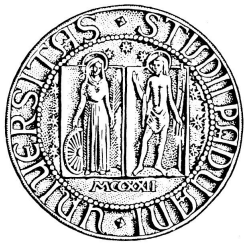

UNIVERSITA' DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

PariLogin 2009

RELATORE: Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi

CORRELATORE: Dott. Paolo Bertasi

LAUREANDO: *Andrea Castronuovo*

A.A.2010-2011

Sommario

In questa tesi di laurea triennale viene esposto il lavoro di ricerca e sviluppo svolto sul nuovo plugin Login all'interno del progetto PariPari.

Lo scopo dell'elaborato è di fornire a PariPari un meccanismo per identificare gli utenti che usufruiscono di alcuni servizi tra quelli che PariPari stesso offre.

L'esposizione inizierà con la spiegazione dell'architettura di PariPari, proseguendo con la presentazione del modulo Login. Verrà poi esposto lo sviluppo del plugin, a partire dalla fase di progettazione fino ad arrivare alla fase di implementazione.

In conclusione verranno evidenziati i risultati raggiunti e illustrati i miglioramenti da effettuare per avere un plugin più efficiente.

Indice

1 PariPari **5**

1.1 Caratteristiche principali 5

1.2 DHT 7

2 Login **9**

2.1 Instabilità nella rete 10

2.2 Algoritmo di replicazione 10

2.3 Integrità del record 12

2.3.1 Tre stati 14

2.3.2 Flag di DHT 16

2.4 Sicurezza nella replicazione 16

2.4.1 Reverse Look Up migliorato 18

3 Funzionamento generale del Login **20**

3.1 Pacchetto network 23

3.2 Creazione di un account 25

3.2.1 Proibire la creazione di un account già presente 26

3.3 Login 27
3.4 Cancellazione di un account 28
3.5 RefreshHandler 30
3.6 Servizi offerti dal Login 31
3.6.1 Ricerca di un utente 31
3.6.2 Challenge 33

4 Conclusioni **35**

Capitolo 1

PariPari

Il progetto PariPari vuole realizzare una rete peer-to-peer (d'ora in poi P2P) severless: una rete priva di un server centrale, che offre agli utenti finali un vasto assortimento di servizi.

PariPari fornisce, inoltre, una piattaforma a sviluppatori attraverso delle API[1] (*Application Programming Interface*) in modo da rendere più agevole l'implementazione di nuove applicazioni. Le API sono un metodo per ottenere un'astrazione tra il software a basso e ad alto livello, evitando così al programmatore di implementare funzioni che interagiscono con l'hardware.

Ora vediamo nel dettaglio la struttura e le principali proprietà di PariPari.

1.1 Caratteristiche principali

PariPari, come precedentemente accennato, è serverless, perciò quasi immune ad attacchi di tipo DoS (*Denial of Service*). Questo tipo di attacco porta al limite delle prestazioni il sistema informatico bersagliato, effettuando un elevato numero di richieste. A causa dell'assenza di nodi centrali, lo sforzo

[1] Cfr. http://en.wikipedia.org/wiki/Application_programming_interface

da parte di un cracker[2] sarebbe inutile.

Un'altra importante caratteristica di questo progetto è la *modularità*. Infatti è possibile suddividere PariPari in tre principali sezioni: *Core*, nucleo centrale che garantisce il funzionamento di PariPari stesso, *plugin della cerchia interna*, gestori delle risorse, e *plugin della cerchia esterna*, altre applicazioni offerte da questa piattaforma.

Per ogni nuovo servizio da fornire all'utente finale, bisogna creare un nuovo plugin. Esso utilizzerà le API messe a disposizione dagli altri plugin per usufruire dei loro servizi e ognuna delle richieste effettuate verrà gestita dal Core che si occuperà di interpellare il modulo in grado di soddisfarla.

La struttura modulare ha un vantaggio anche per gli utenti finali. Infatti essi possono selezionare i programmi da caricare, evitando di appesantire PariPari con applicazioni alle quali essi non sono interessati. Inoltre il caricamento di un nuovo plugin è estremamente semplice (basta digitare da linea di comando *add nomeplugin* per aggiungere il plugin *nomeplugin*) e sarà ancora più intuitivo con la GUI[3] (*Graphical User Interface*) in via di sviluppo.

Un'altra proprietà da evidenziare è la *portabilità*. PariPari è interamente realizzato in linguaggio Java e quindi in grado di essere eseguito su qualsiasi sistema operativo avente la Java Virtual Machine. In particolare PariPari si appoggia a Java Web Start, in modo da poter essere lanciato direttamente dal browser, senza dover installare nulla.

[2]Cfr. http://it.wikipedia.org/wiki/Cracker
[3]Cfr. http://en.wikipedia.org/wiki/Gui

1.2 DHT

PariPari, non avendo una struttura centralizzata, utilizza la DHT[4] (Distribuited Hash Table). Questa architettura, presentata per la prima volta nel 2001, è indispensabile per gestire un vasto numero di nodi.

Ad ogni nodo e ad ogni risorsa della rete viene assegnato un numero identificativo (per il nodo chiamato ID, mentre per la risorsa Hash) solitamente da 160 bit[5]. Il numero di bit è stato scelto sufficientemente elevato da ridurre al minimo l'evenienza di una collisione (quando a due diversi nodi viene assegnato lo stesso ID). La probabilità di collidere dipende, inoltre, dall'uniformità nel distribuire gli ID da parte della funzione hash utilizzata, che accetta in ingresso una chiave e fornisce in uscita l'identificativo cercato (per i nodi la chiave è formata da: indirizzo IP + porta utilizzata da DHT).

Per stabilire a quale nodo una determinata risorsa debba essere destinata, è necessario introdurre il concetto di distanza e il significato di *spazio metrico*. Uno spazio metrico è una struttura matematica formata da una coppia di elementi *(X,d)*, dove X è un insieme e d una funzione (per le reti Kademlia[6] "metrica XOR"). Tale funzione è chiamata distanza e associa a due elementi x e y dell'insieme X un numero reale non negativo *d(x,y)* avente determinate proprietà.

Ora è possibile decretare il nodo più vicino a una determinata risorsa, in particolare il proprietario di tale risorsa. Il vero punto di forza di questa architettura è il ridotto numero di informazioni che ogni nodo deve memorizzare per soddisfare una possibile ricerca. Infatti se consideriamo la rete come una circonferenza (modello tipicamente utilizzato da CHORD[7]) e la dividiamo in n parti, dove n è proporzionale al numero di nodi presente nella

[4]Cfr. http://en.wikipedia.org/wiki/Distributed_hash_table
[5]In PariPai attualmente si usano 256 bit.
[6]Cfr. http://en.wikipedia.org/wiki/Kademlia
[7]Cfr. http://en.wikipedia.org/wiki/Chord_(peer-to-peer)

rete, ogni nodo deve memorizzare in locale h informazioni sui nodi ($5 \leq h \leq 10$) della metà alla quale non appartiene, h nella stessa metà ma nell'altro quarto ecc. La costante h diminuisce per frazioni più piccole.

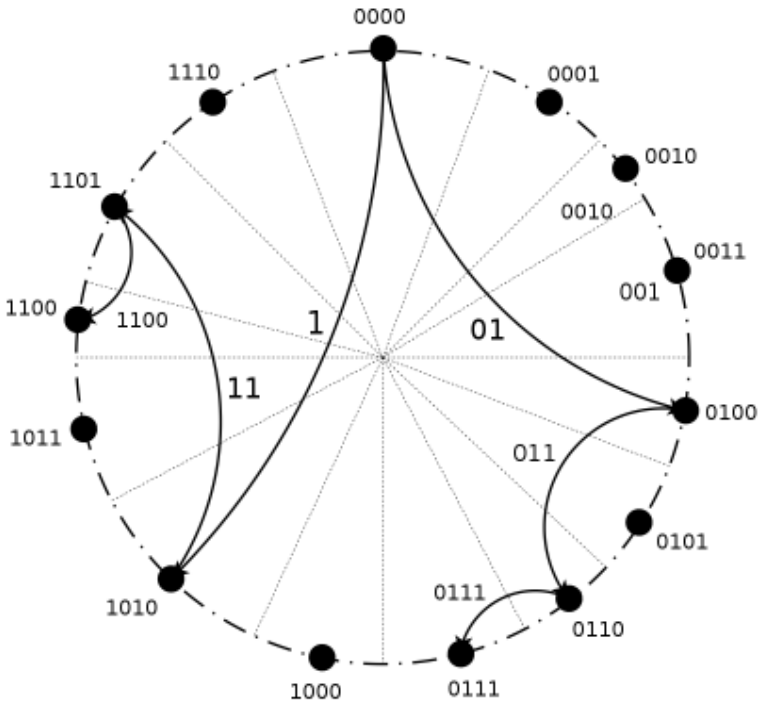

Figura 1.1: Struttura della DHT (CHORD)

Quando un nodo deve condividere una risorsa effettua prima una ricerca per individuare il nodo più vicino alla risorsa e in seguito gli fornisce le coordinate per raggiungerla. Diversamente quando un nodo necessita di una risorsa cerca il proprietario della risorsa (quello avente ID più vicino all'hash della risorsa) che gli restituirà le sue coordinate.

Capitolo 2

Login

Benché possa sembrare contraddittorio in un'architettura serverless come quella di PariPari, il plugin Login è estremamente importante. Esso ha il compito di identificare univocamente gli utenti di PariPari, in particolare risulta necessario per alcuni plugin, come ad esempio DNS[1] (*Domain Name System*), che devono verificare la reale identità dell'utente "*loggato*".

Ad ogni utente, oltre all'*alias* da lui stesso scelto (ovviamente non esistono due utenti con lo stesso *alias*), vengono assegnate due chiavi RSA[2] da 1024 bit (una pubblica e una privata) uniche all'interno di PariPari. Utilizzando la crittografia asimmetrica[3] sarà poi possibile accertare la reale corrispondenza della persona connessa.

Il plugin Login, essendo un server distribuito, memorizza le informazioni di ogni utente nella rete, in modo da poterle successivamente recuperare da qualsiasi postazione.

In seguito verrano analizzati i problemi riscontrati con le rispettive soluzioni.

[1] Cfr. http://en.wikipedia.org/wiki/Domain_Name_System
[2] Cfr. http://en.wikipedia.org/wiki/RSA
[3] Cfr. http://en.wikipedia.org/wiki/Public-key_cryptography

2.1 Instabilità nella rete

In una rete P2P il comportamento di un nodo è riassumibile in tre fasi: *ingresso* nella rete, *partecipazione* e *uscita* dalla rete. Per ingresso nella rete s'intende quando l'utente avvia l'applicazione, per uscita quando la chiude , mentre per partecipazione quando utilizza effettivamente tale applicazione.

Il continuo ingresso e uscita da parte di migliaia di nodi provoca il fenomeno chiamato *churn*. In particolare per churn, più precisamente *churn rate*, si intende il numero di nodi che abbandonano il sistema in un determinato periodo, solitamente un'ora.

Questo fenomeno è un problema significativo per sistemi a larga scala, in quanto essi devono mantenere tutte le informazioni sui nodi presenti nella rete in modo da poter operare efficientemente. Perciò a causa di questo problema è stata intrapresa la scelta di memorizzare il record in più nodi nella rete. Inoltre si fa presente che ogni simulazione o analisi di una rete P2P si basa su qualche modello di churn.

Passiamo ora alla descrizione dettagliata dell'algoritmo nel seguente paragrafo.

2.2 Algoritmo di replicazione

Prima di descrivere l'algoritmo è necessario sapere come avviene la scelta dei nodi sui quali registrare il record[4] relativo all'utente.

Per individuare tali nodi si sfrutta la funzione hash di DHT (vedi par.1.2), fornendogli in ingresso una chiave variabile nel tempo. Tale chiave è una stringa composta dai seguenti tre parametri:

[4]La composizione del record sarà descritta nel par.3.2

$$username(A) + date + hour$$

dove *username(A)* è il nome scelto dall'utente A, *date* la data corrente e infine *hour* l'ora attuale. I campi variabili nel tempo sono ovviamente *date* e *hour*, mentre *username(A)* resta stabile. Precisamente ad ogni inizio d'ora la coppia (*date*, *hour*) muta, perciò il record dovrà essere copiato nella destinazione più vicina in metrica XOR all'hash della chiave precedentemente descritta.

Dopo un'introduzione sull'individuazione dei nodi, viene ora descritto dettagliatamente l'algoritmo di replicazione.

Algorithm 1 Algoritmo di replicazione

1. L'utente A richiede al modulo Login locale di creare un account.
2. Il modulo Login effettuerà le seguenti operazioni:
   (a) usando come data e ora quelle attuali (H) assembla una chiave e richiede a DHT il nodo più vicino all'hash della chiave calcolata;
   (b) usando come data e ora quelle di un'ora precedente (H - *1*) assembla una chiave e richiede a DHT il nodo più vicino all'hash della chiave calcolata;
   (c) ...
   (d) usando come data e ora quelle di k ore precedenti (H - k) assembla una chiave e richiede a DHT il nodo più vicino all'hash della chiave calcolata;
3. Invia il record contente i dati dell'utente A ai nodi individuati;
4. Ognuno di questi nodi effettuerà, se è possibile, lo "*STORE*" su DHT usando come chiave l'alias scelto dall'utente A;

Nel punto *4* dell'algoritmo con le parole: "*se è possibile*" si vuole precisare che la creazione dell'account potrebbe fallire (Ad esempio esiste già un utente con lo stesso alias). Inoltre con il parametro k si indica la molteplicità del record.

A causa del *churn* il numero di copie del record relativo all'utente A potrebbe diminuire fino a estinguersi. Inoltre potrebbero entrare nodi più vicini agli hash precedentemente calcolati. Perciò nell'arco di un'ora sono previsti dei punti di "*refresh*", nei quali si rieseguono le operazioni sopraelencate (dal punto 2 al punto 4 dell'algoritmo di replicazione) per ogni record in locale. In particolare i punti di refresh sono quattro durante l'ora: *hh:00*, *hh:15*, *hh:30* e *hh:45*.

Bisogna definire che le richieste di memorizzazione inviate in ogni punto di refresh si chiamano *refresh request*, mentre i nodi individuati, dopo l'esecuzione dell'algoritmo di replicazione per un determinato record R, verranno denominati "*nodi responsabili del record R*".

Il tempo assume un ruolo determinante in questo algoritmo. Esso ha il compito di coordinare il comportamento dei nodi presenti nella rete, senza che essi si scambiano alcuna informazione.

2.3 Integrità del record

Prima di affrontare i problemi relativi all'integrità del record è indispensabile descrivere il comportamento locale del nodo per poi comprendere meglio tali problemi.

Ogni nodo all'interno di una sessione di refresh riceve un determinato numero refresh request. Il comportamento del nodo cambia a seconda del numero di refresh request uguali ricevute e se il record era già presente in locale. In particolare il nodo può assumere due atteggiamenti:

- se il record è già in suo possesso, lo riconferma[5] se il numero di refresh request per quel determinato record è maggiore di k/2, altrimenti lo cancella;
- se il record non è già in suo possesso, lo conferma e lo memorizza in locale se il numero di refresh request è maggiore di k/2, altrimenti lo ignora;

A causa di questo atteggiamento da parte del nodo sono stati individuati diversi problemi.

Uno di essi si potrebbe verificare se l'utente crea il proprio account a cavallo di una sessione di refresh. In particolare alcuni di questi nodi all'inizio del refresh potrebbero aver memorizzato in locale il record e in seguito invieranno le refresh request.

ESEMPIO[6]: le refresh request sono minori di k/2 (il numero di refresh request uguali e tre mentre k/2 vale quattro), per questo motivo i nodi 5, 6, 7 cancelleranno il record in locale. Finito il refresh R1 i nodi 1, 2, 3 e 4 avranno inserito in locale il record, poiché hanno processato la richiesta d'inserimento. Come si può vedere nella sezione (b) della Fig. 2.1 prima che inizi il refresh R2 il nodo 4 esce dalla rete (*bad node*[7] o guasto), perciò si verifica la stessa situazione precedentemente descritta. Infatti anche i nodi 1, 2 e 3 cancelleranno il record in locale.

[5]Per riconferma o conferma s'intende effettuare lo "STORE" su DHT

[6]Numeriamo i nodi della rete in Fig. 2.1 (a), il numero 1 è in alto a sinistra, il numero 2 è alla destra di quello precedente, fino ad arrivare al numero 4. Il numero 5 è in basso a sinistra, quello a fianco il numero 6 fino ad arrivare al numero 7. Questa numerazione vale anche per la Fig.2.1(b) - (c) e per l'intera Fig. 2.2

[7]Per *bad node* si intendono quei nodi di una rete il cui comportamento crea danno per gli altri, distribuendo falsi servizi o risorse non richieste.

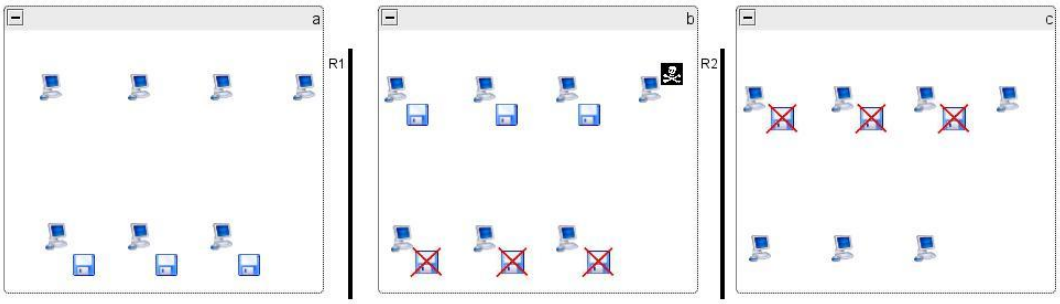

Figura 2.1: Inserimento a cavallo di un Refresh

Tale problema sarà risolto con l'introduzione dei "*Tre stati*"(Vedi par. 2.3.1).

Un altro problema riscontrato si potrebbe verificare se avviene l'ingresso nella rete da parte di alcuni nodi a cavallo di un refresh. Precisamente potrebbe succedere che una parte di nodi invia le refresh request a un determinato insieme di nodi, mentre l'altra a un altro insieme (l'intersezione dei due insiemi potrebbe essere anche vuota). In realtà non si tratta di un vero problema, perchè si pensa che la rete non sia soggetta a un alto churn rate. Questa questione è comunque stata risolta dal "*Flag su DHT* "(Vedi par. 2.3.2).

2.3.1 Tre stati

Un record può assumere tre stati: 0 ,1 o 2. Quando il nodo effettua l'inserimento in locale del record R (in seguito a una richiesta d'inserimento oppure alla ricezione di più di k/2 refresh request uguali), pone il suo stato al valore 0. D'ora in poi lo stato di R verrà incrementato di un'unità ogni inizio di una sessione di refresh, mentre sarà resettato solo successivamente all'ottenimento di un numero maggiore di k/2 refresh request uguali. Al termine di una sessione di refresh se lo stato di R assume valore pari a 2, il nodo provvederà a eliminarlo prima di dichiarare finita tale sessione di refresh.

La politica del nodo in seguito all'introduzione di questa variante diventerà il seguente:

- se il record è già in suo possesso, lo conferma se il numero di refresh request ricevuti è maggiore di k/2, altrimenti non compie nessuna azione;
- se il record non è presente in locale, lo salva e ne effettua lo "STORE" se il numero di refresh request è maggiore di k/2, altrimenti lo ignora;

Ora si è ripreso lo stesso esempio di Fig. 2.1 ma valutato dopo l'introduzione dei "Tre stati".

ESEMPIO: L'inserimento avviene in un periodo antecedente l'inizio della sessione di refresh R1. In particolare nodi 5, 6, 7 soddisfano la richiesta di inserimento qualche istante prima dell'inizio del refresh. Essi, oltre a incrementare lo stato di un'unità, invieranno le refresh request, ma i nodi destinatari ignoreranno tali richieste poiché minore di k/2. Prima dell'inizio del refresh R2 i nodi 1, 2, 3 e 4 avranno processato la richiesta di inserimento. Supponendo che il nodo 4 non invii le refresh request, avrei comunque un numero sufficiente di refresh request per poter continuare a replicare del record.

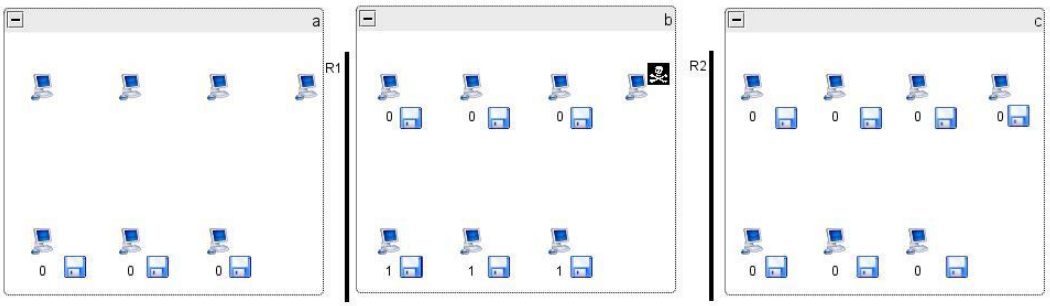

Figura 2.2: Inserimento a cavallo di un Refresh con i "Tre stati"

Attraverso l'introduzione dei "Tre stati" la perdita di un record risulterà più complicata, ma garantisce la soluzione del problema precedentemente esposto (vedi par. 2.3).

2.3.2 Flag di DHT

Il flag su DHT può assumere due valori: 0 o 1 (0 se è inattivo, 1 se è attivo). Questo variazione, rispetto al comportamento base del nodo, risulta essere una miglioria. Infatti serve a eliminare quei nodi che permangono per un breve periodo all'interno di PariPari.

Un nodo, entrato in PariPari, risulta essere inattivo, infatti, non appartiene ancora all'insieme di nodi che formano il server Login (ogni nodo è obbligato a offrire il servizio di Login). Dopo un determinato periodo (circa 30 minuti) il nodo effettua uno "STORE" su DHT ponendo il flag a un valore pari a 1. Tale operazione dovrà essere effettuato necessariamente all'esterno di una sessione di refresh, in modo che non si possa verificare il problema precedentemente descritto (vedi par.2.3).

D'ora in avanti il nodo offrirà il servizio Login fino alla chiusura di PariPari. Nel caso l'utente dovesse riavviare PariPari il flag sarà nuovamente posto a 0, per poi essere riportato a 1 dopo un determinato periodo (sempre 30 minuti).

2.4 Sicurezza nella replicazione

I nodi, sino ad ora considerati, assumevano un comportamento corretto, sintetizzato nel par. 2.3. Questa ipotesi è errata; infatti potrebbero esserci nodi avente un atteggiamento deleterio nei confronti degli altri nodi (Tale ipotesi è stata avanzata per rendere più semplice la comprensione degli argomenti sinora trattati). D'ora in poi il modello di rete prevederà l'esistenza di no-

di cattivi, presumendo che il numero di tali nodi è una piccola percentuale rispetto alla cardinalità dell'intera rete.

Dopo un'attenta analisi si è notato, che l'unico modo disponibile da parte di un nodo malvagio di compromettere il contenuto dei record di altri nodi, risulta essere l'invio di refresh request fasulle. In particolare un insieme di k/2 nodi cattivi, accordandosi, potrebbe modificare qualsiasi record. Ovviamente tali nodi devono inviare k/2 refresh request uguali.

Per far fronte a questo problema è stato introdotto il "*Reverse Look Up*", un meccanismo di filtraggio delle refresh request in ingresso in modo da neutralizzare eventuali attacchi. Questo meccanismo consiste nel compiere, per ogni refresh request diversa ricevuta, le operazioni descritte nel punto *2* dell'algoritmo di replicazione (vedi par. 2.2). Utilizzando il Reverse Look Up si eliminano tutte le refresh request ricevute dai nodi che si sono create un record in locale fasullo e lo hanno in seguito inviato nella rete, senza essere effettivamente il nodo più vicino alla risorsa.

Questa tipologia di Reverse Look Up tende a scartare molte refresh request che dovrebbero essere considerate buone. Infatti le operazioni compiute nel punto *2* dell'algoritmo di replicazione individuano i nodi attualmente proprietari del record e non quelli precedentemente in possesso del record. Per chiarire meglio il contesto, nel quale ci troviamo, è necessario un esempio.

ESEMPIO: Consideriamo i nodi A, B, C, D, E, F e G siano rispettivamente più vicini agli hash H1, H2, H3, H4, H5, H6 e H7 e siano le ore 12:00. Supponiamo che alle ore 12:10 il nodo più vicino all'hash H4 sia M. Ne consegue che al refresh delle 12:15 le refresh request inviate da D saranno considerate proveniente da nodi cattivi.

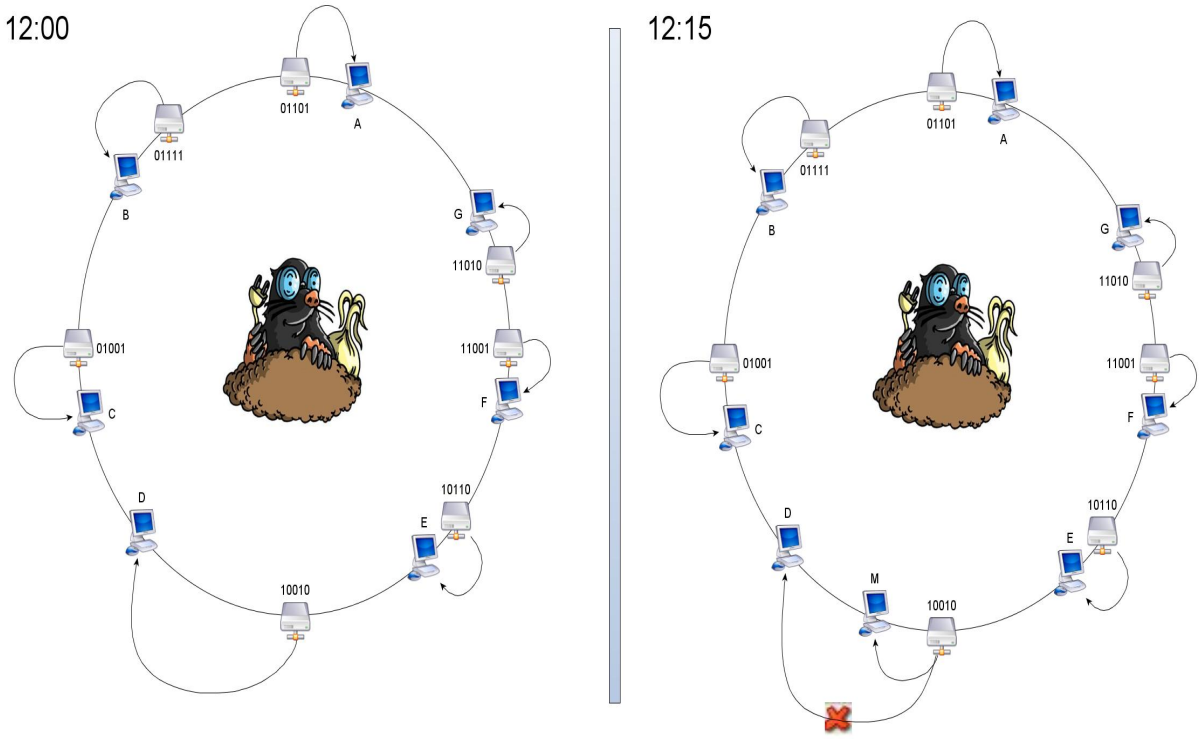

Figura 2.3: Esempio di Reverse Look Up

Con questo esempio si è considerato che solo un nodo venga rimpiazzato da un altro nodo più vicino, ma se i nodi crescono fino a raggiungere un numero maggiore di k/2 il relativo record viene eliminato. Passiamo ora alla descrizione del Reverse Look Up migliorato, una variante di quello precedentemente proposto.

2.4.1 Reverse Look Up migliorato

Con questo nuova diversificazione, rispetto al Reverse Look Up precedentemente descritto, si risolve parzialmente il problema del nodo rimpiazzato da un altro. In particolare il Reverse Look Up migliorato si comporta in due modi differenti a seconda se il record sia già in suo possesso.

Analizziamo il caso nel quale il nodo abbia il record R in locale. All'inizio della sessione di refresh il nodo esegue le operazioni elencate nell'algoritmo di replicazione individuando i nodi proprietari del record, come precedentemente definito saranno chiamati "nodi bersaglio". Tali nodi verranno poi memorizzati in un'opportuna struttura (hashtable), che chiameremo "*tabella di Look Up*", formata dalla coppia *(R, nodi bersaglio)*. Al refresh successivo, qualora il nodo dovesse ricevere refresh request per il record R, sfrutterà la "tabella di Look Up" precedentemente costruita per verificare la provenienza dei record.

Analizziamo ora il caso nel quale il nodo non abbia il record in locale. Non appena il nodo riceve delle refresh request per il record R, non avendo la "tabella di Look Up", effettuerà il "*Reverse Look Up al volo*". Esso è identico al Reverse Look Up del par. 2.4, ma si fa restituire i tre nodi più vicini all'hash di ogni risorsa.

Il problema, esposto nell'esempio del par. 2.4, viene risolto dal "Reverse Look Up al volo". Infatti i nodi non scartano più le refresh request proveniente da C perché sarà compreso nei tre nodi più vicini alla risorsa avente hash "10010". La possibilità di accettare refresh request nocive aumenta, infatti è più probabile che ci sia un nodo malvagio in $(3 * k)$ rispetto ai soli (k).

Capitolo 3

Funzionamento generale del Login

Il plugin Login è scindibile in quattro diverse sezioni (in java "*Package*") ognuna delle quali avente un compito differente. Seguirà una rapida descrizione di queste sezioni e del relativo ruolo:

- *Network*: il suo compito è quello di inviare e ricevere le richieste dalle rete e smistarle nell'opportuna coda;
- *Server*: il suo compito è di valutare opportune condizioni e inserire o aggiornare o cancellare il record in locale;
- *Refresh*: il suo compito è quello di inviare e ricevere le refresh request ed eventualmente fare il Reverse Look Up;
- *Protocol*: il suo compito è quello di definire la struttura del record in caso di richiesta di soluzione;

Definita la struttura iniziamo ora a descrivere il funzionamento generale del Login. La classe principale, che estende *Plugin*, è *Login*. Essa ha il compito di

lanciare i principali thread[1] necessari per l'avvio del plugin stesso, in particolare istanzia *LoginServer* e *NetInizializer*. Quest'ultima classe verrà ampiamente descritta nel prossimo paragrafo (vedi par.3.1), mentre LoginServer verrà rapidamente analizzato.

LoginServer si occupa di inizializzare *RefreshHandler* e *TransactionRequestHandler*. Questi due thread sono rispettivamente il gestore delle refresh request e delle transaction request, che verrano successivamente lanciati, come precedentemente accennato, dalla classe Login. L'esposizione delle principali caratteristiche dei due gestori di richieste è rimandata ai prossimi paragrafi.

Oltre all'inizializzazione dei due thread precedentemente citati, LoginServer contiene i metodi necessari per creare, cancellare un account, loggarsi e per la ricerca di un determinato utente.

Nel prossimo paragrafo analizzeremo e descriveremo il funzionamento del package Network.

[1] Cfr. http://en.wikipedia.org/wiki/Thread_%28computer_science%29

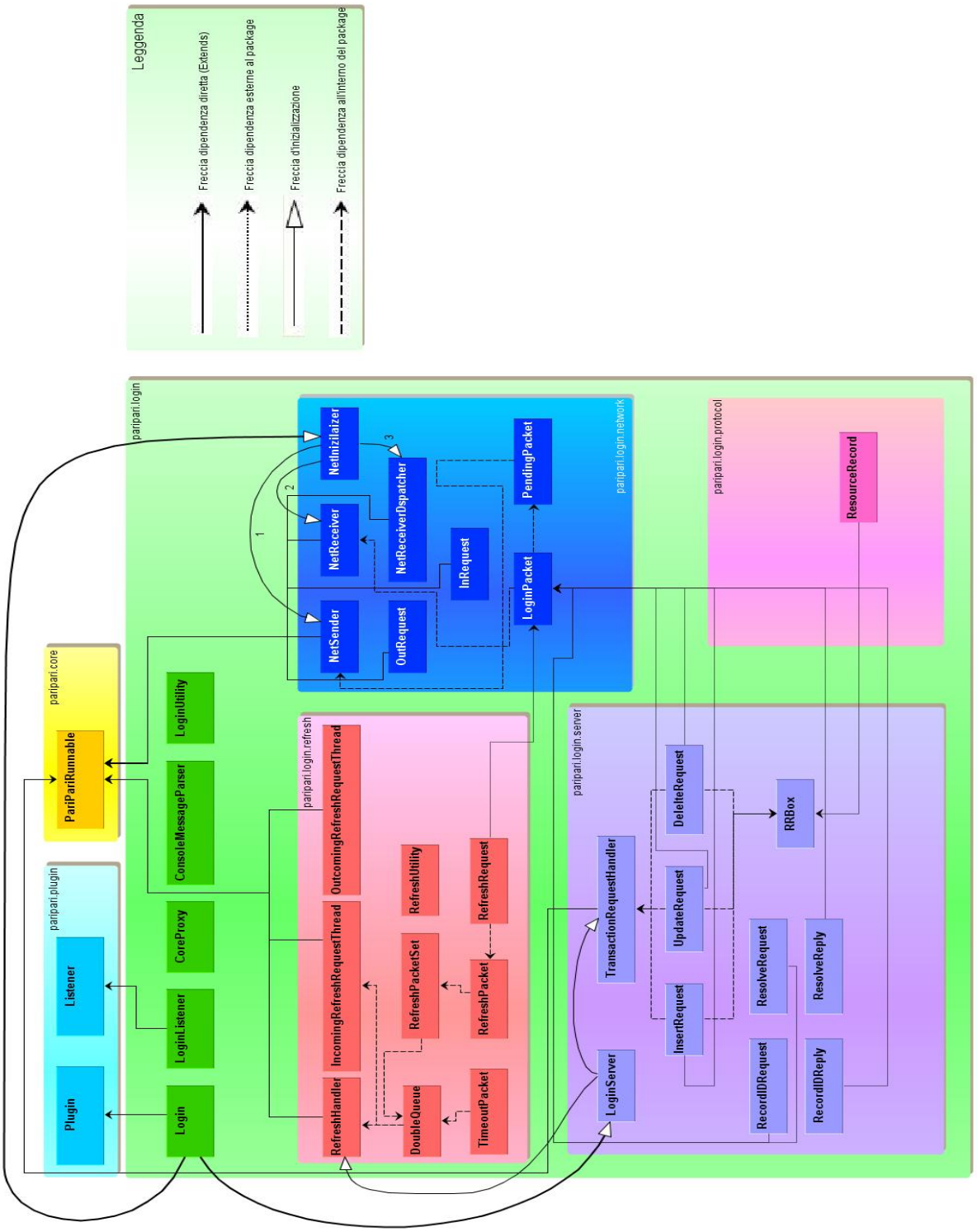

Figura 3.1: Struttura generale del plugin Login

3.1 Pacchetto network

Ogni thread, che deve comunicare all'esterno, genera e lancia un thread chiamato *OutRequest* al quale gli affida l'indirizzo IP del destinatario e un *LoginPacket*. Quest'oggetto contiene un byte che identifica il tipo di richiesta, un id diverso per ogni pacchetto e la richiesta in sé. Inoltre l'OutRequest crea un oggetto chiamato *PendingPacket* che, oltre a essere inserito nell'hashtable contente tutte le richieste in sospeso, lo accoda nella *queue* del *NetSender*.

NetSender prende il primo oggetto dalla coda, preleva la request (un oggetto LoginPacket) e dopo averlo serializzato lo invia al destinatario. A destinazione non appena arriva un pacchetto, il thread *NetReceiver* deserializza tale pacchetto e lo inserisce nella coda condivisa con il *NetReceiverDispatcher*.

A seconda del tipo di request ricevuto il NetReceiverDispatcher si comporta in un modo diverso. Come si può vedere dalla Fig. 3.2 se è una *reply* la inserisce nel relativo PendingPacket e risveglia l'OutRequest in sospeso. Invece se è una *RefreshRequest* o *TransactionRequest* crea un PendingPacket che funge da ack e inserisce la request nell'apposita coda (*DoubleQueue* o *transactionQueue*). Infine se il pacchetto ricevuto è del tipo *ResolveRequest* o *DomainIDRequest* il NetReceiverDispatcher genera e lancia un thread chiamato *InRequest*. Esso si occuperà di soddisfare la richiesta e invierà il risposta al mittente.

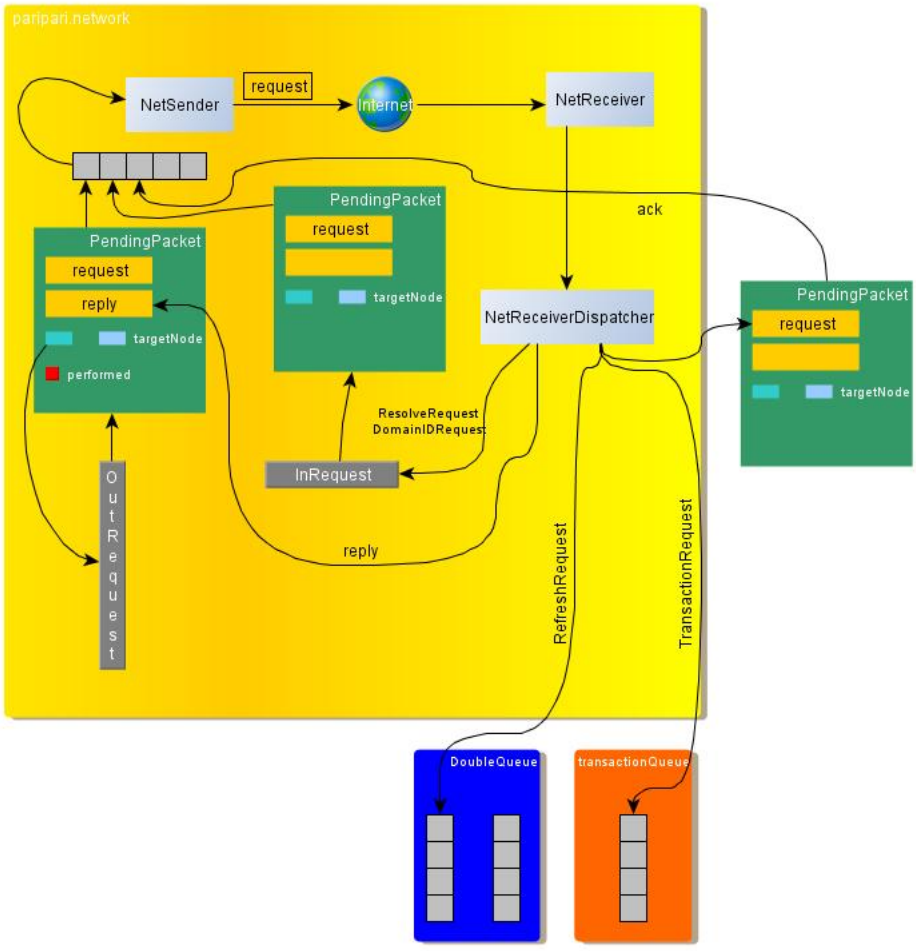

Figura 3.2: Funzionamento del package Network

3.2 Creazione di un account

In questo paragrafo analizzeremo il comportamento di un nodo che invia una richiesta di creazione di un account e le operazione eseguite da un nodo quando riceve una richiesta di questo tipo.

Prima di analizzare tali comportamenti è indispensabile spiegare la composizione del record. Esso è formato da 6 campi: *username*, *recordId* (un numero identificativo del record, uguale per tutti i record di un determinato utente), *chiave pubblica* dell'utente, un oggetto *ResourceRecord* contenente l'*indirizzo IP* dove contattare l'utente, *timestamp* e *chiave privata criptata* con la chiave simmetrica generata dalla password dell'utente. In realtà l'ultimo campo è opzionale, infatti la chiave privata potrebbe essere salvata dall'utente in un apposito dispositivo memorizzazione.
Il comportamento del nodo, nel quale viene effettuata una richiesta di creazione di un determinato account, è sintetizzato nei seguenti punti:

1. Il plugin Login locale al nodo effettua un "FIND" su DHT dell'alias scelto dall'utente: se la ricerca non restituisce alcun risultato prosegue con i successivi punti, altrimenti comunica all'utente che ne esiste già uno con quel determinato username;

2. Individua i proprietari del record, effettuando le stesse operazioni del punto 2 dell'algoritmo di replicazione;

3. Genera una coppia di chiavi RSA da 1024 bit, una pubblica e una privata;

4. Genera dalla password dell'utente una chiave simmetrica e cripta con essa la chiave privata;

5. Assembla il record e invia ai nodi precedentemente calcolati;

Il comportamento del nodo, che si vede arrivare una richiesta d'inserimento, è riassumibile nei seguenti punti:

1. TransactionHandler preleva la richiesta di inserimento dalla coda condivisa con il NetReceiverDispatcher;
2. Se il nodo possiede il record avente lo stesso username in locale scarta l'inserimento, altrimenti prosegue con il punto successivo;
3. Effettua un "FIND" su DHT utilizzando come chiave l'username della richiesta: se il numero di nodi restituiti da DHT è minore della molteplicità k proseguo con il punto successivo, altrimenti termino l'esecuzione;
4. Inserisce il record in locale;

Nel seguente sottoparagrafo verrà spiegato come un nodo non possa creare un account già presente, evidenziando che il comportamento dei nodi precedentemente descritto proibisce già tale azione.

3.2.1 Proibire la creazione di un account già presente

Un utente potrebbe forzare la creazione di un account già esistente, inviando una richiesta di inserimento in modo casuale oppure ai nodi vicini ai proprietari del record. Nel caso in cui questi nodi inserissero il record e fossero maggiori di k/2, ci potrebbe essere il rischio che i nuovi proprietari del record (essendo nuovi devono per forza effettuare il Reverse Look Up al volo) possano inserire in locale una copia errata del record stesso.

Dal momento che non si ha alcun controllo sui nodi "malvagi", la creazione di un account già esistente deve essere impedito dai nodi destinatari della richiesta. Un nodo, prima di soddisfare una richiesta di questo tipo, controlla

in locale se possiede il record ed effettua il "FIND" su DHT dell'username della richiesta. Se queste due operazioni hanno esito negativo, cioè il nodo non possiede il record in locale e il numero di nodi restituiti da DHT è inferiore di k/2, procede all'adempimento della richiesta. Qualora un nodo "cattivo" proceda alla creazione di un account già esistente, i nodi destinatari non soddisferanno tale richiesta poiché il numero di nodi restituiti da DHT sarà maggiore di k/2.

Questo modo di procedere in alcuni casi limite potrebbe avvantaggiare un nodo cattivo:

- un nodo cade prima di effettuare lo "STORE";
- i nodi individuati durante la creazione dell'account non sono tutti differenti: potrebbe essere che utilizzando due chiavi differenti individuo lo stesso nodo. Questo è indice della scarsa presenza di nodi nella rete;

I casi precedentemente elencati, essendo casi limite, possono non essere considerati ai fini del funzionamento del plugin.Ora proseguiamo con la descrizione dell'operazione di "Login" nel prossimo paragrafo.

3.3 Login

L'utente, che effettua il Login, compie implicitamente le operazioni in seguito elencate:

1. Il modulo Login effettua la ricerca dell'username digitato da tastiera dall'utente: se DHT non restituisce alcun record il login fallisce, altrimenti prosegue con il punto successivo;
2. Viene decriptata la chiave privata con la chiave simmetrica generata dalla password dell'utente;

3. Si effettua un "*Challenge*" in locale: si cripta un messaggio casuale con la chiave privata e lo si decripta poi con la chiave pubblica presente nel record, se si riottiene il precedente messaggio, si può proseguire con il punto successivo, altrimenti il login è fallito;

4. Viene assemblato un record con gli stessi campi di quello del punto *1* eccetto l'indirizzo IP al quale contattare l'utente e il timestamp. Questi due campi vengono aggiornati rispettivamente con l'indirizzo IP della macchina attuale e con l'ora corrente;

5. Vengono criptati con la chiave privata tutti campi che compongono il record (Firma);

6. Viene inviata la richiesta ai nodi trovati nel punto *1*;

I nodi che ricevono una richiesta di questo tipo si comportano eseguendo le seguenti operazioni:

1. Verificano se in locale hanno il record che dovrebbe essere aggiornato: se è presente si procede al seguente punto, altrimenti viene scartata la richiesta;

2. Viene controllato la firma utilizzando la chiave pubblica del record in locale: se il controllo va a buon fine il record viene aggiornato, altrimenti la richiesta viene scartata;

Analizzato il comportamento dei nodi durante l'operazione di Login si procede all'eliminazione di un account nel prossimo paragrafo.

3.4 Cancellazione di un account

In questo paragrafo esamineremo il comportamento di un nodo in seguito a una richiesta di cancellazione di un determinato account. In particolare

valuteremo l'atteggiamento del nodo che invia le richieste e quello che ne riceve una richiesta di questo tipo.
Ora verrà analizzato il comportamento del nodo che invia le richieste di cancellazione, elencando i principali punti:

1. Verifico se l'utente è loggato: se è connesso prosegue con il punto successivo, altrimenti termino l'esecuzione;
2. Reperisco per la rete il *recordId* relativo all'utente connesso, effettuando un "FIND" su DHT dell'username;
3. Effettuo la firma dei campi: recordId, username, chiave pubblica, chiave privata criptata e timestamp;
4. Assemblo la richiesta e la invio ai nodi individuati nel punto *2*;

Consideriamo ora l'atteggiamento da parte del nodo che riceve una richiesta di cancellazione, sintetizzandolo nei seguenti punti:

1. Se il nodo non contiene il record relativo alla richiesta di cancellazione, scarta tale richiesta e procede al punto *5*, altrimenti prosegue con il punto successivo;
2. Se il campo *"isStillValid"* è settato a false, procede al punto *5*, altrimenti con il punto successivo;
3. Controllo la firma utilizzando la chiave pubblica presente nel record in locale, se la verifica ha esito positivo prosegue con il punto successivo, altrimenti passa al punto *5*;
4. Setta *"isStillValid = false"* e lo *"status = 2"*;
5. Preleva dalla coda un'altra richiesta da soddisfare;

La cancellazione effettiva avviene alla fine di una sessione di refresh, infatti nel corso del refresh stesso si pone "isStillValid" al valore di false. Il motivo di tale decisione impedisce il verificarsi del seguente problema: in seguito a una cancellazione potrebbero arrivare un numero di refresh request maggiore di k/2, le quali obbligherebbero il nodo a reinserire il record. Esiste inoltre un meccanismo di propagazione della cancellazione, qualora il record permanga nella rete dopo una cancellazione.

Ora passeremo all'analisi di uno dei principali thread del plugin Login, che tra i tanti ruoli cancella i record aventi isStillvalid a false.

3.5 RefreshHandler

Questo thread si occupa, come si può intuire, delle sessioni di refresh. In particolare all'inizio di ogni refresh esso accede in mutua esclusione al database (DB), contente i record affidati al nodo, e ne effettua una copia. Tale copia sarà in seguito affidata a un insieme di thread, chiamati *OutcomingRefreshRequestThread*, che calcoleranno per ogni record i nodi bersaglio e li invieranno le refresh request. Esse saranno ricevute dai nodi destinatari e affidate all'insieme di thread denominati *IncomingRefreshRequestThread*. Una volta terminata la sessione di refresh il RefreshHandler cancella dal database tutti i record aventi lo "status" posto al valore di 2.

Questo thread ha anche il compito di aprire e chiudere ufficialmente il refresh, impostando la variabile booleana "*refreshON*" a true per dichiararlo aperto e a false per dichiararlo a chiuso. Inoltre quando il RefreshHandler accede in mutua esclusione al database il TransactionRequestHandler non può soddisfare alcuna richiesta, sia essa di inserimento, di cancellazione oppure di aggiornamento.

3.6 Servizi offerti dal Login

Il plugin Login offre due principali servizi: "*Ricerca di un utente*" e "*Challenge*[2]". Il primo consiste nel trovare le informazioni relative all'utente, chiave pubblica e indirizzo IP al quale contattare l'utente, mentre il secondo servizio serve a verificare la reale identità di un determinato utente. Questi servizi saranno approfonditi nei successivi due sottopragrafi.

3.6.1 Ricerca di un utente

La ricerca di un determinato utente consiste nell'effettuare un "FIND" su DHT dell'username in modo da conoscere tutti i potenziali nodi in possesso del record. In seguito selezionare i nodi da contattare in modo da escludere eventuali nodi "cattivi".

La soluzione ideale per l'eliminazione dei nodi "malvagi" consiste nel compiere l'intersezione tra i nodi individuati con il "FIND" e quelli trovati con il "Reverse Look Up al volo". Tale soluzione risulterebbe troppo dispendiosa dal momento che essa prevede $(k + 1)$ richiesta a DHT (k per il "Reverse Look Up al volo" e una per il FIND su DHT).

La soluzione adottata risolve il problema della selezione dei nodi da contattare con una sola chiamata a DHT. Ora saranno elencati i principali punti di tale soluzione:

[2]Cfr. http://en.wikipedia.org/wiki/Challenge-response_authentication

Algorithm 2 Selezione dei nodi da interrogare

1. Eseguo il "FIND" su DHT dell'username cercato, ottenendo una lista L di potenziali nodi in possesso del record;
2. Se cardinalità(L) < k/2 comunico all'utente un opportuno messaggio e termino l'esecuzione;
3. Se cardinalità(L) < lowerBound contatto tutti i nodi della lista e termino l'esecuzione;
4. Calcolo in locale gli hash delle k chiavi del record da cercare;
5. Eseguo le successive operazioni per ognuno degli hash precedentemente calcolati:
   (a) Ordino la lista L1 secondo la distanza decrescente dall'hash selezionato;
   (b) Individuo e seleziono i tre nodi più vicini all'hash preso in considerazione;
6. Contatto i nodi selezionati dalla lista L1;

Questo soluzione sostituisce le operazione del Reverse Look Up al volo, eseguendo le precedenti operazioni e non effettuando le k chiamate a DHT. Quest'ultima soluzione è più imprecisa rispetto alla soluzione ideale. La decisione adottata determina sempre i tre nodi più vicini ad un hash, aumentando la probabilità di selezionare un nodo malvagio.

La scelta di utilizzare la seconda soluzione serve per evidenziare l'importanza della prestazioni. Infatti il numero di chiamate a DHT, operazione estremamente dispendiosa, è assai minore rispetto all'opzione con il Reverse Look Up al volo (con la prima soluzione ci sono $(k + 1)$ chiamate mentre con la seconda soluzione si effettua una sola chiamata).

Ora focalizzeremo l'attenzione sull'altro servizio offerto da questo plugin, molto importante per la verifica dell'identità dell'utente connesso.

3.6.2 Challenge

Questo servizio sarà utilizzato da altri plugin per fornire un canale sicuro, in modo da rendere possibile lo scambio di messaggi criptati, e per l'autenticazione di un determinato utente.

In letteratura esistono dei protocolli di autenticazioni, il principale è denominato CHAP[3] (*Challenge Handshake Authentication Protocol*). Tale protocollo è riassumibile nei successivi punti:

1. Il client invia il proprio identificativo utente ed il server risponde con una domanda di "sfida" (challenge), costituita da un numero semicasuale;
2. Il client esegue l'hash (può essere un *MD5*[4]) del challenge assieme alla sua password e lo rinvia;
3. Il server, che conosce la password, è in grado di eseguire lo stesso calcolo e quindi comparare i due valori verificando la correttezza del valore ricevuto;

Inoltre il CHAP offre una protezione contro gli *attacchi replay*[5] grazie all'uso di un identificatore e di un challenge variabili. L'unico aspetto negativo di questo protocollo è che la password dev'essere nota a entrambi i sistemi (client e server). Tale aspetto porterebbe alla propagazione della password dell'utente nella rete, perché il protocollo preso in considerazione non prevede l'utilizzo di server distribuiti. Dato che ogni nodo svolge entrambi i ruoli (client - server) il difetto di conoscere la password, potrebbe portare alla diffusione di un dato sensibile nell'intera rete.

[3]Cfr. http://en.wikipedia.org/wiki/Challenge-handshake_authentication_protocol
[4]Cfr.http://en.wikipedia.org/wiki/MD5
[5]Cfr. http://en.wikipedia.org/wiki/Replay_attack

La scelta si è riversata sull'utilizzo di un protocollo non ufficiale. Diversamente dal CHAP, esso sfrutta la crittografia asimmetrica, dato che ogni utente ha una coppia di chiavi (pubblica e privata). Il protocollo sarà ora esposto nei suoi principali punti:

1. Il client (A) assembla un pacchetto avente la chiave pubblica di A e un numero casuale;
2. A cripta il pacchetto con la chiave pubblica del server B e lo invia a destinazione;
3. B decripta con la propria chiave privata;
4. B assembla un pacchetto avente il numero casuale presente nel pacchetto decriptato e una chiave simmetrica;
5. B cripta il pacchetto con la chiave pubblica di A presente nella e lo invia a destinazione;
6. A decripta tale pacchetto verifica se il numero casuale è lo stesso e utilizzerà la chiave simmetrica per fornire un canale sicuro;

Il server B è la postazione alla quale l'utente è connesso, mentre il client A è il nodo dove la richiesta di Challenge è stata effettuata.

Capitolo 4

Conclusioni

Il lavoro svolto in questa tesi rappresenta solo una parte di ciò che è stato realizzato in realtà. Sono stati illustrati i principali argomenti, tralasciando alcuni dettagli implementativi.

Durante la progettazione sono sorti molti problemi soprattutto dal punto di vista concettuale. La difficoltà maggiore è stata quella di prevedere tutti i possibili casi anomali. In particolare si è cercato di immaginare quali danni poteva arrecare un nodo avente un comportamento nocivo agli altri nodi. Precisamente un nodo malvagio potrebbe modificare i record dei quali non è proprietario in due modi: utilizzando le richieste di aggiornamento/cancellazione oppure attraverso le refresh request. Per impedire il primo atteggiamento anomalo si utilizza la "*Firma*", mentre per il secondo il "*Reverse Look Up*". Esso è presente in due forme, a secondo che un nodo possegga o meno il record in locale. In particolare se a un determinato nodo arrivano delle refresh request e non ha il record in locale effettua il "*Reverse Look Up al volo*", mentre se quel determinato record è già presente nel database utilizza la "*Tabella di Look Up*". Attraverso questi due meccanismi non si ha l'assoluta certezza che le refresh request proveniente da nodi malvagi siano scartate. Tali meccanismi garantiscono con una probabilità,

prossima a uno, che un nodo memorizza il record giusto. Gli argomenti appena citati sono, ulteriormente, approfonditi nel documento in PDF presente nella wiki di PariPari e nella tesi intitolata "*PariDNS 2009*" del laureando Mattia Rossi.

Si è cercato, peraltro, di rendere la struttura del plugin più modulare possibile, in modo stabilire nei minimi dettagli il compito di ogni classe.

Personalmente, mi ritengo molto soddisfatto dei risultati raggiunti. Inoltre grazie a PariPari ho appurato la mia conoscenza di Java e ho ulteriormente approfondito argomenti trattati duranti i corsi.

Bibliografia

[1] **Michele Bonazza:** *PariCore, 2009*

[2] **Jacopo Buriollo:** *PariPari DBMS: (Re-)inizializzazione e churn, 2008*

[3] **Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz:** *Handling Churn in a DHT*, www.srhea.net/papers/bamboo-usenix.pdf

Elenco delle figure

1.1 Struttura della DHT (CHORD) 8

2.1 Inserimento a cavallo di un Refresh 14
2.2 Inserimento a cavallo di un Refresh con i “Tre stati” 15
2.3 Esempio di Reverse Look Up 18

3.1 Struttura generale del plugin Login 22
3.2 Funzionamento del package Network 24