

# JavaRMI

Frasca Luigi      Gargano Riccardo

12 Dicembre 2014

## **Sommario**

Dopo aver studiato il vasto programma del corso di Reti di Calcolatori abbiamo incentrato il nostro progetto su una particolare tecnologia, la Remote Method Invocation(RMI).

Abbiamo progettato e implementato un'applicazione di messaggistica basata su RMI.

L'applicazione é stata realizzata in Java che offre una tecnologia specifica, la Java Remote Method Protocol, per la ricerca e il riferimento a oggetti remoti.

Con questa stesura vogliamo spiegare gli argomenti analizzati e trattati che ci hanno permesso di riuscire nel nostro scopo.

# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
1.1	Socket . . . . .	2
1.2	RPC . . . . .	2
1.3	Middleware . . . . .	3
<b>2</b>	<b>Java Remote Method Invocation</b>	<b>4</b>
2.1	Cosé Java RMI . . . . .	4
2.2	Stub & Skeleton . . . . .	5
2.3	Architettura RMI . . . . .	6
2.4	Creare un'applicazione in Java RMI . . . . .	8
2.4.1	Progettazione ed Implementazione delle componenti di- stribuite . . . . .	8
2.4.2	Compilazioni sorgenti e generazioni stub . . . . .	10
<b>3</b>	<b>La nostra applicazione</b>	<b>11</b>
3.1	Login . . . . .	11
3.1.1	Autenticazione CAS . . . . .	11
3.2	Interfaccia Utente . . . . .	13
3.3	Scambio di Messaggi . . . . .	14
3.4	Crittografia . . . . .	15
3.4.1	Introduzione . . . . .	15
3.4.2	Algoritmi a chiave simmetrica . . . . .	15
3.4.3	Algoritmi a chiave pubblica . . . . .	16
3.4.4	Implementazione nell'applicazione . . . . .	17
3.5	Sviluppi futuri . . . . .	17
3.6	Strumenti utilizzati . . . . .	18

# Capitolo 1

## Introduzione

### 1.1 Socket

I Sistemi Distribuiti, per loro natura, prevedono che computazioni differenti possano essere eseguite su Virtual Machine differenti, possibilmente su host differenti, comunicanti tra loro.

Al fine di consentire la programmazione di sistemi distribuiti (in rete) il linguaggio Java supporta la nozione di socket, un meccanismo flessibile e potente per la programmazione di sistemi distribuiti.

I socket sono un meccanismo di basso livello e, come tutti i linguaggi a basso livello, presentano una serie di difficoltà per il programmatore. Con i socket le difficoltà maggiori derivano dall'eterogeneità delle entità in rete. Occorre definire un "protocollo" per l'invio delle richieste di servizio e delle risposte, con relativa codifica e decodifica dei parametri in sequenze di byte.

### 1.2 RPC

Un'alternativa ai sockets è rappresentato da una tecnologia di più alto livello, comunemente chiamata RPC (Remote Procedure Call), in cui l'interfaccia di comunicazione è rappresentata dall'invocazione (remota) di procedura. La tecnologia RPC consente di invocare procedure che appartengano ad applicazioni remote, in maniera del tutto trasparente all'utente.

Più precisamente il client invoca una procedura del server remoto, il quale si occupa di eseguire la procedura (con i parametri passati dal client) e di ritornare a quest'ultimo il risultato dell'esecuzione.

La connessione remota è trasparente al client che ha l'illusione di invocare una procedura locale.

Grazie alla RPC, il programmatore non deve più preoccuparsi di sviluppare

dei protocolli che si occupino del trasferimento dei dati, della verifica, e della codifica/decodifica. Queste operazioni sono interamente gestite dalla RPC. La tecnologia RPC presenta comunque dei limiti:

- parametri e risultati devono avere tipi primitivi.
- la programmazione é essenzialmente procedurale.
- non vi é programmazione ad oggetti e quindi mancano i concetti di ereditariet , incapsulamento, polimorfismo, etc..

## 1.3 Middleware

A partire dagli inizi degli anni '90 sono state proposte delle tecnologie, dette Middleware, per superare i limiti di RPC:

- CORBA: supporta applicazioni scritte in linguaggi differenti su piattaforme differenti.
- Java RMI: supporta applicazioni Java su una piattaforma Java, le applicazioni possono essere distribuite su differenti Java Virtual Machine.
- DCOM: supporta applicazioni scritte in linguaggi differenti, ma su piattaforme Win32. Esistono delle implementazioni per sistemi Unix.
- .NET remoting: supporta applicazioni scritte in linguaggi differenti, su piattaforma Windows.

# Capitolo 2

## Java Remote Method Invocation

### 2.1 Cosé Java RMI

Java RMI é una tecnologia che consente a processi distribuiti di comunicare attraverso la rete.

L'idea alla base di tutta la programmazione distribuita é semplice

- Un client esegue una determinata richiesta. Tale richiesta viaggia lungo la rete verso un determinato server destinatario
- Il server processa la richiesta e manda indietro la risposta al client per essere analizzata
- Con i socket però dobbiamo gestire personalmente il formato dei messaggi e la gestione della connessione

Java RMI permette l'accesso ad oggetti remoti (cioé su altri nodi della rete Internet) come se fossero oggetti locali, senza doversi preoccupare della realizzazione della connessione, degli stream di input e output, etc..

Quindi possiamo definire Java RMI un insieme di strumenti, politiche e meccanismi che permettono l'invocazione remota di metodi su oggetti che risiedono su diverse Java Virtual Machine.

Vantaggi nell'utilizzare Java RMI:

- É un linguaggio ad oggetti, i concetti di riutilizzabilità (ereditarietà), protezione dell'informazione (incapsulamento), ed accesso dinamico (polimorfismo) sono già definiti.
- Consente il passaggio di referenze ad oggetti remoti.

- Supporta un Java Security Manager per controllare che le applicazioni distribuite abbiano i diritti necessari per essere eseguite.
- Supporta un meccanismo di Distributed Garbage Collection (DGB) per disallocare quegli oggetti remoti per cui non esistano più referenze attive.

## 2.2 Stub & Skeleton

Questa architettura adotta il proxy pattern, i due proxy (**stub** dalla parte client e **skeleton** dalla parte server) nascondono al livello applicativo la natura distribuita dell'applicazione.

- Il server istanzia degli oggetti remoti e li rende visibili.
- Gli oggetti remoti implementano una determinata interfaccia remota che dichiara i metodi accessibili dall'esterno.
- I client ottengono dei riferimenti agli oggetti remoti, Stub, tramite i quali possono invocare i metodi remoti dichiarati.

Quindi, un client non accede a un server remoto direttamente, ma attraverso lo stub, il quale va in esecuzione sulla macchina del client ed agisce da rappresentante locale (proxy) al server remoto.

Un oggetto stub traduce automaticamente ogni invocazione al server remoto in termini di comunicazione di rete, con relativo passaggio di parametri ed eventuale codifica/decodifica (marshalling / unmarshalling).

Definiamo allora lo stub come una referenza al server remoto. Referenza di cui il client deve necessariamente venire in possesso se vuole invocare i metodi del server remoto.

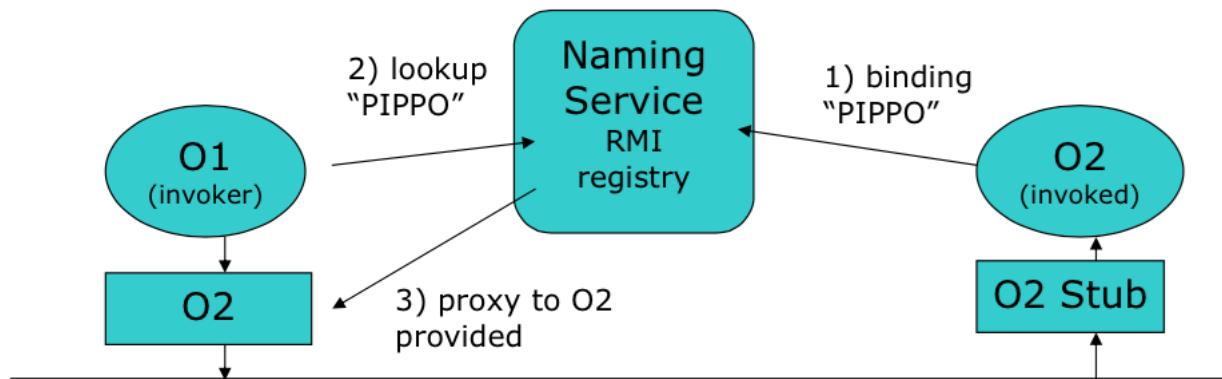
Lo stato di un oggetto stub contiene essenzialmente tre informazioni:

- l'IP dell'host su cui gira il server.
- la porta su cui é in esecuzione il server.
- un identificativo RMI associato al server remoto.

Esiste, invece, sul server un proxy del client lo skeleton. Il suo compito é quello di ricevere presso l'host del server le richieste da parte dello stub decodificarle, inoltrarle al server, codificare i risultati ed inviarli allo stub affinché li passi al client.

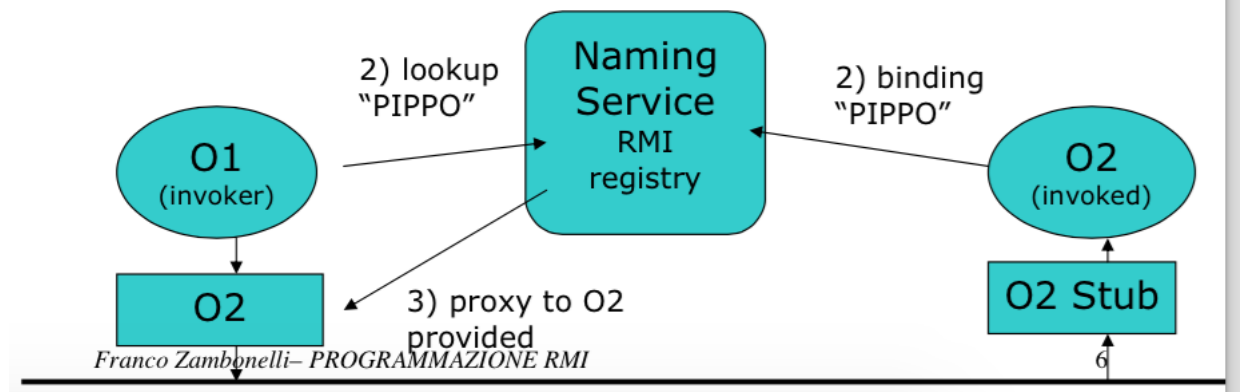
## 2.3 Architettura RMI

Esiste un programma speciale che agisce da "Naming Service", in Java é il RMI Registry.



- registra i nomi e i riferimenti degli oggetti i cui metodi possono essere invocati da remoto (O2 nella figura).
- tali oggetti devono registrarsi, fare il "bind" presso il naming service, con un nome pubblico("PIPPO" nella figura).
- altri oggetti possono richiedere, fare il "lookup" di oggetti registrati chiedendo, a partire dal nome pubblico, un riferimento all'oggetto

Quando il Naming Service riceve una richiesta di lookup, corrispondente a un oggetto che lui conosce (nella figura, il Naming Service sa che l'oggetto di nome pubblico "PIPPO" corrisponde all'oggetto O2, allora:





- viene generato un oggetto "virtuale", lo "skeleton", che viene inviato presso il nodo di colui che aveva richiesto il lookup (O1 in figura).
- dal punto di vista di O1, esso percepisce il proxy di O2 come un normale riferimento a un oggetto O2 locale.

In verità, quando O1 chiede un servizio a O2, lo chiede al suo proxy che:

- senza che O1 se ne accorga, stabilisce una connessione Socket TCP con il nodo dove c'è O2 vero.
- in tale nodo di O2, un componente associato a O2, lo "stub" riceve la connessione e la richiesta di servizio, e provvede a invocarla lui su O2 vero.
- quando O2 vero risponde, lo stub manda la risposta indietro al proxy, il quale poi risponde a O1 come se fosse stato realmente lui a eseguire il servizio.

## 2.4 Creare un'applicazione in Java RMI

### 2.4.1 Progettazione ed Implementazione delle componenti distribuite

Si decide la struttura dell'applicazione e si stabilisce quali funzionalità devono essere espletate da oggetti locali e quali da server remoti.

Vi sono sostanzialmente tre diversi moduli la cui progettazione richiede particolare cura:

- *Definizione dell'interfaccia remota.* Un'interfaccia remota é una particolare interfaccia Java che denota quei metodi del server che possono essere invocati remotamente da un client.

Tale interfaccia remota deve essere conosciuta sia dal server che dal client. Quest'ultimo, infatti, deve conoscere il nome dei metodi remoti ed il tipo dei parametri e dei risultati che vengono passati.

L'interfaccia remota rappresenta quindi la modalità attraverso cui un client ed un server interagiscono tra loro.

Un'interfaccia remota per definirsi tale deve estendere l'interfaccia

```
java.rmi.Remote
```

Tutti i metodi che appartengono ad un'interfaccia remota devono lanciare

```
java.rmi.RemoteException
```

- *Implementazione dei server remoti.* I server remoti devono implementare una o più interfacce remote. Un server può implementare più di una interfaccia remota nel caso voglia fornire "views" differenti a client differenti. L'implementazione di un oggetto remoto deve estendere la classe

```
java.rmi.server.RemoteObject
```

o una sua sottoclasse, e deve implementare tutte le interfacce remote che intende supportare.

- *Implementazione dei client.* Clienti che interagiscono con server remoti possono essere implementati in ogni momento, anche successivamente all'implementazione del server, purché facciano un uso appropriato dell'interfaccia remota per accedere il server.

- *Lancio del registro RMI.* Prima di lanciare il server remoto (e quindi l'applicazione client) é necessario lanciare presso il server il registro di Naming RMI.

In modo che il client possa recuperare le referenze ai server remoti di cui ha bisogno. Il registro di naming puó essere lanciato da linea di comando:

```
rmiregistry
```

Il registro RMI presso l'host del server é semplicemente identificato con l'IP dell'host e la porta su cui é in esecuzione il registro (di default la 1099).

- *Lancio del server.* Dopo essere stato lanciato il server viene messo in ascolto su una porta (definibile dall'utente) da cui accetta richieste di invocazioni remote da parte del sistema RMI.
- *Registrazione del server remoto sul registro RMI.* Il server remoto puó essere registrato sul registro RMI, in modo che gli altri client possano ottenere la referenza remota (stub) per accedere al server.

## 2.4.2 Compilazioni sorgenti e generazioni stub

La compilazione avviene in due fasi:

- Nella prima si utilizza il compilatore **javac** per compilare i sorgenti del server e del client. Più precisamente presso il server si compilano le interfacce remote e l'implementazione del server, mentre sulla macchina client si compilano l'interfaccia remota e l'applicazione client (l'interfaccia remota deve essere conosciuta sia dal server che dal client).
- Nella seconda fase si usa il compilatore **rmic** per generare nella macchina server le classi stub e skeleton relative ai server remoti.  
**NOTA** A partire dalla J2SE 5.0 tale compilazione non sarebbe più necessaria in quanto le classi stub vengono create a tempo d'esecuzione. Il compilatore **rmic** va comunque usato quando si ha a che fare con clients che utilizzano versioni precedenti di Java.

```
rmic ServerRemoto
```

Questo comando produce le classi

```
ServerRemoto_Stub.class e ServerRemoto_Skel.class.
```

Entrambi le classi sono necessarie per l'esecuzione dell'implementazione dell'oggetto remoto.

Un client in possesso di una referenza remota, ovvero un oggetto stub, per poterlo utilizzare deve accedere alla classe di appartenenza dell'oggetto stub:

```
ServerRemoto Stub.class
```

**NOTA** quando un client riceve uno stub oltre a ricevere l'oggetto stub caricherà dinamicamente in maniera trasparente la classe dell'oggetto stub attraverso un meccanismo detto **codebase** che in Java RMI gioca un ruolo molto importante durante la trasmissione di referenze remote.

# Capitolo 3

## La nostra applicazione

### 3.1 Login

Per poter utilizzare la nostra applicazione é necessario effettuare un login con credenziali(user e password).

Queste credenziali vengono controllate tramite la funzione:

```
boolean validate(String user, String pwd, ClientInterface client)
```

che attraverso l'invocazione remota ci permette di controllare le credenziali sul server, se queste sono corrette il server registra in un **HashMap usersOn** l'username e l'interfaccia remota che servirá per comunicare con lo specifico client.

Le credenziali se sbagliate per 3 volte consecutive producono l'uscita dall'applicazione.

La password prima di essere inviata al server viene codificata con un Message Digest, in questo caso MD5, questo tipo di codifica prende in input una stringa di lunghezza arbitraria e ne produce in output un'altra a 128 bit.

L'MD5 é standardizzato con la RFC 1321.

#### 3.1.1 Autenticazione CAS

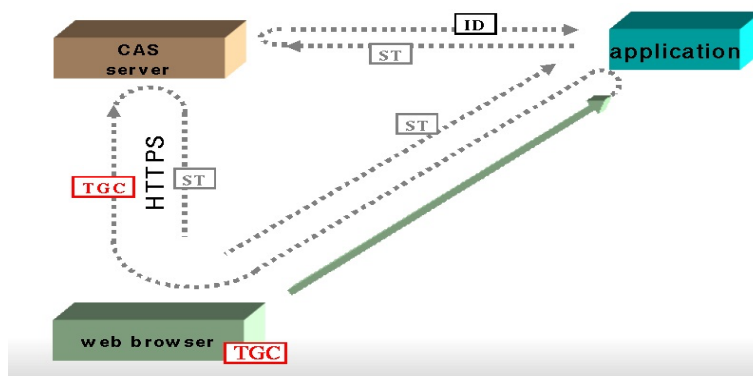
Central Authentication Service (CAS) é un servizio per l'autenticazione Single Sign-On (SSO) di una comunitá Web.

Le applicazioni demandano il problema dell'autenticazione al server centrale CAS, senza piú preoccuparsi dei vari problemi di identificazione dell'utente. L'utente dovrá autenticarsi una sola volta, ad esempio con username e password, o con l'invio di un certificato digitale, e da quel momento in poi potrà utilizzare tutte le applicazioni fornite dal sistema, senza doversi autenticare

nuovamente, finché la sessione è valida.

Il CAS è composto principalmente da due componenti:

- Il CAS server, che gestisce l'identità della comunità'.
- I Service Providers, che forniscono le applicazioni che richiedono accesso autenticato.



Gli scopi principali del server CAS sono:

- Facilitare l'autenticazione centralizzata SSO per più applicazioni comuni.
- Semplificare la procedura stessa di autenticazione
- Consentire un'autenticazione sicura agli utenti, senza che questi debbano rivelare le proprie credenziali ai servizi e agli applicativi ogni volta.

Per quel che riguarda l'autenticazione, le credenziali dell'utente viaggiano solo una volta tra il browser dell'utente ed il server su un canale protetto e crittografato in SSL.

Durante l'autenticazione, il server CAS cerca di salvare nel browser dell'utente uno speciale cookie privato: il Ticket Granting Cookie, TGS. Questo non contiene nessun tipo di dato confidenziale, ma contiene solamente un identificativo di sessione noto al CAS.

Il ticket ha comunque un tempo di vita relativamente breve, e viene cancellato definitivamente alla chiusura del browser da parte dell'utente.

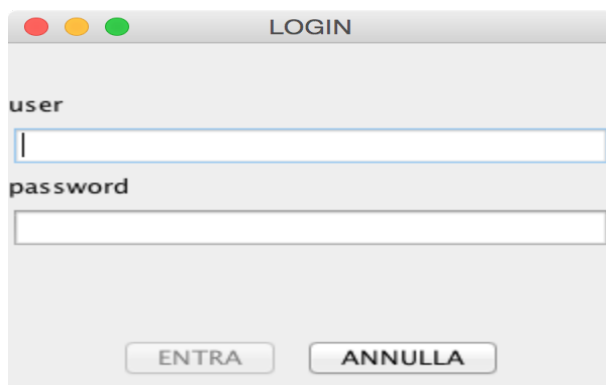
Questo speciale ticket, serve come garanzia di avvenuta autenticazione qualora l'utente intenda accedere ad un'altra applicazione del sistema. In questo modo l'utente in tutta trasparenza non deve re-autenticarsi ogni volta che cambia il servizio applicativo richiesto.

Le applicazioni invece non comunicano attraverso il ticket TGT, ma attraverso altri ticket appositi, detti Service Ticket, ST.

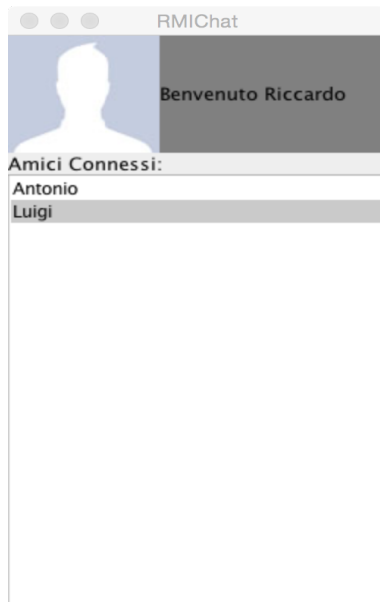
Essi non contengono informazioni personali, e possono essere utilizzati solo una volta (one time ticket). Il ST viene emesso dal server CAS verso l'utente, il quale in modo automatico lo passa al servizio applicativo. Questi lo ridirige di nuovo verso il CAS per confermare l'identità dell'utente, chiudendo il ciclo di autenticazione.

## 3.2 Interfaccia Utente

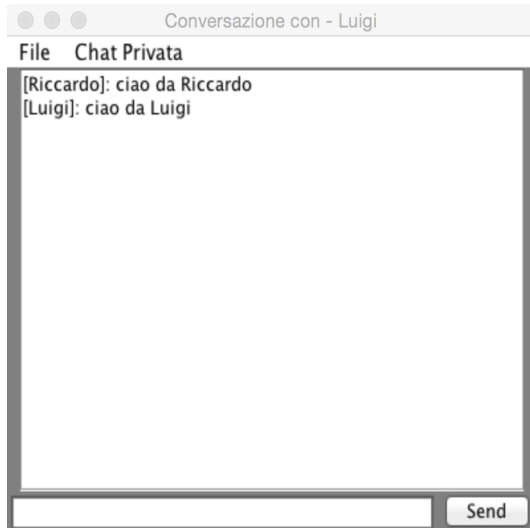
Il login della nostra applicazione si presenta così:

A screenshot of a login window titled "LOGIN". It features two input fields: "user" and "password". Below the fields are two buttons: "ENTRA" and "ANNULLA". The window has a standard macOS-style title bar with red, yellow, and green window control buttons.

Dopo aver effettuato il login, visualizziamo la finestra che elenca tutti gli utenti connessi alla chat:

A screenshot of a chat application window titled "RMIChat". The window is divided into several sections. At the top left is a placeholder for a user's profile picture. To its right, a dark grey bar displays the text "Benvenuto Riccardo". Below this, a section titled "Amici Connessi:" lists two names: "Antonio" and "Luigi". The bottom half of the window is a large, empty white area, likely for chat messages.

In seguito é possibile avviare una singola conversazione con l'amico desiderato:



### 3.3 Scambio di Messaggi

All'interno della nostra applicazione definiamo due interfacce remote :

```
public interface ServerInterface extends Remote
public interface ClientInterface extends Remote
```

Nello specifico la ServerInterface definisce il seguente metodo

```
public void sendMessageTo(int mode, String mitt, String dest, byte[] msg)
```

il quale dopo aver effettuato una ricerca all'interno dell'HashMap **usersOn**, ove gli verrà restituita la ClientInterface di destinazione sulla quale chiamare il metodo definito in essa :

```
public void tell(int type, String user,byte[] msg)
```

che permetterà lato client di visualizzare il messaggio su un'apposita finestra di comunicazione. Lo scambio di messaggi può avvenire in modo criptato.



## 3.4 Crittografia

### 3.4.1 Introduzione

Il crescente utilizzo di internet come mezzo per lo scambio rapido di informazioni, ha enfatizzato la necessità di comunicazioni sicure, private, protette da sguardi indiscreti.

Sfortunatamente la rete, così come è stata concepita, non supporta un buon livello di sicurezza e di privacy. Le informazioni viaggianti sono trasmesse in chiaro e potrebbero essere intercettate e lette da qualsiasi individuo.

Pertanto è stato necessario creare dei metodi che, rendano le informazioni indecifrabili in modo che solo il mittente e il destinatario possano leggerle, e ne assicurino l'integrità.

### 3.4.2 Algoritmi a chiave simmetrica

Gli algoritmi a chiave simmetrica usano la stessa chiave per cifratura e decifratura. Entrambi gli interlocutori conoscono la chiave usata per la cifratura, detta chiave privata, e soltanto loro possono cifrare e decifrare il messaggio.

$\text{MessaggioCriptato} = E(\text{Chiave}, \text{Messaggio})$

$\text{Messaggio} = D(\text{Chiave}, \text{MessaggioCriptato})$

dove E e D sono cipher noti.



### 3.4.3 Algoritmi a chiave pubblica

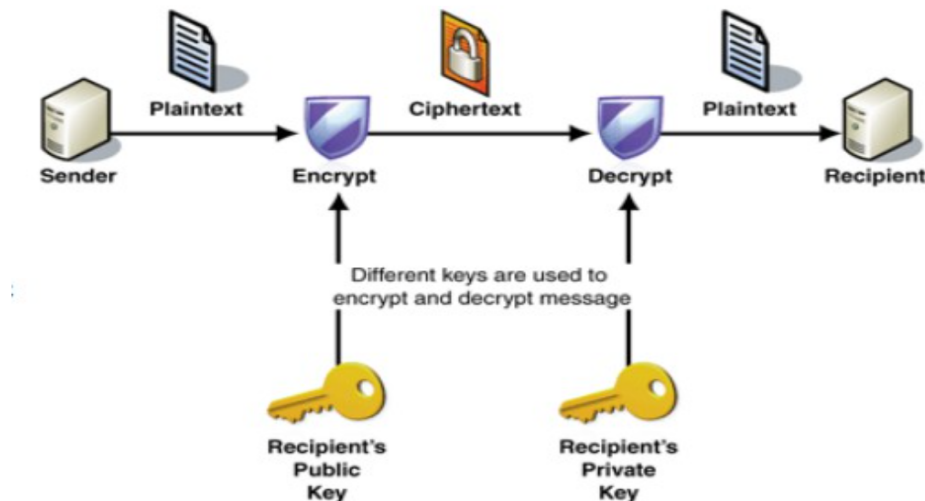
La cifratura a chiave simmetrica ha una grave debolezza nel trasferimento della chiave per renderla condivisa.

Diffie ed Hellmann nel 1976 proposero una tecnica nuova di crittografia, basata sull'aritmetica modulare, in cui vengono utilizzate due chiavi  $K_e$  e  $K_d$  distinte per la codifica

$\text{MessaggioCriptato} = K_e(\text{Messaggio})$

$\text{Messaggio} = K_d(\text{MessaggioCriptato})$

Assegnando ad una chiave il ruolo di "chiave privata" e all'altra il ruolo di "chiave pubblica" si supera la debolezza della chiave condivisa.



Una importante proprietà di questo algoritmo é:

$$K_d(K_e(\text{Messaggio})) = K_e(K_d(\text{Messaggio})) = \text{Messaggio}$$

Questo consente di poter applicare le chiavi in 2 modi diversi ottenendo 2 diverse funzioni:

- **Privacy (crypt/decrypt):**

A deve inviare un messaggio  $P$  riservato a B su di un canale insicuro. B possiede una coppia di chiave asimmetriche  $K_{Be}$  (privata) e  $K_{Bd}$  (pubblica). A cifra  $P$  con la chiave pubblica di B:  $C = K_{Bd}(P)$ . Solo B può decifrarlo  $P = K_{Be}(C)$

- **Autenticazione e Integrità (sign/verify):**

A deve inviare un messaggio  $P$  attraverso un canale insicuro. Tutti lo

possono leggere, ma chi lo riceve deve essere sicuro che é stato inviato da A. A possiede una coppia di chiave asimmetriche  $A_e$  (privata) e  $A_d$  (pubblica). A cifra  $P$  con la propria chiave privata:  $C = A_e(P)$ . Chiunque può applicare la chiave pubblica di A:  $P = A_d(C)$ . La decifratura funziona solo se  $P$  é stato cifrato da A.

### 3.4.4 Implementazione nell'applicazione

Nella nostra applicazione abbiamo utilizzato l'algoritmo a chiave pubblica per lo scambio di messaggi in modo sicuro.

Dopo essersi autenticati con successo la nostra applicazione si occupa della creazione della coppia di chiavi pubblica-privata attraverso la classe **Encryptor**. Quando si vuole inviare un messaggio criptato il client chiede al server di recapitargli la chiave pubblica del destinatario. In questo modo il client può scegliere di cambiare la coppia di chiavi in qualunque momento. L'algoritmo a chiave pubblica utilizzato é l'RSA con chiave 1024bit.

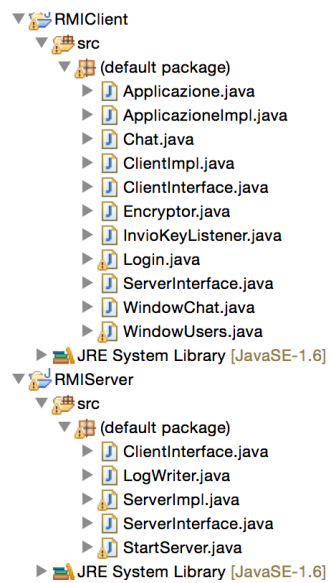
## 3.5 Sviluppi futuri

Si potrebbe dare continuità a questo progetto aggiungendo alcune features:

- Si potrebbe pensare di implementare l'autenticazione in CAS da noi solo studiata ma non implementata.
- la nostra applicazione non tiene conto delle amicizie tra utenti.

## 3.6 Strumenti utilizzati

Il progetto é stato scritto interamente con Eclipse Version: Kepler Service Release 2



La relazione é stata scritta in LaTeX