UNIVERSITÀ DEGLI STUDI DI PADOVA
Dipartimento Di Ingegneria Dell'Informazione
Corso Di Laurea In Ingegneria Informatica

PariBench: Strumento di profiling per reti basate su DHT

RELATORE: Prof. Enoch Peserico Stecchini Negri De Salvi
CORRELATORE: Dott. Michele Bonazza

LAUREANDO: Giacometti Claudio

Anno Accademico 2011/2012

Alla mia famiglia per il loro supporto
e a Jessica per il suo affetto.

Sommario

Negli ultimi anni l'aumento delle connessioni a banda larga ha portato alla diffusione delle reti peer-to-peer[1] (P2P, d'ora in avanti). Applicazioni quali BitTorrent[2], eMule[3] e Skype[4], 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. PariPari avrebbe vita breve senza l'efficienza della propria rete, per questo abbiamo sviluppato un tool che possa provare a misurarne l'efficacia.

Indice

Introduzione

In questo elaborato verrà inizialmente introdotto il progetto PariPari[5] esponendo brevemente obiettivi, architettura e organizzazione.
Nel capitolo 2, verrà presentata l'evoluzione delle reti P2P fino ad arrivare alle odierne DHT[6]. Successivamente verrà descritta la rete Kademlia, presa come riferimento per lo sviluppo di PariDHT, la DHT di PariPari. L'ultima parte del capitolo è dedicata a PariDHT, in particolare verranno sottolineate le differenze presenti rispetto a Kademlia e descritti gli algoritmi di salvataggio e ricerca di una risorsa.
Dopo l'introduzione di PariPari e PariDHT passiamo ad una breve infarinatura di Benchmarking che fa da preludio a PariProfiling ovvero un tool insito in PariDHT in grado tracciare i dati per valutare l'efficienza della rete. Concludendo poi con la definizione di una simulazione per verificare quanto il tool sviluppato sia buono e dove poterlo migliorare.

1 PariPari

PariPari è una rete P2P serverless, la quale si promette di rendere disponibili tutti i servizi che al giorno d'oggi sono erogati da reti separate. Con il solo software di PariPari sarà possibile effettuare videoconferenze, dialogare tramite le varie chat, o utilizzare le reti di filesharing come eDoney, Kademlia[7] o BitTorrent. Inoltre saranno disponibili servizi quali DistributedStorage, DHT, DBMS e quant'altro è possibile ottenere tramite Internet.

1.1 Architettura

PariPari presenta una struttura modulare: 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; 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* : Gestisce 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, Torrent, DNS, distributed storage, distributed backup, VOIP, IM, IRC, WEB, NTP. Alcuni di questi sono già pronti altri invece in via di sviluppo, tuttavia il numero di questi plugin è destinato a crescere, in quanto le applicazioni che possiamo creare sopra a questa struttura sono veramente molte.

2 Reti P2P

Una rete peer to peer si contraddistingue da ogni altra per il fatto che ogni nodo (o peer) è uguale ad ogni altro presente nella rete. Non ci sono server o client ma tutti svolgono le mansioni di cliente e servente. Gli esempi più famosi di queste reti sono nati per la condivisione di file (inizialmente file musicali). Si possono individuare le seguenti architetture:

Modello Centralizzato: es. Napster[8]
Esso utilizzava dei server per mettere in contatto i vari nodi ma non interveniva nel traserimento effettivo del file. Esso è un sistema ibrido nel quale la parte iniziale, ovvero la ricerca da parte di un nodo, è fatto in modo client-server, mentre la fase successiva, di download, tra i due nodi è propriamente P2P.
Il problema fondamentale di Napster, e quindi di questo modello di rete P2P, è la scalabilità. Infatti più la rete cresce più era il carico sui vari server.

Modello distribuito non strutturato: es. Gnutella[9]
A differenza della precedente, l'accesso alla rete non avviene contattando un'entità superiore e anche la ricerca è distribuita, cioè ogni nodo risponde solo dei file che condivide. L'assenza del server implica la necessità di un algoritmo di instradamento per localizzare nodi e risorse. Data la natura non strutturata le ricerche avvengono in broadcast. Uno dei suoi principali problemi era evitare l'inondazione della rete di messaggi dovuti alle ricerche fatte in broadcast. Per questo la circolazione infinita di un pacchetto era prevenuta con il campo TTL (time to live) in cui era impostato il numero massimo di nodi che poteva attraversare il messaggio; inoltre un nodo memorizzava temporaneamente l'ID dei messaggi che passavano attraverso lui e scartava eventuali duplicati. Non conoscendo l'intero sistema ma soltanto alcuni dei nodi vicini non era garantito il determinismo di una ricerca.

Modello distribuito strutturato: es Chord[14], Kademlia
In quest'ultimo caso la topologia delle rete è controllata ed esiste un protocollo di comunicazione che instrada i pacchetti verso i nodi che dovrebbero avere la risorsa cercata. Questo è ottenuto tramite una DHT che abbia i requisiti di:

- *Decentralizzazione*: i nodi formano collettivamente il sistema senza bisogno di server
- *Scalabilità*: il sistema supporta milioni di nodi senza perdere in efficienza

- *Tolleranza ai guasti*: il sistema supporta nodi che entrano e lasciano la rete o sono soggetti a malfunzionamenti con elevata frequenza

2.1 DHT

Le DHT (Distributed Hash Table) sono una classe di sistemi distribuiti decentralizzati che partizionano l'appartenenza di un set di chiavi tra i nodi partecipanti. Ad ogni nodo che si connette alla rete viene assegnato un identificatore (ID): molto spesso accade che sia il nodo che entra nella rete si auto-assegni un ID scelto da uno spazio molto grande. Solitamente le implementazioni delle DHT utilizzano un ID di 160 bit creato tramite una funzione di hash[10] non reversibile (ad esempio, SHA-1). Bisogna poi definire una metrica nello spazio degli indirizzi: questa permetterà di calcolare la distanza fra due nodi qualsiasi della rete.
Le DHT servono a condividere risorse. A queste viene assegnato un globally unique identifier (GUID), il quale è solitamente ricavato a partire dalla risorsa stessa, o da alcuni suoi parametri, per mezzo di hash sicuro. L'utilizzo di un hash sicuro permette alla risorsa di autocertificarsi, più precisamente: i client che ricevono una risorsa possono verificare la sua autenticità, confrontando l'hash ricevuto con quello calcolato a partire dalla risorsa stessa. Per il corretto funzionamento di questa tecnica l'hash della risorsa non deve variare nel tempo; ciò comporta che la risorsa in questione deve essere immutabile. I nodi più vicini ad una risorsa sono i suoi responsabili, il che significa che sono a conoscenza di maggiori informazioni su di essa, come ad esempio da chi è stata condivisa. Per una maggiore disponibilità solitamente vengono create delle repliche degli oggetti, in modo tale che questi siano reperibili anche quando alcuni dei nodi, che ne hanno la competenza, non sono più connessi alla rete. E' necessario poi la realizzazione di un algoritmo di instradamento, il quale ha il compito di localizzare nodi e risorse nella rete. La procedura di instradamento è implementata utilizzando un algoritmo greedy che iterativamente diminuisce la distanza fra il nodo di destinazione e l'i-esimo nodo intermedio al fine di raggiungere ciò che si sta cercando. Le funzioni principali che deve compiere un algoritmo di instradamento di questo tipo sono le seguenti:

1. un client che vuole ricercare una risorsa dovrà inviare una richiesta, comprensiva del GUID cercato, e utilizzando l'algoritmo di instradamento la farà pervenire ad uno dei nodi responsabili;

2. un nodo che desidera condividere un nuovo servizio gli assegna un GUID e lo annuncia alla rete, la quale si assicurerà di renderlo disponibile agli altri partecipanti;

3. quando un client richiede la rimozione di una risorsa, da lui messa a disposizione, l'algoritmo di instradamento deve essere in grado di soddisfare tale richiesta, non necessariamente deve avvenire subito;

4. i nodi possono entrare e uscire a piacimento dalla rete. Quando un nodo entra gli vengono inviate le informazioni di cui deve essere il responsabile. Quando un nodo esce (volontariamente o non) la sua responsabilità viene distribuita tra gli altri nodi.

2.2 Kademlia

Kademlia, da cui PariDHT trae spunto, è l'esempio più famoso di DHT. Questo protocollo è stato ideato da Petar Maymounkov e David Mazières e usa come metrica la funzione XOR, si ha, quindi, che d(x, y) = x $\oplus$ y. Questa metrica fornisce una topologia ad 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. In Kademlia ogni nodo ha
ID = bm−1 , bm−2 , . . . , b1 , b0 , con m = 128
quindi 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−1] contiene nodi che hanno distanza d con 2i <d <2i+1, cioè include i nodi che hanno ID con gli stessi valori nei bit bm−1 , bm−2 , . . . , bi+1 , e un valore diverso nel bit bi. 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.

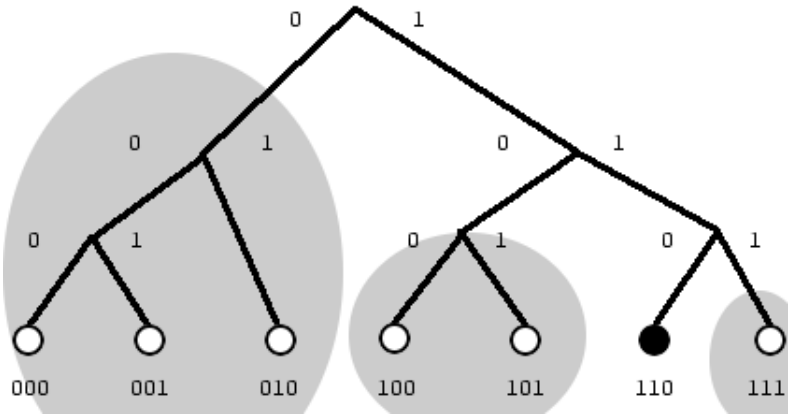

Figura 1: Esempio di partizionamento della rete

2.3 PariDHT

Nella realizzazione del plug-in PariDHT si è cercato di rispettare il più possibile le seguenti specifiche:

Funzionalità. L'obiettivo principale è fornire un modulo DHT che dia la possibilità agli altri plug-in di pubblicare e reperire risorse dalla rete.

Espandibilità. Il plug-in deve essere il più modulare possibile in modo da rendere più semplici eventuali espansioni future.

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 per commentare il codice e della wiki per spiegare come gli altri plug-in possono utilizzare PariDHT.

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.

2.3.1 Struttura generale

In PariDHT un nodo è definito da una tripletta <ID, IP, porta >. Da questo si evince 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 è abbastanza inutile, se non dannoso, per l'utente stesso. Per l'assegnazione degli ID viene usato l'algoritmo SHA-256; in questo modo la probabilità di collisione è estremamente bassa: gli ID infatti, sono composti da 256 bit. Come in Kademlia viene utilizzato l'operazione di XOR come metrica. Le risorse sono rappresentate con la tripletta ¡Key (o ID), value, dataDeath¿, in cui ciascun campo ha il seguente significato:

ID. Rappresenta la chiave della risorsa ed appartiene allo stesso dominio degli ID dei nodi.

value. E' il valore, di qualsiasi tipo, che si vuole memorizzare nella rete.

dataDeath. E' il momento oltre il quale una la risorsa va considerata scaduta (o morta).

Il nodo che pubblica una certa risorsa viene chiamato possessore della stessa, i nodi con 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 sulla base del proprio load factor (indice di memoria utilizzata).
- **SEARCH**. Chiede al destinatario la lista dei valori delle risorse che hanno come chiave un certo ID. Se il nodo non è responsabile effettivo 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.
- **FIND NODE**. Richiede al destinatario di fornire la lista di nodi che conosce più vicini ad un certo ID.
- **PING**. Verifica se il nodo destinatario è ancora connesso alla rete.

2.3.2 Ricerca di nodi e risorse

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, mentre in PariDHT il valore può essere specificato per ogni ricerca e quando non indicato viene usato quello di deafult, cioè k/2. 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 ALPHA 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 (i più vicini). Se nella lista ci sono ancora nodi non contrassegnati allora si ritorna al punto 2 altrimenti la procedura termina e si restituisce la lista.

Il parametro ALPHA rappresenta il numero massimo di richieste pendenti e viene utilizzato per limitare il traffico sulla rete.
La procedura di ricerca di una risorsa è molto simile, con la differenza che ai nodi vengono spediti dei messaggi di tipo SEARCH al posto di FIND NODE.

2.3.3 Salvattaggio di una risorsa

Anche per il salvataggio di una risorsa la procedura di look up riveste un ruolo fondamentale. Principalmente per la memorizzazione di una risorsa di ID h sono richiesti i seguenti passaggi:

- Utilizzando la procedura di look up si ricavano i k nodi più vicini ad h.
- Si selezionano i RESPONSIBILITY NODES nodi pi‘ vicini ad h e si invia una STORE(h) a ciascuno di essi.

Come si può capire dal secondo punto, il salvataggio di una risorsa viene eseguito su più nodi. Inoltre se una delle STORE dovesse fallire, deve essere scelto un altro nodo in modo tale che il numero totale di STORE eseguite con successo sia esattamente RESPONSIBILITY NODES. Di default questo parametro è impostato a 3 ma può essere facilmente modificato attraverso il file di configurazione. Tale replica viene eseguita per permettere di trovare la risorsa anche nel caso in cui uno o più dei nodi responsabili si disconettesse.

3 Benchmarking

Il benchmark si propone di quantificare le performance del sistema o di uno dei suoi componenti. Un benchmark utile, deve soddisfare una serie di requisiti, come[15]:

- Deve essere basato su un carico di lavoro che rappresenta un'applicazione reale
- Deve eseguire ogni operazione critica che la piattaforma offre
- Non deve essere pensato per un prodotto/applicazione specifica
- Deve generare dati dai quali riprodurre risultati
- Non deve avere limitazione in fatto di scalabilità

3.1 Aspetti qualitativi di un sistema P2P

Esistono alcuni aspetti qualitativi molto importanti che ogni rete P2P può valutare. Essi sono:
Validità. Ogni risposta del sistema è completa e corretta.

Efficienza. Definita come la percentuale di performance e costi. Performance sono le abilità del sistema di raggiungere certi livelli di qualità sotto un determinato carico di lavoro. Solitamente si considera il tempo impiegato dal sistema a eseguire un determinato numero di operazioni o, viceversa, il numero di operazioni che riesce ad eseguire in un determinato tempo. Per costo, si intende, quante risorse vengono usate per espletare le operazioni richieste.

Correttezza. Significa che i costi delle operazioni del sistema e la disponibilità dei servizi è distribuita su ogni peer.

Scalabilità. Quanto riesce ad adattarsi il sistema P2P ad un cambiamento di numero di entità, siano esse nodi o servizi, preservando comunque la validità.

Robustezza. La persistenza del sistema P2P quando parti importanti del sistema vengono a mancare. Un sistema robusto non crolla mai, in quanto riesce a ripararsi da solo, non propagando tali guasti al resto del sistema.

Stabilità. La persistenza del sistema P2P sotto stress come un intensivo o frequente uso delle funzioni del sistema. Un sistema stabile deve mantenere la propria funzionalità. Differisce dalla robustezza, in quanto, non vengono presi in considerazione guasti del sistema.

3.2 Aspetti quantitavi di una DHT

Oltre agli aspetti qualitativi sopra elencati ci sono altri aspetti, che definiremo quantitavi[17] [16][18], che in una rete come DHT si possono valutare. Ne abbiamo identificate 2 principali:

LookUp. Quanto veloce è una lookup in una DHT? Più le prestazioni di questa sono elevate migliore sarà la qualità dei servizi offerti. Infatti questa operazione è intrinseca sia nella pubblicazione che nel recupero delle risorse, nonché nella ricerca di nodi.

Traffico di rete visto da un nodo. Quanta banda occupo? Che tipo di pacchetti invio e ricevo? Chi sono i nodi che comunicano con me? Grazie a queste informazioni sarà possibile poi studiare approfonditamente aspetti come: keyword maggiormente ricercate, come si dispongono i vari nodi nello spazio degli ID, ecc
Da questi 2 punti sarà possibile quantificare tutto quello che vogliamo, e capire quindi il comportamento del singolo nodo, di un gruppo di nodi e infine il comportamento dell'intera rete.

3.3 Stato dell'arte e PariDHT

Allo stato dell'arte per valutare i precedenti aspetti si sono utilizzati fin'ora 3 metodi:

- **simulatore**. Esso simula la rete P2P, e quindi i vari nodi presenti e ne definisce il comportamento per andare a valuatare aspetti di tipo qualitativo.
- **crawler[19]**. Un programma che naviga attraversa i peer della rete in esame facendo un'istantanea del suo stato e delle sue caratteristiche. Esso viene utilizzato per valutare gli aspetti quantitavi del sistema. Ma ha bisogno di molto tempo per produrre risultati attentibili (si parla di settimane se non di mesi).

- **peer pasivi**. Una serie di peer passivi, che semplicemente ascoltano e rispondono alle query degli altri peer. In questo modo riescono a carpire molte informazioni dal traffico che li attraversa.

Se per gli aspetti qualitativi del sistema non si può prescindere da un simulatore sviluppato ad-hoc, per gli aspetti quantitativi, in PariDHT, abbiamo voluto sviluppare alcuni strumenti che sostituiranno in parte i vari crawler e peer passivi. Grazie a questi nuovi strumenti, che ogni plugin di PariPari avrà a disposizione, si potrà costruire ogni tipo di valutazione desiderata. Diminuendo notevolmente lo sforzo per ottenerli.

4 PariProfiling

In questo paragrafo andremo a vedere quali strumenti abbiamo sviluppato per le valutazioni di tipo quantitativo.
Abbiamo cercato di creare strumenti per sostituire, almeno in parte, i crawler e peer passivi, quindi PariProfiling si occuperà di registrare ogni operazione ritenuta importante per fornire dati il più possibile completi. Ad esempio andremo a registrare ogni pacchetto che passa attraverso il peer in questione. Allo stesso modo registreremo dati sul nodo locale e su ogni nodo con cui comunicheremo, sostituendo una parte delle funzioni del crawler. A tutto questo si asscocia una nuova operazione che chiameremo **Statistic**, la quale offre la possibilità di ricevere dati statistici da nodi remoti.
L'insieme di queste funzionalità sostiuiscono, se non appieno, almeno in parte il crawler e i peer passivi; di certo i risultati che si otterrebbero col loro uso, si possono ottenere sviluppando accuratamente i dati ricevuti da questo nuovo tool.

4.1 Requisiti funzionali

Prima di partire con l'implementazione effettiva dei tools, abbia tracciato alcune caratteristiche base che deve avere:

- deve registrare accuratamente ogni operazione scelta
- deve introdurre un overhead minimale, se utilizzato
- deve avere overhead nullo in PariDHT, se non viene utilizzato
- è una scelta dell'utente, usarlo o meno
- recupera dati statistici da qualsiasi nodo nella rete PariDHT
- è una scelta dall'utente rispondere positivamente alle query di Statistic

4.2 Metriche

Come primo step abbiamo indivuato quali dati e quali operazioni di DHT registare, individuandone le seguenti:

- *Core Message*
- *LookUp*
- *Keys*

- *Node*
- *Network Packets*

Ora le vedremo in modo più approfondito.

Core Message. I messaggi del Core che arrivano a DHT sono semplicemente le richieste di operazioni specifiche, quali STORE o SEARCH, provenienti dagli altri plugin. Noi li prendiamo in considerazione in quanto ci interessa conoscere il tempo di esecuzioni delle varie richieste pervenute a DHT da plugin interni a PariPari. Quindi l'efficienza nella risposta a richieste di altri plugin.
Le informazioni che registriamo di ogni messaggio del Core sono:

- **messageID** : un identificativo unico del messaggio
- **pluginName** : il nome del plugin richiedente
- **request** : la richiesta, es Store, Search
- **finishGood** : come è stata completata, correttamente o meno
- **timeElapsed** : il tempo impiegato

LookUp. Come Lookup s'intende la classica ricerca di nodi vicini ad un determinato ID. Come già visto precedentemente questa funzione è basilare per una DHT, e le sue performance devono essere sempre tenute sotto controllo. Per performance intendiamo quanta banda utilizza, il numero di nodi interrogati, e deve comunque produrre dei risultati (quindi nodi) corretti.
Le informazioni che quindi registriamo di ogni LookUp sono:

- **targetID** : un identificativo unico utilizzato come chiave di ricerca
- **nodeFound** : il numero di nodi ritornati
- **N° nodi contattati** : quanti nodi contatto in totale
- **N° hop** : definito come il numero di salti tra un nodo e uno più vicino alla chiave cercata
- **timeElapsed** : il tempo impiegato

Keys. Sotto questo nome troviamo tutte le informazione riguardanti il KeyStorer di PariDHT. Esso è il contenitore di tutte le risorse di cui l'istanza corrente è responsabile. In questo modo sappiamo quali chiavi girano per la rete, ma soprattutto scopriamo il tempo di vita delle varie chiavi. Quest'ultima informazione è fondamentale per capire se una ricerca che fallisce (non trova la chiave desiderata) sia colpa del sistema o se effettivamente non esiste la chiave cercata. Nel caso in cui fosse il sistema a non poter recuperare una risorsa che dovrebbe ancora essere viva, allora dovremmo rivedere l'implementazione della DHT e apporre le modifiche più appropriate.
Le informazioni che registriamo sul KeyStorer sono:

- **keyID** : la chiave
- **value** : il valore
- **insertTime** : il momento in cui viene aggiunta una chiave al keyStorer
- **deleteTime** : il momento in cui una chiave viene eliminata dal KeyStorer

Node. Sotto questa nome troviamo ogni informazione relativa al NodeStorer di PariDHT. Esso è il contenitore di tutti i nodi conosciuti dall'istanza corrente, ritenuti buoni. Un nodo si definisce buono quando non è dietro NAT e ogni informazione che lo riguarda, nel nostro caso è tutti riassunto dall'ID, sia ritenuta valida attraverso un procedimento per l'identificazione sicura di un nodo.
Le informazioni che registriamo di ogni Nodo sono:

- **NodeID** : un identificativo unico del nodo
- **socketAddress** : la coppia indirizzoIP:porta
- **insertTime** : il momento in cui viene aggiunto un nodo al NodeStorer
- **deleteTime** : il momento in cui un nodo viene eliminato dal NodeStorer

Network Packets. Qui registriamo ogni informazione di ogni pacchetto di rete che attraversa l'istanza corrente. Grazie a queste informazioni sarà possibile definire il comportamento del nodo corrente.
Le informazioni che andremo a registrare per ogni pacchetto di rete le trovate in Tabella 1

PingTX	packID	ID relativo al pacchetto
	nodeTarget	Nodo a cui spediamo il pacchetto
	dimTX	Dimensione del pacchetto
PingRX	packID	ID relativo al pacchetto
	nodeTarget	Nodo da cui riceviamo il pacchetto
	targetSocketAddress	La coppia <IP:porta >relativa alla mia istanza
	dimRX	Dimensione del pacchetto
FindTX	packID	ID relativo al pacchetto
	nodeTarget	Nodo a cui spediamo il pacchetto
	SearchID	ID cercato
	dimTX	Dimensione del pacchetto
FindRX	packID	ID relativo al pacchetto
	nodeTarget	Nodo da cui riceviamo il pacchetto
	numNodeReturn	Numero di nodi trovati
	dimRX	Dimensione del pacchetto
SearchTX	packID	ID relativo al pacchetto
	nodeTarget	Nodo a cui spediamo il pacchetto
	SearchID	ID cercato
	dimTX	Dimensione del pacchetto
SearchRX	packID	ID relativo al pacchetto
	nodeTarget	Nodo da cui riceviamo il pacchetto
	findValues	indica la presenza di valori relativi alla chiave, nel caso non ci siano valori il pacchetto conterrebbe nodi
	numReturn	Numero di valori o nodi trovati
	dimRX	Dimensione del pacchetto
StoreTX	packID	ID relativo al pacchetto
	nodeTarget	Nodo a cui spediamo il pacchetto
	keyID	la chiave da salvare
	dimValue	Dimensione del valore da salvare
	dimTX	Dimensione del pacchetto
StoreRX	packID	ID relativo al pacchetto
	nodeTarget	Nodo da cui riceviamo il pacchetto
	accepted	indica se la chiave è stata accettata dal nodo contattato
	loadFactor	in caso non venga accettata la STORE indica il proprio stato di memoria
	stored	inidca se la chiave è stata salvata con successo

	dimRX	Dimensione del pacchetto
StatTX	packID	ID relativo al pacchetto
	nodeTarget	Nodo a cui spediamo il pacchetto
	query	La query trasmessa
	dimTX	Dimensione del pacchetto
StatRX	packID	ID relativo al pacchetto
	nodeTarget	Nodo da cui riceviamo il pacchetto
	resultAvailable	Indica se il nodo ha accettato di eseguire la query e darci quindi informazioni sulle proprie statistiche
	dimRX	Dimensione del pacchetto

Tabella 1: Informazioni relative a tutti i pacchetti di rete di PariDHT

4.3 Scelta implementiva

Per registare tutte le informazioni definite nel paragrafo precedente, e per aver un overhead minimale, abbiamo deciso di usare un DataBase, in particolare HSQLDB[11], come contenitore e di usare il pattern Observer[12] come metodo di sincronizzazione interno al plugin.

4.3.1 HSQLDB

HSQLDB (Hyper Structured Query Language Database) è un sistema di gestione di DB relazionali scritto in Java. Per interfacciarsi con java utilizza i driver JDBC. Esso offre un ambiente veloce e leggero che offre sia la scrittura in memoria centrale, molto veloce, sia la scrittura su disco. Sono disponibili due modi di utilizzo, embedded (incorporato) o server. Esso può funzionare nella modalità runTime di java, ed è per questa peculiarità che lo abbiamo scelto, oltre all'ovvio fatto di essere un software libero.

4.3.2 Pattern Observer

Per definizione un pattern, nell'ambito della programmazione orientata agli oggetti, rappresenta una schema progettuale che fornisce una soluzione ad un problema ricorrente.
Il pattern Observer, in particolare, rappresenta una soluzione nei casi in cui è necessaria una certa relazione (da uno a molti) tra un oggetto caratterizzato da diversi stati interni, ed una serie di altri oggetti che reagiscono in

conseguenza al cambimento di questi stati. Quindi abbiamo un'oggetto che viene osservato (detto subject) e tanti oggetti che osservano i cambiamenti di quest'ultimo (detti observers).
Uno dei tanti vantaggi di questo schema è quello di limitare la dipendenza fra due tipi di oggetti, nel senso che ambedue le parti possono modificare la loro struttura senza avere un'impatto sull'altro, lavorando in maniera totalmente disaccoppiata.

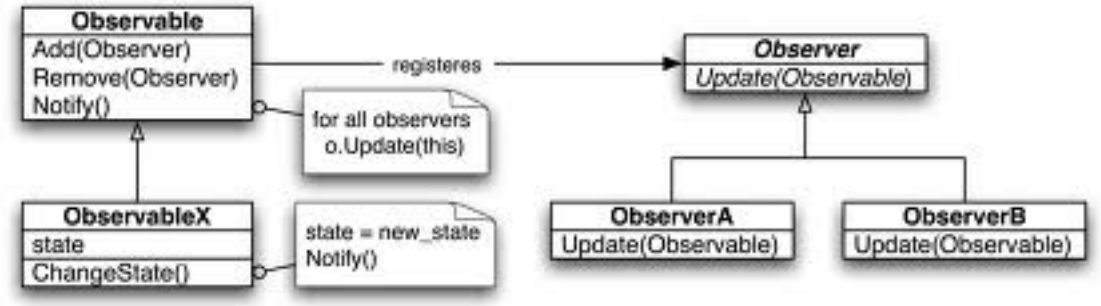

Figura 2: Diagramma UML del pattern Observer

Nel package java.util, sono presenti la classe Observable (soggetto) e l'interfaccia Observer (osservatore), che abbiamo usato nella nostra implementazione. La classe Observable ci fornisce i meccanismi necessari per registrare gli osservatori e notificare i cambiamenti avvenuti. L'operazione di notifica di cambiamento di uno stato avviene utilizzando il metodo notifyObservers(Object args) preceduto da una chiamata al metodo setChanged(). Ad esempio:

```
...
public class Subject extends Observable {
  ...
  public void startProcess() {
    ...
    setChanged();
    notifyObservers("Processo iniziato");
    ...
    setChanged();
    notifyObservers("Processo concluso");
    ...
  }
```

```
    ...
}
...
```

Gli osservatori invece implementano l'interfaccia Observer, che contiene un unico metodo update():

```
class Osservatore implements Observer {
    ...
    public void update(Observable obs, Object args) {
        ...
    }
    ...
}
```

Il metodo update oltre a ricevere un argomento riceve anche un riferimento all'Observable che ha notificato l'evento. Si deduce da ciò che un oggetto Observer può a sua volta registrarsi a più subject.
Grazie all'ausilio di questo pattern è stato possibile rendere minimale l'overhead dovuto al profiling e allo stesso tempo rendere nullo l'overhead in caso di mancato utilizzo dello stesso. Infatti, ad ogni operazione sottoposta a profiling, viene richiamato un metodo composto da 3 semplici operazioni. Vediamole nel dettaglio:

- controllo che il profiling sia attivo, in caso negativo termino
- creo la query di inserimento nel DB
- eseguo la query

Nel caso di profiling disattivato si esegue solo la prima operazione di controllo, per cui si può considerare come overhead nullo.

4.4 Statistic API

Per ricavare dati da istanze remote, abbiamo dovuto definire 2 API da apporre al plugin di Interfaces[1].

[1]Dentro Interfaces vengono inserite tutte le API offerte da ogni plugin interno di PariPari

4.4.1 DHTPP

Prima di vedere suddette API, vediamo brevemente come è strutturato il protocollo di comunicazione di PariDHT.

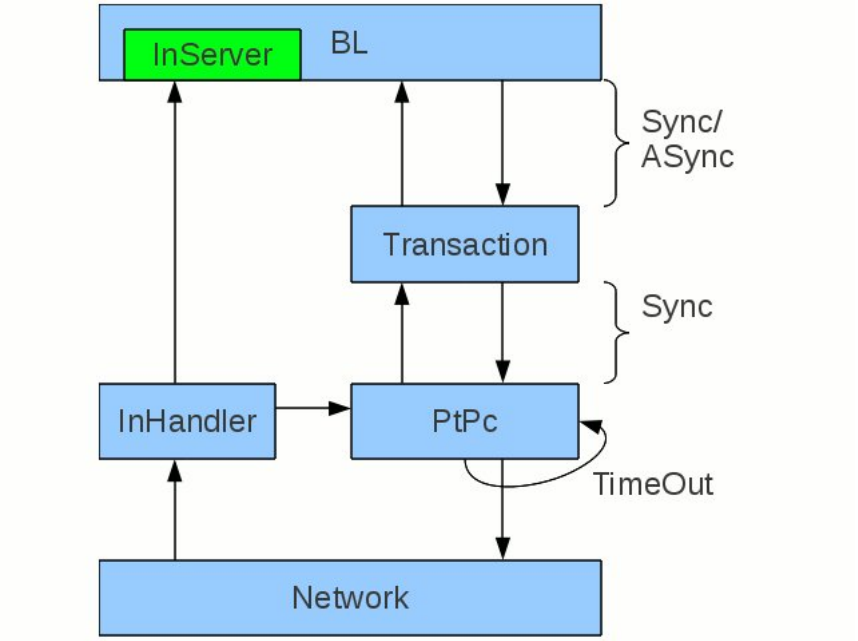

Figura 3: Struttura dello strato di rete di PariDHT

Dalla fig. 3 si nota chiaramente come questo strato si voglia frappore tra lo strato kernel e il livello più basso di rete (streaming di pacchetti). Esso si occuperà di ricevere richieste (per noi chiamate transazioni) da parte del kernel, risolverle tramite il network e rispondere al kernel concludendo la transazione. Allo stesso modo riceverà richiesta da parte del network, risolvendole tramite il kernel, il quale risponderà avviando una transazione di risposta.
Per capire meglio come funziona questo strato vediamo ora le funzionalità delle singole classi interessate:

- **Transaction**: definito come serie di botta e risposta tra 2 nodi.
- **PtPc**: (Point to Point Comunication) singolo botta e risposta[2] tra 2 nodi.

[2]Equivale alla comunicazione di 2 pacchetti una di richiesta e uno di risposta

- **InHandler**: Riceve richieste da InHandler, le soddisfa tramite il kernel e avvia la corretta transazione di risposta.
- **Inserver**: Riceve richieste/risposte dal Network e le smista a seconda del tipo verso InServer/PtPc.

In realtà sia Transaction che PtPc non sono classi semplici ma una serie di classi che implementano le stesse funzionalità per le diverse primitive presenti in PariDHT, come STORE, SEARCH, ...

4.4.2 Implementation

La prima API che andremo a vedere sarà DHTResultQueryAPI, essa si è resa necessaria in quanto gli oggetti della classe *ResultSet*[3] non potevano venire trasmessi ad un altro nodo ed essere ancora considerati utili. Quindi gli oggetti di questa classe vengono usati per comunicare risultati di query eseguite sul DB di PariProfiling. Essi infatti rappresentano precisamente il ResultSet.

```
public abstract class DHTResultQueryAPI implements API{
  public abstract List<String> getNameColoumn();

  public abstract List<Integer> getTypeColoumn();

  public abstract List<List<Object>> getValueColoumn();
}
```

La seconda classe, quella rimanente, è DHTStatAPI. Essa rappresenta l'operazione di recupero dati da nodi remoti o dalla propria istanza.

```
public abstract class DHTStatAPI implements API{
   public DHTStat(IFeatureValue[] features,
                  DHTNodeAPI node, String query)
    throws InterruptedException, ExecutionException {}

  public DHTStat(IFeatureValue[] features, String query)
    throws InterruptedException, ExecutionException {}

  public abstract boolean hasResult();

  public abstract DHTResultQueryAPI getResult();
```

[3]Oggetto risultante dopo l'esecuzione di una query su un DB

```
    public abstract String getQuery();

    public abstract DHTNodeAPI getRemoteNode();
}
```

Nel momento in cui PariDHT riceve una richiesta di questo tipo, mettiamoci nel caso voglia informazioni da un nodo remoto, semplicemente crea e poi lancia la transazione appropriata, nel nostro caso *Statistic*. La quale manda un pacchetto al nodo in questione contenente la query desiderata, e attende la risposta. A questo punto, ci possono essere 3 casi:

- ricevo una risposta corretta con risultati
- ricevo una risposta corretta senza risultati
- non ricevo risposte o arrivano troppo tardi

Nel primo caso risponderemo alla richiesta iniziale con l'oggetto DHTResultQuery completo di risultati, mentre negli altri casi avremo il medesimo oggetto privo di risultati.

4.4.3 JSON

Per trasmettere e ricostruire oggetti complicati come DHTResultQuery, non abbiamo usato l'interfaccia Serializable, bensì JSON[13]. JSON (JavaScript Object Notation) è un semplice formato per lo scambio di dati. Per le persone è facile da leggere e scrivere, mentre per le macchine risulta facile da generare e analizzarne la sintassi. Si basa su un sottoinsieme del Linguaggio di Programmazione JavaScript, Standard ECMA-262 Terza Edizione - Dicembre 1999.
JSON è un formato di testo completamente indipendente dal linguaggio di programmazione, ma utilizza convenzioni conosciute dai programmatori di linguaggi della famiglia del C, come C, C++, C#, Java, JavaScript, Perl, Python, e molti altri. Questa caratteristica fa di JSON un linguaggio ideale per lo scambio di dati.
JSON è basato su due strutture:
Un insieme di coppie nome/valore. In diversi linguaggi, questo è realizzato come un oggetto, un record, uno struct, un dizionario, una tabella hash, un elenco di chiavi o un array associativo. Un elenco ordinato di valori. Nella maggior parte dei linguaggi questo si realizza con un array, un vettore, un elenco o una sequenza. Queste sono strutture di dati universali.

Virtualmente tutti i linguaggi di programmazione moderni li supportano in entrambe le forme. E' sensato che un formato di dati che è interscambiabile con linguaggi di programmazione debba essere basato su queste strutture.
In JSON, assumono queste forme:
Un oggetto (si veda Figura 4) è una serie non ordinata di nomi/valori. Un oggetto inizia con (parentesi graffa sinistra) e finisce con (parentesi graffa destra). Ogni nome è seguito da : (due punti) e la coppia di nome/valore sono separata da , (virgola).

Un array (si veda Figura 5) è una raccolta ordinata di valori. Un array

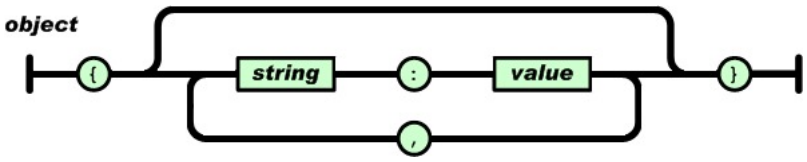

Figura 4: Struttura di un oggetto in JSON

comincia con [(parentesi quadra sinistra) e finisce con] (parentesi quadra destra). I valori sono separati da , (virgola).

Un valore (si veda Figura 6) può essere una stringa tra virgolette, o un

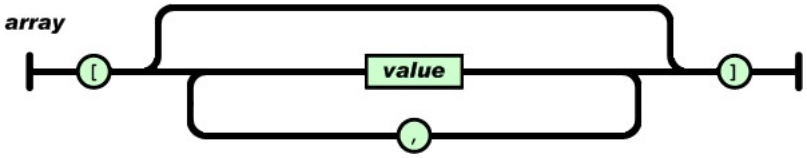

Figura 5: Struttura di un array in JSON

numero, o vero o falso o nullo, o un oggetto o un array. Queste strutture possono essere annidate.

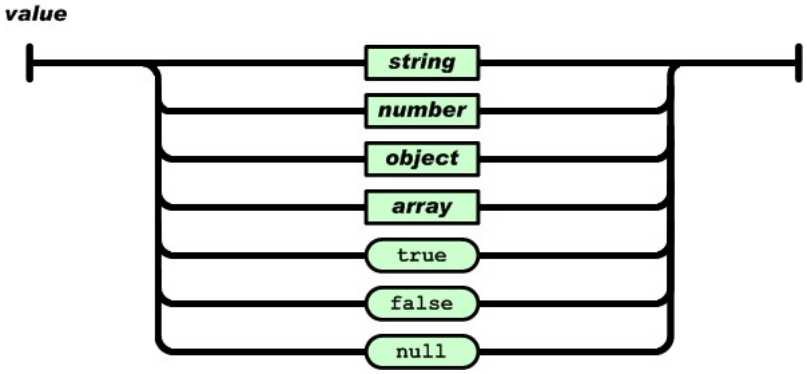

Figura 6: Struttura del valore in JSON

5 Simulazione PariDHT

Abbiamo quindi voluto creare una simulazione della nostra rete PariDHT che la mettesse sotto forte stress, con un numero di operazioni elevato, e con un buon numero di nodi, per osservare quanto il tool PariProfiling sia sensibile. In particolare se registra tutte le operazioni precedentemente descritte in modo corretto e veloce, o se lascia qualche informazione di troppo non registrata.

Abbiamo deciso di fare le simulazioni di PariDHT sul cluster eridano del DEI. Esso è stato diviso in 16 nodi ognuno dei quali possiede

- CPU: nehalem i7 950 @ 3.07 GHz (LGA 1366)
- RAM: 3 * 4 GB @ 1600MHz tri-channel
- HARD DISK: 6 * SAMSUNG HD103SJ

In particolare la simulazione prevederà la creazione di una rete PariDHT con l'ausilio di 16 server posti in ascolto nei vari nodi del cluster. Essi andranno a creare la rete di base alla quale si aggiungeranno i vari nodi per la simulzione effettiva.
Non sapendo quante istanze di PariDHT riusceremo a caircare per ogni nodo, cominceremo da un minimo di 4 istanze per ogni nodo, per poi proseguire aumentando il numero d'istanze. Ma prima vediamo di cosa necessita la simulazione.

5.1 Preparazione

La preprazione della simulazioe si divide tra la generazione dei nodi, la comunicazione tra i nodi e la definizione del loro comportamento.

5.1.1 Generazione

Come primo passo abbiamo definito e sviluppato uno script che generasse le istanze che andranno a comporre la simulazione. Suddetto script genera un file di output in cui ogni riga definisce:

- *Nodo* : Nodo del cluster responsabile della sua esecuzione
- *Istanza* : L'istanza del cluster responsabile della sua esecuzione
- *Tipo* : Il tipo di nodo da simulare (si veda 5.1.3)
- *(X, Y)*: La posizione del nodo in un'area
- *Start* : il momento in cui partirà la sua simulazione
- *End* : il momento in cui terminerà la sua esecuzione

Inoltre ogni riga definisce intrinsecamente la porta sulla quale l'istanza comunicherà, essa viene definita da X + N°riga, dove X è una porta iniziale valida.

Posizione nodo. Per simulare il ritardo di comunicazione tra i nodi, si è definito un aerea temporale dentro la quale ogni punto, definito dalle coordinate (X,Y), rappresenta un possibile nodo. La distanza tra due punti, in quet'area, definisce il ritardo con cui i pacchetti vengono scambiati.
Lo script selezionerà N punti (o nodi) in base ad alcuni parametri, definiti in un file di configurazione, quali:

- *simulation time*: il tempo massimo della simulazione
- *simulation instances*: il numero di istanze desiderate
- *cluster instances*: quanti nodi del cluster utilizziamo
- *min*: ping minimo tra 2 nodi generati
- *max*: ping massimo tra 2 nodi generati
- *min life*: la vita minima del nodo
- *type X*: definisce la percentuale di nodi di tipo X, dove X verrà nella sezione 5.1.3
- *older than X*: definisce la percentuale di nodi con vita >= X. Dove X rappresenta una percentuale del tempo massimo della simulazione

Inoltre sono state scelte delle sottoaeree dell'aerea iniziale, le quali sono definite da altri paremetri, come:

- *nome* : il nome della sottoaera, ad es. Italy
- *N° nodi*: numero di nodi genrati in quest'aerea
- *pingMax*: il ping massimo tra due nodi generati sotto questa aerea
- *min Delay To X*: il ritardo minimo verso X. Dove X è un altra sottoarea.

Grazie a tutti questi parametri possiamo generare il numero richiesto di nodi dalla simulazione e definirne il loro comportamento.

5.1.2 Comunicazione

Per la comunicazione tra i vari nodi, si sono sviluppati due plugin che sostiuiscono connectivityNIO. In particolare:

- *connectivityFake*: che sostituirà NIO, fornendo i socket per PariDHT utilizzando direttamente le classi di java. Questo plugin verrà utilizzato dai vari server che formeranno la rete iniziale.
- *dummy*: che fornisce anch'esso i socket di PariDHT, ma grazie al file generato dallo script visto nella sezione 5.1.1, aggiungerà il ritardo desiderato. Calcolato sulla distanza nello piano tra l'istanza corrente e chi vogliamo raggiungere.

5.1.3 Comportamento

Il plugin Dummy sarà colui che eseguirà la vera e propria simulazione ovvero, oltre a fornire i socket a PariDHT, eseguirà le seguenti operazioni:

- caricherà da file le informazioni su chi sono, ovvero su quale nodo del cluster è quale istanza rappresenta
- dal file generato in precedenza, selezionerà le righe appropriate
- avvierà e spegnerà le istanze di PariDHT nei momenti descritti
- simulerà un plugin esterno che utilizzi PariDHT in un determinato modo dato anch'esso nel file generato precedentemente

Il modo in cui simulerà l'utilizzo di PariDHT, viene dato in un altro file di configurazione. Esso descrive le operazioni che l'istanza in questione eseguirà.

- *nome*: nome associato al tipo di nodo da simulare
- *store*: percentuale di store sulle richieste da sottoporre a PariDHT
- *storeNew*: percentuale di store di nuove chiavi
- *search*: percentuale di search sulle richieste da sottoporre a PariDHT
- *searchOld*: percentuale di ricerche di chiavi sicuramente già esistenti
- *stat*: percentuale di Statistic sulle richieste da sottoporre a PariDHT

5.2 Risultati

Purtroppo non abbiamo potuto ottenere risultati in quanto ogni simulazione lanciata ha messo in luce un unico grande problema di sincronizzazione interno a PariPari. Al momento ne stiamo cercando le cause.

Conclusione

In questo elaborato si è discusso di un nuovo tool per la registrazione di quei dati necessari a misurare gli aspetti quantitativi di una DHT. Purtroppo non abbiamo potuto provarne l'efficacia, ma ci stiamo adoperando in tal senso.
Attualmente è in via di definizione una nuova PariDHT, in quanto quella attuale è funzionante ma come ogni rete distribuita soffre dei problemi derivanti dai sybilAttack. Con la nuova struttura andremo ad accettare, per un periodo limitato, i problemi di questi attacchi ma allo stesso tempo si cercherà di raggirarla spostando ogni risorsa da un responsabile ad un altro ogni periodo (l'idea sarebbe di 15minuti). Per fare ciò sono in fase di definizione le nuove API assieme al nuovo protocollo di rete che dovrà supportare il tutto ed essere performante, infatti genererà un notevole traffico di rete in aggiunta a quello normale.
Solo a questo punto si potrà adeguare anche il tool sviluppato portandolo a tracciare correttamente anche la nuova PariDHT.
Infine dovremo verificare quanto l'utilizzo di JSON nell'intero pacchetto di rete porterà vantaggi e/o svantaggi. I primi in termine di facilità d'utilizzo e portabilità a livello di linguaggi di programmazione, i secondi in termine di dimensione maggiorata dei pacchetti e quindi di un carico maggiore che attraverserà la rete.
Concludendo, l'esperienza accumulata in questo progetto, come tester e sviluppatore, saranno un proficuo bagaglio per il mio futuro.

Riferimenti bibliografici

[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] **Wiki PariPari**
http://www.paripari.it/mediawiki/index.php/Main_Page

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

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

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

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

[10] **Hash**
http://it.wikipedia.org/wiki/Hash

[11] **HSQLDB**
http://hsqldb.org/

[12] **Pattern observer**
http://www.javacamp.org/designPattern/

[13] **JSON**
http://www.json.org/

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

[15] Max Lehn, Tonio Triebel, Christian Gross, Dominik Stingl, Karsten Saller, Wolfgang Effelsberg, Alexandra Kovacevic and Ralf Steinmetz3
Designing Benchmarks for P2P Systems

[16] Moritz Steiner, Taoufik En-Najjary, and Ernst W. Biersack
Analyzing Peer Behavior in KAD

[17] Di Wu, Ye Tian, Kam Wing Ng
Analytical Study on Improving Lookup Performance of Distributed Hash Table Systems under Churn

[18] Elena Digor
Kademlia Measuremenets

[19] Ghulam Memon, Reza Rejaie, Yang Guo, Daniel Stutzbach
Large-Scale Monitoring of DHT Traffic