Stefano Bonora
Fabio Bucci

SERVIZIO DI AUTENTICAZIONE E DISTRIBUZIONE DI CHIAVI PUBBLICHE

1. Introduzione: Crittografia a chiave pubblica.

La sicurezza è uno dei problemi fondamentali del nostro tempo e la mancanza di sicurezza nella maggior parte delle reti ha contribuito alla ricerca di soluzioni che hanno aiutato molti utenti a mantenere segrete le loro comunicazioni. I fattori che hanno contribuito alla mancanza di sicurezza delle reti (puntiamo l'attenzione su Internet) si possono sintetizzare in:
- vulnerabilità del protocollo di comunicazione;
- trasmissione di dati su canali insicuri e condivisi;
- facilità di monitoraggio dell'attività di rete;
- debolezza dei meccanismi di autenticità e di controllo degli accessi.

La crittografia gioca un ruolo importante nell'implementazione di un sistema sicuro.
E' usata per nascondere informazioni private che sono esposte in parti del sistema, come canali di comunicazione fisici, che sono vulnerabili ad attacchi come "eavesdropping" e "messagge tampering". In questo modo tali informazioni possono essere decifrate solo dal proprietario della chiave di cifratura.
E' usata come supporto al meccanismo di **autenticazione** tra coppie di utenti. Un utente che decifra un messaggio usando una "chiave inversa" può assumere che il messaggio è autentico se contiene determinati valori attesi. E' improbabile pensare di decifrare il messaggio con un'altra chiave, e il ricevente può inferire che il mandante del messaggio sia il possessore della chiave di cifratura. Se la chiave è mantenuta segreta, o, nel caso di chiavi pubbliche, se una delle coppie di chiavi è mantenuta segreta, una successiva decifratura autentica il messaggio cifrato come derivante da un particolare sender.
E' usata per implementare un meccanismo conosciuto come "digital signature", che simula il ruolo della firma convenzionale e verifica che una copia non è stata alterata.

L'algoritmo più semplice è quello a chiave segreta o simmetrica: due utenti che vogliano comunicare devono condividere una chiave segreta che permette solo a loro di cifrare e decifrare il messaggio.
E' evidente come il numero di chiavi da distribuire sia elevato: se si assume che per ogni coppia di interlocutori sia richiesta una chiave distinta, una rete di n utenti utilizza $n(n-1)/2$ chiavi.
La crittografia a chiave pubblica evita questa necessità.
Il protocollo di comunicazione si basa sulla disponibilità di solo due chiavi per ogni utente: una privata conosciuta solo dal proprietario e che viene mantenuta segreta, ed una pubblica che deve essere facilmente reperibile da tutti gli altri utenti.
Chiave pubblica e privata sono funzione di numeri primi grandi; la fattorizzazione di numeri "grandi" è un problema computazionalmente complesso e su questo fonda la sua sicurezza l'algoritmo.

Un messaggio cifrato con la chiave privata può essere decifrato esclusivamente con la chiave pubblica, e viceversa.

- Alice reperisce la chiave pubblica di Bob da un database.
- Alice cifra il messaggio con la chiave pubblica di Bob e lo invia a Bob.
- A questo punto solo Bob può decifrare il messaggio.

I problemi di questo protocollo sono:

- La minor velocità di cifratura e decifratura rispetto a quelli a chiave simmetrica.
- Il protocollo non fornisce **autenticazione del mittente** dei dati ed integrità.
- **L'autenticità della chiave**: come può Alice essere sicura che la chiave pubblica reperita dal database appartenga effettivamente a Bob e non a qualcun altro?

2. Analisi dei Requisiti

Per realizzare la sicurezza delle comunicazioni con un protocollo a "**chiave pubblica**" è indispensabile che ci sia un gestore (*Certification Authority*) che assicuri l'autenticità delle chiavi pubbliche utilizzate. Il progetto vuole realizzare un **servizio di autenticazione e distribuzione di chiavi pubbliche**.
I requisiti del servizio devono essere:

- **SICUREZZA:**
  E' ovviamente il requisito fondamentale, definiamo in breve le *politiche* di sicurezza:
  -Il servizio utilizza canali di comunicazione non sicuri.
  -Le chiavi pubbliche devono essere distribuite a chi le richiede. Eventuali modifiche o sostituzioni devono poter essere riconosciute assicurando l'autenticità della chiave pubblica ricevuta.

  -Supporremo in questa sede che la creazione e la registrazione delle chiavi pubbliche (e la consegna delle corrispondenti chiavi private) sia realizzata esternamente al servizio in modo sicuro.
  -Supporremo anche che i sistemi su cui si esegue il servizio siano sicuri (non consentano accessi non autorizzati).

- **TRASPARENZA:**
  - L'utente non conosce direttamente chi realizza il servizio, né il suo indirizzo fisico, è previsto un *sistema di nomi* per la gestione delle richieste.

- **SCALABILITA':**
  - Il sistema deve essere espandibile, introduciamo quindi il concetto di *località* e *partizionamento* delle risorse.

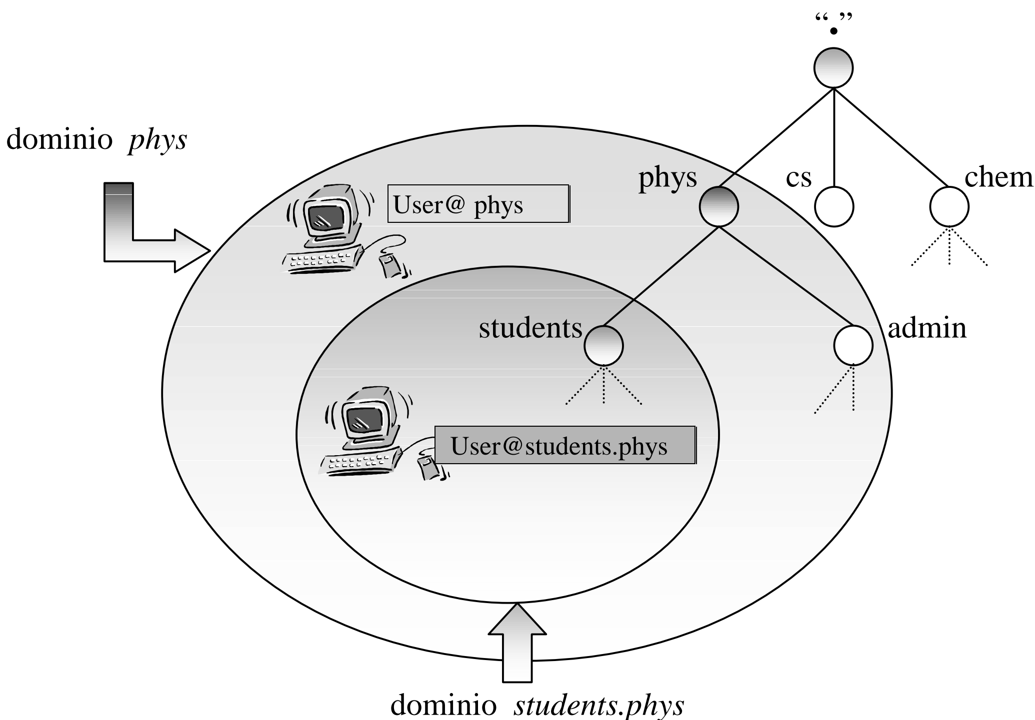
- **FAULT TOLERANCE:**
  -Le informazioni riguardanti le chiavi devono essere sempre disponibili anche a fronte di guasti, è prevista dunque la *replicazione* delle risorse.

3. **Specifica dei Requisiti**

3.1 Gestione dei nomi

Il progetto è stato pensato per lavorare in un'organizzazione limitata, ad esempio un'azienda o un'università.
Poiché in tutte le organizzazioni di questo tipo sono presenti determinati confini di località (ad esempio uffici, distretti, dipartimenti…...) introdurremo un **Sistema di Nomi Gerarchico**, tipo il **DNS** (Domain Name System) utilizzato in Internet. Dividiamo quindi in **domini** limitati lo spazio di nomi degli utenti che utilizzano il servizio. La suddivisione in domini è puramente logica e non avviene secondo principi di effettiva vicinanza fisica (ad es. suddivisione in reti o sottoreti).

Lo spazio dei nomi dei domini appare dunque come una struttura ad **albero:** il dominio "radice" di partenza viene suddiviso in sottodomini di primo livello. La suddivisione può poi avvenire per ciascun sottodominio.



Nell'esempio (una possibile organizzazione universitaria) il dominio radice (indicato con "**.**") ha tre sottodomini di primo livello, mentre il dominio "phys" ne ha due ("students" e "admin"). Da notare che ogni sottodominio fa parte del dominio di livello superiore. L'unica limitazione per quanto riguarda lo spazio dei nomi è che questi non possono essere ripetuti in domini che derivano dallo stesso dominio di livello superiore.

Per quanto riguarda gli utenti essi sono indicati da un nome unico nell'ambito del dominio di appartenenza, seguito dal simbolo "@", e dal cammino assoluto che dal dominio radice porta al dominio di appartenenza.

E’ rispettato il requisito di **trasparenza:** gli utenti sono individuati da nomi logici e non fisici, la corrispondenza tra nomi logici e fisici (indirizzo IP) è realizzata dal DNS

La comunicazione avviene più frequentemente tra utenti dello stesso ufficio o dipartimento (ecc.), e solo in alcuni casi coinvolge utenti in ambiti differenti. Il DNS gerarchico e il partizionamento dei nomi permettono anche una maggiore efficienza nelle comunicazioni locali a scapito di un peggioramento di quelle “globali”, che comunque sono molto meno frequenti. Rispettiamo così il requisito di **scalabilità** del sistema.

3.2 Come avviene la comunicazione

Il principio di località ci permette di effettuare un partizionamento del database delle chiavi pubbliche. Ogni dominio è responsabile del mantenimento delle chiavi pubbliche dei propri utenti, perciò avremo un **Authentication Server** (**AS**) che mantiene l’associazione

<nome logico - chiave pubblica >

per ogni utente registrato.
Un client (all’interno di un dominio) si rivolge al DNS per ottenere l’indirizzo dell’AS a cui far riferimento per le informazioni richieste.
Ottenuta la risposta il cliente ha a disposizione l’indirizzo dell’AS e può iniziare il protocollo di comunicazione.

In pratica:
- A conosce il nome logico di B
- A ottiene dal DNS l’indirizzo fisico di B e l’indirizzo fisico dell’AS locale a B
- A avvia la comunicazione con l’AS locale a B per ottenere le informazioni (chiave pubblica) per comunicare con B.

3.3 Replicazione

La replicazione è realizzata dinamicamente ovvero più AS che svolgono il servizio agendo su una copia della tabella.

Vantaggi di questa realizzazione:
-bilanciamento del carico dei nodi della rete in cui si trovano i server (un’altra soluzione prevedev copie Master Slave che introducono un collo di bottiglia poiché è il Master che prende e gestisce tutte le richieste, con un carico molto elevato per il nodo)
-la replicazione consente tolleranza ai guasti e recovery dei dati in caso di errore o caduta di un nodo.

Naturalmente la prima operazione svolta dall’AS è la registrazione presso il DNS . Il tutto avviene in modo trasparente al client.
In prima istanza il problema è comunque stato limitato implementando solo la distribuzione e non anche la registrazione (inserimento) di chiavi pubbliche.

4. Progettazione del Sistema

4.1 Il protocollo di scambio delle chiavi

Viene utilizzato il protocollo di autenticazione a chiave pubblica di Needahm-Schroeder che prevede le seguenti operazioni:

Header		Message	
1.	A➔S	A,B	A richiede a S la chiave pubblica di B.
2.	S➔A	$\{PK_B,B\}_{SK_S}$	S manda ad A la chiave pubblica di B criptata con la sua chiave segreta. Il messaggio viene cifrato per assicurare che non sia possibile modificarlo. A (e chiunque altro) può decifrare il messaggio usando la chiave pubblica del Server, PKs.
3.	A➔B	$\{N_A,A\}_{PK_B}$	A manda un messaggio contenente un nonce a B cifrato con la chiave pubblica di B. Solo B Solo B può decifrarlo per ottenere il nome di A.
4.	B➔S	B,A	B richiede ad S la chiave pubblica di A.
5.	S➔B	$\{PK_A,A\}_{SK_S}$	S manda ad B la chiave pubblica di A cifrata con la sua chiave segreta.
6.	B➔A	$\{ N_A,N_B\}_{PK_A}$	B manda ad A una coppia di nonce cifrati con la chiave pubblica di A
7.	A➔B	$\{N_B\}_{PK_B}$	A manda a B il nonce che ha appena ricevuto cifrato con la chiave pubblica di B. Questo prova la freschezza della comunicazione, e assicura che è proprio A l'utente che sta comunicando (solo A poteva decifrare il messaggio 6).

PK_A	chiave pubblica di A
PK_B	chiave pubblica di B
PK_S	chiave pubblica di S
SK_A	chiave segreta di S

E' interessante notare come il protocollo assicuri l'autenticità del mittente (il nonce deve essere unico e deve essere generato su domanda). Infatti se il protocollo completa con successo, sia A che B sono sicuri, mediante lo scambio di nonce, che i messaggi che hanno ricevuto vengono proprio dall'altro perché solo lui aveva la possibilità di decifrare il messaggio e rispondere. Naturalmente al termine di questo protocollo avviene la comunicazione vera e propria tra A e B.

Il punto debole del protocollo è l'assenza di protezioni contro la replicazione di vecchi messaggi.
I messaggi che possono dare problemi sono i numeri 2 e 5. Un intruso potrebbe intercettare uno di questi messaggi e ripeterlo successivamente ,ad esempio quando la chiave pubblica in questione non è più valida.
La soluzione più ovvia è quella di inserire nel messaggio un timestamp. Utilizzando il clock fisico delle macchine si crea un problema di sincronizzazione di tutti i server. Ammettendo di effettuare tale sincronizzazione, anche il protocollo con cui essa è realizzata dovrebbe essere sicuro. Il problema diventa quindi difficilmente risolvibile e viene inutilmente complicato.

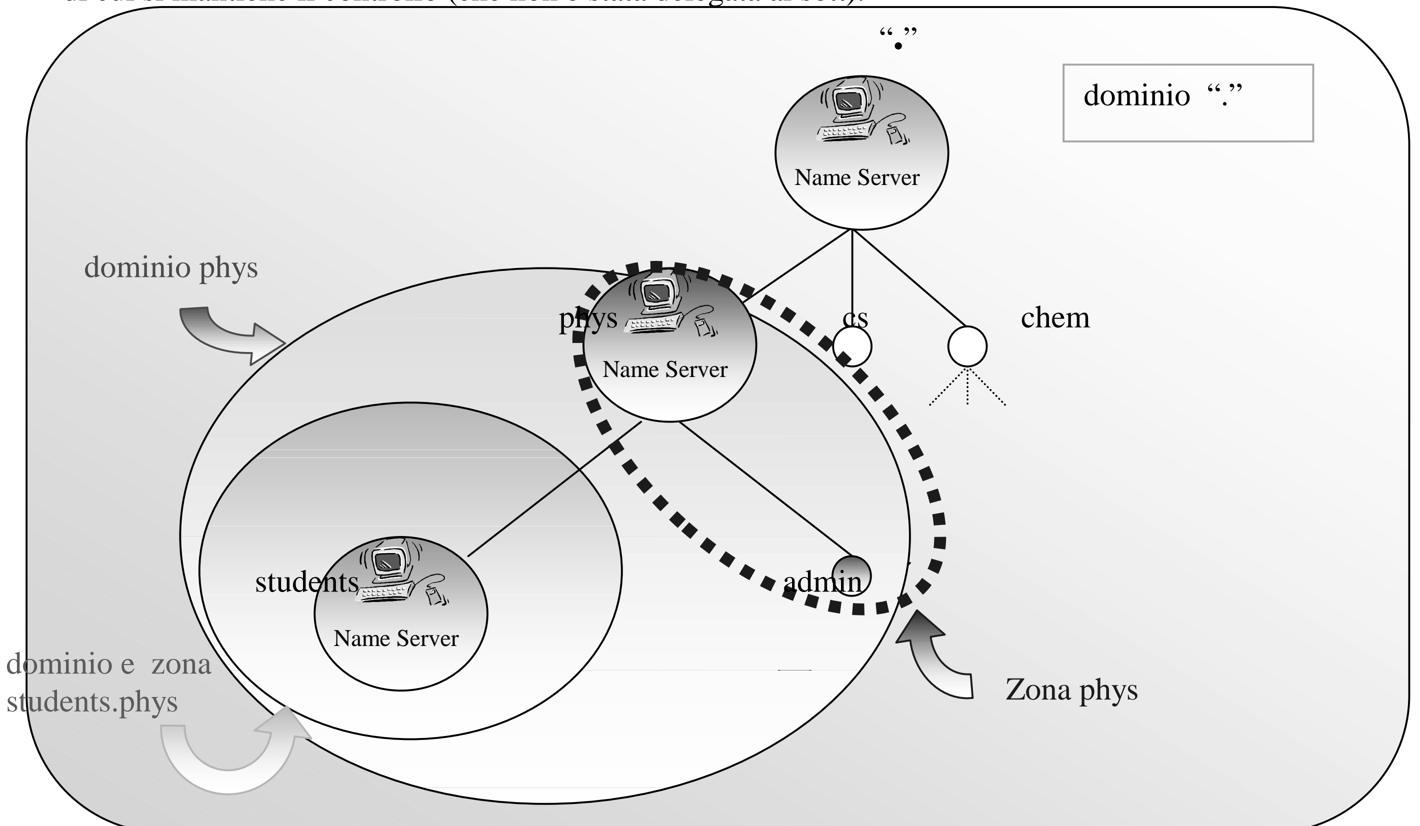
Per risolvere il problema utilizzeremo un clock logico (contatore) unico per ogni client. A invia, oltre la propria identità e quella del destinatario, anche un timestamp incrementale (t_1). L'AS risponde con lo

stesso valore; in questo modo siamo sicuri della freschezza del messaggio, poiché se questo venisse riutilizzato A vedrebbe recapitarsi un messaggio con un numero inferiore a quello che lui ha attualmente in memoria. Il messaggio di richiesta non ha bisogno di essere crittografato poiché è importante solo la corrispondenza del valore nei messaggi tra A e S. Nel caso eventuali intrusioni modificassero tale messaggio A non riconoscerebbe più il Server come mittente del messaggio e inizierebbe nuovamente la procedura di richiesta. Analogamente lavorerà B.

1.	A➔S	A,B,t_1	A richiede a S la chiave pubblica di B
2.	S➔A	$\{PK_B, B, t_1\}_{SK_S}$	S manda ad A la chiave pubblica di B cifrata con la sua chiave segreta. Il messaggio viene cifrato per assicurare che non sia possibile modificarlo. A (e chiunque altro) può decifrare il messaggio usando la chiave pubblica del Server, PKs.
…	…	…	…
4.	B➔S	B,A, t_2	B richiede ad S la chiave pubblica di A.
5.	S➔B	$\{PK_A, A, t_2\}_{SK_S}$	S manda ad B la chiave pubblica di A cifrata con la sua chiave segreta.
…	…	…	…

4.2 Il DNS

Il DNS è suddiviso in domini gerarchici. Organizzeremo il DNS secondo un principio di **delegazione.** Ciascun dominio cioè, può delegare la gestione dello spazio dei nomi di ogni suo sottodominio al sottodominio stesso.
Distinguiamo quindi la definizione di **dominio** da quella di **zona** che indica quella frazione del dominio di cui si mantiene il controllo (che non è stata delegata ai sott).



Il DNS sarà quindi organizzato come un insieme di **Name Server** (NS), ciascuno competente per una determinata zona.
Un NS mantiene quindi i seguenti tipi di associazioni:

<nome logico utente - indirizzo fisico utente>
<nome logico dominio - NS competente>
<indirizzo AS per il dominio>

In ciascun client sarà presente un **resolver** che ad ogni richiesta accederà al NS locale, questo deve interagire con gli altri NS per restituire al resolver la risposta.
Si è molto discusso, infatti, su quali strategie adottare per il coordinamento tra i Server.

Una possibile soluzione è quella **iterativa**: il resolver interroga il NS locale, che a sua volta interroga una serie di NS alla ricerca di una risposta . Ogni NS a cui è stata inoltrata la domanda, se non è in grado di fornire i dati richiesti, risponde con l'indirizzo del NS per lui "migliore" (vedremo in seguito cosa significa). Alla fine il NS in grado di rispondere restituirà la risposta.
Ad esempio se ogni NS conosce l'indirizzo del nodo in cui si trova il NS root, e l'indirizzo degli NS dei sottodomini, si può ottenere una soluzione di questo tipo:

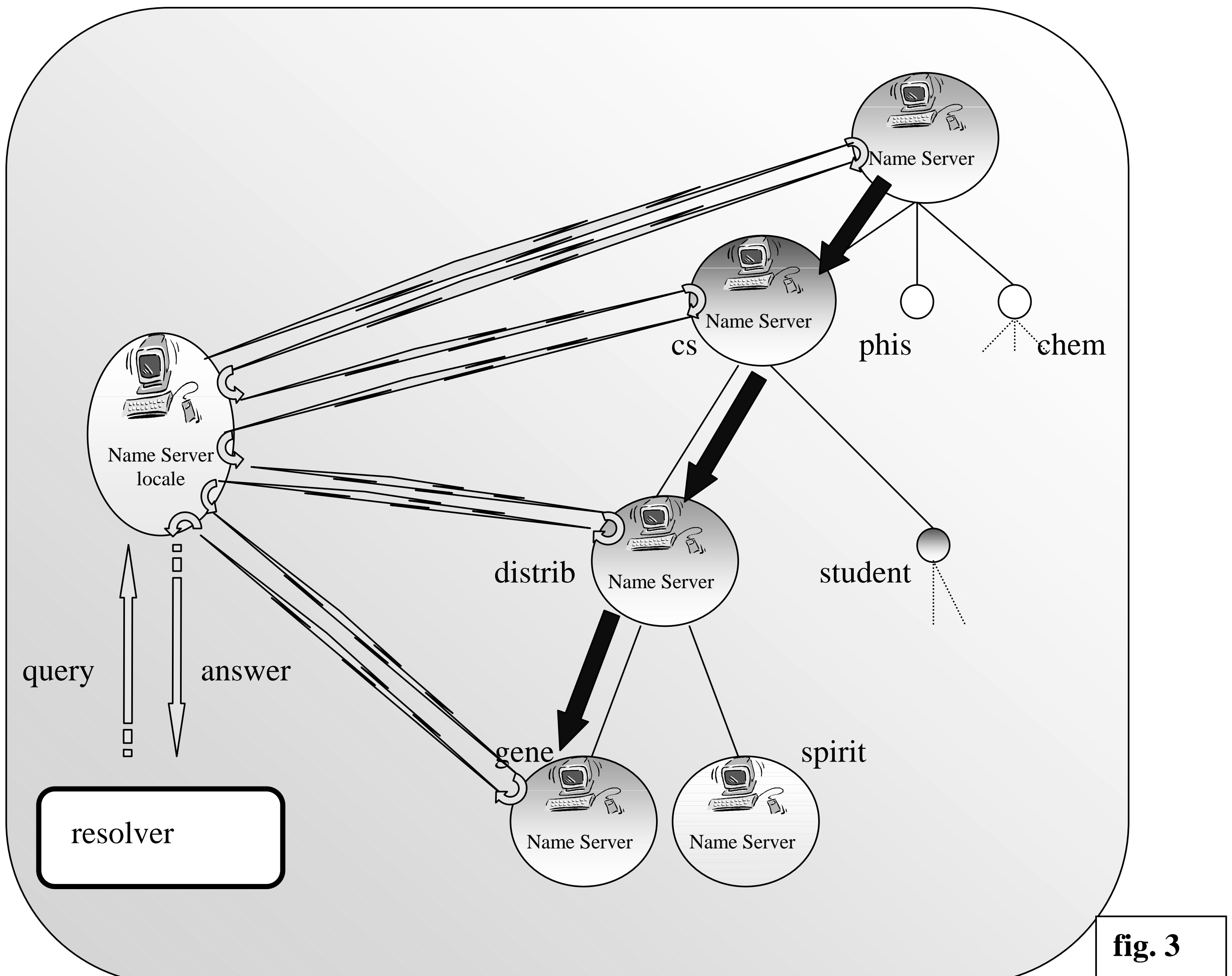


fig. 3

Tale soluzione presenta dei vantaggi per il fatto che ogni NS deve conoscere solo l'indirizzo della root, unico server che deve rimanere "fisso". Questa realizzazione può introdurre un collo di bottiglia perché tutte le richieste sono inviate alla root che potrebbe trovarsi troppo carico di richieste. Comunque l'intero procedimento non viene effettuato tutte le volte poiché ogni NS ha la possibilità di fare del caching.

Un altro approccio iterativo può essere quello di partire dal basso. Cioè ogni NS conosce l'indirizzo non più della root ma del NS del domino superiore. Ovviamente in caso di spostamenti spetta al NS andare a registrarsi sotto un nuovo gestore. Supponiamo di trovarci nel dominio di "spirit".

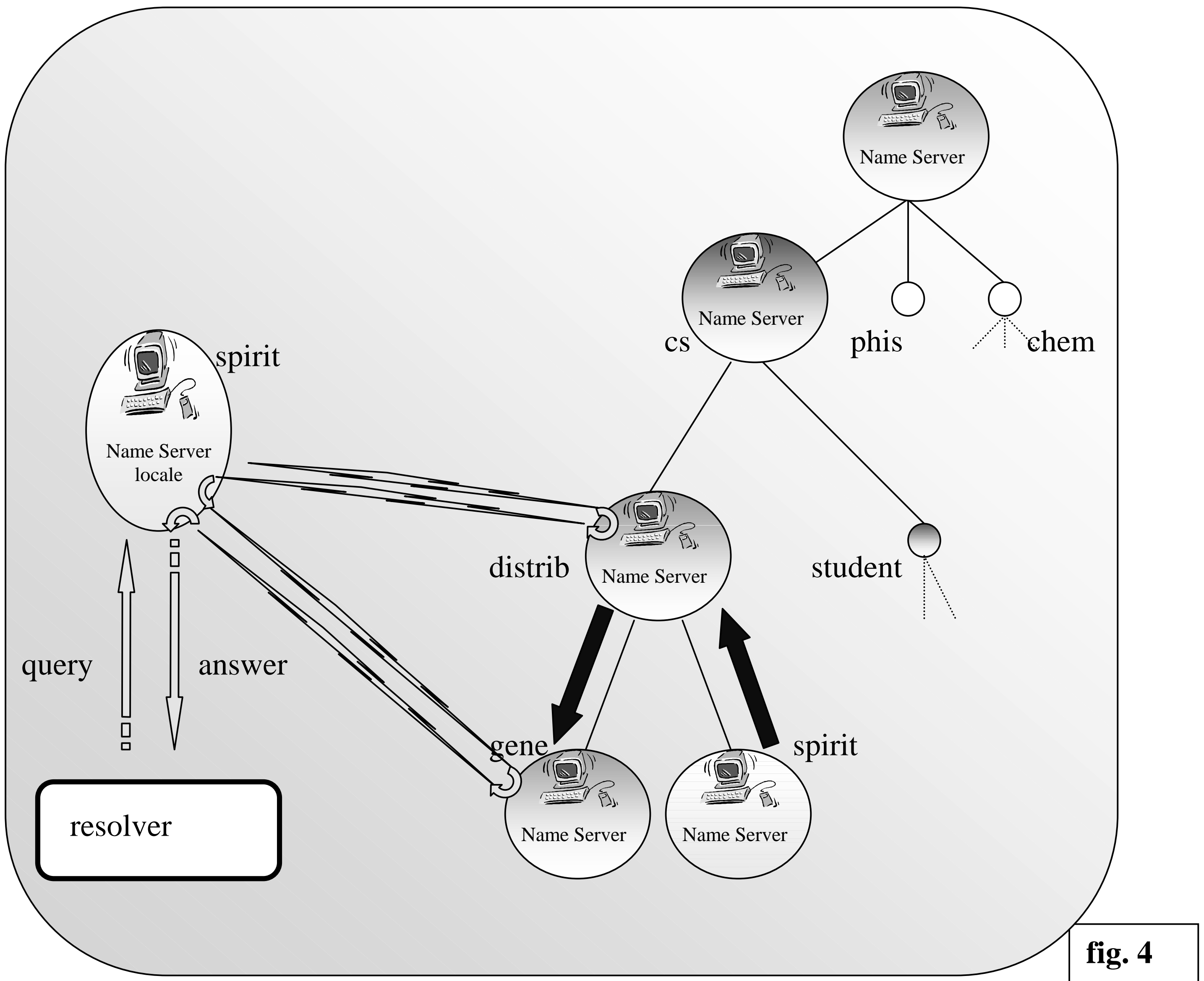


fig. 4

Questo è naturalmente un esempio in cui si avvantaggiano le richieste locali rispetto a quelle non locali (fig. 4). Se, infatti, fossimo stati in un altro ramo ad inoltrare la richiesta, ad esempio "chem" avremmo

dovuto risalire fino alla root per poi ridiscendere esattamente come abbiamo fatto nell'esempio precedente (fig. 3). Anche in questo caso si ipotizza di effettuare del caching che velocizzi la procedura.

Si può pensare anche ad una soluzione **transitiva**. Ovvero il NS interrogato non restituisce al NS richiedente un indirizzo ma inoltra lui stesso la richiesta al server sottostante. Il server locale, che conosce direttamente il destinatario, provvederà a comunicarlo al NS richiedente.

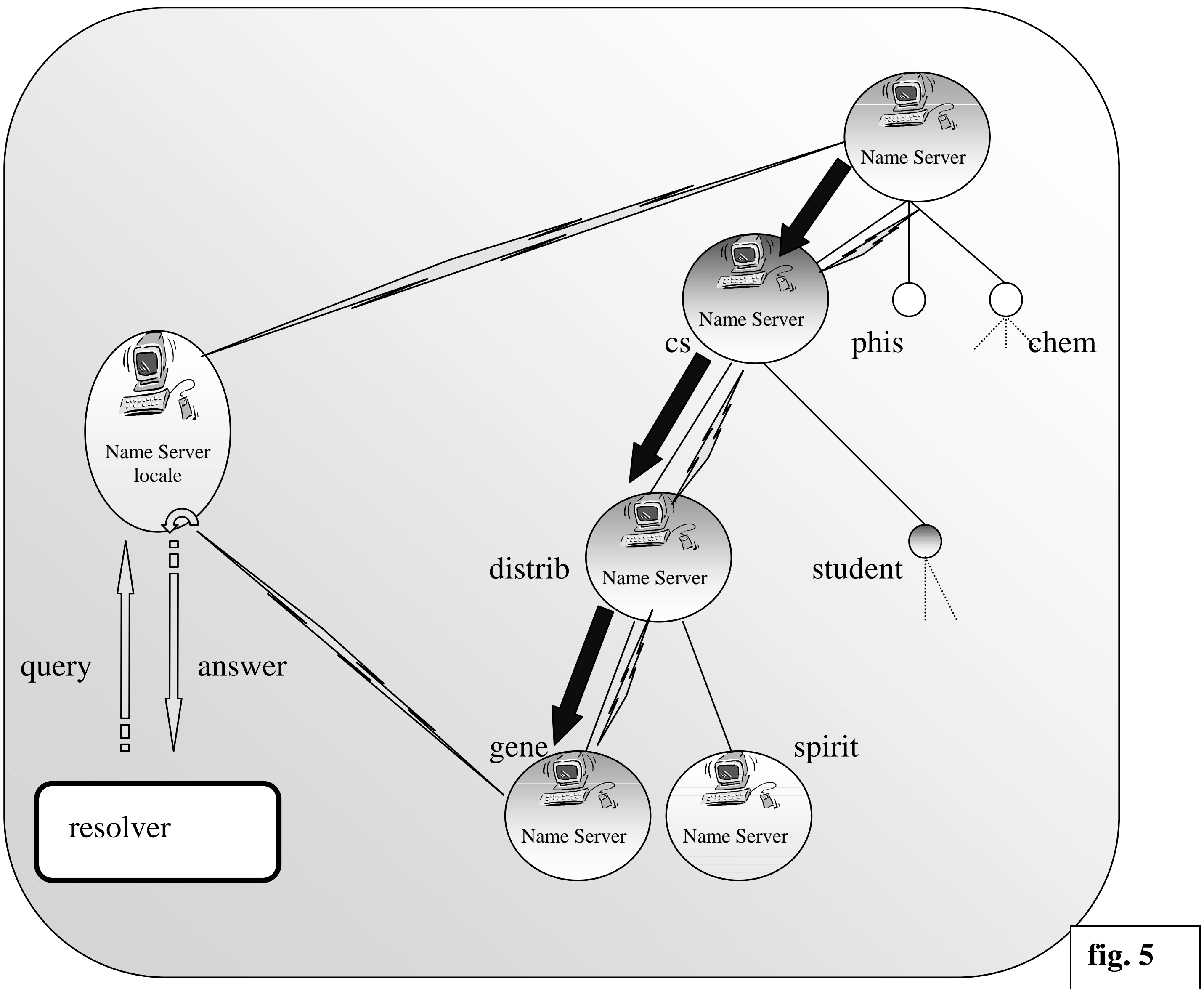
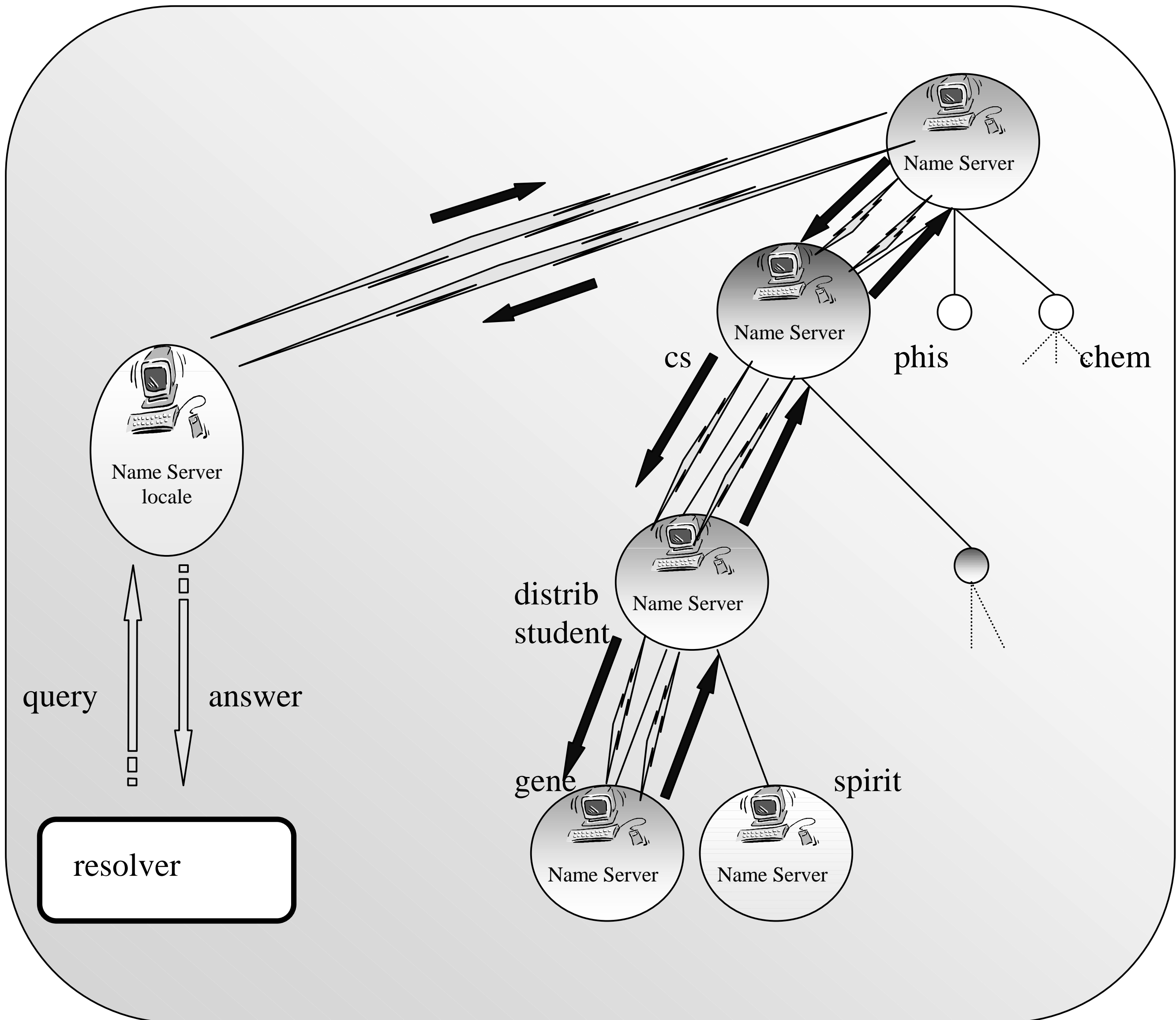


fig. 5

Questa soluzione impegna meno il server locale distribuendo il carico di lavoro tra tutti i Name Server.

Esiste infine la soluzione ricorsiva che in molti aspetti è simile a quella transitiva. Quando la richiesta giunge al NS competente, non è inviata al NS richiedente ma percorre il cammino inverso a quello della query.



Tale soluzione ha il vantaggio di avere dei messaggi di dimensioni ridotte rispetto a quella transitiva a scapito di un maggior numero di messaggi.

Ovviamente non c'è una soluzione migliore in assoluto ma dipende dal tipo di richieste che sono effettuate maggiormente.

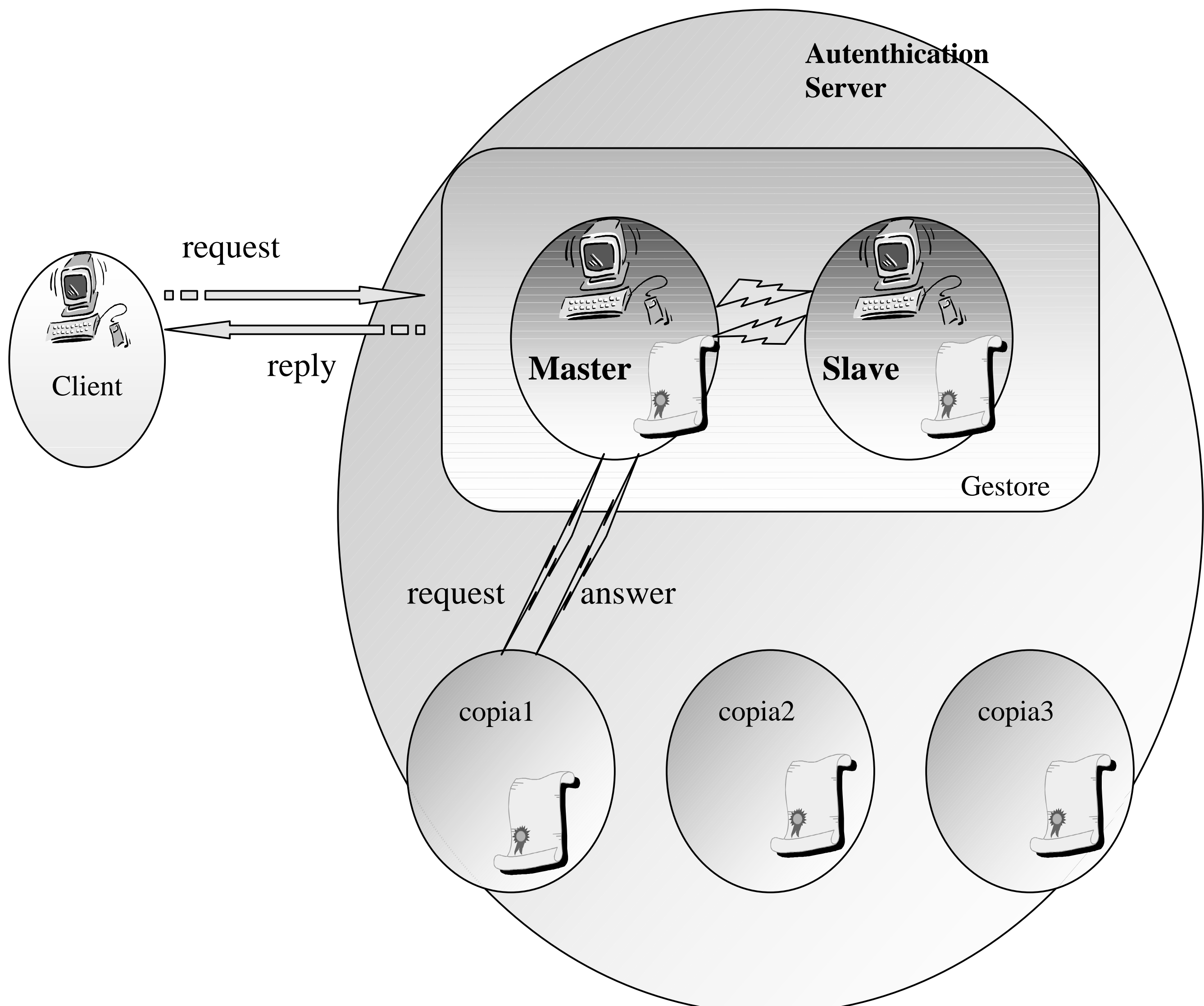
Tra tutte si è scelta la soluzione iterativa poiché anche se a prima vista può sembrare onerosa, in realtà mediante il meccanismo di caching permette un successivo riutilizzo degli indirizzi reperiti nella ricerca. Ricordiamo che nella soluzione transitiva e quella ricorsiva tale meccanismo è meno vantaggioso poiché la ricerca è esterna al NS locale che inoltra la domanda e aspetta la risposta.

Nella gestione del DNS assume un ruolo fondamentale il **caching.** In seguito ad una richiesta iterativa il NS locale memorizza gli indirizzi dei NS con cui comunica, in modo che ad ogni richiesta successiva non si debba ripartire ogni volta dalla root.
Effettuando caching però si può andar incontro a problemi di consistenza. Un NS, infatti, potrebbe mantenere in memoria un indirizzo che non è più valido perché nel frattempo il NS corrispondente è stato spostato. In questi casi il NS non ottenendo risposta dal NS di cui ha l'indirizzo dovrà ripetere il protocollo di richiesta e invalidare la cache.
Ogni NS assocerà alle informazioni che comunica un **Time To Live** (TTL), cioè un tempo di vita delle informazioni.
Il tempo di vita delle informazioni della cache è un compromesso tra **performance** e **consistenza** dei dati.

Ovviamente anche la gestione dei nomi deve essere tollerante ai guasti. E' necessario formulare delle ipotesi di guasto per quanto riguarda i NS. Considereremo solo **guasti singoli,** per quanto riguarda la caduta di un nodo in cui è presente un NS. I NS saranno dunque replicati in due (o più) copie, tra cui distingueremo un **Primary Server** su cui sono mantenuti i dati aggiornati e **Secondary Server** che periodicamente si aggiorneranno collegandosi al Primary Server. Primary e Secondary Server devono essere ovviamente failure-independent e quindi collocati su nodi differenti. La replicazione, oltre che per la tolleranza ai guasti, è utile anche per alleggerire il carico del server primario e distribuirlo su nodi diversi.

4.3 Coordinamento e Replicazione

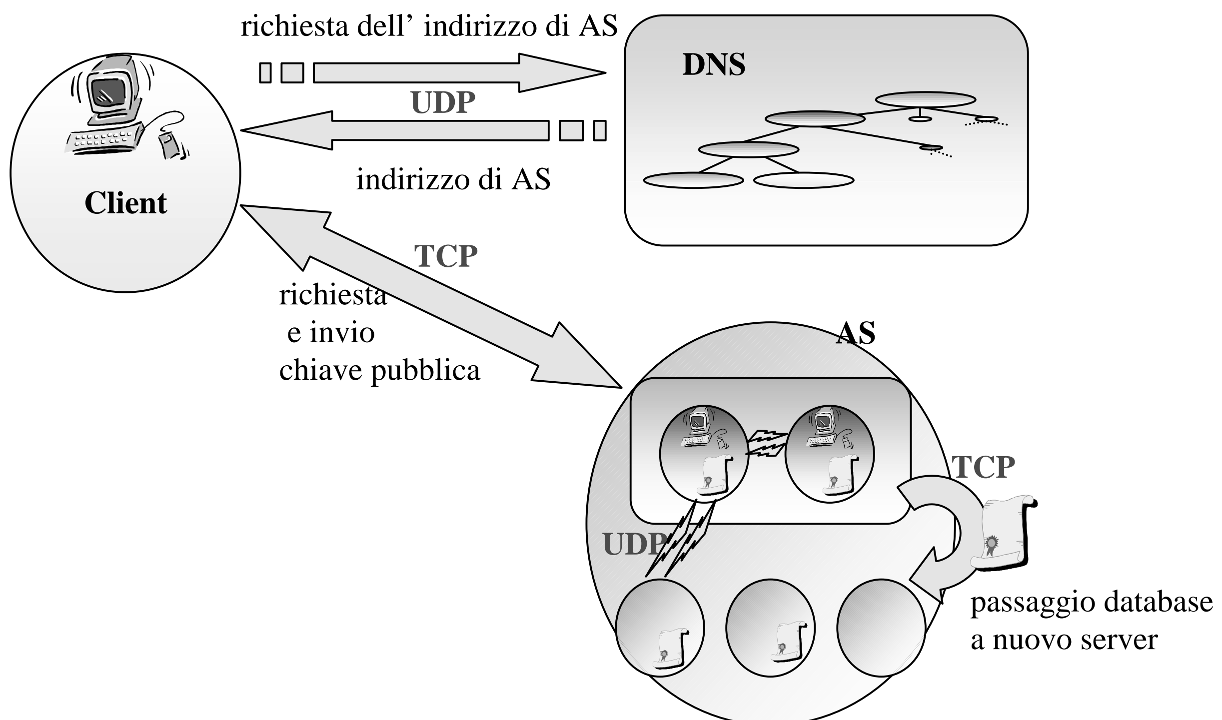
Nella realtà, l'indirizzo reperito dal Client non sarà direttamente quello dell'AS che detiene le informazioni sugli utenti locali, ma quello di un gestore. Tale gestore ha la funzione di bilanciare le richieste fra le varie copie che svolgono il servizio. Il gestore (costituito da una coppia Master Slave per motivi di robustezza ai guasti) inoltra la richiesta ad una copia, attende la risposta e provvede ad inviarla al Client.
Il numero di copie è variabile e dipende anche dalla quantità di richieste che mediamente bisogna soddisfare.
Trattando solo il problema della distribuzione delle chiavi, e non quello della registrazione di nuovi utenti, (compito svolto da un'autorità di certificazione esterna), non è previsto il coordinamento tra le varie copie.



Il Gestore deve distribuire le richieste alle varie copie. Poiché il tempo d'elaborazione delle richieste è uniforme tra tutte le copie (a prescindere dalla velocità della macchina su cui esegue la copia) un semplice approccio è di assegnare sequenzialmente le richieste a tutte le copie. Se una copia non risponde entro un determinato intervallo di tempo, il messaggio viene ripetuto. Se neanche questo messaggio ottiene risposta, la copia viene considerata inattiva e la richiesta viene inoltrata alla successiva copia. Una copia inattiva per tornare a ricevere richieste deve registrarsi nuovamente.
Una copia quando si registra ha bisogno del database delle chiavi. Il gestore mantiene una copia in memoria permanente utilizzata solo per inizializzazione delle copie. Ovviamente l'invio del database alle copie deve essere cifrato con la chiave privata dell'AS

4.4 Comunicazioni

A seconda del tipo di comunicazione si è pensato di utilizzare un diverso protocollo. Le comunicazioni nell'ambito del DNS vengono implementate mediante un protocollo senza connessione (UDP) in quanto trattasi di messaggi di dimensione contenuta. Si preferisce, infatti, in questa fase della comunicazione, puntare più alla velocità che all'affidabilità. Si introducono pertanto controlli aggiuntivi per rimediare alle situazioni di perdita di pacchetti.
Quando invece è la sicurezza il fattore fondamentale si utilizza un protocollo a stream (TCP) affidabile e sicuro. In questo caso si predilige l'affidabilità all'efficienza. Tali comunicazioni coinvolgono i due Client, l'AS e il Client: in particolare, il gestore dell'AS e il Client. Un'altra comunicazione a stream riguarda l'invio del database delle chiavi pubbliche a seguito della registrazione di una nuova copia. Infine i messaggi di richiesta e risposta tra gestore e copie utilizzano datagrammi.



5. Specifica del Software

Il linguaggio di programmazione utilizzato per sviluppare il progetto è **Java** (JDK versione 1.2).
Questa scelta ci permette di utilizzare una organizzazione ad Oggetti nella progettazione del sistema rendendo più modulare e semplice la realizzazione dell'intero sistema.

Per quanto concerne la parte di Crittografia (non disponibile nelle distribuzioni europee di Java), abbiamo scelto l'**Entrust Java Toolkit,** che rispettando le interfacce definite nel package ***java.security*** consente una perfetta integrazione con il linguaggio Java.

Utilizzeremo, almeno in parte, il package **JNDI** 1.2, che definisce la struttura di un generico servizio di nomi.

Il progetto è stato suddiviso, nel suo sviluppo, in due parti principali:

- Il Sevizio di Nomi (**DNS**)
- Il Servizio di Autenticazione e distribuzione delle chiavi (**AS**)

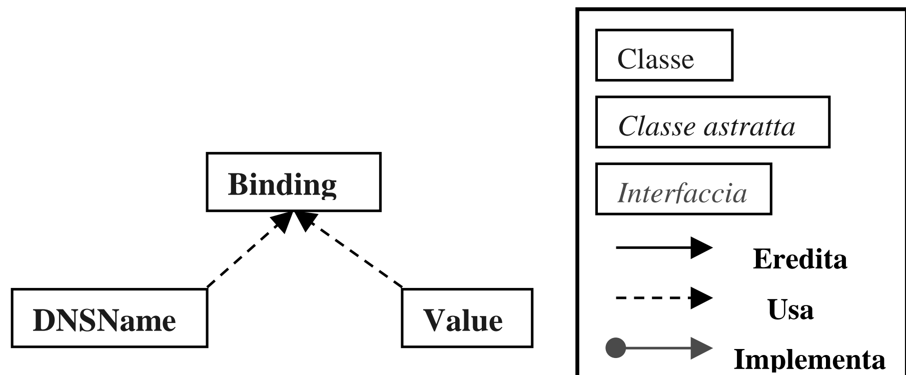
sviluppate parallelamente pur mantenendo un buon grado di coesione.

5.1 Il DNS

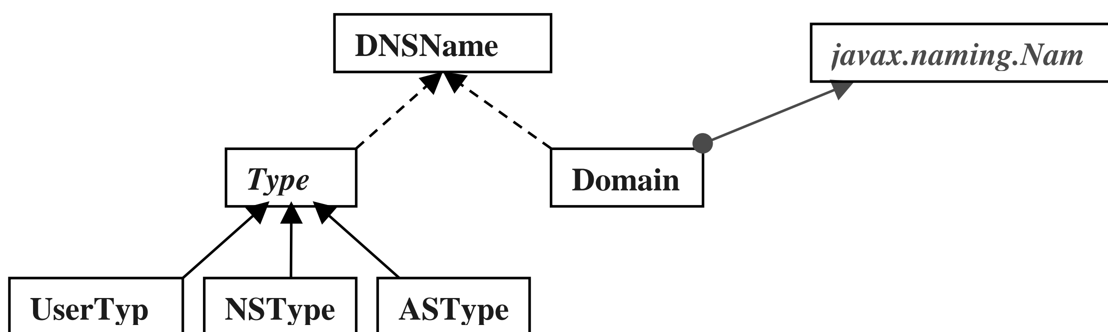
Il principale obiettivo nella realizzazione del DNS sarà quello di non realizzare un prodotto “ad-hoc” per il problema specifico che vogliamo risolvere, bensì un sistema software modulare e facilmente espandibile.

5.1.1 Le classi per la gestione dei nomi

Con “binding” del DNS si intende ogni tipo di associazione che può essere presente in un Sistema di Nomi, in generale sarà una coppia del tipo: < Name , Value >.



La classe **DNSName** modella l’astrazione di nome del DNS, si può considerare costituita da un identificativo (il nome utente,o identificativo di un servizio, nel nostro caso NS o AS) e dal dominio di appartenenza che può essere eventualmente nullo per spazi di nomi "flat".



Type è una classe astratta, le cui sottoclassi rappresentano un "tipo" di nome, nel nostro caso abbiamo un identificativo di utente (**UserType**), di Name Server (**NSType**), e di Authentication Server (**ASType**). Per aggiungere un'altro tipo di servizio da utilizzare tramite il DNS è sufficiente aggiungere una sottoclasse corrispondente.

Domain è la classe che generalizza il nome di dominio, implementa l'interfaccia ***javax.naming.Name*** che definisce le caratteristiche di un generico nome in uno spazio di nomi gerarchico.

Definiamo brevemente la sintassi dei nomi di dominio in notazione BNF:

***Dominio* ::= *root* | *Nome* .**
Dominio
***root* ::= ""**
Nome*::= *alfa* | *alfa Nome
***alfa* ::= a..z | 0..9**

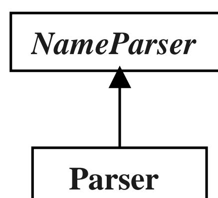
I nomi di dominio sono assoluti, rappresentano dunque una lista, da destra verso sinistra, di nomi semplici, separati da '.' il cui primo elemento è sempre il nome di root rappresentato per convenzione dalla stringa vuota.
Per cui, in un dominio, dovrebbe esserci sempre il punto finale (ad es. "b. a."), dove non presente ("b.a") si intende sottointeso.
L'utilizzo dei nomi è libero tranne che per il nome di root che non può essere replicato (ad es. "b..a." è errato).

In caso di costruzione di un nome errato viene sollevata l'eccezione **javax.naming.InvalidNameException** .

I nomi utente seguono la seguente semplice sintassi:

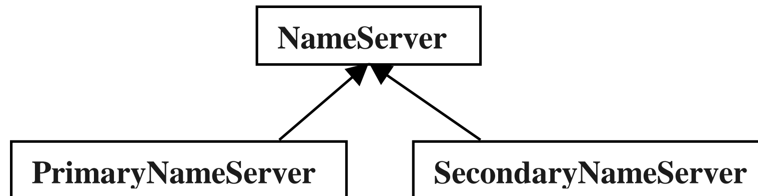
NomeUtente* ::= *UserId* @ *Dominio

La classe **Parser** data una stringa permette di ottenere il DNSName corrispondente, questa eredita dalla classe astratta **NameParser** in modo che sia possibile definire diversi Parser per sintassi dei nomi differenti.



5.1.2 Le classi per la gestione del servizio di nomi

Il DNS è costituito, come già sottolineato, da un insieme di server partizionato e replicato. La classe principale è **NameServer,** che svolge un servizio di Naming senza replicazione.

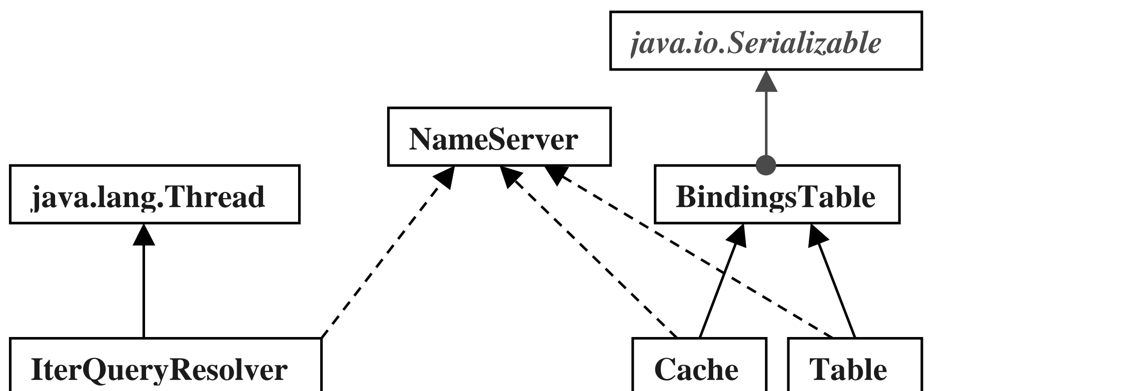


Le sottoclassi **PrimaryNameServer** e **SecondaryNameServer** aggiungono il meccanismo della replicazione.

Funzionalità realizzate dalla classe **NameServer**:

- Richiesta di lookup di un Binding
- Invio di una richiesta di lookup ad un altro NameServer.
- Caching.
- TimeToLive associato ad ogni risposta di lookup

Il NameServer è un caratteristico Server *concorrente* e *parallelo*:



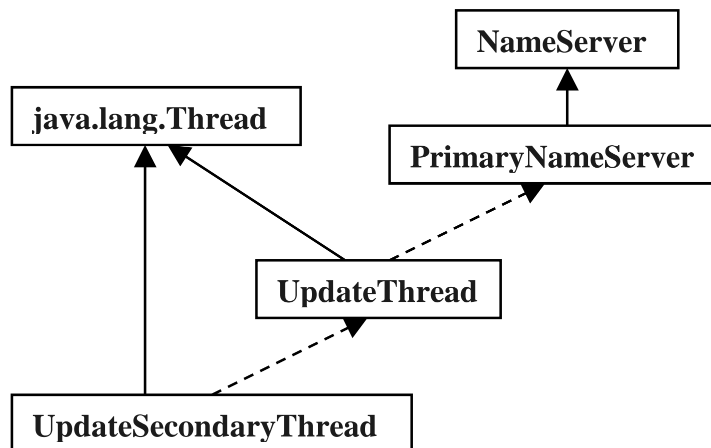
La classe **IterQueryResolver**, che rappresenta un Thread separato, implementa tutta la logica della risoluzione iterativa dei nomi, in questo modo sarebbe possibile realizzare altri tipi di risoluzione (ricorsiva o transitiva), senza modificare la struttura del DNS.

BindingsTable è una tabella che permette di memorizzare e accedere ai Binding. Un **NameServer** ne utilizza due sottoclassi:

> **Table** mantiene tutte le informazioni necessarie ad un NameServer, per cui alla tabella delle associazioni aggiunge due liste, una per i sottodomini delegati e una per quelli non delegati.
> **Cache**, gestisce il caching delle informazioni, ad ogni Binding associa un tempo di vita in secondi (quello del NameServer da cui è stato reperito), allo scadere del tempo di vita, l'associazione perde di validità e può essere eliminata.

Funzionalità realizzate dalla classe **PrimaryNameServer**:

- Booting con lettura da file dei Binding
- Update dei Name Server secondari.
- Binding e unbinding :
  - Creazione di un sottodominio
  - Delega di un sottodominio
  - Inserimento binding
  - Eliminazione binding



Oltre alle funzionalità ereditate da NameServer , la classe **PrimaryNameServer** gestisce l'inserimento delle associazioni e la creazione dei sottodomini, con i relativi controlli di correttezza, e l'aggiornamento dei secondari. A questo scopo utilizza la classe **UpdateThread** ,che crea una porta di ascolto TCP per i secondari, ed utilizza a sua volta la classe **UpdateSecondaryThread** per il trasferimento dei dati.

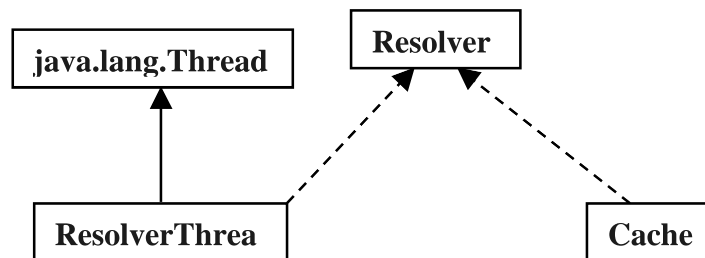
Funzionalità realizzate dalla classe **SecondaryNameServer** :

- Boot con richiesta dei dati al Server primario.

- Richiesta di update al primario.

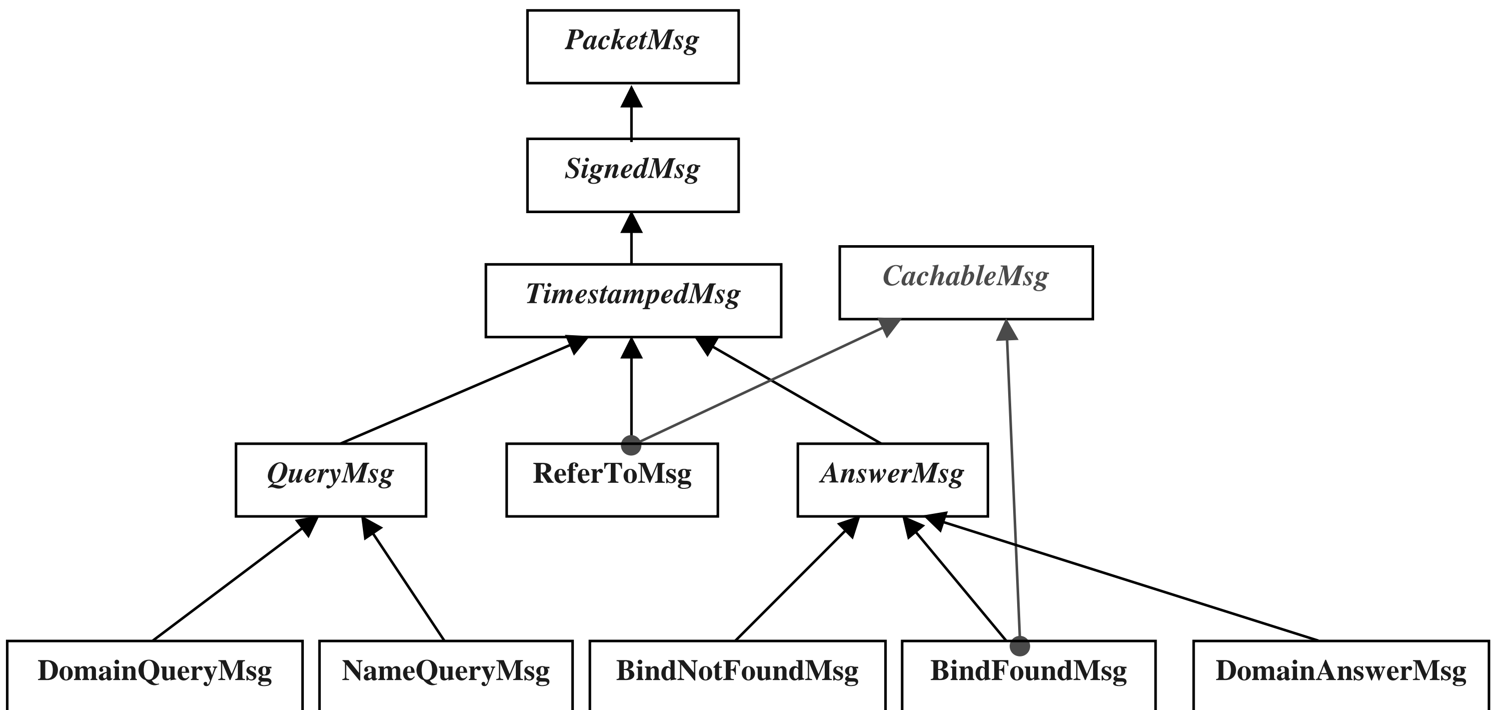
La classe **SecondaryNameServer** richiede periodicamente una tabella aggiornata al Server Primario.

Le interrogazioni da parte dei client vengono indirizzate ad un **Resolver**, che invia le richieste ad un Name Server del dominio. Il Resolver è un Server multithreading che accetta solo richieste locali. In questo modo il caching effettuato dal Resolver è disponibile per tutti i client del nodo.



5.1.3 Le classi per la gestione dei messaggi

Le comunicazioni tra Name Server avvengono tramite scambio di messaggi:



La classe astratta **PacketMsg** rappresenta il generico messaggio, implementa i metodi per trasformare il messaggio in **DatagramPacket** e viceversa.

SignedMsg aggiunge al messaggio un meccanismo di **Signature:**

Il sistema di nomi potrebbe rappresentare un punto debole per la sicurezza (soprattutto nel nostro caso di un servizio basato sulla sicurezza), perciò deve assicurare:

- **l' *autenticità* del mittente.**
- **la *freshness* delle informazioni.**

Poichè non è necessaria la segretezza delle informazioni (che sono di dominio pubblico) è sufficiente un meccanismo di Signature associato al timestamping dei messaggi, che permette di ridurre le dimensioni del pacchetto rispetto alla cifratura del messaggio.

A ciascun pacchetto viene quindi aggiunta la signature delle informazioni contenute, e questa viene controllata alla ricezione. In caso di violazioni della sicurezza o corruzione del pacchetto viene sollevata una eccezione.

TimestampedMsg gestisce il timestamping dei messaggi.

QueryMsg è la classe astratta che rappresenta i messaggi di richiesta di informazioni, mentre **AnswerMsg** i messaggi di risposta.

ReferToMsg è il messaggio che specifica qual è il Name Server cui rivolgersi per ottenere l'informazione richiesta.

Classi che ereditano da **QueryMsg**:
- **NameQueryMsg** : richiesta dei binding di un nome.
- **DomainQueryMsg**: richiesta da parte di un client del dominio di appartenenza.

Classi che ereditano da **AnswerMsg**:
- **BindFoundMsg** : il binding richiesto è stato trovato.
- **BindNotFoundMsg**: il binding richiesto non è stato trovato, poichè il nome non è esistente, o non è stato possibile contattare il Name Server competente.
- **DomainAnswerMsg**: il nome del Dominio richiesto.

Lo scambio di messaggi avviene con un protocollo UDP senza connessione. La scelta è stata fatta poichè i messaggi sono generalmente di dimensioni contenute e quindi non sarebbe giustificato l'overhead introdotto dalla connessione.
La semantica per l'invio dei messaggi è *at-least-once* poichè si assume che nel breve termine tutte le richieste siano idempotenti:
Il richiedente invia il messaggio e si pone in attesa con timeout, allo scadere si effettua il reinvio del messaggio (ad una altro Server se disponibile), fino al numero massimo di ripetizioni consentite, al ricevimento di un messaggio di risposta (qualunque sia la sua provenienza), tutti i successivi vengono ignorati.
Come ipotesi di guasto fissiamo il numero massimo di tentativi di reinvio a 3.

Le classi principali sono:

PacketMsg:
La classe PacketMsg definisce i due seguenti metodi:
- *public DatagramPacket* ***writeToPacket****(InetAddress addr,int port)*
  Questo metodo trasforma il messaggio in un *DatagramPacket* affinchè possa essere inviato tramite una *DatagramSocket* all'indirizzo addr:port.
- *public static PacketMsg* ***readFromPacket****(DatagramPacket p)*
  Il metodo dato un pacchetto restituisce il messaggio corrispondente.

Per la trasformazione dei messaggi in pacchetti e viceversa si utilizza il concetto di *Serializzazione* del linguaggio Java. In questo modo è molto più agevole estendere il sistema con nuovi messaggi poiché non ci si deve preoccupare della codifica degli stessi in pacchetti e viceversa.
Il problema della Serializzazione è che l'interprete Java inserisce molte informazioni non necessarie, per poter codificare gli oggetti negli stream. Per questo tutti gli oggetti utilizzati nei messaggi (come DnsName, Domain ecc.) dovranno ridefinire la serializzazione, in modo da ridurre la dimensione dei messaggi, tramite l'overriding dei metodi:

- *private void **writeObject**(ObjectOutputStream out)*
- *private void **readObject** (ObjectInputStream in)*
-

In questo è possibile non serializzare le strutture che contengono i dati (come ArrayList o HashTable) per ricostruirle soltanto in seguito, oppure inviare delle rappresentazioni compatte dei dati.

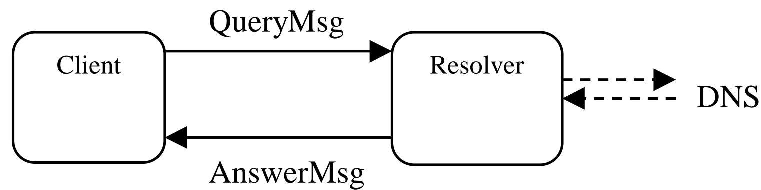
Per evitare di eccedere nella dimensione dei pacchetti, la classe PacketMsg stabilisce una dimensione massima di 1024 byte, che viene controllata alla creazione del **DatagramPacket**, in caso di dimensioni eccessive viene sollevata un'eccezione **MsgTooLongException**.

SignedMsg:
La classe Signed message esegue l'overriding dei rispettivi metodi della classe PacketMsg:

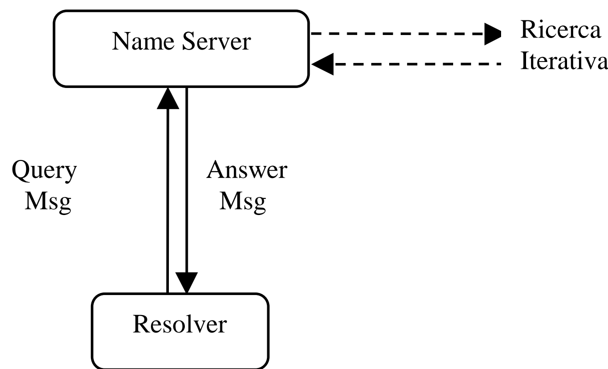
- *public DatagramPacket **writeToPacket**(InetAddress addr,int port)*
  Oltre alla serializzazione dell'oggetto aggiunge una signature dell'array di byte che lo rappresenta, mediante l'oggetto **iaik.java.security.Signature**
- *public static PacketMsg **readFromPacket**(DatagramPacket p)*
  Trasforma il pacchetto in messaggio e ne verifica l'integrità tramite la signature allegata.

5.1.4 La gestione del Servizio

Si assume che la porta utilizzata dal servizio sia nota e comuni a tutti i Server (Resolver, e NS primari e secondari).
Un Client alla necessità di risolvere un nome invia un **QueryMsg** (DomainQueryMsg per conoscere il proprio dominio, NameQueryMsg per ottenere l'indirizzo associato ad un nome), al Resolver locale (reperibile all'indirizzo locale alla porta 5656), questi conosce gli indirizzi dei NS del dominio, si occupa di risolvere la query e restituisce il risultato al Client come **AnswerMsg** (a seconda del risultato della query).



Il Resolver innanzitutto verifica che il nome richiesto nome richiesto non sia presente in Cache, nel caso in cui risponde immediatamente al Client. Altrimenti invia il QueryMsg ricevuto ad un Name Server del dominio, che si occuperà della ricerca iterativa e attende da questo la risposta.

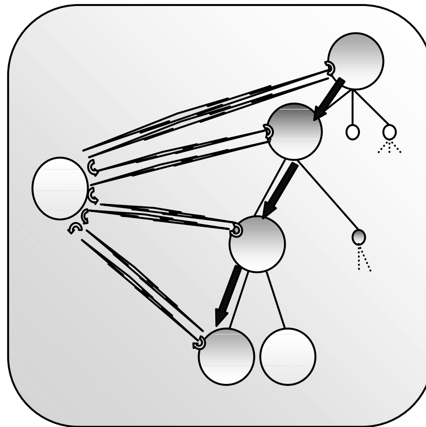


Il NameServer all'attivazione entra in un ciclo infinito in cui si mette in ascolto su di una **DatagramSocket** in attesa di QueryMsg. All'arrivo di un messaggio ne affida la gestione ad un IterQueryResolver.

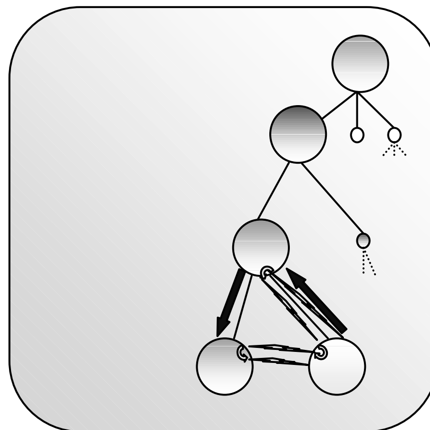
La classe IterQueryResolver svolge quindi due compiti:

- Se la richiesta proviene da un Resolver si fa carico della ricerca iterativa .
- Altrimenti se la richiesta proviene da un altro NameServer (che sta eseguendo una ricerca iterativa) si ha il controllo della richiesta e la risposta.

In realtà sono disponibili due tipi di ricerca iterativa, una normale con la prima richiesta verso la Root:



, ed una con la prima richiesta verso il dominio di livello superiore:



La scelta è fatta utilizzando una funzione euristica:

- *protected boolean* ***iterUpSearch****(Domain query)*

che determina la "distanza" dal dominio richiesto, intesa come numero di Name Server che devono essere coinvolti nel caso peggiore, cioè senza l'intervento della cache.
Per fare ciò si verifica qual è il dominio superiore comune a quello richiesto e a quello richiedente (nel caso peggiore il dominio di root), e si confronta la distanza tra questo e, rispettivamente, il dominio richiedente e quello di root.

Flusso di esecuzione di **IterQueryResolver**:

Per prima cosa IterQueryResolver controlla se il nome richiesto è locale o appartiene ad un sottodominio non delegato, in questo caso il Name Server è competente e si risponde immediatamente al richiedente.

Se ciò non avviene si controlla la cache. Se il nome è presente si risponde, altrimenti si ricerca una "scorciatoia", in altre parole si ricerca un Name Server per un dominio superiore gerarchicamente a quello richiesto, la ricerca ha termine con successo quando il NS è trovato, e comunque quando si raggiunge un dominio genitore del dominio che effettua la ricerca (compreso il dominio stesso e la root) poiché potrebbe non essere più vantaggioso.

A questo punto si decide se effettuare:

A. **Ricerca Iterativa:**
   Primo passo:
   Se in cache è stata trovata un Refer si utilizza, oppure se il nome richiesto appartiene ad un sottodominio delegato si invia la richiesta al NS corrispondente. Altrimenti si decide se effettuare la ricerca iterativa verso l'alto o se inviare la richiesta alla Root.

   Passi Successivi:
   Alla ricezione di un ReferTo si invia la query all'indirizzo corrispondente finchè non si riceve un AnswerMsg, che viene inviato come risposta al richiedente.

B. **Controllo e risposta:**
   Si controlla che il dominio sia un sottodominio, in questo caso sia invia un ReferTo del sottodominio delegato competente. Altrimenti si tratta di una ricerca iterativa verso l'alto per cui si invia il ReferTo del dominio superiore.

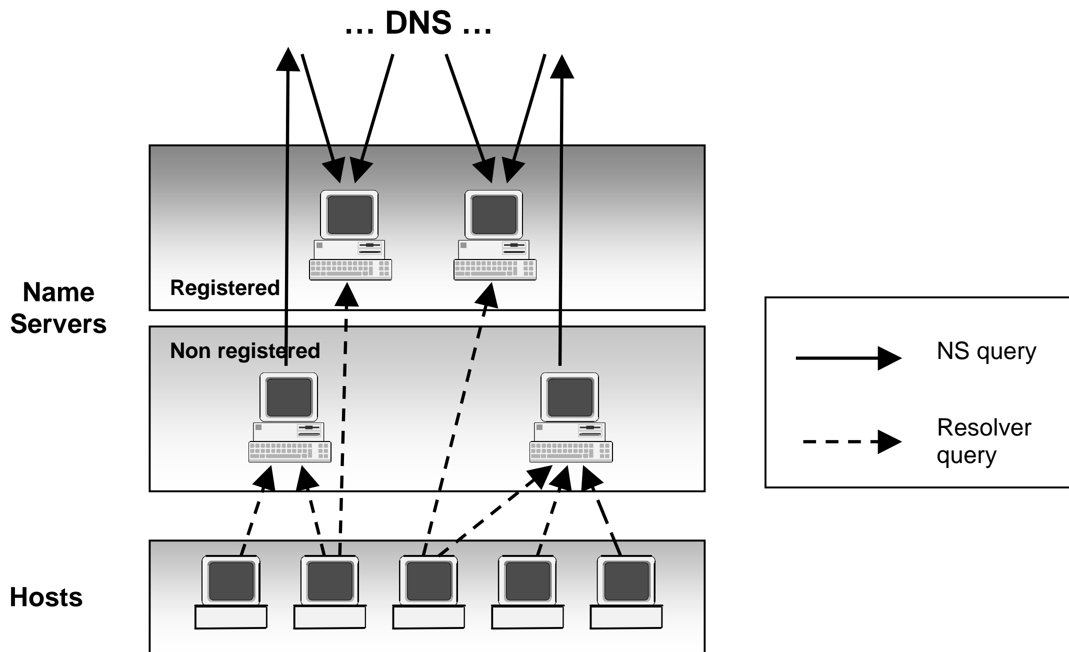
In caso di nomi errati o inesistenti bisogna prevenire l'innescarsi di cicli infiniti di ricerca. E' sempre possibile determinare quando un dominio è competente per un dato nome, e quindi se un nome è corretto o meno. Inoltre poniamo un limite massimo al numero di cicli iterativi che possono essere eseguiti.

Questo è il meccanismo di funzionamento dei Name Server indipendentemente dal fatto che siano primari o secondari. L'unica differenza è il metodo di reperimento della tabella. Il Primario mantiene la propria aggiornando periodicamente un file che è letto allo startup. Inoltre avvia un Thread che apre un a porta TCP per le richieste di aggiornamento da parte dei secondari.

All'avvio il Secondario attiva la connessione e riceve dal Primario la tabella.
Periodicamente il Secondario richiede l'aggiornamento della tabella al Primario, il tempo che intercorre tra un aggiornamento e l'altro è un parametro del Server, e può variare la consistenza dei dati. L'aggiornamento non avviene indiscriminatamente, ma solo se ci sono state variazioni della tabella dall'ultimo aggiornamento.

Poiché la classe PacketMsg ha una dimensione massima, viene posto un limite di 10 nella registrazione di valori da associare ad un determinato nome.

Non è quindi possibile registrare più di 10 Name Server per un determinato dominio (1 Primario e 9 Secondari).
Si noti che è comunque possibile inserire altri Name Server "non registrati" con il solo compito di effettuare ricerche e caching.



5.1.5 Tolleranza ai Guasti

Bisogna sottolineare che, nella progettazione, si è tenuto conto che, come avviene in generale, è ammesso che il DNS in alcuni casi possa non rispondere alle query, o rispondere con dati non consistenti.

In caso di caduta di un Server Secondario, al riavvio, chiederà la tabella al Primario, la sua caduta quindi non causa alcun problema, in quanto ai Resolver viene assegnato più di un Name Server di riferimento.

In caso di caduta di un Server Primario, il sistema continua a funzionare basandosi sui Secondari, in questo caso però potrebbero esserci inconsistenze per binding non aggiornati sino al ripristino del Primario, questo è ammissibile poiché si suppone che i cambiamenti nel DNS siano poco frequenti.

Il sistema può continuare a funzionare temporaneamente anche in caso di partizionamento della rete, basandosi sul caching effettuato in precedenza, ovviamente le prestazioni andranno degradando. Lo stesso discorso è valido per il temporaneo isolamento di un nodo client.

5.2 Authentication Server

I punti chiave dell’ AS sono:
- i Gestori
- le Copie

5.2.1 IL GESTORE

Il Gestore mantiene una porta di ascolto dove attende le richieste da parte di:
- Client, per ottenere la chiave pubblica di altri Client in modo da poter effettuare la comunicazione;
- Copie, per ottenere la tabella delle chiavi pubbliche in modo da poter iniziare a fornire il servizio. Ottenuta la tabella, la Copia invierà un messaggio di “conoscenza” (il suo indirizzo) a tutti i gestori del dominio;
- Server (gestori) Secondari, per ottenere l’aggiornamento sulle Copie attualmente attive;

Inoltre gestisce la lista delle Copie attualmente in funzione, mantiene la tabella delle chiavi pubbliche (sempre consistente) relativa al proprio dominio, fornisce un timestamp unico per ogni operazione in modo che tutti i messaggi inviati siano riconoscibili come inviati dal mittente Gestore e attuali (cioè non riutilizzati).
Per gestire la comunicazione, a seguito di ogni richiesta, genera un thread il quale svolge tutto il lavoro.
La replicazione è realizzata mediante una serie di Gestori Secondari il cui primo passo è richiedere lo stato attuale delle Copie “attive” al Gestore primario; e quindi iniziare a fornire il servizio al pari del Gestore Primario.
Le nuove Copie che si rendono disponibili a fornire il servizio si registrano su tutti i Gestori del proprio dominio, in questo modo si sono evitati i messaggi di coordinazione tra il Gestore master e gli slave poiché tutti possiedono le stesse informazioni (ricordiamo che abbiamo trattato solo il problema della distribuzione delle chiavi, e non quello della registrazione di nuovi utenti, compito svolto da un autorità di certificazione esterna, e di conseguenza per quel che ci riguarda lavoriamo su una tabella di chiavi pubbliche “statica”).
Un ultima considerazione riguarda il fatto che il Gestore contattato dal Client non è quello relativo al proprio dominio ma è quello del Client su cui viene effettuata la richiesta; è questo infatti che possiede le informazioni di cui abbiamo bisogno.

5.2.1.2 Aggiornamento di una Copia

Ottenuta dal Dns la lista dei Gestori attivi, la Copia inoltra la domanda al primo Gestore della lista (ovvero al Master, sempre in cima alla lista), e resta in attesa dell’invio della tabella mediante la Socket TCP aperta (dal Gestore) in conseguenza alla sua richiesta.
Il primo passo svolto dal Gestore è la lettura dell'Hashtable (la lettura viene effettuata da file per semplicità nella realtà la tabella dovrebbe essere sempre a disposizione del Gestore memorizzata in un area protetta della memoria). Nel caso non sia possibile accedere alla tabella viene chiusa la socket in modo che la Copia non rimanga indefinitivamente in attesa. In realtà viene terminato anche il Gestore in quanto non ha senso di esistere se non è in possesso di una tabella (consistente) da fornire alle Copie, inoltre l' assenza della tabella potrebbe significare problemi o manomissioni. Noi trattando la distribuzione delle chiavi partiamo sempre dall’ipotesi che il Gestore abbia, in ogni momento, a sua disposizione una tabella consistente.

Si è scelto di non inviare la tabella come singolo oggetto ma di inviare singolarmente le varie entry della tabella stessa (viene inviata su socket la dimensione della tabella da leggere quindi ogni singola entry viene firmata, e spedita). In questo modo si è ridotto il numero di byte inviati limitando il problema derivante dall'invio di tabelle di una certa dimensione (si sono così evitati piccoli problemi interni al pacchetto della sicurezza java in cui ci siamo imbattuti legati alla signature di tali pacchetti, inoltre viene così ridotto, seppure di poco, il tempo per la codifica/decodifica del messaggio).
Notare che ogni entry viene numerata a partire da un timestamp unico cioè i timestamp delle singole entry seguono un ordine logico di numerazione. Se anche il timestamp di una sola entry non coincide dalla parte della Copia, il processo di aggiornamento fallisce. Terminato l'invio delle entry della tabella delle chiavi pubbliche è il Gestore che si mette in attesa che la Copia gli invii l'ultimo timestamp ricevuto, incrementato come riprova del fatto che solo la Copia poteva essere a conoscenza della chiave per decodificare quel messaggio, che la tabella è stata inviata correttamente in tutte le sue entry, e che i messaggi sono effettivamente attuali.
L'ultimo messaggio serve infatti da conferma per verificare che le entry ricevute dalla Copia non siano TUTTI messaggi vecchi intercettati e riutilizzati.

Se anche dalla parte del Gestore tutto coincide: il messaggio è corretto, non ha cioè subito modifiche durante il suo cammino e il timestamp ricevuto coincide con l'ultimo timestamp inviato alla Copia incrementato di uno, la Copia viene considerata attiva (funzionante) altrimenti attende il riinoltro della richiesta da parte della Copia.

5.2.1.3 Comunicazione con Client e interfacciamento con la Copia

Il Client, ottenuti gli indirizzi dei Gestori attivi dal DNS, effettua una richiesta per ottenere la chiave pubblica di un altro Client ad un Gestore, il quale la invia ad una Copia. Le richieste vengono distribuite sequenzialmente in modo da mantenere un carico di lavoro abbastanza uniforme tra le varie Copie (considerando un medesimo tempo di risposta).
Naturalmente la richiesta effettuata dal Gestore alla Copia puo avere esito positivo o negativo. Nel primo caso siamo in possesso della chiave pubblica richiesta dal Client e dobbiamo effettuare la risposta in modo che il Client sia sicuro della consistenza della chiave ricevuta. In caso negativo viene inviato al Cliente un messaggo indicante l'impossibilità di reperire la chiave o risposte indicanti il rifiuto o l'assenza del Cliente specificato nel database.
I messaggi di rifiuto li abbiampo solo nel caso in cui i messaggi vengano modificati durante il loro cammino, in questo caso il Gestore, o la Copia ricevendoli si rende conto della non consistenza di suddetti messaggi e li invalida.
Fondamentale in tutti i messaggi è il controllo della coincidenza dei timestamp per evitare il replying dei msg (anche in quest' ultimo caso viene invalidata la risposta ricevuta e subito inviata un ulteriore richiesta).
Ma ci sono casi in cui non riceviamo risposta dalla Copia:
- la Copia per un qualsiasi motivo non risponde (potrebbe essere caduta, o qualcuno potrebbe intercettare i messaggi impedendone l'arrivo alla Copia) in questo caso scatta il TIMEOUT;
- la Copia non risponde semplicemente per l'assenza di Copie attive per fornire il servizio.

Nel caso una Copia non risponda (msg di timeout per 2 volte consecutive) viene cancellato il suo indirizzo da quelli disponibili e viene inoltrata la domanda a un altra Copia;

Ricevuto correttamente il messaggio dalla Copia recante la chiave del Client richiesta, il Gestore gli manda un messaggio con il SUO (del Client) timestamp e con una signature per l' autenticazione.

5.2.2 LA COPIA

I passi svolti da una Copia sono:
- legarsi alla prima porta di lavoro libera;
- richiedere al DNS gli indirizzi dei Gestori attivi (UDP);
- richiedere al Gestore Master la tabella su cui lavorare (da qui in poi i passi saranno duali a quelli svolti dal Gestore per l' aggiornamento); la richiesta è, in pratica, anche una richiesta di registrazione presso il Gestore stesso;
- avvisare (o almeno tentare di avvisare) tutti i Gestori Secondari del fatto che ora è abilitata a svolgere il servizio (nel caso un Server sia caduto, vista l'impossibilità di avvertirlo, continua ad aggiornare i restanti server);
- mettersi in ascolto sulla porta per rispondere alle query.

La scelta di non generare un thread per gestire le richieste delle Copie è dettata dal fatto che:
- per quanto concerne la richiesta di aggiornamento è inutile creare un thread che gestisca la ricezione della tabella delle chiavi poiché la Copia per poter rispondere alle domande del Gestore ha bisogno della tabella stessa;
- la singola richiesta di reperimento e invio di chiave pubblica è talmente semplice e veloce che si è ritenuto inutile appesantire il codice per creare un thread il cui unico compito è leggere un dato e rispedirlo. Inoltre il Gestore distribuendo le richieste sequenzialmente mantiene un carico di lavoro abbastanza uniforme tra le varie Copie.

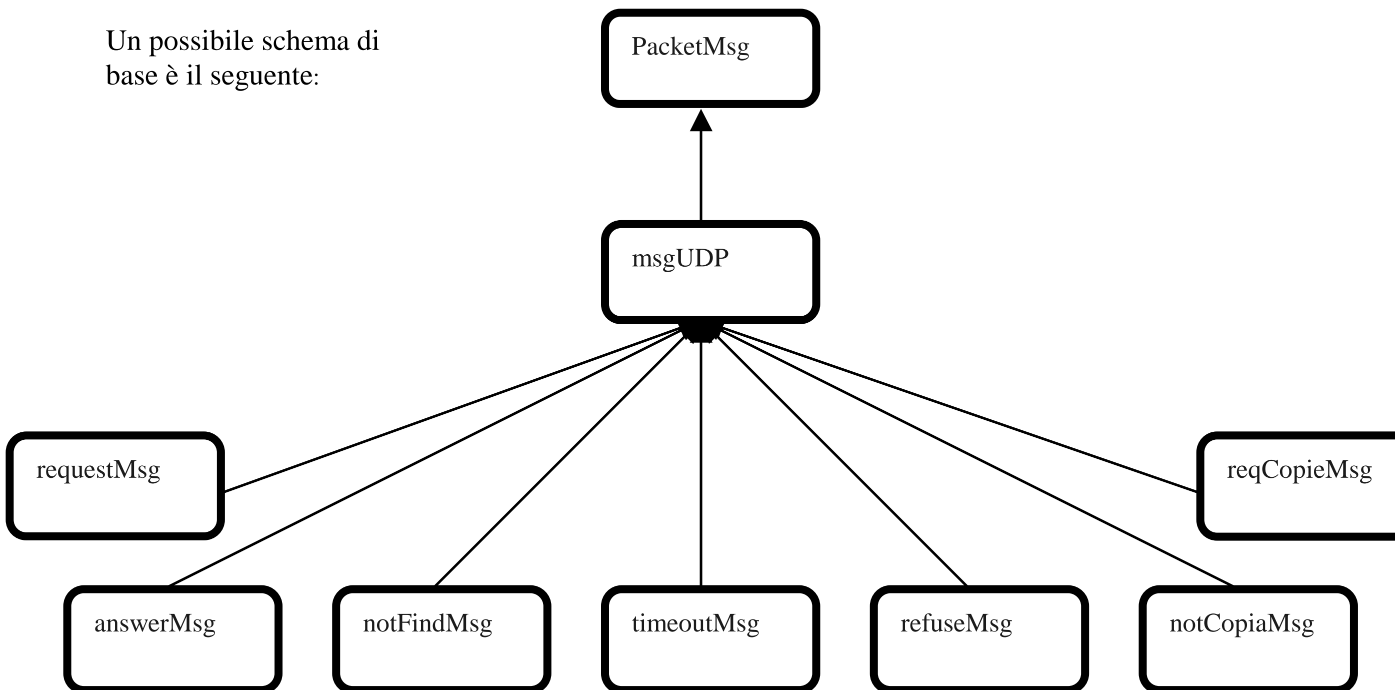
La tabella hash è passata con il protocollo TCP. Nel caso si verifichino problemi, come ad esempio intromissioni di terzi, viene reinoltrata la richiesta di aggiornamento. La comunicazione tra Copia e Gestore per il servizio di distribuzione delle chiavi pubbliche utilizza invece UDP; si sono considerati quindi tutti i possibili casi di riemissione dei messaggi in caso di perdita o modifica degli stessi.

5.2.3 PROGETTAZIONE E DEFINIZIONE DELLE CLASSI

5.2.3.1 I Messaggi

Per messaggi si intendono tutti i messaggi utilizzati nell'ambito dell'Authentication Server e quindi quelli scambiati tra Client, Gestore e Copie. E' stata creata una classe astratta PacketMsg che fornisce tutte le funzioni per creare pacchetti (tale funzione la ritroviamo anche come base per i messaggi del DNS).

Un possibile schema di base è il seguente:

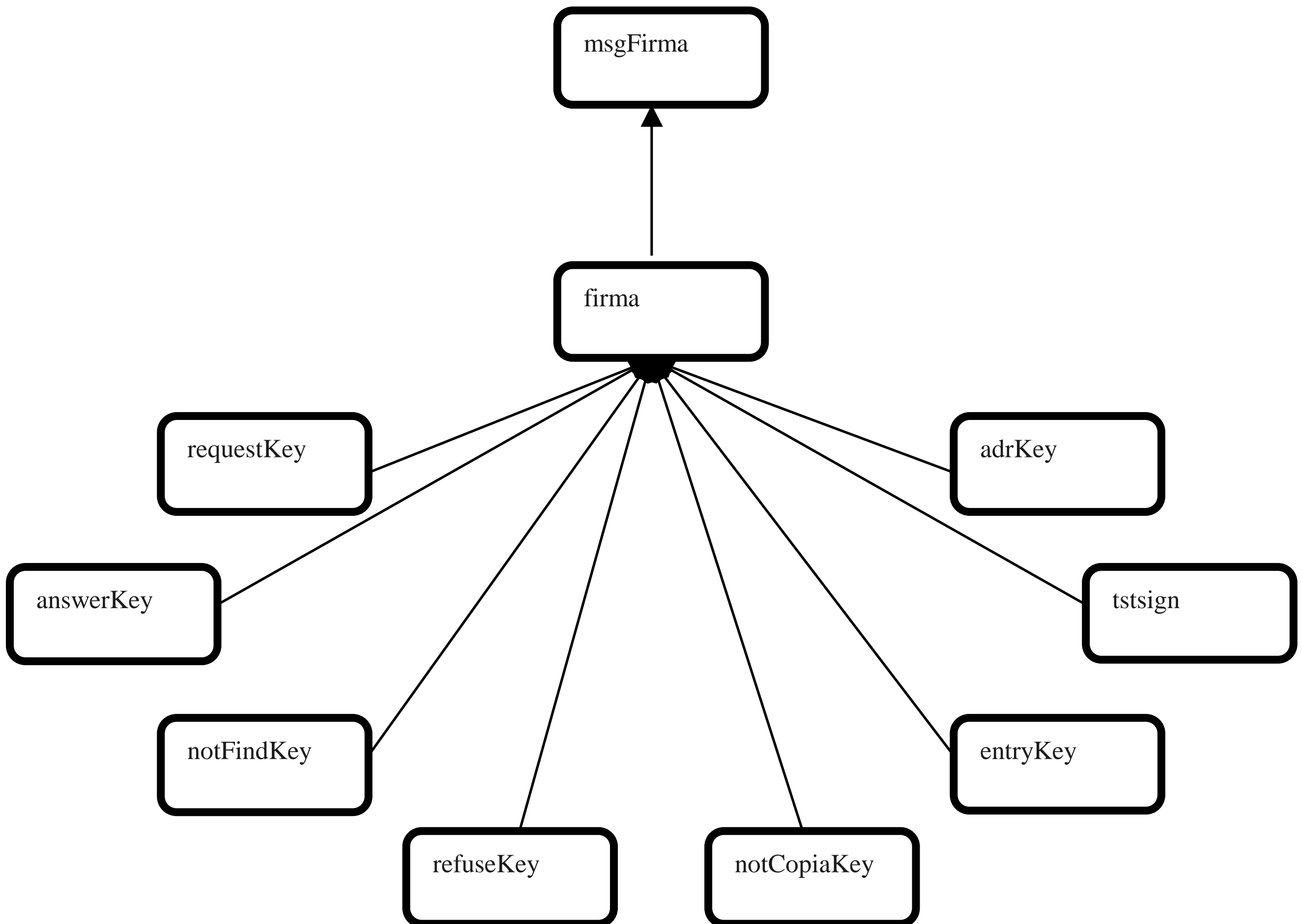


Descrizione messaggi:

- **PacketMsg**: vedi DNS.
- **msgUDP** aggiunge a questa classe un intero che svolge la funzione di timestamp; tutte le classi che ereditano da msgUDP mantengono quindi questo timestamp, che deve essere presente in tutti i messaggi e deve essere una ulteriore garanzia sulla validità dei messaggi.
- **requestMsg** contiene il nome logico del Client di cui si vuole conoscere la chiave pubblica e con cui si vuole comunicare.
- **answerMsg** è il messaggio di risposta contenente la chiave pubblica richiesta
- **notFindMsg** non definisce nessun campo particolare (oltre quello ereditato) e indica l'assenza del nome richiesto all'interno della tabella
- **timeoutMsg** non definisce nessun campo particolare (oltre quello ereditato) e indica che la Copia non ha risposto, o comunque non ho ottenuto risposta in un tempo pari a 5 sec (tempo max stimato per la ricezione di un messaggio)
- **refuseMsg** non definisce nessun campo particolare (oltre quello ereditato) e indica che il messaggio di richiesta è stato modificato
- **notCopiaMsg** non definisce nessun campo particolare (oltre quello ereditato) e indica che nessuna Copia ha risposto o non ci sono Copie disponibili a fornire il servizio;
- **reqCopieMsg** contiene un campo timestamp (viene usata solamente dal Gestore secondario per richiedere l'aggiornamento al primario).

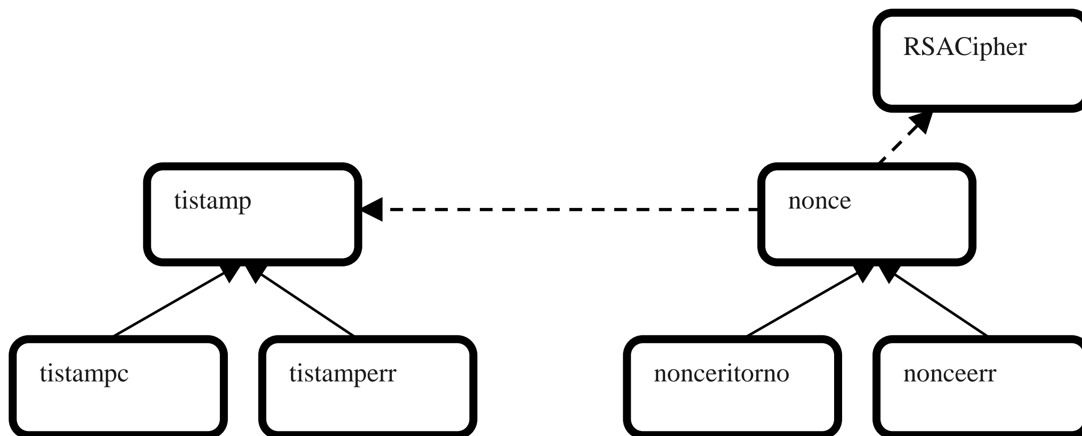
Questi messaggi non vengono inviati direttamente perché non permetterebbero l'autenticazione dell'identità del mittente (discorso a parte per il messaggio di timeout che non viene mai spedito, e il messaggio di reqCopieMsg che non necessita di autenticazione). Si è quindi preferito incapsularli all'interno di altri messaggi che contengano anche un campo signature in modo che sia sempre possibile verificare chi sia il mittente, se il messaggio è attuale e non è stato modificato . Questi

messaggi "contenitori" sono quelli costruiti a partire dalla classe msgFirma che fornisce tutte la funzioni per l'autenticazione e la verifica degli oggetti inviati.



Descrizione messaggi di autenticazione:

- **msgFirma** contiene le funzioni di:
- autenticazione dell'oggetto;
- verifica signature;
- **firma** aggiunge il campo signature che permette la verifica della consistenza del messaggio. Notare che è l'intero oggetto (requestMsg, answerMsg…) che viene codificato in modo che non sia possibile accedere al campo timestamp. In questo modo siamo sicuri per quanto concerne eventuali modifiche nel messaggio ma anche per potenziali replying di messaggi poiché in quest'ultimo caso il timestamp non andrebbe a coincidere.
- **requestKey** contiene un oggetto requestMsg ;
- **answerKey** contiene un oggetto answerMsg;
- **notFindKey** contiene un oggetto notFindMsg;
- **refuseKey** contiene un oggetto refusseMsg;
- **notCopiaKey** contiene un oggetto notCopiaMsg;
- **tstsign** contiene un oggetto tistamp;
- **entryKey** contiene un oggetto entry;
- **adrKey** contiene un oggetto adrC.



La classe tistamp contiene un timestamp (un intero) e le funzioni per recuperarlo. La classe tistampc aggiunge un ulteriore timestamp, mentre la classe tistamperr permette di inviare un messaggio di errore. La classe nonce è un oggetto contenente l'identità di chi invia il messaggio e un tistamp codificato (array di byte). Fornisce anche le funzioni per preparare (preparanonce) e verificare (verificanonce) i nonce. La classe nonceritorno contiene due nonce. Naturalmente c'è anche un messaggio di errore per indicare eventuali modifiche sul messaggio.

entry

La classe entry mantiene un nome, una chiave pubblica (in pratica una entry della tabella delle chiavi pubbliche) e un timestamp.

indirizzo

La classe indirizzo mantiene un indirizzo IP e un intero (che dovrebbe rappresentare la porta di ascolto del processo in questione).

msgByte

La classe msgByte contiene un array di byte che rappresenta il messaggio del Client criptato.

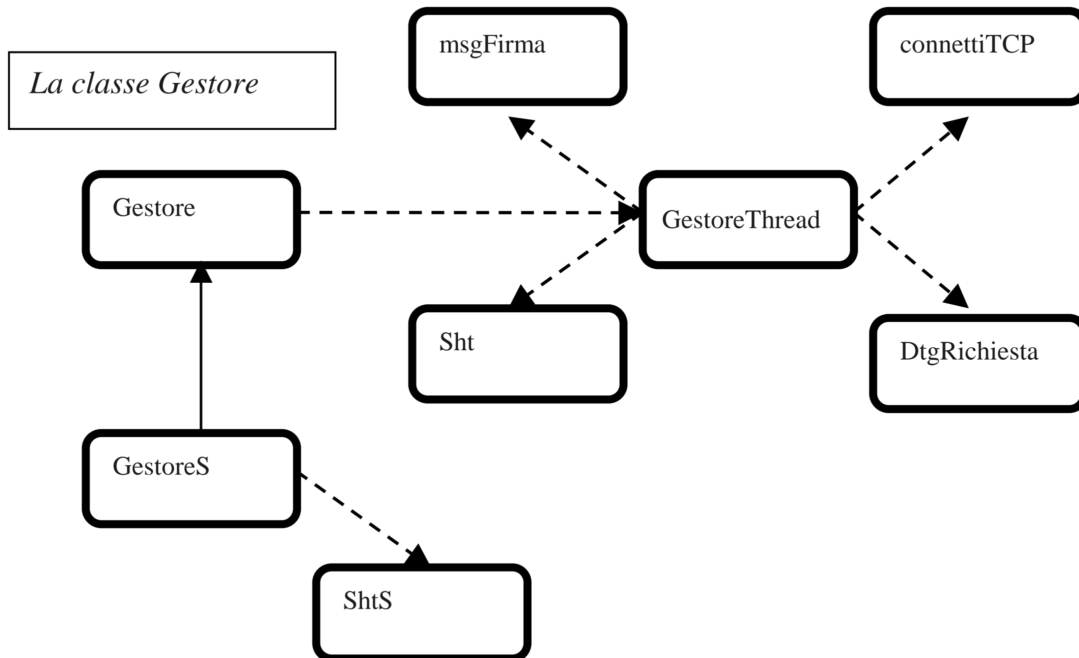
adrC

La classe adrC contiene un campo indirizzo e un timestamp. Il suo utilizzo è legato esclusivamente all'invio della lista delle Copie attive dal Gestore primario al Secondario. Il protocollo di aggiornamento è identico a quello usato tra Gestore e Copia per l'invio della tabella delle chiavi (si inviano gli indirizzi uno alla volta firmati e numerati).

RSACipher

La classe RSACipher è la classe contenente tutte le funzioni per la cifratura e decifratura di messaggi. La vedremo meglio in seguito.

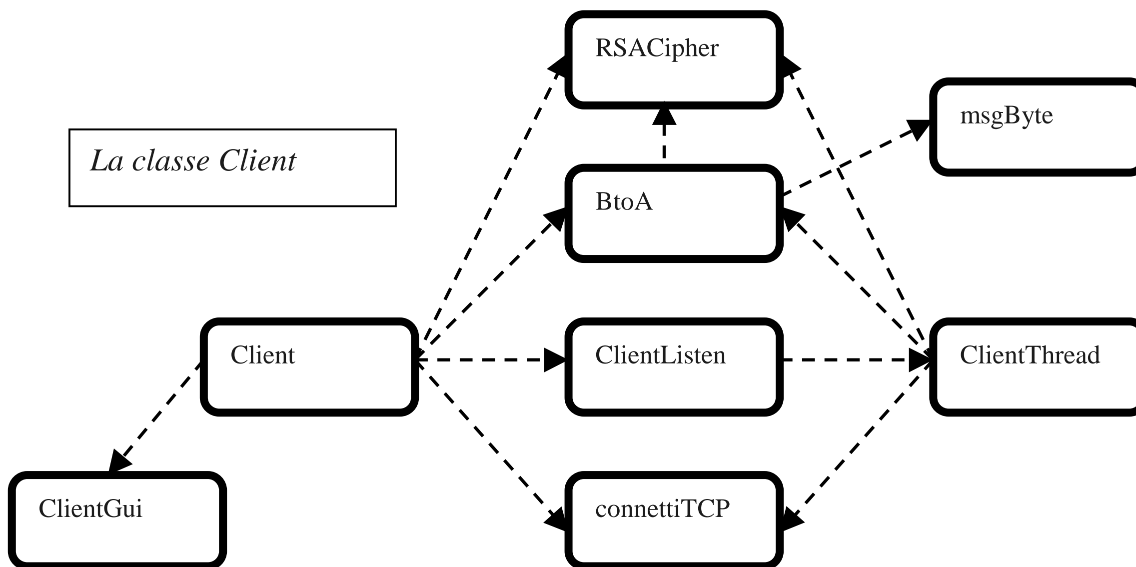
5.2.3.2 LE CLASSI PRINCIPALI



Gestore gestisce la porta di ascolto;
GestoreS eretita da Gestore, l'unica funzione nuova che implementa riguarda la richiesta di aggiornamento per la lista degli indirizzi delle Copie attualmente attive (classe ShtS)
msgFirma: vedi sopra.
GestoreThread: gestisce tutta la comunicazione del Gestore (dopo che è arrivata una richiesta).
connettiTCP: mantiene tutte le funzionalità per gestire una connessione TCP.
- creaione di una Socket TCP (lato server);
- aggancio ad una socket TCP (lato Client);
- gestione invio oggetti;
- gestione ricezione oggetti;
- chiusura socket.

DtgRichiesta: mantiene tutte le funzionalità per gestire una comunicazione UDP:
- creazione DatagramSocket;
- gestione trasmissione e ricezione datagrammi.

Sht: mantiene tutte le funzionalità per gestire il trasferimento della tabella delle chiavi pubbliche da Gestore a Copia:
- lettura Hashtable;
- generazione random di timestamp;
- trasmissione entry (firmate e numerate).

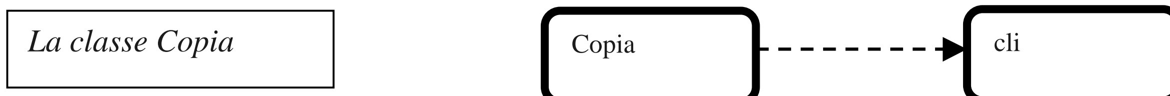


Client gestisce la comunicazione del Client richiedente;
ClientListen: apre una porta di ascolto;
ClientThread gestisce tutta la comunicazione del Cliente “richiesto”;
ClientGui gestisce l’interfaccia grafica del Client;
connettiTCP: vedi sopra;
BtoA gestisce tutta la comunicazione tra i due Client. mantiene le funzioni per:

- trasmettere e ricevere i messaggi (msgByte);
- cifrare e decifrare il messaggio;

RSACipher è la classe contenente tutte le funzioni per la codifica dei messaggi:

- scelta dell’algoritmo di crittografia da utilizzare;
- codifica/decodifica un testo cifrato con la chiave privata;
- codifica/decodifica un testo cifrato con la chiave pubblica;

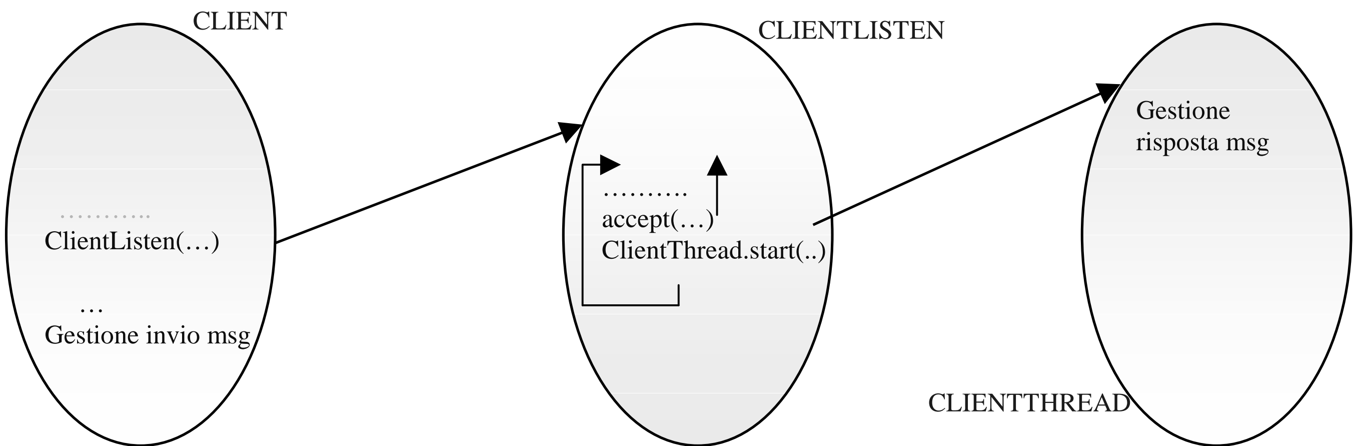


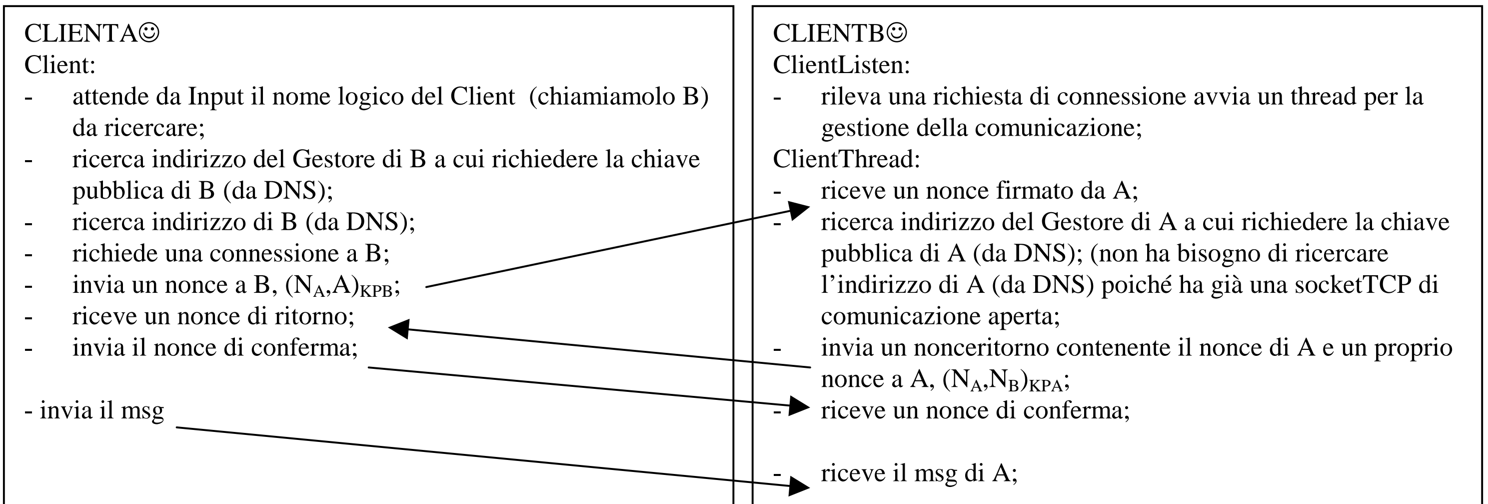
Copia gestisce tutto il lavoro della Copia;
cli mantiene tutte le funzionalità per gestire il ricevimento della tabella delle chiavi pubbliche da Gestore a Copia ed effettua tutti i controlli relativi al caso;

5.2.4 IMPLEMENTAZIONE

5.2.4.1 COMUNICAZIONE TRA CLIENT

Il Client deve in ogni momento esser pronto a inviare e ricevere messaggi. Proprio per questo il corpo principale del Client chiama un thread che si mette in ascolto su una determinata porta. Chiunque voglia comunicare con il Client utilizzerà questa porta per instaurare una connessione. Ad ogni richiesta di connessione viene generato un ulteriore thread che gestisce la comunicazione stessa (si è cercato di evitare il più possibile la serializzazione).





Ottenuta la chiave pubblica dal Gestore (di B), il Client è pronto per iniziare la procedura di autenticazione per la comunicazione con l'altro Client (naturalmente dopo averne ottenuto l'indirizzo dal DNS). Il protocollo cui facciamo riferimento è quello di Needahm-Schroeder lievemente modificato:

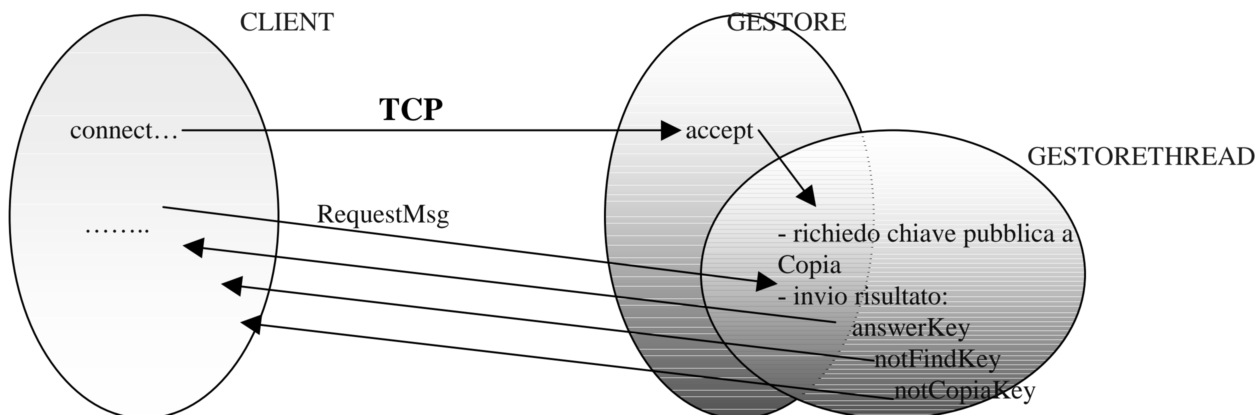
- ClientA invia un messaggio di <u>nonce</u> al ClientB in cui indica la sua identità ed un timestamp codificato con la chiave pubblica di B
- ClientB riceve il nonce, lo decifra con la sua chiave privata (solo B è in grado di decodificare il messaggio e quindi di leggere il timestamp). Reperisce quindi l'identità di A, ottiene dal Gestore la

chiave pubblica di A e risponde con un nonceritorno in cui inserisce il nonce di A decifrato precedentemente e un suo nonce. Il tutto è criptato con la chiave pubblica di A.
- ClientA riceve il nonceritorno lo decifra con la sua chiave privata (solo A è in grado di decodificare il messaggio e quindi di leggere i due nonce) e rinvia il nonceB al ClientB.

Finito questo protocollo i due Client sono a conoscenza delle rispettive identità e mantengono una socket TCP per la comunicazione. Tale comunicazione è stata semplificata e realizzata come un unico messaggio da A a B (in quanto si è ritenuto più importante il procedimento utilizzato rispetto al numero di messaggi che si potevano inviare).

Tale messaggio non può viaggiare in chiaro nel rispetto della privacy dei due Client e quindi non si è potuta utilizzare una signature per i messaggi. Inoltre la signature viene autenticata con la chiave privata e verificata con la chiave pubblica; qui avviene l'esatto contrario il messaggio viene autenticato con la chiave pubblica e verificato con la chiave privata. Si è scelto quindi di codificare il messaggio e di inviarlo(msgByte). Nella realtà il messaggio è stato suddiviso in tanti blocchi, che sono alla fine riuniti in modo che venisse spedito un unico stream di byte. Tale stream viene poi letto e decifrato all'arrivo. Tutto questo avviene in modo totalmente trasparente all'utente. La suddivisione in blocchi è stata realizzata in quanto la procedura di crittografia di java non supporta blocchi superiori ai 64 byte.

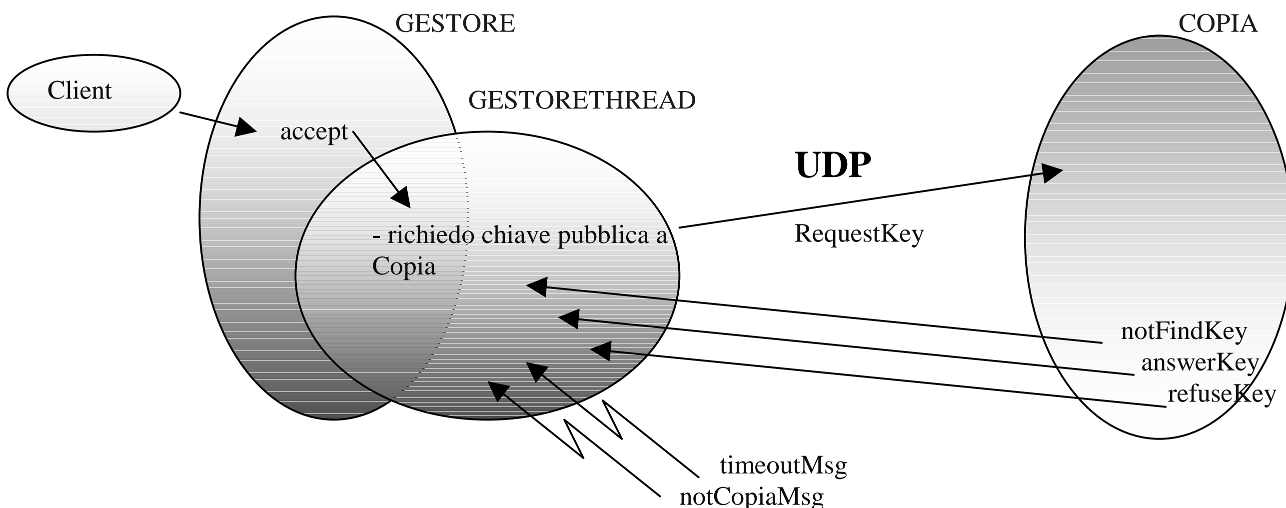
5.2.4.2 COMUNICAZIONE TRA CLIENT E GESTORE



Il ClientA richiede il nome del ClientB da ricercare. Successivamente manda un requestMsg al Gestore. Il messaggio del Client viene inviato al Gestore in chiaro tanto il Client non ha bisogno di qualificarsi perché richiede informazioni pubbliche, inoltre qualsiasi modifica al messaggio porterebbe all'invalidazione del messaggio da parte dello stesso Client o all' invio di una chiave sbagliata, nel qual caso fallirebbe la procedura di riconoscimento da parte dell' altro Client. Il Gestore comunica con una Copia per ottenere la chiave pubblica del Client richiesto (answerKey) e la invia codificata con la SUA (del Gestore) chiave privata al Client. Nel caso non sia possibile ottenere la chiave pubblica del Client il Gestore invia al richiedente un messaggio (sempre codificato con la sua chiave privata) indicante il motivo dell'impossibilità nel reperire la chiave (nessuna Copia disponibile (notCopiaMsg), nome inesistente (notFindMsg) …).

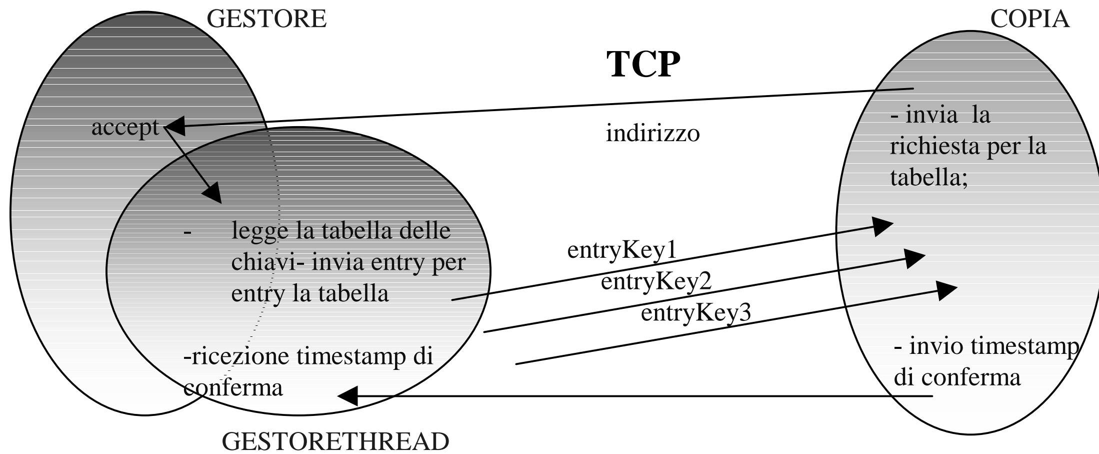
Bisogna effettuare una distinzione tra il timestamp inviato al Gestore dal Client e quelli utilizzati tra Gestore e Copie. Il timestamp di ritorno inviato dal Gestore al Client deve coincidere con quello inviato dal Client. Allo stesso modo il Gestore genera un suo timestamp interno che viene man mano incrementato a seconda dei messaggi che invia, (o in certi casi viene creato in modo random) e questo viene utilizzato per le comunicazioni con le Copie.

5.2.4.3 COMUNICAZIONE TRA GESTORE E COPIE (per ottenimento chiave pubblica)



Una volta arrivata una richiesta e verificato che si tratta di una richiesta da parte di un Client, il Gestore deve decidere a quale Copia inviare la richiesta. Il protocollo utilizzato è, come abbiamo già detto, UDP. Il Gestore consulta la lista in suo possesso contenente le Copie in funzione, le richieste vengono inoltrate ciclicamente a tutte le Copie in modo che il lavoro sia suddiviso equamente e non si creino colli di bottiglia. Determinato l'indirizzo e la porta di ascolto della Copia (tale operazione è stata resa esclusiva in modo che più richieste non vengano inoltrate contemporaneamente alla stessa Copia mentre altre Copie rimangono senza lavoro), il Gestore invia un messaggio di richiesta (requestKey) in forma di datagramma alla Copia. In caso non si riceva risposta entro un tempo prefissato (15 sec) scatta un messaggio di timeout (timeoutMsg) e il messaggio viene riinoltrato ALLA STESSA Copia. Nel caso la richiesta fallisca ancora la Copia viene considata inaffidabile e il suo indirizzo viene cancellato dalla lista mantenuta dal Gestore. Nel caso non ci siano Copie "attive" o le Copie presenti non rispondano viene inviato un messaggio indicante l'assenza di Copie in grado di fornire il servizio (notCopiaKey). In tutti gli altri casi la Copia risponde fornendo le informazioni richieste (answerKey), o indicando l'assenza del Client richiesto nella tabella (notFindKey) o avvisando che il messaggio ricevuto ha subito modifiche (refuseKey).

5.2.4.4 COMUNICAZIONE TRA COPIA E GESTORE (per la richiesta della tabella delle chiavi pubbliche)



Il primo passo svolto dalla Copia dopo la registrazione su una porta libera è l'invio di una richiesta al Gestore. Nella realtà la Copia invia al Gestore il suo indirizzo (inteso come indirizzo IP e porta di ascolto ➔ indirizzo); a questo punto il Gestore è pronto per inviare la tabella delle chiavi pubbliche. Vengono quindi inviati, in sequenza, una serie di oggetti entryKey contenenti un oggetto entry e la signature dell'oggetto stesso. Il protocollo utilizzato per l' invio della tabella parte con:

- invio della dimensione della tabella
- invio delle singole entry (nome_logico,chiave,timestamp) firmate. Una scelta implementativa è stata quella di *numerare serialmente* tutte le entry che il Gestore invia (partendo da un timestamp generato in modo casuale). Poiché il protocollo utilizzato è TCP le varie entry devono arrivare nello stesso ordine con cui sono state spedite in caso contrario viene rilevata da parte della Copia questa anomalia e viene riinoltrata la richiesta di aggiornamento. Si è quindi pensato che un numero si tentativi pari a 5 sia sufficiente per un invio corretto della tabella in caso contrario si ha l' abbandono da parte della Copia del tentativo di aggiornamento.
- ricezione di un timestamp (tstsign) firmato indicante l'ultimo timestamp ricevuto dalla Copia (e incrementato di uno). In questo modo ho intrinsecamente anche il numero di entry lette dalla Copia (che deve coincidere con la dimensione della tabella).

Mediante questo scambio di messaggi siamo certi della consistenza della tabella e dell'autenticità del mittente.

5.2.5 Modifiche al documento di specifica dei requisiti

Il progetto è stato realizzato cercando di seguire fedelmente le specifiche, ma in fase di programmazione certi aspetti sono stati lievemente modificati.

- Il DNS è stato pensato per un azienda o comunque per una entità limitata ma è stato comunque sviluppato in modo da potersi adattare a qualsiasi tipo di struttura grande o piccola che sia.

- La registrazione dell'AS presso il DNS non viene effettuata direttamente dall'AS durante la sua inizializzazione ma viene effettuata precedentemente dall' amministratore di rete. Questo anche per un discorso di sicurezza: *chi decide chi è autorizzato e chi no a registrarsi?*
- Si è ritenuto inoltre più corretto non fornire una copia della tabella ai gestori secondari ipotizzando che questa venga fornita dall'autorità responsabile alla consistenza della tabella, al pari del master
- La replicazione dell'AS, fissata inizialmente esclusivamente come una coppia Master-Slave, è stato ampliata (quasi senza accorgercene) fornendo la possibilità di aggiungere un numero qualsiasi di Slave (sebbene la soluzione ottimale rimane quella relativa a due soli gestori attivi è un contributo a scalabilità e fault-tollerance non indifferente).
- Non è più così netta neanche la distinzione tra Gestore primario e secondario. E' diventata impropria infatti la distinzione tra i gestori Master e Slave in quanto anche se su carta restano entità diverse, e così si era pensato di realizzarli inizialmente (cioè il Gestore Slave come Gestore di supporto solo in caso di caduta del Master), svolgono le stesse funzioni e lavorano sulle stesse informazioni in ogni momento.

5.2.6 Tolleranza ai Guasti

Nel caso cada il master il servizio viene comunque reso possibile dai gestori Slave. Naturalmente il Gestore quando torna in funzione non può partire da zero, ma deve conoscere la lista delle Copie attualmente attive (si trova in una situazione duale a quella di un nuovo Gestore che deve aggiornarsi presso il master). Poiché abbiamo detto che questa distinzione tra gestori Master e Slave è solo virtuale, e il DNS conosce già il suo indirizzo, può aggiornarsi come Gestore secondario facendo richiesta a un Gestore Slave (in questo modo lo Slave funziona virtualmente da Master e passa la sua lista al nuovo entrato, che così può iniziare a lavorare; la lista che si vedrà passare avrà se stesso in testa).
La caduta di uno slave non causa invece preoccupazioni, nel momento in cui verrà riavviato inizierà la sua normale procedura di aggiornamento presso il Master. Manteniamo sempre l'ipotesi di guasto singolo, inoltre ricordiamo che per convenzione il primo Gestore che deve essere attivato è sempre il Master. Quindi l'aggiornamento dei server slave rimane comunque successivo al boot del server definito come master e fa riferimento a lui (ciò non toglie che in linea di principio uno slave potrebbe richiedere l'aggiornamento ad un altro slave; si è comunque preferito inoltrare queste richieste, a meno di gravi evenienze, sempre al master).
Nel caso cadano 1 o più (anche tutte) Copie i gestori se ne accorgono e invalidano gli indirizzi in loro possesso fino al momento in cui non è più possibile fornire il servizio.