolo fatto.

In base ai tipi di minacce elencate si possono ottenere le principali problematiche delle comunicazioni in rete:

- Autenticazione, è la conferma dell'identità dichiarata da un organismo o da un utente. Per molte applicazioni e servizi sono necessarie adeguate procedure di autenticazione per garantire che un messaggio provenga effettivamente dal mittente che ci si aspetta;
- Integrità, è la conferma che sui dati trasmessi e/o ricevuti non sia stata effettuata nessuna opera di modifica o alterazione del codice;
- Non ripudio, riguarda la certezza che il mittente di un messaggio non possa negare in nessun modo di averlo inviato;
- Riservatezza, è la protezione dei dati trasmessi per evitarne l'intercettazione e la lettura da parte di persone non autorizzate. [3]

Capitolo 2

La crittografia

"Esistono due tipi di crittografia: la crittografia che non permette a tua sorella di leggere i tuoi documenti, e la crittografia che non permette ai più grandi governi di non leggere i tuoi documenti. Questo libro si occupa della seconda."

— Bruce Schneier, Applied Cryptography

Il crescente sviluppo di Internet come mezzo di comunicazione per lo scambio di informazioni ha posto l'accento sulla necessità di risolvere problematiche legate a sicurezza, privacy e protezione da sguardi indiscreti.

Si è resa quindi necessaria la creazione di algoritmi e metodi che possano rendere indecifrabili le informazioni, in modo che solamente i soggetti coinvolti nella comunicazione possa leggerle, favorendo così l'integrità e l'autenticazione.

La crittografia, diventata ormai strumento fondamentale per la realizzazione dei meccanismi di sicurezza informatica, nasce con l'obbiettivo di ricercare algoritmi capaci di proteggere le informazioni ad alto valore da qualsiasi tipo di attacco o danneggiamento e comprende tutti gli aspetti relativi alle problematiche generali delle comunicazioni in rete.

Metodi e sistemi crittografici fanno da sempre parte della storia dell'uomo, anche se sono principalmente stati usati per scopi militari; durante la seconda guerra mondiale, ad esempio, alcuni successi degli Alleati, grazie al contributo fondamentale di Alan Turing, sono riconducibili alla scoperta della logica dietro gli algoritmi di crittografia utilizzati dai Tedeschi per nascondere i propri messaggi.

Oggigiorno la crittografia non riguarda più solamente ambiti militari, ma si sta cercando di sfruttarne tutte le potenzialità per costruire una rete sempre più sicura, la quale richiede algoritmi sempre più potenti e sicuri per autenticare gli utenti e proteggere le informazioni. [4]

2.1 Ruolo nella sicurezza

Il ruolo della crittografia nelle applicazioni informatiche è però spesso frainteso e si tende a considerare il suo utilizzo come sinonimo di assoluta garanzia di sicurezza. [5]

È necessario quindi fare una distinzione tra sistemi crittografici forti e deboli. La forza della crittografia viene misurata in base al tempo e alle risorse necessarie per ottenere il messaggio originale. Il risultato di un sistema forte è quindi l'ottenimento di un testo cifrato molto difficile da comprendere; questo aspetto è strettamente legato alla generazione delle chiavi, poiché gli sforzi e le risorse crescono esponenzialmente al crescere della lunghezza di queste ultime.

La difficoltà è però relativa, pur disponendo di una potenza di calcolo molto elevata e di un tempo molto ampio non sarebbe possibile giungere ad una decodifica in tempi brevi, anche se le innovazioni che ci riserva il futuro potrebbero mettere in discussione le metodologie usate al momento. [3]

2.2 Tipologie di crittografia

Un algoritmo crittografico può essere definito come una funzione matematica usata in un processo di criptazione e decriptazione; funziona grazie all'utilizzo di una o più chiavi (solitamente una stringa di numeri), le quali servono per criptare un testo in chiaro. La sicurezza del risultato ottenuto dipende da due fattori: la forza dell'algoritmo e la segretezza della chiave. [3]

Gli algoritmi possono essere divisi in due classi: simmetrici (o a chiave privata) e asimmetrici (o a chiave pubblica)

2.2.1 Crittografia simmetrica

Nella crittografia simmetrica viene generata una sola chiave, denominata chiave privata, la quale viene utilizzata sia dal mittente che dal destinatario per cifrare i rispettivi messaggi, a patto che essa venga mantenuta segreta da entrambi, se non si vuole vedere compromessa la sicurezza.

L'approccio a questa classe di algoritmi consente di cifrare grosse moli di dati molto velocemente, ma soffre di due problemi:

- viene richiesta una chiave separata per ogni coppia di partecipanti allo scambio dei dati, rendendone così complessa la gestione in caso di invio delle informazioni ad un numero elevato di destinatari;

- è necessario determinare un metodo per la condivisione della chiave. Il mantenimento della segretezza è di fondamentale importanza, poiché chiunque la possieda sarà in grado di ottenere il messaggio in chiaro.
  È impensabile inviare la chiave tramite la rete senza crittografare anch'essa, questo comportamento equivarrebbe a trasmettere i propri dati in chiaro, aumentando di conseguenza il rischio che la chiave privata venga intercettata. [6]

Figura 2.1: Esempio di crittografia simmetrica

2.2.2 Crittografia asimmetrica

Con l'introduzione degli algoritmi di crittografia asimmetrica è stato possibile superare i limiti imposti dall'utilizzo di un'unica chiave privata.

Vengono utilizzate due chiavi complementari, denominate pubblica e privata, create in modo che dalla prima sia impossibile ottenere la seconda.

Con l'adozione di questa classe di algoritmi due ipotetici interlocutori A e B possiedono entrambi una coppia di chiavi. A richiede a B la sua chiave pubblica, la quale verrà utilizzata per cifrare il messaggio e spedirlo; quest'ultimo può essere decifrato solo tramite l'utilizzo della chiave privata di B. In questo modo si assicura che solo i due interlocutori, mantenendo segreta la chiave privata e distribuendo a chiunque la chiave pubblica, saranno in grado di interpretare i messaggi inviati e ricevuti.

Anche in questo caso però sorgono delle problematiche, dovute soprattutto alla considerevole lentezza di questi algoritmi rispetto a quelli simmetrici, soprattutto per quanto riguarda grosse quantità di dati. [4]

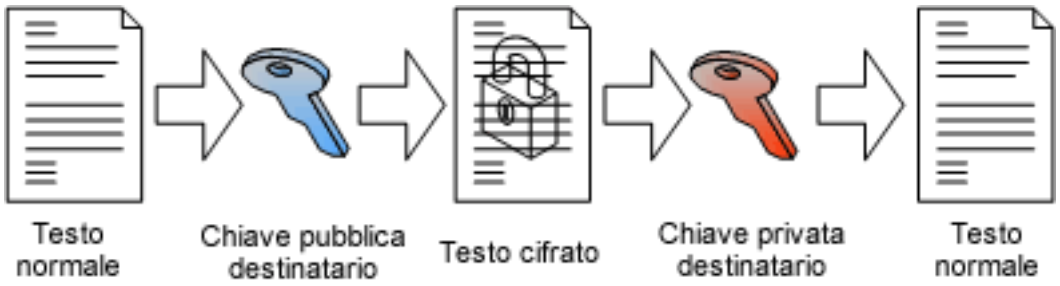

Figura 2.2: Esempio di crittografia asimmetrica

2.2.3 Un approccio ibrido

La particolarità dell'utilizzo di una o dell'altra classe di algoritmi di crittografia descritti in precedenza sta nel fatto che le problematiche di una rappresentino i punti di forza dell'altra e viceversa.

Da questo deriva la preferenza per l'utilizzo di sistemi che le includano entrambe per sfruttarne appieno tutte le potenzialità. Il mittente procede alla generazione di una chiave simmetrica (in questo caso denominata anche chiave di sessione), con la quale cifra un messaggio, e successivamente cifra anche la chiave generata con la chiave pubblica del destinatario, il quale, in seguito, decifra e ottiene, con l'utilizzo della propria chiave privata, la chiave simmetrica, che poi verrà utilizzata per decifrare il messaggio.

Si ottiene così un sistema che risolve la problematica della difficoltà di distribuire la chiave privata, insito nella crittografia simmetrica, e che è in grado di cifrare grosse moli di dati, operazione molto lenta con l'utilizzo della crittografia asimmetrica.

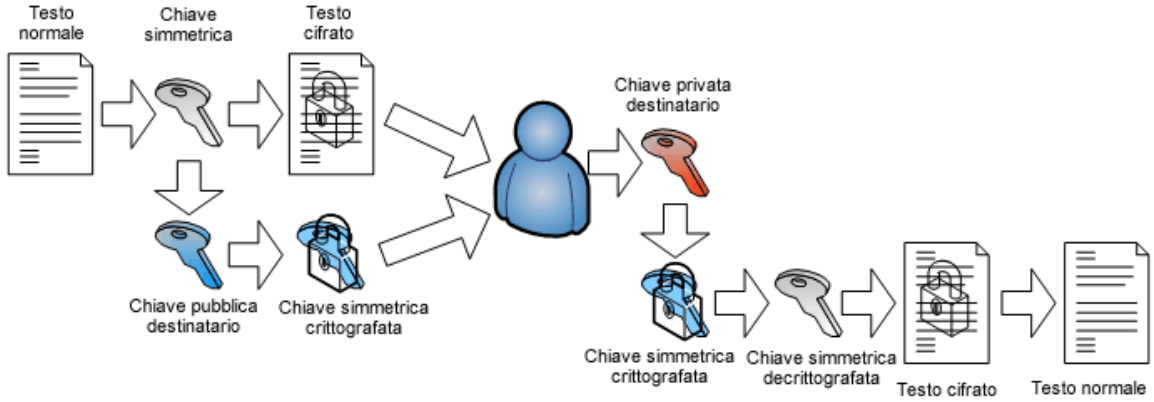

Figura 2.3: Esempio di crittografia ibrida

2.3 Ulteriori strumenti

2.3.1 Certificati

Un certificato è riconducibile ad un documento di identità digitale, contenente un insieme di attributi che identificano il possessore. Viene rilasciato da organismi ufficiali, denominati autorità di certificazione, che garantiscono l'autenticità delle informazioni in esso contenute.

Oltre agli aspetti riguardanti il soggetto certificato, contiene anche la sua chiave pubblica, informazioni sull'autorità di certificazione che lo ha rilasciato, la firma digitale dell'organismo ed il periodo di validità.

Per ottenere la firma da parte di un autorità è necessario compilare una richiesta di certificazione ed attendere la verifica dell'autenticità dei dati dichiarati. In caso positivo, viene prodotto un certificato firmato dall'autorità, il quale contiene la chiave pubblica del richiedente e che può essere utilizzato per autenticarsi con altri individui durante uno scambio di dati.

I certificati digitali assumono quindi un ruolo essenziale nella crittografia asimmetrica, avendo come obiettivo l'autenticazione di un individuo, certificando che la chiave pubblica contenuta in esso, la quale verrà utilizzata per stabilire una comunicazione sicura, appartenga realmente al soggetto per il quale è stato rilasciato. [4]

2.3.2 Firme digitali

Gli algoritmi di firma digitale sfruttano le caratteristiche dell'utilizzo di chiavi pubbliche e private per verificare la reale provenienza del messaggio, ottenendo così l'autenticazione da parte del mittente.

La firma digitale è una stringa ricavata da un messaggio applicando un particolare algoritmo, cifrata mediante la chiave privata del mittente e dopodiché spedita insieme al messaggio. La successiva decifratura della firma tramite la chiave pubblica prova che è stata cifrata dal mittente, o comunque da qualcuno in possesso della sua chiave privata. Inoltre, confrontando la stringa decifrata con una ricavata dal messaggio applicando lo stesso algoritmo, consente di verificarne l'integrità qualora ci fosse corrispondenza.

Riassumendo, il mittente produce un'impronta del messaggio (denominata hash o message-digest) e la cifra con la propria chiave privata, il risultato rappresenta la firma digitale. Il destinatario riceve la firma e il messaggio, ricavando da quest'ultimo l'impronta. Utilizzando la chiave pubblica decifra l'hash del mittente e lo confronta con quello ricavato a partire dal messaggio, se coincidono si ottengono l'autenticazione del mittente e l'integrità del messaggio. [4]

2.3.3 Hashing

Alla base del funzionamento degli algoritmi di firma digitale si trovano gli algoritmi di hashing, i quali producono, a partire da una stringa di lunghezza variabile, una stringa a lunghezza fissa (solitamente tra i 64 e i 255 bit) che è caratteristica di quella originale.

La loro potenza è data dalle seguenti proprietà:

- ottenuto un hash (altrimenti denominata "impronta digitale") è computazionalmente impossibile ricavare il messaggio dal quale è stato generato poiché questi algoritmi funzionano a "senso unico";

- ottenere la stessa stringa di hash da due diversi messaggi risulta impossibile;
- sottoponendo lo stesso messaggio ad un qualsiasi numero di applicazioni dello stesso algoritmo verrà sempre prodotta la stessa impronta.

L'utilizzo di questi algoritmi risulta fondamentale per garantire l'integrità di un messaggio, poiché, conoscendo l'impronta di quello originale, è possibile produrre un hash di quello ricevuto e verificare la corrispondenza di entrambi; nel caso sia positiva si ottiene la sicurezza che non vi siano state modifiche da parte di soggetti non autorizzati. [4]

2.4 Applicazione degli strumenti di crittografia

Attualmente la crittografia è oggetto di grande interesse da parte del mondo scientifico, dovuta soprattutto alla crescente richiesta di aumento della sicurezza nell'utilizzo di strumenti come Internet ed allo sviluppo del commercio elettronico.

Grazie ad essa è stato possibile sviluppare protocolli per rendere più sicura, ad esempio, la navigazione sul web, attraverso l'utilizzo di SSL, il quale verrà introdotto nel prossimo capitolo.

Inoltre, grande successo stanno riscuotendo le tecniche di firma digitale. Il comune di Bologna è stato il primo in Italia ad adottare questa tecnologia per consentire ai cittadini di richiedere il rilascio di documenti ed altri certificati, accedendo ad uno degli sportelli virtuali attraverso l'utilizzo del proprio certificato digitale e della propria chiave privata. Questa coppia, denominata talvolta PSE (Personal Security Environment) rappresenterà la futura carta d'identità. [4]

Capitolo 3

Il protocollo SSL/TLS

Dopo aver fornito una panoramica sulla crittografia, vengono analizzate ora, in particolare, le comunicazioni di rete sicure ed uno degli strumenti più utilizzati per implementarle, il protocollo SSL/TLS, che si basa su alcuni dei concetti espressi in precedenza.

3.1 Descrizione

SSL (Secure Sockets Layer), ed il suo successore TLS (Transport Layer Security), sono i protocolli più diffusi ed utilizzati per garantire una comunicazione sicura dal sorgente al destinatario all'interno di un collegamento in rete; essendo uno l'evoluzione dell'altro per comodità verrà utilizzato il termine SSL come sinonimo di TLS.

Una delle sue caratteristiche principali è l'indipendenza rispetto al protocollo utilizzato, il che significa che può mettere in sicurezza sia transazioni sul web tramite HTTP sia connessioni tramite FTP, POP3 o IMAP. Sostanzialmente agisce come un livello supplementare, permettendo una trasmissione sicura dei dati, interponendosi tra l'applicazione ed il canale trasporto. [7]

Questo protocollo garantisce la sicurezza di milioni di dati ogni giorno ed è applicato a numerose tecnologie, una su tutte il Web, essendo utilizzato, tra le varie possibilità, nell'ambito degli e-commerce in modo da proteggere le transazioni economiche oppure lo scambio di dati confidenziali. [8]

Figura 3.1: Posizionamento intermedio di SSL

Un browser, come ad esempio Mozilla Firefox attraverso l'icona di un lucchetto a fianco dell'URL, segnala ogni qualvolta si stia utilizzando il protocollo SSL, anche se l'utente non ne è a conoscenza poiché tutto avviene in maniera trasparente.

Figura 3.2: Esempio di segnalazione dell'utilizzo di SSL

Gli scopi di SSL sono, in ordine di priorità:

- Sicurezza del collegamento, stabilire una connessione sicura tra due sistemi;
- Interoperabilità, programmatori di diverse organizzazioni dovrebbero essere in grado di sviluppare applicazioni utilizzando SSL, accordandosi sui parametri utilizzati dagli algoritmi di crittografia senza necessità di conoscere il codice l'uno dell'altro;
- Ampliamento, la struttura fornita deve essere in grado di incorporare futuri metodi di crittografia senza dover ricorrere alla creazione di un nuovo protocollo;

- Efficienza, alcune operazioni effettuate da SSL sono molto laboriose per la CPU[1] (es. crittografia asimmetrica), per questa ragione viene incorporato uno schema di memorizzazione di sessione opzionale per ridurre il numero di collegamenti che hanno bisogno di essere stabiliti ex-novo, riducendo così le tempistiche e l'attività sulla rete. [9]

3.2 Perché usare SSL?

Questo protocollo nasce per garantire la sicurezza e la privacy durante le comunicazioni in rete, permettendo alle applicazioni Client/Server di comunicare in modo da prevenire intrusioni, manomissioni e falsificazione dei messaggi.

Tutto ciò viene garantito mediante alcune funzionalità fondamentali:

- Autenticazione, vengono verificate le identità durante la connessione tramite l'utilizzo dei certificati a chiave pubblica per assicurare che la comunicazione stia avvenendo con il server corretto. Il protocollo non determina tipologie di certificati obbligatorie da utilizzare, ma al momento si ricorre esclusivamente allo standard X.509. [10]
  L'autenticazione è obbligatoria da parte del Server, risulta invece opzionale per quanto riguarda il Client;

- Integrità, viene effettuato un controllo sull'integrità del messaggio basandosi sulle proprietà degli hash, ottenuti attraverso l'utilizzo di determinate funzioni sicure (es. SHA, MD5, ecc.), in modo da verificare che i dati spediti tra Client e Server non siano stati intercettati e alterati durante la trasmissione;

- Negoziazione, permette ad un Server ed un Client di accordarsi sugli algoritmi di cifratura e le funzioni hash da utilizzare in base a quelli "noti" a ciascuno dei due;

- Riservatezza, garantita per mezzo della crittografia, che rende illeggibile a terzi il contenuto della comunicazione.

[1] Central Process Unit, comunemente chiamata processore, è l'unità centrale che si occupa di sovrintendere tutte le operazioni eseguite da un computer.

La tipologia utilizzata per il trasporto dei dati è quella a chiave segreta, che viene generata nuovamente ad ogni sessione. La chiave viene calcolata da entrambe le parti ricorrendo ad una serie di numeri casuali ed il *Pre Master Secret*, che vengono scambiati durante la fase di Handshake (step preliminare che permette ad entrambe le parti di autenticarsi e negoziare i parametri della comunicazione prima dell'effettiva trasmissione dei dati), la quale utilizza la crittografia a chiave pubblica, non risentendo quindi del problema della riservatezza nella comunicazione della chiave segreta.
In sintesi, Client e Server si scambiano due numeri casuali e li memorizzano; ottenuta la chiave pubblica del Server, il Client provvederà all'invio del *Pre Master Secret* tramite un messaggio criptato (utilizzando la chiave pubblica ricevuta), il Server decripterà il messaggio utilizzando la sua chiave privata ed entrambi avranno ora la possibilità di calcolare ed utilizzare per le successive comunicazioni la chiave segreta generata;

Fase	**Crittografia utilizzata**	**Dati trasportati**
Handshake	Asimmetrica	Politiche di sicurezza, Pre master secret
Comunicazione	Simmetrica	Dati applicazione

Tabella 3.1: Tipologie di crittografia utilizzate da SSL

- Riutilizzo, permette di memorizzare i parametri utilizzati in precedenza, rendendo così il funzionamento più veloce. [11] [12]

Sebbene SSL fornisca tutte queste funzionalità, non garantisce il non ripudio, che consiste nell'impossibilità da parte di un soggetto di negare l'invio di un messaggio. Solamente quando l'equivalente digitale di una firma è associato al messaggio la comunicazione può essere successivamente dimostrata, ma SSL da solo non ne è in grado. [11]

Soprattutto per quanto riguarda il Web uno degli aspetti fondamentali per un buon online business è la creazione di un ambiente sicuro in cui i potenziali clienti si sentano tutelati nel momento in cui effettuano degli acquisti.

Se il vostro sito raccoglie informazioni sulle carte di credito è tenuto ad avere un certificato SSL valido, oppure se vengono inviate/ricevute informazioni private e confidenziali (indirizzo, numero di telefono, ecc.) è necessario proteggerne il contenuto. [8]

Un riconoscimento dell'importanza da parte di un sito web di fornire sicurezza tramite SSL è stato dato da Google, il quale, nell'Agosto del 2014, ha annunciato che un fattore determinante agli occhi del motore di ricerca sarà composto dall'affidabilità e dalla possibilità di connessione tramite protocollo HTTPS, ovvero il risultato della sovrapposizione di SSL ad HTML. [13]

3.3 Cenni storici

SSL è un protocollo aperto e non proprietario progettato in origine da Netscape per l'uso nei propri browser e server, in risposta alla crescente preoccupazione riguardante la sicurezza in Internet, che stava diventando sempre più popolare.

La versione 1.0 risale al 1994 ma non è mai stata utilizzata in quanto considerata poco affidabile.

Nel 1995 vide la luce la versione 2.0, la quale iniziò ad essere utilizzata in Netscape Navigator 1; successivamente fece da base per la versione 3.0, rilasciata nel 1996, ed implementata all'interno di Netscape Navigator 3.

La diffusione di SSL è stato immediata, facendolo diventare una tecnologia chiave in Internet ed uno standard de facto per la sicurezza delle comunicazioni, portandolo così, nel 1999, al punto di essere analizzato e gestito dall'IETF[2].

Le conseguenze sono state la sua standardizzazione e ridefinizione, che hanno portato alla creazione del protocollo TLS (che altro non è che l'evoluzione di SSL) ed all'uscita della versione 1.0 nel 1999. [12]

Lo sviluppo è continuato fino a portare alla versione 1.1, datata 2006, ed alla versione 1.2, risalente al 2008; le modifiche hanno riguardato soprattutto la risoluzione di

[2]Internet Engineering Task Force, comunità aperta di tecnici, specialisti e ricercatori interessati all'evoluzione tecnica e tecnologica di Internet.

problematiche di vulnerabilità, l'utilizzo di algoritmi di cifratura più sicuri e la creazione di chiavi con un maggior numero di bit.

Per garantire la compatibilità, la comunicazione tra host con diverse versioni del protocollo è consentita, ma il supporto per versioni precedenti è attivo fino ad SSL 3.0, poiché la versione 2.0 è stata considerata troppo vulnerabile e non sicura.

Attualmente è in fase di elaborazione la versione 1.3 di TLS.

3.4 Funzionamento

Una comunicazione che utilizza SSL inizia con uno scambio di informazioni tra il Client ed il Server, denominato Handshake.

In questa prima fase non vengono ancora scambiati i dati veri e propri ma vengono effettuati una serie di passaggi, composti da un insieme di messaggi, per negoziare e stabilire le politiche di sicurezza, gli algoritmi di cifratura e le chiavi da utilizzare durante la connessione. [14]

Una volta terminata con successo questa prima fase è possibile agli interlocutori cominciare a scambiare i propri dati.

3.4.1 La fase di Handshake

Verranno ora elencati nel dettaglio i passi che compongono questa fase preliminare:

- Il Client avvia la comunicazione spedendo un messaggio *Client hello* in cui specifica la più recente versione di SSL supportata, una lista degli algoritmi di cifratura, un identificatore di sessione ed un numero generato casualmente, che verrà utilizzato in seguito per creare la chiave segreta.
  Questo primo step può essere avviato dal Client oppure dal Server, che tramite un messaggio *Hello request* richiede al Client di spedire il messaggio *Client hello*;

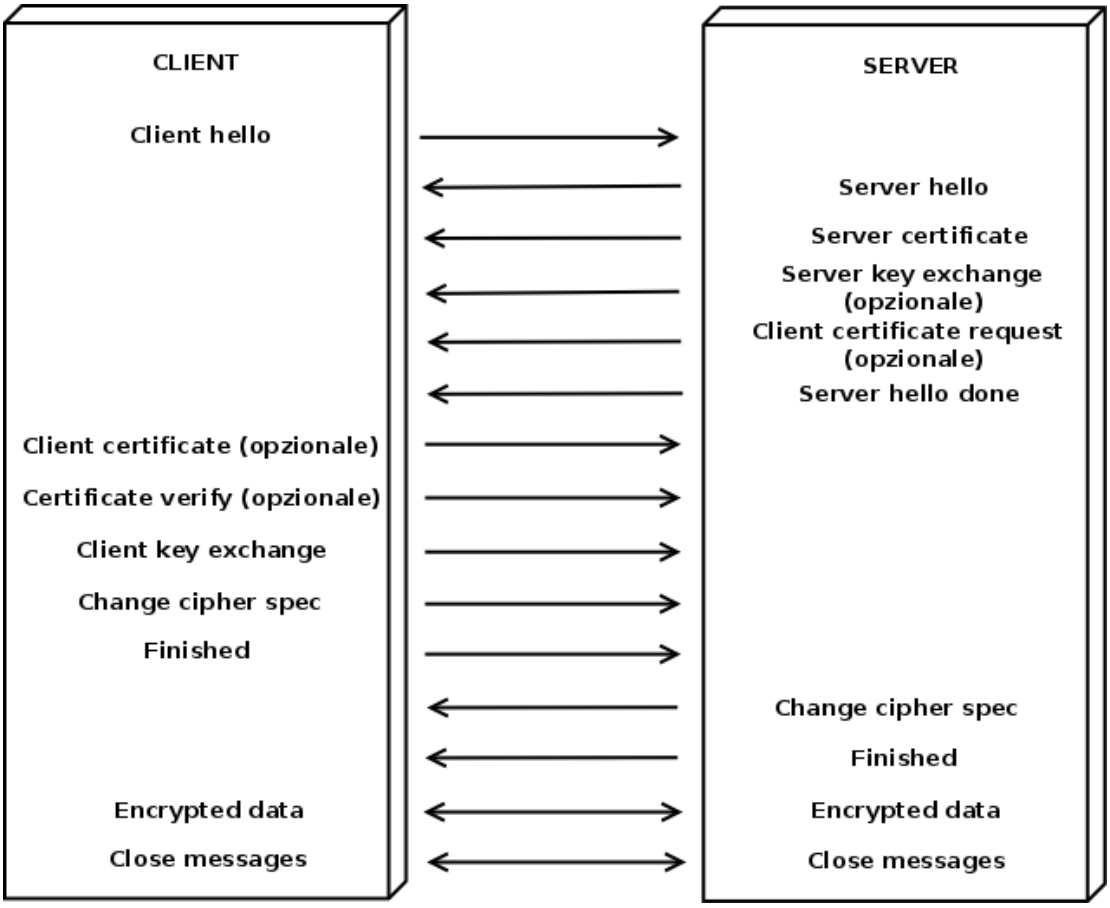

Figura 3.3: Esempio di sequenza di messaggi in SSL

- In risposta il Server invia un messaggio *Server hello* che contiene la versione scelta del protocollo e dell'algoritmo di cifratura, insieme ad un identificatore di sessione ed un numero generato casualmente, utilizzato in seguito per ottenere la chiave segreta;

- Il Server invia il proprio certificato per permetterne il riconoscimento tramite il messaggio *Server certificate*.
  In alcuni casi è possibile che il solo certificato non basti, poiché potrebbe non contenere la chiave pubblica. Si passa quindi attraverso uno step opzionale, nel quale il Server invia un messaggio *Server key exchange* che specifica la chiave pubblica per criptare il successivo messaggio *Client key exchange*;

- Tramite il messaggio *Server hello done*, viene indicato il completamento della negoziazione;

- Il Client verifica il certificato ricevuto, ottenendo così la chiave pubblica del Server, e risponde con un messaggio *Client key exchange*, cifrato e contenente il *Pre Master Secret*.
  Questo step è cruciale per provare l'autenticità del Server, poiché solo quest'ultimo è in grado di decifrare il messaggio (attraverso la coppia chiave pubblica-privata) e proseguire nella negoziazione.
  Il messaggio inviato contiene anche la versione del protocollo scelta, la cui corrispondenza con il valore originale verrà verificata dal Server. Questa misura viene adottata per proteggersi da attacchi di tipo *rollback* che puntano a causare l'utilizzo di una versione meno recente e meno sicura del protocollo;

- Entrambi i soggetti utilizzano i numeri casuali scambiati inizialmente ed il *Pre Master Secret* per generare la chiave segreta, che verrà utilizzata in seguito per criptare le comunicazioni;

- Il messaggio *Change cipher spec* inviato dal Client specifica che ogni successiva comunicazione verrà criptata secondo le politiche e le chiavi definite durante la negoziazione, viene utilizzato anche per rinegoziare l'algoritmo di cifratura da utilizzare;

- Con l'invio del messaggio *Finished* (rappresentato da un hash dell'intera conversazione) il Client si dichiara pronto alla trasmissione dei dati;

- Il Server prova a decifrare il messaggio ed a verificarne l'integrità, qualora fallisse l'Handshake è considerato non concluso e la connessione viene chiusa;

- In caso contrario il Server invia a sua volta un messaggio *Change cipher spec*;

- Come avvenuto da parte del Client viene inviato un messaggio *Finished* per verificare la corretta criptazione e integrità;

- Se la verifica da parte del Client ha successo l'Handshake può dirsi concluso. [14]

3.4.1.1 Mutua autenticazione

In alcuni casi, solitamente poco frequenti, può essere richiesto al Client di autenticarsi, servendosi del proprio certificato. La fase di Handshake si arricchisce quindi dei seguenti passaggi:

- Dopo aver inviato il proprio certificato, il Server invia un messaggio *Certificate request* per richiedere l'autenticazione del Client;
- Tramite un messaggio *Client Certificate* viene recapitato al Server quanto richiesto;
- Inviando un ulteriore messaggio *Certificate verify*, contenente un hash dei precedenti messaggi di Handshake e cifrato con la propria chiave privata, il Client fornisce la possibilità al Server di confermarne l'identità;
- Utilizzando la chiave pubblica estratta dal certificato ricevuto, il Server prova a decifrare il messaggio; in caso di successo si può considerare completata l'autenticazione da parte del Client. [14]

3.4.1.2 Rinegoziazione e riesumazione

Una volta che è stato completato il primo Handshake e i dati vengono scambiati, è possibile per entrambe le parti richiedere di avviare una nuova procedura in qualsiasi momento, chiamata rinegoziazione.

Un'applicazione potrebbe decidere di utilizzare un diverso algoritmo di cifratura rispetto a quello in uso per poter eseguire operazioni particolarmente critiche oppure un Server potrebbe richiedere l'autenticazione da parte del Client. [11]

Il Client ed il Server potrebbero anche decidere di ristabilire una precedente connessione o di duplicarne una esistente invece di negoziare di nuovo tutti i parametri di sicurezza, si parla quindi di riesumazione di sessione.

Viene inviato un messaggio *Client hello* che specifica l'ID della sessione da riprendere; a questo punto il Server controlla in memoria se esiste una corrispondenza:

- se la trova invia un messaggio *Server hello*, contenente lo stesso ID ricevuto e la fase di Handshake può procedere, partendo dall'invio dei messaggi *Change cipher spec* fino al suo completamento;
- in caso contrario ne verrà generato uno nuovo ed entrambe le parti eseguiranno un Handshake completo. [9]

Entrambe queste operazioni permettono di creare una comunicazione basata su parametri già stabiliti in una precedente connessione, aumentando così la possibilità di riutilizzo di SSL.

Capitolo 4

Implementazione di SSL in Java

Dopo aver dato uno sguardo ai fondamenti, alle problematiche ed alle possibili soluzioni per realizzare comunicazioni sicure tramite il protocollo SSL, l'attenzione viene ora focalizzata su un possibile utilizzo di quest'ultimo all'interno del linguaggio di programmazione Java, il quale, oltre a rappresentare uno degli strumenti più diffusi per la realizzazione di applicazioni Client-Server, ha la particolarità di essere indipendente dalla piattaforma sul quale viene eseguito, concedendo così la possibilità di creare software eseguibili su sistemi operativi diversi, a patto che essi posseggano la Java Virtual Machine, utilizzata per l'esecuzione dei suddetti programmi.

Java rappresenta la base con la quale viene gestito ed eseguito il linguaggio di programmazione Jolie, che verrà introdotto nel capitolo successivo.

4.1 Introduzione

All'interno delle librerie di Java è presente il pacchetto *javax.net.ssl* che fornisce l'implementazione del protocollo SSL/TLS.

La creazione di una comunicazione sicura più semplice ed immediata avviene tramite l'utilizzo della classe classe *SSLSocket* [15], poiché bastano alcuni semplici passi per ottenere un buon risultato:

- creare un'istanza della classe *SSLContext* [16], contenente i dati richiesti per la sicurezza;

- utilizzare l'oggetto ottenuto in precedenza per istanziare un *SSLSocketFactory* [17] oppure un *SSLServerSocketFactory* [18], le quali rappresentano un'interfaccia per la creazione dell'oggetto, delegando la definizione dei dettagli e del tipo di oggetto da istanziare ad una classe derivata;

- tramite l'utilizzo delle istanze precedenti, creare un oggetto di tipo *SSLSocket* oppure *SSLServerSocket* [19].

Il risultato ottenuto consente l'utilizzo di questi oggetti nello stesso modo in cui si ricorre a socket con trasporto dei dati in chiaro, senza la necessità di dover apportare modifiche al codice.

La semplicità di implementazione ed utilizzo di questa soluzione possono trarre in inganno, poiché presenta in realtà una serie di problematiche e limitazioni. Supporta il tradizionale I/O basato su stream, il quale è bloccante e costringe gli sviluppatori a mantenere un thread[1] attivo per ogni comunicazione di rete al posto di averne uno solo (o un numero limitato) che abbia la possibilità di gestirne più di una, compromettendo così la scalabilità dei sistemi che la utilizzano. [20] Ad esempio, nei casi in cui un sistema necessiti di dirigere centinaia di connessioni che inviano piccole porzioni di dati, come potrebbe accadere per un server di chat, l'utilizzo di questa soluzione diventerebbe problematico.

Poiché si è interessati ad una soluzione che possa unire nella maniera più efficace possibile sicurezza e scalabilità, anche se *SSLSocket* permette di soddisfare la maggior parte dei casi d'uso che richiedono una connessione affidabile, l'attenzione verrà concentrata sulla classe *SSLEngine* [21], fornita con Java a partire dalla versione 5.

4.1.1 Keystore e truststore

L'estensione di Java riguardante la sicurezza introduce due nozioni molto importanti, keystore e truststore, entrambe rappresentate all'interno di un applicazione da istanze della classe *java.security.KeyStore*. [22]

[1] Sequenza di istruzioni di un processo in corso di esecuzione.

In una conversazione SSL il Server deve disporre di una chiave privata ed un certificato che ne attesti l'identità. La chiave privata è utilizzata dal server nell'algoritmo di scambio ed il certificato è inviato al client per informarlo della propria identità; queste informazioni sono ottenute dal keystore.

Qualora il Server richieda l'autenticazione del Client, anche quest'ultimo deve disporre di un keystore con chiave privata e certificato.

Il truststore è utilizzato dal Client per verificare il certificato inviato dal Server, firmato da un'autorità riconosciuta. Quando il client riceve il certificato deve verificarlo, il che significa che il certificato dell'autorità di certificazione riconosciuta deve trovarsi nel truststore locale; in generale tutti i Client devono avere un truststore.

Nel caso in cui un Server richieda l'autenticazione del Client, anch'esso deve disporre di un truststore.

Riassumendo, i keystore vengono utilizzati per fornire credenziali, mentre i truststore servono per verificare le credenziali. I Server utilizzano i keystore per ottenere i certificati che presentano ai Client, i Client utilizzano i truststore per verificare il certificato, e quindi l'identità, dei Server. [6]

4.2 La classe SSLEngine

4.2.1 Descrizione

Sempre all'interno del pacchetto *javax.net.ssl* è contenuta la classe *SSLEngine*, la quale incapsula una macchina a stati che esegue tutte le operazioni richieste dal protocollo SSL/TLS, quali handshake, criptazione e decriptazione.

La particolarità di questa classe è rappresentata dal fatto che sia indipendente dal livello di trasporto, lasciando allo sviluppatore il compito di gestire il canale di comunicazione, ossia inviare i byte dei messaggi scambiati tra gli end-point della connessione.

Con questa particolare separazione dal livello di trasporto si può ottenere un significativo aumento in termini di scalabilità e flessibilità, in quanto è possibile supportare tutti i metodi di I/O e modelli di threading, sia presenti che futuri. Un esempio è rappresentato

dalla libreria Java NIO (New Input/Output) [23], la quale mette a disposizione numerosi strumenti per costruire applicazioni scalabili e che eseguono operazioni non bloccanti; inoltre, tramite l'utilizzo dei selettori e dei canali, consente di mantenere un singolo thread in grado di gestire diverse connessioni, permettendo così, nel momento in cui si necessita di eseguire operazioni bloccanti all'interno di una determinata connessione, di passare all'esecuzione di operazioni diverse su altre connessioni.

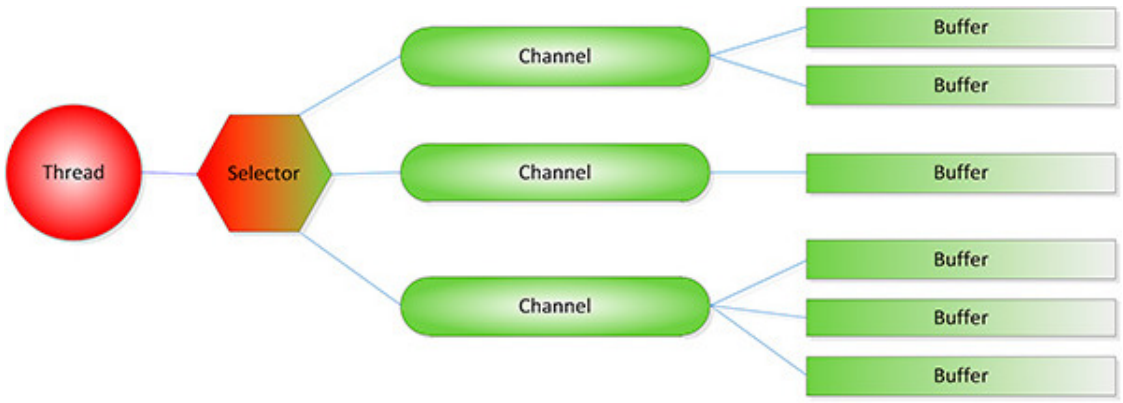

Figura 4.1: Esempio di schema di utilizzo di selettori e canali

Purtroppo per gli sviluppatori questa flessibilità viene pagata con un aumento della complessità; numerosi dettagli e verifiche del protocollo SSL/TLS che nel modello tradizionale venivano nascoste (handshaking, controllo degli stati interni di *SSLEngine*, esecuzione di task delegati, riassemblaggio di pacchetti SSL, gestione dei buffer di memoria) ora sono da gestire.

Proprio a causa di questa sua natura complessa la Oracle ne sconsiglia l'utilizzo da parte dei più novizi alla programmazione in Java, favorendo quindi l'approccio tradizionale basato su *SSLSocket*, a meno che non ne sia strettamente necessario l'utilizzo, come ad esempio nei casi in cui si voglia ottenere scalabilità oppure I/O non bloccante. [20]

Nelle prossime sezioni verranno analizzate alcune delle caratteristiche più importanti di *SSLEngine* e verrà fornito un esempio pratico del suo utilizzo.

4.2.2 Ciclo di vita

All'interno di questa sezione viene descritto e trattato il ciclo di vita di un'istanza di *SSLEngine*.

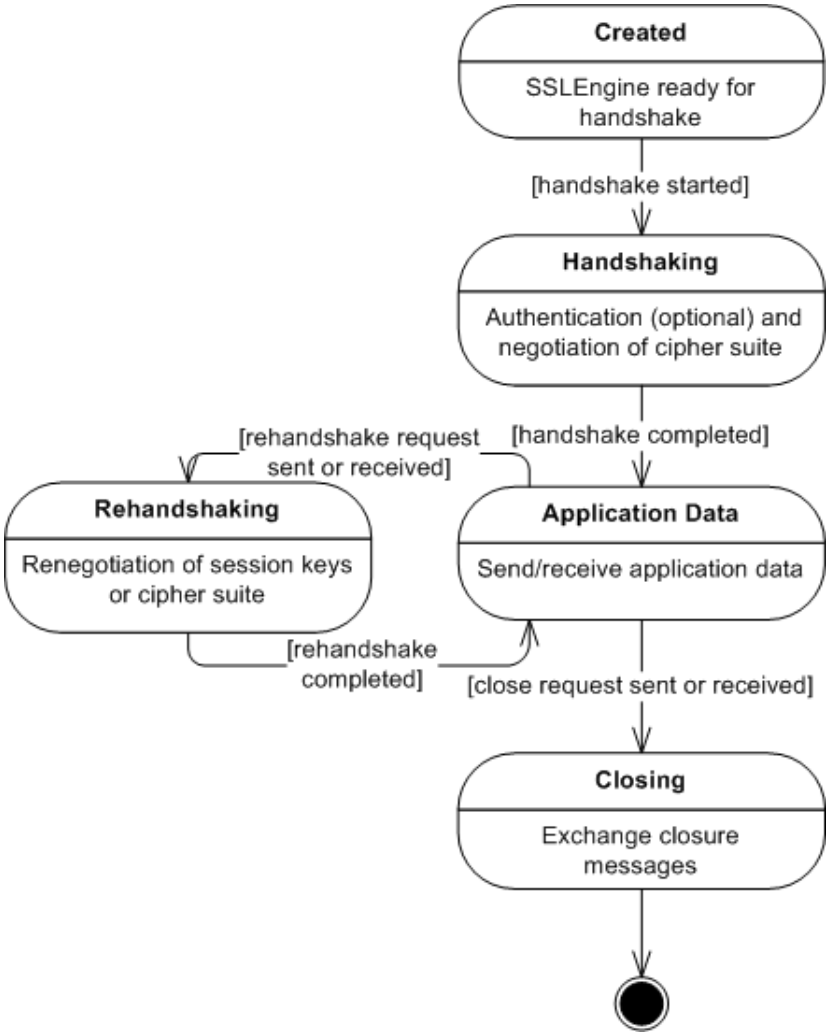

Figura 4.2: Ciclo di vita di un istanza di *SSLEngine*

Creazione Tramite la classe *SSLContext* vengono specificati i parametri per la creazione di un oggetto *SSLEngine*, così come avveniva per la classe *SSLSocket*. Successivamente, tramite il metodo *createSSLEngine()* viene creato l'oggetto desiderato ed è possibile specificarne alcuni parametri, quali le suite di cifratura abilitate e la

modalità Client o Server. È possibile anche utilizzare il metodo *createSSLEngine(peerHost, peerPort)* per specificare un determinato contesto, formato da nome dell'host e la porta utilizzata per la connessione. [16]

Handshaking In questa fase Client e Server stabiliscono le regole comuni da utilizzare per poter comunicare in sicurezza; è caratterizzata dallo scambio di numerosi messaggi e nessun dato dell'applicazione viene ancora inviato. Una volta che l'Handshake è iniziato, ogni nuovo parametro (a parte la specifica della modalità Client/Server) verrà utilizzato a partire dal successivo Handshake.

Dati dell'applicazione Una volta che l'Handshake è stato completato tutti i parametri per garantire la sicurezza sono stati stabiliti e si può quindi procedere allo scambio dei messaggi contenenti i dati; i messaggi in uscita vengono criptati e ne viene protetta l'integrità, l'operazione inversa (decriptazione) viene eseguita per i messaggi in entrata.

Rehandshaking In qualsiasi momento, durante lo scambio dei dati, ciascuna delle due parti può chiedere di eseguire nuovamente un Handshake. I nuovi parametri possono essere mescolati ai dati dell'applicazione, dettaglio non previsto nella fase iniziale. Durante questa fase possono essere modificati solo i parametri riguardanti il protocollo SSL/TLS e non la modalità Client/Server. Così come per la fase iniziale, un qualsiasi cambio della configurazione verrà utilizzato a partire dall'Handshake successivo.

Chiusura Quando la connessione non è più necessaria, l'applicazione dovrebbe chiudere l'*SSLEngine* e inviare/ricevere qualsiasi messaggio rimanente prima di chiudere il meccanismo di trasporto sottostante. Una volta che è stata effettuata la chiusura non è più possibile utilizzare lo stesso oggetto, è necessario eseguire una nuova creazione.

4.2.3 Inizializzazione

Nella sezione precedente è stato dato uno sguardo generale al ciclo di vita di un oggetto di tipo *SSLEngine*.

L'attenzione viene ora focalizzata sulla fase di inizializzazione di *SSLContext* e la conseguente creazione dell'oggetto desiderato.

```
import javax.net.ssl.*;
import java.security.*;

//Conversione della password in array di caratteri
char[] passphrase = "passphrase".toCharArray();

//Inizializzazione di chiave e certificati
KeyStore ksKeys = KeyStore.getInstance("JKS");
ksKeys.load(new FileInputStream("testKeys"), passphrase);
KeyStore ksTrust = KeyStore.getInstance("JKS");
ksTrust.load(new FileInputStream("testTrust"), passphrase);

//Creazione gestore di chiavi
KeyManagerFactory kmf = KeyManagerFactory.getInstance("SunX509");
kmf.init(ksKeys, passphrase);

//Creazione gestore di certificati
TrustManagerFactory tmf = TrustManagerFactory.getInstance("SunX509");
tmf.init(ksTrust);

//Dichiarazione del protocollo da utilizzare
sslContext = SSLContext.getInstance("TLS");
//Inizializzazione del contesto
sslContext.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);

//Creazione dell'oggetto SSLEngine
SSLEngine engine = sslContext.createSSLEngine();

//Utilizzo della modalità Client
engine.setUseClientMode(true);
```

Figura 4.3: Esempio di creazione di un oggetto *SSLEngine*

Per poter inizializzare il contesto con cui creare l'engine è necessario prima il settaggio di alcuni parametri:

- creazione di keystore e truststore, il primo utilizzato per fornire credenziali, il secondo per verificarle.
  Per poterne accedere al contenuto entrambi necessitano della dichiarazione di una password, formata da un array di caratteri;

- inizializzazione, tramite i due oggetti creati precedentemente, di un gestore di chiavi ed un gestore di certificati, i quali seguiranno lo standard X.509;
- dichiarazione della versione del protocollo SSL/TLS da utilizzare.

Una volta specificati questi parametri è possibile inizializzare *SSLContext*, fornendo come argomenti i gestori creati precedentemente. È anche possibile specificare un oggetto *java.security.SecureRandom*, che è un generatore di numeri casuali molto più affidabile del classico *java.utili.Random*, utilizzato soprattutto per scopi di criptazione e sicurezza.

L'inizializzazione appena eseguita fornirà la base per la creazione dell'oggetto *SSLEngine*, al quale verrà specificato se assumere un comportamento da Client o da Server.

4.2.4 Interazione con l'applicazione

I due metodi principali che costituiscono *SSLEngine* sono *wrap()* ed *unwrap()*, utilizzati rispettivamente per la criptazione e la decriptazione dei dati.

Necessitano della specifica di 2 buffer di byte come argomento, i quali assumono due diversi significati a seconda del metodo che si utilizza:

- *wrap()* riceve dall'applicazione il buffer dei dati da criptare (*src*) e produce il buffer da inviare sul canale di trasporto (*dst*);
- *unwrap()* al contrario riceve i dati dal canale di trasporto (*src*) e li decripta per fornirli all'applicazione (*dst*).

Qualora non fosse ancora stato eseguito l'Handshake iniziale, entrambi i metodi provvederanno al suo completamento prima di poter trasferire i dati.

L'applicazione inserisce i dati da inviare all'interno di un Application buffer, il quale viene inviato all'oggetto *SSLEngine*, incaricato di criptare i dati tramite metodo *wrap()* e renderli disponibili all'interno di un Network buffer; a questo punto i dati sono pronti ad essere inviati all'altro end-point della connessione, tenendo sempre in considerazione che la logica del trasporto è a carico dell'applicazione.

In seguito, i dati ricevuti tramite il Network buffer vengono decriptati dal metodo *unwrap()* e vengono scritti all'interno dell'Application buffer.

Durante la fase di Handshake è l'oggetto *SSLEngine* stesso ad occuparsi dei dati da inviare/ricevere per stabilire la connessione sicura. [11]

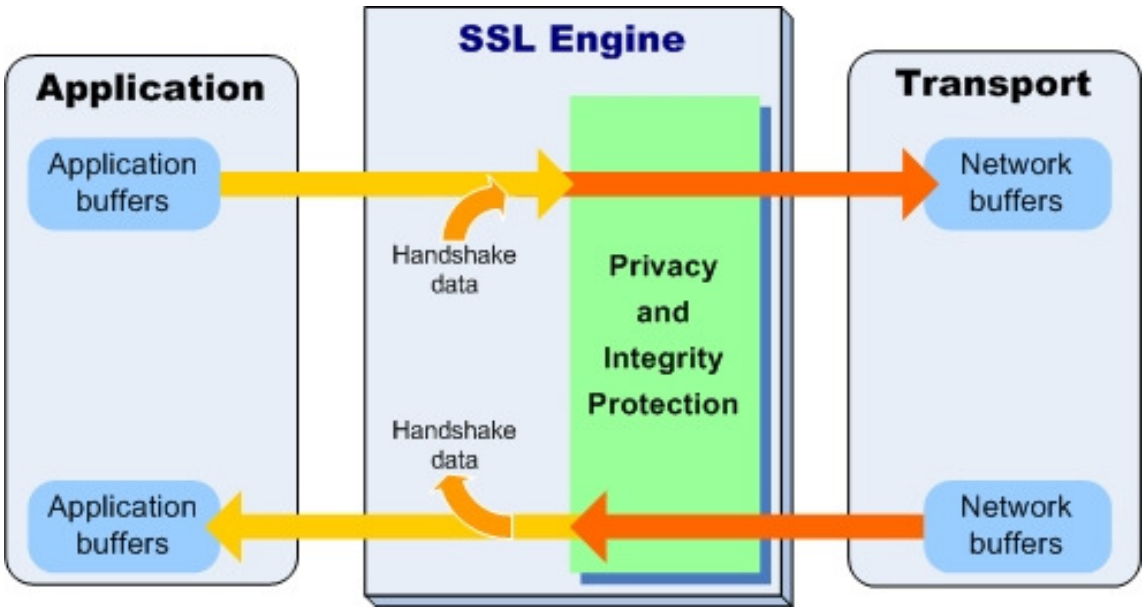

Figura 4.4: Fasi dell'interazione

Il controllo del flusso è mantenuto dall'engine, il quale, tramite un oggetto *SSLEngineResult* restituito dai metodi *wrap()* e *unwrap()*, segnala i suoi avanzamenti tramite un set di stati. Compito dell'applicazione è esaminarli ad ogni interazione.

Dall'oggetto ricevuto in risposta è possibile estrarre alcuni valori:

- bytesConsumed, numero di bytes letti da *src*;
- bytesProduced, numero di bytes scritti in *dst*;
- una costante che indica il risultato dei metodi *wrap()*/*unwrap()*;
- una costante che indica lo stato dell'Handshake. [24]

Verranno ora analizzati i diversi valori che assumono queste costanti.

4.2.4.1 SSLEngineResult.HandshakeStatus

Durante la fase di Handshake, l'engine si occupa dello scambio dei messaggi e ne controlla i risultati tramite cinque possibili stati:

- FINISHED, indica che l'ultima operazione eseguita ha determinato la terminazione della fase di Handshake;
- NEED_TASK, prima di continuare l'Handshake l'engine deve effettuare un'operazione bloccante o che richiede molto tempo.
  Per la sua caratteristica non bloccante, la quale richiede che questo tipo di operazioni non vengano eseguite nello stesso thread che si occupa delle richieste I/O (poiché potrebbero bloccarne tutte le connessioni), l'operazione viene delegata ad un thread esterno, invocando il metodo *getDelegatedTask()* per ottenere l'istanza di un oggetto *Runnable*, incapsularne il task da eseguire ed avviarlo separatamente. Una volta terminato, l'applicazione controlla nuovamente lo stato dell'Handshake per conoscere il passo successivo da effettuare.
  Le operazioni di questo tipo riguardano di solito la richiesta di password all'utente, la validazione di un certificato remoto oppure la generazione di chiavi di sessione;
- NEED_UNWRAP, bisogna attendere che dal canale di trasporto arrivino altri dati da decriptare per poter proseguire;
- NEED_WRAP, è necessario criptare dei dati da inviare successivamente tramite il canale di trasporto per poter proseguire;
- NOT_HANDSHAKING, specifica che non è in esecuzione la fase di Handshake.

4.2.4.2 SSLEngineResult.Status

Oltre allo stato dell'Handshake è necessario anche analizzare l'esecuzione delle operazioni *wrap()* ed *unwrap()*, che possono assumere uno dei quattro seguenti stati:

- BUFFER_OVERFLOW, non è presente abbastanza spazio all'interno del buffer *dst* per contenere tutti i dati che verrebbero generati. È necessario che l'applicazione provveda a liberare spazio oppure ad aumentare la dimensione del buffer;

- BUFFER_UNDERFLOW, non ci sono abbastanza dati all'interno del buffer in entrata per eseguire l'operazione, l'applicazione dovrebbe leggere più dati dalla rete. Il protocollo SSL/TLS è di tipo *packet-based* ed il metodo *unwrap()* può operare solo con pacchetti completi, quindi se il buffer in entrata non contiene un pacchetto completo verrà restituito il seguente risultato.
  Questo tipo di stato viene generato esclusivamente dalla chiamata al suddetto metodo, poichè, al contrario, attraverso *wrap()* l'engine è in grado di creare pacchetti SSL/TLS con qualsiasi dato sia disponibile;
- CLOSED, l'engine è stato chiuso e non è più possibile utilizzarne l'istanza;
- OK, l'operazione è stata portata a termine con successo. [20]

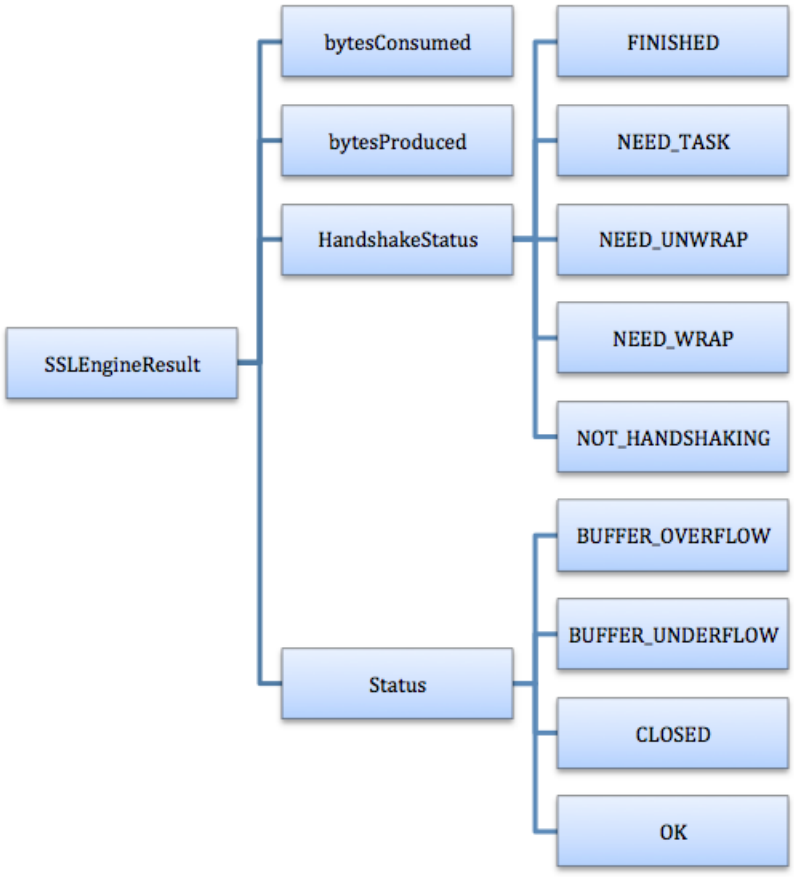

Figura 4.5: Contenuto di un oggetto *SSLEngineResult*

Capitolo 5

Da Java a Jolie: utilizzo concreto di SSL

5.1 Breve panoramica su Jolie

Jolie è un linguaggio di programmazione che permette di realizzare architetture orientate ai servizi e di gestirle in maniera specifica. È un progetto open source sviluppato all'interno dell'Università di Bologna ed è liberamente consultabile ed utilizzabile. [25]

Alla base del suo funzionamento vi è il linguaggio Java, utilizzato per definire tutte le librerie e le operazioni potenzialmente eseguibili tramite Jolie; la sintassi quindi risulta molto simile al suddetto linguaggio ed al C.

Tramite l'utilizzo di questo linguaggio la progettazione e la creazione di un servizio avvengono ad alto livello, permettendo così una più semplice gestione di tutte le problematiche che riguardano le comunicazioni; è possibile in questo modo realizzare semplici servizi elementari scrivendo poche righe di codice, ma sono presenti anche numerosi strumenti per la realizzazione di progetti più complessi.

I costrutti forniti riguardano la chiamata e la ricezione di operazioni e messaggi, la definizione di porte di ingresso e uscita, l'identificazione di sessioni, la definizione di variabili strutturate, operazioni, procedure ed altri meccanismi utili per la gestione di servizi. Tra le varie opportunità, è anche possibile l'esecuzione di codice Java esterno all'interno del proprio applicativo Jolie.

Basandosi su Java, Jolie ne eredita le principali proprietà, come ad esempio la possibilità di esecuzione su differenti piattaforme, purché presentino la Java Virtual Machine, e la flessibilità, supportando l'introduzione di nuovi protocolli, media di comunicazione e operazioni tramite lo sviluppo di librerie esterne, scritte utilizzando Java, e la loro successiva inclusione all'interno del linguaggio.

5.1.1 Behaviour e Deployment

Un servizio realizzato in Jolie è composto fondamentalmente da due parti, rappresentate dalle informazioni per la realizzazione della comunicazione (deployment) e dalla definizione delle funzionalità offerte dal servizio (behaviour). [26]

All'interno della parte di deployment viene definito il supporto alla comunicazione, rappresentato dalle porte di input e di output, le quali sono composte da una sintassi simile e si basano su tre concetti fondamentali:

```
inputPort In {
        Location: "socket://localhost:80/"
        Protocol: sodep
        Interfaces: InterfacciaEsempio
}
```

- location, che esprime il medium di comunicazione che un servizio utilizza per contattarne un altro. È rappresentata da un URI e supporta socket, btl2cap, rmi e localsocket;
- protocol, che definisce come i dati da spedire devono essere codificati (output) o decodificati (input). Sono supportati i protocolli HTTP, HTTPS, JSON/RPC, XML/RPC, SOAP, SODEP e SODEPS, questi ultimi 2 rappresentano protocolli nativi creati allo scopo di fornire un trasferimento efficiente dei dati nelle comunicazioni tra servizi Jolie. Ogni protocollo possiede i propri parametri di configurazione, in modo da essere adattato alle richieste di una specifica comunicazione.

Come si può notare dalla lista dei protocolli supportati è presente anche l'implementazione di SSL, utilizzata per quanto riguarda HTTPS e SODEPS. È possibile quindi impostare una comunicazione sicura semplicemente specificando uno di questi due protocolli all'interno della porta;

- interface, che specifica i tipi di dato e le operazioni che ci si aspetta vengano utilizzati dai servizi che usano la porta in questione. In alternativa, è possibile specificare solamente le operazioni supportate al posto dell'interfaccia.

La porta di input espone le operazioni che possono essere richiamate dagli altri servizi, mentre la porta di output definisce come invocare le operazioni di altri servizi. [27]

La parte di behaviour definisce l'implementazione delle funzionalità offerte da un servizio, le quali, in base al compito, si dividono in funzionali, gestione delle eccezioni e gestione della comunicazione.

Un servizio è un applicazione eseguita su una macchina e, tramite le porte di input e output, realizza una comunicazione attraverso lo scambio di messaggi. Queste operazioni vengono richiamate all'interno del behaviour e possono essere di due tipi:

- One-Way, definisce operazioni che restano in attesa di un messaggio da parte di un altro servizio, senza la necessità di dover rispondere;
- Request-Response, definisce operazioni che restano in attesa di un messaggio da parte di un altro servizio, prevedendo anche un messaggio di risposta. [26][27]

Con questa separazione il comportamento di un servizio Jolie risulta indipendente sia dal protocollo che dal mezzo di comunicazione, consentendo la creazione di funzionalità che potranno essere riutilizzate anche nei casi in cui si renda necessario il cambiamento della tipologia di comunicazione, basterà effettuare gli opportuni aggiustamenti alla parte di deployment.

Inoltre, la definizione delle porte di input ed output risulta molto semplice ed intuitiva, sollevando lo sviluppatore da onerose perdite di tempo per quanto riguarda la gestione e l'impostazione dei protocolli e dei mezzi di comunicazione, compito che verrà svolto dalle librerie che compongono la base del linguaggio Jolie.

5.1.2 L'interprete Jolie

Jolie pone le sue basi sul linguaggio Java ed è implementato tramite un interprete, scritto utilizzando proprio quest'ultimo. Il codice sorgente viene analizzato e trasformato in oggetti che implementano la semantica desiderata. Questi oggetti sono organizzati all'interno di un albero, chiamato OOIT (Object-Oriented Interpretation Tree), la cui esecuzione è supportata da un Runtime Environment. Un componente separato, chiamato Communication Core, viene utilizzato per gestire ed effettuare le comunicazioni.

L'architettura dell'interprete risulta quindi essere composta da quattro principali componenti:

- Runtime Environment, responsabile per l'istanziazione dei componenti e del supporto all'esecuzione dell'albero OOIT;
- Parser, legge il servizio ricevuto in input e genera l'OOIT;
- Object-Oriented Interpretation Tree, un albero di oggetti che implementa l'esecuzione delle regole semantiche relative al servizio;
- Communication Core, gestisce le comunicazioni, permettendo agli altri componenti di trattare messaggi di input e output astraendo dai sottostanti protocolli e mezzo di comunicazione.

All'avvio di un servizio quindi, vengono eseguiti i seguenti step:

- lettura degli argomenti da linea di comando e successivo avvio del Parser;
- analisi del codice sorgente da parte del Parser ed inizializzazione dell'OOIT;
- creazione del Runtime Environment e del Communication Core;
- invocazione da parte del Runtime Environment del metodo run contenuto all'interno del nodo radice dell'OOIT (corrispondente all'istruzione main di un servizio, rappresentante il punto di ingresso per l'esecuzione di qualsiasi servizio Jolie). [28]

5.1.3 Gestione dei protocolli di comunicazione ed SSL

Come affermato in precedenza, la gestione delle comunicazioni tra i servizi è affidata al componente denominato Communication Core, permettendo alle funzionalità di risultare indipendenti da protocolli e mezzi di comunicazione sottostanti.

Questa caratteristica è supportata tramite l'utilizzo delle classi astratte *CommChannel* e *CommProtocol*, le quali rappresentano rispettivamente il mezzo di comunicazione ed il protocollo definito all'interno della porta.

Ogni medium di comunicazione supportato è rappresentato da una sottoclasse di *CommChannel*, che ne consente la gestione.

Lo stesso concetto è applicato ai protocolli supportati ed alla classe *CommProtocol*; in aggiunta, all'interno è contenuto un oggetto *CommMessage*, utilizzato come formato standard per i messaggi da inviare e ricevere tramite le comunicazioni in Jolie.

Una volta avviato un servizio ed interpretato il codice sorgente, il Runtime Environment si occupa della creazione di un oggetto derivante da *CommChannel*, corrispondente al mezzo di comunicazione, e ad esso associa ad un oggetto derivante da *CommProtocol*, corrispondente al protocollo da utilizzare; tutte le precedenti creazioni tengono conto di parametri di connessione specificati dalle istruzioni contenute all'interno della definizione delle porte.

Quando risulta necessario inviare un messaggio, i dati vengono inseriti all'interno di un oggetto *CommMessage*, indipendente da protocollo e mezzo di comunicazione, ed inviati all'istanza della sottoclasse di *CommChannel*. Successivamente, utilizzando l'istanza contenuta in essa, il messaggio viene codificato secondo il protocollo di comunicazione scelto ed inviato, tramite il mezzo di comunicazione scelto; il processo inverso viene applicato in fase di ricezione.

Questo tipo di progettazione aumenta la modularità e l'estendibilità, sia dei medium che dei protocolli supportati. L'aggiunta di ulteriori opzioni risulta quindi legata all'implementazione di sottoclassi di *CommChannel* oppure *CommProtocol*.

Basandosi sulle logiche di SSL e la particolare progettazione applicata alle comunicazioni da parte di Jolie, è possibile creare una classe che gestisca l'invio sicuro del

messaggio, codificato secondo il protocollo scelto.

La classe incaricata di tale compito risulta essere *SSLProtocol*, la quale incapsula al suo interno un oggetto che rappresenta il protocollo scelto, interponendosi così tra il Runtime Environment ed il mezzo di comunicazione.

In questo modo, un messaggio viene inizialmente codificato secondo le regole del protocollo e successivamente, tramite *SSLProtocol* e l'oggetto *SSLEngine* in essa contenuto, vengono applicate ad esso le politiche di sicurezza dettate da SSL, prima di poterlo inviare tramite il mezzo di comunicazione. In fase di ricezione avviene il processo inverso, ossia al messaggio vengono prima rimosse le politiche SSL e poi applicata la decodifica secondo il protocollo.

Questa soluzione gode di una notevole flessibilità, in quanto *SSLProtocol* si limita all'applicazione delle politiche SSL, delegando la gestione del messaggio risultante al protocollo. Risulta quindi possibile realizzare, senza particolari sforzi, versioni sicure dei protocolli, sia esistenti che di futura implementazione, messi a disposizione da Jolie ed applicare a quelli esistenti modifiche, senza che esse influiscano sull'utilizzo di *SSLProtocol*.

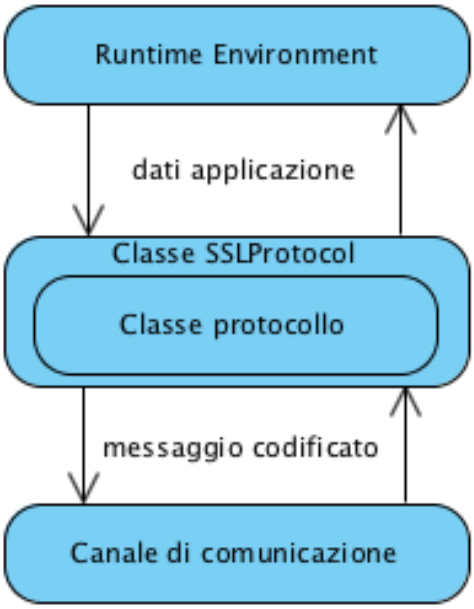

Figura 5.1: Architettura utilizzata per comunicazioni sicure tramite SSL

5.2 Analisi della classe SSLProtocol

SSLProtocol è la classe che si occupa della gestione di SSL, indipendentemente dal protocollo di trasporto scelto. Incorpora al suo interno l'esecuzione della fase di Handshake, la criptazione/decriptazione dei dati e del loro invio sul canale; viene istanziata dall'ambiente Jolie nei casi in cui venga richiesta una comunicazione sicura. Tramite l'utilizzo di un oggetto *SSLEngine*, opportunamente configurato secondo i parametri stabiliti, applica le logiche del protocollo SSL.

Interagisce con gli altri componenti dell'architettura attraverso la dichiarazione di un oggetto che descrive il protocollo da utilizzare (es. HTTP) e l'implementazione dei metodi *send()* e *recv()*, i quali gestiscono la comunicazione con il canale di trasporto:

- *send()* viene invocato quando è richiesto l'invio di un messaggio. Riceve i dati criptati e li incapsula in modo da poterli inviare nella maniera corretta utilizzando il tipo di trasporto scelto.
  Prende come parametri di input il canale di destinazione, il messaggio da inviare ed il canale di ricezione;

```
void send(OutputStream, CommMessage, InputStream)
{
...
}
```

- *recv()* viene invocato quando, tramite il canale di comunicazione, viene ricevuto un messaggio, il quale viene riportato al suo stato originario tramite la decriptazione.
  Prende come parametri in input il canale di ricezione e quello di destinazione.
  Restituisce un messaggio contenente i dati applicazione.

```
CommMessage recv(InputStream, OutputStream)
{
...
}
```

All'interno vengono definite alcune classi ausiliarie:

- *SSLInputStream* ed *SSLOuptutStream*, le quali implementano i metodi per la lettura e la scrittura all'interno dei buffer che verranno condivisi con il canale di comunicazione, oltre a gestire alcuni casi particolari;

- *SSLResult*, incaricata dell'aumento della dimensione del buffer e della memorizzazione dei risultati di SSLEngine

5.2.1 Descrizione del codice

Dopo un'analisi generale della classe, si passa ora alla descrizione del codice di esecuzione di *SSLProtocol*.

Una volta creato, un oggetto SSLProtocol si pone in attesa di una chiamata per scrivere o leggere dati, in base all'invocazione di *send()* oppure *recv()*.

La chiamata ad uno dei due metodi attiva la verifica dell'esecuzione della fase di Handshake, tramite due tipi di controllo: il primo viene effettuato leggendo il valore contenuto all'interno dell'HandshakeStatus, il secondo verificando che sia stata creata un'istanza di SSLEngine.

Se è stato invocato *send()*, al completamento dell'Handshake il messaggio viene criptato attraverso il metodo *wrap()* e successivamente inviato al canale di destinazione.

Al contrario, se viene invocato *recv()*, al completamento dell'Handhsake il messaggio viene decriptato: se contiene un messaggio di chiusura della connessione viene aggiornato lo stato di *SSLEngine* a CLOSED, altrimenti i dati vengono letti e passati all'applicazione.

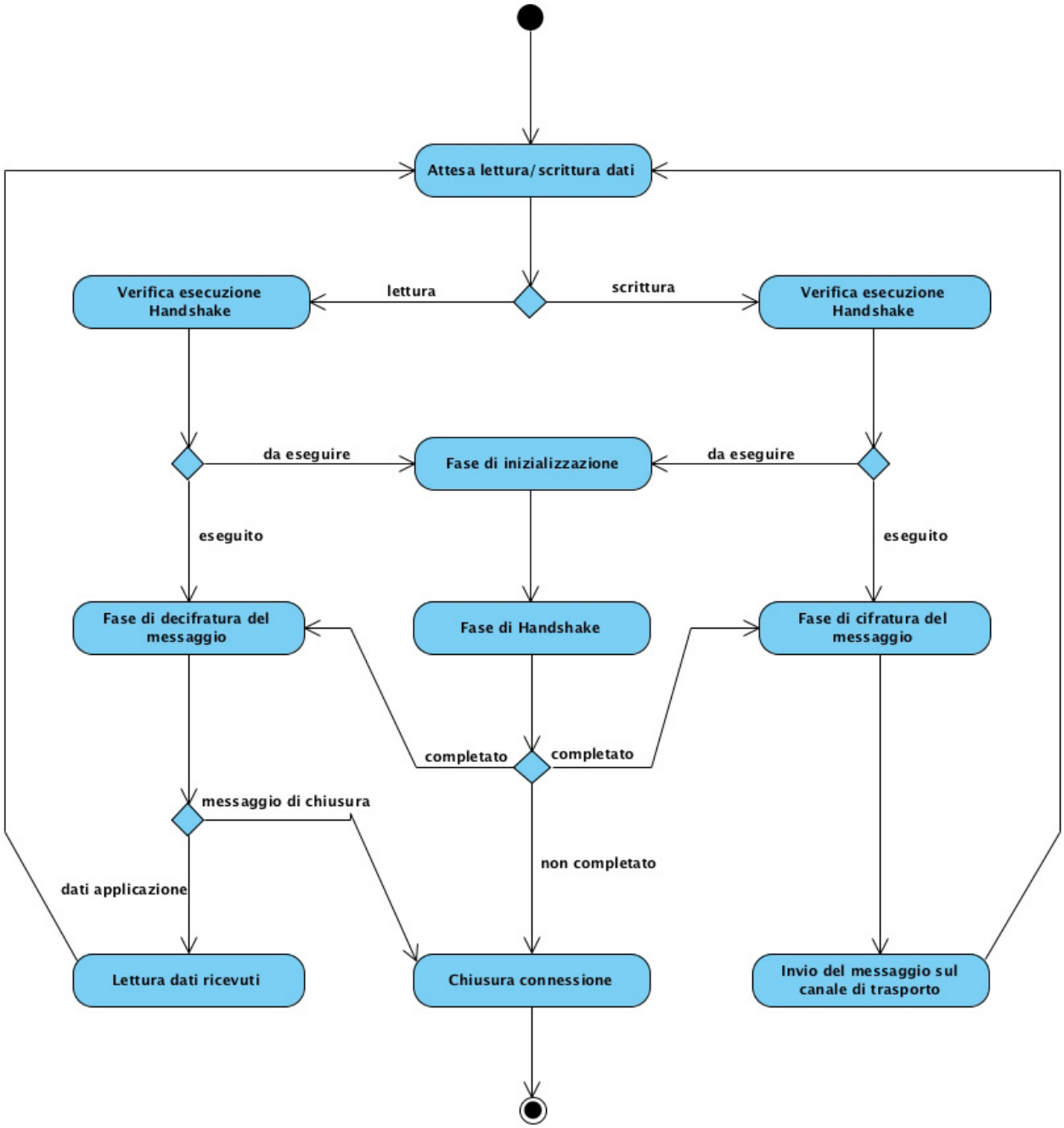

Figura 5.2: Diagramma di flusso del codice di *SSLProtocol*

Vengono ora descritte alcune delle fasi principali presenti in figura 5.2.

5.2.1.1 Fase di inizializzazione

SSLEngine può supportare una serie di parametri di configurazione riguardanti il protocollo SSL/TLS. Questi parametri possono essere specificati nel codice Jolie del servizio che si sta costruendo. Una lista di tali parametri verrà illustrata successivamente.

Questa fase, gestita tramite il metodo *init()*, riceve la configurazione specificata e la utilizza per creare un oggetto *SSLContext*, il quale successivamente rappresenterà la base per la creazione dell'istanza di *SSLEngine*; l'inizializzazione viene eseguita solo se necessaria, ovvero se non è ancora stato creato nessun oggetto *SSLEngine*.

Questo procedimento viene avviato nel momento in cui si richiede la scrittura o la lettura di dati dal canale di trasporto, verificando prima se è necessario effettuare un nuovo Handshake (anche tramite il controllo dell'esistenza di un'istanza di SSLEngine).

La modalità di esecuzione (Client/Server), obbligatoria ed impossibile da cambiare durante il ciclo di vita dell'istanza, viene impostata in base ad una variabile utilizzata dai protocolli di comunicazione superiori per inizializzare l'oggetto *SSLProtocol*.

5.2.1.2 Fase di Handshake

Questa fase avvia il processo di Handshake tra i due terminali della comunicazione, controllando ciclicamente l'*HandshakeStatus* ed, in base al risultato restituito, effettuando le operazioni adeguate per il corretto proseguimento. Il ciclo termina quando lo stato risultante corrisponde a FINISHED oppure NOT_HANDSHAKING.

5.2.1.3 Fase di cifratura

Durante questa fase il messaggio da inviare viene passato all'oggetto *SSLEngine*, che si occupa della cifratura, in modo da prepararlo per il successivo invio all'altro end-point.

Viene invocata ogni volta che si necessita di inviare dati oppure quando lo stato dell'Handshake risulta impostato su NEED_WRAP, che specifica il bisogno di ulteriori dati da inviare.

Il metodo *wrap()* può generare lo stato di BUFFER_OVERFLOW nei casi in cui il buffer di destinazione dove deve essere scritto il messaggio risultante dal metodo non è grande a sufficienza per contenerlo; al verificarsi di questo caso, al buffer viene aumentata la capacità e si ritenta l'operazione.

Viene gestito anche lo stato di CLOSED, il quale specifica che l'istanza di *SSLEngine* è stata chiusa e che quindi non può più essere utilizzata per la comunicazione.

5.2.1.4 Fase di decifratura

Durante questa fase il messaggio ricevuto viene passato al metodo *unwrap()*, il quale si occupa di riportarlo al suo stato originale in modo da poterlo analizzare.

Viene invocata ogni volta che si necessita di ricevere dati oppure quando lo stato dell'Handshake risulta impostato su NEED_UNWRAP, che specifica l'attesa di dati dall'altro end-point.

Il metodo *unwrap()* può generare lo stato di BUFFER_OVERFLOW, che viene gestito allo stesso modo di *wrap()*.
In aggiunta, può anche generare lo stato di BUFFER_UNDERFLOW, che segnala la necessità di attendere ulteriori dati poiché quelli ricevuti non sono sufficienti a completare l'operazione (si ricorda che il protocollo SSL in ricezione opera solo con pacchetti completi). Viene pertanto effettuata un'ulteriore lettura sul canale, unendo i dati ricevuti ai precedenti, e ritentando l'operazione.

Viene gestito anche lo stato di CLOSED, il quale specifica che l'istanza di *SSLEngine* è stata chiusa e che quindi non può più essere utilizzata per la comunicazione.

5.2.2 Gestione degli stati di SSLEngine

Durante la comunicazione l'oggetto *SSLEngine* può assumere una serie finita di stati interni, ai quali è associato un determinato comportamento da parte di *SSLProtocol*, che possono essere analizzati tramite due indicatori:

- HandshakeStatus, indica la prossima operazione da eseguire per proseguire l'Handshake, se assume un valore diverso da FINISHED oppure NOT_HANDSHAKING

viene avviato il metodo per il monitoraggio degli stati. Può assumere i seguenti valori:

  - FINISHED, determina la fine della fase di Handshake;
  - NEED_TASK, viene analizzato il task da eseguire facendo ricorso al metodo *getDelegatedTask()*. Successivamente viene avviato e si attende il suo completamento;
  - NEED_UNWRAP, ci si rimette in ascolto sul canale per attendere l'arrivo di ulteriori dati e proseguire;
  - NEED_WRAP, determina la cifratura di un messaggio che, se la variabile bytesProduced assume un valore maggiore di zero, viene inviato all'altro endpoint;
  - NOT_HANDSHAKING, non determina nessuna operazione da intraprendere;

- Status, che monitora i risultati delle operazioni di cifratura/decifratura e può assumere i seguenti valori:

  - BUFFER_OVERFLOW, determina l'aumento della capacità del buffer ed un nuovo tentativo di eseguire l'operazione;
  - BUFFER_UNDERFLOW, ci si rimette in ascolto sul canale per ricevere ulteriori dati, da accorpare ai precedenti, e comporre un pacchetto completo, prima di ritentare l'operazione;
  - CLOSED, durante la fase di Handshake indica l'avvenimento di un errore fatale ed il conseguente lancio dell'eccezione *SSLExcpetion*, in questo caso la comunicazione non può proseguire e viene segnalato l'errore dall'applicazione. Al di fuori invece segnala la chiusura della connessione e viene lanciata un eccezione *IOException*, stampando a video un messaggio.
  - OK, indica che l'operazione è stata completata con successo.

5.3 Casi d'uso con Jolie

All'interno di questa sezione vengono descritti tre possibili casi di applicazione del protocollo SSL, implementato in Jolie attraverso la classe *SSLProtocol*.

Ciascuno utilizza uno dei protocolli sicuri forniti all'interno delle librerie, ossia HTTPS e SODEPS, e, attraverso la configurazione di alcuni parametri fondamentali, realizza comunicazioni sicure.

Lo scopo quindi risulta essere la verifica del funzionamento del protocollo SSL e della corretta implementazione di *SSLProtocol*.

5.3.1 Configurazione dei parametri

Utilizzando la corretta sintassi in Jolie, è possibile impostare alcuni parametri con cui verrà configurato SSL e di conseguenza *SSLEngine*. Essi vanno specificati subito dopo la dichiarazione del protocollo di comunicazione scelto (sono validi solo per quanto riguarda HTTP e SODEP) e sono i seguenti:

- .ssl.protocol, specifica la versione del protocollo SSL da utilizzare. Le versioni a disposizione sono: SSL, SSLv2, SSLv3, TLS, TLSv1, TLSv1.1, TLSv1.2.
  Valore default: TLSv1;

- .ssl.keyStore, indirizzo del keystore dal quale ottenere le informazioni da inviare per eseguire l'autenticazione, obbligatorio per la modalità Server.
  Valore default: nessuno;

- .ssl.keyStorePassword, password per accedere al keystore, può non essere impostata qualora non sia stata aggiunta in fase di creazione. Se è errata viene generato un errore di accesso.
  Valore default: nessuno;

- .ssl.keyStoreFormat, formato utilizzato per la memorizzazione delle chiavi. Può assumere i valori JKS, JCEKS oppure PKCS12.
  Valore default: JKS;

- .ssl.trustStore, indirizzo del truststore dal quale ottenere le informazioni per verificare l'autenticità del certificato ricevuto.
  Valore default: ricerca il truststore di default contenuto nella JRE della macchina in uso, diventa obbligatorio se la ricerca non ha successo;

- .ssl.trustStorePassword, password per accedere al truststore, può non essere impostata qualora non sia stata aggiunta in fase di creazione. Se è errata viene generato un errore di accesso.
  Valore default: nessuno;

- .ssl.trustStoreFormat, formato utilizzato all'interno del truststore per la memorizzazione. Può assumere i valori JKS, JCEKS oppure PKCS12.
  Valore default: JKS. [29]

5.3.2 Generazione di keystore e truststore

Requisito fondamentale per il corretto funzionamento delle comunicazioni sicure è la presenza di keystore e truststore, utilizzati per fornire e verificare credenziali.

Per semplificarne l'utilizzo è possibile creare delle versioni in locale attraverso il comando *keytool* fornito con Java, in modo da ottenere una chiave privata contenuta all'interno del keystore e, successivamente, creare un certificato pubblico, che sarà autofirmato e quindi non riconosciuto da autorità certificate, da inserire all'interno del truststore.

Le operazioni da eseguire sono le seguenti:

- generare un keystore contenente la chiave privata attraverso il comando **keytool -genkey -alias privateKey1 -keystore keyStore.store**, il quale, in seguito all'inserimento di alcune informazioni per la configurazione, genera un file locale contenente la chiave privata, caratterizzata da un particolare alias. Durante la fase di creazione viene richiesta la specifica della password per la protezione del file;

- ottenere un certificato temporaneo, il quale verrà utilizzato per inserire all'interno di un truststore la chiave pubblica. Il comando da eseguire è il seguente **keytool -export -alias privateKey1 -file certfile.cer -keystore keyStore.store** e genera anche esso un file locale.

- creare un truststore a partire dal certificato ottenuto in precedenza, importandone le informazioni all'interno di un nuovo file, in modo da memorizzare la chiave pubblica. Il comando è il seguente **keytool -import -alias publicCert -file certfile.cer -keystore trustStore.store**.
  Anche in questo caso viene richiesta la specifica della password per la protezione del file. [30]
  Questa operazione può essere eseguita anche quando si dispone già di un certificato salvato sulla propria macchina, ad esempio quello fornito da un qualsiasi sito web.

Ottenuti questi file è possibile utilizzarli all'interno di applicazioni Client/Server che supportino comunicazioni sicure tramite SSL.

5.3.3 Caso d'uso 1: chat tra due terminali

In questo primo caso d'uso viene realizzato un'applicativo che simula una semplice chat tra due utenti, i quali comunicano inviando messaggi attraverso un canale reso sicuro dal protocollo SSL, sfruttando la classe *SSLProtocol* ed *SSLEngine*, insieme ai suoi parametri di configurazione.

Per un corretto funzionamento sono stati creati un keystore ed un truststore per ogni utente, successivamente il truststore di un utente è stato utilizzato dall'altro; vi è stata quindi un operazione di scambio dei file per consentire la decifratura dei dati ricevuti utilizzando la propria chiave privata, contenuta nel keystore, e la cifratura attraverso la chiave pubblica del destinatario, contenuta nel truststore scambiato in precedenza. Senza l'esecuzione di questa operazione ogni utente avrebbe avuto un truststore che rappresentava fondamentalmente la chiave pubblica della propria chiave privata, impedendo quindi la connessione sicura verso l'altro utente.

Le figure 5.3 e 5.4 mostrano il codice necessario al funzionamento, successivamente si passerà alla descrizione nel dettaglio.

Per semplicità viene mostrato solo il codice riguardante uno dei due utenti, essendo l'altro molto simile, a parte qualche parametro di configurazione.

Questa prima parte di codice mostra le configurazioni fondamentali per eseguire un servizio in Jolie che comunichi, come in questo caso, con un altro servizio situato su un indirizzo diverso, il quale rappresenta l'utente a cui inviare i messaggi.

La riga caratterizzata dall'istruzione *include* specifica quali librerie di Jolie si vogliono utilizzare in determinati punti del programma, in questo caso si farà uso della sola libreria *Console* [31] per poter leggere i dati passati da tastiera dall'utente.

Successivamente, è necessario configurare le porte di ingresso e uscita in modo da poter inviare e ricevere i messaggi da parte dell'altro utente.

```
//Inclusione delle librerie utili
include "console.iol"

//Configurazione della porta di ingresso
inputPort In {
    Location: "socket://localhost:8000"
    Protocol: sodeps {
        .ssl.keyStore = "keyStore.store";
        .ssl.keyStorePassword = "keystore"
    }
    RequestResponse: send(string)(void)
}

//Configurazione della porta di uscita
outputPort Out {
    Location: "socket://localhost:8001"
    Protocol: sodeps {
        .ssl.trustStore = "trustStore.store";
        .ssl.trustStorePassword = "truststore"
    }
    RequestResponse: send(string)(void)
}

//Inizializzazione oggetti utili
init
{
    registerForInput@Console()();
    //Nome dell'utente da cui si riceveranno i messaggi
    nomeSender = "Client2"
}

//Inclusione file contenente istruzioni per invio/ricezione
include "../chatMessage.ol"
```

Figura 5.3: Configurazione del servizio di chat

La porta di ingresso viene configurata all'interno di un indirizzo locale, tutti i messaggi quindi dovranno essere inviati a questo indirizzo. Il protocollo da utilizzare è SODEPS, il quale sfrutta la sicurezza aggiuntiva del protocollo SSL; in questo caso viene specificato dove è possibile trovare il keystore, contenente la chiave privata utilizzata per decifrare i messaggi in arrivo, e quale sia la password per l'accesso. In seguito vengono dichiarate le operazioni che caratterizzano l'invio dei messaggi.

Anche la porta in uscita viene configurata all'interno di un indirizzo locale, il quale specifica a quale destinazione devono essere inviati i messaggi. Le configurazioni sono simili alla porta in ingresso, eccezion fatta per i parametri del protocollo SODEPS; in questo caso viene specificato il truststore, contenente la chiave pubblica ottenuta dall'altro utente, in modo da poterla utilizzare per cifrare i messaggi, i quali saranno decifrati dalla corrispondente chiave privata.

All'interno dell'istruzione *init* vengono inizializzati alcuni oggetti utili, ad esempio viene lanciata la ricezione di dati inviati dalla tastiera da parte dell'utente e viene creata una stringa che servirà per specificare da quale utente giunge il messaggio.

Infine, viene incluso il file contenente le istruzioni per la gestione dell'invio e ricezione dei messaggi, mostrato di seguito; questa scelta è stata fatta per evitare di dover copiare ed incollare codice simile in file diversi, operazione spesso sconsigliata. I file che rappresentano i due utenti sono molto simili tra loro, cambiano solamente gli indirizzi delle porte, keystore, truststore e la stringa per la specifica dell'utente che ha inviato il messaggio; non avrebbe quindi senso ripetere lo stesso codice ma è più corretto includere un file esterno che rappresenta le operazioni comuni.

All'interno dell'istruzione *main*, che rappresenta il punto di ingresso per l'esecuzione di qualsiasi servizio Jolie, viene lanciato un ciclo *while*, dove vengono inizialmente gestite le possibili eccezioni generate da errori di connessione, ad esempio se l'utente a cui mandare un messaggio fosse disconnesso.

Successivamente vengono dichiarate due *input choice*, le quali restano in ascolto in attesa della ricezione dell'input per poter eseguire determinate operazioni:

- nel primo caso, qualora venisse ricevuto da tastiera il messaggio da inviare, il servizio provvede a richiamare l'istruzione che gestisce il recapito del messaggio all'altro utente;

- nel secondo caso, quando viene richiamata la funzione di ricezione di un messaggio viene stampato a video il contenuto corrispondente.

```
//Punto di ingresso
main
{
    keepRunning = true;
    //Messaggio di benvenuto
    println@Console("Digita un messaggio per avviare la conversazione\n")();
    //Ciclo while sempre attivo
    while(keepRunning)
    {
        //Gestione delle eccezioni
        scope( fault_conn )
        {
            //Messaggi per segnalare eccezione
            install( IOException => println@Console( "\nErrore di connessione" )() );
            install( ConnectException => println@Console( "\nErrore di connessione" )() );

            //Invio di un messaggio
            [in(command)]{
                send@Out(command)()
            }

            //Ricezione di un messaggio
            [send(messaggio)()
            {
                println@Console(nomeSender + " scrive: " + messaggio)()
            }]
        }
    }
}
```

Figura 5.4: Istruzioni per invio/ricezione messaggi per servizio di chat

Se il codice è stato scritto senza errori e le configurazioni delle porte in ingresso ed uscita, per entrambi gli utenti, sono corrette, il risultato finale è rappresentato dalla figura sottostante.

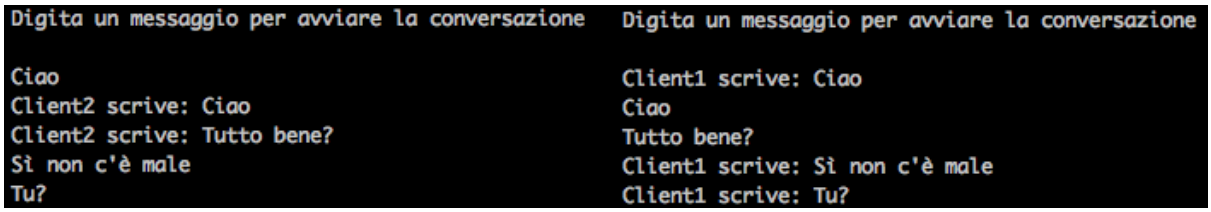

Figura 5.5: Esempio di esecuzione del servizio di chat

5.3.4 Caso d'uso 2: download di una pagina web

In questo secondo caso d'uso viene realizzato un'applicativo che scarica una pagina web da un sito specifico e la salva all'interno di un file, comunicando attraverso un canale reso sicuro dal protocollo SSL, sfruttando la classe *SSLProtocol* ed *SSLEngine*, insieme ai suoi parametri di configurazione.

Rispetto al precedente viene utilizzato solamente un truststore, contenente la chiave pubblica del sito da cui scaricare la pagina, per consentire la cifratura dei messaggi inviati in maniera corretta.

Per ottenere la chiave pubblica, da inserire successivamente nel truststore attraverso le operazioni di *keytool*, è necessario eseguire il download del certificato SSL fornito dal sito di interesse.

Viene richiesto quindi di collegarsi e, tramite le operazioni consentite dai browser, di scaricare il certificato, da importare successivamente all'interno di un file che costituirà il truststore da utilizzare all'interno dell'applicativo. Lo stesso file può contenere più chiavi pubbliche, a patto che siano rappresentate da alias diversi.

Una interessante implementazione futura è rappresentata dalla creazione di un servizio per il download e l'aggiunta automatica del certificato all'interno del truststore, evitando così operazioni esterne.

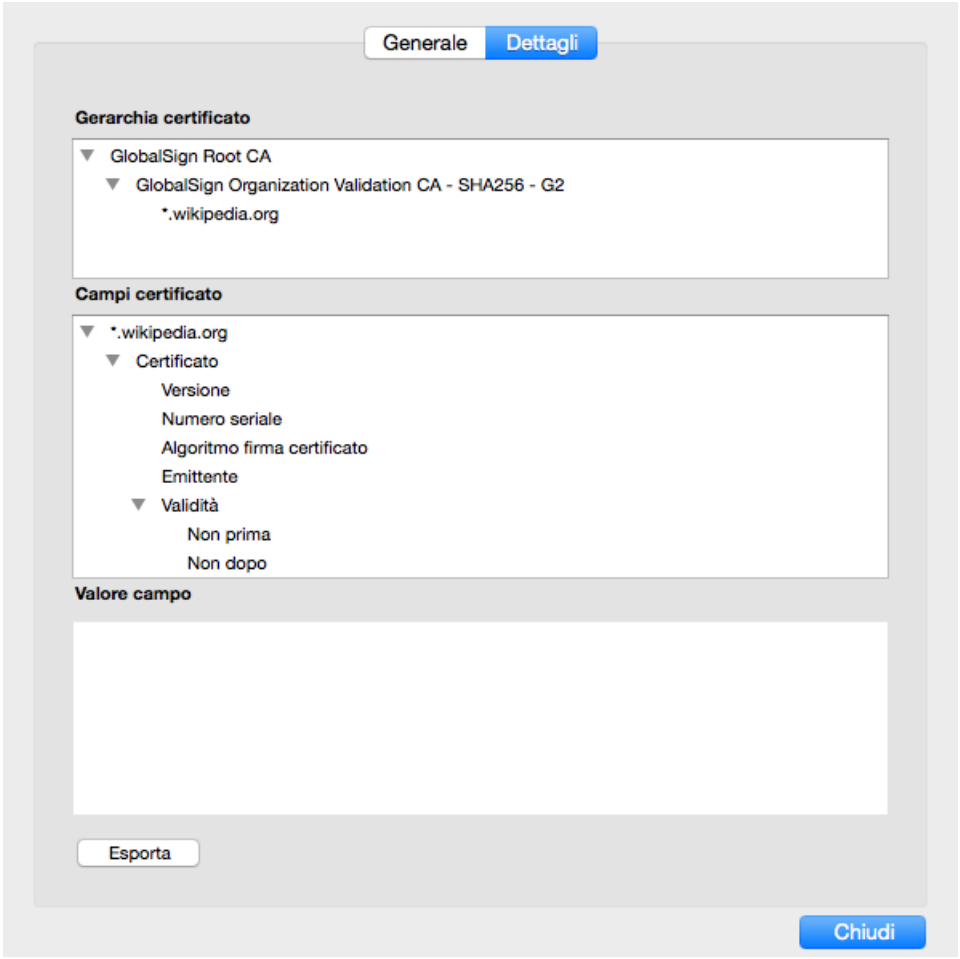

Figura 5.6: Pagina per l'esportazione di un certificato in Mozilla Firefox

Oltre alla libreria *Console*, precedentemente utilizzata, questa volta viene aggiunta anche la libreria *File*, la quale consente la gestione di tutte le operazioni caratterizzanti i files.

La configurazione delle porte in questo caso riguarda solamente quella di uscita, essendo questo servizio caratterizzato dalla sola richiesta di invio di una pagina, inoltrata ad un sito web. L'indirizzo rappresenta il sito da cui si desidera scaricare la pagina, facendo attenzione a specificare come porta la numero 443, utilizzata universalmente per le connessioni HTTP che supportano SSL.

Successivamente è necessario configurare il protocollo HTTPS affinchè consenta una connessione sicura, impostando il truststore, con la relativa password per l'accesso, l'alias della pagina che si vuole scaricare e la compressione del risultato, in questo caso disattivata. [32]

In seguito viene specificata l'operazione che si occuperà della richiesta e della ricezione della pagina web.

```
//Inclusione delle librerie utili
include "console.iol"
include "file.iol"

//Configurazione della porta di uscita
outputPort Out {
    Location: "socket://en.wikipedia.org:443/wiki"
    Protocol: https {
        .compression = false;
        .osc.download.alias = "/Jolie_(programming_language)";
        .ssl.trustStore = "certificates.store";
        .ssl.trustStorePassword = "truststore"
    }
    RequestResponse: download (void)(string)
}

//Punto di ingresso
main
{
    //Richiesta della pagine
    download@Out( )( pagina );
    //Creazione del file
    risultato.filename = "web_page.txt";
    risultato.content = pagina;
    //Scrittura del risultato all'interno del file
    writeFile@File( risultato )();
    //Conferma in caso di successo
    println@Console( "Pagina scaricata" )()
}
```

Figura 5.7: Configurazione e istruzioni per servizio di download pagina web

All'interno dell'istruzione *main* viene lanciato il metodo che si occupa di collegarsi al sito web, specificato dalla porta di uscita, ed ottenere la pagina desiderata. In caso di successo viene creato un nuovo file ed il contenuto del download viene scritto al suo interno, tramite l'utilizzo dei metodi e delle richieste della libreria *File*. [33]

Il risultato, in caso di codice senza errori ed una configurazione corretta per la connessione SSL, è la creazione di un file contenente il codice HTML della pagina specificata.

```
<!DOCTYPE html>
<html lang="en" dir="ltr" class="client-nojs">
<head>
<meta charset="UTF-8" />
<title>Jolie (programming language) - Wikipedia, the free encyclopedia</title>
<script>document.documentElement.className = document.documentElement.className.replace( /
<script>window.RLQ = window.RLQ || []; window.RLQ.push( function () {
mw.config.set({"wgCanonicalNamespace":"","wgCanonicalSpecialPageName":false,"wgNamespaceNu
"wgMediaViewerEnabledByDefault":true,"wikilove-recipient":"","wikilove-anon":0,"wgPoweredB
} );</script>
<link rel="stylesheet" href="https://en.wikipedia.org/w/load.php?debug=false&amp;lang=en&a
<link rel="stylesheet" href="https://en.wikipedia.org/w/load.php?debug=false&amp;lang=en&a
<meta name="ResourceLoaderDynamicStyles" content="" />
<link rel="stylesheet" href="https://en.wikipedia.org/w/load.php?debug=false&amp;lang=en&a
<style>a:lang(ar),a:lang(kk-arab),a:lang(mzn),a:lang(ps),a:lang(ur){text-decoration:none}<
<script async="" src="https://en.wikipedia.org/w/load.php?debug=false&amp;lang=en&amp;modu
<meta name="generator" content="MediaWiki 1.26wmf18" />
<link rel="alternate" href="android-app://org.wikipedia/http/en.m.wikipedia.org/wiki/Jolie
<link rel="alternate" type="application/x-wiki" title="Edit this page" href="/w/index.php?
<link rel="edit" title="Edit this page" href="/w/index.php?title=Jolie_(programming_langua
<link rel="apple-touch-icon" href="/static/apple-touch/wikipedia.png" />
<link rel="shortcut icon" href="/static/favicon/wikipedia.ico" />
<link rel="search" type="application/opensearchdescription+xml" href="/w/opensearch_desc.p
<link rel="EditURI" type="application/rsd+xml" href="//en.wikipedia.org/w/api.php?action=r
<link rel="copyright" href="//creativecommons.org/licenses/by-sa/3.0/" />
<link rel="alternate" type="application/atom+xml" title="Wikipedia Atom feed" href="/w/inc
<link rel="canonical" href="https://en.wikipedia.org/wiki/Jolie_(programming_language)" />
<link rel="dns-prefetch" href="//meta.wikimedia.org" />
<!--[if lt IE 7]><style type="text/css">body{behavior:url("/w/static/1.26wmf18/skins/Vectc
</head>
<body class="mediawiki ltr sitedir-ltr ns-0 ns-subject page-Jolie_programming_language ski
        <div id="mw-page-base" class="noprint"></div>
        <div id="mw-head-base" class="noprint"></div>
        <div id="content" class="mw-body" role="main">
```

Figura 5.8: Frammento del risultato del servizio di download pagina web

5.3.5 Caso d'uso 3: upload di una pagina web

In questo terzo caso d'uso viene realizzato un'applicativo che carica una pagina web ad uno specifico indirizzo, fungendo quindi da server, comunicando attraverso un canale reso sicuro dal protocollo SSL, sfruttando la classe *SSLProtocol* ed *SSLEngine*, insieme ai suoi parametri di configurazione.

Nonostante il codice ed i parametri di configurazione di SSL siano corretti, con particolare attenzione rivolta alla creazione di un keystore per la fornitura delle credenziali ed il successivo inserimento del certificato derivante all'interno del sistema, si sottolineano errori che identificano quindi un caso limite ancora non coperto dall'attuale implementazione.

```
include "console.iol"

inputPort In {
    Location: "socket://localhost:8000"
    Protocol: https {
        .compression = false;
        .contentType = "text/html" ;
        .format = "html";
        .ssl.keyStore = "keyStore.store";
        .ssl.keyStorePassword = "keystore"
    }
    RequestResponse: index (void)(string)
}

main
{
  index()( "Pagina caricata" )
}
```

Figura 5.9: Configurazione e istruzioni per servizio di upload pagina web

Capitolo 6

Conclusioni

All'interno di questa tesi è stata affrontata la tematica della realizzazione di comunicazioni sicure, in modo da ottenere l'indipendenza di queste ultime dal canale utilizzato, con l'ausilio di strumenti in grado di fornire supporto per la creazione di applicativi orientati allo scambio di dati e messaggi, quali i linguaggi di programmazione Java e Jolie, il quale è basato sul precedente.

Sono state inizialmente analizzate le principali caratteristiche e le problematiche più importanti che è necessario dover risolvere in modo da poter arrivare al risultato desiderato.

Successivamente, è stato dato un ampio sguardo ad una delle scienze più applicate per risolvere i problemi tipici che affliggono questo tipo di comunicazioni, la crittografia. Sono stati elencati gli strumenti messi a disposizione ed il loro funzionamento. Purtroppo questa non è una scienza esatta e la sicurezza totale non è garantita, soprattutto dando uno sguardo alla nascita di future innovazioni.

La crittografia viene poi applicata al protocollo SSL, il quale rappresenta la soluzione maggiormente diffusa, sia sul Web che in altri ambiti, per proteggere le informazioni personali che transitano tra gli end-point di una comunicazione. Sono state elencate le principali caratteristiche, alcuni cenni riguardanti la nascita e lo sviluppo ed è stato descritto il funzionamento di questo protocollo, soprattutto per quanto riguarda la sua fase preliminare, che è una delle parti che lo caratterizzano maggiormente.

In seguito, è stata analizzata la soluzione fornita all'interno delle librerie del linguaggio Java per realizzare comunicazioni indipendenti dal mezzo di comunicazione che soddisfino le politiche dettate dal protocollo SSL. Questa soluzione è rappresentata dalla classe *SSLEngine*, che è quindi stata esaminata, a partire dal ciclo di vita e dall'inizializzazione, fino ad arrivare all'interazione all'interno di un applicazione.

Quanto esplorato in precedenza viene poi applicato a Jolie, un linguaggio di programmazione basato sulle comunicazioni e sviluppato in Java, all'interno dell'Università di Bologna. Dopo uno sguardo generale alle sue caratteristiche è stata approfondita la gestione dei protocolli, e di conseguenza, l'introduzione di SSL all'interno di essi, realizzata tramite la classe *SSLProtocol*. Questa classe contiene ed implementa i concetti analizzati nel capitolo riguardante Java, adattandoli all'architettura ed alla progettazione pensata appositamente per Jolie; è stata quindi effettuata un'analisi del codice e della gestione della classe *SSLEngine* per realizzare comunicazioni sicure. Infine, per verificare l'effettivo funzionamento, sono stati creati due semplici casi d'uso per poter sfruttare i vantaggi offerti da Jolie, il quale è particolarmente indicato per la creazione di applicazioni orientate ai servizi.

Il lavoro precedentemente descritto è rivolto ad un'analisi delle soluzioni adottate all'interno delle librerie che compongono Jolie. In particolare sono stati identificati casi d'uso non ancora coperti dall'attuale implementazione di SSL in Jolie che perciò richiederà future correzioni e verifiche. Questo contributo rappresenta una base per tale intervento.

L'analisi, la descrizione e l'utilizzo di tutte le tecnologie contenute in questa tesi è stato molto interessante e motivante, in quanto vanno a risolvere in maniera generale una delle tematiche più importanti ai giorni nostri, poiché le tecnologie che fanno affidamento sulla rete, soprattutto il Web, sono in continua evoluzione e sono sempre di più i servizi che operano scambi di dati sensibili, i quali sono a rischio costante di intercettazione da parte di terzi.

Bibliografia

[1] Alfredo De Santis. *Corso di Sicurezza su Reti*. Università di Salerno, 2002.
`http://www.di.unisa.it/~ads/corso-security/www/CORSO-0102/tesina_SSH/testo%201.htm`

[2] Sinte. *La guida di Internet - Approfondimenti tecnici - Parte II*. 2010.
`www.sinte.net/files/help/tecnici.htm`

[3] Simone Faro. *Corso di Informatica (Scienze Umanistiche)*. Università di Catania, 2015.
`http://www.dmi.unict.it/~faro/informatica/dispense/Capitolo7.pdf`

[4] Ugo Chirico. *La Crittografia per la protezione delle informazioni sulla rete*. Giugno 1999
`http://www.mokabyte.it/1999/06/crittografia.htm`

[5] Alberto Bartoli. *Il ruolo della crittografia nella sicurezza informatica*. Nuova Secondaria, 2008.
`http://www.dmi.units.it/~fabris/Crittografia.pdf`

[6] Scott Oaks. *Sicurezza in Java*. Apogeo, 2001

[7] CCM - Community di assistenza e consulenza. *Crittografia - Secure Sockets Layers (SSL)*. 2015.
`http://it.ccm.net/contents/815-crittografia-secure-sockets-layers-ssl`

[8] DigiCert, Inc. *What Is SSL (Secure Sockets Layer) and What Are SSL Certificates?*. 2015.
https://www.digicert.com/ssl.htm

[9] Alfredo De Santis. *Corso di Sicurezza su Reti*. Università di Salerno, 2004.
http://www.di.unisa.it/~ads/corso-security/www/CORSO-9900/SSL/main.htm

[10] Oracle America, Inc. *X.509 Certificates - Oracle Documentation*. 2015
https://docs.oracle.com/javase/8/docs/technotes/guides/security/cert3.html

[11] Oracle America, Inc. *Java Secure Socket Extension (JSSE) Reference Guide*. 2015
https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html

[12] Federico Reali. *Elementi di Sicurezza Informatica*. Università di Perugia, 2010.
http://www.dmi.unipg.it/bista/didattica/sicurezza-pg/seminari2009-10/SSLpresentazione.pdf

[13] Google Webmaster Central Blog. *HTTPS as a ranking signal*. Agosto 2015
http://googlewebmastercentral.blogspot.it/2014/08/https-as-ranking-signal.html

[14] Microsoft Corporation. *SSL/TLS in Detail*. Luglio 2003
https://technet.microsoft.com/en-us/library/cc785811%28v=ws.10%29.aspx

[15] Oracle America, Inc. *SSLSocket (Java Platform SE 8) - Oracle Documentation*. 2015
http://docs.oracle.com/javase/8/docs/api/javax/net/ssl/SSLSocket.html

[16] Oracle America, Inc. *SSLContext (Java Platform SE 8) - Oracle Documentation*. 2015
https://docs.oracle.com/javase/8/docs/api/javax/net/ssl/SSLContext.html

[17] Oracle America, Inc. *SSLSocketFactory (Java Platform SE 8) - Oracle Documentation*. 2015
`http://docs.oracle.com/javase/8/docs/api/javax/net/ssl/SSLSocketFactory.html`

[18] Oracle America, Inc. *SSLServerSocketFactory (Java Platform SE 8) - Oracle Documentation*. 2015
`http://docs.oracle.com/javase/8/docs/api/javax/net/ssl/SSLServerSocketFactory.html`

[19] Oracle America, Inc. *SSLServerSocket (Java Platform SE 8) - Oracle Documentation*. 2015
`http://docs.oracle.com/javase/8/docs/api/javax/net/ssl/SSLServerSocket.html`

[20] Nuno Santos. *Using SSL with Non-Blocking IO*. Marzo 2004
`http://www.onjava.com/pub/a/onjava/2004/11/03/ssl-nio.html`

[21] Oracle America, Inc. *SSLEngine (Java Platform SE 8) - Oracle Documentation*. 2015
`https://docs.oracle.com/javase/8/docs/api/javax/net/ssl/SSLEngine.html`

[22] Oracle America, Inc. *KeyStore (Java Platform SE 8) - Oracle Documentation*. 2015
`http://docs.oracle.com/javase/8/docs/api/java/security/KeyStore.html`

[23] Oracle America, Inc. *Package java.nio (Java Platform SE 8) - Oracle Documentation*. 2015
`http://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html`

[24] David Flanagan. *Java in a Nutshell: A Desktop Quick Reference, 5th Edition*. O'Reilly, 2005

[25] Jolie. *Jolie Programming Language - Official Website*. 2015
`http://www.jolie-lang.org/`

[26] The Jolie Team. *Jolie Documentation - Getting started - Behaviour and deployment.* 2015
`http://docs.jolie-lang.org/#!documentation/getting_started/behavior_and_deployment.html`

[27] The Jolie Team. *Jolie Documentation - Basics - Communication Ports.* 2015
`http://docs.jolie-lang.org/#!documentation/basics/communication_ports.html`

[28] Fabrizio Montesi. *JOLIE: a Service-oriented Programming Language.* Università di Bologna, 2010
`http://www.fabriziomontesi.com/files/m10.pdf`

[29] The Jolie Team. *Jolie Documentation - Protocols - SSL.* 2015
`http://docs.jolie-lang.org/#!documentation/protocols/ssl.html`

[30] Alvin Alexander. *The Java keytool command, keystore files, and certificates.* 2015
`http://alvinalexander.com/java/java-keytool-keystore-certificates`

[31] The Jolie Team. *Jolie Documentation - Standard Library API - Console.* 2015
`http://docs.jolie-lang.org/#!documentation/jsl/Console.html`

[32] The Jolie Team. *Jolie Documentation - Protocols - HTTP.* 2015
`http://docs.jolie-lang.org/#!documentation/protocols/http.html`

[33] The Jolie Team. *Jolie Documentation - Standard Library API - File.* 2015
`http://docs.jolie-lang.org/#!documentation/jsl/File.html`

[34] La Maestra Roberto. *Implementazione di un modulo HTTPS in Jolie per l'orchestrazione di servizi.* Università di Bologna, 2010
`http://amslaurea.unibo.it/1457/1/LA_MAESTRA_ROBERTO_Implementazione_modulo_HTTPS_in_Jolie.pdf`

Ringraziamenti

Sono passati quattro anni dal primo giorno di lezione di questo Corso di Laurea e finalmente sono arrivato anche io al momento di concludere questo percorso ed iniziare nuove esperienze. Guardandomi indietro sono passati tanti giorni, che sono diventanti mesi, ed in seguito anni, ognuno ha rappresentato tante piccole difficoltà che ho dovuto affrontare e superare. Mi reputo una persona a cui piace tenersi impegnato e non avere troppi momenti "di vuoto", purtroppo a volte esagero e questo mi porta, a malincuore, a dover posticipare o mettere in secondo piano altri impegni. Più volte ho dovuto fare i salti mortali per poter conciliare due impegni lavorativi, passioni personali e l'Università; questo mi ha portato a dover allungare i tempi per finire gli esami ed ha iniziato lentamente ed in maniera progressiva a farmi perdere la cognizione di quello che effettivamente sono in grado di fare, facendomi affrontare alcuni ostacoli non con entusiasmo ma con molta ansia e la paura di non essere all'altezza delle aspettative.

Fortunatamente, intorno a me ci sono persone che mi hanno sostenuto e spronato durante queste difficoltà, facendomi capire che prima di tutto vengono le soddisfazioni personali e non il giudizio degli altri, ma soprattutto insegnato a guardare sempre il bicchiere mezzo pieno, perché se io non credo in me stesso per primo, nessuno lo farà al posto mio.

Vorrei quindi ora ringraziare alcune di queste persone perché è anche grazie a loro se sono arrivato a questo traguardo e probabilmente non ho mai avuto occasione di esprimere quello che penso.

Grazie a Gianmarco e Grazia, i miei genitori. Siete le persone più importanti della mia vita e senza di voi tutto questo non sarebbe stato possibile. Non mi avete mai fatto

mancare nulla e, soprattutto, avete dato libero sfogo alle mie idee, rimanendo sempre un passo indietro, senza mai farmi sentire obbligato a fare determinate scelte, ma anzi sostenendomi e facendomi sentire il vostro appoggio. Vorrei restituirvi tutto questo ma penso sia impossibile, quindi cercherò di farvi capire che quello che sono oggi deriva da tutto quello che voi avete fatto per me. Mi avete insegnato il valore della fatica e del sacrificio, guardandovi affrontare i problemi e le difficoltà ho imparato come essere forte quando il mondo che ci si è costruiti intorno sembra crollare.

Grazie a Simone, mio fratello. Siamo due persone dal carattere ed abitudini diverse, ma nonostante tutto tornare a casa ogni giorno e trovare una persona con cui poter parlare e sfogarsi, anche di problemi di poco conto, è uno dei regali più belli che la vita possa darti. Con il passare degli anni stiamo diventando "grandi", e rendermi conto che hai 19 anni mi risulta ancora difficile, e questo sta portando ad una crescita del nostro legame e sento che ora puoi capirmi davvero, nonostante debba ancora insegnarti molte cose. Proprio per questo voglio dirti che per te ci sarò sempre quando avrai bisogno.

Grazie a Talvanne detto Nino, Marisa, Romano e Paola, i miei nonni. Non sono molte le persone che possono contare su nonni fantastici come voi. Siete sempre stati presenti per aiutarmi e per essere i miei "genitori" nel momento del bisogno, dispensando consigli utilissimi, facendomi trovare sempre un piatto caldo di ritorno dal lavoro o dalle lezioni e mostrandomi sempre il vostro sostegno e la vostra ammirazione nei miei confronti.

Grazie a Olimpia, che considero come una sorella. Sei una delle persone che negli ultimi 3 anni mi è stata più vicina, siamo passati da semplici conoscenti ad avere un grandissimo rapporto, e proprio grazie alla tua presenza quotidiana posso dire di essere cresciuto e migliorato, cercando di aiutare te a fare lo stesso. Ci sei stata in tutti i momenti, belli o brutti che fossero, e nel periodo più difficile che ho dovuto affrontare non ti sei fatta da parte, ma anzi ti sei messa in gioco e mi hai dato una mano a risollevarmi. Anche a te voglio dire che ci sarò sempre, ogni volta che avrai bisogno non esitare a chiamarmi.

Grazie a Matteo, amico di una vita e per me un fratello. Ci conosciamo da tantissimo tempo e, nonostante ultimamente ci siamo persi un po' di vista, non per contrasti ma per impegni diversi, per me rimani sempre un grande e sento che il nostro rapporto è ancora lo stesso dai tempi dell'adolescenza. Insieme abbiamo fatto qualsiasi cosa e siamo

cresciuti, ma in fondo, nonostante tutto, siamo rimasti gli stessi ragazzi spensierati ed ognuno dei due presta sempre molta attenzione ai consigli ed alle parole dell'altro.

Grazie a Vanessa, la mia ragazza. Il nostro legame è stato altalenante, ma nonostante tutto mi hai fatto capire che a volte è meglio mettere da parte l'orgoglio ed esprimere quello che veramente si pensa. Hai affrontato soprattutto l'ultimo periodo del mio percorso di studi, quello più duro probabilmente, ma mi hai sempre mostrato la tua ammirazione ed il tuo rispetto, spronandomi a fare sempre meglio, soprattutto per me stesso, dovendo anche utilizzare molta pazienza quando ce n'è stato bisogno. Non sono portato per le smancerie come ben sai ed a volte sono abbastanza pesante, ma la tua voglia di esserci, nonostante tutto, è per me molto importante.

Grazie a tutti gli amici e le amiche della "balotta". Siamo un gruppo davvero numeroso ed ognuno di noi ha un carattere particolare, ma proprio questo ci porta ad essere molto uniti e a fare tantissime esperienze sempre insieme. Nei momenti di difficoltà siete sempre stati presenti ed avete sempre cercato una soluzione per sistemare le cose. Non dimenticherò mai le infinite serate sempre in giro e tutte le volte che, soprattutto durante ricorrenze particolari, ci siamo sempre fatti riconoscere. Il tempo purtroppo passa inesorabilmente, ma per me rimarremo sempre i ragazzi dalla "testa vuota".