UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria dell'Informazione

Corso di laurea in Ingegneria Informatica

TESI DI LAUREA

PARIDNS 2009

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

CORRELATORE: Ing. Paolo Bertasi

LAUREANDO: Mattia Rossi

A.A. 2009-2010

Indice

Sommario

Questa tesi tratta la progettazione e la realizzazione del Name Server distribuito PariDNS, sviluppato all'interno del progetto PariPari per la gestione del dominio *paripari.it*.
Il primo capitolo è dedicato alla descrizione ad alto livello del progetto e in particolare alla presentazione dell'architettura strutturata della rete peer-to-peer PariPari, che dal progetto trae il suo nome.
Nel secondo capitolo, dopo un introduzione al Domain Name System, è affrontato il problema della distribuzione del name server PariDNS all'interno della rete PariPari, e sono presentati rispettivamente l'algoritmo sviluppato per la risoluzione del problema e la sua estensione al fine di rendere il server sicuro.
Il terzo capitolo descrive singolarmente il funzionamento dei servizi di inserimento, aggiornamento e cancellazione di un dominio, e il servizio di risoluzione, offerti dal name server.
Infine il quarto capitolo presenta l'architettura fortemente modulare del plugin DNS, le cui istanze attive all'interno della rete PariPari concorrono alla realizzazione del name server distribuito PariDNS.

Capitolo 1

PariPari

PariPari è una nuova rete *peer-to-peer* in fase di sviluppo presso il Dipartimento di Ingegneria dell'Informazione dell'Università di Padova.

In controtendenza rispetto alle più diffuse reti peer-to-peer PariPari ambisce ad essere una rete multifunzionale: PariPari offre infatti la possibilità di fruire dei più tradizionali servizi peer-to-peer (file sharing, distributed storage, VoIP...), ma al tempo stesso esporta su un'architettura completamente *serverless* molti altri servizi tipicamente *server-based* (email hosting, IRC chat, web hosting, DNS...).

PariPari è una rete peer-to-peer completamente decentralizzata: la sua struttura non prevede quindi la presenza di server e ogni macchina al suo interno (*peer*) agisce contemporaneamente sia da client che da server. Questa caratteristica rende PariPari una rete solida e difficilmente sabotabile dal momento che la caduta, o semplice disconnesione, di alcuni peer difficilmente si manifesterà in un qualche disservizio per gli altri.

Rispetto ad altre reti completamente decentralizzate, quali Gnutella, la cui struttura evolve sostanzialmente in modo casuale, la rete PariPari sfrutta un algoritmo di organizzazione topologica a *tabelle di hash distribuite (DHT)* che permette alla rete di adeguarsi ad un grafo dalla particolare struttura. Questa caratteristica rende PariPari una **rete strutturata**.
Se da un lato l'applicazione di questo algoritmo introduce una maggior complessità in fase di costruzione e manutenzione della rete, rispetto alle più comuni reti non strutturate, dall'altro permette una localizzazione delle risorse affidabile ed efficiente, laddove una rete non strutturata non potrebbe che limitarsi ad una ricerca casuale e non affidabile.
In particolare questa architettura non pone vincoli alla scalabilità di PariPari poichè l'efficienza nelle operazioni di ricerca non degrada velocemente all'aumentare delle dimensioni della rete.

PariPari adotta un approccio a *plugin*. Ogni peer è costituito da un nucleo centrale, il *core*, attorno al quale ruota una "galassia" di plugin, o moduli, ognuno dei quali offre un particolare servizio.
All'interno di questa galassia i plugin non operano tutti allo stesso livello: alcuni di essi, i *gestori delle risorse*, operano infatti ad un livello più basso al fine di garantire la gestione delle risorse locali alla macchina (*Storage*), l'accesso alla rete (*Connectivity*), la reperibilità dei nodi sulla rete (*DHT*) e l'economia della stessa (*Credits*).
I rimanenti plugin si interfacciano invece principalmente con l'utente e sfruttano i servizi messi a disposizione dai *gestori delle risorse* per operare ad un maggior livello di astrazione. Questo non significa necessariamente che questi plugin non possano offrire servizi agli altri plugin di PariPari: come i *gestori delle risorse*, ogni modulo può infatti prevedere delle *API*[1] attraverso le quali rendere disponibili i propri servizi. La comunicazione tra moduli non è tuttavia diretta: è infatti compito del core arbitrare le comunicazioni tra moduli della stessa macchina.

La struttura fortemente modulare del client PariPari presenta molteplici vantaggi. Innanzitutto rende PariPari un piattaforma facilmente espandibile, dal momento che un qualsiasi programmatore potrà sviluppare in modo agile e veloce un nuovo plugin concentrando la sua attenzione esclusivamente sul servizio che mira ad offrire, piuttosto che sulla gestione delle risorse della macchina o della rete, inoltre garantisce all'utente finale una maggior sicurezza dal momento che la presenza dei *gestori delle risorse* impedisce ad un plugin malevolo di accedere alle risorse della macchina senza l'esplicito consenso dell'utente. Infine, grazie anche alla tecnologia Java Web Start, che consente all'utente di avviare PariPari direttamente dal browser senza la necessità di un'installazione, la modularità permette all'utente di scaricare e avviare i soli plugin di cui realmente ha bisogno.

La struttura modulare e rivolta all'espandibilità del client PariPari unitamente alla natura serverless della sua rete sono alla base dell'obiettivo del progetto PariPari: la realizzazione di "*...una macchina virtuale capace di provvedere a tutti i bisogni dell'utente di internet mantenendosi dipendente solo dalla comunità di nodi da cui è formata.*"[1]

1.1 Hash Table Distribuita

Questa sezione vuole essere un'introduzione all'architettura *DHT (Distributed Hash Table)*: l'autore non pretende infatti di affrontare in modo esaustivo l'argomento, ma vuole fornire delle nozioni base necessarie alla comprensione dei successivi capitoli.

[1] Application Program Interface.

Sebbene vi siano diverse implementazioni di DHT, di cui le reti Chord [11], CAN [5], Pastry [6], Tapestry [7] e la più recente Kademlia [8] sono degli esempi, ognuna è basata su un algoritmo che prende il nome di *hashing coerente (consistent hashing)* e permette di distribuire uniformemente un insieme di "oggetti" all'interno di un vasto spazio di identificativi o indirizzi.
Indicata con b la dimesione dello spazio degli indirizzi, un algoritmo di hashing coerente riceve in ingresso una stringa di lunghezza arbitraria (detta anche chiave) e produce in uscita una stringa di b bit. Ai fini della trattazione è importante sottolineare che ad uno stesso ingresso corrisponderà sempre la stessa uscita.

Nell'architettura DHT l'algoritmo di hashing coerente è utilizzato per assegnare ad ogni nodo e ad ogni risorsa della rete un indirizzo a b bit, chiamato *ID* per i nodi, *hash* per le risorse.
All'interno di questo spazio è formalizzata una metrica, che varia a seconda dell'implementazione (Kademlia, ad esempio, adotta la metrica XOR), e permette la ripartizione dello spazio stesso. Ogni nodo può quindi suddividere virtualmente la rete in più settori: il settore contenente i 2^{b-1} indirizzi nell'altra metà della rete, il settore contenente i 2^{b-2} indirizzi nella sua stessa metà ma nell'altro quarto, il settore contenente i 2^{b-3} indirizzi nel suo stesso quarto ma nell'altro ottavo, e così via.
Ogni nodo deve poi memorizzare k nodi tra quelli presenti nell'altra metà della rete, k tra quelli nella sua stessa metà ma nell'altro quarto, k tra quelli nel suo stesso quarto ma nell'altro ottavo, e così via. Teoricamente un valore di k pari a 1 sarebbe sufficiente, ma per una maggiore affidabilità è preferibile utilizzare un valore più elevato (in genere si ha $5 \leq k \leq 10$). Ovviamente i settori più piccoli potrebbero contenere meno di k nodi, quindi per ognuno di questi settori il nodo provvederà a memorizzare tutti i nodi disponibili.

É importante notare che la politica utilizzata per scegliere i contatti del singolo nodo fa sì che questo abbia una conoscenza della rete che è tanto più approfondita quanto minore è la distanza dal suo *ID*: ad esempio, mentre il nodo conoscerà molti dei nodi nella sua stessa metà, ne conoscerà solo k tra tutti quelli appartenenti alla metà opposta.

Dal momento che nodi e risorse sono mappati nello stesso spazio degli indirizzi, è possibile associare ogni risorsa r al nodo *v(r),* che rappresenta il nodo con l'*ID* più vicino all'*hash* di r nella metrica adottata.
Nel momento in cui un nodo in possesso della risorsa r voglia renderla disponibile alla rete dovrà quindi innanzitutto calcolare l'*hash* di r, ricercare nella rete il nodo *v(r)*, e infine fornire a questo le proprie coordinate. Questa operazione è nota come primitiva $\text{STORE}(r)$.
L'operazione attraverso la quale recuperare la risorsa r è analoga alla precedente: è infatti necessario calcolare nuovamente l'*hash* della risorsa r, cercare il nodo *v(r)*, e infine

farsi restituire le coordinate del nodo che materialmente possiede la risorsa. Questa operazione è nota come primitiva FIND(r).
Notiamo che associare la risorsa r al nodo *v(r)* non significa affidargli materialmente la risorsa, ma semplicemente fornirgli le coordinate del nodo presso il quale è possibile reperirla.

Affrontiamo ora nel dettaglio la ricerca del nodo *v(r)* da parte del nodo u, tenendo presente che questa operazione è necessaria quanto nella primitiva STORE(r), quanto nella primitiva FIND(r).
Il nodo u invia una richiesta per la risorsa r al nodo *u'*, che rappresenta il nodo più vicino all'*hash* di r tra i contatti di u. Nel caso peggiore il nodo *u'* si troverà nell'altra metà della rete rispetto a u, ma comunque nella stessa metà di rete dove si trova il nodo *v(r)*. *u'* inoltra allora la richiesta al nodo *u''*, che rappresenta il nodo più vicino all'*hash* di r tra i contatti di *u'*, e si trova sicuramente nello stesso quarto di rete di *v(r)*. La richiesta continua ad essere inoltrata fino a quando raggiunge il nodo u^i il quale tra i suoi contatti non ha un nodo con *ID* più vicino del suo all'*hash* di r: si ha allora che u^i è proprio il nodo *v(r)*.
Dal momento che ad ogni passo lo spazio di ricerca si dimezza possiamo affermare che la ricerca si concluderà con grande probabilità in un numero di passi i logaritmico rispetto alle dimensioni della rete.

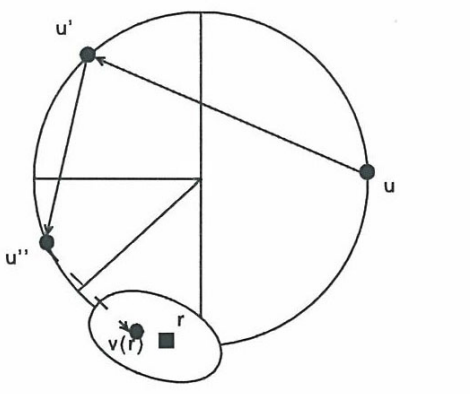

Figura 1.1: Struttura dell'algoritmo di ricerca in DHT.

Allo stato attuale la DHT implementata all'interno di PariPari risulta essere una versione modificata di Kademlia.

Capitolo 2

PariDNS

PariDNS è un *Name Server* distribuito, autoritativo per il dominio *paripari.it*.

PariDNS è progettato per offrire all'utente, e a qualsiasi plugin interno alla rete PariPari, la possibilità di registrare al volo e aggiornare in modo dinamico e veloce un quasiasi nome nel dominio *paripari.it*. PariDNS offre quindi un servizio di *DNS hosting*.

La possibilità di registrare e aggiornare dinamicamente un dominio è alla base del corretto funzionamento di plugin quali *Web Server* e *IRC (Internet Relay Chat)*, che mirano ad offrire gli omonimi servizi senza tuttavia la presenza di un server centralizzato.
Consideriamo ad esempio il funzionamento del plugin *Web Server*. In genere alla creazione di un sito web il plugin affida il sito ad un *cluster* di peer sui quali è attivo il medesimo plugin: grazie a questo espediente il sito risulta disponibile su diverse macchine, ognuna delle quali opera da web server, rendendo possibile un bilanciamento del carico e garantendo la persitenza del sito web anche a fronte della caduta di alcuni peer del cluster. Affinchè un utente possa richiedere una qualunque pagina del sito web è tuttavia necessario che il plugin *Web Server* associ un nome di dominio al cluster, e registri l'associazione *<nome di dominio-nodi del cluster>* presso un qualche Name Server. In genere questa operazione non è affatto immediata. PariDNS permette invece la registrazione al volo di un qualunque dominio interno a *paripari.it*.
É poi fondamentale notare che la composizione del cluster è inevitabilmente destinata a variare nel tempo, dal momento che la disconnesione di un qualche peer renderà necessaria la sua sostituzione. Emerge quindi una necessità ancora più stringente: poter aggiornare dinamicamente e in ogni momento l'associazione *<nome di dominio-nodi del cluster>* al fine di garantire la raggiungilità del sito web. PariDNS offre questa possibilità.

PariDNS è un name server completamente distribuito e alla sua base è il plugin *DNS*.
Come risulterà evidente al termine della sezione 2.1 un name server è riducibile ad un

database contenente tutte le associazioni inerenti al dominio per il quale il server è autoritativo. PariDNS distribuisce il suo database tra le numerose istanze del plugin *DNS* attive all'interno della rete PariPari: in questo modo ogni istanza del plugin possiede una piccola porzione del database e la totalità delle istanze concorre alla realizzazione di un unico grande database distribuito e quindi alla realizzazione del name server stesso. La natura completamente distribuita del name server risulta trasparente ai suoi utilizzatori: ogni istanza del plugin DNS rende diponibili tutti i servizi del name server e fornisce quindi l'illusione di una comunicazione diretta con questo. In realtà il plugin non ha alle spalle alcun server, ma solo una vasta rete di altri nodi PariPari su ognuno dei quali è attivo lo stesso plugin, e coi quali può rendersi necessario interagire per soddisfare le richieste dell'utilizzatore.

I servizi diponibili sono la risoluzione di un qualunque nome di dominio, la registrazione di un nome di dominio interno a *paripari.it*, il suo aggiornamento e cancellazione. Il plugin è in ascolto sulla porta 53: per ottenere la risoluzione di un nome è quindi possibile interrogare PariDNS utilizzando il protocollo DNS standard. Il plugin fornisce inoltre delle API attraverso le quali è possibile fruire di tutti i servizi offerti dal name server: dalla risoluzione, alla cancellazione di un dominio precedentemente registrato. I servizi diponibili attraverso le API sono resi disponibili all'utente attraverso la Console del plugin.

In questo capitolo, dopo un'introduzione al Domain Name System nella sezione 2.1, verrà affrontato il principale problema incontrato nella realizzazione di PariDNS: la distribuzione del suo database all'interno della rete PariPari. In particolare nella sezione 2.2 sarà presentata la soluzione attualmente presente in letteratura e utilizzata come modello di riferimento, ma al tempo stesso saranno esposte le ragioni dell'impossibilità di una sua diretta applicazione in PariPari; nella sezione 2.3 sarà invece introdotta una versione base dell'algoritmo sviluppato per la risoluzione del problema, e infine nella sezione 2.4 si assisterà ad una leggera complicazione dell'algoritmo dettata dalla possibile presenza nella rete di nodi malevoli.

2.1 Domain Name System

Ad ogni host della rete è tipicamente assegnato un nome (*hostname*) che permette di identificare in modo univoco l'host ma non fornisce alcuna informazione sulla sua dislocazione all'interno della rete. Il sistema dei nomi di dominio (*DNS*, *Domain Name System*) ha il compito di tradurre gli hostname in indirizzi IP: questa operazione prende il nome di *risoluzione*.

La rete non usa il sistema DNS da sempre: nelle sue prime fasi di operatività, quando in Internet vi erano soltanto poche centinaia di host, esiteva una tabella di corrispondenze fra nomi e indirizzi, contenuta in un file chiamato *host.txt* e gestita da un'autorità centrale, il NIC (Network Information Center). L'aggiunta di un host a Internet prevedeva quindi l'inserimento manuale nella tabella di una coppia *hostname/indirizzo dell'host*, e la tabella modificata era inviata per posta elettronica a tutti gli host, i quali potevano aggiornare la loro tabella locale.
La risoluzione dei nomi era quindi realizzata da una semplice procedura che cercava il nome di un host nella copia locale della tabella, restituendo l'indirizzo corrispondente.

Il sistema dei nomi di dominio venne introdotto solo intorno alla metà degli anni Ottanta, a fronte della crescita del numero di host della rete Internet, utilizza uno *spazio dei nomi gerarchico* e suddivide in più parti distinte la "tabella" delle corripondenze per poi distribuirla nella rete.
In particolare queste sottotabelle sono messe a disposizione da server per i nomi (*name server*) che possono essere interrogati attraverso la rete stessa.

2.1.1 Gerarchia di domini

Uno *spazio dei nomi* definisce l'insieme dei nomi possibili. Uno spazio dei nomi può essere *gerarchico* (e i nomi dei file nei sistemi Unix ne sono un esempio) oppure *non gerarchico* (*flat*, uno spazio nel quale i nomi non sono decomponibili in più parti).

Il DNS utilizza uno spazio dei nomi gerarchico, ma diversamente dai nomi nel sistema operativo Unix, che vengono elaborati da sinistra verso destra e le cui componenti sono separate da sbarre di divisione, i nomi del DNS, o *nomi di dominio*, vengono elaborati da destra verso sinistra e usano il punto come separatore. Un esempio di nome di dominio per un host è *cicada.cs.princeton.edu*.

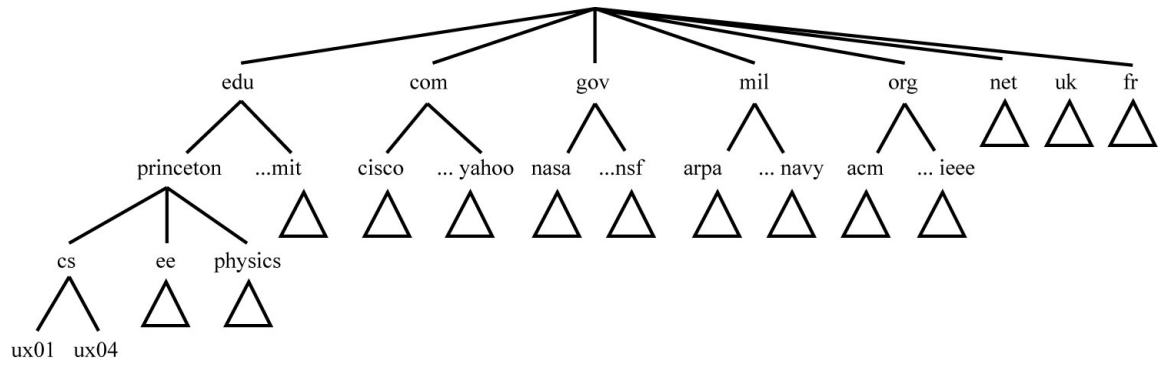

Figura 2.1: Esempio di una gerarchia di domini.

La gerarchia del DNS può essere visualizzata mediante un albero, dove ogni nodo dell'albero corrisponde ad un dominio, e le foglie rappresentano gli host a cui si assegna un nome, come esemplificato in Figura 2.1: in particolare ogni dominio è denominato

dal percorso compreso tra esso e la radice (non denominata), e il numero di componenti del nome di dominio indica il livello del dominio stesso.
Notiamo che la gerarchia al primo livello non è molto estesa: ci sono domini per ciascun Paese, più i "sei grandi" domini, o "big six"(*.edu*, *.com*, *.gov*, *.mil*, *.org* e *.net*).

2.1.2 Server per i nomi

La gerarchia dei domini è divisa in *zone* non sovrapposte, e ad ognuna è associato un server per i nomi (*name server*) il quale contiene tutte le informazioni sulla zona stessa. In figura 2.2 è presentata una possibile suddivisione della gerarchia di figura 2.1.

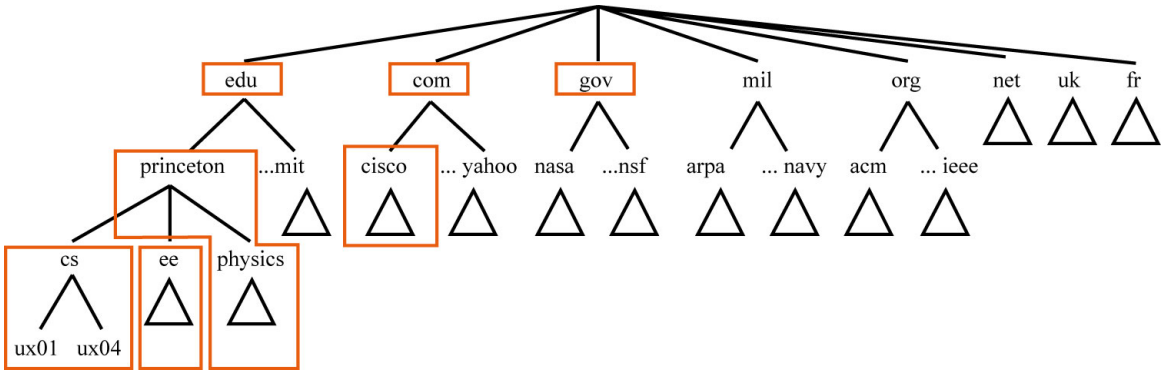

Figura 2.2: Gerarchia di domini suddivisa in zone.

Al fine di comprendere la suddivisione della gerarchia dei domini in zone è utile procedere per passi successivi.
Inizialmente è possibile considerare la gerarchia dei domini come composta da un'unica zona alla quale è associato un'unico name server (*server radice*) responsabile per i nomi di dominio dell'intero albero. Date le dimensioni dell'albero e quindi l'elevato numero di nomi da gestire il server radice delega la gestione di ogni sottoalbero avente per radice un dominio di primo livello (*TLD, Top Level Domain*) ad un differente name server. Questa operazione porta alla divisione della gerarchia in tante zone quanti sono i domini di primo livello.
Consideriamo ora a titolo di esempio la zona *.edu*. Al momento la zona *.edu* rappresenta l'intero sottoalbero della gerarchia di radice il dominio *.edu* ed è quindi impensabile che l'intero insieme dei nomi di dominio all'interno del sottoalbero possa essere gestito da un solo name server: per questo motivo il name server del dominio *.edu* delega la gestione di tutti i suoi sottodomini ai rispettivi proprietari. In particolare il name server del domino *.edu* affida ogni sottoalbero della gerarchia avente per radice un dominio di secondo livello nella forma *<domain_name>.edu* ad un differente name server: si assite quindi alla nascita di una nuova zona per ogni sottoalbero, mentre *.edu* diviene una zona a sè.
Una delle zone nate dalla precedente operazione è la zona associata al dominio *prince-*

ton.edu. La Princeton University possiede diversi dipartimenti e alcuni, come il Department of Physics, al quale è associato il dominio *physics.princeton.edu*, preferiscono non doversi occupare della gestione del proprio dominio, mentre altri, come il Department of Computer Science, al quale è affidato il dominio *cs.princeton.edu*, preferiscono gestirlo autonomamente. Al fine di permettere al Department of ComputerScience di gestire autonomamente il proprio dominio il name server di *princeton.edu* delega la gestione del sottoalbero della gerarchia di radice *cs.princeton.edu* ad un nuovo name server. Si ha quindi che una porzione della zona associata al dominio *princeton.edu* si separa e diviene una zona a sè, dotata di un proprio name server. In particolare in figura Y nascono due nuove zone: una relativa al dominio *cs.princeton.edu* e un'altra relativa al dominio *ee.princeton.edu*. Notiamo invece che il dominio *physics.princeton.edu* resta sotto la gestione del name server di *princeton.edu*.
Ora la gerarchia risulta divisa in diverse zone, ognuna amministrata in modo autonomo da un diverso server dei nomi, il quale si dice *autorevole* per la zona di sua competenza.

Ogni name server gestisce le informazioni relative ad una zona come un insieme di *resource records (record delle risorse)*. Un resource record è una corrispondenza tra un nome e un valore, o, più precisamente, un insieme di cinque valori che contiene i seguenti campi:
<Name, Value, Type, Class, TTL>
Il campi *Name* e *Value* rappresentano rispettivamente un nome di dominio e il valore a questo associato, mentre il campo *Type* specifica come debba essere interpretato il campo *Value*. Esistono principalmente due diversi tipi di record:

- A: Il campo *Value* è l'indirizzo IP dell'host specificato.
- NS: Il campo *Value* fornisce il nome di dominio di un host che sta svolgendo la funzione di server per i nomi e che sa risolvere i nomi all'interno del dominio specificato.

Il campo *Class* indica la famiglia di protocolli a cui appartiene il resource record, ma ad oggi l'unico valore di *Class* ad aver raggiunto un'ampia diffusione è quello usato da Internet, che viene indicato con IN. Infine, il campo *TTL* indica il periodo di validità del resource record. É utilizzato dai server che memorizzano in una cache i resource record provenienti da altri server: quando scade il periodo di durata *TTL*, il server rimuove il record dalla propria memoria cache.

Segue ora un esempio utile a comprendere l'utilizzo dei resource record e in particolare la suddivisione della gerarchia in zone di competenza. L'esempio fa riferimento alla Figura X e per semplicità trascura il campo *TTL* di ogni record.
Innanzitutto, il server radice (*root name server*) contiene un record di tipo NS per ogni

name server di un TLD. Inoltre contiene altrettanti record di tipo A che traducono tali nomi negli indirizzi IP corrispondenti. Ciascuna di queste coppie di record, uno di tipo NS e il corrispondente di tipo A, realizza un puntatore dal name server radice a uno dei server TLD.
<edu, a3.nstld.com, NS, IN>
<a3.nstld.com, 192.5.6.32, A, IN>
<com, a.gtld-servers.net, NS, IN>
<a.gtld-servers.net, 192.5.6.30, A, IN>
...
Il server *a3.nstld.com* è il name server associato al dominio *.edu* e contiene i record relativi ai name server responsabili per i domini di secondo livello al di sotto di *.edu*, come questi:
<princeton.edu, dns.princeton.edu, NS, IN>
<dns.princeton.edu, 128.112.129.15, A, IN>
...
In questo caso troviamo un record di tipo NS e un record di tipo A per il server dei nomi che ha la responsabilità per la porzione *princeton.edu* della gerarchia. Tale server è in grado di risolvere direttamente alcune interrogazioni (per esempio in merito a *user01.physics.princeton.edu*), mentre per altre rinvia ad un ulteriore server per i nomi (come accade, per esempio, per un'interrogazione relativa a *penguins.cs.princeton.edu*):
<user01.physics.princeton.edu, 128.112.198.35, A, IN>
<cs.princeton.edu, dns1.cs.princeton.edu, NS, IN>
<dns1.cs.princeton.edu, 128.112.136.10, A, IN>
...
Infine il name server associato al dominio di terzo livello *cs.princeton.edu*, ovvero il server *dns1.cs.princeton.edu*, contiene i record di tipo A per tutti i propri host:
<penguins.cs.princeton.edu, 128.112.155.166, A, IN>
<cicada.cs.princeton.edu, 128.112.136.35, A, IN>
<user01.cs.princeton.edu, 128.112.136.72, A, IN>
<user04.cs.princeton.edu, 128.112.192.35, A, IN>
...
Il Department of Computer Science gestisce infatti in modo autonomo il proprio dominio e rappresenta una zona a sé.

2.1.3 Traduzione dei nomi

Consideriamo ora il problema di come possa un client richiedere a tali server la traduzione di un nome di dominio. Supponiamo che il client voglia risolvere il nome *penguins.cs.princeton.edu* relativamente all'insieme di server presentati al termine della precedente sottosezione. Innanzitutto il client invia al server radice una richiesta contenente tale nome. Il server radice, non essendo in grado di trovare una corrispondenza

per l'intero nome, restituisce la migliore che trova: il record di tipo NS per il dominio *.edu*, che punta al server di TLD di nome *a3.nstld.com*. Assieme il server radice fornisce anche il corrispondente record di tipo A. Il client, non avendo ottenuto la risposta che cercava, invia la stessa richiesta al server per i nomi in esecuzione nell'host avente indirizzo IP *192.5.6.32*. Nemmeno questo server riesce a trovare una corrispondenza per il nome completo, per cui restituisce il record di tipo NS e il relativo record di tipo A per il dominio *princeton.edu*. Di nuovo, il client invia la stessa richiesta al server per i nomi in esecuzione nell'host avente indirizzo IP *128.112.129.15* e questa volta ottiene in risposta il record di tipo NS e il relativo record di tipo A per il dominio *cs.princeton.edu*. Questa volta le informazioni ottenute consentono di contattare il server in grado di risolvere completamente l'interrogazione. Viene quindi inviata per l'ultima volta la medesima richiesta all'indirizzo *128.112.136.10*, ricevendo come risposta il record di tipo A associato a *penguins.cs.princeton.edu*, dal quale il client apprende che l'indirizzo IP corrispondente è *128.112.155.166*.

Sebbene il procedimento descritto sia corretto nella maggior parte dei casi il client non esegue direttamente la ricerca. In genere il client invia la richiesta di risoluzione ad un server dei nomi locale (il name server del proprio ISP[1]) il quale a sua volta agisce nel ruolo di client ed esegue la ricerca per conto del client originario. La procedura è riprodotta in Figura 2.3.

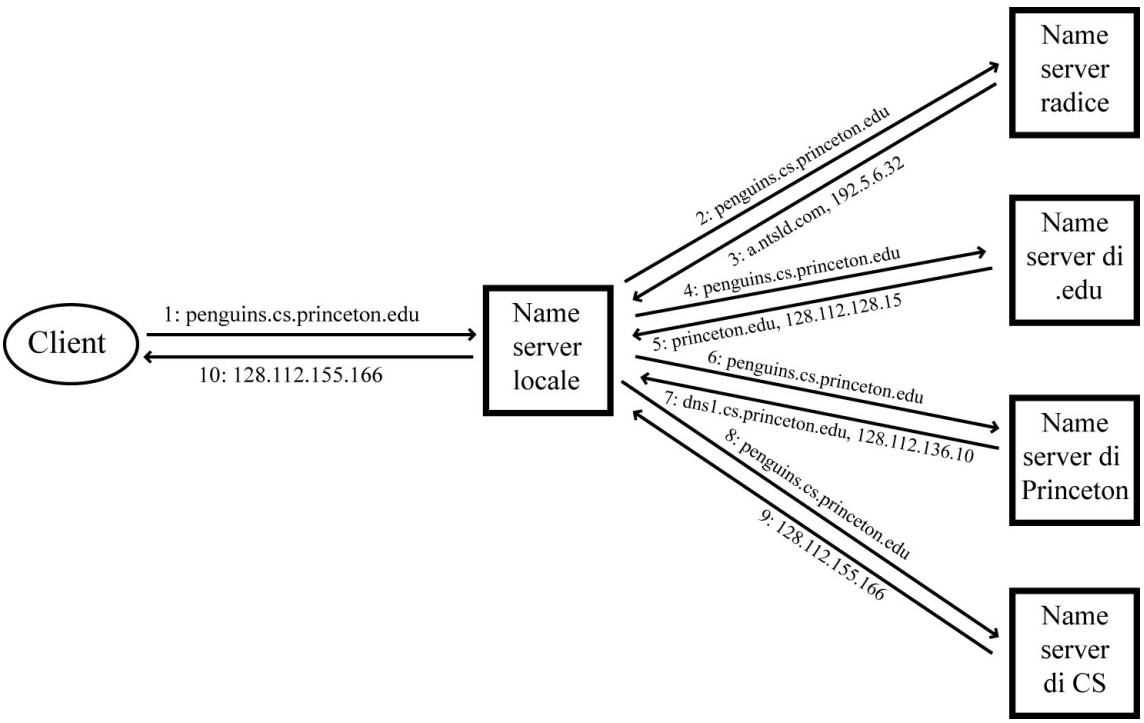

Figura 2.3: Il funzionamento concreto della traduzione dei nomi, dove i numeri da 1 a 10 mostrano la sequenza dei passi coinvolti nel procedimento.

[1]Internet Service Provider.

2.2 Il modello di riferimento

Nella gerarchia dei nomi il sottoalbero di radice il dominio *paripari.it* rappresenta un'unica grande zona e la sua gestione è affidata interamente al name server PariDNS. Questa sezione vuole essere un'introduzione al problema della sua distribuzione all'interno della rete strutturata PariPari.

La distribuzione di un name server all'interno di una rete *DHT-based* non è un problema nuovo in letteratura. Una soluzione molto semplice, ma al contempo efficace, prevede l'applicazione della funzione di *hashing coerente* utilizzata dalla rete, al campo *Name* di ogni resource record. In particolare, dato un record r (con r valore del campo *Name*) la funzione associa ad r il suo *hash* e il record r è affidato al nodo *v(r),* che rappresenta il nodo con *ID* più vicino all'*hash* di r.
La risoluzione del nome di dominio r risulta allora molto semplice: è sufficiente calcolare l'*hash* di r, individuare il nodo *v(r)*, e infine richiedere a questo il record r che permetterà di portare a termine la risoluzione.
Questa soluzione è alla base di name server distribuiti quali DDNS, sviluppato al MIT, e CoDoNS, sviluppato presso la Cornell Univesity: in particolare, all'interno del server CoDoNS, il nodo *v(r)* è denominato HOMENODE(r).

Sebbene questa soluzione appaia molto semplice nasconde in realtà una complicazione.
Supponiamo che all'istante t_1 si presenti la necessità di inserire nella rete il record r. Si calcola quindi l'*hash* di r e si ricerca all'interno della rete il nodo con *ID* più vicino all'*hash* calcolato. La ricerca restituisce il nodo A: si ha quindi *v(r)* = A e ad A è affidato il record r. Supponiamo ora che all'istante t_2, con $t_2 > t_1$, entri nella rete il nodo B, e in particolare supponiamo che la distanza tra l'*ID* di B e l'*hash* di r sia minore della distanza tra l'*ID* di A e l'*hash* di r. Si ha quindi che all'istante t_2 il nodo con *ID* più vicino all'*hash* di r è il nodo B, e si ha quindi *v(r)* = B. Ne consegue che il tentativo di reperire il record r in un qualche istante t_i, con $t_i \geq t_2$, si concluderà con un insuccesso. Il record r sarà infatti cercato nel nodo *v(r)*, cioè B, quando invece il record è nelle mani di A.
É dunque necessario che il record r sia periodicamente riposizionato all'interno della rete in modo tale da trovarsi "sempre" nel nodo *v(r)*.

L'utilizzo nelle righe precedenti della stessa notazione introdotta nella sezione 1.1 non è casuale. L'operazione di posizionamento del record r all'interno della rete è infatti molto simile all'operazione STORE(r) descritta in 1.1: tuttavia le due operazioni hanno un importante differenza. Mentre l'operazione STORE(r) affida al nodo *v(r)* solo un puntatore al nodo che materialmente possiede la risorsa r, l'operazione di posizionamento del record r descritta in questa sezione affida a *v(r)* la risorsa stessa (il record

r).
PariPari utilizza esattamente la DHT descritta in 1.1 quindi la soluzione alla distribuzione di un name server descritta in questa sezione si dimostra non direttamente applicabile all'interno della rete PariPari. É inoltre importante notare che la DHT di PariPari è realizzata come un comune plugin e quindi ogni altro modulo non può che interagire con la rete attraverso le API del modulo DHT stesso: il singolo plugin non ha un accesso diretto alla rete realizzata da DHT. Nella soluzione presentata, al contrario, il name server e la DHT vengono a coincidere.
Nonostante ciò la primitiva STORE(r) fornita dalla DHT di PariPari risolve il problema del riposizionamento del record r presente nella soluzione descritta. Vediamone ora il motivo.
Nel momento in cui un generico nodo U, in possesso della risorsa r, decida di renderla diponibile alla rete è necessario utilizzi la primiva STORE(r). Il nodo *v(r)* memorizza allora un puntatore al nodo U in modo tale che la risorsa r sia raggiungibile: tuttavia *v(r)* conserva il puntatore solo per un periodo di tempo prefissato. Affinchè la risorsa sia sempre raggiungibile il nodo U deve quindi richiamare la primitiva STORE(r) periodicamente, e in particolare prima che scada l'effetto della precedente chiamata. Ne consegue che il puntatore alla risorsa è periodicamente riposizionato sul nodo corretto.

Notiamo che teoricamente la primitiva STORE(r) permetterebbe di posizionare un generico record r in un nodo qualsiasi della rete: sarebbe infatti sufficiente che il nodo eseguisse periodicamente una chiamata alla primitivia STORE(r) per rendere il record raggiungibile in ogni istante. Tuttavia all'interno di una rete peer-to-peer l'abbandono della stessa da parte di un nodo non è in genere prevedibile e si rende quindi necessario garantire una qualche forma di replicazione del record r, tale da garantirne la persistenza. Ad esempio la soluzione per la distribuzione di un name server presentata all'inizio di questa sezione affida al nodo *v(r)* il compito di replicare il record r tra i suoi vicini. PariDNS adotta una soluzione che permette di sfruttare le potenzialità della primitiva STORE(r), ma al tempo stesso garantisce la presenza di un prefissato numero di copie del record r nella rete.

PariDNS rivisita il concetto di HOMENODE(r) presentato all'inizio di questa sezione. Al momento dell'inserimento del record r nella rete PariDNS calcola, utilizzando la funzione di *hashing coerente* e la stringa r, un'insieme di k *hash*. La disponibilità di k *hash* permette di associare al record r k differenti HOMENODE ognuno dei quali riceve una copia del record r ed esegue una chiamata alla primitiva STORE(r). Periodicamente PariDNS ricalcola l'insieme di k hash, che può differire dall'insieme calcolato all'iterazione precedente, e individua nuovamente k HOMENODE, ognuno dei quali riceve una copia del record r ed esegue una chiamata alla primitiva STORE(r). Questo procedimento, iterato fino al momento della cancellazione del record, fa sì che nella rete siano sempre presenti k copie del record r e le rende accessibili con una semplice chia-

mata alla primitiva FIND(r). Questa primitiva restituirà infatti i puntatori ai nodi che attualmente sono in possesso del record. Notiamo infine che l'utilizzo della funzione di *hashing coerente* nel calcolo degli HOMENODE permette di distribuire i record del name server in modo uniforme all'interno della rete.
La sezione seguente presenta l'algoritmo alla base di PariDNS nella sua versione base.

2.3 L'algoritmo

L'intero algoritmo si basa sull'utilizzo di particolari stringhe, denominate **chiavi**. Considerato un qualsiasi istante temporale t, e indicati rispettivamente con *date* e *hour*, la data e l'ora associate all'istante t, definiamo come "chiave associata al record r e relativa all'istante t" la stringa ottenuta dalla seguente concatenazione:

$$< \text{“}Name \text{ del record } r\text{”} + date + hour >$$

Se consideriamo ad esempio le ore 15:36:25 del 20 Agosto 2009 e il record *acg.paripari.it* la chiave corrispondente risulta essere *"acg.paripari.it 20.08.2009 15"*. Notiamo che la chiave utilizza solo le ore e tralascia invece minuti e secondi, quindi per qualsiasi istante compreso tra le ore 15:00:00 e le ore 15:59:59 del giorno 20 Agosto 2009 la chiave associata al record *acg.paripari.it* resta comunque *"acg.paripari.it 20.08.2009 15"*. Più in generale possiamo affermare che all'interno di ogni ora la chiave associata ad un determinato record r non varia.

Supponiamo ora che all'istante t il generico nodo X debba registrare all'interno del name server PariDNS il record r. Il nodo X esegue innanzitutto le seguenti k operazioni:

- assembla la chiave associata al record r ma relativa a (k-1) ore prima e richiede al modulo DHT locale il nodo più vicino all'hash della chiave;
- assembla la chiave associata al record r ma relativa a (k-2) ore prima e richiede al modulo DHT locale il nodo più vicino all'hash della chiave;
- ...
- assembla la chiave associata al record r ma relativa a (k-i) ore prima, con 2 $<$ i $< k$, e richiede al modulo DHT locale il nodo più vicino all'hash della chiave;
- ...
- assembla la chiave associata al record r ma relativa all'ora attuale (istante t) e richiede al modulo DHT locale il nodo più vicino all'hash della chiave.

Riassumendo il nodo X assembla k distinte chiavi key_i (con $0 \leq i \leq k$-1)[2], per ogni key_i richiede al modulo DHT di calcolarne il corrispondente *consistent hashing* $hash_i$

[2]La chiave key_i rappresenta la chiave relativa a i ore prima: in particolare, key_{k-1} è la chiave relativa a k-1 ore prima, mentre key_0 è la chiave relativa all'ora attuale.

e di individuare il nodo $node_i$ più vicino a questo. L'intera operazione è denominata **ricerca dei *target nodes*** *(nodi bersaglio)*.
A questo punto X invia ad ogni target node una copia del record r e ognuno di questi una volta memorizzato il record esegue una chiamata alla primitiva STORE(r). Il record r è ora registrato all'interno del name server e in particolare è presente in k copie, dove il parametro k è denominato **molteplicità** ed è un numero dispari.
Sebbene il record sia ora presente all'interno della rete è necessario garantirne la persistenza. Per questo motivo l'algoritmo prevede degli eventi periodici denominati **sessioni di refresh**, il cui obiettivo è garantire che il numero di copie del record r all'interno della rete sia pari alla molteplicità k.

2.3.1 Refresh

Le sessioni di refresh si collocano in orari ben precisi all'interno di ogni ora: nell'implementazione attuale alle ore hh:00, hh:15, hh:30 e hh:45.[3] In particolare durante una sessione di refresh ogni nodo della rete esegue le seguenti operazioni per ogni record r in suo possesso:

- ricerca i target nodes per il record r;
- invia ad ogni target node una richiesta di memorizzazione del record r: tale richiesta è denominata **refresh request**.

Durante il refresh ogni nodo riceve quindi un certo numero di refresh request, alcune per record già in suo possesso, altre per record a lui nuovi. Il singolo nodo adotta la seguente politica:

- per quanto riguarda un record già in suo possesso, lo conferma se riceve per questo un numero di refresh request maggiore di $k/2$, lo rimuove altrimenti;
- per quanto riguarda un record a lui nuovo, lo memorizza e invoca la primitiva STORE(r) se riceve per questo un numero di refresh request maggiore di $k/2$, lo ignora altrimenti.

Al fine di comprendere nel dettaglio cosa accada durante un refresh esaminiamo ora cosa accade al singolo record r nel primo refresh successivo al suo inserimento nella rete.
All'inizio del refresh il record r è memorizzato su ognuno dei k target nodes individuati al momento dell'inserimento. Ognuno di questi nodi calcola i nuovi target nodes ed invia ad ognuno una refresh request per r; notiamo che i k nodi in possesso di r dovrebbero essere concordi nella scelta dei nuovi target nodes dal momento che utilizzano ognuno lo

[3] I nodi della rete PariPari sono tra loro sincronizzati grazie al modulo NTP.

stesso insieme di chiavi[4]. Ogni nuovo target node riceve k refresh request per il record r e quindi lo memorizza, o eventalmente lo conferma semplicemente. Un nodo già in possesso del record r ma non appartenente al gruppo dei nuovi target nodes elimina invece il record. Nella rete il record r è quindi ancora presente in k copie, ma non necessariamente negli stessi nodi su cui si trovava prima del refresh.
In sostanza i k nodi che conservavano r prima del refresh hanno inconsapevolemente eletto un nuovo gruppo di k nodi col compito di memorizzare r fino al refresh successivo, quando avverà una nuova elezione. In particolare è importante notare che i nodi che hanno partecipato all'elezione non hanno dovuto comunicare tra loro per accordarsi sui nuovi target nodes, e questo è stato possibile grazie alle k chiavi.

L'importanza del refresh risulta evidente se riconsideriamo l'esempio precedente supponendo che prima del refresh alcuni dei k nodi in possesso di r abbiano abbandonato la rete. All'inizio del refresh i nodi ancora in possesso del record r ignorano che il numero di copie di r è sceso al di sotto del valore k, tuttavia, se il numero di questi nodi è maggiore di $k/2$ il refresh si conclude con l'elezione di k nuovi target nodes, ognuno dei quali conserva una copia del record. Si ha quindi che la molteplicità di r è ripristinata: le copie di r presenti nella rete sono nuovamente k.

É necessario precisare il motivo per il quale la politica del nodo prevede la memorizzazione o conferma del record r solo a fronte della ricezione di un numero di refresh request maggiore di $k/2$. Le motivazioni sono due:

1. se i nuovi target nodes potessero memorizzare o confermare r solo a fronte della ricezione di k refresh request, l'abbandono della rete, prima del refresh, da parte di uno solo dei nodi "elettori" determinerebbe la perdita del record;

2. anche se molto raramente, può accadere che i nodi elettori non siano concordi nell'elezione di uno o più target nodes. Supponiamo che il nodo elettore A individui come nodo più vicino all'$hash_i$ il nodo C. Supponiamo poi che un'istante dopo entri nella rete il nodo D, più vicino rispetto a C all'$hash_i$: se ora il nodo elettore B ricerca il nodo più vicino all'$hash_i$ vorrà eleggere D e non C. L'obbligo di ricevere un numero di refresh request superiore a $k/2$ per poter memorizzare il record rende impossibile che siano eletti sia il nodo C che il nodo D.

Notiamo infine che ricevere un numero di refresh request maggiore di $k/2$ si traduce, data la natura dispari di k, nel riceverne almeno $\lceil k/2 \rceil$.

Osservazione importante sui target nodes

Consideriamo l'insieme delle k chiavi associate al record r, e notiamo che il contenuto di tale insieme varia, da un refresh al successivo, per un elemento al più. In particolare

[4] I nodi calcolano le chiavi associate ad r alla stessa ora (l'ora del refresh) quindi ottengono lo stesso insieme di chiavi.

l'insieme delle k chiavi varia al ritmo di una chiave ogni ora. Segue ora un esempio che innanzitutto chiarisce questo aspetto e successivamente ne mette in luce le interessanti conseguenze. Nell'esempio si assume $k = 5$ e per semplicità non si utilizza la data nella costruzione delle k chiavi.

ESEMPIO_1
Consideriamo il primo refresh delle ore 12:00: le chiavi associate al record r sono *"r_12"*, *"r_11"*, *"r_10"*, *"r_09"*, *"r_08"*. Notiamo che nei refresh delle ore 12:15, 12:30 e 12:45 le chiavi saranno esattamente le stesse. Al refresh delle ore 13:00 le chiavi diventano invece *"r_13"*, *"r_12"*, *"r_11"*, *"r_10"*, *"r_09"*: la chiave *"r_08"* ha quindi lasciato il posto alla chiave *"r_13"*. Al primo refresh dell'ora successiva la chiave *"r_09"* lascerà a sua volta il posto alla chiave *"r_14"*, e così via.

Supponiano ora che la rete sia stabile: non ci siano nè nodi entranti, nè nodi uscenti. Siano A, B, C, D ed E i target nodes individuati al refresh delle ore 12:00 e associati rispettivamente alle chiavi *"r_12"*, *"r_11"*, *"r_10"*, *"r_09"*, *"r_08"*. Alla fine del refresh corrente ognuno dei nodi elencati presenta una copia del record r.
Al refresh delle ore 12:15 ognuno dei nodi A, B, C, D ed E ricerca i nuovi target nodes, ma data l'ipotesi di rete stabile, ognuno individua come target nodes gli stessi nodi A, B, C, D ed E. Ognuno dei k nodi riceve quindi k refresh request e conferma il record r. Questa situazione di ripresenta sia al refresh delle ore 12:30, sia al refresh delle ore 12:45.
Al refresh delle ore 13:00 è introdotta la chiave *"r_13"* e rimossa la chiave *"r_08"*: sia N il nodo con *ID* più vicino all'*hash* della chiave *"r_13"*. Ognuno dei nodi A, B, C, D ed E ricerca i nuovi target nodes e individua i nodi N, A, B, C e D, perchè rispettivamente i più vicini agli *hash* delle chiavi *"r_13"*, *"r_12"*, *"r_11"*, *"r_10"*, *"r_09"*. Si ha quindi che i nodi A, B, C e D ricevono k refresh request e confermano nuovamente il record, N riceve k refresh request e lo memorizza per la prima volta, mentre il nodo E non riceve alcuna refresh request e quindi lo rimuove.

Dall' ESEMPIO_1 si evince che nell'ipotesi di rete stabile il ricambio dei target nodes è molto lento, al ritmo di un nuovo target node ogni ora: in particolare si tratta dello stesso ritmo col quale varia l'insieme delle k chiavi. Rimuovendo l'ipotesi di stabilità della rete potrebbe accadere che tra un refresh e il successivo i target nodes cambino anche tutti, ma questo è inverosimile. Con la rete a regime il comportamento dell'algoritmo non si discosta infatti molto dall'esempio presentato, a meno di un'elevata *churn rate*.

Al fine di chiarire il funzionamento generale dell'algoritmo, viene ora presentato un ulteriore esempio che descrive l'inserimento del record r e i tre refresh successivi. L'esempio assume $k = 3$ e ancora una volta, poichè gli eventi descritti si collocano

all'interno dello stesso giorno, nella costruzione delle chiavi è trascurata la data.
Alle ore 12:25 del 25 Agosto 2009 il nodo N (user) inserisce il record r all'interno del server: N assembla le k chiavi *"r_12"*, *"r_11"*, *"r_10"*, ne calcola gli *hash* *h(r_12)*, *h(r_11)* e *h(r_10)*, ricerca per ognuno il nodo più vicino (cioè i target nodes) e individua rispettivamente i nodi E, A ed I, quindi N invia loro il record r. I nodi E, A ed I memorizzano il record (Figura 2.4).

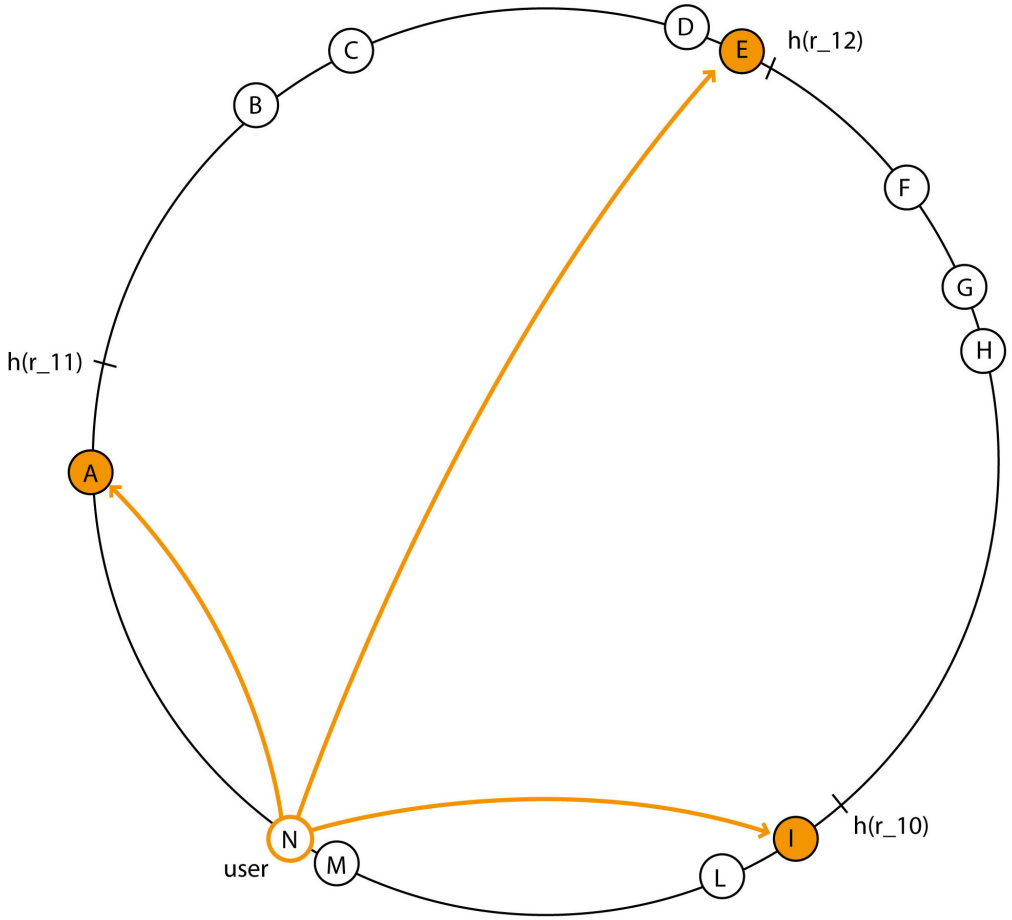

Figura 2.4: Alle ore 12:25 del 25 Agosto 2009 il nodo U inserisce il record r nel server.

Alle ore 12:30 del 25 Agosto 2009 è prevista una sessione di refresh. All'inizio del refresh i nodi in possesso del record r dovrebbero essere E, A ed I, ma il nodo E ha abbandonato la rete: i nodi A ed I assemblano comunque le chiavi, e queste sono ancora *"r_12"*, *"r_11"*, *"r_10"*. Il nodo più vicino ad *h(r_10)* è ancora I, il nodo più vicino ad *h(r_11)* è ancora A, mentre il nodo più vicino ad *h(r_12)* è ora il nodo D. I nodi A ed I inviano ciascuno una refresh request per il record r ad ognuno dei nuovi target nodes I, A e D, ognuno dei quali riceve $2 \geq \lceil k/2 \rceil = 2$ refresh request e memorizza quindi il record: in particolare D lo memorizza, mentre A ed I lo confermano (Figura 2.5).

Alle ore 12:45 del 25 Agosto 2009 è prevista una nuova sessione di refresh. I nodi

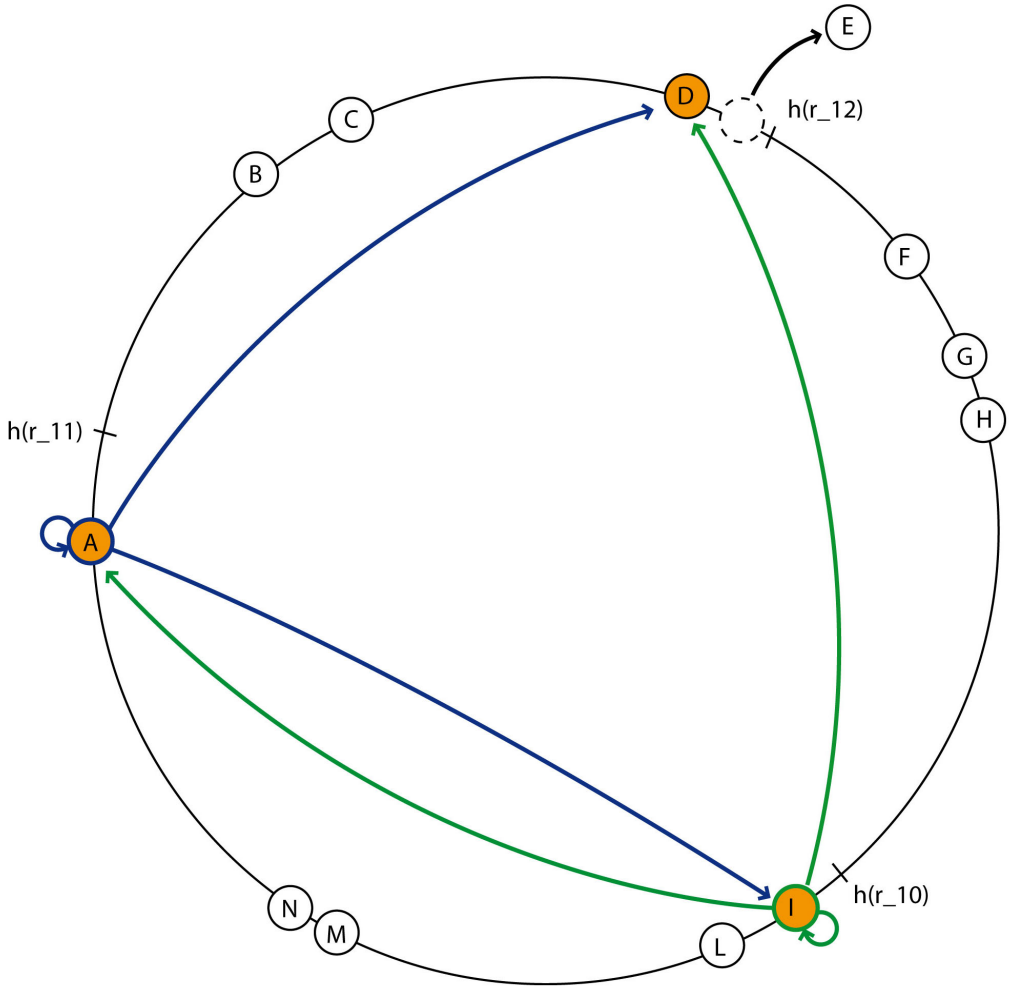

Figura 2.5: Refresh delle ore 12:30 del giorno 25 Agosto 2009.

in possesso del record r sono A, I e D: ognuno assembla le k chiavi e si tratta ancora una volta delle chiavi *"r_12"*, *"r_11"*, *"r_10"*. Il nodo più vicino ad *h(r_12)* è ancora il nodo D, il nodo più vicino ad *h(r_10)* è ancora il nodo I, mentre il nodo più vicino ad *h(r_11)* non è più il nodo A, ma il nodo O, entrato nella rete in un qualche istante successivo al precedente refresh, ma precedente al refresh corrente. I nodi in possesso del record r, A, I e D inviano ciascuno una refresh request per il record r ad ognuno dei nuovi target nodes D, I ed O, ognuno dei quali riceve $3 \geq \lceil k/2 \rceil = 2$ refresh request e memorizza quindi il record: in particolare O lo memorizza, mentre D ed I lo confermano. Notiamo che il nodo A cancella il record (Figura 2.6).

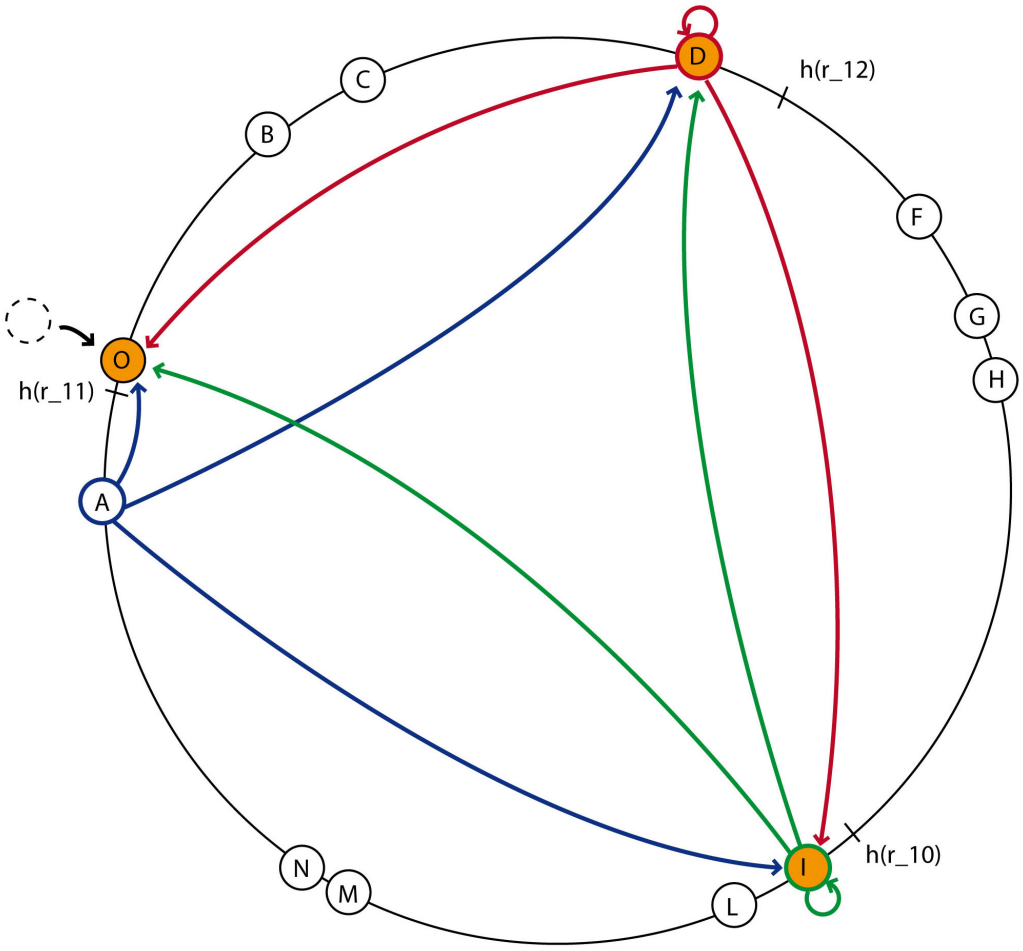

Figura 2.6: Refresh delle ore 12:45 del giorno 25 Agosto 2009.

Alle ore 13:00 del 25 Agosto 2009 è prevista una nuova sessione di refresh. I nodi in possesso del record r sono D, I ed O: ognuno assembla le k chiavi ma questa volta esse sono *"r_13"*, *"r_12"* e *"r_11"*. Il nodo più vicino ad *h(r_11)* è ancora il nodo O, il nodo più vicino ad *h(r_12)* è ancora il nodo D, e il nodo più vicino ad *h(r_13)* è il nodo M. I nodi in possesso del record r, D, I ed O inviano ciascuno una refresh request per il record r ad ognuno dei nuovi target nodes O, D ed M, ognuno dei quali riceve $3 \geq$

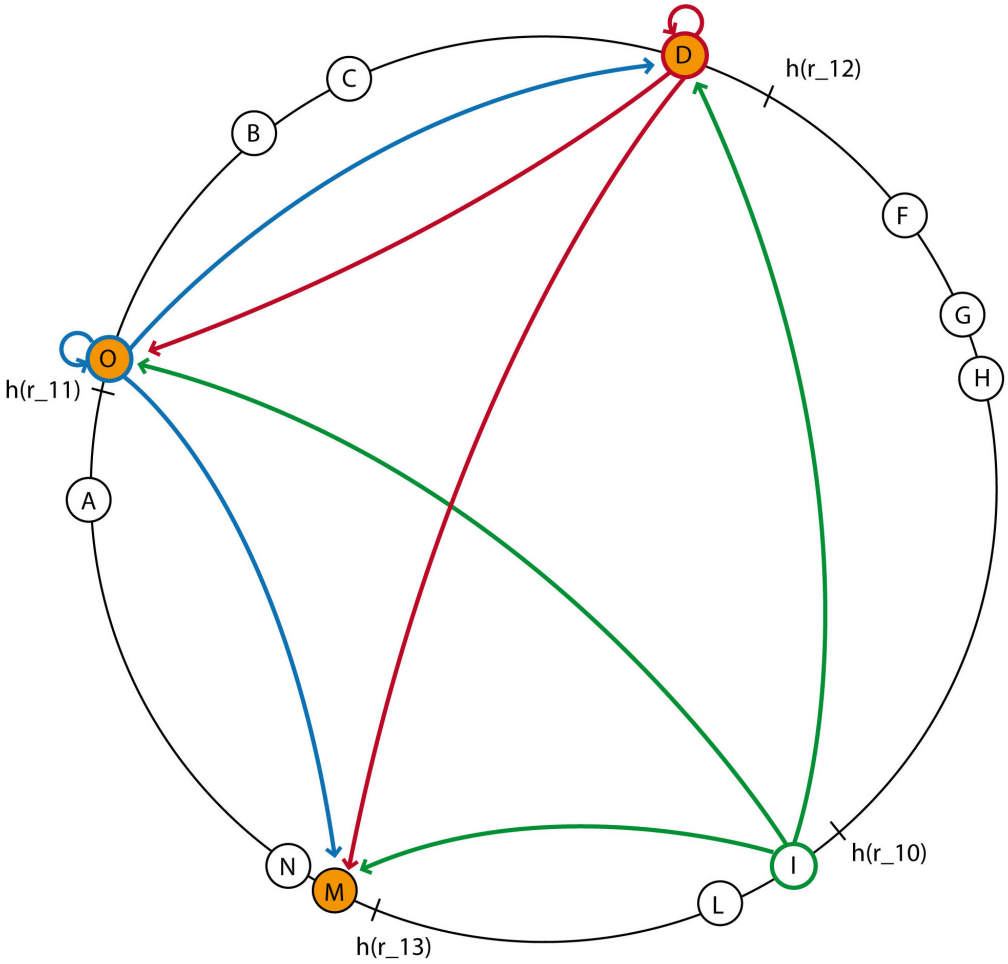

Figura 2.7: Refresh delle ore 13:00 del giorno 25 Agosto 2009.

$\lceil k/2 \rceil = 2$ refresh request e memorizza quindi il record: in particolare M lo memorizza, mentre D ed O lo confermano. Notiamo che il nodo I cancella il record (Figura 2.7).

2.4 Sicurezza

Nella presentazione dell'algoritmo utilizzato da PariDNS si è implicitamente supposto che ogni nodo all'interno della rete operasse correttamente. In realtà all'interno della rete alcuni nodi possono operare in modo disonesto e cercare di compromettere la consistenza dei record. Questa sezione sviluppa ulteriormente l'algoritmo presentato nella sezione precedente affrontando il problema della sicurezza. Ancora una volta, ai fini della trattazione, ci concentreremo sul singolo record r.

Nella sottosezione 2.3.1, nel descrivere la politica adottata dal nodo alla ricezione delle refresh request, si è trascurata la possibilità che le refresh request per uno stesso record r potesserero essere tra loro differenti, cioè contenere versioni di r tra loro diverse.

Generalmente tale situazione è indice di un tentativo da parte di altri nodi della rete di sabotare il record r. Questi nodi d'ora in poi saranno denominati "nodi cattivi".
É possibile distinguere principalmente due tipologie di nodi cattivi, che indicheremo rispettivamente con C1 e C2.

- Il nodo C1 è caratterizzato dal seguente comportamento.
  Al refresh i il nodo memorizza il record r in qualità di target node (è quindi eletto correttamente dai target nodes del refresh *i-1*), modifica la sua copia del record r, e al refresh *i+1* la inoltra, sotto forma di refresh request, a uno o più nodi della rete.

- Il nodo C2 è invece caratterizzato dal seguente comportamento.
  Al refresh i il nodo non è eletto dai target nodes del refresh *i-1* e quindi non riceve alcuna copia del record r, tuttavia il nodo assembla localmente una versione del record r differente dall'originale e al refresh *i+1* la inoltra, sotto forma di refresh request, a uno o più nodi della rete.

Notiamo che il nodo C1, sebbene invii refresh request "cattive", è autorizzatto all'invio di refresh request per il record r, perchè eletto correttamente dai target nodes del refresh *i-1*. Al contrario il nodo C2 non ha alcuna autorità per inviare refresh request per il record r. Ne consegue che proteggere un nodo dalle refresh request provenienti da un nodo di tipo C1 è più difficile che proteggerlo dalle refresh request di un nodo C2.

Consideriamo ora il comportamento adottato da un generico nodo alla ricezione di un flusso di refresh request per il record r durante il refresh *i+1*. Innanzitutto il nodo si preoccupa di eliminare le eventuali refresh request provenienti da nodi del tipo C2: questo è possibile grazie ad un procedimento denominato **Reverse Look Up**, che opera come un filtro e permette di selezionare le sole refresh request provenienti dai target nodes del refresh i. Il Reverse Look Up è oggetto della sottosezione 2.4.1.
Poichè il filtro dovrebbe aver eliminato tutte le refresh request provenienti da nodi del tipo C2, ora il nodo dovrebbe disporre di un numero di refresh request pari al più a k, alcune delle quali possono tuttavia provenire da nodi del tipo C1. Indicato con S l'insieme delle refresh request che hanno superato il filtro, si possono allora verificare due casi:

1. S contiene almeno $\lceil k/2 \rceil$ refresh request **uguali**: il nodo salva la versione del record r associata alle refresh request uguali ed invoca la primitiva STORE(r). In particolare se un record r è già presente in locale lo sovrascrive.

2. S **non** contiene almeno $\lceil k/2 \rceil$ refresh request **uguali**: se in locale è già memorizzato un record r il nodo lo rimuove, altrimenti il nodo ignora semplicemente le refresh request.

Infine sottolineiamo che l'evento *"il nodo possiede in locale un record r, ma per questo durante il refresh non riceve alcuna refresh request"* è equivalente al precedente caso 2: si ha infatti che S è vuoto e quindi il nodo rimuove localmente il record r.

Grazie al filtro introdotto, che seleziona solo le refresh request provenienti dai target nodes del refresh precedente, la politica descritta non si discosta molto da quella presentata in 2.3.1 per la gestione delle refresh request in ingresso. Tuttavia la possibilità di ricevere dai target nodes del refresh precedente refresh request diverse per uno stesso record r rende necessario il vincolo delle $\lceil k/2 \rceil$ request uguali: se infatti il nodo si limitasse a scegliere una qualsiasi delle versioni di r tra quelle in S potrebbe decidere di memorizzare proprio la versione di r contenuta nella refresh request proveniente da un nodo del tipo C1 e sabotare quindi r. La nuova politica permette invece al record r originale di sopravvivere ad un possibile attacco da parte di nodi del tipo C1 con grande probabilità. Segue ora la spiegazione.

Notiamo che la nuova politica del nodo fallisce solo nel caso in cui al refresh i l'algoritimo scelga, come target nodes per il refresh i, almeno $\lceil k/2 \rceil$ nodi cattivi su k[5]. In tal caso nessuno dei nuovi target nodes del refresh *i+1* riceverà abbastanza refresh request contenenti il record r originale e questo può portare a due diversi tipi di insuccesso:

1. se i nodi C1 si sono accordati su una stessa versione errata del record r, ogni target node del refresh *i+1* memorizzerà la versione errata di r;
2. se i nodi C1 sono tra loro indipendenti, cioè promuovono ognuno una diversa versione di r, nessuno dei target nodes del refresh *i+1* riceverà un numero di refresh request uguali sufficienti a memorizzare il record r, che quindi sparirà dal server.

Indicato con X l'evento *"l'algoritmo sceglie k nodi e tra questi ve ne sono un numero pari ad i di cattivi"* e con p la probabilità che un nodo all'interno della rete sia cattivo, la probabilità che l'algortimo scelga k nodi e tra questi ve ne siano almeno $\lceil k/2 \rceil$ cattivi è:

$$P\{X \geq \lceil k/2 \rceil\} = \sum_{i=\lceil k/2 \rceil}^{k} \binom{k}{i} p^i (1-p)^{k-i} \tag{2.1}$$

Il grafico di Figura 2.8 decrive la (2.1) in funzione di p e ogni curva corrisponde ad un diverso valore della molteplicità k. Notiamo come la probabilità che l'algoritmo scelga in un qualsiasi refresh k nodi di cui almeno $\lceil k/2 \rceil$ cattivi assuma valori molto bassi anche in presenza di un'alta percentuale di nodi cattivi ($p = 0.1$, cioè il 10% dei nodi della rete è cattivo) e piccola molteplicità ($k = 5$). Ne consegue che la probabilità che

[5]Si assume che i target nodes del refresh *i-1* siano concordi nella scelta dei target nodes per il refresh *i*.

la nuova politica adottata fallisca a causa della presenza nella rete di nodi malevoli si può considerare bassa: inoltre è possibile contrastare un eventuale aumento della percentuale di nodi malevoli aumentando la molteplicità k dei record.

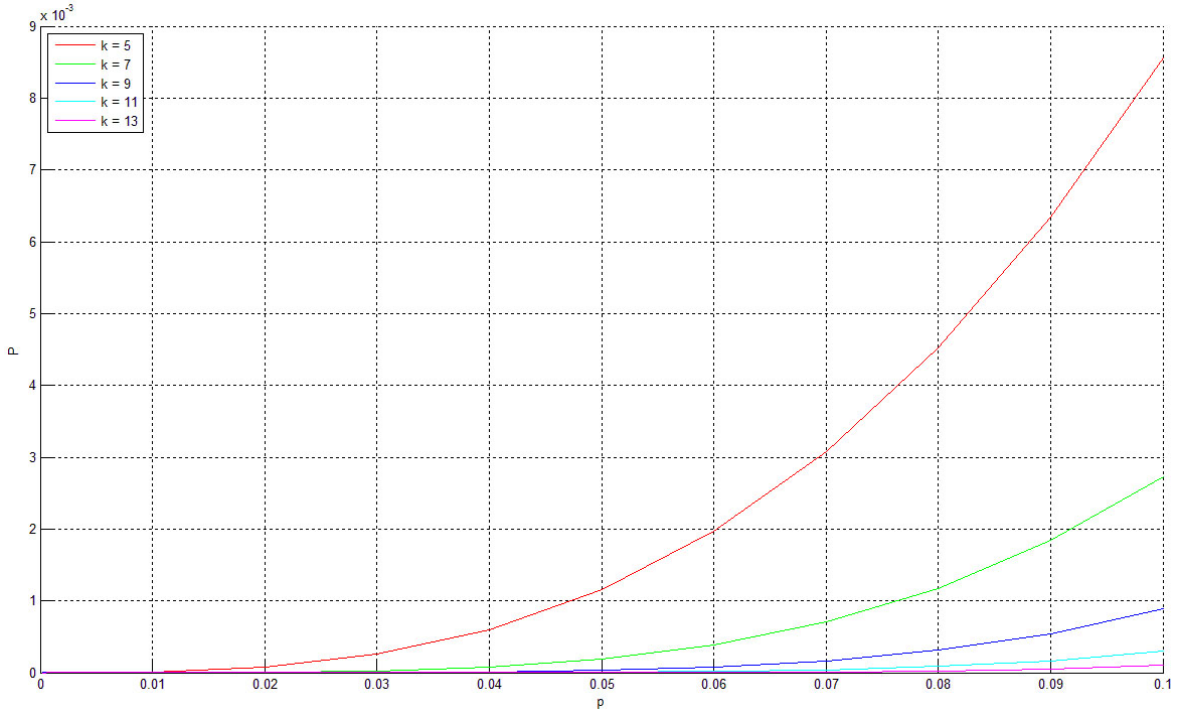

Figura 2.8: La probabilità $P\{X \geq \lceil k/2 \rceil\}$ in funzione di p.

2.4.1 Reverse Look Up

Questo procedimento, attuato al refresh *i+1*, è finalizzato ad individuare e quindi eliminare le refresh request per il record r provenienti da nodi non autorizzati al loro invio nella sessione di refresh corrente (nodi C2). I soli nodi autorizzati all'invio di refresh request per r durante il refresh *i+1* sono i k target nodes eletti al precedente refresh i, e quindi legittimi detentori del record r.

Il procedimento prevede che alla ricezione della prima refresh request associata al record r il nodo calcoli le chiavi associate al refresh precedente (i) e che per ogni chiave richieda al modulo DHT di individuare il nodo attualmente più vicino all'hash della chiave: il nodo "dovrebbe" in questo modo individuare i k target nodes eletti al precedente refresh i. A questo punto il procedimento prevede che il nodo accetti solo le refresh request provenienti dai nodi individuati.

L'efficienza di questo procedimento è tuttavia legata al livello di churn rate cui è soggetta la rete. In particolare il suo funzionamento risulta perfetto in assenza di churn, ma all'aumentare di questo il procedimento descritto rischia di scartare non solo le refresh request dei nodi C2, ma anche quelle provenienti dai nodi autorizzati al loro invio (i target nodes del refresh i). Per individuare i target nodes del refresh i, il nodo cerca

infatti di eseguire le stesse operazioni attuate dai target nodes del refresh *i-1* per individuare i target nodes da eleggere al refresh i: tuttavia questo procedimento può restituire nodi diversi se attuato in istanti differenti. Segue un esempio. Nell'esempio si supporrà $k = 5$ e per semplicità considereremo sessioni di refresh caratterizzate dalle stesse chiavi.

ESEMPIO_2
Al refresh delle ore 9:15 le chiavi associate al record r sono *"r_09"*, *"r_08"*, *"r_07"*, *"r_06"* e *"r_05"*: denominiamo i rispettivi *hash* con *h9*, *h8*, *h7*, *h6* e *h5*. Supponiamo quindi che al refresh delle ore 9:15 i target nodes del refresh precedente eleggano quali nuovi target nodes i nodi A, B, C, D ed E, perchè rispettivamente i più vicini agli hash *h9*, *h8*, *h7*, *h6* e *h5*. Il record r è ora affidato ai soli nodi A, B, C, D ed E.
Supponiamo ora che alle ore 9:25 entri nella rete il nodo J, più vicino di C all'hash *h7*.
Al refresh delle ore 9:30 le chiavi associate ad r sono le stesse del refresh precedente e quindi anche gli *hash*. I nodi A, B, C, D ed E individuano i nuovi target nodes, si tratta dei nodi A, B, J, D ed E, e inviano ognuno una refresh request ad ognuno dei nuovi target nodes.
Ognuno dei nuovi target nodes, di fronte alla ricezione delle refresh request per il record r, esegue il Reverse Look Up e ottiene la seguente lista di nodi: A, B, J, D ed E. Ne consegue che ognuno dei nuovi target nodes scarterà la refresh request del nodo C, sebbene questa sia corretta e soprattutto leggittima.
La situazione diventa pericolosa quando i nodi nuovi come J diventano molti.

2.4.2 Reverse Look Up migliorato

Migliorare il Reverse Look Up appena descritto, ovvero, con riferimento all'ESEMPIO_2, cercare di evitare che la refresh request del nodo C sia ritenuta malevola, è possibile. Nel Reverse Look Up presentato in 2.4.1 il nodo calcola le k chiavi key_i relative al precedente refresh e successivamente per ogni key_i richiede a DHT il nodo più vicino all'*hash* della chiave. Notiamo tuttavia che chiedendo invece che il solo nodo più vicino, gli α più vicini, è possibile, ad esempio con $\alpha = 2$, correggere il Reverse Look Up in modo tale che il nodo accetti anche la refresh request di C. Notiamo tuttavia che se in 2.4.1 la lista di nodi dai quali il nodo accetta refresh request per il record r contiene solo k nodi, la modifica introdotta porta la lista a comprendere $k*\alpha$ nodi: ne consegue che all'aumentare di α dimunuisce la probabilità di scartare refresh request "buone", come nel caso di C, ma al tempo stesso diminuisce anche la protezione offerta dal Reverse Look Up perchè il nodo accetta refresh request da un numero sempre maggiore di altri nodi.

É possibile migliorare ulteriormente il Reverse Look Up sfruttando quanto argomentato in "*Osservazione importante sui target nodes*" (2.3.1). In genere un nodo X eletto al ruolo di target node al refresh i è rieletto anche per un certo numero di refresh consecutivi al refresh i: è possibile sfruttare questo fatto.

Al refresh *i+1* il nodo X individua i nuovi target nodes e invia a ciascuno di essi una refresh request per il record r. Il nodo X sta eleggendo, in accordo con gli altri target nodes del refresh i, i target nodes del refresh *i+1*: X sa allora che qualora dovesse ricevere delle refresh request al refresh *i+2* i nodi incaricati di inviargliele saranno tutti e soli i nodi da lui eletti al refresh *i+1*.
Il nodo opera quindi nel seguente modo. Quando al refresh *i+1* il nodo X calcola i nuovi target nodes memorizza una coppia (r, *target nodes del refresh i+1*) all'interno di una tabella denominata **Tabella di Look Up**, e qualora al successivo refresh *(i+2)* dovesse ricevere delle refresh request per r riterrà attendibili tutte e sole quelle provenienti dai nodi memorizzati nella coppia (r, *target nodes del refresh i+1*).
Notiamo che il nodo utilizza la tabella di look up riempita al refresh precedente per controllare le refresh request in arrivo nella sessione di refresh corrente, e nella sessione di refresh corrente riempie la tabella di look up necessaria a controllare le refresh request che arriveranno nel prossimo refresh.
Risulta ovvio che qualora il nodo X riceva delle refresh request per un record r, e non abbia nella tabella di look up reimpita al precedente refresh alcuna voce associata ad r, il nodo dovrà necessariamente intraprendere la via del Reverse Look Up base.

Notiamo che l'utilizzo della tabella di look up introduce una maggior efficenza nel Reverse Look Up. In particolare la coppia (r, *target nodes del refresh precedente*), se presente nella tabella, da un lato evita inutili ricerche al modulo DHT, dall'altra fornisce una sicurezza maggiore del Reverse Look Up basato su DHT perchè al contrario di questo la coppia fornisce con esattezza i nodi autorizzati ad inviare le refresh request per r nel refresh corrente.

Capitolo 3

Primitive

Il name server PariDNS fornisce quattro servizi:

- inserimento/registrazione di un dominio;
- aggiornamento di un dominio;
- cancellazione di un dominio;
- risoluzione di un nome di dominio.

Il plugin DNS prevede una differente primitiva per ognuno dei primi tre servizi, mentre ne prevede due per il solo servizio di risoluzione: una finalizzata alla risoluzione dei nomi nel dominio *paripari.it*, e una seconda finalizzata alla risoluzione dei nomi appartenenti ai domini esterni.

I servizi di inserimento, aggiornamento e cancellazione sono accessibili sia dalla console del plugin, per l'utente, sia attraverso le sue API, per gli altri plugin residenti sulla stessa macchina. In entrambi i casi il richiedente il servizio è obbligato ad autenticarsi localmente. Se il richiedente è un utente egli deve essere loggato presso il modulo *Login* locale, in modo tale che il modulo DNS possa avere accesso alla sua chiave pubblica e indirettamente anche alla sua chiave privata. Qualora il richiedente sia un plugin è invece compito del plugin stesso fornire nelle API, al momento della richiesta, una coppia chiave pubblica-privata. L'associazione di una coppia di chiavi ad ogni utente permette a PariDNS di tutelare i dati registrati da ogni utente all'interno del server.
Ai fini della nostra trattazione supporremo che il richiedente il servizio sia un utente correttamente loggato.

Il servizio di risoluzione è accessibile, come i precedenti servizi, sia da console, sia attraverso le API, ma è raggiungibile anche interrogando il plugin sulla porta 53 utilizzando il protocollo DNS standard.

Prima di affrontare nel dettaglio i servizi offerti dal name server PariDNS è necessaria una precisazione. All'interno del Domain Name System ad uno stesso nome di dominio

r possono essere associati più Resource Record: in particolare l'insieme di questi record è denominato *Resource Record Set (RRSet)*. Nel precedente capitolo si è affrontato il problema della distribuzione del server PariDNS con riferimento al posizionamento nella rete del singolo record r, tuttavia lo stesso PariDNS, come ogni altro name server, raggruppa i record in RRSet e di conseguenza non tratta singoli record r ma singoli RRSet.

Nel trattare le operazioni di inserimento, aggiornamento e cancellazione supporremo, per semplicità, che il server gestisca solo domini del terzo livello, cioè nella forma *<name>.paripari.it*.[1]

3.1 Inserimento

Il servizio di inserimento è realizzato al lato client dalla primitiva REMOTE_INSERT: questa prevede l'invio ai target node, introdotti nel capitolo 2, di un messaggio denominato INSERT_REQUEST.
Ogni INSERT_REQUEST contiene i seguenti campi:

- *domainName*: il nome del dominio da registrare;
- *rrset*: il Resource Record Set associato al nome di dominio *domainName*;
- *publicKey*: la chiave pubblica del richiedente l'inserimento;
- *timestamp*: il timestamp relativo all'istante di creazione del messaggio;
- *domainID*: un intero generato in modo pseudocasuale alla creazione del messaggio stesso;
- *targetNodes*: un vettore contenente gli indirizzi di tutti i nodi destinatari del messaggio.

Il plugin DNS, a fronte di una richiesta di registrazione del dominio *domainName*, intraprende la seguente procedura.

1. Invoco la primitiva FIND(*domainName*): se il modulo DHT non restituisce alcun nodo proseguo, altrimenti termino e segnalo all'utente l'impossibilità di registrare il dominio *domainName* poichè già registrato.
2. Richiedo al modulo Login la chiave pubblica dell'utente.
3. Sulla base dell'ora attuale assemblo le k chiavi associate al nome di dominio *domainName*.
4. Richiedo al modulo DHT i target nodes.

[1] PariDNS prevede in realtà anche la gestione dei livelli superiori.

5. Assemblo il messaggio INSERT_REQUEST e ne inoltro una copia a ciascun target node.

Prima di descrivere l'inserimento al lato server è necessario presentare la struttura utilizzata da ogni nodo per la memorizzazione del RRSet associato al nome di dominio *domainName*. Questa struttura è denominata RRBOX e contiene sostanzialmente gli stessi campi presenti in ogni INSERT_REQUEST. Obiettivo della struttura è infatti la memorizzazione dei dati contenuti nel messaggio INSERT_REQUEST. Seguono i campi della struttura RRBOX:

- *domainName*: il nome di dominio associato al RRSet contenuto nell'RRBOX.
- *rrset*: il Resource Record Set;
- *publicKey*: la chiave pubblica del proprietario del dominio *domainName*.
- *timestamp*: il timestamp associato al RRSet memorizzato.
- *domainID*: il *domainID* associato al dominio *domainName* al momento del suo inserimento nella rete.

Alla ricezione di un messaggio INSERT_REQUEST il plugin DNS intraprende la seguente procedura.

1. Verifico se il dominio *domainName* è già registrato in locale: in caso affermativo termino, altrimenti proseguo.
2. Invoco la primitiva FIND(*domainName*): se il modulo DHT restituisce un numero di nodi superiore o uguale a k termino, altrimenti proseguo.
3. Utilizzo il messaggio INSERT_REQUEST per assemblare un'RRBOX.
4. Inserisco una coppia (*domainName*, *targetNodes*) all'interno della tabella di look up[2].
5. Invoco la primitiva STORE(*domainName*).

Impedire ad un nodo cattivo la registrazione di un dominio già presente nella rete

É necessario considerare la possibile presenza di nodi cattivi anche nelle operazioni di inserimento. Un nodo corretto, alla richiesta di inserimento del dominio *domainName* verifica prima se questo è già registrato nel server attraverso la chiamata alla primitiva FIND(*domainName*). In caso affermativo il nodo corretto non permette l'inserimento.

[2]Questo espediente consente ad ogni target nodes eletto in fase di inserimento di evitare il Reverse Look Up base al primo refresh successivo all'inserimento qualora sia rieletto. Funge da bootstrap.

Un nodo cattivo potrebbe invece ignorare questo controllo e decidere di inoltrare volontariamente nella rete un messaggio di inserimento per un dominio già registrato al fine di associare a quel dominio un RRSet diverso: questo non deve essere possibile. Dal momento che impedire al nodo cattivo l'invio del messaggio di inserimento non è possibile, l'inserimento deve essere impedito dai nodi destinatari del messaggio, sui quali il nodo cattivo non dovrebbe avere alcun controllo. Le prime due istruzioni della procedura prevista da ogni nodo alla ricezione di un messaggio INSERT_REQUEST hanno esattamente questo obiettivo (punti 1 e 2).

ESEMPIO_3
Supponiamo che il nodo X (corretto) decida di registrare il dominio r. Verifica allora la sua presenza nel server e appurato che il dominio non è già registrato invia un messaggio INSERT_REQUEST ai target nodes. Ogni nodo destinatario nota che in locale il dominio non è registrato, verifica che nella rete il dominio è registrato su meno di k nodi, e quindi registra localmente il dominio ed invoca la primitiva STORE(r). Notiamo che la procedura descritta è seguita da ognuno dei nodi destinatari (target nodes), quindi una chiamata alla primitiva FIND(r) al termine dell'inserimento del dominio r nel server dovrebbe restituire esattamente k nodi.
Supponiamo ora che il nodo cattivo Y voglia inserire nella rete una versione errata del dominio r. Le situazioni possibili sono due:

- il messaggio è inoltrato ad uno dei target nodes di r: in tal caso il nodo scarta il messaggio al punto 1 della procedura;
- il messaggio non è inoltrato ad uno dei target nodes di r: il messaggio supera il controllo al punto 1 ma il controllo al punto 2 restituisce un numero di nodi pari almeno a k quindi il messaggio è comunque scartato.

3.2 Aggiornamento

Il servizio di aggiornamento è realizzato al lato client dalla primitiva REMOTE_UPDATE, e prevede l'utilizzo di un messaggio denominato UPDATE_REQUEST.
Ogni UPDATE_REQUEST contiene i seguenti campi:

- *domainName*: il nome del dominio da aggiornare;
- *rrset*: il Resource Record Set con cui aggiornare il dominio *domainName*;
- *timestamp*: il timestamp relativo all'istante di creazione del messaggio;
- *signature*: la firma del messaggio.

Il plugin DNS, a fronte di una richiesta di aggiornamento del dominio *domainName*, intraprende la seguente procedura.

1. Invoco la primitiva FIND(*domainName*): se il modulo DHT non restituisce alcun nodo termino e segnalo all'utente che il dominio *domainName* non risulta registrato nel server, altrimenti memorizzo temporaneamente i nodi e proseguo.
2. Assemblo un messaggio UPDATE_REQUEST.
3. Converto in byte il contenuto di ognuno dei campi *domainName*, *rrset* e *timestamp*. Concateno le tre sequenze di byte e ne richiedo la firma al modulo Login con la chiave privata dell'utente loggato[3].
4. Inserisco la firma nel campo *signature* del messaggio assemblato.
5. Inoltro il messaggio UPDATE_REQUEST ad ognuno dei nodi individuati attraverso la chiamata alla primitiva FIND(*domainName*) al punto 1.

Alla ricezione di un messaggio UPDATE_REQUEST il plugin DNS intraprende la seguente procedura.

1. Verifico se il dominio *domainName* è registrato localmente: in caso affermativo proseguo, altrimenti termino.
2. Se il *timestamp* associato al messaggio è minore o uguale del *timestamp* associato all'RRBOX del dominio *domainName* termino, altrimenti proseguo.
3. Verifico la firma del messaggio di aggiornamento con la chiave *publicKey* contenuta nell'RRBOX del dominio *domainName*: se il controllo ha esito negativo termino, altrimenti proseguo.
4. Aggiorno i campi *rrset* e *timestamp* dell'RRBOX del dominio *domainName* col contenuto dei rispettivi campi del messaggio di aggiornamento.

La firma del messaggio di aggiornamento garantisce che solo il proprietario possa aggiornare i propri domini. Notiamo infine che qualora uno dei nodi destinatari del messaggio fosse un nodo cattivo, il nodo potrebbe mettere da parte il messaggio firmato per inoltrarlo nella rete in un secondo momento al fine di provocare la perdita di uno o più aggiornamenti. Tale operazione non risulta tuttavia possibile perchè la procedura adottata da un nodo alla ricezione di un messaggio UPDATE_REQUEST prevede il controllo del suo *timestamp* (punto 2) e questo non risulta sabotabile perchè all'interno della firma.

3.3 Cancellazione

Il servizio di cancelazione è realizzato al lato client dalla primitiva REMOTE_DELETE, e prevede l'utilizzo di un messaggio denominato DELETE_REQUEST.
Ogni DELETE_REQUEST contiene i seguenti campi:

[3]Il modulo DNS non ha accesso diretto alla chiave privata dell'utente.

- *domainName*: il nome del dominio da rimuovere dal server;
- *domainID*: il *domainID* associato al dominio *domainName* al momento del suo inserimento nella rete.
- *timestamp*: il timestamp relativo all'istante di creazione del messaggio;
- *signature*: la firma del messaggio.

Il plugin DNS, a fronte di una richiesta di cancellazione del dominio *domainName*, intraprende la seguente procedura.

1. Invoco la primitiva FIND(*domainName*): se il modulo DHT non restituisce alcun nodo termino e segnalo all'utente che il dominio *domainName* non risulta registrato nel server, altrimenti memorizzo temporaneamente i nodi e proseguo.
2. Richiedo ai nodi individuati al punto 1 il *domainID* associato al dominio *domainName*.
3. Assemblo un messaggio DELETE_REQUEST.
4. Converto in byte il contenuto dei soli campi *domainName* e *recordID*. Concateno le due sequenze di byte e ne richiedo la firma al modulo Login con la chiave privata dell'utente loggato.
5. Inserisco la firma nel campo *signature* del messaggio assemblato.
6. Inoltro il messaggio DELETE_REQUEST ad ognuno dei nodi individuati attraverso la chiamata alla primitiva FIND(*domainName*) al punto 1.

Alla ricezione di un messaggio DELETE_REQUEST il plugin DNS intraprende la seguente procedura.

1. Verifico se il dominio *domainName* è registrato localmente: in caso affermativo proseguo, altrimenti termino.
2. Verifico la firma del messaggio di cancellazione con la chiave *publicKey* contenuta nell'RRBOX del dominio *domainName*: se il controllo ha esito negativo termino, altrimenti proseguo.
3. Se il *domainID* contenuto nel messaggio è diverso dal *domainID* contenuto nell'RRBOX del dominio *domainName* termino, altrimenti proseguo;
4. Rimuovo l'RRBOX associato al dominio *domainName*.

L'operazione di cancellazione, come l'operazione di aggiornamento, ricorre alla firma del messaggio. Tuttavia la primitiva di cancellazione non comprende il *timestamp* nella firma, ma il contenuto del campo *domainID*.
La procedura adottata da un nodo alla ricezione di un messaggio DELETE_REQUEST prevede la possibilità, per il nodo ricevente, di inoltrare nuovamente il messaggio all'interno della rete. Se infatti l'utente richiede la cancellazione del dominio *domainName*, all'inizio di una sessione di refresh, il messaggio potrebbe raggiungere i target nodes nel momento in cui questi hanno già eletto i loro successori e affidato a questi la custodia del dominio di cui è richiesta la cancellazione. La possibilità di inoltrare il messaggio DELETE_REQUEST permette ai nodi che lo hanno ricevuto di richiedere la cancellazione del dominio ai nuovi target nodes senza che l'utente debba intervenire direttamente.
Questa operazione, denominata **propagazione**, prevede tuttavia che il nodo "intermediario" possa modificare il *timestamp* del messaggio e di conseguenza questo campo non può essere firmato. Notiamo tuttavia che firmare il contenuto del solo campo *domainName* si rivelerebbe pericoloso, anche se consideriamo che l'utente sta rimuovendo il dominio dalla rete. Supponiamo infatti che uno dei nodi destinatari del messaggio sia cattivo e memorizzi, dopo aver portato a termine la cancellazione, il messaggio DELETE_REQUEST. Qualora, in un secondo momento, l'utente che ha richiesto la rimozione del dominio *domainName* lo registri nuovamente, il nodo cattivo potrebbe inoltrare nella rete il messaggio precedentemente memorizzato e provocarne la cancellazione. Qualsiasi controllo sul *timestamp* del messaggio risulterebbe inutile perchè questo non è firmato ed è quindi sabotabile.
Si è deciso allora di assegnare ad ogni dominio un intero generato in modo pseudocasuale al momento della registrazione, il *domainID*, al fine di utilizzarlo assieme al *domainName* nella generazione della firma del messaggio DELETE_REQUEST.
Con riferimento all'esempio precedente si ha che il messaggio DELETE_REQUEST utilizzato dall'utente per richiedere la cancellazione del dominio *domainName* non può essere riutilizzato da un nodo cattivo per cancellare in un secondo momento il nuovo dominio *domainName* registrato dallo stesso utente, perchè questo è sicuramente identificato da un *domainID* differente rispetto al precedente.

3.4 Risoluzione

Le primitive previste per la risoluzione sono due:

- RESOLVE_FROM_LEGACY
- RESOLVE_FROM_NODES

Il plugin invoca la primitiva RESOLVE_FROM_LEGACY a fronte della richiesta di risoluzione di un nome esterno al dominio *paripari.it*. L'implementazione della primitiva prevede che il nodo interroghi la legacy in modo iterativo sino alla risoluzione del nome:

PariDNS non restituisce quidi record di tipo NS, ma la risposta finale alla richiesta di risoluzione.

Il plugin invoca la primitiva RESOLVE_FROM_NODES per la risoluzione di nomi interni al dominio *paripari.it*. A livello teorico, la risoluzione del nome di dominio r potrebbe avvenire nei seguenti passi:

- il plugin invoca la primitiva FIND(r);
- il plugin interroga un nodo qualsiasi tra quelli restituiti dal modulo DHT;
- il nodo interrogato risponde al plugin col RRSet associato al nome di dominio r;
- il plugin risponde alla query di risoluzione col RRSet ricevuto.

Questo esempio di risoluzione non fornisce tuttavia alcuna sicurezza. Vediamone il motivo.
Consideriamo il refresh i e il successivo refresh *i+1*, e indichiamo rispettivamente con t_i e t_{i+1} i loro istanti di inizio. Supponiamo quindi che la richiesta di risoluzione del dominio r avvenga in un qualche istante t nell'intervallo $[t_i, t_{i+1}[$. Notiamo allora che il record r (ai fini della spiegazione trascureremo il concetto di RRSet) si dovrebbe trovare solo nei target nodes eletti all'inizio del refresh i. Tuttavia è necessario tenere presente che nella rete potrebbero esserci anche alcuni nodi non eletti al ruolo di target nodes al refresh i, ma comunque in possesso di una versione del record r e per giunta differente dall'originale: si tratta in sostanza dei nodi cattivi del tipo C2 introdotti nella sezione 2.4. Ne consegue che una chiamata alla primitiva FIND(r) non restituirebbe solo i target nodes ma anche i nodi C2: contattare un nodo qualunque tra i nodi restituiti dal modulo DHT si rivela dunque pericoloso.

Nella risoluzione di un nome di dominio r è innanzitutto necessario individuare e scartare i nodi C2 dall'insieme dei nodi restituiti dalla chiamata a DHT.
Una prima idea prevede di utilizzare il Reverse Look Up nella sua versione base. Il nodo adotta il seguente procedimento:

1. invoco la primitiva FIND(r): denomino S l'insieme dei nodi restituiti dal modulo DHT;
2. assemblo le k chiavi associate ad r e relative al refresh i;
3. richiedo a DHT, per ogni chiave assemblata, gli α nodi più vicini al suo *hash*: denomino T l'insieme dei nodi restituiti da questa operazione;
4. calcolo l'intersezione degli insiemi S e T: denomino l'insieme risultante U.

Possiamo notare che nei punti 2 e 3 il nodo esegue il Reverse Look Up nella sua versione base. Il Reverse Look Up individua per ogni *hash* gli α nodi più vicini, e con α sufficientemente piccolo si può ritenere improbabile che i nodi C2 si concentrino esattamente tra i nodi più vicini ai k *hash*, cioè che i nodi C2 si trovino in T: l'intersezione con l'insieme S dovrebbe allora permettere di scartare i nodi C2 e l'insieme U dovrebbe coincidere con l'insieme dei target nodes del refresh i.
L'applicazione di questo procedimento per l'eliminazione dei nodi C2 non è tuttavia utilizzabile in fase di risoluzione. Il procedimento descritto richiede infatti al modulo DHT k+1 ricerche, k per il Reverse Look Up e una al momento dell'invocazione della primitiva FIND(r): un tale numero di ricerche richiederebbe troppo tempo provocando un ritardo eccessivo nella risoluzione di r.

A fronte dell'impossibità di adottare il procedimento descritto si è deciso di "simulare" localmente il Reverse Look Up, cioè senza ricorre al modulo DHT: il prezzo è una minor sicurezza. Il nodo adotta il seguente procedimento:

1. invoco la primitiva FIND(r): denomino S l'insieme dei nodi restituiti dal modulo DHT;
2. assemblo le k chiavi associate ad r e relative al refresh i;
3. calcolo per ogni chiave assemblata il rispettivo *hash*;
4. per ogni *hash* calcolato cerco nell'insieme S il nodo più vicino e lo marco[4];
5. denomino U l'insieme dei nodi marcati: U contiene al più k nodi.

Anche nell'ipotesi che tutti i nodi C2 siano stati rimossi, cioè nell'ipotesi che i k nodi nell'insieme U siano i soli target nodes del refresh i, tra questi possono comunque essere presenti dei nodi cattivi: si tratta dei nodi C1 della sezione 2.4. Il nodo interroga allora tutti i nodi nell'insieme U e confronta i record ottenuti in risposta: in particolare considera attendibile la sola versione del record r della quale siano pervenute almeno $\lceil k/2 \rceil$ copie. Con tale versione del record r il nodo risponde al mittente della query di risoluzione. Il nodo adotta quindi una politica analoga a quella adottata in fase di refresh.
Notiamo come questo procedimento preveda una sola ricerca da parte del modulo DHT: la ricerca innescata dalla chiamata alla primitiva FIND(r).

Segue ora un esempio il cui fine è evidenziare la maggior sicurezza del primo procedimento rispetto al secondo nello scartare i nodi del tipo C2.
Fissiamo innanzituto $k = 1$ e supponiamo che al refresh i sia eletto al ruolo di target

[4]Due o più hash possono comunque marcare lo stesso nodo in S.

node il nodo A in quanto nodo più vicino ad $hash_0$, *hash* dell'unica chiave key_0. Supponiamo poi che il record r sia reso disponibile anche da un altro nodo, Z: questo è però un nodo C2, cioè dotato di una copia del record r anche se non eletto al ruolo di target node al refresh i e la sua versione di r è differente dall'originale. La richiesta di risoluzione avviene ancora una volta all'istante t compreso nell'intervallo $[t_i, t_{i+1}[$ e supponiamo che il nodo A sia caduto poco prima: Z è ora l'unico nodo in possesso del record r. Descriviamo il comportamento dei due diversi procedimenti in questa particolare situazione.
Primo procedimento:

1. invoco la primitiva FIND(r): $S = \{Z\}$;
2. assemblo la chiave key_0;
3. richiedo a DHT gli $\alpha = 2$ nodi più vicini ad $hash_0$: DHT restituisce, in ordine di distanza crescente, i nodi B e C, e si ha quindi $T = \{B, C\}$;
4. $U = S \cap T = 0$.

Il primo procedimento scarta il nodo Z e la risoluzione termina.
Secondo procedimento:

1. invoco la primitiva FIND(r): $S = \{Z\}$;
2. assemblo la chiave key_0;
3. cerco in S il nodo più vicino ad $hash_0$ e individuo il nodo Z;
4. $U = \{Z\}$.

Il secondo procedimento non scarta il nodo Z perchè Z è l'unico nodo disponibile in S ed è quindi anche il più vicino ad $hash_0$. L'esempio presentato propone tuttavia una situazione molto particolare: in genere k non assume mai il valore 1, e con $k > 1$ anche qualora un nodo C2 dovesse essere presente in U la sua versione del record r verrebbe utilizzata per rispondere alla query di risoluzione solo nel caso in cui nell'insieme U siano presenti almeno $\lceil k/2 \rceil$ nodi con la stessa versione errata del record r posseduta dal nodo C2.

Capitolo 4

Architettura del plugin

Il plugin DNS è organizzato in quattro package, a ognuno dei quali è affidato un particolare compito:

- `paripari.dns.protocol` - contiene tutte le classi che implementano il protocollo DNS standard.
- `paripari.dns.server` - rappresenta il *kernel* del plugin, e contiene tutte le classi che implementano i servizi presentati nel precedente capitolo 3.
- `paripari.dns.refresh` - contiene le classi che implementano l'algoritmo di distribuzione del server descritto nelle sezione 2.3 ed esteso nella sezione 2.4.
- `paripari.dns.network` - gestisce la comunicazione tra le istanze del plugin DNS attive all'interno della rete PariPari.

L'intero plugin è inizializzato dalla classe `DNS`: questa si occupa in particolare di istanziare le classi `NetInitializer` e `DNSServer` che rappresentano rispettivamente le classi principali dei package `network` e `server`.

4.1 Il package dns.server

Il package `server` contiene principalmente tre classi:

- `DNSServer` - classe principale del package, contiene il database del nodo, le primitive per i servizi di inserimento, aggiornamento, cancellazione e risoluzione, e istanzia i tre principali thread del plugin: `DNSQueryReceiver`, `TransactionRequestHandler` e `RefreshHandler`[1].
- `DNSQueryReceiver` - thread server in ascolto sulla porta 53 predisposto alla gestione delle query DNS provenienti dalla rete.

[1] I thread sono poi lanciati dalla classe `DNS` al termine della creazione della classe `DNSServer`.

- `TransactionRequestHandler` - thread predisposto all'elaborazione dei messaggi di inserimento, aggiornamento e cancellazione, provenienti dalle altre istanze del plugin DNS attive nella rete.

Come anticipato nel capitolo 3 ogni primitiva prevede l'utilizzo di un determinato messaggio, il package `server` contiene una classe per ognuno di questi:

- `InsertRequest` - classe che realizza il messaggio per la richiesta di inserimento del dominio r.
- `UpdateRequest` - classe che realizza il messaggio per la richiesta di aggiornamento del dominio r.
- `DeleteRequest` - classe che realizza il messaggio per la richiesta di rimozione del dominio r.
- `ResolveRequest` - classe che realizza il messaggio per la richiesta del RRSet associato al dominio r: è utilizzata nella primitiva RESOLVE_FROM_NODES per interrogare i nodi sui quali il dominio r è memorizzato.
- `ResolveReply` - classe che realizza il messaggio di risposta ad un'oggetto `ResolveRequest`.

Gli oggetti `InsertRequest`, `UpdateRequest` e `DeleteRequest` sono genericamente denominati transaction request.

Il package `server` contiene infine la classe `RRBox`, che realizza la struttura RRBOX introdotta nella sezione 3.1: in particolare il database del plugin, contenuto nella classe `DNSServer`, è un'hashtable di coppie (nome di dominio r , `RRBox` associato ad r).

DNSQueryReceiver

Il thread `DNSQueryReceiver` ha il compito di gestire le query DNS in arrivo dalla rete, ma in realtà il thread si limita solo alla loro ricezione. Al fine di garantire il massimo parallelismo il thread server affida infatti ogni query ad un nuovo thread `DNSQueryResolver`. Quest'ultimo si occupa di analizzare il pacchetto attraverso le classi del package `protocol`, risolve la query invocando la primitiva RESOLVE_FROM_LEGACY o RESOLVE_FROM_NODES disponibili all'interno della classe `DNS Server`, e infine, sempre ricorrendo alle classi del package `protocol`, risponde al mittente.

TransactionRequestHandler

Le transaction request rivolte al nodo sono ricevute attraverso le classi del package `network` e depositate nella coda *transactionReqQueue* all'interno della classe `DNSServer`. Il compito del thread `TransactionRequestHandler` è prelevare dalla coda ogni transaction request e se possibile soddisfarla.

4.2 Il package dns.refresh

La classe principale del package è `RefreshHandler` e in particolare si tratta di un thread attivo esclusivamente nelle sessioni di refresh.
All'inizio di ogni refresh il thread si sveglia e congela il database del nodo per il tempo strettamente necessario a copiare ogni oggetto `RRBox` del database e ad inserirne la copia in una coda: il thread esegue quindi una copia dell'intero database locale. A questo punto il thread istanzia e lancia j `IncomingRefreshRequestThread` e j `OutcomingRefreshRequestThread`: a questi ultimi j thread `RefreshHandler` affida la coda contenente la copia del database.

OutcomingRefreshRequestThread

I j thread `OutcomingRefreshRequestThread` hanno il compito di inoltrare nella rete le refresh request per ogni dominio registrato in locale, per questo motivo `RefreshHandler` affida loro una copia dell'intero database. Ognuno dei j thread esegue il seguente loop sino allo svuotamento della coda, poi termina:

- prelevo un oggetto `RRBox` e noto che è relativo al dominio r;
- assemblo k refresh request per r;
- calcolo i target nodes per il dominio r;
- invio una refresh request ad ogni target node.

É importante sottolineare che il numero di thread lanciati, j, è un parametro variabile scelto sulla base del numero di oggetti `RRBox` presenti nella coda. Notiamo infine che poichè ogni refresh request rappresenta un messaggio, esattamente come i messaggi associati alle primitive, il package `refresh` contiene una classe `RefreshRequest` che implementa il messaggio refresh request.

IncomingRefreshRequestThread

Le refresh request provenienti dalle altre istanze del plugin attive nella rete sono ricevute attraverso le classi del package `network` e depositate nella coda *refreshQueue* all'interno della classe `DNSServer`: l'elaborazione delle refresh request nella coda è affidata ai j thread `IncomingRefreshRequestThread` i quali hanno il compito di applicare l'algoritmo descritto nelle sezioni 2.3 e 2.4.
I j thread utilizzano una particolare struttura dati organizzata in bucket, nella quale ogni bucket è destinato alla memorizzazione di tutte e sole le refresh request associate ad uno stesso nome di dominio: denominiamo S tale struttura. Senza entrare nel dettaglio le operazioni eseguite da ciascun thread possono essere approssimate dal seguente loop:

- prelevo una refresh request dalla coda *refreshQueue*: la refresh request è associata al nome di dominio r;
- sottopongo la refresh request al Reverse Look Up: se il controllo scarta la refresh request torno al primo punto, altrimenti proseguo;
- se in S è già presente un bucket per il dominio r inserisco la refresh request nel bucket, altrimenti procedo alla sua creazione;
- se il bucket contiene $\lceil k/2 \rceil$ refresh request contenenti la stessa versione del record r memorizzo/sovrascrivo il record r nel database, altrimenti torno al primo punto;
- marco il bucket associato al dominio r in modo tale che ulteriori refresh request per il dominio r prelevate dalla coda *refreshQueue* siano ignorate;
- torno al primo punto.

Notiamo che i j thread non rimuovono alcun dominio dal database locale: è infatti compito del thread `RefreshHandler`, al termine della sessione di refresh, rimuovere dal database tutti i domini non memorizzati o sovrascritti dai thread `IncomingRefreshRequestThread`, cioè i domini per i quali il nodo non è più target node.

4.3 Il package dns.network

Il package `network` gestice lo scambio di messaggi tra i peer che compongono pariDNS attraverso l'invio e la ricezione di oggetti della classe `DNSPacket`: il protocollo utilizzato è UDP. Il package contiene principalmente le seguenti classi:

- `NetInitializer` - classe principale del package, istanzia e lancia i thread `NetSender`, `NetReceiver` e `NetReceiverDispatcher`;
- `NetSender` - thread predisposto all'invio nella rete degli oggetti `DNSPacket` generati internamente al plugin;
- `NetReceiver` - thread server predisposto alla ricezione di oggetti `DNSPacket`;
- `NetReceiverDispatcher` - thread col compito di smistare gli oggetti `DNSPacket` ricevuti dal thread `NetReceiver` sulla base del messaggio contenuto al loro interno.

Segue ora una descrizione generale del funzionamento del package.
I messaggi utilizzati dal plugin sono i seguenti:

- `InsertRequest`;
- `UpdateRequest`;
- `DeleteRequest`;

- `ResolveRequest` e `ResolveReply`;
- `RefreshRequest`.

L'invio di un qualsiasi messaggio prevede il suo inserimento all'interno di un'oggetto `DNSPacket` nel quale è necessario specificare le coordinate del destinatario e il tipo di messaggio contenuto: *request* o *reply*. Se il messaggio è una request è necessario affidare il pacchetto ad un nuovo thread `OutRequest`, anch'esso contenuto nel package `network`. Il thread `OutRequest` ha il compito di inserire il pacchetto `DNSPacket` nella coda dei pacchetti da inviare del thread `NetSender`, e di attendere l'arrivo del pacchetto contenente la relativa reply. Notiamo che se la request è un messaggio `ResolveRequest` allora la reply è un messaggio `ResolveReply`, mentre per le altre request la reply non è rappresentata da un apposito messaggio ma da un semplice pacchetto vuoto che funge da *acknowledge*.

NetSender

I pacchetti da inoltrare nella rete sono affidati alla coda *packetSenderQueue* nella classe `NetInitializer`. Il thread `NetSender` serializza ogni pacchetto, lo inserisce in un datagramma UDP, e lo invia al destinatario.

NetReceiver

Alla ricezione di un datagramma UDP il thread `NetReceiver` deserializza il contenuto del datagramma, ottiene un'oggetto `DNSPacket`, e lo inserisce nella coda *packetReceiverQueue* della classe `NetInitializer`.

NetReceiverDispatcher

Il thread `NetReceiverDispatcher` preleva ad uno ad uno i pacchetti nella coda *packetReceiverQueue*, ne esamina il contenuto, e intraprende una delle seguenti procedure:

- se il messaggio all'interno del `DNSPacket` è una reply affido il pacchetto al rispettivo thread `OutRequest`;
- se il messaggio all'interno del `DNSPacket` è un `InsertRequest`, un `UpdateRequest` o una `DeleteRequest`, accodo il messaggio nella coda *transactionReqQueue* della classe `DNSServer` e invio un pacchetto di acknowledge al mittente: il pacchetto è inserito direttamente nella coda *packetSenderQueue*;
- se il messaggio all'interno del `DNSPacket` è una `RefreshRequest` inserisco il messaggio nella coda *refreshQueue* della classe `DNSServer` e invio un pacchetto di acknowledge al mittente: il pacchetto è inserito direttamente nella coda *packetSenderQueue*;

- se infine il messaggio all'interno del `DNSPacket` è una `ResolveRequest` affido il pacchetto ad un nuovo thread `InRequest`: il thread assembla un messaggio `ResolveReply` per il messaggio `ResolveRequest` affidatogli, lo inserisce all'interno di un oggetto `DNSPacket` destinato al mittente della request, e lo affida direttamente alla coda *packetSenderQueue*, poi termina.

La decisone di affidare la gestione di ogni messaggio `ResolveRequest` ad un nuovo thread `InRequest` è dettata dalla necessità di garantire una maggior velocità nelle operazioni di risoluzione. Nel momento in cui un nodo del server PariDNS riceve attraverso il thread `DNSQueryReceiver` una query di risoluzione per il nome di dominio r interno a *paripari.it*, il nodo deve interrogare i nodi del server in possesso del record r attraverso l'invio di messaggi `ResolveRequest`: ne consegue che la velocità con la quale il nodo risponderà al mittente della query dipende in modo diretto dalla velocità con la quale i nodi interrogati invierrano al nodo i messaggi `ResolveReply`. La gestione seriale dei messaggi `Resolve Request` da parte del thread `NetReceiverDispatcher` potrebbe introdurre notevoli ritardi, mentre l'affidamento di ogni `ResolveRequest` ad un thread `InRequest` riduce l'intervallo temporale compreso tra l'invio della request e la ricezione della relativa reply.

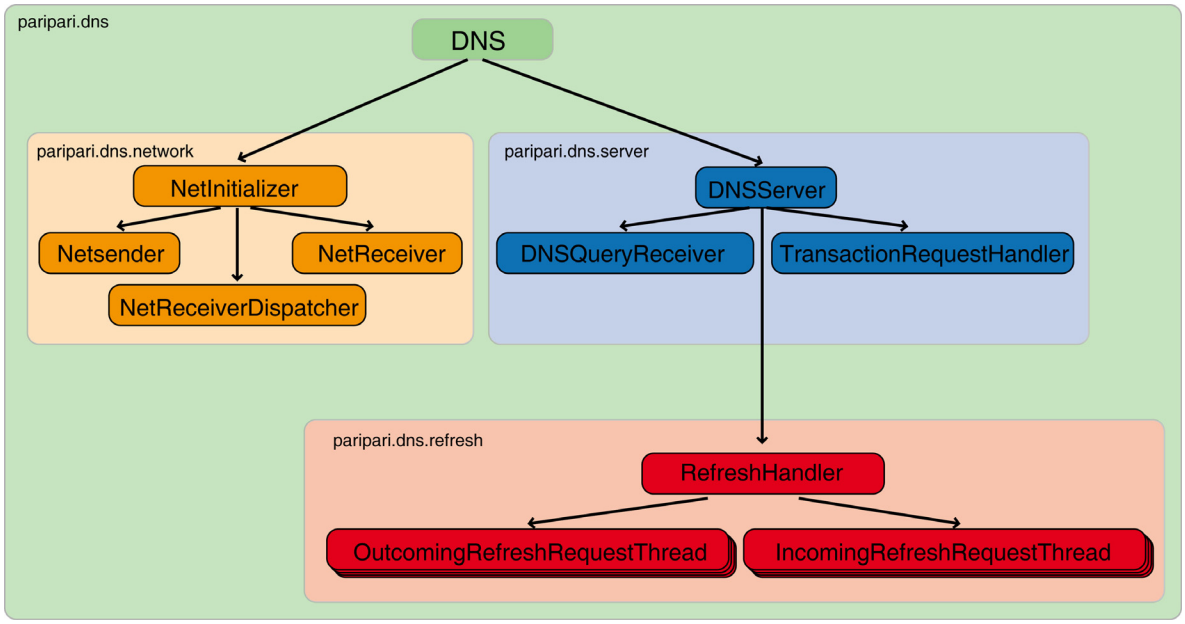

Figura 4.1: Struttura del plugin DNS.

Capitolo 5

Conclusioni

La realizzazione del name server PariDNS si è rivelato un obiettivo piuttosto ambizioso. L'esportazione di un servizio tipicamente server based all'interno di un contesto completamente decentralizzato quale la rete peer-to-peer PariPari pone infatti seri problemi, soprattutto per quanto concerne persistenza dei dati e sicurezza. All'interno di una rete peer-to-peer il comportamento dei singoli nodi non è prevedibile: ogni nodo può sia abbandonare la rete in qualunque momento rendendo irraggiungibili i dati in esso memorizzati, sia agire in modo disonesto nei confronti degli altri nodi. Il problema della sicurezza risulta ancora più evidente se si considera che il codice sorgente dell'intero client PariPari è *opensource* e che quindi ogni utente ha la possibilità di riscrivere qualunque modulo.
Per fronteggiare questi due problemi la progettazione del server è stata suddivisa in due fasi. Nella prima fase si è ricercato un metodo che permettesse di distribuire in modo efficiente il name server, ma che al tempo stesso permettesse di garantire la persistenza dei dati in esso memorizzati: il risultato di questa prima fase di progetto è l'algoritmo presentato nella sezione 2.3. Nella seconda fase ci si è invece occupati di rendere sicuro l'algoritmo sviluppato, cercando di individuarne gli aspetti su cui dei possibili nodi malevoli avrebbero potuto far leva al fine di sabotare il corretto funzionamento del server: il risultato di questa fase è rappresentato principalmente dalla sezione 2.4.

Rispetto ai più comuni name server centralizzati, PariDNS è caratterizzato da un elevato grado di *fault tolerance* e risulta quindi immune ad attacchi DDOS (Distributed Denila Of Service). Ogni istanza del plugin DNS attiva nella rete rappresenta infatti un diverso punto di accesso al server e di conseguenza la caduta di alcuni nodi non preclude l'accesso al server stesso. La disponibilità di numerosi punti di accesso rende inoltre possibile attuare una politica di bilanciamento del carico, ripartendo le query dirette al server tra più istanze del plugin DNS (*load balancing*).

Possiamo definire PariDNS un name server innovativo poichè da un lato mira a garantire la persistenza dei dati e la sicurezza offerta dalla comune architettura centralizzata, tipica dei name server della legacy, ma al tempo stesso eredita le principali proprietà della rete strutturata su cui poggia, quali la *fault tolerance* e il *load balancing*,

che rappresentano proprietà di più difficile realizzazione all'interno di un'architettura centralizzata.

Bibliografia

[1] Paolo Bertasi. *Progettazione e realizzazione in Java di una rete peer to peer anonima e multifunzionale.* Dipartimento di Ingegneria dell'Informazione, Università di Padova, 2004.

[2] Russ Cox, Athicha Muthitacharoen, Robert T. Morris. *Serving DNS using a Peer-To-Peer Lookup Service.* MIT Laboratory for Computer Science, 2002.

[3] Venugopalan Ramasubramanian, Emin Gün Sirer. *The Design and Implementation of a Next Generation Name Service for the Internet.* Dept. of Computer Science, Cornell University, 2004.

[4] R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan. *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*. ACM SIGCOMM 2001, San Diego, CA, Settembre 2001.

[5] S. Ratnasamy, P. Francis, S. Shenker, R. Karp, M. Handley. *A Scalable Content-Addressable Network.* In Proceedings of ACM SOGCOMM, 2001, pp. 161-172.

[6] A. Rowstron, P. Druschel. *Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*. 2001.

[7] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, J. Kubiatowicz. *Tapestry: A Resilient Global-Scale Overlay for Service Deployment.* IEEE Journal on Selected Areas in Communications, vol. 22, No. 1, Gennaio 2004.

[8] Petar Maymounkov, David Mazières. *Kademlia: A peer-to-peer Information System Based on the XOR Metric.* 2002.

[9] RFC 1035 *http://www.ietf.org/rfc/rfc1035.txt* 1987

[10] Paul V. Mockapetris, Kevin J. Dunlap. *Development of The Domain Name System.* Proceedings of SIGCOMM '88, Computer Communication Review Vol. 18, No. 4, August 1988, pp. 123-133.

[11] Larry L. Peterson, Bruce S. Davie. *Reti di Calcolatori.* Apogeo, 2004.