Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea in Informatica

IMPLEMENTAZIONE DI UN MODULO HTTPS IN JOLIE PER L'ORCHESTRAZIONE DI SERVIZI

Tesi di Laurea in
Informatica

Relatore
Chiar.mo Prof. Gorrieri Roberto

Candidato
La Maestra Roberto

Correlatori
Dr. Guidi Claudio
Dott. Montesi Fabrizio

Sessione II
Anno Accademico 2009/2010

Indice

Elenco delle figure

Introduzione

Nel contesto tecnologico attuale, il crescente sviluppo dei sistemi distribuiti e il loro utilizzo pressoché capillare in una rete altamente etereogenea come Internet, ha sollevato l'esigenza di formalizzare un'architettura per il supporto dell'interoperabilità tra elaboratori con caratteristiche differenti, in maniera indipendente dalle stesse. In questo contesto assumomo rapidamente un'elevata diffusione i *Web Services*.

Quest'effetto ha portato allo sviluppo di un nuovo paradigma di programmazione orientato ai servizi (SOC) che consente di progettare applicazioni distribuite componendo servizi esistenti e riducendo le dipendenze tra di essi. SOC propone anche l'utilizzo di linguaggi di composizione ad alto livello.

La composizione di servizi può avvenire tramite la *coreografia* o l'*orchestrazione*. Mentre con la prima si intende la descrizione globale di ruoli e interazioni dei servizi, con orchestrazione si intende invece la progettazione di servizi visti come unità separate. Con l'approccio dell'orchestrazione, in ogni servizio è definita la logica applicativa e di interazione con gli altri servizi, senza avere una visione globale dell'intera applicazione distribuita.

JOLIE è un linguaggio ad alto livello progettato per creare orchestratori di servizi, ed è stato sviluppato all'interno del progetto europeo SENSORIA, di cui l'Università di Bologna è parte attiva. E' interamente realizzato in Java e distribuito in formato *Open*.

La semantica di JOLIE è basata sul calcolo formale SOCK, che formalizza tutte le primitive essenziali per la descrizione dei meccanismi del paradigma SOC.

La comunicazione dei dati è fornita allo sviluppatore ad un alto livello di astrazione, che si limita alla dichiarazione del protocollo da utilizzare, tra quelli supportati. Questo rappresenta un grosso vantaggio nell'utilizzo di JOLIE, che si fa carico della gestione della comunicazione e delle sue problematiche.

Tuttavia, i protocolli supportati sono SOAP, HTTP, BTL2CAP e SODEP (nativo di JOLIE), tutti in chiaro e *human-readable*, e non forniscono alcuna garanzia di sicurezza e riservatezza dei dati. Questo è un grosso limite di JOLIE, che ne restringe largamente il campo di applicazione.

Per avere garanzie di protezione dei dati contro eventuali interlocutori in ascolto, bisogna rispettare le tre proprietà di una comunicazione sicura, che sono: *riservatezza dei dati*, che devono risultare indecifrabili a terze parti in ascolto, *autenticazione degli interlocutori*, i quali si assicurano dell'identità dell'altro capo della connessione, e *integrità dei dati*, che non devono subire alterazioni durante il trasporto. Il protocollo piu' utilizzato per garantire la sicurezza della comunicazione è SSL/TLS.

Questo lavoro di tesi realizza un modulo d'estensione per JOLIE che applica alla comunicazione le politiche di sicurezza di SSL/TLS, e fornisce il supporto del protocollo HTTPS. Il modulo utilizza l'API Java *SSLEngine*.

Viene inoltre realizzato un caso d'uso in cui si utilizza JOLIE per orchestrare un servizio di autenticazione integrata tramite HTTPS tra l'Università di Bologna e l'ELMS Webstore del progetto Microsoft MSDNAA.

La tesi è così suddivisa:

- Nel capitolo 1 si illustrano caratteristiche, protocolli e linguaggi utilizzati nei Web Services;

- Nel capitolo 2 si introduce il linguaggio JOLIE e l'architettura dell'interprete, con approfondimento alla parte rilevante alla realizzazione del modulo aggiuntivo;

- Nel Capitolo 3 si introduce il protocollo SSL/TLS, le politiche di sicurezza che adotta e le procedure di Handshake. Viene inoltre descritta l'API SSLEngine;

- Il capitolo 4 è dedicato all'implementazione del modulo HTTPS. Vengono descritte le modalità secondo le quali esso si integra nell'architettura di JOLIE esistente, la comunicazione con l'API SSLEngine, la descrizione del codice (ad alto livello) e la lista dei parametri di configurazione del modulo;

- Il Capitolo 5 riguarda il caso d'uso che utilizza il modulo HTTPS, con il quale si crea un servizio per l'autenticazione integrata tra l'Università di Bologna e il sito Microsoft ELMS-MSDNAA.

Capitolo 1

I Web Service

Secondo la definizione utilizzata dal consorzio W3C[1], un *Web Service* è un'applicazione progettata in modo da garantire l'interoperabilità tra elaboratori in una rete, in maniera indipendente dalla piattaforma operativa delle macchine, dal tipo di rete e dai protocolli utilizzati.

E' corredato da un'interfaccia descritta in un formato interpretabile dall'elaboratore (nello specifico WSDL[2]) che specifica agli altri sistemi le proprie modalità di interazione. La comunicazione con il servizio avviene tramite l'invio di messaggi SOAP[3], tipicamente trasportati tramite HTTP[4] o altri protocolli standard del Web quali SMNP[5] e FTP[6].

I messaggi SOAP sono di natura testuale, costruiti con tag XML[7]. Il servizio può essere pubblicato in un apposito database centralizzato chiamato

[1]World Wide Web Consortium, associazione con lo scopo di migliorare gli esistenti protocolli e linguaggi per il WWW e di aiutare il Web a sviluppare tutte le sue potenzialità.

[2]Web Service Description Language, per la produzione di documenti che descrivono formalmente le interfacce dei Web Service, per supportare l'interoperabilità.

[3]Simple Object Access Protocol

[4]Hyper Text Transfer Protocol, principale protocollo di trasferimento ipertesti di internet, alla base del World Wide Web assieme al linguaggio di mark-up HTML.

[5]Simple Mail Transfer Protocol, protocollo usato per la trasmissione in Internet di e-mail.

[6]File Transfer Protocol, protocollo utilizzato principalmente il download di file.

[7]eXtensible Markup Language.

UDDI[8] e messo a disposizione della rete.

Il consorzio OASIS[9] e W3C sono responsabili delle architetture e della standardizzazione dei *Web Service*. Il consorzio WS-I[10] crea e standardizza collezioni di protocolli (*profiles*), ad esempio SOAP 1.1.

Il documento *Web Service Architecture* [WSA] di W3C definisce l'architettura dei Web Services:

- **Service Requestor** Utilizzatore del Servizio
- **Service Provider** Entità che mette a disposizione il Servizio
- **Service Registry** Database centralizzato dei servizi

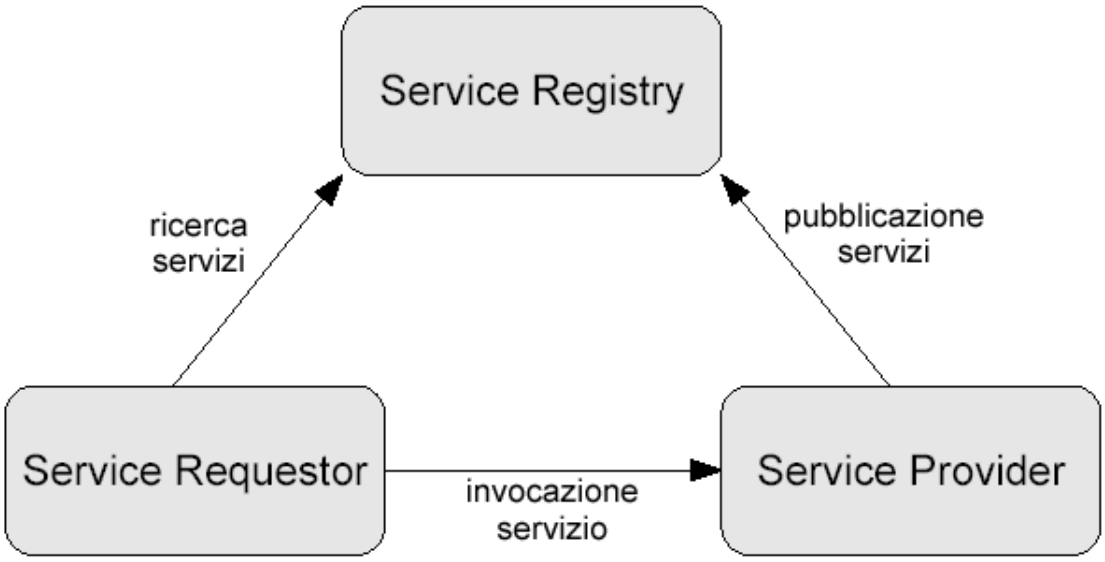


Figura 1.1: L'architettura dei Web Service: le tre entità principali

Il *Service Provider* offre alla rete il Servizio con la propria interfaccia e lo pubblica nel *Service Registry*.

[8] Universal Description Discovery and Integration, descritto nella sezione 1.1.4.

[9] Organization for the Advancement of Structured Information Standards.

[10] Web Services Interoperability Organization, gestito da IBM, Microsoft, BEA Systems, SAP, Oracle, Fujitsu, Hewlett-Packard, Intel, Sun Microsystems e webMethods.

Il *Service Requestor* ricerca il servizio desiderato nel *Service Registry*, ne preleva l'interfaccia e apprende da essa le modalità di interazione.
Il *Service Requestor* interagisce con il *Service Provider* invocando ed utilizzando il servizio.

1.1 Stato dell'arte: protocolli

Un Web Service è realizzato con una pila protocollare, ovvero un insieme di protocolli gerarchico, in questo caso suddivisibile in quattro categorie. Esse sono le seguenti:

1.1.1 Protocolli di trasporto

Il protocollo di trasporto del Servizio è responsabile della trasmissione e consegna dei messaggi sulla rete. Il protocollo più utilizzato è HTTP, meno frequentemente sono utilizzati SMTP, FTP, Jabber XMPP e BEEP12[11].

1.1.2 Messaggi

I messaggi scambiati tra i Web Service sono testuali e strutturati, per questo motivo è stato scelto il formato XML. Essi sono costruiti secondo le regole del protocollo SOAP, piu' raramente si utilizza anche JAX-RPC [JAX], XML-RPC [XML] o REST13[12].

Il protocollo SOAP

Acronimo di *Simple Object Access Protocol* [SOA] , è nato per lo scambio di messaggi tra componenti software. Il messaggio SOAP è di natura testuale e in formato XML. La struttura comprende:

[11] Blocks eXtensible eXchange Protocol Framework.

[12] Representational State Transfer, alternativa più leggera e facile da leggere e programmare di SOAP, a scapito di regole strutturali meno rigide e mancanza di strumenti adatti allo sviluppo.

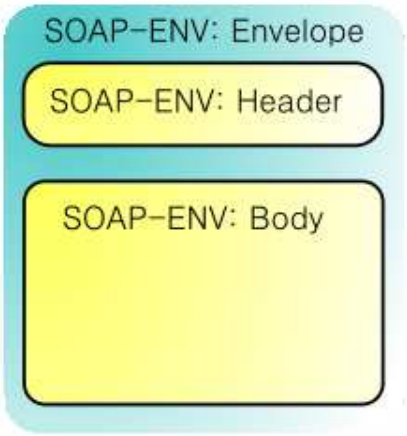


Figura 1.2: Struttura di un messaggio SOAP

Envelope è il contenitore del messaggio SOAP, contiene al suo interno *header* e *body*;

Header contiene informazioni riguardo routing, sicurezza, e transazioni;

Body contiene il contenuto informativo vero e proprio;

1.1.3 Descrizione del servizio: WSDL

Il *descrittore del servizio* è un documento pubblico che descrive formalmente le modalità di interazione con lo stesso. E' scritto in un formato automaticamente interpretabile dall'elaboratore. La descrizione riguarda le operazioni offerte dal servizio, i protocolli utilizzati, i vincoli da soddisfare, il formato dei messaggi in ingresso e uscita, e la locazione URI[13] di accesso al servizio descritto.

[13]Uniform Resource Identifier, identifica univocamente una risorsa generica.

Il linguaggio utilizzato per la stesura del documento è WSDL [WSD], che è l'unico attualmente standardizzato da W3C. E' in formato XML. La struttura del documento WSDL prevede la definizione di tipi, messaggi, operazioni, e protocolli usati. Essi vengono specificati all'interno di appositi tag XML. Sono i seguenti:

- **types** contiene le definizioni, in formato XSD, dei tipi di dati utilizzati dal Servizio;
- **message** definisce il formato dei dati utilizzati dalle operazioni, utilizzando anche i tipi definiti nell'elemento *types*;
- **portType** descrive le operazioni messe a disposizione dal Servizio. Ogni operazione è costituita da un insieme di messaggi (definiti in message) del tipo *Input*, *Output* o *Fault*.
- **binding** specifica regole di costruzione dei messaggi da scambiare e protocolli da utilizzare.
- **service** riporta locazione, porta e *binding* per l'accesso al Servizio.

Struttura di un documento WSDL:

```
<definitions>

  <types>

  [... definizione dei tipi in formato XSD...]

    <xsd:schema targetNamespace=[...] xmlns:xsd=[...]>
       <xsd:element [...]>
       [...]
       </xsd:element>
```

```
        [...]

    </xsd:schema>

  </types>

  <message>
  [... definizione messaggio...]
  </message>

  <portType>

  [... definizione delle operazioni...]
    <operation [...]>
    </operation>

    [...]

  </portType>

  <binding>
  [... definizione vincoli e protocolli dei messaggi...]
  </binding>

  <service>
  [... definizione servizio e URI di accesso...]
  </service>

</definitions>
```

Il Service Requestor (utilizzatore del Servizio) invoca una o più operazioni messe a disposizione dal Servizio. Ciascuna operazione è costituita da una serie di messaggi prestabiliti. Essi possono essere dei seguenti tipi:

- **Input** mandato dal Service Requestor al Servizio
- **Output** mandato dal Servizio al Service Requestor
- **Fault** messaggio di risposta che comunica la mancata riuscita di un'operazione

Lo standard W3C prevede quattro tipi di operazioni:

One-Way

Il Servizio riceve un messaggio. Quest'operazione è formalizzata nel WSDL come di seguito:

```
<wsdl:portType [...] >
  <wsdl:operation name="nmtoken">
    <wsdl:input name="nmtoken"? message="requestType"/>
  </wsdl:operation>
</wsdl:portType >
```

RequestType è il tipo di dato inviato, specificato precedentemente in *types*.

Notification

Il Servizio invia un messaggio al Service Requestor. La descrizione è analoga:

```
<wsdl:portType [...] >
  <wsdl:operation name="nmtoken">
```

```
    <wsdl:output name="nmtoken"? message="requestType"/>
  </wsdl:operation>
</wsdl:portType >
```

Figura 1.3: Schema di un'operazione One-Way o Notification

Request-Response

Il Servizio riceve un messaggio, esegue le sue computazioni e manda un messaggio di risposta al Service Requestor. Esso sarà di tipo *Output* o *Fault* a seconda della riuscita o meno dell'operazione.

```
<wsdl:portType [...] >
 <wsdl:operation name="nmtoken">
   <wsdl:input name="nmtoken"? message="requestType"/>
   <wsdl:output name="nmtoken"? message="responseType"/>
   <wsdl:fault name="nmtoken" message="faultType"/>
 </wsdl:operation>
</wsdl:portType>
```

Solicit-Response

Operazione duale al Request-Response, in questo caso è il Servizio che invia un messaggio al Service Requestor, il quale produce un messaggio di risposta di tipo Output, oppure Fault nel caso in cui si sia verificato un errore.

```
<wsdl:portType [...] >
  <wsdl:operation name="nmtoken>
    <wsdl:output name="nmtoken"? message="requestType"/>
    <wsdl:input name="nmtoken"? message="responseType"/>
    <wsdl:fault name="nmtoken" message="faultType"/>
  </wsdl:operation>
</wsdl:portType >
```

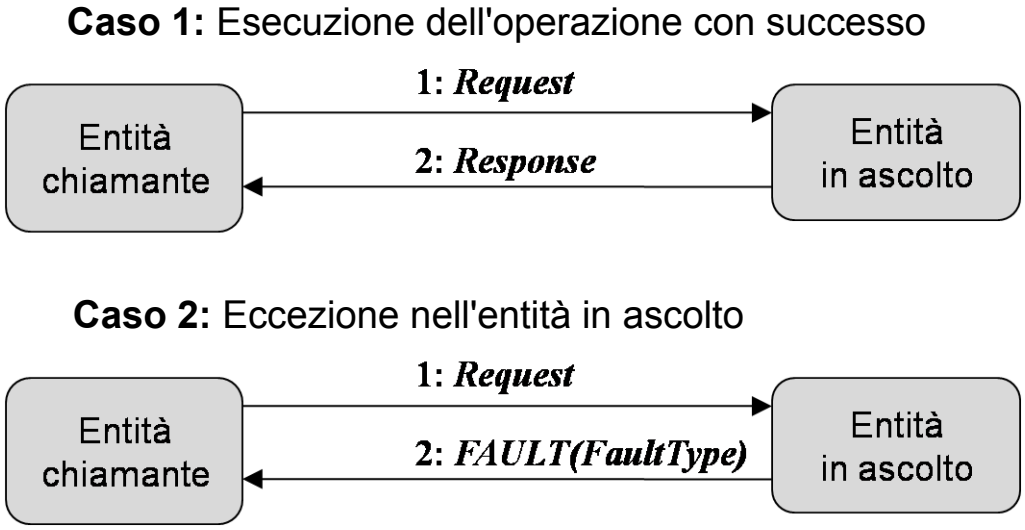


Figura 1.4: Schema di un'operazione Request-Response o Solicit-Response

1.1.4 Registro di servizi: UDDI

L'ampio numero di servizi presenti sulla rete Internet ha reso necessaria la creazione di un registro centralizzato dei servizi. A tale scopo è stato realizzato UDDI (Universal Description Discovery and Integration) [UDDa], un'iniziativa *Open* supportata dal consorzio OASIS.

Un servizio che viene registrato in UDDI è disponibile alla libera fruizione dell'intera rete Internet. La registrazione comprende la pubblicazione del descrittore di interfaccia (documento WSDL), che viene fornito all'utilizzatore del servizio.

UDDI consente di effettuare operazioni di ricerca del servizio desiderato, che avvengono tramite interrogazioni per mezzo di messaggi SOAP; tuttavia sono presenti in rete numerosi siti di *UDDI Browsing* che permettono la ricerca dei servizi attraverso pagine Web [UDDb].

Ogni registrazione UDDI è corredata dai seguenti componenti:

- Pagine bianche (*White Pages*): indirizzo, contatti (dell'azienda che offre uno o più servizi) e identificativi;
- Pagine gialle (*Yellow Pages*): categorizzazione dei servizi basata su tassonomie standardizzate;
- Pagine verdi (*Green Pages*): informazioni (tecniche) dei servizi fornite dall'azienda.

1.2 SOA

Una Service-Oriented Architecture (SOA) [New05] è un'architettura software che garantisce l'interoperabilità tra diversi sistemi mediante l'uso dei Web Service. Ogni singola applicazione può essere vista come un componente del processo di business da eseguire. Il consorzio OASIS definisce la SOA così:

> Un paradigma per l'organizzazione e l'utilizzazione delle risorse distribuite che possono essere sotto il controllo di domini di proprietà differenti. Fornisce un mezzo uniforme per offrire, scoprire, interagire ed usare le capacità di produrre gli effetti voluti consistentemente con presupposti e aspettative misurabili.

Tale architettura prevede il front-end dell'applicazione, il canale di collegamento tra i vari servizi, e il servizio stesso. Il servizio, oltre alla specifica dei servizi già vista, ha una logica applicativa e i dati da elaborare. Oltre all'interoperabilità tra sistemi etereogenei, un altro obiettivo di SOA è di facilitare gli adattamenti al cambiamento delle esigenze di business, astraendo dai dettagli implementativi e riducendo al minimo la dipendenza tra i diversi componenti del sistema distribuito (*accoppiamento*).

Un'applicazione distribuita puo' essere coordinata in due modi [NB06b]:

Orchestrazione

Nell'Orchestrazione vi è un singolo processo produttivo che interagisce con i Web Service interni ed esterni. Il processo comprende un *workflow* in cui è specificato il modo in cui i servizi comunicano tra di loro (ovvero la logica dei messaggi che si scambiano) per arrivare al completamento del processo. L'esecuzione del processo ed il controllo del workflow (sequenza e ordine delle operazioni) è affidato ad un singolo gestore logico. Tra i linguaggi piu' comuni per creare orchestratori di servizi si citano BPEL[14] for Web Services (WS-BPEL) [BPE] e JOLIE[15] [JOL].

Coreografia

Con Coreografia [NBZ05] si intende la descrizione ad alto livello delle interazioni tra i servizi, con la quale si definiscono globalmente le regole e ruoli tramite i quali i messaggi vengono scambiati. Nella Coreografia ciascuna entità descrive una parte del processo produttivo ed è responsabile della

[14] Business Process Execution Language.

[15] Descritto nel capitolo 2

parte che svolge. Nessuna delle entità possiede una visione globale del processo, ciascuna collabora alla sua esecuzione. Il linguaggio attualmente più accreditato per la coreografia è WS-CDL[16] [WSC].

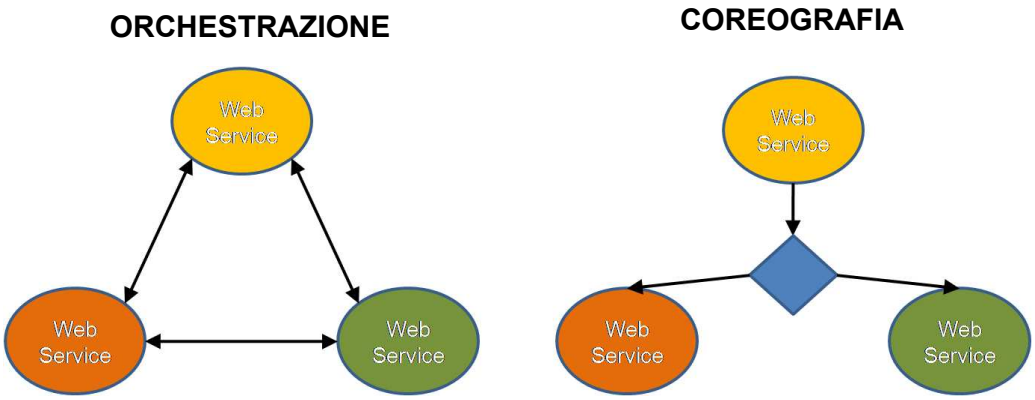


Figura 1.5: Orchestrazione e Coreografia: Schemi a confronto

1.3 SOC

L'avvento dei Web Service ha portato la creazione di un nuovo paradigma di programmazione: il Service Oriented Computing (SOC). Il servizio è considerato elemento centrale nella programmazione di applicazioni e nell'integrazione di sistemi informativi. SOC si basa sulla composizione di servizi e architetture distribuite. Possiamo ora definire un Web Service anche come una tecnologia SOC basata su XML.

[16]Web Services Choreography Description Language, linguaggio XML-based standardizzato da W3C.

Capitolo 2

JOLIE

2.1 Introduzione

JOLIE (Java Orchestration Language Interpreter Engine) [FM06] è l'interprete per l'omonimo linguaggio di programmazione progettato specificatamente per architetture orientate ai servizi (SOA) e loro orchestrazione.

JOLIE pone le sue basi teoriche sul calcolo formale SOCK (Service Oriented Computing Kernel), sviluppato per formalizzare tutte le primitive essenziali per la descrizione dei meccanismi del paradigma SOC, e di cui Jolie ne rappresenta un'implementazione concreta. E' stato realizzato interamente in Java ed è distribuito in formato *Open*.

La sintassi del linguaggio è simile al C e al Java, la progettazione di un servizio avviene ad alto livello, astraendo dalle problematiche di comunicazione. Tale astrazione consente la programmazione di un servizio elementare in una decina di righe di codice, essendo tuttavia presenti numerosi strumenti per la creazione di servizi più complessi.

Il linguaggio prevede costrutti per chiamata e ricezione di operazioni, definizione di porte, identificativi sessioni, definizione variabili strutturate, assegnazioni, operazioni, definizione procedure e altri meccanismi utili per l'orchestrazione di servizi, deducibili dalle specifiche WSDL.

E' altresì possibile l'esecuzione di codice JAVA esterno, implementato come JavaService, dalla propria applicazione Jolie.

JOLIE supporta anche librerie per connessione a database server[1] istruzioni per input e output da console, invio e-mail tramite SMTP, lancio di processi nel sistema, elaborazione di stringhe, funzioni per gestione date e orari, e librerie per la comunicazione Bluetooth, richiamabili tramite semplici operazioni predichiarate. E' possibile incorporare applicazioni GWT[2], servizi (embedding) e interfacce grafiche, effettuare redirecting.

JOLIE funziona su differenti piattaforme (qualunque supporti Java), può utilizzare tutte le librerie Java ed integrare applicazioni legacy.

L'architettura di JOLIE è flessibile, progettata per favorire lo sviluppo di moduli ed estensioni per l'aggiunta di nuove features o il supporto di protocolli futuri.

2.1.1 La teoria dietro a JOLIE: SOCK

Jolie fonda la sua semantica sul solido modello matematico SOCK[3] [NB06a]. Si elencano di seguito le caratteristiche fondamentali:

- Modalità di interazione: comportamento come da specifiche WSDL;
- Modello di composizione workflow: la composizione avviene tramite workflows;
- Funzionalità di computazione: il servizio è in grado di eseguire ogni tipo di computazione funzionale;
- Modello di esecuzione: descrive le modalità di esecuzione del servizio che puo' avvenire in modo sequenziale o concorrente;
- Sessioni correlate: possibilità per un servizio di mantenere più sessioni per diversi partecipanti (Client);

[1] Come Ms SQL Server, PostgreSQL, MySQL, Java DB.
[2] Google Web Toolkit, http://code.google.com/webtoolkit/ .
[3] Service Oriented Computing Kernel.

- Stato del servizio: consiste nello stato di esecuzione del servizio, che può essere condiviso tra le varie sessioni.

SOCK organizza le precedenti caratteristiche in 3 livelli:

- Livello di comportamento (**Behaviour Layer**): è relativo alla singola sessione del Client;
- Livello di servizio (**Service Layer**: si occupa del servizio e della sua esecuzione;
- Livello di rete (**Network Layer**): si occupa dell'aggregazione dei servizi.

Un servizio puo' essere avviato in modalità concorrente o, persistente o non persistente, dando luogo alle seguenti modalità di esecuzione:

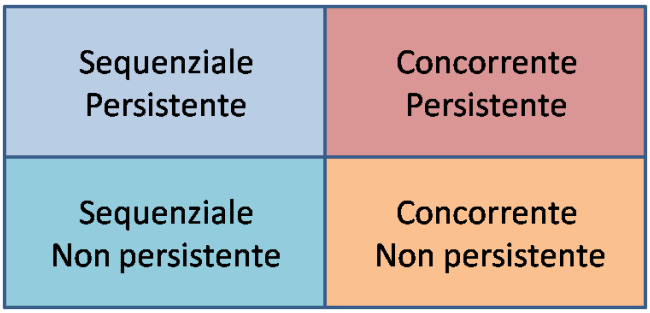


Figura 2.1: SOCK: Modalità di esecuzione dei servizi

Nel caso di esecuzione concorrente, il linguaggio JOLIE prevede apposite primitive per l'identificazione delle sessioni. La persistenza o meno specifica il mantenimento dei dati per le sessioni successive o la loro distruzione all'uscita dalla sessione.

2.1.2 Servizi e operazioni

Ogni servizio comunica tramite l'invio e la ricezione di messaggi, che sono rispettivamente operazioni di input e operazioni di output.

Le specifiche WSDL definiscono 2 operazioni di input: *One-Way* e *Request-Response*. La prima si limita ad inviare un messaggio, la seconda invia il messaggio e si mette in attesa di un messaggio di risposta, che può anche essere di tipo Fault in caso di mancata riuscita dell'operazione.

Per quanto riguarda l'output sono definite 2 operazioni: *Notification* e *Solicit-Response*, che rappresentano la versione duale delle operazioni di input (per approfondimenti vedi la sezione 1.1.3).

In Jolie sono tutte supportate, tuttavia sono presenti ad un livello di astrazione maggiore. Le operazioni sono di esclusivo tipo One-Way o Request-Response e vengono specificate all'interno di una porta, che può essere di input o di output. La Notification si ottiene definendo una One-Way in una porta di output, e l'engine di Jolie provvederà ad eseguire l'operazione corretta. La stessa meccanica si utilizza per realizzare la Solicit-Response.

2.1.3 Protocolli e mezzi di comunicazione supportati

JOLIE supporta attualmente i seguenti protocolli di comunicazione: SOAP, HTTP, BTL2CAP[4] e in più SODEP[5] [SOD], un protocollo nativo creato appositamente allo scopo di fornire un sistema efficiente e lightweight per le comunicazioni tra servizi JOLIE. Esso permette inoltre il trasporto di tutte le strutture dati supportate dal linguaggio.

I mezzi di comunicazione utilizzabili sono: *Socket*, *File*, *Pipe*.

In JOLIE il comportamento dei servizi è indipendente sia dal punto di vista del protocollo e sia dal punto di vista del mezzo di comunicazione.

JOLIE solleva lo sviluppatore dall'oneroso compito di occuparsi della comunicazione e delle sue problematiche, è sufficiente impostare il mezzo di comunicazione e il protocollo desiderato e l'engine provvederà ad occuparsene.

I protocolli supportati attualmente sono in chiaro e human-readable, e in quanto tali risultano facilmente intercettabili da parti terze. Non vi è attual-

[4]Bluetooth Logical Link Control and Adaptation Protocol.

[5]Simple Operation Data Exchange Protocol.

mente alcuna misura in JOLIE atta a garantire la sicurezza e la riservatezza dei dati trasmessi. Il lavoro di questa tesi intende sopperire a tale carenza.

2.2 Architettura di JOLIE

In questo paragrafo verrà descritta l'architettura dell'interprete JOLIE [FM06].

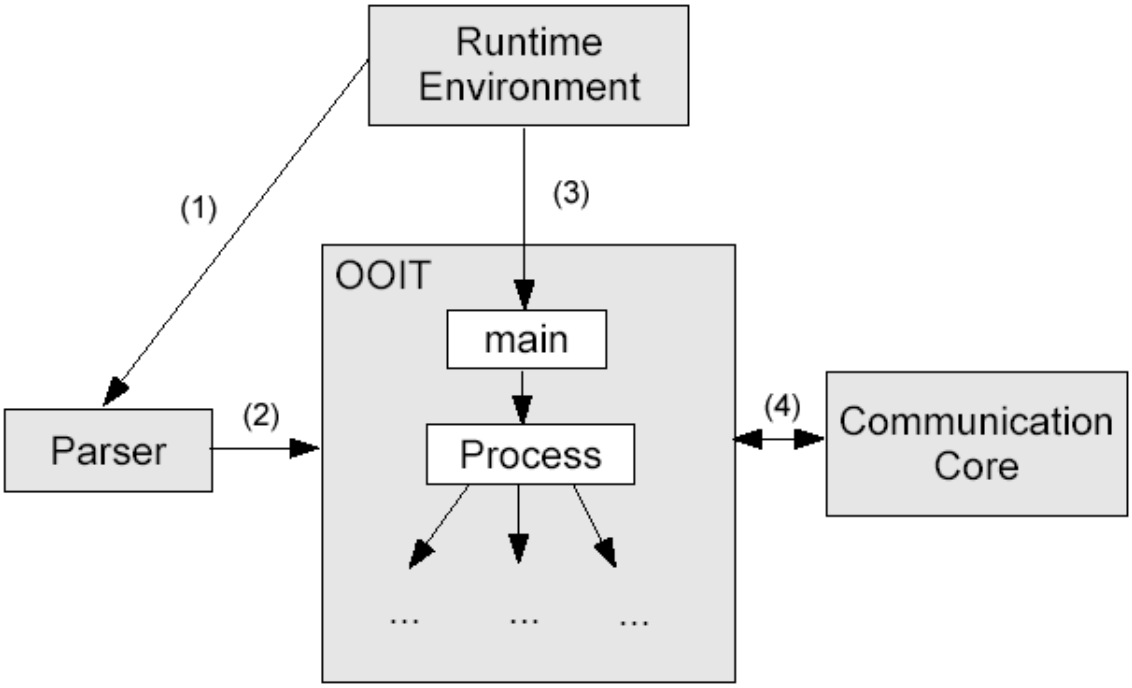


Figura 2.2: Architettura dell'interprete JOLIE

L'immagine descrive la struttura dell'interprete JOLIE e le relazioni tra le varie parti che la compongono. All'avvio JOLIE esegue i seguenti step numerati in figura:

(1) inizializzazione del Communication Core

(2) creazione di un'istanza del Parser

(3) creazione dell'*Object Oriented Interpretation Tree* (OOIT)

(4) invocazione del metodo **run()** del nodo radice dell'OOIT (che corrisponde al main).

Verranno adesso descritte ciascuna delle parti che compongono l'interprete.

2.2.1 Runtime Environment

L'ambiente di esecuzione di JOLIE gestisce la creazione di nuove sessioni, sincronizzazione dei processi e dello stato dei servizi, e l'interazione con gli altri componenti.

2.2.2 Parser e Object Oriented Interpretation Tree

L'implementazione dell'interprete si basa su un'infrastruttura object-oriented ad albero (OOIT) che viene creata in seguito al parsing del codice da eseguire, per mezzo del *parser* che è di tipo recursive-descendant (ricorsione sul discendente).

Ogni nodo contiene un oggetto della classe generica *Process* che rappresenta una porzione del codice da eseguire.

Un oggetto Process implementa il metodo run() e può essere di 2 tipologie, sottoclassi di Process:

Basic Process

Un Basic Process è un'oggetto composto da un singolo statement elementare di codice JOLIE, ad esempio un'assegnazione o un'operazione di input o di output. Il metodo *run()* da' luogo all'esecuzione dello statement.

Composite Process

Un Composite Process è un oggetto che si compone di altri oggetti Process. Essi possono essere eseguiti in parallelo, in sequenza o in scelta non-deterministica, a seconda di come è indicato nel codice sorgente. Il metodo *run()* esegue tale composizione e in secondo luogo invoca i metodi *run()* degli oggetti incapsulati.

Questo principio di incapsulamento è presente in tutto l'albero OOIT, pertanto è sufficiente invocare il metodo *run()* del nodo radice (che corrisponde al *main*) per far partire l'esecuzione dell'intero programma JOLIE.

2.2.3 Communication core

Il Communication Core si occupa della gestione della comunicazione tra i servizi separando gli altri componenti dalle eventuali problematiche che ne derivano.

Esso permette la comunicazione astrendo sia dal punto di vista del mezzo di comunicazione, che può essere Socket, File o Pipe, sia dal punto di vista del protocollo da utilizzare.

Tale astrazione è supportata per mezzo delle classi astratte *CommChannel* e *CommProtocol*; la prima rappresenta un canale di comunicazione, mentre la seconda si riferisce al protocollo da utilizzare.

Ciascuno dei medium di comunicazione supportati implementa una sottoclasse di *CommChannel* che corrisponde alla gestione di quel dato mezzo di comunicazione.

Allo stesso modo, ogni protocollo supportato in JOLIE implementa una sottoclasse di *CommProtocol*, che corrisponde alla gestione di quel protocollo. Per i protocolli, tale sottoclasse contiene al suo interno un oggetto *CommMessage*, che è il formato standard della comunicazione in JOLIE e rappresenta il messaggio da inviare o ricevuto.

All'avvio dell'interprete, per ogni canale di comunicazione da aprire il runtime environment istanzia un oggetto del sottotipo di *CommChannel* corrispondente, e vi associa un oggetto della sottoclasse di *CommProtocol* relativa al protocollo da utilizzare. Quando l'interprete deve inviare un messaggio, incapsula i dati applicazione all'interno di un messaggio generico *CommMessage*, ignorando protocolli e mezzi di comunicazione, e lo invia all'istanza di *CommChannel* che gestisce il canale di comunicazione destinatario. Tale istanza provvede ad incapsulare il *CommMessage* in un messaggio del tipo corrispondente al protocollo di comunicazione (ad esempio HTTP) del cana-

le che gestisce, e lo invia sul canale. In ricezione si applica il processo inverso.

Ad esempio, supponiamo di dover mandare un messaggio ad un server remoto (tramite Socket) con il protocollo HTTP. Il *runtime environment* inizializza un oggetto di tipo *SocketChannel* con i parametri per la connessione con l'host remoto (indirizzo IP e porta) ed un oggetto di tipo *HTTPProtocol*. La comunicazione verrà gestita leggendo e scrivendo *CommMessage* inviati e ricevuti tramite il *SocketChannel*.

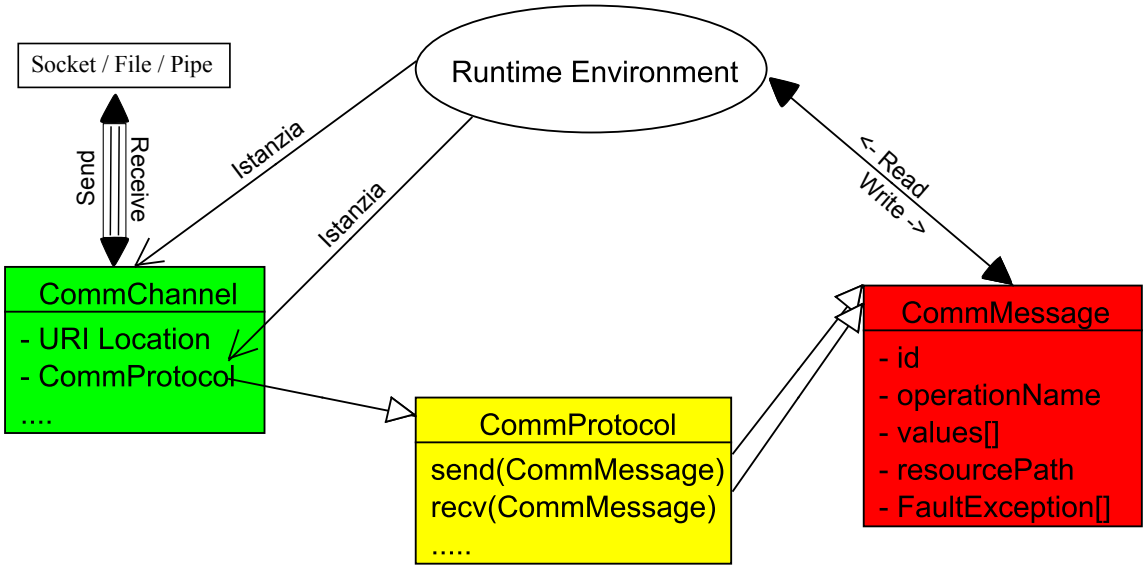


Figura 2.3: Relazioni tra CommChannel, CommProtocol, CommMessage e runtime environment.

Un oggetto *CommMessage* è composto da:

- ID del messaggio;
- Nome dell'operazione da eseguire;
- Un array di valori;
- Resource Path (utilizzato in caso di reindirizzamento);
- Lista (anche vuota) dei *Fault* che l'operazione può sollevare.

Una progettazione siffatta favorisce la modularità e l'estensibilità dei medium e dei protocolli supportati; il supporto di un nuovo protocollo avviene implementando una sottoclasse di *CommProtocol* apposita. Discorso analogo per un nuovo mezzo di comunicazione (in questo caso la superclasse di riferimento è *CommChannel*).

2.3 Il linguaggio Jolie

Questa sezione è dedicata alla descrizione delle componenti fondamentali del linguaggio JOLIE, e di tutte quelle presenti nel caso d'uso nel capitolo 5. La sintassi del linguaggio è simile a quella dei linguaggi C e Java, arricchita di costrutti aggiuntivi che permettono l'orchestrazione dei servizi e la gestione delle funzionalità offerte.

2.3.1 Dichiarazione interfacce, porte di input, porte di output, servizi in entrata

Per l'interazione tra i servizi, è necessario dichiarare le porte di input e di output che verranno utilizzate, ciascuna delle quali si riferisce ad una location che indica il protocollo e le operazioni disponibili.

JOLIE supporta la dichiarazione di gruppi di operazioni tramite la definizione di un'interfaccia che le contiene. Le porte di input e output possono elencare le singole operazioni supportate oppure indicare un'interfaccia. In quest'ultimo caso, le operazioni disponibili saranno quelle contenute nell'interfaccia.

Un'interfaccia si dichiara nel seguente modo:

```
interface <nomeInterfaccia>
{
  [OneWayOperation: <OnewayDecl_1>[,    <OnewayDecl_2>
```

```
    [, <OnewayDecl_N> ...] ] ]
  [RequestResponseOperation: <RRDecl_1>[, <RRDecl_2>
    [,<RRDecl_N> ...] ] ]
}
```

Dopo il tipo dell'operazione vanno indicati uno o più nomi di operazioni, il cui codice viene specificato in seguito. I tipi di operazioni previsti sono *One-Way* e *Request-Response*, la prima composta da un messaggio da inviare o ricevere (in base alla dichiarazione della porta) e la seconda che trasmette (o aspetta) inoltre il messaggio di risposta (o di *Fault*, nel caso di mancata riuscita dell'operazione).

Dichiarazione di un operazione One-Way

```
<RRDecl> ::= <nomeOperazione>
```

Dichiarazione di un operazione Request-Response

```
<RRDecl> ::= <nomeOperazione>
  [throws <nomeFault1> [<nomeFault2>[<nomeFaultN> ...]]]
```

Il costrutto throws indica la lista dei Fault (*nomeFault1*, ... *nomeFaultN*) che l'operazione può sollevare.

Una porta di input si dichiara nel seguente modo:

```
inputPort <nomePorta> {
  Location: <nomeLocation>
  Protocol: <nomeProtocollo>
  <operazioni o interfacce>
}
```

Analogamente, una porta di output si dichiara nel seguente modo:

```
inputPort <nomePorta> {
  [Location: <nomeLocation>]
  Protocol: <nomeProtocollo>
  <operazioni o interfacce>
}
```

Come locazione è possibile specificare un identificatore di locazione (variabile) precedentemente dichiarato oppure si può dichiarare direttamente con la sintassi:

```
<defLocation> ::= <id_location> =
    "<medium>://<hostname>:<port>"
```

in cui *id_location* rappresenta il nome identificativo e univoco della locazione, a cui viene assegnato il mezzo di trasmissione, il nome di rete e la porta. Esempi di locazioni possono essere:

```
file://config.ini
socket://localhost:1001
smtp://mail.server.com: 25
```

Le locazioni possono essere definite anche in modo dinamico, specificandone il valore successivamente alla dichiarazione della porta, con la sintassi:

```
<nomePorta>.location = <id_location>
```

La sintassi di dichiarazione delle operazioni direttamente nelle porte è la seguente:

```
<operazioni o interfacce> ::=
      [OneWayOperation: <OnewayDecl_1>
                  [, <OnewayDecl_N> ...] ]
      [RequestResponseOperation: <RRDecl_1>
                  [, <RRDecl_2>[,<RRDecl_N> ...] ] ]
```

La sintassi di dichiarazione delle interfacce (già dichiarate) nelle porte è invece la seguente:

```
<operazioni o interfacce> ::=
    Interfaces: <interfaccia1>
[, <interfaccia2>
[ ... , <interfacciaN>] ]
```

Le operazioni One-Way e Request-Response dichiarate nelle porte di output corrispondono rispettivamente alle operazioni Notification e Solicit-Response.

2.3.2 Procedure e procedura principale (main)

JOLIE supporta la dichiarazione di procedure, tuttavia non è possibile il passaggio di valori e variabili. La sintassi è la seguente:

```
define <nomeProcedura>
{
  <processo>
}
```

Per ogni servizio è necessario specificare la procedura principale (*main*) nel seguente modo:

```
main
{
  <processo>
}
```

La procedura *init* è una procedura speciale da dichiarare prima della procedura principale. Essa si utilizza per effettuare operazioni di inizializzazione e viene eseguita all'avvio del servizio prima del lancio di tutte le altre procedure.

2.3.3 Processi, istruzioni e composizione

Un processo può essere l'insieme di più processi eseguiti in sequenza:

```
<processo1> ; <processo2> ; ... ; <processoN>
```

oppure processi eseguiti in parallelo:

```
<processo1> | <processo2> | ... | <processoN>
```

oppure processi in scelta non-deterministica:

```
[operazione1(...)] {codiceRamo1}
[operazione2(...)(...){procInterno2}] {codiceRamo2}
...
```

```
[operazioneN(...)] {codiceRamoN}
```

oppure un'istruzione. Nell'esecuzione non-deterministica viene avviato in modo deterministico il relativo processo. Nel caso ci siano operazioni Request-Response in ascolto nelle varie scelte, verrà avviato il primo processo che riceve un messaggio. Negli esempi sopracitati, verranno eseguiti prima i processi interni, poi i codici del ramo relativo. Un'istruzione può essere un assegnamento tra variabili, una chiamata di operazione (Notification o Solicit-Response), una chiamata di procedura, un blocco condizionale di istruzioni, un'espressione e altro[6].

2.3.4 Ricezione di operazioni

Il comando per un'operazione One-Way ha la seguente sintassi:

```
<nomeOperazione>(<oggettoRequest>)
```

Questo comando se utilizzato in una porta input fa' attendere il processo finché non arriva un messaggio. Quando un'operazione Notification invia il messaggio e questo viene ricevuto correttamente dalla One-Way, esso è reso disponibile nella variabile *oggettoRequest* e sarà utilizzabile successivamente.

Il comando per un'operazione Request-Response ha la seguente sintassi:

```
<nomeOperazione>(<oggettoRequest>)(<oggettoResponse>) {
    <processoDiRisposta>
}
```

[6] si omettono alcuni costrutti JOLIE in quanto non rilevanti per questa tesi. Per approfondimenti visitare la risorsa [JOL].

Tale comando in una porta di input fa attendere il processo fino alla ricezione di un messaggio. Quando l'operazione Solicit-Response dell'altro peer invia il messaggio ed esso viene ricevuto correttamente, è reso disponibile nella variabile *oggettoRequest* e sarà utilizzabile dentro *processoDiRisposta*. Al termine del processo di risposta viene inviato un messaggio all'operazione Solicit-Response chiamante. Tale messaggio è il contenuto della variabile *oggettoResponse* oppure un Fault nel caso di mancata riuscita dell'operazione.

Il funzionamento è analogo nel caso delle operazioni duali, Notification e Solicit-Response, e verrà descritto sul paragrafo successivo.

2.3.5 Chiamata di operazioni

Un'operazione di tipo Notification ha la seguente sintassi:

```
<nomeOperazione>@<nomePortaOutput>
            (<oggettoRequestDaInviare>)
```

dove *nomePortaOutput* è l'identificativo della porta di output, che deve essere obbligatoriamente e precedentemente dichiarata. Esso deve riferirsi ad una locazione valida. Il messaggio inviato è quello contenuto nell'oggetto *oggettoRequestDaInviare*. L'operazione Notification non riceve nessuna risposta pertanto non è possibile verificare dallo stesso servizio se il messaggio è stato inviato con successo.

Un comando Solicit-Response ha la sintassi:

```
<nomeOperazione>@<nomePortaOutput>
            (<requestDaInviare>)(<risposta>)
```

dove *nomePortaOutput* è l'identificativo della porta di output, che deve es-

sere obbligatoriamente e precedentemente dichiarata. Essa deve riferirsi ad una locazione valida. Il messaggio inviato è quello contenuto nell'oggetto *requestDaInviare*. Il comando attende una risposta dall'operazione Request-Response corrispondente nell'altro peer. Tale risposta sarà disponibile tramite l'oggetto *risposta*. Il messaggio di risposta puo' anche essere di tipo Fault nel caso di mancata riuscita dell'operazione. La sintassi e la semantica di cattura dei Fault è descritta nel paragrafo 2.3.7.

2.3.6 Utilizzo delle strutture dati in JOLIE

In JOLIE le variabili non necessitano di dichiarazione e inizializzazione. Il tipo delle variabili è implicitamente determinato in base al contenuto assegnato ed è immediatamente utilizzabile. Notare che, nella sintassi degli esempi precedentemente riportati, le variabili *request* e *response* non sono infatti dichiarate.

Una variabile contiene un tipo di dato semplice (cioè un valore di tipo *stringa* o *numerico* che può essere direttamente assegnato). Tale valore verrà d'ora in poi chiamato anche *valore nativo* della variabile. Una variabile può contenere, indipendentemente dal valore nativo, anche:

- una lista ordinata (o vettore) di sottoelementi;
- una serie di sottoelementi reperibili tramite identificatore (chiave).

Un *sottoelemento* può avere la struttura, appena descritta, di una variabile. In questo modo è possibile rappresentare i dati ad albero senza limiti di profondità utilizzando tali strutture. E' possibile creare variabili che non hanno alcun valore nativo (o non è ancora stato impostato). In questo caso se la variabile ha almeno un sottoelemento, il valore nativo è impostato su *void*, se non contiene sottoelementi esso diventa *undefined*.

Per riempire e usare la lista ordinata si può procedere nel seguente modo:

```
var[1] = "primo elemento";
var[2] = "secondo elemento";
var[3] = "terzo elemento";
var[4] = "quarto elemento";
for (i=0, i<#var, i++)
{
 println@Console( var[i] )
}
```

Il precedente esempio stampa a video le quattro stringhe. #var è la lunghezza del vettore *var* e corrisponde al valore 4. Per inserire valori reperibili tramite identificatore (chiave) si utilizza la seguente sintassi:

```
<variabile>.<identificatoreChiave> = <valoreDaAssegnare>
```

Un valore può avere la struttura di una variabile, quindi può contenere altri valori reperibili tramite identificatori. Ad esempio:

```
var.chiave1.chiave2.chiave3 = 10;
var.chiave1.chiave2 = "stringa2";
```

Notare che *var* è una variabile con tipo nativo vuoto (*void*) che contiene il sottoelemento *chiave1*, il quale contiene il sottoelemento *chiave2*. Quest'ultimo ha tipo nativo *stringa* e contiene anche il sottoelemento *chiave3*, di tipo nativo *intero*. Con riferimento alla struttura ad albero di tale variabile, *var* è la radice, *chiave1* è un sottoelemento figlio di var, *chiave2* è un sottoelemento (livello 2 nell'albero) figlio di chiave1, *chiave3* è un sottoelemento (livello

3) figlio di chiave2. Per copiare il contenuto della variabile (o sottoelemento a qualsiasi livello) B nella variabile (o sottoelemento a qualsiasi livello) A, si utilizza l'operatore di copia << nel seguente modo:

```
A << B
```

2.3.7 Cattura fault dinamici

Una tematica importante nell'ambito dei servizi distribuiti è la gestione dei Fault, con il quale il fallimento di un'operazione di un servizio deve essere segnalata e propagata automaticamente ai servizi interessati. I meccanismi per gestire i Fault sono:

- Fault handlers;
- Termination handlers;
- Compensation handlers.

I *Fault handler* (gestori di errori) sono usati per gestire direttamente i Fault dei processi interni allo scope, i *Termination handler* si usano per catturare i Fault dai processi paralleli o esterni, i *Compensation handlers* sono invece usati per annullare gli effetti di un processo in cui è stato sollevato un Fault e il cui scope è già terminato.

L'installazione di un Fault handler in JOLIE ha la seguente sintassi:

```
install(<nomeFault> => <processoHandler>)
```

In JOLIE il meccanismo di cattura dei Fault avviene tramite un meccanismo di *scoping*. Supponendo di aver inserito tale istruzione nella procedura *main* e

nello scope principale, se si verifica un Fault di nome *nomeFault*, l'esecuzione della procedura main viene sospesa, ed è lanciato il *processoHandler*.

La sintassi è la seguente:

```
[<installazioneHandlerLivelloSuperiore>]
scope (<nomeScope>)
{
  [<installazioneHandler>]
  <istruzioni che potrebbero causare Fault>
}
<istruzioniSuccessive>
```

Se all'interno dello scope viene lanciato un Fault, viene prima cercato un handler nello scope corrente. Se viene trovato (*nomeScope* nell'esempio), viene lanciato il relativo handler e poi il servizio continua l'esecuzione con le istruzioni successive al di fuori dallo scope. Se non viene trovato l'handler corrispondente dentro lo scope, la ricerca prosegue ricorsivamente nello scope superiore (cioè che comprende lo scope corrente), e fino ad arrivare al livello del main. In quest'ultimo caso si interrompe l'esecuzione dello scope main. Nell'esempio, se *nomeScope* non fosse dichiarato o non fosse relativo al Fault appena lanciato, viene controllato se con *installazioneHandlerLivelloSuperiore* si sia creato un handler che può gestire tale Fault. Risalendo viene poi cercato l'handler, e quando viene trovato, è avviata la procedura handler e l'esecuzione dello scope corrente (cioè a livello dell'installazione del Fault che ha effttuato la cattura, eventualmente al livello main) si interrompe.

L'installazione di un Termination handler si effettua nel seguente modo:

```
install(this => <processoHandler>)
```

L'installazione di un Compensation handler si effettua invece nel seguente

modo:

```
install(<nomeFault> => comp(<idScope>) )
```

Con tale comando, come handler per *nomeFault* verrà usato l'ultimo handler installato nello scope *idScope*. E' possibile anche installare un handler che aggiunga delle istruzioni all'handler precedentemente installato. La sintassi è la seguente:

```
install(<nomeFault> => cH;<processoHandler>)
```

La sintassi per lanciare direttamente un Fault è la seguente:

```
throw(<nomeFault>[, <variabile con informazioni fault>])
```

Installazione dinamica dei Fault handler

Nei linguaggi di orchestrazione normalmente è possibile installare i Fault handler solo in modo statico. JOLIE supporta anche l'installazione dinamica a runtime di tali handler [FMZ06]. In tale modo è cioè possibile settare un gestore di errore in fase di esecuzione in qualsiasi parte del codice.

Gestione dei Fault nelle operazioni

Le operazioni Solicit-Response e Request-Response possono generare Fault. Un Fault generato nel processo di risposta di una operazione Request-Response viene automaticamente inserito nel messaggio di risposta della Solicit-Response.

Se essa era contenuta a sua volta nel processo di risposta di un'altra operazione Request-Response, il Fault viene inserito anche in tale messaggio di risposta.

In questo modo il Fault si propaga in modo da segnalare l'eccezione avvenuta a tutti i servizi interessati. L'ultimo processo a cui arriva il Fault può gestirlo localmente, ad esempio segnalando all'utente l'errore.

Dal lato della Solicit-Response, se dopo l'invio del messaggio il processo viene terminato (ad esempio per terminazione di un processo parallelo), prima che il processo venga terminato, la Solicit-Response viene forzata a ricevere in ogni caso il messaggio. In questo modo il processo Request-Response in ascolto non viene bloccato.

Se nella Solicit-Response il Fault viene generato prima dell'invio del messaggio, la Request-Response non è ovviamente coinvolta in quanto la Solicit-Response termina prima. Anche se il Fault avviene dopo la ricezione, la Request-Response non è coinvolta in quanto ha già spedito il messaggio di risposta.

Capitolo 3

SSL / TLS

3.1 Introduzione

I dati che attraversano una rete possono essere soggetti ad intercettazione ad opera di terzi parti non autorizzate. Esse entrerebbero così in possesso di informazioni riservate.

La necessità di mantenere una comunicazione confidenziale è uno scenario molto comune, specie quando i dati da trasmettere riguardano dati sensibili come password, dati della carta di credito, informazioni bancarie.

Tale esigenza ha reso necessario l'impiego di opportune tecnologie (dapprima utilizzate solo per scopi militari) per la protezione e la salvaguardia dei dati da terze parti in ascolto non autorizzate. E' altresì importante assicurarsi che i dati non siano stati alterati durante il trasporto, sia accidentalmente che intenzionalmente.

Una connessione è definita *sicura* quando riesce a garantire le seguenti 3 proprietà [DS04]:

- **Riservatezza della comunicazione**: Solo i componenti della comunicazione devono essere in grado di ricevere ed interpretare i dati. A garanzia di questo, i dati vengono resi illeggibili ad eventuali terze parti in ascolto.

- **Autenticazione**: Tutti gli interlocutori devono assicurarsi dell'identità delle controparti. Un utente malintenzionato potrebbe impersonare l'identità dell'interlocutore aspettato e ricevere informazioni riservate. E' necessario adottare degli strumenti per certificare l'identità delle parti remote.

- **Integrità dei dati**: I dati devono giungere a destinazione senza alterazioni. Esse possono essere causate da errori durante la transmissione dei dati oppure da terze parti in ascolto per scopi illeciti. Il ricevente pertanto otterrebbe delle informazioni corrotte (inutilizzabili) nel primo caso, o false, nel secondo. E' necessario introdurre degli strumenti che consentono di verificare se i dati abbiano subìto modifiche.

3.2 SSL / TLS

Il Secure Sockets Layer (SSL) è il protocollo più comunemente diffuso e utilizzato sul Web per garantire una comunicazione sicura.

E' un protocollo aperto e non proprietario sviluppato da *Netscape Communications*. L'ultima versione è la 3.0 rilasciata nel 1998 [SSL96].

La sua immediata diffusione lo ha rapidamente imposto come *standard de facto* per la sicurezza delle comunicazioni, al punto da essere analizzato e sottoposto a standardizzazione da IETF[1], che lo rinomina in TLS (Transport Secure Layer) nel 1999.

La versione 1.0 di TLS è definita dal documento RFC2246 [IET99], nella quale si identifica come SSL 3.1. L'ultima versione è la 1.2 rilasciata ad Agosto 2008 [IET08]. Rispetto al primo rilascio, la differenza con le versioni successive sono la risoluzione di problematiche di vulnerabilità, l'utilizzo di algoritmi di cifratura piu' sicuri e chiavi aventi con un maggiore numero di bit. Tali cambiamenti sono sufficienti a rendere impossibile la comunicazione tra host che implementano differenti versioni del protocollo. A soluzione di

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

ciò, TLS supporta il downgrade della comunicazione fino a SSL 3.0 (SSL 2.0 contiene numerose vulnerabilità ed è stato classificato come non sicuro).

Il protocollo SSL è composto da:

- **Protocollo SSL/TLS Handshake**, permette al Server ed al Client di autenticarsi a vicenda e di negoziare un algoritmo di crittografia e le relative chiavi (vedi paragrafo 3.2.4) prima che il livello di applicazione trasmetta o riceva il suo primo byte. Un vantaggio di SSL è la sua indipendenza dal protocollo di applicazione, in tal modo un protocollo di livello più alto può interfacciarsi sul protocollo SSL/TLS in modo trasparente;
- **Protocollo SSL Record**, è interfacciato su un protocollo di trasporto affidabile come il TCP[2]. Questo protocollo è usato per l'incapsulamento dei dati provenienti dai protocolli superiori.

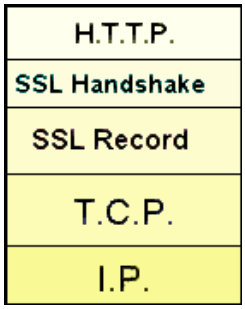


Figura 3.1: Collocamento di SSL/TLS tra l'Application Layer (nell'esempio HTTP) e il Transport Layer (TCP)

Il protocollo SSL/TLS trasmette **record**, che incapsulano i dati da scambiare. Ogni record può essere cifrato, compresso, corredato di un codice di autenticità (MAC, Message Authentication Code), e da *padding*[3], a seconda dello stato della connessione.

[2]Transmission Control Protocol.

[3]Codice riempitivo casuale, utilizzato per complicare la decifratura da parte di host non autorizzati.

Si approfondiscono 2 concetti espressi precedentemente: la crittografia e i certificati.

3.2.1 La crittografia

L'alterazione (encryption) è operata da appositi algoritmi che cifrano i dati in funzione di una chiave crittografica, che viene tenuta segreta. Tale chiave rende possibile operare il processo inverso (decryption) ed ottenere così i dati originari. I dati crittografati sono come la serratura di porta: solo la giusta chiave permette di aprirla e di accedervi.

La crittografia adempie al suo compito di creare dati offuscati, tuttavia al momento non esiste alcuna tecnica crittografica che si possa definire sicura in senso assoluto, in quanto il dato è reso sicuro solo per un certo arco temporale e non è teoricamente possibile garantire la durata della segretezza, che è dipendente dalla lunghezza della chiave. Tuttavia effettuare una decriptazione su dati cifrati mediante i più robusti algoritmi attuali richiederebbe delle quantità di tempo misurabili in secoli, con le potenze di calcolo di oggigiorno; pertanto la tecnica è largamente diffusa e considerata la piu' sicura.

Gli algoritmi di crittazione e decrittazione si dividono in 2 tipi:

Crittografia a chiave segreta

In questo tipo di crittografia esiste una sola chiave, utilizzata sia per la crittazione che per la decrittazione dei dati. E' chiamata anche *crittografia simmetrica*.

Il processo di crittazione e decrittazione dei dati è veloce, e gli algoritmi tutt'ora esistenti sono robusti e garantiscono un'ottima sicurezza dei dati.

Tuttavia, questo metodo pone il problema della comunicazione della chiave tra gli interlocutori, che deve essere in possesso di ciascuno.

Crittografia a chiave pubblica

Questa tipologia prevede 2 chiavi crittografiche, chiamate chiave pubblica e chiave privata, ed è detta anche *crittografia asimmetrica*.

Nel processo di cifratura viene utilizzata soltanto una delle due chiavi. Per attivare il processo inverso è necessario conoscere l'altra chiave.

In questo metodo, ciascuno dei componenti della comunicazione genera entrambe le chiavi, e comunica ai suoi interlocutori la propria chiave pubblica. La chiave privata non verrà mai comunicata. All'atto dell'invio di un messaggio, il mittente lo crittografa utilizzando la chiave pubblica ricevuta in precedenza. Il ricevente è l'unico in grado di decifrarlo in quanto è in esclusivo possesso della chiave privata, necessaria per decifrare ciò che è cifrato con la chiave pubblica.

In questo scenario il problema di una comunicazione sicura della chiave è risolto perché le chiavi comunicate sono sempre quelle pubbliche che servono solo a crittografare, ne consegue che un utente malitenzionato in ascolto non potrà decrittare i messaggi in quanto sprovvisto di chiave privata (che non viene mai comunicata).

Il rovescio della medaglia di questo metodo è l'estrema lentezza della computazione, che lo rendono utilizzabile esclusivamente per quantitativi di dati limitati.

3.2.2 I certificati a chiave pubblica

Un certificato a chiave pubblica (anche conosciuto come certificato digitale) [SUN06] è un documento elettronico che lega una chiave pubblica ad un'identità (informazioni riguardo il nome della persona o dell'organizzazione, l'indirizzo internet e fisico, ecc..).

Viene rilasciato da un'organizzazione largamente fidata che è chiamata Certification Authority (CA), ed è corredato dalla firma digitale di quest'ultima che garantisce l'autenticità del certificato.

E' in possesso dell'entità che vuol dimostrare la propria identità e viene inviato su richiesta.

Un certificato contiene al suo interno una serie di informazioni. Di seguito si elencano le piu' importanti (ed obbligatorie):

- **Emittente**: Certification Authority che ha rilasciato il certificato. Le informazioni sulle CA conosciute e le loro chiavi pubbliche sono largamente note e contenute all'interno dei comuni browser/applicazioni.
- **Periodo di validità**: Ogni certificato ha una validità temporale limitata, al termine della quale il certificato non è piu' valido (scaduto) ed è necessario richiederne un altro alla Certification Authority.
- **Soggetto**: Ente di cui il certificato garantisce l'identità.
- **Chiave pubblica del soggetto**: Contiene la chiave pubblica che viene attribuita al soggetto. Le comunicazioni con esso verranno cifrate utilizzando questa chiave.
- **Firma**: Firma digitale della Certification Authority che ha rilasciato il certificato. Consiste in un hash calcolato sulla prima parte del certificato stesso, che è stato successivamente cifrato utilizzando la chiave privata della CA. Il ricevente del certificato verifica l'autenticità della firma decrittandola per mezzo della chiave pubblica della CA (largamente nota e diffusa).

Catene di certificati

Certificati multipli possono essere collegati tra loro in una catena di certificati. Una *catena* è un raggruppamento gerarchico tra 2 o più certificati in cui ognuno garantisce ricorsivamente per il precedente. Il certificato più in basso è relativo all'entità di cui si è richiesta l'autenticazione. Ogni certificato successivo garantisce l'identità dell'ente che ha rilasciato il certificato precedente. La cima della gerarchia contiene il certificato di una CA largamente nota e fidata da tutti.

3.2.3 La sicurezza in SSL

In questa sezione verrà descritto in che modo SSL garantisce le 3 proprietà fondamentali di una connessione sicura [SUN06].

Riservatezza della comunicazione

SSL garantisce la riservatezza della comunicazione rendendone il contenuto illeggibile a terzi per mezzo della crittografia.

La tecnica di crittografia utilizzata per trasportare i dati applicazione è quella a chiave segreta, che viene generata *ex-novo* ogni sessione. La chiave viene calcolata da entrambi i peer utilizzando una serie di numeri casuali e il *PreMasterSecret* che vengono scambiati durante la fase di Handshake della comunicazione. Questa fase, preventiva alla trasmissione dei dati applicazione, utilizza la crittografia a chiave pubblica (asimmetrica) che non risente del problema di riservatezza nella comunicazione della chiave.

I numeri che permettono di generare la chiave segreta e il *PreMasterSecret* vengono pertanto comunicati utilizzando la crittografia a chiave pubblica. Ciascun peer possiede adesso i parametri per generare la chiave segreta senza che la riservatezza sia compromessa. Una volta finita la fase di Handshake si passerà alla comunicazione vera e propria, cifrata per mezzo della chiave segreta.

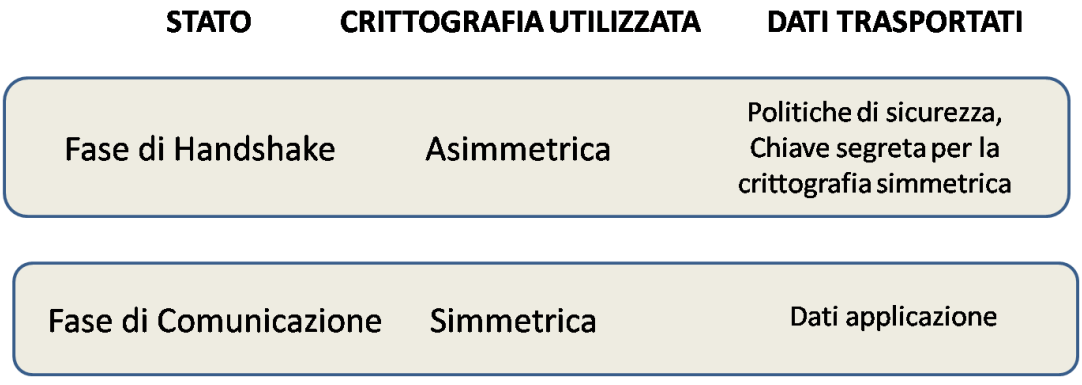


Figura 3.2: SSL: Tipologie di crittografia utilizzate

La scelta di utilizzare due diversi sistemi di cifratura in fasi differenti trae motivo dall'eccessiva lentezza computazionale della crittografia asimmetrica, che la rende sconsigliabile per grosse quantitativi di dati.

Autenticazione

SSL garantisce l'identità degli interlocutori mediante l'utilizzo dei certificati a chiave pubblica. Il protocollo non fornisce vincoli sulla tipologia di certificati da utilizzare, tuttavia ad oggi è esclusivamente utilizzato *X.509*.

La verifica dell'identità avviene nella fase di Handshake. Il Client emette una richiesta di esibizione del certificato e il Server è obbligato a fornirlo. Se il certificato è valido, il server è autenticato e l'Handshake puo' proseguire. Opzionalmente, è possibile trasmettere il certificato alla Certification Authority che l'ha rilasciato, per ottenere da essa un'ulteriore conferma di validità.

L'autenticazione del Client non è obbligatoria, tuttavia è prevista dal protocollo nonostante essa sia richiesta in scenari molto rari. In questo caso si parla di *mutua autenticazione*.

Integrità dei dati

Per assicurare che i dati giungano a destinazione inalterati, SSL utilizza una funzione di *hash crittografico*. L'hash è una piccola porzione di bit che viene calcolata sulla base dei dati da trasmettere. Ogni modifica nei dati produce un cambiamento dell'hash risultante.

In SSL/TLS, l'hash crittografico prende il nome di MAC (Message Authentication Code), e viene inviato insieme ai dati da trasmettere.

Il ricevente calcola l'hash sui dati ricevuti e lo confronta con il MAC di cui sono corredati. Se vi è corrispondenza, non vi è stata alterazione.

Le funzioni di hash crittografico più comunemente utilizzate da SSL/TLS sono l'MD5 (Message Digest 5) e l'SHA (Secure Hash Algorithm).

3.2.4 La fase di Handshake

Prima di iniziare ad inviare il primo byte, gli interlocutori devono negoziare le politiche di sicurezza di sicurezza della connessione, gli algoritmi di cifratura e le chiavi da utilizzare. In questo scenario il protocollo si trova nella fase di Handshake [MI03], che richiede uno scambio di messaggi preventivi al trasporto dei dati applicazione.

Vengono adesso descritti i passi che compongono questa fase:

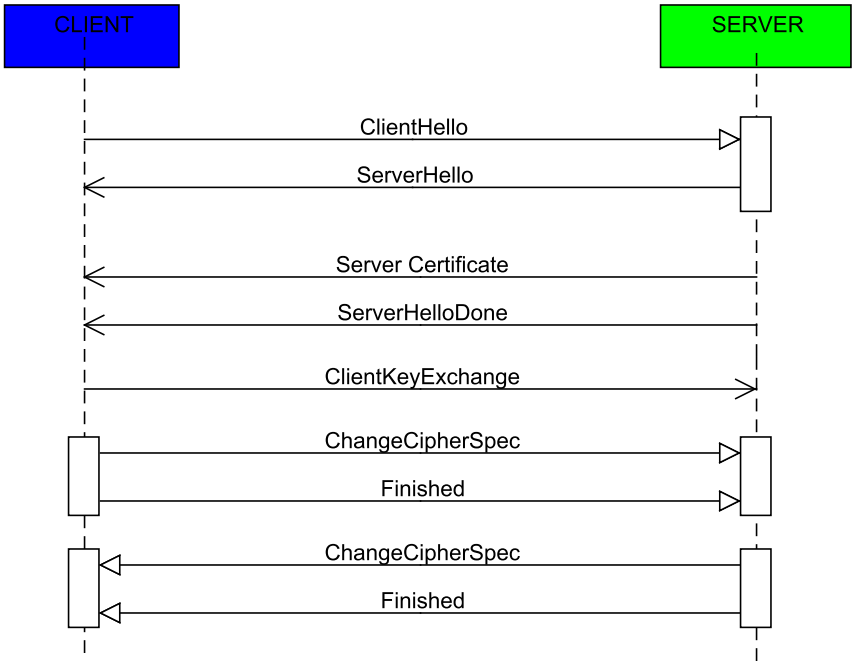


Figura 3.3: Diagramma di sequenza dell'Handshake in SSL.

1. Inizio della fase di Handshake:

   - Il Client inizia la comunicazione mandando un messaggio *ClientHello* in cui specifica la versione piu' recente di SSL/TLS che supporta, un numero generato casualmente (utilizzato in seguito come prima parte per creare la chiave segreta), una lista degli algoritmi di cifratura e di compressione che supporta.

- Il Server risponde con un messaggio *ServerHello*, che contiene la versione scelta del protocollo, un numero generato casualmente (utilizzato in seguito come seconda parte per creare la chiave segreta), e l'algoritmo di cifratura e quello di compressione che ha scelto.
- Il Server invia un messaggio contenente il suo certificato digitale (*Server Certificate*).
- Il Server invia un messaggio *ServerHelloDone*, che indica che ha completato la negoziazione.
- Il Client verifica la validità del certificato ricevuto e risponde con un messaggio *ClientKeyExchange*, che può contenere una parte di chiave segreta (*PreMasterSecret*), la chiave pubblica, o niente (dipende dall'algoritmo di cifratura).
- Il Client e il Server utilizzano i numeri casuali scambiati all'inizio e il *PreMasterSecret* per calcolare la chiave segreta.

2. Il Client adesso manda un record di tipo *ChangeCipherSpec* che indica che ogni successiva comunicazione che invierà d'ora in avanti sarà crittata, secondo le politiche e la chiave definite in precedenza.
   - Il Client manda un messaggio *Finished*, autenticato e cifrato, contenente un hash e un MAC calcolati sui tutti i messaggi di Handshake precedentemente inviati.
   - Il Server prova ad decifrare il messaggio *Finished* e verificare l'hash e il MAC. Se questo processo fallisce, l'Handshake è considerato non concluso e la connessione viene chiusa.
3. Specularmente, il Server manda un record *ChangeCipherSpec*, che indica anch'esso l'inizio della comunicazione crittografata.
   - Il Server manda il suo messaggio *Finished*, autenticato e crittografata.

- Il Client esegue la stessa verifica e decodifica.

Da questo momento l'Handshake è completato, gli interlocutori possono cominciare a scambiare i propri dati applicazione.

Handshake con mutua autenticazione

In alcuni casi poco frequenti, il Server può richiedere al Client di autenticarsi. In questo caso si parla di mutua autenticazione, e il relativo Handshake prevede alcuni step aggiuntivi:

- L'Handshake avviene come da procedura con lo scambio dei messaggi *ClientHello*, *ServerHello* e *Server Certificate*. Successivamente:
  - Il Server invia un messaggio di tipo *Certificate Request*, comunicando al Client di esibire il proprio certificato.
  - Il Client risponde con un messaggio *Client Certificate* che contiene il suo certificato.
  - Il Client invia un altro messaggio, di tipo *CertificateVerify*, che contiene un hash dei precedenti messaggi di Handshake, cifrato con la chiave privata del Client.
- Il server preleva la chiave pubblica del Client dal certificato che ha appena ricevuto e la utilizza per effettuare l'operazione di decifratura sul messaggio *CertificateVerify*. Se vi riesce, ha la prova che il Client della chiave privata del certificato, e quindi la sua identità è confermata.

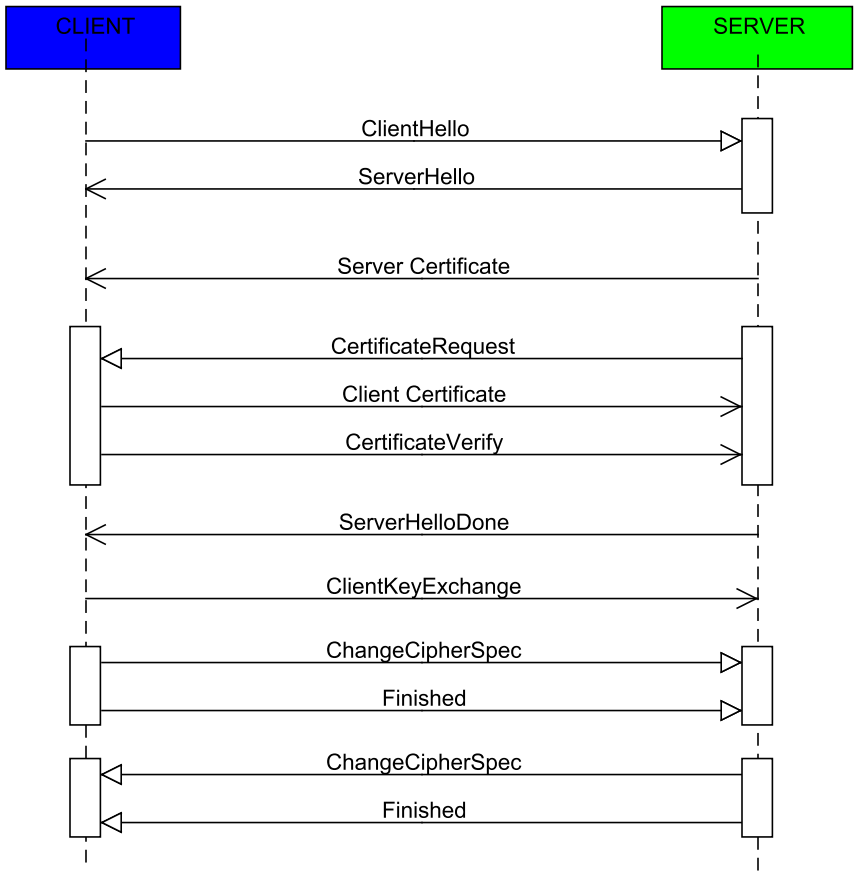


Figura 3.4: Handshake con mutua autenticazione.

3.3 SSL/TLS in Java: SSLEngine

3.3.1 Introduzione

Java fornisce l'implementazione del protocollo SSL/TLS con il pacchetto *javax.net.ssl*.

Il metodo più semplice per avviare una comunicazione è tramite la classe *SSLSocket* [SSS]. Il lavoro dello sviluppatore è limitato a:

- Creazione di un oggetto *SSLContext*. Esso contiene i parametri iniziali per la comunicazione SSL, che sono:
  - Indirizzo del proprio certificato, formato e password;
  - Indirizzo del file di elenco delle Certification Authority fidate, formato e password;
  - Versione di SSL/TLS da utilizzare.

  L'oggetto *SSLContext* è richiesto per inizializzare *SSLSocket*.
- Sostituzione della classe *java.net.Socket* (utilizzato per la comunicazione in chiaro) con *SSLSocket*, senza necessità di porre alcuna modifica al codice.

Sebbene sia semplice da utilizzare, questa soluzione pone una serie di limitazioni: si basa sul tradizionale I/O basato su stream, che è bloccante e costringe ad un approccio *thread-pooled*[4] che richiede un thread per ogni connessione attiva [NS04].

Benché SSLSocket soddisfi la maggior parte delle casistiche che prevedono una connessione sicura, questa soluzione non è percorribile nei casi in cui è richiesto di coniugare sicurezza e scalabilità, ed è necessario utilizzare la complessa API *SSLEngine* [SSE].

[4] Approccio basato sulla creazione di un *thread* per ogni comunicazione da gestire. Il thread si pone in attesa di ricezione dei dati e resta attivo per tutta la durata della comunicazione.

Essa implementa un'automa a stati che esegue tutte le operazioni relative al protocollo SSL/TLS, ed è indipendente dal trasporto e comunica utilizzando Buffer di byte. E' competenza dello sviluppatore recapitare i byte prodotti da SSLEngine all'altro end-point della connessione.

Separando SSL dal trasporto si ottiene il significativo vantaggio di supportare tutti i possibili metodi di I/O e modelli di threading esistenti, sia presenti che futuri. Tuttavia, questa flessibilità paga il prezzo nella complessità. Molti dei dettagli del protocollo SSL/TLS nascosti allo sviluppatore nella tradizionale implementazione basata su stream sono adesso esposti in SSLEngine e vanno gestiti dallo sviluppatore. Questo include dettagli di handshaking, riassemblaggio di pacchetti SSL, controllo degli stati interni di SSLEngine ed esecuzione di task delegati (vedi paragrafo 3.3.4).

SSLEngine è un'API avanzata, sconsigliata per la sua complessità dalla Sun stessa[5] in favore di SSLSocket a meno che non sia strettamente necessario. Tuttavia nel caso di soluzioni che richiedono scalabilità o I/O non bloccante è necessario utilizzare SSLEngine. Verrà descritta adesso l'API nel dettaglio.

SSLEngine implementa una macchina a stati che esegue tutte le operazioni relative al protocollo SSL/TLS, incluso handshaking, cifratura e decifratura.

3.3.2 Ciclo di vita

In questa sezione viene descritto il ciclo di vita di SSLEngine.

1. **Creazione** - L'istanza di SSLEngine è creata, ma non è ancora stata utilizzata. La creazione avviene tramite la classe *SSLContext*. Essa inizializza un'ambiente SSL specificando il certificato da utilizzare, la lista della CA fidate e la versione del protocollo.

[5] Consultare il riferimento bibliografico [NS04].

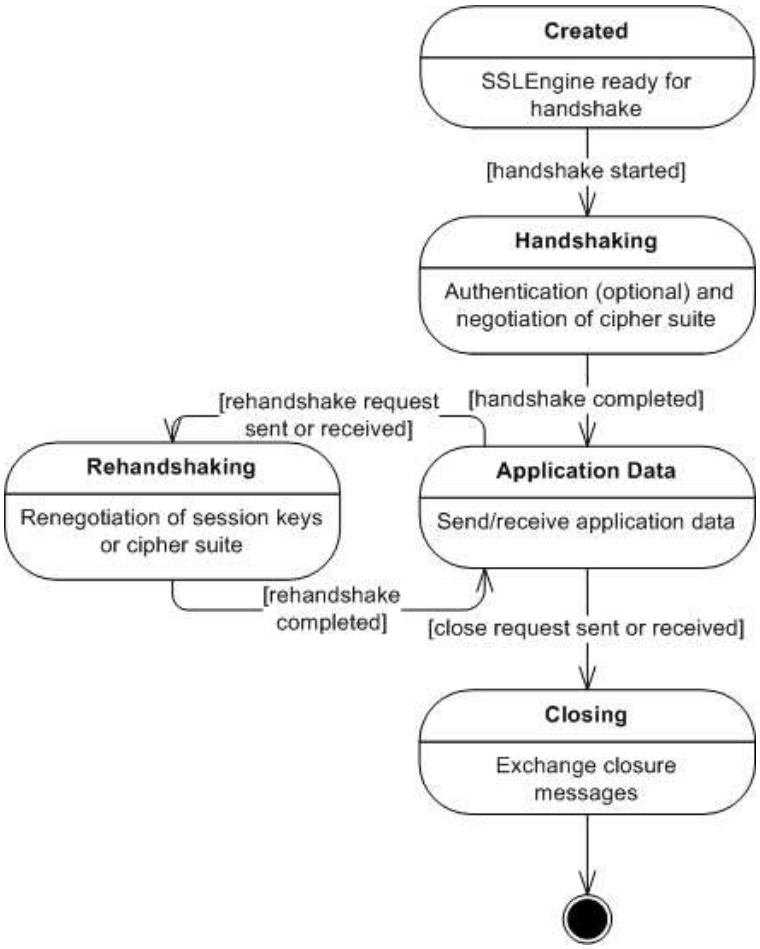


Figura 3.5: Ciclo di vita di un'istanza SSLEngine

Dopo la creazione dell'ambiente si impostano i parametri specifici di SSLEngine: suite di cifratura abilitati, modalità Client o Server.

2. **Handshaking** - L'Handshake è una fase in cui i due peer si scambiano i parametri per la comunicazione e la sicurezza. Questa fase genera lo scambio di numerosi messaggi. I dati applicazione non possono essere inviati in questa fase.

3. **Dati Applicazioni** - Una volta che i parametri di comunicazione e sicurezza sono stabiliti e l'Handshake è completato, i dati ap-

plicazione possono transitare tramite SSLEngine. I messaggi in uscita sono cifrati e protetti contro le alterazioni, quelli in arrivo attivano il processo inverso.

4. **Rehandshaking** - Ciascuna delle parti può chiedere di rinegoziare la sessione in qualunque momento durante la fase Dati Applicazioni. Prima di cominciare nuovamente l'Handshake, l'applicazione può modificare i parametri di comunicazione e sicurezza, ma non può cambiare la modalità Client/Server.
5. **Chiusura** - L'applicazione invia/riceve i messaggi pendenti all'altro peer, poi chiude il meccanismo di trasporto e successivamente SSLEngine. Una volta che l'engine è chiuso, non è piu' riutilizzabile: è necessario creare un'altra istanza.

3.3.3 Interazione con le applicazioni

SSLEngine è costituito da due metodi principali: *wrap()* e *unwrap()*, rispettivamente per la cifratura e la decifratura dei dati. Entrambi prendono in entrata 2 Buffer di byte: *source* e *dest*, il primo contenente i dati in input da manipolare e il secondo per la scrittura dei dati di output generati.

Per capire come avviene l'interazione con l'API, viene adesso illustrata la struttura-tipo di un'applicazione che utilizza SSLEngine:

Il diagramma si legge dall'alto verso il basso.

L'applicazione scrive in chiaro i dati da trasportare nel buffer *outAppBuffer*. Esso viene inviato ad SSLEngine tramite il metodo *wrap()*. I dati vengono cifrati e resi disponibili in *outNetBuffer*, e sono pronti per essere recapitati all'altro peer (il trasporto è a carico dell'applicazione).

I dati ricevuti dall'IO subiscono il processo inverso (dal basso all'alto).

Quando ci si trova nella fase di Handshake è l'engine stesso a produrre (*consumare*) i dati da inviare (*ricevere*).

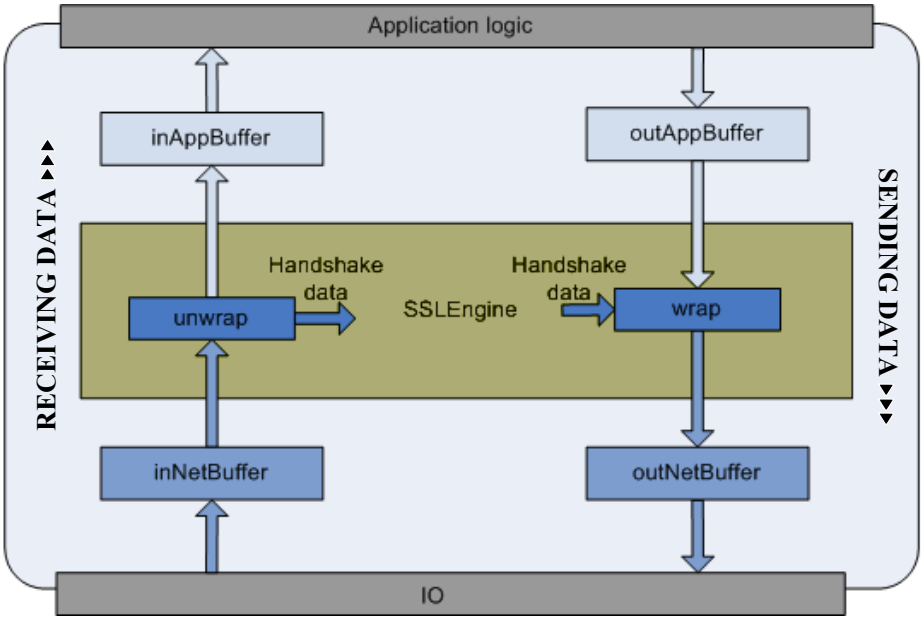


Figura 3.6: Il flusso dei dati in un'applicazione che utilizza SSLEngine

Le fasi sono controllate da SSLEngine, che segnala i propri avanzamenti passando attraverso una serie di stati interni.

L'applicazione deve esaminare gli stati prima e dopo ogni interazione con l'API.

3.3.4 Gli stati di SSLEngine

Questa sezione descrive gli stati interni di SSLEngine. Essi sono i seguenti:

SSLEngineResult Status

Ogniqualvolta viene invocato uno dei metodi *wrap()* o *unwrap()*, viene ritornato un oggetto *SSLEngineResult* che contiene:

– **BytesConsumed** - Numero dei bytes consumati da SSLEngine, prelevati da *source*;

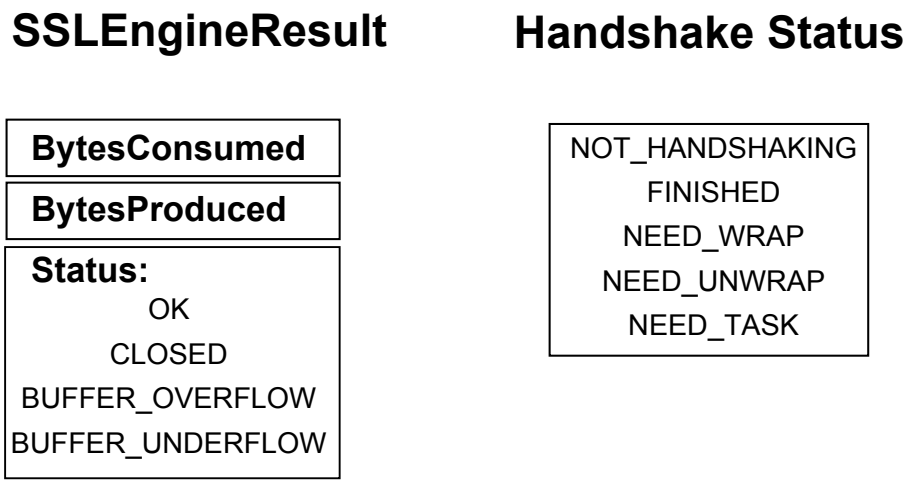


Figura 3.7: I possibili stati interni di SSLEngine

- **BytesProduced** - Numero dei bytes prodotti da SSLEngine, disponibili in *dest*;

- **Status** - Stato interno dell'engine: Puo' assumere i seguenti valori:

  * **OK**: L'operazione è stata eseguita con successo. Alcuni dati sono stati consumati, prodotti o entrambi.

  * **CLOSED**: SSLEngine è chiuso. Quest'istanza non può piu' essere utilizzata.

  * **BUFFER_OVERFLOW**: Il Buffer *dest* è troppo piccolo per contenere tutti i dati generati dall'engine; L'applicazione dove liberare spazio nel Buffer o aumentarne le dimensioni ed invocare nuovamente il metodo.

  * **BUFFER_UNDERFLOW**: Il protocollo SSL/TLS è *packet-based* e può operare solo su pacchetti completi. Questo stato indica che il Buffer *source* non contiene abbastanza dati e l'applicazione deve continuare a leggere i dati dalla rete ed invocare nuovamente il metodo.

Handshake Status

Nella fase di Handshake, SSLEngine dirige la sequenza delle azioni da eseguire, e si serve dell'*Handshake Status* per comunicare all'applicazione la prossima operazione da effettuare. Esso può assumere i seguenti valori:

- **NOT_HANDSHAKING**: L'Handshake non è in corso.
- **FINISHED**: L'ultima operazione ha determinato la fine dell'Handshake con successo.
- **NEED_WRAP**: Per proseguire è necessario invocare il metodo *wrap()*. L'engine produrrà dei byte da recapitare all'altro peer.
- **NEED_UNWRAP**: Per proseguire è necessario attendere la risposta dell'altro peer che deve essere inviata all'engine tramite il metodo *unwrap()*. Esso consumerà i byte forniti.
- **NEED_TASK**: L'engine ha bisogno di eseguire un'operazione che blocca o che richiede molto tempo. Secondo il Non-Blocking Model, questo tipo di operazioni non può essere eseguito internamente. L'applicazione deve invocare il metodo *getDelegatedTask()* per ottenere un'istanza *Runnable* che incapsula il task da eseguire ed avviarla separatamente. Al termine di essa l'applicazione deve ricontrollare l'*Handshake Status* per conoscere la prossima operazione da effettuare.

  Solitamente questo tipo di task corrisponde alla generazione delle chiavi o alla richiesta di una password all'utente.

Capitolo 4

Implementazione del modulo HTTPS in Jolie

4.1 Introduzione

JOLIE è un valido strumento emergente per la creazione di orchestratori di servizi. Tuttavia, gli unici protocolli di comunicazione che supporta sono in chiaro e non forniscono alcuna protezione per la sicurezza dei dati. Nello scenario attuale, operazioni come richieste di password, compravendita online e transazioni bancarie sono molto comuni nel web, e necessitano di opportune garanzie di sicurezza della comunicazione.

La mancanza di una comunicazione sicura pone un grosso limite al campo di applicazioni di JOLIE, restringendone il dominio di utilizzo.

Questo lavoro di tesi intende sopperire a questa carenza implementando un modulo aggiuntivo per JOLIE atto a garantire la comunicazione attraverso lo standard di sicurezza più comunemente utilizzato sul web: SSL/TLS.

Il modulo è implementato come estensione di JOLIE e fornisce il supporto al protocollo di comunicazione HTTPS[1].

Il codice è scritto interamente in Java ed interagisce con l'API *SSLEngine* per effettuare le operazioni prescritte dal protocollo SSL/TLS.

4.2 La comunicazione HTTPS in Jolie

Cenni sul protocollo HTTPS

HTTPS [IET00] è la versione sicura del protocollo HTTP. La comunicazione avviene tramite le regole di HTTP a cui viene aggiunto un layer, collocato prima del livello di trasporto, che effettua le operazioni cifratura/ decifratura e autenticazione previste da SSL/TLS. Questo processo produce un nuovo tipo di messaggio (di tipo *SSL/TLS*) che incapsula al suo interno il messaggio originario HTTP. Per questo motivo, HTTPS è chiamato anche **HTTP over SSL**. A causa dell'aggiunta del layer per la sicurezza, HTTPS diventa un protocollo *stateful* e necessita di effettuare un'Handshake prima dell'invio dei dati applicazione.

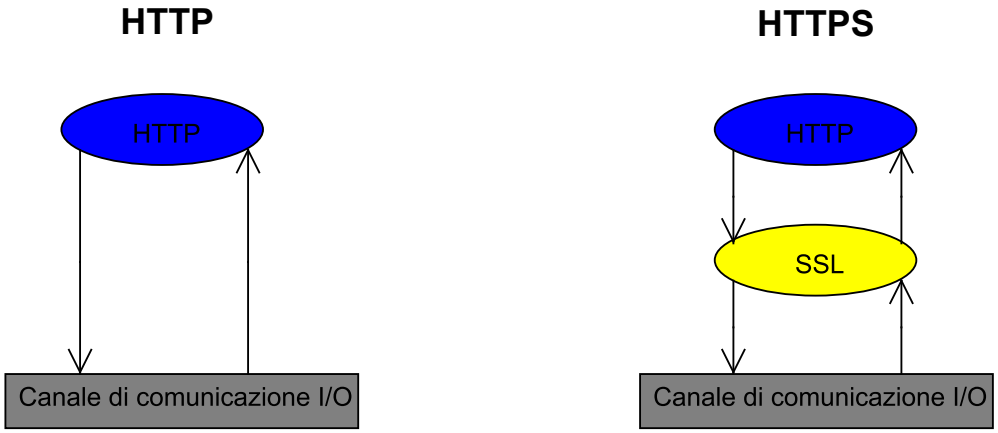


Figura 4.1: Differenze tra HTTP e HTTPS

[1]Hypertext Transfer Protocol over Secure Socket Layer, detto HTTPS

Architettura della comunicazione HTTPS in Jolie

Utilizzando la stessa logica di HTTPS, l'idea è di interporre il nuovo modulo tra l'estensione esistente che implementa HTTP e il canale di comunicazione di I/O. In questo modo, ogni messaggio che vi transita viene prima catturato dal modulo, il quale provvede ad applicare le politiche di sicurezza SSL/TLS (nel caso di un messaggio in uscita) o a rimuoverle e consegnare il messaggio in chiaro al layer sovrastante HTTP (*messaggio in entrata*).

Avremo pertanto un'architettura di questo tipo:

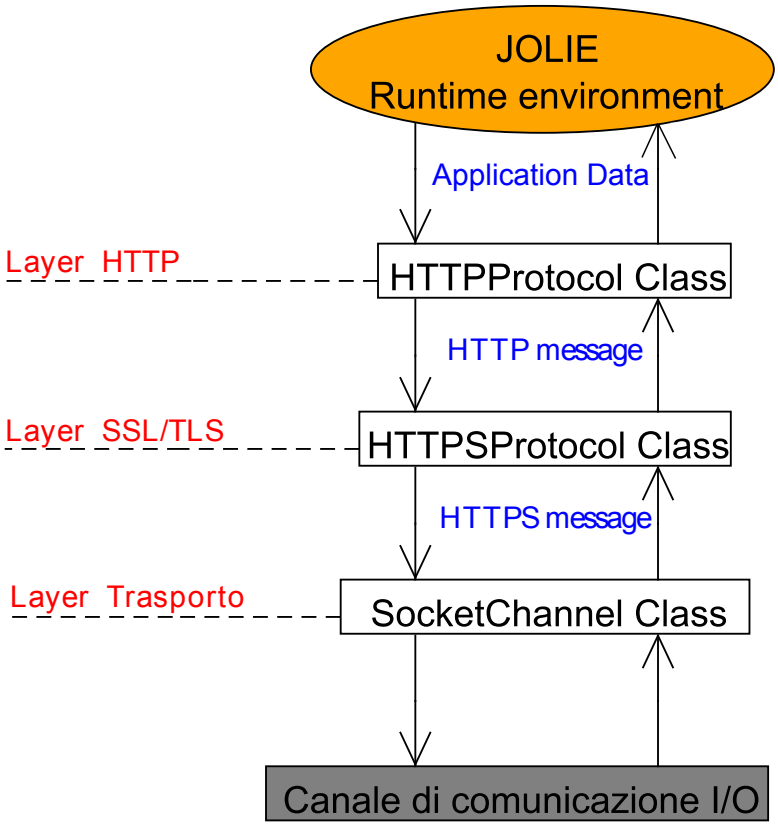


Figura 4.2: Da dati applicazione a messaggio HTTPS e viceversa: Gli step.

Nella figura, quando un servizio orchestrato da JOLIE sceglie di inviare dati tramite HTTPS, il messaggio contenente i dati applicazione viene

processato da *HTTPProtocol* e trasformato in un messaggio HTTP. Nello step successivo, esso viene cifrato ed incapsulato all'interno di un messaggio HTTPS da *HTTPSProtocol*. Il messaggio viene poi inviato tramite Socket. In ricezione si attiva il processo inverso.

Questa metodologia consente il totale riutilizzo del codice di *HTTP-Protocol* in quanto le operazioni del protocollo HTTPS sono totalmente invisibili ad esso. Tuttavia, si rende necessaria una modifica all'ambiente Jolie per istanziare due diverse classi (*HTTPProtocol* e *HTTPSProtocol*) per la gestione di un unico protocollo.

Il modulo realizzato risolve quest'esigenza utilizzando una soluzione che non comporta modifiche all'architettura.

Soluzione utilizzata

Architettura del modulo HTTPS:

HTTPSProtocol istanzia al suo interno un oggetto *HTTPProtocol*, che utilizza per effettuare tutte le operazioni richieste dal protocollo HTTP, in maniera totalmente invisibile al resto dell'architettura. Questo permette di astrarre i layer HTTP e SSL/TLS in un unico layer HTTPS che li incorpora, senza modificare l'architettura.

Flessibilità

Questa soluzione fornisce inoltre un vantaggio di flessibilità.

Poiché *HTTPSProtocol* si limita ad aggiungere SSL/TLS al messaggio che riceve e delega la gestione di HTTP alla classe apposita, è possibile, con piccole modifiche al codice, realizzare moduli analoghi che, similarmente, aggiungono SSL ad altri protocolli (creando ad esempio SODEPS, SOAPS, ecc..).

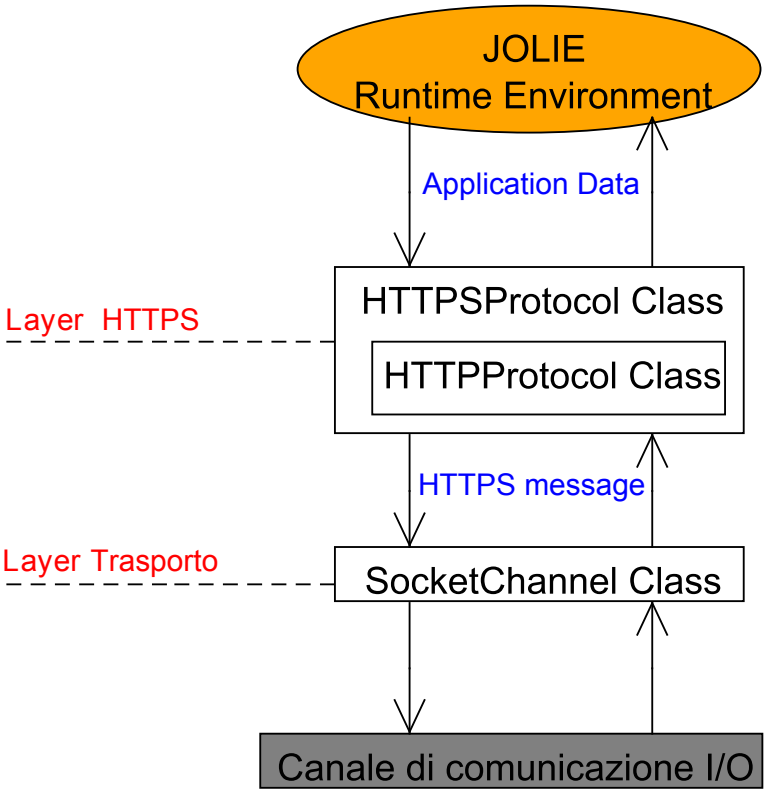


Figura 4.3: Architettura utilizzata per la comunicazione tramite HTTPS

4.3 Implementazione del modulo HTTPS

Il modulo è costituito da tre classi:

- **HTTPSProtocol**
- **HTTPSProtocolFactory**
- **SSLInputStream**

HTTPSProtocol è la classe principale, e rappresenta la gestione del protocollo HTTPS. Incorpora tutte le funzioni eseguite del modulo ed è istanziata dall'ambiente JOLIE qualora venga richiesta la comunicazione tramite HTTPS. Essa svolge le sue seguenti funzioni:

- Istanziazione della classe *HTTPProtocol* per l'applicazione delle logiche di HTTP;
- Applicazione delle logiche HTTPS attraverso l'utilizzo dell'API *SSLEngine*;
- Implementazione delle primitive *send()* e *recv()*, per la comunicazione con il Communication Core e l'ambiente Jolie;
- Applicazione dei parametri configurabili di HTTPS specificati nel servizio.

La classe interagisce con gli altri componenti dell'architettura tramite i metodi pubblici *send()* e *recv()*.

Il metodo *send()* viene invocato dall'ambiente JOLIE quando il servizio richiede l'invio di un messaggio. I dati applicazione vengono trasformati ed incapsulati all'interno di un HTTPS message, che viene inviato alla destinazione.

Prende in input il canale di destinazione, il messaggio da inviare e il canale di ricezione.

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

Il metodo *recv()* è invocato quando è stato ricevuto un messaggio HTTPS tramite il canale di comunicazione. Le politiche SSL/TLS vengono rimosse e il messaggio originario è ricostruito (Dati Applicazione).

Prende in input il canale di ricezione e quello di destinazione.

Ritorna un *CommMessage* che contiene l'Application Data ricevuto.

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

Le altre classi di cui è composto il modulo sono:

- *HTTPSProtocolFactory*: produce un'istanza di HTTPSProtocol secondo il design pattern Factory;

- *SSLInputStream*, è una classe ausiliaria utilizzata da *HTTPSProtocol* per la comunicazione con *HTTPProtocol*. E' descritta nella sezione 4.3.3.

4.3.1 Descrizione del codice

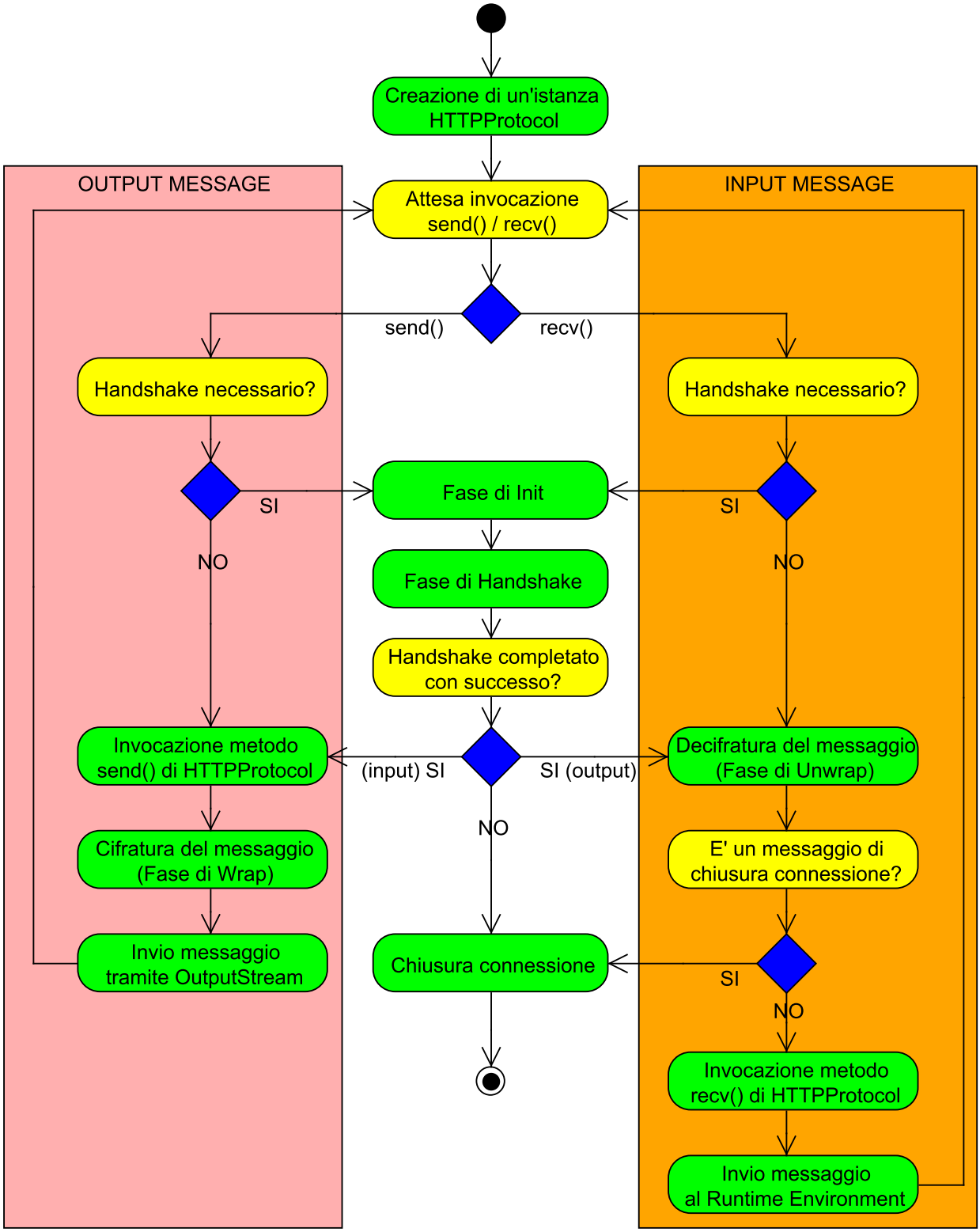


Figura 4.4: Diagramma di flusso del codice di HTTPSProtocol

La figura descrive l'esecuzione di *HTTPSProtocol*. Alla creazione dell'oggetto, il costruttore crea a sua volta un'istanza di HTTP-Protocol.

Quando è invocato il metodo *send()*, viene controllato se bisogna effettuare l'Handshake. Ciò avviene in due casi: quando è segnalato dall'*HandshakeStatus*, oppure quando non è ancora stata creata l'istanza di SSLEngine. Una volta che l'eventuale Handshake è stato completato, si invoca il metodo *send()* di HTTP-Protocol passando come parametro l'Application Data da inviare (sotto forma di *CommMessage*), ed esso viene trasformato in un messaggio HTTP. Nella fase successiva, il nuovo messaggio viene inviato a SSLEngine per la cifratura attraverso il metodo *wrap()*. Il messaggio è ora di tipo HTTPS e viene inviato a destinazione tramite l'OutputStream.

L'invocazione del metodo *recv()* attiva il controllo sull'handshake, se è necessario viene effettuato. Successivamente il messaggio ricevuto viene inviato a SSLEngine (metodo *unwrap()*) per la decifratura. Se è un messaggio di chiusura connessione, lo stato di SSLEngine diventa CLOSED. In questo caso il metodo termina ritornando un *CommMessage* vuoto; diversamente viene invocato il metodo *recv()* di HTTPProtocol passando il messaggio HTTP da ricevere (sotto forma di *CommMessage*), ed esso viene trasformato in Application Data. A questo punto il metodo termina ritornando tale messaggio.

Vengono adesso descritte nel dettaglio le fasi presenti in figura:

Fase di Init - init()

SSLEngine supporta una serie di parametri di configurazione della comunicazione SSL/TLS.

Il modulo realizzato permette di impostare tali parametri specificandoli nel codice JOLIE del servizio. La lista dei parametri confi-

gurabili e le modalità per l'impostazione degli stessi è consultabile nella sezione 4.4.

Questa fase, gestita dal metodo *init()*, preleva dal codice JOLIE i parametri di configurazione e li utilizza per impostare conseguentemente l'istanza di SSLEngine.

Se è avviata per la prima volta, verrà creata una nuova istanza SSLEngine.

Questa fase è avviata dal primo *send()* o *recv()* che viene invocato, o qualora sia necessario effettuare nuovamente l'Handshake. Il motivo per cui si posticipa questa fase collocandola dopo la ricezione del primo input o output è per l'impostazione della modalità Client/Server. Essa viene dedotta dalla prima operazione. Se è un *send()* SSLEngine sarà configurato come Client, diversamente come Server. Tale comportamento evita allo sviluppatore di dover specificare esplicitamente la modalità richiesta nel codice JOLIE, ed è stato appositamente concordato con i creatori di JOLIE. La modalità Client/Server è richiesta obbligatoriamente da SSLEngine e non può essere cambiata per tutto il ciclo di vita dell'istanza, pertanto non verrà reimpostata nel caso di una nuova richiesta di Handshake.

Fase di Handshake - startHandshake()

Questa fase avvia il processo di Handshake con l'altro peer. Il metodo controlla ciclicamente l'*Handshake Status* di SSLEngine ed in base ad esso effettua le operazioni adeguate per proseguire nell'Handshake. Il ciclo termina quando lo stato è impostato su 'FINISHED' o 'NOT_HANDSHAKING'.

La sezione 4.3.2 descrive il comportamento del modulo rispetto ad ogni stato di SSLEngine.

Fase di Wrap - wrap()

Questa fase applica le politiche di HTTPS ad un messaggio da inviare. Esso viene passato ad SSLEngine tramite il metodo SSLEngine.wrap() insieme ad un *ByteBuffer* per contenere il messaggio trasformato.

La fase è invocata inoltre durante l'Handshake quando l'*Handshake Status* è impostato su NEED_WRAP. Questo stato indica che SSLEngine richiede di inviare dei dati all'altro peer, pertanto il messaggio passato a *wrap()* sarà vuoto in questo caso.

L'operazione di *wrap()* puo' generare lo stato di BUFFER_OVERFLOW. Questo accade quando il Buffer di destinazione passato a *wrap()* non è grande a sufficienza da contenere il risultato.

HTTPSProtocol controlla lo stato di SSLEngine ogni esecuzione di *wrap()*. Se si è verificato il Buffer Overflow, le dimensioni del Buffer vengono aumentate e si ritenta l'operazione.

Fase di Unwrap - unwrap()

Questa fase serve a rimuovere le logiche di SSL/TLS da un messaggio ricevuto. Esso viene passato ad SSLEngine tramite il metodo SSLEngine.unwrap(), insieme ad un *ByteBuffer* per contenere il messaggio trasformato.

La fase è invocata inoltre durante l'handshake quando l'Handshake Status è impostato su NEED_UNWRAP. Questo indica che SSLEngine sta aspettando una risposta dall'altro peer, in questo caso il messaggio processato non contiene dati applicazione.

L'operazione di *unwrap()* può generare lo stato di BUFFER_OVERFLOW o BUFFER_UNDERFLOW. Ad ogni esecuzione del metodo, lo stato viene monitorato. Nel caso di Buffer Overflow, le dimensioni del buffer vengono aumentate e si ritenta l'operazione. Lo stato di BUFFER_UNDERFLOW indica che i dati forniti non so-

no sufficienti ad effettuare l'operazione di decifratura (SSLEngine è in grado di operare solo su pacchetti completi) ed è necessario attendere la ricezione della parte restante dal canale. Il metodo pertanto effettua una lettura sul canale, unisce i dati ricevuti ai precedenti e ritenta l'operazione.
Prima della sua terminazione, il metodo effettua un controllo sul messaggio decifrato per impostare il flag *MoreToUnWrap*.

Il flag MoreToUnwrap

Secondo le specifiche del protocollo SSL/TLS, nella fase di Handshake ci sono degli step in cui il Server o il Client inviano 2 messaggi o più messaggi contemporanei senza aspettare la risposta dall'altro peer. Essendo composti da pochi byte, nella pratica avviene molto frequentemente che questi messaggi vengano inviati in un unico pacchetto.
Ciò nonostante, ciascun messaggio richiede comunque una distinta operazione di Unwrap.
Per esempio, si suppone adesso che il Client mandi al Server il messaggio *ClientHello*. Esso risponde mandando, come da protocollo HTTPS, i messaggi di *ServerHello*, *Server Certificate*, *Certificate Request*. Essi vengono accopparti dal layer di trasporto in un unico pacchetto ed inviati al Client. L'operazione di Unwrap sul pacchetto ricevuto processerà soltanto il primo messaggio e l'*Handshake Status* continuerà ad essere su NEED_UNWRAP. In questo caso sarebbe un errore mettersi in attesa sul canale in quanto si possiede già il messaggio aspettato.
Il flag MoreToUnwrap è fornito in ritorno da HTTPSProtocol.unwrap() ed indica se dopo l'unwrapping del pacchetto ricevuto sono presenti ancora dati da processare. Il controllo viene effettuato monitorando la variabile *Bytesconsumed*, ritornata da SSLEngine.unwrap() dopo ogni operazione di Unwrap. Se *Bytesconsumed* è inferiore

alla lunghezza del pacchetto ricevuto, i byte processati vengono rimossi dal pacchetto e *moreToUnwrap* è impostato su true. Diversamente se *Bytesconsumed* è uguale alla lunghezza del pacchetto questo indica che tutti i dati sono stati processati e *moreToUnwrap* è impostato su False.

Una soluzione alternativa poteva essere la continuazione dell'unwrapping del messaggio direttamente nel metodo *unwrap()* nel caso in cui il pacchetto contenesse ancora dati, ma non è applicabile in quanto lo stato di SSLEngine può cambiare in seguito all'operazione di unwrap (ad esempio richiedendo l'esecuzione di un task, operazione che va effettuata nel metodo di gestione dell'Handshake *startHandshake()*).

4.3.2 La gestione degli stati di SSLEngine

Durante la comunicazione con SSLEngine, l'API assume una serie finita di stati. Questa sezione descrive il comportamento del modulo HTTPS rispetto ad ogni stato di SSLEngine.

L'API contiene due indicatori di stato: *Status* ed *HandshakeStatus*.

Status

Può assumere i seguenti valori:

- **OK** - Operazione eseguita con successo;
- **CLOSED** - Nella fase di Handshake questo stato indica che è avvenuto un errore fatale che ha impedito l'Handshake. Esso non può piu' continuare e l'API lancia l'eccezione *SSLHandshakeException* che comunica il fallimento dell'Handshake. La comunicazione non può proseguire.

  Fuori dalla fase di Handshake indica che la connessione è stata appena chiusa. Se questo è avvenuto in seguito ad un'o-

perazione di Unwrap significa che è stato appena ricevuto un *gracefully shutdown*[2];

- **BUFFER_OVERFLOW** - Il modulo aumenta le dimensioni del *ByteBuffer* da inviare a SSLEngine e ritenta l'operazione;
- **BUFFER_UNDERFLOW** - Il modulo si rimette in ascolto sul canale. Quando vengono ricevuti dei dati, essi vengono aggiunti ai precedenti e si ritenta l'operazione.

HandshakeStatus

Indicatore della prossima operazione da eseguire per proseguire nell'Handshake.

Se assume un valore diverso da NOT_HANDSHAKING o FINISHED viene invocato il metodo *startHandshake()*, che avvia l'Handshake e gestisce tutti gli stati di *HandshakeStatus*. Essi possono essere:

- **NOT_HANDSHAKING** - Nessuna azione da intraprendere;
- **FINISHED** - Determina la terminazione del metodo *startHandshake()*;
- **NEED_TASK** - Il task da eseguire viene prelevato tramite il metodo SSLEngine.*getDelegatedTask()*. Esso viene avviato e si resta in attesa del completamento;
- **NEED_WRAP** - Il modulo invoca il metodo HTTPSProtocol.wrap(). Successivamente controlla l'indicatore *Bytesproduced*: Se è maggiore di zero, i dati prodotti vengono inviati all'altro peer;

[2]Richiesta esplicita di chiusura connessione. Indica che l'altro peer ha terminato le operazioni con successo e la connessione può terminare

- **NEED_UNWRAP** - SSLEngine sta aspettando la ricezione di un messaggio. Il flag *moreToUnwrap* è controllato, se è False si mette in ascolto sul canale, diversamente si eliminano i dati già processati dal pacchetto ricevuto in precedenza e si procede ad effettuare l'unwrapping sulla parte restante.

4.3.3 La classe SSLInputStream

La classe SSLInputStream è una classe ausiliaria utilizzata da HTTPSProtocol nella comunicazione con HTTPProtocol. E' sottoclasse di ByteArrayInputStream e sovrascrive i metodi *read()* e *read(byte,int,int)*.

La sua implementazione si è resa necessaria per gestire i casi in cui viene inviato a HTTPProtocol uno stream contenente un messaggio HTTP incompleto. In questo caso HTTPProtocol non può proseguire autonomamente a leggere dal canale in quanto i dati sono cifrati e devono prima ricevere il processo di Unwrap.

Per gestire questo caso, lo stream passato a HTTPProtocol è un'istanza della classe SSLInputStream. In questo modo quando viene invocato il metodo *read()* e non ci sono piu' dati presenti nel Buffer, SSLInputStream provvede a leggere dal canale, eseguire l'Unwrap sul pacchetto ricevuto e ritornare i dati in chiaro.

4.4 Configurazione del modulo HTTPS

Il modulo HTTPS permette di impostare alcuni parametri di configurazione. Essi vanno specificati nel codice Jolie subito dopo la dichiarazione del protocollo, come nell'esempio:

```
inputPort provaPort
{
```

```
 Location: "socket://localhost:443"
 Protocol: https
 {
  .ssl.property1 = ``....'';
  .ssl.property2 = ``....'';
  ....
 }
 RequestResponse: prova1, prova2
}
```

I parametri impostabili sono i seguenti:

* **.ssl.protocol** - Specifica il protocollo di sicurezza da utilizzare. **Default:** SSL versione 3.0;
* **.ssl.keystore** - URL del certificato da utilizzare. Verrà inviato all'altro peer per garantire l'autenticazione. Questo parametro è obbligatorio per la modalità server. **Default:** *nessuno*
* **.ssl.keystorepwd** - Password del certificato. Questo parametro non deve essere impostato nel caso di certificati senza password, diversamente sarà generato un errore di errata password. **Default:** *nessuna password*
* **.ssl.keystoreformat** - Formato del certificato. Può essere JKS, JCEKS o PKCS12. **Default:** *JKS*
* **.ssl.truststore** URL del TrustStore da utilizzare (elenco delle Certification Authority fidate). **Default:** Ricerca del TrustStore di default nella JRE sulla macchina in uso. E' obbligatorio qualora la ricerca non vada a buon fine.
* **.ssl.truststorepwd** - Password del TrustStore, se presente. In caso contrario questo parametro non deve essere imposta-

to per evitare errori di errata password. **Default:** *nessuna password*

* **.ssl.truststoreformat** Formato del TrustStore. I formati supportati sono gli stessi del certificato e sono: JKS, JCEKS o PKCS12. **Default:** *JKS*

Riepilogo parametri di configurazione

Parametro	Obbligatorio	Default	Parametri supportati
.ssl.protocol	No	SSLv3	SSL / SSLv2 / SSLv3 / TLS / TLSv1 / TLSv1.1
.ssl.keystore	In Server Mode	nessuno	
.ssl.keystorepwd	No	nessuna	
.ssl.keystoreformat	No	JKS	JCKES / PKCS12 / JKS
.ssl.truststore	No se presente nella JRE	Ricerca nella JRE	
.ssl.truststorepwd	No	nessuna	
.ssl.truststoreformat	No	JKS	JCKES / PKCS12 / JKS

Capitolo 5

Caso d'uso: Autenticazione integrata per i servizi MSDNAA

5.1 Descrizione del caso d'uso

Questo capitolo illustra un caso di applicazione del modulo HTTPS implementato in questa tesi. Il caso in esame riguarda la realizzazione di un *Web Service* in JOLIE che interagisce con altre entità e servizi esterni per mezzo di messaggi HTTPS. Tale servizio effettua un'integrazione di servizi esistenti per progredire verso il completamento del processo da eseguire.

Lo scopo è di fornire un meccanismo di autenticazione integrata tra il sito Microsoft ELMS[1] [ELMa] del programma MSDNAA[2] [MSD] e l'Università di Bologna. Le operazioni necessarie al Web Service per implementare tale meccanismo includono:

* Scambio di messaggi e parametri di configurazione con il sito Microsoft ELMS tramite il protocollo HTTPS;

[1] E-academy License Management System.

[2] MicroSoft Development Network Academic Alliance

* Presentazione allo studente di una pagina HTTP cifrata per la richiesta delle credenziali d'Ateneo;
* Comunicazione con il server universitario interno per l'autenticazione dello studente.

Si descrivono le entità coinvolte:

5.1.1 Il programma MSDNAA

MSDN Academic Alliance è un progetto Microsoft riservato ai dipartimenti universitari di Informatica. Esso permette, previo pagamento di un canone annuale, l'installazione senza limiti della maggior parte del software Microsoft nei laboratori di dipartimento e la libera fruizione agli studenti e al personale delle Università che lo sottoscrivono.

Gli studenti possono inoltre ottenere gratuitamente copie licenziate dei prodotti Microsoft coperti dal programma, collegandosi al sito ELMS Webstore [ELMa] che permette il download del software disponibile.

L'accesso a ELMS necessita di autenticazione: lo studente deve possedere un account.

La creazione dello stesso avviene in uno dei seguenti seguenti modi:

* Da parte del personale universitario che gestisce il progetto MSDNAA ed è in possesso di un account amministratore su ELMS. Esso deve provvedere a collegarsi al Webstore per creare manualmente l'account allo studente che lo richiede;
* Automaticamente tramite un Web Service di ELMS, dopo le dovute autenticazioni (*Integrated Authentication*).

Microsoft fornisce le specifiche per la connessione con il Web Service nel documento *MSDNAA: ELMS Integrated User Verification Implementation Guide* [MSD08].

Autenticazione integrata

Lo scopo è di sviluppare un meccanismo di autenticazione integrata tra l'esistente sistema di autenticazione di Ateneo e il Server ELMS. Microsoft chiama questo processo *Integrated User Verification*.

Il documento ELMS Integrated User Verification Implementation Guide distingue 5 step:

1. **L'Utente si collega a ELMS:** Quando ELMS richiede l'autenticazione, effettua un redirect dell'utente sul sito di login dell'Università, passando come parametri un *Token* di sicurezza e un *returnURL* che contiene l'indirizzo a cui connettersi dopo l'autenticazione.
2. **Il sito dell'Università autentica l'utente:** Il sito universitario richiede all'utente le sue credenziali, e ne verifica l'identità tramite il proprio sistema di autenticazione.
3. **L'Utente sceglie di accedere al sito ELMS:** L'utente indica che vuole recarsi al sito ELMS per effettuare il download del software (per esempio cliccando su di un link o un button nel sito dell'Università).
4. **Il sito dell'Università abilita l'utente nel sito ELMS:** Il sito universitario manda una richiesta HTTPS al Web Service ELMS passando come parametri alcune informazioni sull'utente ed una chiave di sicurezza per identificarlo.
5. **Il sito dell'Università ridirige l'utente al sito ELMS:** Se la richiesta HTTPS è stata ricevuta con successo (viene inviato un codice di ritorno), il sito universitario ridirige l'utente al *returnURL* ricevuto dallo *Step 1.* Quest'azione completa il processo di verifica, abilita l'utente nel sito ELMS e ne visualizza l'homepage.

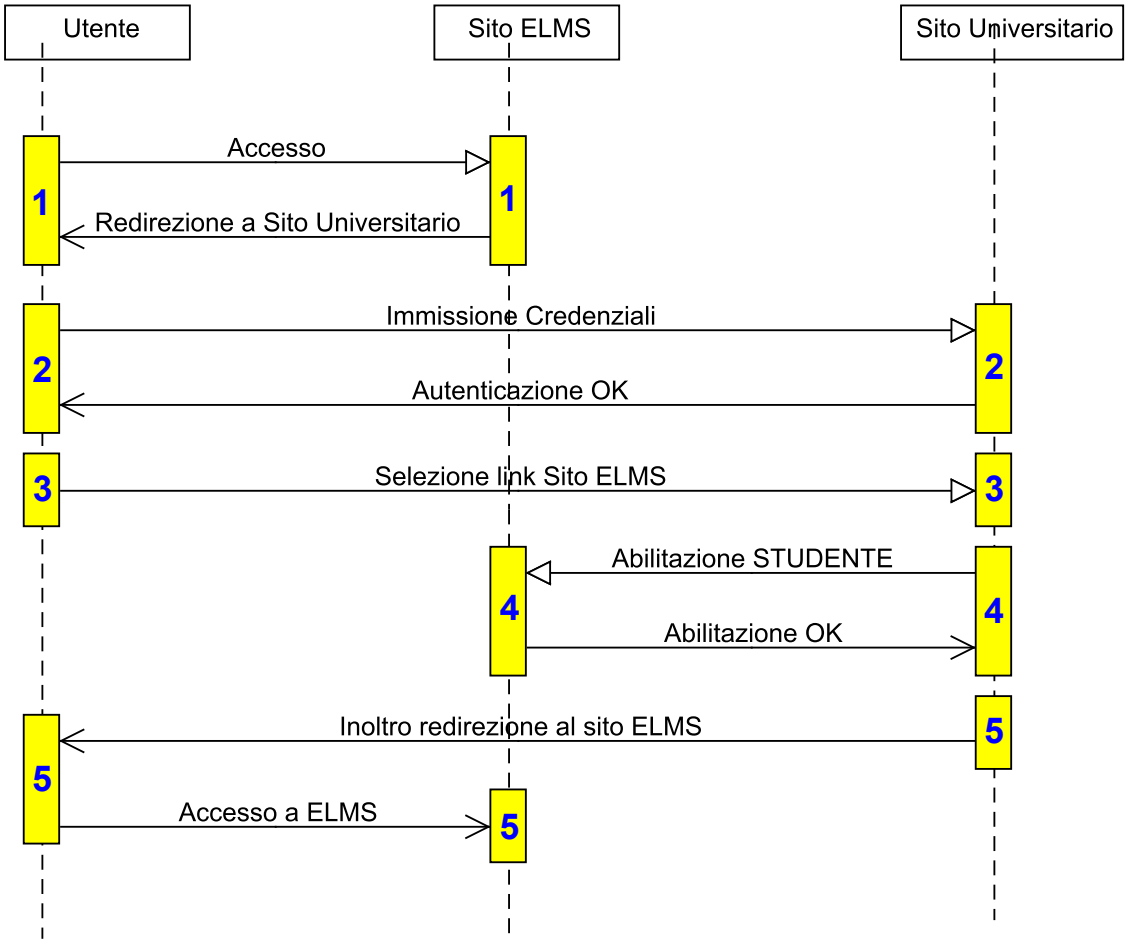


Figura 5.1: Integrated User Autentication - i 5 step.

All'atto dell'autenticazione presso ELMS (*Step 4*), è richiesto un identificativo dello studente che vuole accedere al servizio di download. ELMS controlla nel proprio database se esiste un account per quello studente. Se non esiste, lo crea. Altrimenti associa lo studente all'account esistente. In questo modo egli può avere uno storico del software scaricato, e ELMS può tenere traccia dei download effettuati da ogni singolo studente.

5.1.2 L'Università di Bologna

L'Università di Bologna gestisce le richieste di autenticazione inoltrandole ad un apposito Server *Radius* dedicato. Tale server contiene le credenziali di accesso di tutti gli studenti ed il personale di Ateneo, ed è utilizzato per verificare le credenziali di accesso in maniera centralizzata.

Ogniqualvolta un servizio necessita di autenticazione dell'utente, chiede ad esso le sue credenziali e le invia al server Radius. Quest'ultimo confronta le credenziali ricevute con quelle presenti nel database, ed invia la risposta al servizio richiedente. I possibili risultati sono due: Autenticazione effettuata o Autenticazione negata.

Per effettuare l'interrogazione al Server Radius, è necessario fornire:

* *Username*, nome dell'utente da autenticare;
* *Password*, la relativa password, inviata in chiaro;
* *Gruppo*, puo' essere STUDENTI o PERSONALE.
* *Secret*, una parola segreta, sempre uguale, che identifica il Servizio che richiede di interrogare il Server Radius.

5.2 Realizzazione del caso d'uso

In questa sezione viene descritto il caso d'uso realizzato. Prima di iniziare è necessario introdurre uno strumento aggiuntivo all'architettura da utilizzare.

5.2.1 RadiusClient

L'implementazione del modulo realizzato con questo lavoro di tesi rende Jolie in grado di gestire la comunicazione tramite HTTPS sia

con l'utente che con il sito ELMS. Tuttavia è necessario aggiungere uno strumento che consente l'interrogazione al Server Radius.

Con questo scopo è stato realizzato il JavaService *RadiusClient*. Esso si occupa di comunicare con il Server Radius, viene richiamato all'interno del programma JOLIE e comunica con esso. Rende disponibili due operazioni:

* **InitRadius** - Inizializza la connessione con il Server Radius. Prende in input:
  - **IP del Server Radius**;
  - **Secret**.

  Restituisce un messaggio vuoto indicante l'avvenuta connessione con il Server oppure il Fault **ConnectionFault** se essa non è avvenuta.
* **Authenticate** - Effettua l'interrogazione al Server Radius. Prende in input:
  - **Nome dell'utente**;
  - **Password**;
  - **Gruppo dell'utente**.

  In base alla risposta, RadiusClient fornisce in output al programma JOLIE un messaggio vuoto indicante l'avvenuta autenticazione oppure uno dei seguenti Fault:
  - **ConnectionFault** - Si è verificato un errore nella connessione al Server Radius;
  - **AuthenticationFault** - La risposta ricevuta dal Server Radius indica che le credenziali fornite sono errate.

Queste operazioni sono invocabili dall'interno del programma Jolie, e sono il mezzo di comunicazione con il JavaService RadiusClient.

5.2.2 Diagramma di Sequenza

L'implementazione del Servizio è aderente agli step descritti nella sezione 5.1.1 con l'eccezione dello *Step 3*, che è stato omesso.

Questa decisione è stata presa in accordo con i tecnici di laboratorio referenti del Servizio.

In questo modo, una volta che l'utente si è autenticato presso il Web Service (*Step 2*), viene immediatamente rediretto all'homepage del sito ELMS, togliendo la necessità di cliccare su un link per comunicare tale intenzione (lo *Step 3* appunto).

La figura seguente mostra il Diagramma di Sequenza comprensivo di interazione con il Server Radius:

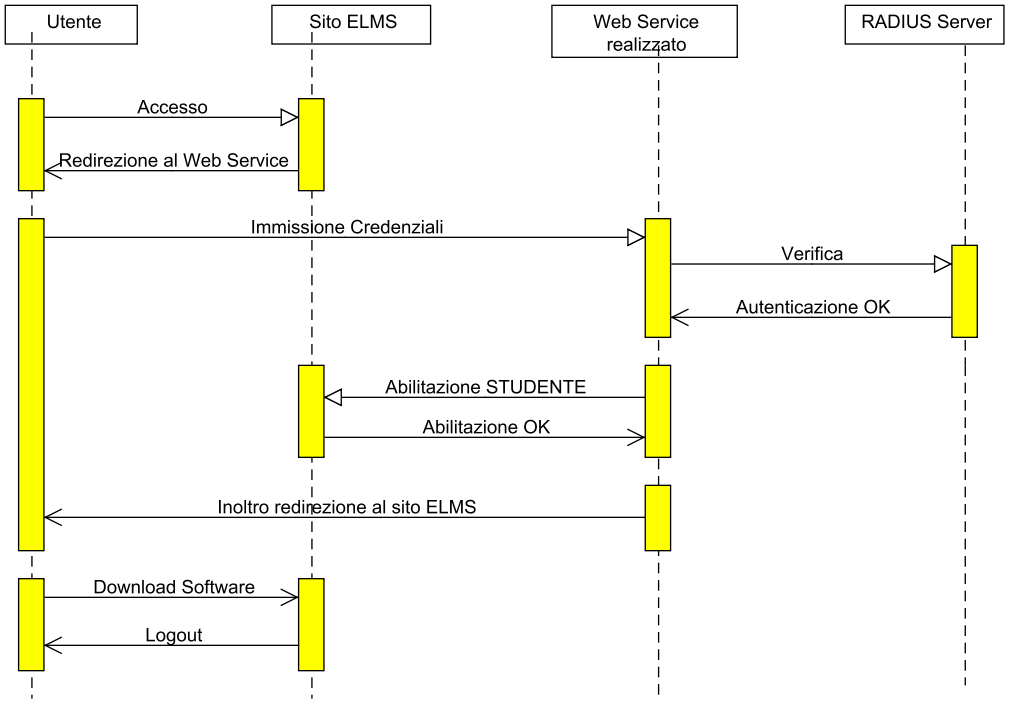


Figura 5.2: Diagramma di Sequenza del caso d'uso.

L'utente si collega al sito ELMS, e viene rediretto al Web Service. In questa fase esso si comporta come un Webserver e fornisce una pagina HTTPS di richiesta credenziali. Dopo l'immissione,

esse vengono inviate al Server Radius, che viene interrogato. Se l'autenticazione è avvenuta con successo, il Servizio abilita lo studente presso l'ELMS Webstore. Successivamente manda una pagina HTTPS di risposta all'utente che contiene un messaggio di avvenuta autenticazione e la redirezione al sito ELMS. Come conseguenza l'utente si troverà nella homepage di ELMS abilitato ad effettuare il download del software desiderato.

5.2.3 Descrizione del codice

Questa sezione è dedicata alla descrizione del codice JOLIE che realizza il Web Service. Esso utilizza tre porte:

* **MSDNAA** (output) - Utilizzata per l'abilitazione dell'utente presso il sito ELMS;
* **loginRequest** (input) - Riceve le connessioni degli utenti allo *Step 2*, fornendo in risposta una pagina HTTPS di richiesta credenziali;
* **Radius** (output) - Utilizzata per comunicare con il JavaService RadiusClient.

All'avvio del Servizio viene eseguita la procedura Init che legge da file il codice HTML da inviare successivamente all'utente.

Dopo si passa all'esecuzione del blocco principale (*main*). Esso contiene tre operazioni, tutte di tipo Request-Response, in una scelta non-deterministica (viene eseguita la prima invocata):

* **Login** - E' attivata dall'accesso di un utente al Servizio (Step 2). Fornisce in risposta una pagina HTTPS di richiesta credenziali;
* **AuthenticateMSDNAA** - Preleva le credenziali inviate dall'utente e accodate nell'URL con il metodo GET. Avvia la funzione *initRadius* del JavaService che inizializza la connessione con il server Radius. Se va a buon fine, avvia la funzione

Authenticate passando le credenziali per l'autenticazione. Se anche questa va a buon fine, avvia l'operazione *GrantAccess* che abilita l'utente presso il sito ELMS, e redirige l'utente sul sito ELMS.

Qualora una delle operazioni genera un Fault, l'utente viene rediretto alla schermata di Login.

I redirect vengono effettuati fornendo all'utente un pagina HTML con codice Javascript apposito.

* **Default** - L'interprete Jolie ricava il nome dell'operazione da invocare prelevandolo dall'URL che è stato richiesto. Nel caso in cui non ne esista una con il nome corrispondente, viene invocata questa operazione, che mette il servizio in modalità Webserver. Il nome dell'operazione viene utilizzato per cercare su disco un file che vi corrisponde. Se la ricerca da' esito, il contenuto del file è dato in risposta all'utente.

Esecuzione del Web Service

Il Servizio si attiva in ricezione sul socket in locale alla porta 443 (HTTPS). Ogni richiesta HTTPS ricevuta invoca l'operazione del nome corrispondente.

Ad esempio, una richiesta su *https://localhost/AuthenticateMSDNAA* invoca l'operazione *AuthenticateMSDNAA*.

Quando l'utente si collega al Web Service, effettua una richiesta HTTP GET di *https://localhost/Login* . Tale richiesta invoca l'operazione *Login*, che restituisce all'utente la pagina di richiesta credenziali. Questa schermata è corredata da immagini, ciascuna delle quali avvia a sua volta una richiesta HTTP GET dal browser dell'utente. Il Web Service riceve queste richieste, e non trovando un'operazione del nome corrispondente invoca l'operazione *Default*, che si occupa di cercare su disco un file di tale nome, e lo ritorna all'utente. Il suo browser riceve così le immagini.

Dopo che l'utente ha immesso le sue credenziali e premuto il tasto di login, viene avviata l'operazione *AuthenticateMSDNAA*. Essa le comunica i dati di accesso al JavaService per l'interrogazione al Server Radius. Se quest'ultima è avvenuta correttamente e le credenziali sono verificate, invoca l'operazione GrantAccess che effettua una richiesta HTTPS al sito ELMS includendo i parametri per identificare e abilitare l'utente. Il sito ELMS risponde con un messaggio che comunica l'esito dell'abilitazione. Se tutti gli step sono avvenuti con successo, AuthenticateMSDNAA ritorna all'utente una pagina HTML con il codice Javascript per la redirezione sul sito ELMS. Diversamente il redirect viene effettuato verso la schermata di Login.

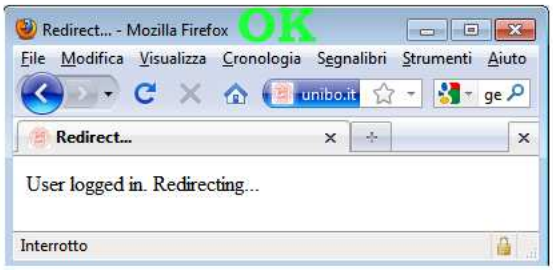
5.3 Il caso reale: Accesso a ELMS Webstore

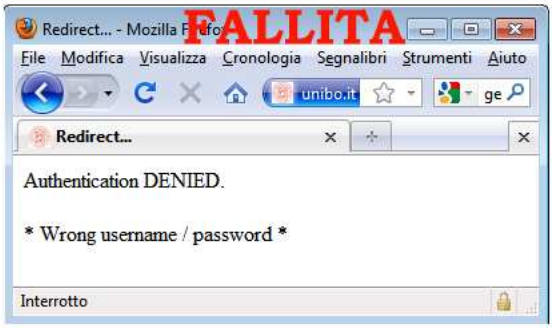
In questa sezione viene descritto un accesso reale al sito ELMS. La maggior parte dei dettagli implementativi è nascosto all'utente, che deve limitarsi ad inserire i dati di accesso. I passi sono i seguenti:

* L'utente si connette al sito ELMS: viene rediretto al Servizio, che richiede le credenziali di accesso:

∗ L'utente immette le proprie credenziali e preme il tasto Invia:

∗ A seconda della riuscita autenticazione o meno, l'utente riceve la corrispondente schermata di risposta:





* Se l'autenticazione è avvenuta con successo, l'utente viene reindirizzato al sito ELMS, altrimenti viene riportato alla schermata di Login.

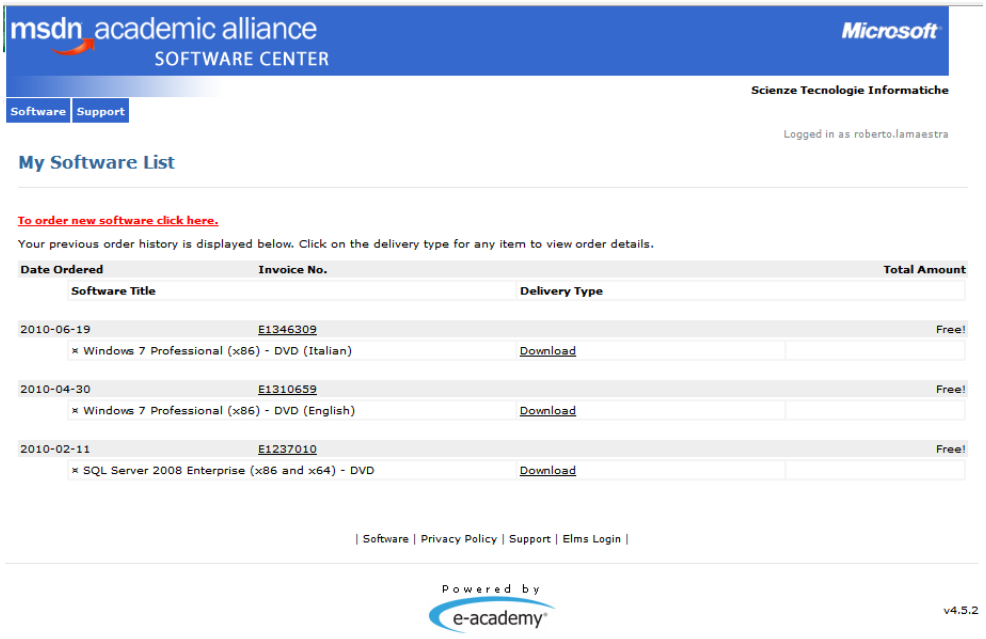


Figura 5.3: Il sito dell'ELMS Webstore.

Di seguito viene riportato il diagramma delle attività dal punto di vista dell'utente:

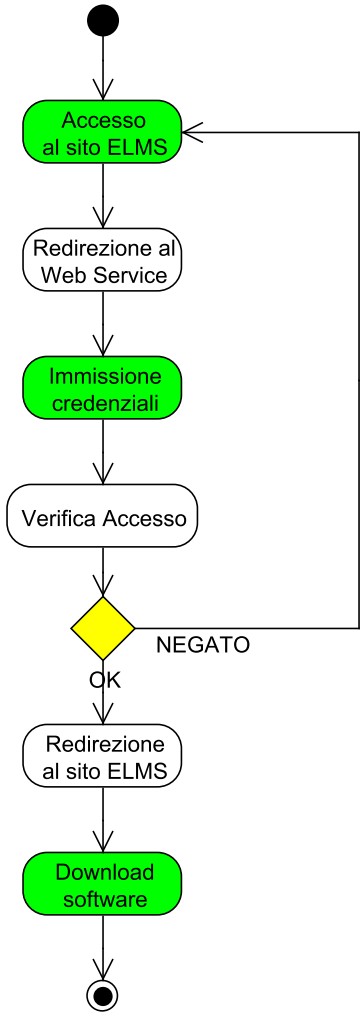


Figura 5.4: Diagramma delle attività. I blocchi in bianco sono invisibili all'utente.

Conclusioni

Con il lavoro di questa tesi è stato dato il supporto del protocollo HTTPS a JOLIE, rendendolo in grado di offrire allo sviluppatore una comunicazione sicura. Come diretta conseguenza, il campo di applicazione del linguaggio è notevolmente aumentato: JOLIE è adesso in grado di supportare servizi di e-commerce, operare transazioni bancarie, fornire sistemi di autenticazione o richiesta password. Grazie all'aggiunta di questo nuovo supporto, JOLIE coniuga adesso facilità di utilizzo e completezza degli strumenti che offre per la creazione di orchestratori dei servizi.

La struttura interna con cui è stato realizzato il modulo HTTPS e l'architettura di JOLIE permettono, con la modifica di poche righe di codice, la creazione di moduli analoghi che implementano la versione sicura di altri protocolli supportati (creando così SOAPS, SODEPS, ecc..) che potrebbe avvenire in un prossimo futuro.

Bibliografia

[WSA] http://www.w3.org/TR/ws-arch/.

[JAX] https://jax-rpc.dev.java.net/.

[XML] http://www.xmlrpc.com/.

[SOA] http://www.w3.org/TR/soap/.

[WSD] http://www.w3.org/TR/wsdl/.

[UDDa] http://www.uddi.org/.

[UDDb] UDDI Browsing. http://soapclient.com/uddisearch.html.

[New05] Greg Newcomer, Eric; Lomow. *Understanding SOA with Web Services*. Addison Wesley, 2005.

[NB06a] Claudio Guidi Roberto Lucchi Gianluigi Zavattaro Nadia Busi, Roberto Gorrieri. *SOCK: a calculus for service oriented computing.* In Proc. of 4th International Conference on Service Oriented Computing (ICSOC 2006).

[NB06b] Claudio Guidi Roberto Lucchi Gianluigi Zavattaro Nadia Bu- si, Roberto Gorrieri. *Choreography and Orchestration conformance for system design.* Technical report, In Proc. of 8th International Conference on Coordination Models and Languages (COORDINATION 2006). Volume 4038 of LNCS, pages 63-81, 2006.

[NBZ05] C. Guidi R. Lucchi N. Busi, R. Gorrieri and G. Zavattaro. *Towards a formal framework for Choreography.* In Proc. of International Workshop on Distributed and Mobile Collabora-

tion (DMC 2005), IEEE Computer Society Press. WETICE, pages 107-112. 2005.

[BPE] http://www.oasis-open.org/committees/ tc_home.php?wg_abbrev=wsbpel.

[WSC] http://www.w3.org/TR/ws-cdl-10/.

[JOL] http://www.jolie-lang.org/.

[FM06] Roberto Lucchi Gianluigi Zavattaro Fabrizio Montesi, Claudio Guidi. *JOLIE: a Java Orchestration Language Interpreter Engine.* Technical report, In CoOrg 06, Volume to appear of ENTCS, 2006.

[SOD] http://jolie.sourceforge.net/contents/sodep.html.

[FMZ06] Ivan Lanese Fabrizio Montesi, Claudio Guidi and Gian- luigi Zavattaro. *Dynamic fault handling mechanisms for service-oriented applications.* Department of Computer Science, University of Bologna, Italy. 2006.

[DS04] Alfredo De Santis. *Corso di Sicurezza su Reti.* Università di Salerno, 2004. http://www.dia.unisa.it/˜ads/corso-security/www/CORSO-9900/SSL/main.htm .

[SSL96] Netscape Communications. *The SSL Protocol Version 3.0.* 1996. http://home.netscape.com/eng/ssl3/draft302.txt.

[IET99] Internet Engineering Task Force. *The TLS Protocol Version 1.0.* 1999. http://www.ietf.org/rfc/rfc2246.txt .

[IET00] Internet Engineering Task Force. *HTTP over TLS.* Maggio 2000. http://tools.ietf.org/html/rfc2818.

[IET08] Internet Engineering Task Force. *The Transport Layer Security (TLS) Protocol Version 1.2.* 2008. http://www.ietf.org/rfc/rfc5246.txt .

[SUN06] Sun Microsystems, Inc.

Java Secure Socket Extension Reference Guide. 2006. http://download.oracle.com/javase/6/docs/technotes/guides/security/jsse/JSSERefGuide.html.

[MI03] Microsoft Corporation. *SSL/TLS in Detail.* Luglio 2003. http://technet.microsoft.com/en-us/library/cc785811%28WS.10%29.aspx.

[NS04] Nuno Santos *Using SSL with Non-Blocking IO.* Marzo 2004. http://onjava.com/pub/a/onjava/2004/11/03/ssl-nio.html.

[SSS] http://download.oracle.com/javase/1.4.2/docs/ /api/javax/net/ssl/SSLSocket.html.

[SSE] http://download.oracle.com/javase/6/docs/ /api/javax/net/ssl/SSLEngine.html.

[ELMa] https://msdn60.e-academy.com/elms/ /Security/Login.aspx?campus=msdnaa_km3967&tma=1.

[MSD] http://msdn.microsoft.com/it-it/academic/default.aspx.

[MSD08] E-Academy Inc. *ELMS-MSDNAA, Integrated User Verification Customer Implementation Guide.* Febbraio 2008.