UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Informatica

PariDHT: Sistema di permessi applicato alle operazioni sulla rete

RELATORE: Prof. Enoch Peserico Stecchini Negri De Salvi

CORRELATORE: Dott. Michele Bonazza

LAUREANDO: Maicol Bozzato

Anno Accademico 2012-2013

Alla mia famiglia per il loro pieno sostegno.
A Elisa per la spensieratezza trasmessami
durante il periodo degli studi.

Prefazione

Negli ultimi anni l'aumento delle connessioni a banda larga ha portato alla diffusione delle reti peer-to-peer (P2P). Applicazioni quali BitTorrent[2], eMule[3], Skype[4] e Tor[5] permettono agli utenti di condividere file, conversare e tutelare la propria privacy proprio grazie alle reti P2P. PariPari si inserisce in questo contesto offrendo questi ed altri servizi attraverso un unico software, ponendo rimedio ai conflitti che sorgono se si eseguono contemporaneamente applicazioni come quelle citate. PariPari vuole, inoltre, essere un servizio completamente decentralizzato, cioè un servizio che non necessiti di alcun server per funzionare. Questi, insieme alla facilità di utilizzo e alla garanzia di anonimato, sono i punti di forza del progetto. Per raggiungere questi obiettivi è fondamentale basarsi su una rete che sia il più efficiente e sicura possibile. Per poter aumentare la sicurezza, si sta cercando di distinguere PariPari dalle altre reti, integrando un sistema di permessi che possa offrire agli utenti la possibilità di decidere se e con chi condividere certe risorse, e una strategia che renda difficile l'oscuramento di una parte (se non tutte) delle risorse per colpa di nodi malevoli.

Indice

Introduzione

In questo elaborato verrà inizialmente introdotto il progetto PariPari esponendo brevemente obiettivi, architettura e organizzazione. Nel capitolo 2, verrà presentata l'evoluzione delle reti P2P fino ad arrivare alle odierne DHT. Successivamente verrà descritta la rete Kademlia, base per lo sviluppo di PariDHT, la DHT di PariPari. La penultima parte è dedicata a PariDHT, dove saranno trattate le caratteristiche sia base sia estese, e per finire si parlerà delle nuove funzionalità che dovranno essere supportate e che sono in fase di sviluppo, cioè del sistema di permessi che PariDHT si prefigge di offrire, analizzando come le operazioni messe a disposizione cambino in funzione ad esso e degli strumenti che saranno usati.

Capitolo 1

PariPari

PariPari è una rete P2P serverless, pensata per fornire servizi sia agli utenti che ne fanno parte, sia a computer esterni. Con il solo software di PariPari sarà possibile effettuare videoconferenze, dialogare tramite le varie chat, o utilizzare le reti di filesharing come eDonkey, Kademlia[8] o BitTorrent[2]. Saranno disponibili anche servizi quali storage distribuito, DNS, web server e quant'altro è possibile ottenere tramite Internet, ma una delle peculiarità che distingue PariPari è la gestione di tutte queste features tramite un unico programma, avviabile da un qualsiasi browser.

Figura 1.1: Logo di PariPari

1 Architettura

PariPari presenta una struttura modulare, scelta sia per semplificare lo sviluppo stesso sia per una migliore organizzazione degli addetti ai lavori: i moduli o plugin sono sezioni di programma a sè stanti che offrono un servizio specifico. Ogni plugin di PariPari può comunicare con gli altri esclusivamente tramite il sistema di scambio messaggi del Core. Quest'ultimo rappresenta l'asse portante dell'intero sistema perché si fa carico dell'avvio della piattaforma, di reperire e "installare/caricare" gli altri moduli e gestirne le intercomunicazioni e le richieste; insieme ad esso, ci sono una serie di altri moduli che formano la cerchia interna di PariPari in quanto offrono servizi essenziali agli altri plugin. Essi sono:

- *Credits* : Gestiscono i crediti dei vari plugin, nonché la tassazione di ogni richiesta che passa tramite il Core
- *GUI* : Offre l'interfaccia grafica
- *ConnectivityNIO* : Gestisce i socket e quindi le comunicazioni tra i vari nodi di PariPari
- *Local Storage* : Gestisce le operazioni su disco richieste dagli altri plugin (es. lettura, scrittura su file)
- *DHT (Distributed Hash Table)* : definisce la struttura della rete, e le operazioni che si svolgono su di essa (es. salvataggio e reperimento delle risorse).

Al di fuori di questa cerchia ci sono i plugin rivolti al pubblico, come Mulo[22] (eMule[33]), Torrent[23] (BitTorrent[31], μTorrent[32]), DNS, distributed storage, distributed backup, VOIP, IM[20][21] (Pidgin[30]), , IRC, WEB, NTP. Tuttavia il numero di questi plugin è destinato a crescere, in quanto le applicazioni che possiamo creare sopra a questa struttura sono veramente molte.

Capitolo 2

Reti P2P

Come già detto, una rete P2P è una rete in cui i nodi non sono divisi in client e server, ma svolgono il ruolo e i compiti di cliente e servente allo stesso tempo. I vantaggi e svantaggi di una rete di questo tipo sono principalmente i seguenti:

- Non è necessario un server con potenzialità elevate, quindi non bisogna sostenerne i costi per l'acquisto e la manutenzione. D'altra parte, però, ogni computer della rete deve avere i requisiti per sostenere l'utente in locale e in rete, ma anche eventuali altri utenti che desiderassero accedere alle risorse del computer.

- Ogni utente condivide le proprie risorse, ed è quindi l'amministratore del proprio client-server. Questo ha però lo svantaggio di richiedere delle competenze ad ogni utente soprattutto per quanto riguarda la protezione.

- La velocità media di trasmissione dei dati è potenzialmente molto più elevata di una classica rete client-server, infatti l'informazione richiesta da un client può essere reperita da molti altri nodi, anziché da un unico server. Questo tipo di condivisione diventa tanto più efficace tanti più sono i client connessi, in contrapposizione alla rete tradizionale in cui un elevato numero di client connessi riduce la velocità di trasmissione dei dati per utente.

- La rete è più robusta. In una rete client-server è sufficiente che il server non sia più connesso affinché la rete non sia più accessibile; in una rete P2P, invece, finché continueranno ad essere connessi un certo numero di nodi la rete continuerà a funzionare.

1 Evoluzione

1.1 Prima generazione

Le prime reti P2P si basavano su un modello centralizzato, Napster[9] ne è un esempio. Infatti, permetteva agli utenti di condividere file musicali attraverso alcuni server utilizzati per tenere in contatto i nodi tra loro, ma che non prendeva parte nei trasferimenti di file. Dopo l'avvio del programma e l'avvenuta connessione al server, ogni nodo inviava a quest'ultimo la lista di tutti i file musicali che intendeva condividere. Quindi, la procedura di richiesta di un file consisteva nell'inviare una query al server che in risposta spediva una lista di nodi che possedevano i file desiderati. A questo punto i nodi, avendo questa lista, comunicavano tra loro per iniziare il trasferimento del file richiesto. Napster era quindi un sistema ibrido in cui la ricerca avveniva attraverso un sistema client-server mentre il trasferimento era più simile ad una rete P2P. I limiti di questo sistema sono evidenti: da una parte la poca scalabilità, dall'altra la scarsa robustezza. Infatti era particolarmente esposta ad attacchi **DoS** (*denial of service*), cioè una volta abbattuto il server principale la rete non poteva più funzionare.

1.2 Seconda generazione

Successivamente venne introdotto il modello distribuito non strutturato, di cui l'esempio più famoso è Gnutella[10]. In questo caso tutte le operazioni, ricerca, avvio e trasferimento, avvenivano in modo distribuito, cioè senza l'uso di un server. Non essendo la rete strutturata, le ricerche avvenivano in broadcast (*flooding*[26]), cioè, veniva inviato un messaggio a tutti i nodi conosciuti che, a loro volta, lo inviavano ai loro conoscenti fino a raggiungere il nodo o la risorsa desiderata. Per evitare che i messaggi circolassero per sempre sulla rete, a ciascuno di essi era associato un valore chiamato **TTL** (*time to live*) che indicava il massimo numero di nodi che poteva percorrere il messaggio. Ciascun nodo, inoltre, memorizza temporaneamente l'identificativo dei messaggi che passavano attraverso di esso, così da scartare eventuali messaggi già ricevuti senza inviarli nuovamente ai propri conoscenti. Lo svantaggio di questa metodologia è nel non determinismo di una ricerca, nell'overhead di comunicazione e nella possibilità di falsi negativi (non perfettamente scalabile), quindi meno efficiente rispetto ad una rete centralizzata.

1.3 Terza generazione

Si giunse dunque allo sviluppo del modello distribuito strutturato, in cui la topologia della rete è ben definita, permettendo una maggiore scalabilità e affidabilità delle operazioni di ricerca e salvataggio. La strada seguita per ottenere questo risultato è l'utilizzo delle DHT, che porta i seguenti vantaggi:

> **Decentralizzazione**. Non è necessario alcun coordinamento centrale, sono i nodi nel loro insieme a formare e gestire il sistema.
>
> **Scalabilità**. Il sistema può essere utilizzato in modo efficiente anche in presenza di centinaia di milioni di nodi.
>
> **Tolleranza ai guasti**. Il sistema, con i dovuti accorgimenti, è in grado di funzionare anche con nodi che si connettono e disconnettono continuamente dalla rete o che sono spesso soggetti a malfunzionamenti.

Protocolli che utilizzano le DHT sono, ad esempio, Chord[11], BitTorent[2] e Kademlia[8].

2 DHT

DHT è un acronimo per *Distributed Hash Table* e identifica un particolare sistema di immagazzinamento e reperimento delle informazioni che sfrutta una rete di calcolatori. Le informazioni che è possibile salvare all'interno di una DHT sono delle coppie <K, V> (K = chiave, V = valore), dove la chiave identifica univocamente la risorsa, ovvero il valore stesso; la differenza rispetto a una normale tabella hash sta nel fatto che queste coppie <K, V> sono distribuite il più possibile equamente tra tutti i partecipanti alla rete; ciò presenta un problema di riperimento delle informazioni condivise ma, a questo scopo, esistono degli algoritmi che, nel caso medio, raggiungono una complessità computazionale logaritmica nel numero di nodi.

2.1 Come funziona

La prima particolarità di questa infrastruttura sta nel fatto che i nodi sono identificati all'interno della rete da un ID che condivide lo stesso dominio delle chiavi: questa accortezza permette di individuare i nodi responsabili di una determinata risorsa, ovvero quei partecipanti aventi ID più vicino

alla chiave memorizzata. Parlando di vicinanza resta ancora da definire cosa intendiamo per nodi vicini o lontani e per fare questo introduciamo il concetto di metrica.

Definizione 1 (Metrica)**.** Si dice metrica una funzione matematica $d: X \times X \to \Re$ tale che, $\forall\ x, y, z \in X$ gode delle seguenti proprietà:

- d(x, y) ≥ 0
- d(x, y) $= 0 \Leftrightarrow$ x = y
- d(x, y) = d(y, x)
- d(x, z) $\leq$ d(x, y) + d(y, z)

I nodi responsabili di una determinata chiave sono quindi quei nodi con un ID tale da rendere minima la distanza d(ID, chiave).

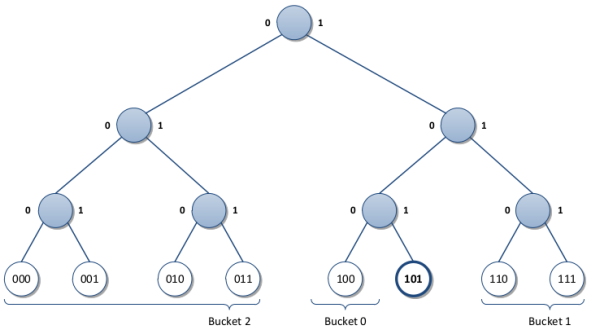

Figura 2.1: Esempio di partizione della rete

Ciò che la contradistingue, però, è l'*overlay network*. La rete, per essere operativa, necessita che ogni nodo mantenga dei contatti, non casuali, con altri nodi, strutturando così la topologia della rete. Questa topologia deve godere della proprietà che, per ciascuna chiave k, o il nodo possiede k oppure conosce un nodo più vicino ad essa. Detta caratteristica permette, da qualsiasi nodo della rete, di spedire un messaggio al responsabile di una certa risorsa r, secondo un algoritmo che si ripete ad ogni hop della comunicazione; esso prevede che chi riceve il messaggio abbia due possibilità:

1. tutti i contatti a sua disposizione sono più distanti da k rispetto a se stesso, ciò ne fa automaticamente il responsabile della risorsa e può quindi elaborare il messaggio;

2. identifica il nodo di cui è a conoscenza avente l'ID più vicino possibile a k, inoltrandogli il messaggio.

Una simile sovrastruttura è fondamentale per permettere il funzionamento della rete stessa, dato che, viene assunta vera da tutti gli algoritmi di salvataggio e reperimento delle informazioni.

3 Kademlia

Kademlia, alla base dello sviluppo di PariDHT, è l'esempio più famoso di DHT (eMule[3] la usa). Questo protocollo è stato ideato da Petar Maymounkow e David Mazières e usa come metrica la funzione *XOR*, quindi si ha che d(x, y) = x $\oplus$ y. Questa metrica fornisce una topologia ad

X	**Y**	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

Tabella 2.1: Tabella della verità dell'operazione XOR

albero binario della rete. Se si considerano i nodi della rete come le foglie dell'albero, e si assegna 0 ai rami di sinistra e 1 ai rami di destra allora il percorso dalla radice al nodo ne definisce il suo identificativo.

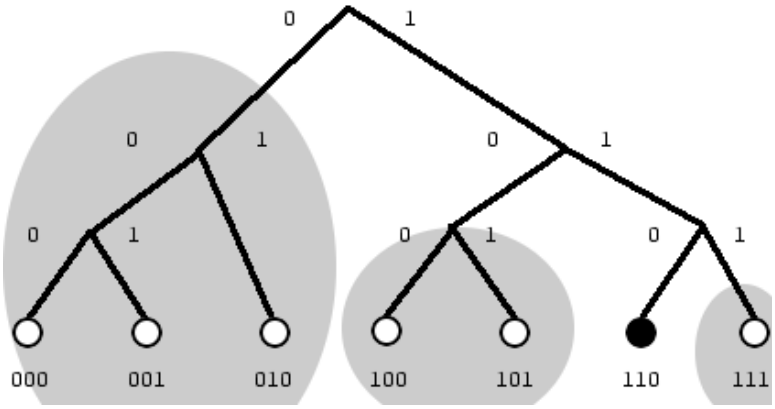

Figura 2.2: Esempio di implementazione della rete

In Kademlia, ogni nodo ha $ID = b_{m-1}, b_{m-2}, ..., b_1, b_0$ con $m = 128$, l'albero è formato da 128 bit. Ogni nodo mantiene una lista di m bucket, detti k-bucket, in cui memorizza le informazioni per contattare un certo nodo. Il generico bucket $i \in [0, ..., m\text{-}1]$ contiene nodi che hanno distanza d con $2^i \leq d < 2^{i+1}$, cioè include i nodi che hanno ID con gli stessi valori nei bit $b_{m-1}, b_{m-2}, ..., b_{i+1}$, e un valore diverso nel bit b_i. Il simbolo k, presente nel termine k-bucket, rappresenta una costante (generalmente posta pari a 20) che indica il numero massimo di nodi che può contenere ciascun bucket.

Capitolo 3

PariDHT

In questo paragrafo verranno esposti gli aspetti principali di PariDHT tralasciando per ora gli algoritmi (che sono in fase di implementazione) per la ricerca, il salvataggio e la rimozione di una risorsa che verranno trattati nel capitolo successivo più in dettaglio. Sarà accennata la procedura di *look up*, utile per selezionare i nodi con cui i nodi possono scambiare messaggi, per aumentare la conoscenza della rete di un nodo e per il semplice ricerca di una risorsa.

1 Requisiti funzionali

Nella realizzazione del plug-in si sta cercando di rispettare il più possibile le seguenti specifiche:

- **Funzionalità**. Fornire un modulo DHT che dia la possibilità agli altri plug-in di pubblicare, reperire ed eliminare risorse dalla rete.
- **Espandibilità**. La scelta di rendere il plug-in il più modulare possibile è atta a rendere più semplici eventuali espansioni future (*Open-Closed principale* [1]).
- **Documentazione**. Vista la frequenza con la quale i componenti del gruppo vengono cambiati, è molto importante che la documentazione sia di buona qualità. Per questo motivo si fa uso di Javadoc[13] per commentare il codice e della wiki per spiegare come gli altri plug-in possono utilizzare PariDHT.

[1] Software entities (classes, modules, functions, etc.) should be open for extension, but clodes for modification.

- **Prestazioni**. Per una DHT è di fondamentale importanza valutare attentamente alcuni parametri come il numero e la dimensione dei messaggi scambiati tra i vari nodi. Il rischio è, altrimenti, che l'overhead generato dalla rete renda il plug-in lento e sconveniente per l'utente. Le prestazioni locali, valutate ad esempio come numero di cicli macchina o occupazione della memoria, possono essere inizialmente considerate di minor importanza e migliorate in fasi più avanzate dello sviluppo.

2 Struttura generale

Il plug-in in questo momento si trova in fase di riscrittura, sia per adattarlo alla nuova politica sulle *API* e del *Core* sia per creare una DHT che soddisfi i nuovi requisiti senza dover integrare il tutto nel vecchio codice, cosa che potrebbe costare di più rispetto ad una nuova stesura. In PariDHT un nodo è una tripletta <ID, IP, porta>, dove <IP, porta> corrisponderanno all'ID della macchina su cui l'utente si trova e l'ID identificherà l'account utente. Da questo si ricava che non esiste una corrispondenza biunivoca tra nodi e macchine fisiche, infatti in una singola macchina potrebbero essere in esecuzione due istanze diverse di PariPari, anche se questo risulta abbastanza inutile, se non dannoso, per l'utente stesso. In più, un utente potrà collegarsi con macchine diversa, ma non contemporaneamente. Per l'assegnazione degli ID, a differenza di Kademlia, viene usato l'algoritmo SHA-256[16][2]; in questo modo la probabilità di collisione[3] è estremamente bassa: gli ID infatti, sono composti da 256 bit[4]. Altra motivazione di questa scelta sta nel fatto che PariPari vuole offrire altri servizi oltre al file sharing, quindi serve uno spazio degli indirizzi maggiore rispetto a quello offerto da Kademlia. Quello che c'è di nuovo nella DHT sarà l'introduzione di:

- permessi associati alle chiavi/risorse;
- identità associate gli utenti (indipendenti dal nodo) che li identificheranno sulla rete;
- possibilità di rinnovo della risorsa;

[2]L'input dell'algoritmo è la chiave di una risorsa oppure le informazioni per la generazione dell'ID nel caso dei nodi.

[3]Si ha una collisione quando a due nodi connessi diversi oppure a risorse con chiavi diverse viene assegnato lo stesso ID.

[4]Con 256 bit è possibile generare $2^{256} \approx 1,16 \cdot 10^{77}$ valori diversi.

- possibilità di rimozione di una chiave prima che essa scada.

Le identità, permetteranno di associare le risorse salvate nella rete agli utenti della DHT, piuttosto che ai nodi. Non dimentichiamo però che anche le identità diventano risorse a tutti gli effetti (che come vedremo più avanti, saranno dotate di *autorefresh*, sezione 3.4), perché dovranno essere salvate nella rete permettendo all'utente, ad ogni suo accesso, di recuperare le informazioni necessarie per eseguire operazioni che richiedono permessi. I permessi invece, forniscono la possibilità a ciascun utente di impostare nel dettaglio le opzioni di condivisione per le proprie risorse salvate nella rete: possono essere condivise o meno con alcuni o tutti gli utenti della rete.

Permessi	**Sigla**
Scrittura	W (Write)
Lettura	R (Read)
Eliminazione	D (Delete)

Tabella 3.1: Tabella dei permessi

Come in Kademlia viene utilizzato l'operazione di XOR come metrica. Le risorse sono rappresentate tramite la coppia <K, V>, dove K e V non indicano esattamente chiave e valore, ma bensì un insieme di dati: Le informazioni presenti nella coppia sono:

- **K/ID**:
    1. K, la chiave della risorsa (appartiene allo stesso dominio degli ID dei nodi) identificata da parole chiavi (e.g. Robbie Williams Bodies);
    2. Insieme di permessi impostati per la chiave;
    3. Tempo di scadenza della chiave nella rete;
    4. Flag di autorefresh.
- **V**:
    1. ID dell'utente che ha eseguito un'operazione di store su quella chiave;
    2. Insieme di puntatori che puntano alla risorsa vera e propria, o in alcuni casi, la risorsa stessa. Se sulla chiave sono presenti dei permessi, questo valore sarà criptato.

K	V
Tags	ID utente 1 - puntatore a utente 1
(e.g. Madonna American Life)	ID utente 2 - puntatore a utente 2
Permessi	ID utente 3 - puntatore a utente 3
Tempo di durata nella rete	....
Flag autorefresh	ID utente n - puntatore a utente n

Tabella 3.2: Tabella dei contenuti della coppia <K, V>

Il nodo che pubblica una certa risorsa viene detto *possessore* della risorsa, i nodi che hanno ID più vicino alla risorsa vengono detti *responsabili potenziali* della risorsa mentre quelli che effettivamente detengono la risorsa sono detti *responsabili effettivi* (o semplicemente responsabili). Per svolgere le operazioni di una DHT, i nodi della rete comunicano tra loro con i seguenti messaggi:

- `STORE`. Viene utilizzato per richiedere al nodo destinatario di memorizzare una risorsa. Il destinatario può decidere se accettare o rifiutare la richiesta.

- `FIND NODE`. Richiede al destinatario di fornire la lista di nodi che conosce più vicini ad un certo ID.

- `SEARCH/FIND VALUE`. Chiede al destinatario di trasmettere la lista dei valori che hanno come chiave un certo ID. Se il nodo non è responsabile effetivo di quell'ID allora il risultato che fornisce è analogo a `FIND NODE`. In questo modo il mittente può continuare la ricerca dei valori grazie ai nodi che gli sono stati forniti.

- `PING`. Verifica se il nodo destinatario è ancora connesso alla rete.

In più, rispetto a Kademlia:

- `DELETE`. Al contrario della `STORE`, come si evince dal nome, richiede ai nodi responsabili di una chiave di "eliminarla".

- `SHARE`. Serve a condividere le informazioni per poter operare su risorse a cui sono stati applicati dei permessi. Questo fa sì che si possa estendere l'accesso a più di un utente.

2.1 Ricerca di un nodo e di una risorsa

Alla base del funzionamento di una DHT c'è la capacità di cercare un nodo nella rete. Si supponga infatti di dover salvare una risorsa. Si dovranno cercare i responsabili potenziali, cioè si dovranno trovare i nodi più vicini all'ID della risorsa. La ricerca di una risorsa avviene sostanzialmente allo stesso modo della ricerca di un nodo.
In PariDHT la procedura di ricerca dei nodi più vicini viene detta *look up*. L'equivalente di una *look up* in Kademlia restituisce un numero di nodi pari a k, cioè pari alla dimensione massima di un bucket. Sia k il numero di nodi vicini all'ID h che si vogliano ottenere, allora la procedura di *look up* si articola principalmente in tre passi:

1. Si crea una lista con i k nodi conosciuti più vicini ad h.
2. Si prendono i primi n nodi della lista, cioè i più vicini ad h, si contrassegnano e si invia a ciascuno di essi una `FIND NODE`(h). Ogni nodo quindi risponderà con i k nodi più vicini ad h che conosce.
3. Tra tutti i nuovi nodi ottenuti si inseriscono nella lista quelli opportuni[5]. Se nella lista ci sono ancora nodi non contrassegnati allora si ritorna al punto 2 altrimenti la procedura termina e si restituisce la lista.

[5]Non si inseriscono duplicati o nodi che non siano più vicini di quelli già presenti.

Capitolo 4

Permessi

Prima di parlare dei permessi che PariDHT vuole mettere a disposizione sulla rete, facciamo una panoramica sull'uso dei permessi in locale e su quelli offerti nella rete da altri servizi.

1 Permessi in locale

Per vedere cosa PariDHT vuole offrire a livello di rete, vediamo prima cosa i sistemi operativi offrono.

1.1 Unix(-like)

Nei sistemi Unix e Unix-like vi sono dei permessi di base o attributi per file e directory che sono applicabili a tre classi distinte:

- utente o proprietario
- gruppo
- altri

Vi sono poi ulteriori permessi che si applicano globalmente al file o alla directory. Questi permessi sono associati all'*inode*[17] che rappresenta i dati del file o della directory, e non al nome del file, per cui due o più collegamenti fisici allo stesso inode avranno necessariamente gli stessi permessi, anche se collocati in directory diverse.

Classi

A ciascun file e directory in un file system Unix viene assegnato un utente che ne è il proprietario (solitamente il suo creatore). Per tale utente si

applica la classe di permessi detta *utente*. A ciascun file e directory viene inoltre assegnato esattamente un gruppo di appartenenza (solitamente il gruppo principale del suo creatore). Agli utenti che non sono proprietari del file, ma che sono membri del gruppo di appartenenza del file si applica la classe di permessi detta *gruppo*. A tal proposito si ricorda che mentre un file appartiene esattamente ad un gruppo di utenti, un utente del sistema può essere membro di uno o più gruppi (di cui uno è detto principale mentre gli altri sono detti supplementari). Come si può notare nella figura 4.1, il proprietario è l'utente root e ai file elencati la classe gruppo assegnata è quella root.
Per tutti gli altri utenti che non ricadono nei due casi sopra elencati si applica invece la classe di permessi detta *altri*.

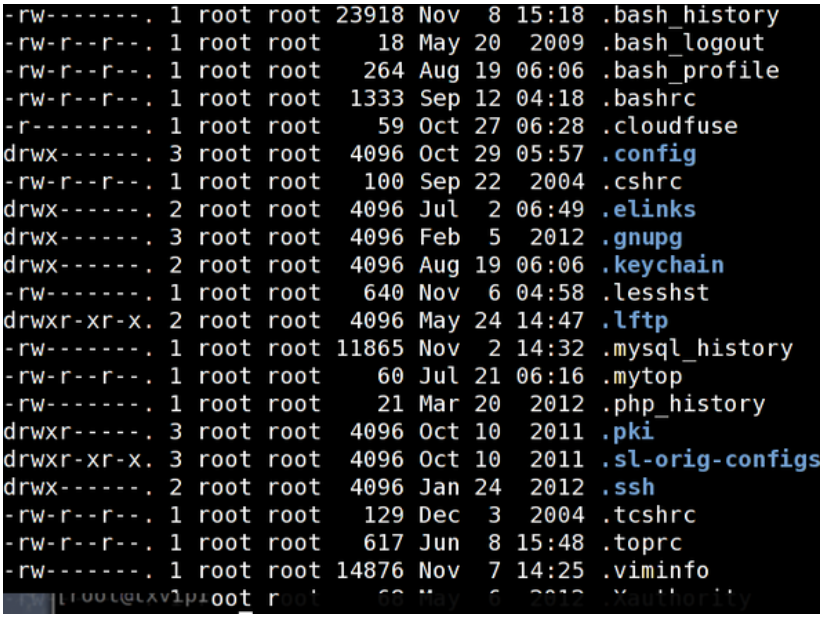

Figura 4.1: Risultato di un comando ls

Permessi di base

I permessi di base, che si applicano nell'ambito delle tre classi sopra elencate, sono:

r	permesso di lettura	Applicato ai file garantisce la possibilità di leggerne il contenuto di un file; applicato alle directory consente di elencare i nomi dei file e delle sottodirectory che contengono senza però visualizzare qualsiasi informazione (dimensione, permessi, ecc..)
w	permesso di scrittura	Applicato ai file permette di modificarne il contenuto; applicato alle directory, questo permesso garantisce l'abilità di modificare i file contenuti in esse. Questo include la creazione, la cancellazione e la possibilità di rominare i file.
x	permesso di esecuzione	Applicato ai file permette di eseguirli; applicato alle directory permette di attraversarle per accedere ai file ed alle sottodirectory in esse contenute (ma non di elencarne il contenuto, per il quale serve anche il permesso di lettura).
-	permesso non presente	Sostituisce r,w e x indicando che lo specifico permesso non è attivo.

Tabella 4.1: Permessi su sistemi Unix/Unix-like

Esempio:

- -rwxr-xr-x

	Lettura	Scrittura	Esecuzione
Utente	Sì	Sì	Sì
Altri	Sì	No	Sì
Gruppo	Sì	No	Sì

Tabella 4.2: Esempio

1.2 Non Unix-like

Nei sistemi operativi Microsoft e nelle varianti del DOS IBM tra cui DOS, MS-DOS, PC DOS, Windows 95, Windows 98, Windows 98 SE e Windows ME, non esiste una simile gestione dei permessi. È presente un attributo *read-only*, che può essere impostato o meno su un file da qualsiasi utente o programma, e quindi non impedisce di modificare/cancellare il file. Non ci sono le autorizzazioni in questi sistemi che impediscano all'utente di leggere un file. Il tutto, però, è cambiato con l'arrivo del nuovo file system di casa Microsoft, l'NTFS[18].

2 Permessi in rete

Visto che PariPari si è posto l'obiettivo di fornire agli utenti servizi quali storage distribuito e backup distribuito, confrontiamo che tipo di permessi offre un servizio quale **Dropbox** per esempio (lo stesso vale per **Google Drive**).

2.1 Dropbox e Google Drive

In questi anni Dropbox si è fatto avanti per mettere a disposizione un servizio grazie la quale gli utenti che lo utilizzano possono caricare i propri file in rete, anche se con uno spazio che dipende dal tipo di opzione scelta (free o a pagamento). Oltre a questo, un qualsiasi utente, se lo desidera, può condividere le directory[1] (quindi quelle di cui è proprietario dove ha pieni permessi) con altri utenti, sia che si sia iscritti o meno al servizio (ma questi alla fine dovranno effettuare l'iscrizione). Questa condivisione permette quindi agli utenti a cui viene estesa la visione delle varie directory di poter sia aggiungere altri file, sia di scrivere, leggere, cancellare e eseguire qualsiasi file.
Invece, Google Drive, disponibile da meno tempo agli utenti della rete, in aggiunta, offre la possibilità di condividere anche singoli file con qualsiasi utente (iscritto o meno, senza dover poi iscriversi).

[1] Non singoli file.

3 I permessi in PariDHT

Per potersi differenziare dagli altri servizi che si basano sulle DHT, PariDHT si prefigge di aumentare la sicurezza delle risorse condivise nella rete, mettendo a disposizione ai plugin, che fanno uso delle operazioni fondamentali di una DHT, un sistema di permessi simile a quello dei sistemi operativi Unix-like. A differenza del permesso di esecuzione, e della semplice condivisione privata o aperta ad altre persone, PariDHT offre i seguenti permessi:

- READ
- WRITE
- DELETE

I plugin imposteranno questi permessi ai valori di **Restricted** o **Unrestricted**, ma con significato diverso a seconda del permesso.

	Restricted	Unrestricted
READ	Limitata	Chiunque può leggere i valori
WRITE	Limitata	Chiunque può scrivere/aggiornare valori
DELETE	Limitata	Nessuno può cancellare l'intera chiave

Tabella 4.3: Significato attivazione dei permessi

Come si può notare dalla tabella, il permesso di *delete* con valore *Unrestricted* fa si che la chiave possa essere rimossa solo quando scade (sempre se non è stato attivato il meccanismo di *autorefresh*3.4).

Comportamento dei permessi

Read e *Delete* agiscono sull'intera chiave, ovvero se un nodo ha il permesso di leggere potrà leggere l'intero contenuto della chiave, ovvero **V**; analogamente, se un nodo ha il permesso di cancellare, potrà eliminare la chiave e tutti i valori associati dalla DHT.
Write invece non garantisce **mai** il permesso di sovrascrivere i valori impostati da altri nodi, ma solo di aggiungere il proprio valore ad una chiave o equivalentemente di aggiornarlo.

Condivisione delle risorse

Pensando ancora a come funziona Dropbox o Google Drive, quando un utente condivide delle risorse con altri membri del servizio, fa si che si crei una sorta di *gruppo*, che può operare a pieno titolo su qualsiasi risorsa condivisa. Dal punto di vista dei permessi, in PariDHT questo concetto è simile, ma i permessi che ogni membro del gruppo ha, possono non essere equivalenti: una chiave può avere restrizioni su una o più azioni (R, W, D) ma non è mai specificato quali siano i nodi che possono eseguire tali azioni. La condivisione delle risorse ha ovviamente senso solo fra nodi che appartengono alla DHT, per cui tramite la primitiva definita nella sezione 2, un nodo può distribuire ad altri nodi il set di permessi (o un suo sottoinsieme) che una risorsa possiede.

3.1 Ricerca

Analizziamo come avviene la ricerca di una risorsa nella rete.

1. Dopo aver ottenuto l'hash della chiave che si vuole cercare (questo per poter utilizzare le proprietà della metrica XOR), cerco i k nodi che hanno ID più vicino all'hash della chiave, eliminando man mano parte della rete per arrivare alla fine al nodo più vicino in un numero logaritmico di passi;

2. se i nodi custodiscono insiemi di puntatori, provvedo a contattare i nodi puntati per ottenere la risorsa, altrimenti il nodo si fa restituire la risorsa direttamente da uno dei nodi contattati; la ricerca fallisce se nessun nodo ha subito operazioni di `STORE` in precedenza per la chiave ricercata.

Da notare che i nodi che ospitano effettivamente le risorse possono avere ID che non hanno **alcuna relazione** con l'hash delle risorse che ospitano. Questo perché la relazione fra ID del nodo ed hash della chiave riguarda solo l'indicizzazione della risorsa stessa, e non il luogo in cui questa risorsa viene materialmente mantenuta. La relazione ovviamente vale nel caso, però, in cui i nodi nella DHT ospitino direttamente la risorsa (e non dei puntatori a nodi che ospitino la risorsa).
Rispetto alla precedente versione della DHT di PariPari non cambia il procedimento di questa operazione. I messaggi scambiati tra i nodi rimangono gli stessi e quindi non abbiamo overhead.

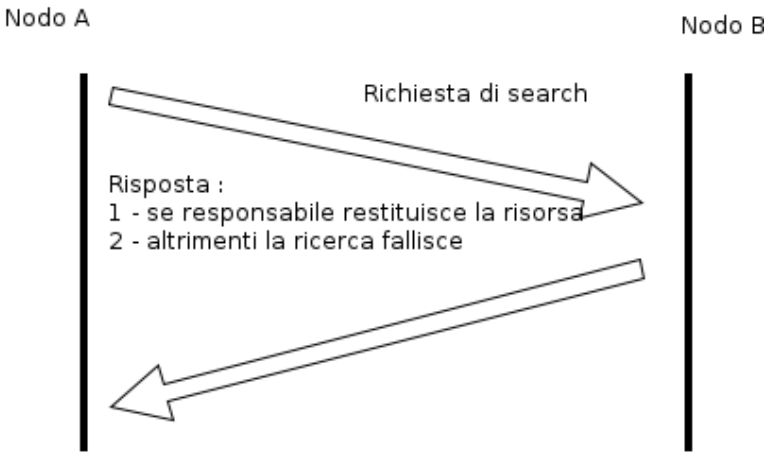

Figura 4.2: Schema di funzionamento della `SEARCH`

3.2 Salvataggio

Con l'introduzione dei permessi, il meccanismo di questa operazione fondamentale cambia rispetto a come è definito nelle tesi di Davide Zanin[19] e Claudio Giacometti[24], dove semplicemente si inviava un messaggio di `STORE` ad un insieme di nodi, senza verificare o meno condizioni che ora diventano fondamentali. Analizziamo come avviene un salvaggio di una risorsa nello specifico.

1. Quando un nodo vuole effettuare una `STORE` deve dichiarare il set di permessi che intende applicare alla chiave che vuole salvare;
2. Utilizzando la *lookup*, si ricava un insieme di nodi Y, di cardinalità n, più vicini a k come descritto nella sezione 2.1;
3. A questi nodi si richiede di poter eseguire il salvataggio di una risorsa con chiave **K**;
4. I nodi a cui arriva la richiesta controllano se è presente la chiave o meno;
5. Se i nodi rispondono che la chiave è libera:
    - A meno di fallimenti[2] la coppia <K,V> viene salvata interamente con ID dell'identità dell'utente del nodo ad identificare il valore salvato.

[2]La `STORE` fallisce se il nodo destinatario è disconnesso

6. Se invece la chiave risulta essere già presente:

   - se la chiave è priva di permessi, la `STORE` non sarà rifiutata, sempre a meno dei fallimenti già citati, e potranno verificarsi due situazioni in base all'ID dell'identità che ha richiesto l'operazione:

     – **V** verrà aggiunto alla lista di valori associati alla chiave **K** se non è presente una risorsa con quell'ID;

     – **V** sovrascriverà il valore già presente se esiste già una risorsa con quel determinato ID.

   - altrimenti:

     – se ci sono permessi attivi sulla chiave, tranne quello di *Write*, per poter andare a buon fine, i permessi scelti per quella chiave dovranno combaciare con quelli che la chiave già salvata possiede;

     – se il permesso di *Write* è attivo, i nodi a cui è arrivato il messaggio di `STORE` sottoporanno il nodo richiedente a delle challenge, con la quale i nodi verificano chi può effettivamente aggiungere/sovrascrivere valori per quella determinata chiave (come per il punto precedente). Se il nodo vede che le risposte alle challenge sono corrette, l'operazione potrà andare a buon fine. Le challenge sono decise dai nodi proprietari e sono assimilabili concettualmente alla conoscenza di una password.

Come si può vedere, la `STORE`, in alcuni casi, si presenta più complessa di quanto fosse in assenza del nuovo sistema di permessi. Questo, quindi, fa sì che tra i nodi vengano scambiati più messaggi, il che porta alla presenza di overhead. Nell'immagine 4.3, è messo in evidenza la transizione per l'operazione di *STORE* per la precendete versione della DHT, che poteva andare a buon fine o meno in base al *load factor*[25][3]. Come si può notare il massimo numero di messaggi scambiati era 4.

[3]Il load factor è dato dal numero di chiavi e valori che il nodo già memorizza.

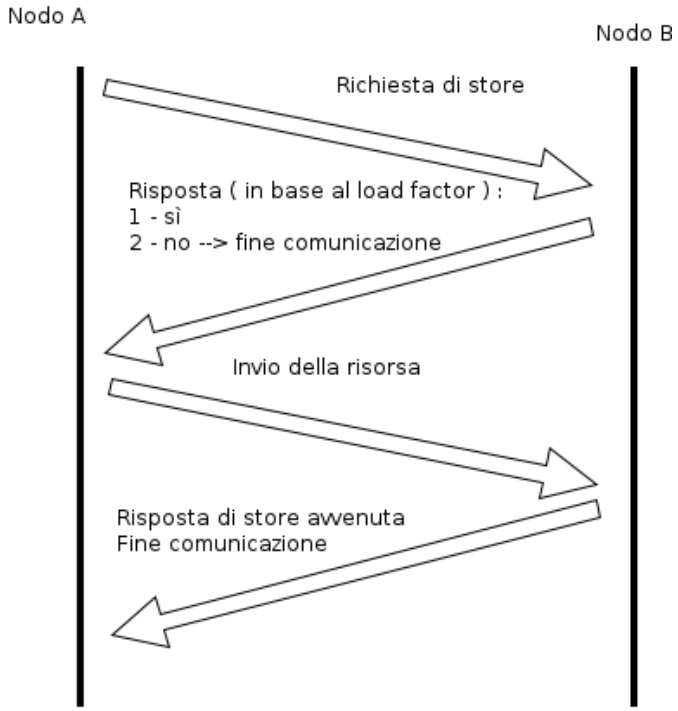

Figura 4.3: Scambio di messaggi tra due nodi per la versione precedente della `STORE`

Ora questo non è più così perchè i nodi potranno salvare nel disco le risorse con chiavi che subiscono poche operazioni di salvataggio.
Paragonando 4.4 con 4.3 si può notare che differiscono solo nelle risposte che un nodo fornisce, perché il numero totale di messaggio scambiati non varia, quindi non abbiamo overhead. Questo non accade, però, nella situazione illustrata in figura 4.5, dove il numero di messaggi risulta essere più variabile e con un valore minimo di 6 (nel caso si possa eseguire l'operazione). Se la chiave della risorsa che si desidera salvare è presente nella rete e l'attributo del permesso di *Write* è restricted, il nodo, come già accennato, sarà sottoposto ad una serie di n challenge con $1 \leq n \leq m$, dove m potrà essere al massimo 3, perchè si vuole offrire sì più sicurezza, ma come accennato nella sezione 1, bisogna tener conto anche delle prestazioni della rete.

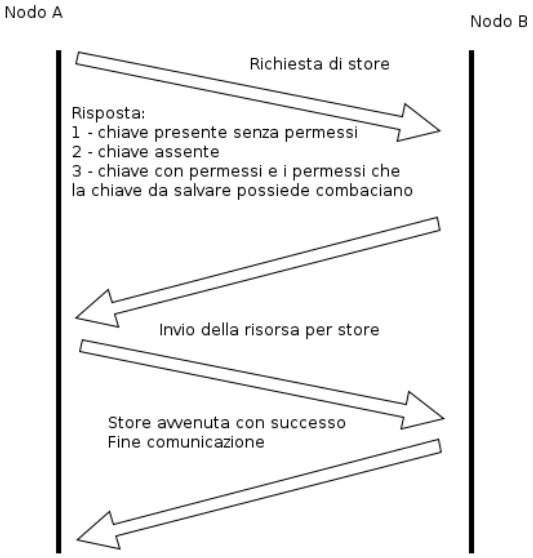

Figura 4.4:

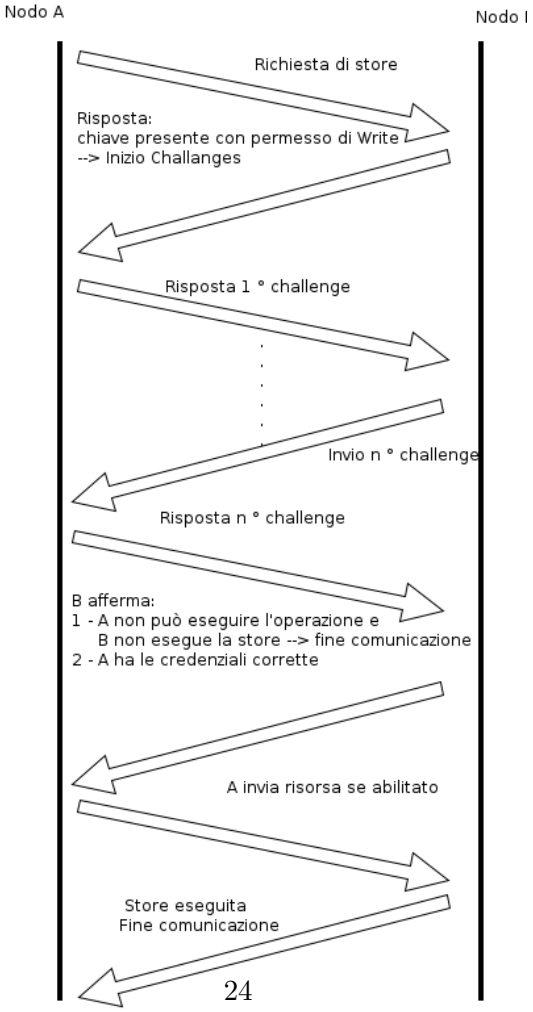

Figura 4.5:

Figura 4.6: Scambio di messaggi tra due nodi

3.3 Rimozione

Essendo un'operazione nuova, non si ha un confronto con un metodo della precedente DHT in merito allo scambio di messaggi e all'overhead. Analizziamola nel dettaglio. L'operazione di `DELETE` che PariDHT offre, è atta a permettere all'utente di rimuovere le risorse associate ad una specifica chiave prima che sia rimossa automaticamente allo scadere del suo tempo di vita o bloccando il meccanismo di autorefresh.

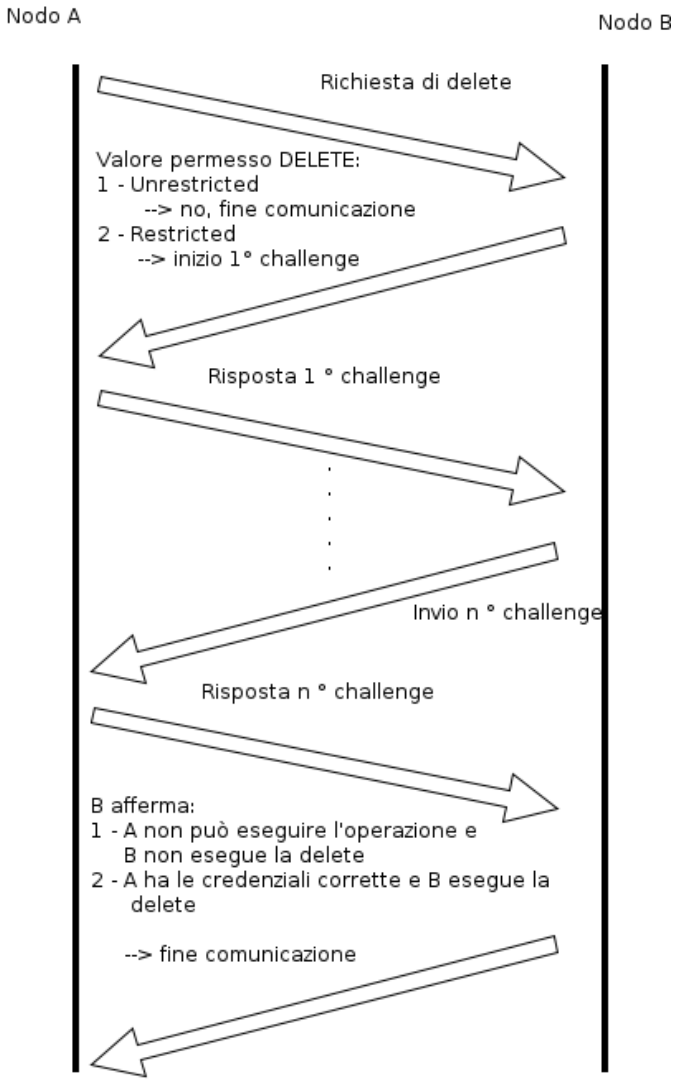

Figura 4.7: Scambio di messaggi tra due nodi per una `DELETE`

La richiesta di rimozione dovrà essere inviata a tutti i nodi responsabili della chiave per cui un nodo esegue l'operazione. Un nodo non avrà mai la certezza che la chiave sia stata rimossa completamente, questo perché può accadere che uno dei responsabili si sia disconnesso prima che il nodo in questione abbia inviato la richiesta.

3.4 Approfondimento su challenge, permessi e scadenza delle chiavi

Tempo di scadenza e autorefresh

Ogni chiave avrà un tempo di vita massimo, oltre il quale i nodi che stanno ospitando (puntatori al)le risorse scadute saranno autorizzati a *dimenticarsene*. L'esistenza della data di scadenza per le chiavi (e la presenza di *churn*[4]) impone un meccanismo di refresh delle stesse; in Kademlia il nodo che pubblica una chiave è il responsabile del refresh della stessa (ad intervalli di un'ora). Per servizi che richiedano la presenza online del nodo che ha effettuato lo store per poter funzionare (come appunto il file sharing) possiamo utilizzare la stessa strategia. Per servizi però che si basano sull'assenza dalla rete del nodo *padre*, come ad esempio backup distribuito, abbiamo bisogno di un meccanismo di autorefresh.
Per le chiavi, quindi, è possibile scegliere se esse possono avere una durata prestabilita oppure rimanere per un tempo indeterminato nella rete (sempre a meno di rimozioni). In sostanza, si tratta di un flag che i client potranno impostare o meno, a seconda delle risorse salvate. Se una chiave è contrassegnata per l'autorefresh, sarà il nodo ospite a doversi preoccupare di ripubblicare il contenuto sulla rete.

Permessi e crittografia

Fino ad ora si è parlato di permessi associati alle chiavi e a valori che questi possono assumere. Non è tutto. Se fossero solamente dei flag, i nodi che ospitano le risorse potrebbero tranquillamente leggere, scrivere e/o rimuovere le risorse ospitate. Allora, cosa impedisce a questi nodi di poter fare ciò?
L'attivazione di un permesso, richiede l'uso di coppie di chiavi a crittografia asimmetrica[27]. Grazie a questo, un nodo, prima di effettuare un'operazione di `STORE` di una chiave, ad esempio, con il permesso di lettura attivo, dovrà criptare il valore associato con la chiave pubblica, in

[4]Con *churn* si intende la disconnessione improvvisa di un nodo dalla rete che, agli occhi degli altri partecipanti, risulta nell'impossibilità di comunicare con il nodo disconnesso

modo che solo chi conosce quella privata possa leggere, o meglio, interpretare correttamente ciò che il valore significa.
Lo stesso vale per gli altri due permessi.

Challenge

Cosa sono queste *challenge* citate precedentemente? La definizione corretta, secondo il protocollo Zero-Knowledge (che PariDHT seguirà), sono un metodo interattivo per verificare che un'affermazione sia vera. Principalemente sono dei meccanismi di protezione per i nodi che ospitano risorse per assicurarsi che un nodo malevolo non richieda operazioni di `STORE` e `DELETE` per chiavi che non gli competono. Saranno inoltre usate come meccanismo per far risparmiare banda ai nodi ospite (non dovranno cioè trasferire chiavi della DHT ad altri nodi a meno che questi non siano abilitati). Vediamo un esempio teorico/astratto per spiegare l'idea alla base di queste challenge.
Alice, che deve provare un'affermazione, e Bob, che deve verificarla, sono i protagonisti del nostro esempio.
Alice ha scoperto la parola segreta usata per aprire una porta magica in una grotta. La grotta è a forma di cerchio, con ingresso su un lato e la porta magica che blocca il lato opposto. Bob dice che pagherà profumatamente per il segreto conosciuto da Alice, ma non fino a quando non è sicuro che lei lo sappia davvero. Alice, per suo tornaconto, vuole avere i soldi prima di poter rivelare la parola magica. Allora i due per venirsi incontro, decido di escogitare un piano per far si che Alice dimostri di conoscere la parola senza dirla a Bob.

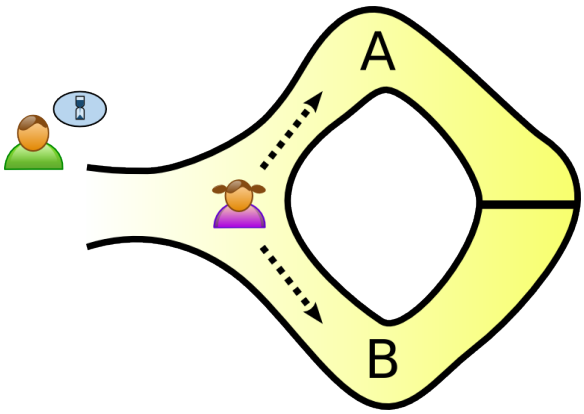

Figura 4.8: Situazione iniziale challenge

Per prima cosa, Alice si reca all'interno della grotta, prima del bivio, mentre Bob attende fuori. I percorsi di sinistra e destra dall'ingresso sono stati nominati A e B, come da loro deciso. Alice, per arrivare alla porta magica, può prendere a caso uno tra i due percorsi. Poi, Bob entra nella grotta e grida il nome del percorso dalla quale Alice deve uscire, scelto sempre a caso. Se Alice conoscesse la parola magica, per lei sarebbe facile. Aprirebbe la porta, se fosse necessario, e ritornerebbe lungo la via scelta. Bob, però, non può sapere che percorso abbia scelto Alice per recarsi al centro della grotta.

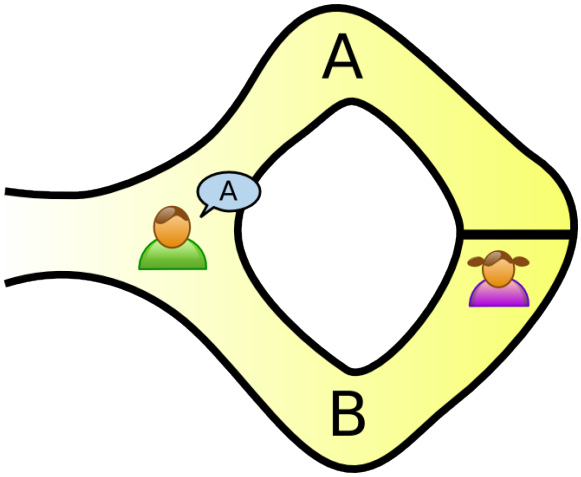

Figura 4.9: Scelta del percorso da parte di Bob

Immaginiamo che Alice non fosse a conoscenza della parola magica. Alice sarebbe in grado di tornare all'ingresso tramite il percorso scelto da Bob, solo se questo fosse stato scelto all'inizio da Alice per recarsi al centro. Dal momento che Bob ha scelto a caso tra A e B, Alice avrebbe avuto il 50% di possibilità di indovinare. Se dovessero ripetere questo trucco molte volte, ad esempio 20 volte di seguito, la sua possibilità di successo di anticipare le scelte di Bob diventano davvero piccole. Quindi, se Alice dovesse uscire molte volte dalla via scelta da Bob, allora potrebbe concludere che è molto probabile che lei conosca la parola.

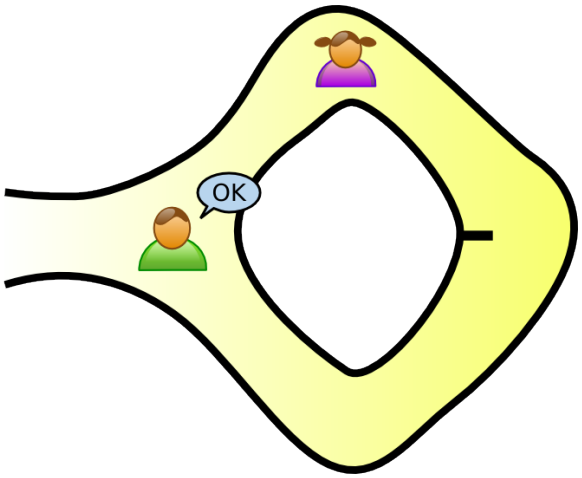

Figura 4.10: Alice prova a Bob di conoscere il segreto

Una challenge Zero-Knowledge deve soddisfare tre proprietà:

1. **Completezza**: se l'affermazione è vera, un dimostratore onesto potrà convincere del fatto un verificatore onesto (cioè chi segue esattamente il protocollo).

2. **Correttezza**: se l'affermazione è falsa, nessun baro potrà convincere l'onesto verificatore che essa è vera, se non con qualche piccola probabilità.

3. **Zero-Knowledge**: se l'affermazione è vera, nessun imbroglione potrà sapere altro oltre a tale informazione.
   Le dimostrazioni a conoscenza zero non sono dimostrazioni in senso matematico poiché c'è sempre una piccola probabilità[5] che un dimostratore imbroglione riesca a convincere un verificatore di un'affermazione falsa. In altre parole, questi tipi di algoritmi sono probabilistici e non deterministici. Tuttavia, ci sono tecniche per ridurre l'errore fino a valori trascurabili. Nella pratica tutti i problemi **NP-completi**[29] si prestano bene ad essere usati in questo protocollo, come ad esempio, dato un grafo, trovarne un ciclo Hamiltoniano.

Permessi e challenge nei servizi

- **File Sharing**. I client imposteranno tutti e 3 i permessi R, W, D ad *Unrestricted* e i nodi che ospitano le risorse non chiederanno challenge di alcun tipo, perché sarebbe inutile. Il servizio in questo caso è come descritto in Kademlia.

- **Identità**. Salvate in chiavi con R = *Unrestricted*, mentre W e D saranno *Restricted*. Per poter eliminare la chiave dalla rete, il nodo che ospita la chiave dovrà sottoporre il nodo che ha inoltrato la richiesta a delle challenge, e solo chi ha eseguito il salvataggio sarà autorizzato a procedere.

- **Backup distribuito**. I client setteranno i 3 permessi al valore *Restricted*. Qui la procedura di challenge verrà sottoposta sia a chi tenta di scrivere valori, sia per chi tenta di rimuoverli.

[5] Chiamato *soundness error*.

Capitolo 5

Conclusioni e sviluppi presenti e futuri

In questo elaborato si è discusso dell'introduzione dei permessi legati alle chiavi che vengono salvate nella rete, confrontandolo con quanto messo a disposizione dai sistemi operativi e analizzando il lavoro che i nodi devono fare quando eseguono le varie operazioni. Non sono stati affrontati i problemi che la rete si ritroverà ad affrontare perché, non avendo ancora un plugin funzionante in grado di sostenere la rete, non si possono avere dei riscontri in merito, ma le previsioni sono ottimistiche e si cercherà di implementare il tutto per il nuovo anno. Quello che si può dire è che PariDHT non vuole che la sua rete sia vulnerabile ed essere in grado di offrire i servizi descritti potrebbe farla diventare una delle prime DHT con queste caratteristiche, pur continuando a soffrire (come tutte le DHT) dei problemi derivanti dai sybilAttack. Certamente, con lo sviluppo della DHT descritta, il traffico presente nella rete sarà maggiore rispetto a prima, ma i risultati che il tool di benchmarking, sviluppato da Claudio Giacometti, potrebbe fornirci, permetterà di studiare come provare a migliorare queste prestazioni.

Bibliografia

[1] ***Peer to Peer***
http://en.wikipedia.org/wiki/Peer-to-peer

[2] ***BitTorrent***
http://www.bittorrent.com/intl/it/

[3] ***eMule***
http://www.emule-project.net/

[4] ***Skype***
http://www.skype.com/

[5] ***Tor***
https://www.torproject.org/

[6] *Wiki PariPari*

[7] ***DHT***
http://it.wikipedia.org/wiki/Tabella_di_hash_distribuita

[8] ***Petar Maymounkov, David Mazieres.***
Kademlia: Information System Based on the XOR Metric.

[9] ***Napster***
http://it.wikipedia.org/wiki/Napster

[10] ***Gnutella***
http://en.wikipedia.org/wiki/Gnutella

[11] ***Robert Morris, David Karger, Frans Kaashoek, Hari Balakrishnan.***
Chord: a scalable peer-to-peer lookup service for Internet applications.

[12] ***The Open-Closed principle - C++ Paper***

[13] ***Oracle javadoc***
http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html

[14] ***Wiki PariDHT***
http://www.paripari.it/mediawiki/index.php/DHT_en

[15] ***Extreme Programming***
http://en.wikipedia.org/wiki/Extreme_Programming

[16] ***Secure Hash Algorithm-2***
http://en.wikipedia.org/wiki/SHA-2

[17] ***Inode***
http://it.wikipedia.org/wiki/Inode

[18] ***NTFS***
http://en.wikipedia.org/wiki/NTFS

[19] ***PariDHT: Gestione dei contatti***
http://tesi.cab.unipd.it/33098/1/davide_zanin_580496.pdf

[20] ***PariTalk***
http://tesi.cab.unipd.it/40190/

[21] ***PariIM***
http://tesi.cab.unipd.it/26404/

[22] ***PariMulo***
http://tesi.cab.unipd.it/28311/1/tesiMattiaFrancesco604632.pdf

[23] ***PariTorrent***
http://tesi.cab.unipd.it/37652/1/Pozzobon_-_PariTorrent_Refactoring.pdf

[24] ***PariDHT: Strumento di profiling per reti basate su DHT***
http://tesi.cab.unipd.it/39646/1/Tesi-Claudio.pdf

[25] ***Load Factor***
http://en.wikipedia.org/wiki/Hashtable

[26] ***Flooding***
http://it.wikipedia.org/wiki/Flooding

[27] ***Crittografia Asimmetrica***
http://it.wikipedia.org/wiki/Crittografia_asimmetrica

[28] ***Zero-Knowledge Challenge***
http://en.wikipedia.org/wiki/Zero-knowledge_proof

[29] ***Problemi NP-Completi***
http://en.wikipedia.org/wiki/NP-complete

[30] ***Pidgin***
http://it.wikipedia.org/wiki/Pidgin_(software)

[31] ***BitTorrent***
http://it.wikipedia.org/wiki/BitTorrent_(software)

[32] μ***Torrent***
http://it.wikipedia.org/wiki/%CE%9CTorrent

[33] ***eMule***
http://it.wikipedia.org/wiki/EMule

Elenco delle figure

Elenco delle tabelle