Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

PariDHT: Testing

Relatore: Prof. **Enoch Peserico**

Correlatore: Ing. **Michele Bonazza**

Laureando: **Gianni Mezzalana**

Anno Accademico 2012/2013

Indice

Elenco delle figure

Elenco delle tabelle

Sommario

La sfida di PariPari è progettare una rete *peer-to-peer* (*P2P*) senza alcun server, che garantisca anonimato e metta a disposizione un ricco portfolio di applicazioni e servizi, accessibili semplicemente attraverso una connessione internet.

Sono otto anni che attraverso versioni di collaudo interne PariPari tenta di trovare la propria forma definitiva con la prima *release* pubblica.

In questo elaborato viene presentato il lavoro di testing svolto all'interno di PariPari, ponendo in evidenza aspetti e tecniche di progettazione della Rete su cui si costruisce l'intero progetto. Saranno esposte le motivazioni che hanno portato all'attuale gestione delle risorse umane e le scelte progettuali effettuate. Verrà presentato il lavoro di *testing* ponendo particolare enfasi sulle tecniche adottate e gli strumenti utilizzati. Infine saranno indicate le linee guida che si ritengono necessarie per condurre il progetto alla sua *release* ufficiale.

Capitolo 1

PariPari

1.1 Introduzione a PariPari

PariPari è un progetto attivo ormai da anni nel dipartimento di Ingegneria dell'Informazione dell'università di Padova. Tra i primi documenti ufficiali sul progetto troviamo le Tesi di laurea di Simonetto, A., *Progettazione e realizzazione in Java di una rete P2P anonima e multifunzionale: connettività sicura e affidabile*, Padova, 2005 e di Bertasi, P., *Progettazione e realizzazione in Java di una rete peer to peer anonima e multifunzionale*, Padova, 2006.

«PariPari è una rete *serverless* basata su una variante di *kademlia* che garantisce l'anonimato dei suoi nodi e, soprattutto, è multifunzionale.» [1]

La multifunzionalità è l'aspetto certamente più importante di questo progetto. Non a caso la struttura di PariPari è modulare: attorno ad un nucleo centrale (*Core*) si sviluppano altri moduli fondamentali per il funzionamento base della Rete. Questi moduli sono l'insieme dei *plug-in* della "cerchia interna", come *Connectivity*, *Storage* e *DHT*. Inoltre è tutt'ora in sviluppo un insieme di *plug-in* per fornire servizi come ad esempio *storage* distribuito, condivisione di file, *instant messaging*, e *database* distribuito.

[1] Bertasi, P., *Progettazione e realizzazione in Java di una rete peer to peer anonima e multifunzionale*, 2006, Padova, p.4

Grazie alla documentazione messa a disposizione, PariPari è anche un *framework* di sviluppo software, che offre ad un programmatore esterno la possibilità di progettare e costruire i propri *plug-in*, in linea con la filosofia *"opensource"*.

«PariPari si propone, quindi, di creare una macchina virtuale capace di provvedere a tutti i bisogni dell'utente di internet mantenendosi dipendente solo dalla comunità dei nodi da cui è formata.»[2]

Altro punto di forza di PariPari è la scelta del linguaggio di programmazione. Scegliere un linguaggio di programmazione ad oggetti come Java scavalca ostacoli di compilazione, conferendo a PariPari una grande portabilità. Infine la scelta di una architettura modulare permette una forte parallelizzazione dello sviluppo, per cui si possono modificare intere funzionalità di PariPari senza compromettere il funzionamento del sistema stesso. L'esempio più evidente è la possibilità di cancellare o modificare interi *plug-in* appartenenti alla cerchia esterna senza influenzare il funzionamento degli altri *plug-in* (a meno di dipendenze esplicite).

1.2 Cenni sull'architettura

L'architettura di PariPari è Core-centrica. Per ogni nodo della Rete è attiva una istanza di PariPari. Il *Core* di una particolare istanza è il modulo fondamentale che si occupa della gestione delle risorse e dello scambio di informazioni tra tutti gli altri moduli. Come accennato in precedenza, è possibile identificare una "cerchia interna" di *plug-in* che offrono servizi necessari per il funzionamento della Rete; questi sono *Connectivity*, *DHT* e *Storage*. Il modulo *Connectivity* si occupa di instaurare le connessioni tra vari nodi della rete lavorando ai livelli 3 e 4 della classificazione ISO/OSI, quindi trasporto e rete. *DHT* si occupa di gestire la logica della "tabella hash distribuita" e quindi la topologia della rete,

[2] *Ivi* p.5

rimanendo nella classificazione ISO/OSI si occupa dei livelli 5 e 6 (sessione e presentazione). Infine a *Storage* è delegata la gestione di inserimento e recupero delle informazioni dalla memoria di massa.

Attraverso documentate Interfacce di Programmazione di Applicazione API (*Application Programming Interface*) i *plug-in* della cerchia interna permettono anche ad uno sviluppatore esterno di interagire facilmente. Un esempio sono i moduli aggiuntivi come *Mulo* [3] o *Torrent* [4], che riescono ad offrire i loro servizi grazie all'utilizzo delle primitive dei *plug-in* della "cerchia interna".

La struttura modulare può essere riassunta dallo schema in figura 1.1 .

[3]Una variante del celebre software di condivisione di file **eMule** costruita appositamente per PariPari

[4]Una variante del noto software *BitTorrent* di condivisione dei file, che utilizza per l'appunto file di estensione *".torrent"* codificati con *Bencode*

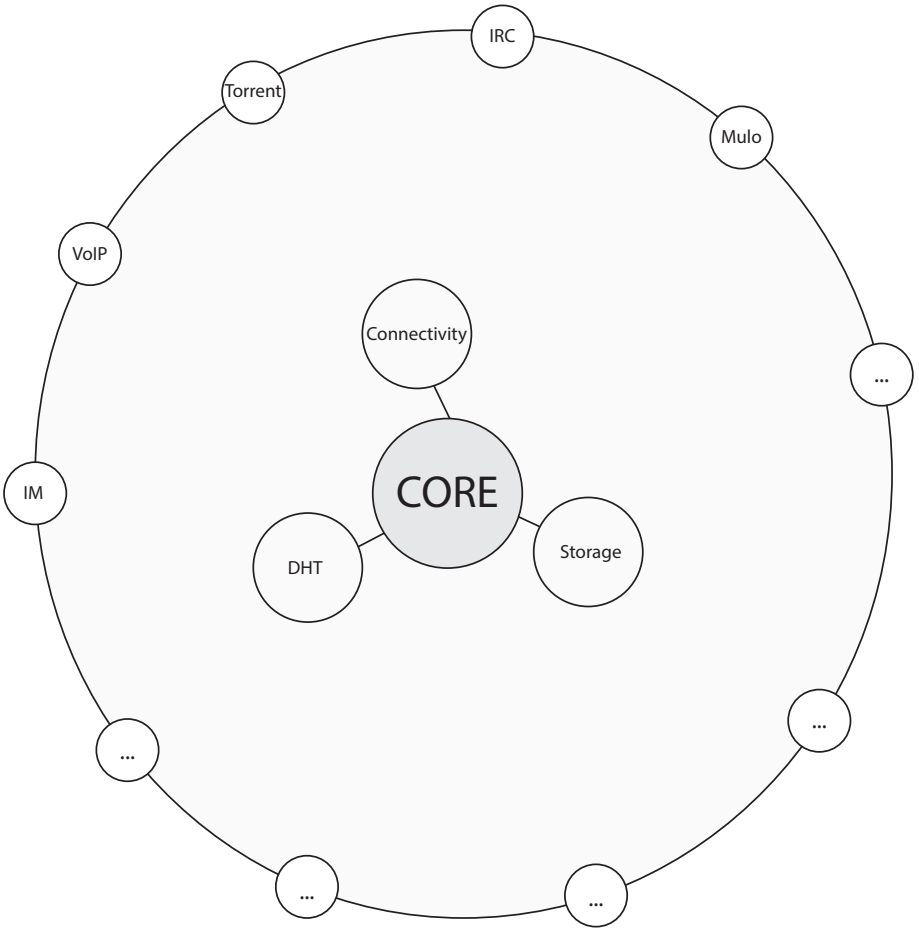


Figura 1.1: La struttura di PariPari.

Capitolo 2

Il nuovo approccio

PariPari è nato come progetto ambizioso e come tale è stato oggetto di numerose discussioni e problematiche tecnico-progettuali. Spesso tali problemi non erano legati solamente a concetti prettamente tecnici e di programmazione, ma anche alle risorse umane. Infatti con un *team* che vantava più di 60 studenti impegnati nello sviluppo parallelo di circa dieci unità operative (*plug-in*) indipendenti, i problemi di gestione erano tutt'altro che banali.

A cavallo tra il 2011 e il 2012, si sono fatte diverse scelte radicali nelle linee di conduzione del Progetto. Innanzitutto si è diminuito il personale per aumentare l'efficacia e il controllo. Quindi si è scelto di adottare una piattaforma di sviluppo software dedicata al lavoro in *team*: Redmine. Questa è una piattaforma che permette una maggiore flessibilità e una maggior funzionalità rispetto alla precedente gestione tramite *Google Groups*. Redmine è un'applicazione *web* scritta attraverso *Ruby on Rails* [1]. È stata progettata per permettere la gestione di progetti mantenendo come obiettivo flessibilità e accessibilità, coniugando la facile gestione e interazione di uno strumento *web based*, con ricche funzionalità come diagrammi Gantt, calendari, scadenziari, *Time Tracking*, Forum, Wiki, Mul-

[1] Un framework web opensource, `http://rubyonrails.org` (visitato il 7 febbraio 2013)

tiutenza, Multilingua, Multiprogetto.

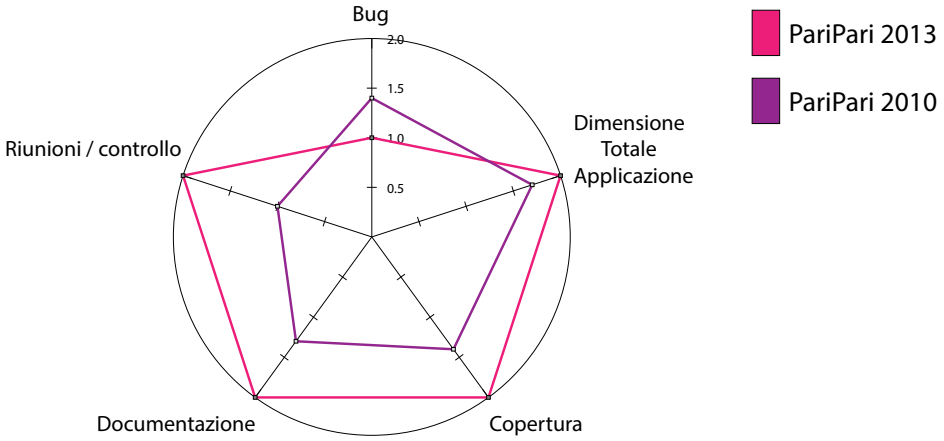


Figura 2.1: Tendenza di PariPari
(per dettagli fare riferimento all'appendice A)

Il problema spesso sottovalutato in un progetto delle dimensioni di PariPari è appunto il fattore umano. Gli studenti impegnati in PariPari rimangono all'interno del progetto solo per qualche anno, a volte addirittura qualche mese non garantendo la continuità necessaria per un progetto di queste dimensioni. Questo importante fenomeno di *"turnover"* dei programmatori di PariPari è decisamente una delle cause che fin'ora hanno rallentato lo sviluppo.

È interessante notare come tutti gli ex-sviluppatori di PariPari abbiano dato un contributo al Software significativo solo dopo qualche mese dall'ingresso nel *team* di sviluppo. Inizialmente infatti ogni programmatore ha dovuto maturare un livello di competenze tali da riuscire a muoversi autonomamente all'interno del progetto. Solo quando non sono più stati necessari aiuti esterni dai membri formatori di PariPari, il neo-programmatore riesce finalmente a dare il suo contributo. Anche per questo motivo è stato introdotto Redmine, per offrire al neo-programmatore tutti gli strumenti per studiare monitorare e comprendere il progetto.

2.1 Redmine

La piattaforma *Redmine* è stata adottata appositamente per rendere "documentata", "centralizzata" e "procedurale" la progettazione e lo sviluppo del software. Essa si compone di diversi moduli ognuno dei quali assolve a compiti specifici. Saranno presentati di seguito i moduli più utilizzati, ponendo attenzione al loro ruolo all'interno del ciclo di sviluppo del modulo DHT.

Redmine organizza lo sviluppo software in "progetti". Ogni progetto è un macro-modulo a se stante in cui si possono utilizzare gli stessi strumenti e in cui ritroviamo sempre la stessa organizzazione interna. PariPari, si compone infatti dei sotto progetti *DiESeL*, *Distributed Storage*, *GUI*, *IM*, *IRC*, *Local Storage*, *Mulo*, *Core*, *PariSync*, *DHT*, *Torrent*, *WEB*.

A questo livello di astrazione Redmine risulta essere una sorta di "cruscotto aziendale"[2].

L'accesso a Redmine è subordinato ad una autenticazione con username e password per identificare durante tutta la sessione di collegamento l'autore di qualsiasi tipo di attività. Tra i vari sotto progetti di PariPari presenti, sarà preso in considerazione quello relativo al nuovo modulo DHT, ma questa scelta non è vincolante in quanto l'interfaccia di base è uguale per ogni sotto progetto.

[2] *Management cockpit* strumento di presentazione delle informazioni a supporto delle decisioni aziendali. Esso viene realizzato a supporto del singolo decision maker ai vertici dei dipartimenti aziendali. Solitamente, un management cockpit viene realizzato con applicazioni che si distinguono per l'eccellente capacità di sintesi, mostrando i principali indicatori di performance aziendale in maniera ergonomica, al fine di rappresentarne con immediatezza l'andamento dei fenomeni rispetto agli obiettivi prefissati o agli standard aziendali. A ogni indicatore sono legati, in una struttura ad albero (value tree), gli indicatori di livello inferiore a esso correlati, via via fino alla reportistica di livello base. Tratto da `http://it.wikipedia.org/wiki/Management_cockpit`, (visitato il 9 marzo 2013).

2.1.1 Panoramica

Una delle prime viste di progetto e la più generale è la "Panoramica". All'interno di questa è possibile estrarre informazioni riguardo il tempo totale impiegato nel progetto, i membri del progetto e i loro ruoli, e il "*tracking*" delle segnalazioni. Le segnalazioni includono: *bug*, *features*, *support*, *testing*, *review*, *planning*. Sempre a questo livello è possibile visualizzare le segnalazioni in uno schematico calendario o in un *diagramma di Gantt*.

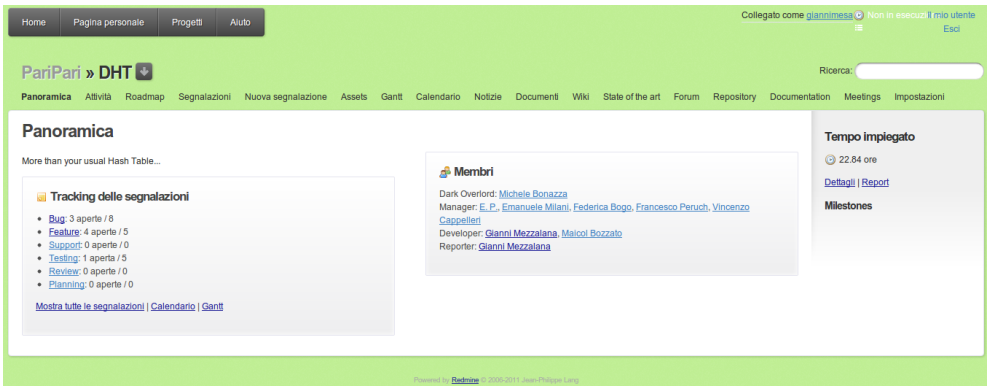


Figura 2.2: Redmine - Panoramica

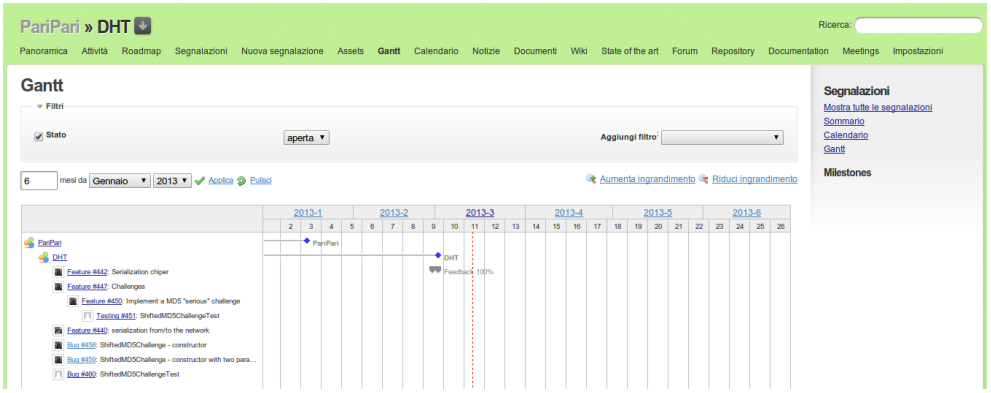


Figura 2.3: Redmine - Diagramma di Gantt

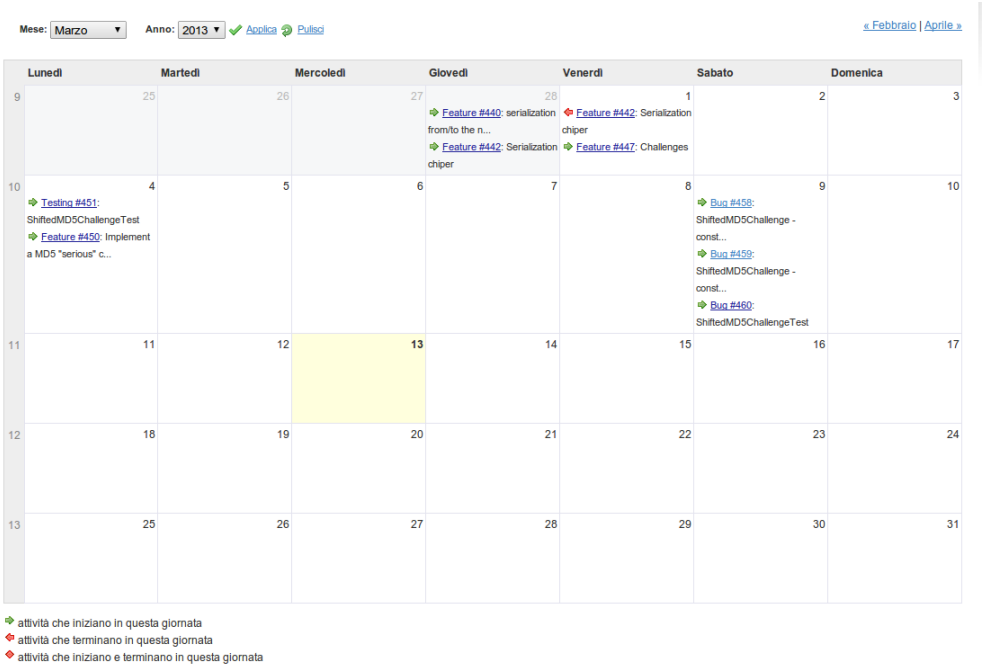


Figura 2.4: Redmine - Calendario

2.1.2 Attività

Nel momento in cui viene effettuato un qualsiasi tipo di operazione come il *commit* [3] del codice scritto nel *repository* [4] tramite *Subversion*[5] o un commento nella *wiki* di progetto o ancora un messaggio nel forum di progetto, questo viene registrato nella lista delle attività che viene mantenuta costantemente aggiornata. Con opportune configurazioni sull'*account* di Redmine, ogni utente può essere notificato in tempo reale (tramite una *e-mail*) di qualsiasi nuovo inserimento nel registro della attività.

2.1.3 Segnalazioni

Fornire una documentazione con il massimo dettaglio richiede inevitabilmente rigide procedure di progetto.

[3]Operazione di invio dei dati e dei cambiamenti effettuati in locale nel codice al database remoto

[4]Database remoto implementato con un sistema di gestione delle basi di dati. È un ambiente di un sistema informativo (ad es. di tipo ERP), in cui vengono gestiti i metadati, attraverso tabelle

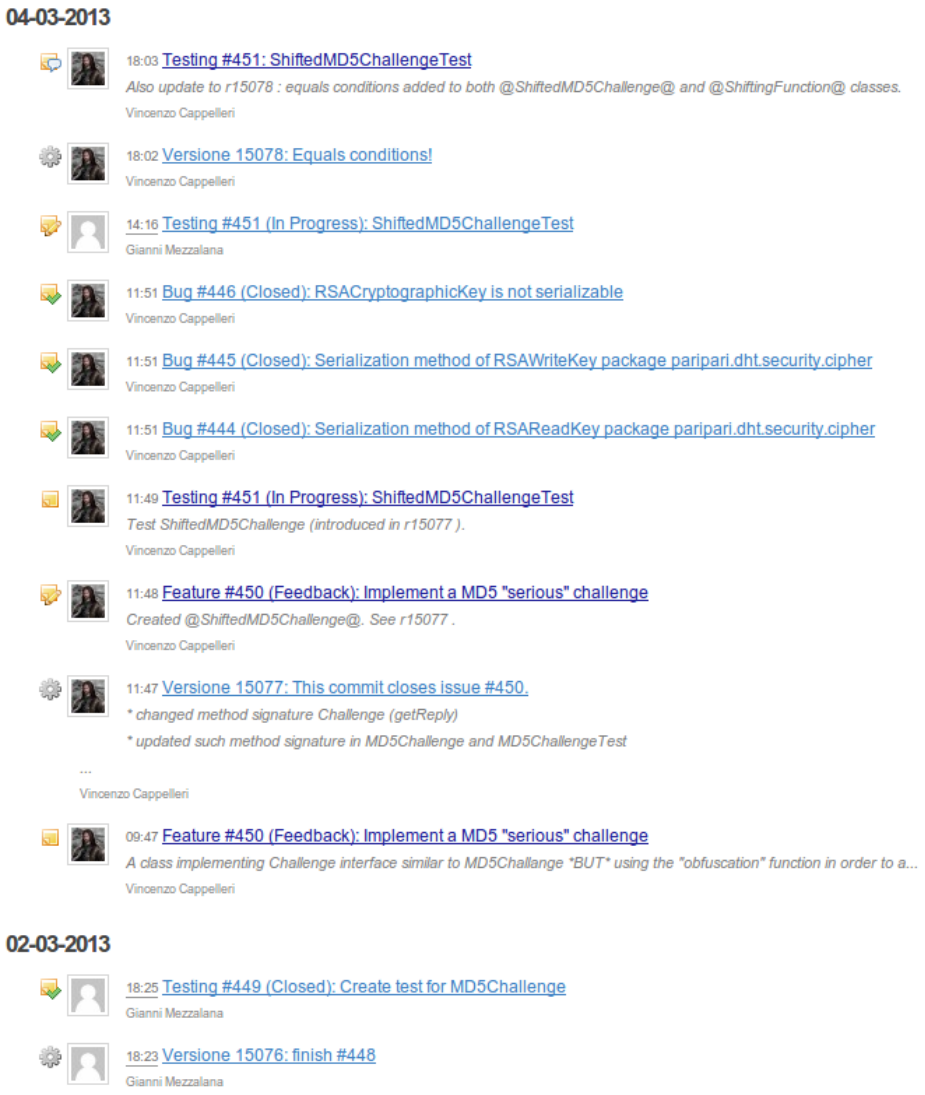


Figura 2.5: Redmine - Attività

Per questo ogni modifica di progetto va documentata e collegata alla *release* di riferimento e inserendola, dove possibile, come attività subordinata ad una attività principale. Così facendo è possibile avere una visione completa dell'andamento dello sviluppo software. Inoltre si facilita la comprensione del software e delle relative scelte progettuali, fondamentali per un eventuale nuovo inserimento nel *team* di sviluppo.

relazionali. Tratto da `http://it.wikipedia.org/wiki/Repository`, (visitato il 6 marzo 2013).

[5]Conosciuto anche come *"svn"* è un sistema di controllo di versioni di programmi.

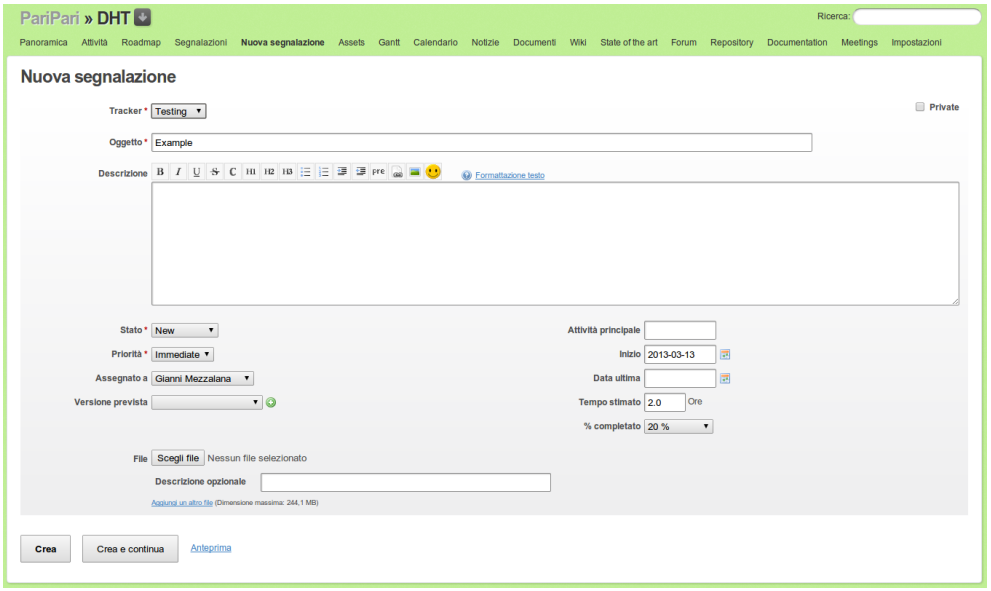


Figura 2.6: Redmine - Nuova Segnalazione

Come professato da Kent Beck con *eXtreme Programming Explained: Embrace Change* [6] , il ruolo dei test è importantissimo, ma lo sono anche i commenti del codice e la documentazione. Fondamentale è l'applicazione del metodo di sviluppo software *Test Driven Develop* (T.D.D.) ossia un metodo di sviluppo software guidato dai test (vedi sezione 2.2).

Un software ben commentato e lineare riduce la fase iniziale di apprendimento aiutando nella comprensione un neo-programmatore che legge per la prima volta quel codice. Allo stesso modo, riduce i tempi di debug [7] e facilita programmazione e *refactoring* del software stesso. Un buon ciclo di sviluppo software, che utilizza i test come guida per la scrittura di codice, minimizza l'inattività e massimizza il risultato nell'unità di tempo. Di seguito (capitolo 5) sarà presentato il lavoro di affiancamento nella riscrittura del modulo DHT, con approfondimenti sul *testing* e in particolare nei Test di Modulo (*Customer Test*), Test di Unità (*Unit test*), *Beta Test*, e i Test di Integrazione (*Integration Test*).

[6] In molti testi di ingegneria del software non solo ci si riferisce a questo libro, ma lo si indica come un libro evangelico che descrive il processo e le esperienze di programmazione estrema, Sommerville, I., *Ingengeria del software*, 2007, p.87.

[7] Il debug o meglio il processo di *debugging è una conseguenza del collaudo; ossia, quando un caso di prova rileva un errore, il debugging è il processo che porta all'eliminazione di quest'ultimo.* Pressman, Roger S., *Principi di Ingegneria del software*, 2004, p.413

2.2 T.D.D.

Ogni progetto software abbraccia una sua particolare strategia di sviluppo. Esistono numerosi *"modelli dei processi software"* o *"paradigmi dei processi software"*, ma tutti secondo Sommerville[8] hanno in comune:

- specifiche
- progettazione
- convalida
- evoluzione

Tra i paradigmi classici ritroviamo il modello a cascata, il modello evoluzionistico, il modello a componenti. Tra questi verranno approfonditi i paradigmi incrementali o ciclici. Tali paradigmi videro il loro sviluppo alla fine degli anni '80 dopo l'avvento della programmazione ad oggetti. Uno dei più rinomati paradigmi ciclici è il paradigma incrementale che prevede una forte comunicazione programmatore-utilizzatore e il rilascio frequente di versioni dimostrative parziali[9]. Questo significa da una parte frequenti riunioni tra programmatori e utilizzatore (se non addirittura la presenza fissa di un rappresentante degli utilizzatori in sede di sviluppo software), dall'altra la presentazione al cliente di ogni versione dimostrativa del programma (*demo*) che introduca anche piccole funzionalità.

Tra la fine del XX secolo e l'inizio del XXI, questo modello fu ripreso e rielaborato da Kent Beck proponendo la *"programmazione estrema"* che unisce i concetti della programmazione incrementale con quelli della programmazione orientata ai test (T.D.D.). Si aggiunge quindi un controllo sul codice scritto tramite svariate

[8]Sommerville, I., *Ingengeria del software*, 2007, p.87

[9]Cfr. *Ciclo di vita del software*, da `http://it.wikipedia.org/wiki/Ciclodivitadelsoftware`, (visitato il 17 febbraio 2013).

tipologie di test, introducendo il rilascio delle versioni dimostrative con grande frequenza. Si inizia a parlare quindi di programmazione difensiva[10].

L'obiettivo della TDD è sviluppare software minimizzando i difetti. Si procede definendo un processo di verifica continuo e iterativo all'interno del processo di sviluppo software, non solo con l'obiettivo di individuare errori prima che divengano difetti, ma anche con l'obiettivo di guidare i programmatori verso le giuste scelte. Un banale esempio di quanto detto è quello che ha riguardato la scrittura della documentazione *Javadoc* [11] della Nuova DHT.

Come si vedrà nei capitoli 4 e 5 spesso per costruire *White box Test* di integrazione è necessaria una nutrita documentazione in grado di spiegare con esattezza le interazioni fra le varie classi. Senza tale documentazione è semplicemente impossibile creare dei test. In termini estremamente generali si può dire che il test è uno strumento per verificare che il funzionamento dell'oggetto sotto esame sia corretto. Se non è ben chiaro quale funzionamento debba avere un determinato modulo, non solo non è possibile testarlo, ma non è nemmeno possibile scrivere il codice del modulo stesso. Posto in questi termini il ragionamento sembra quasi banale; tuttavia spesso ancora oggi leggiamo funzioni,

[10] Nel Blog di Andrew Phillips, *Software Develop* `http://devmethodologies.blogspot.it/2012/05/defensive-programming.html` (visitato il 18 febbraio 2013), un articolo riguardante *Defensive Programming* indica il libro di Kernighan, B.W. e Ritchie D.M., *The C Programming Language*, 1° ed., Prentice Hall Software Series, 1978, come il primo scritto tecnico in cui si fa riferimento alla "Programmazione Difensiva". Nel loro testo Kernighan and Ritchie (p.53) descrivono la programmazione difensiva come l'insieme delle tecniche per prevenire errori, basate su previsioni. Nel contesto infatti viene spiegato come, adottare accorgimenti difensivi apparentemente inutili (*break* nell'ultimo *case* di uno *switch*), potrebbe essere l'unico modo per non creare un errore (inserimento di un nuovo *case* dopo l'ultimo).

L'idea di base della programmazione difensiva consiste pertanto nel non considerare mai affidabile una porzione di software, un dato da elaborare o in genere qualsiasi tipo di oggetto interno o esterno che sia.

[11] JavaDoc nacque come strumento interno utilizzato dai ricercatori della Sun che stavano lavorando alla creazione del linguaggio Java e delle sue librerie; la grande mole di sorgenti spinse alcuni membri del team a creare un programma per la generazione automatica di documentazione HTML. Tratto da *Javadoc* `http://it.wikipedia.org/wiki/Javadoc`, (visitato il 18 febbraio 2013).

metodi o addirittura interi programmi non opportunamente documentati. Specialmente nei primi anni dello sviluppo di PariPari questa era una pratica spesso adottata da studenti/programmatori non ancora istruiti sulla TDD e sui principi di ingegneria del software.

Con la T.D.D. si pongono le basi per la *"Garanzia di Qualità del Software"*, SQA. SQA parte dal presupposto che il software è un oggetto matematico e pertanto si possono definire rigorosamente sintassi e semantica. Equivalentemente questo può essere proiettato nello sviluppo del software. Un software sviluppato secondo criteri rigorosi, con alla base approcci corretti nella specifica dei requisiti, definendo sistema, scopo, vincoli, logiche, interfacce e funzionamento, risulterà con buona probabilità un software corretto [12] .

SQA è un'attività ausiliaria allo sviluppo software che si applica a tutte le fasi del processo di costruzione del software e fornisce una guida per l'istituzione di un protocollo di garanzia della qualità, che viene fortemente raccomandato dall'Istituto degli Ingegneri Elettrici ed Elettronici (*Institute of Electrical and Electronic Engineers*, I.E.E.E. [13])[14].

Come, dal punto di vista lavorativo, una *softwarehouse* dovrebbe mirare a produrre software *"corretto"*[15], un gruppo di sviluppatori come quello di PariPari dovrebbe applicare piani di SQA per rendere più affidabile il proprio sviluppo.

[12]Secondo Pressman (Pressman, Roger S., *Principi di Ingegneria del software*, 2004, p.648), definendo rigorosamente le specifiche si possono applicare tecniche matematiche per provare la correttezza di un programma, che in altri termini è la prova che il programma soddisfa i requisiti.
In realtà la *"Tesi di Dijkstra"* afferma che il test può indicare la presenza di errori, ma non ne può garantire l'assenza. Un corollario diretto è che il software *error-free* non esiste o non è certificabile. Tratto da `http://it.wikipedia.org/wiki/Test`, (visitato il 21 febbraio 2013).

[13] Un'associazione internazionale di scienziati professionisti con l'obiettivo della promozione delle scienze tecnologiche. Ad oggi l'IEEE annovera più di 320'000 membri in 150 nazioni; comprende tecnici, ingegneri, ricercatori, studenti, professori, nonché amatori di tutto il mondo nel settore elettrotecnico ed elettronico. Le pubblicazioni dello IEEE sono il 30% della bibliografia e documentazione ingegneristica globale e coprono quasi tutti gli aspetti dell'elettronica e dell'informatica moderna. Inoltre IEEE ha definito oltre 900 standard industriali. Da Institute of Electrical and Electronics Engineers `http://it.wikipedia.org/wiki/InstituteofElectricalandElectronicsEngineers`, (visitato il 17 febbraio, 2013).

[14]Cfr. Pressman, Roger S., *Principi di Ingegneria del software*, 2004, pp. 648,655,656

[15]vedi nota 12 a p.16

In particolare, molte delle raccomandazioni SQA sono già implementate e seguite per effetto della politica interna universitaria e del nuovo sistema di gestione *"redmine"* (vedi p.7). Tra queste troviamo ad esempio i ruoli, le responsabilità, i metodi, il controllo, la registrazione delle modifiche.

Produrre un software con una garanzia di qualità significa produrre un software seguendo determinate politiche di sviluppo e progettazione con l'obiettivo di migliorare correttezza e affidabilità dello stesso.

Secondo Pressman[16] l'affidabilità viene valutata misurando la frequenza e la gravità dei guasti, la precisione dei risultati di output, il tempo MTTF (Mean-Time-To-Failure), la capacità di recuperare una situazione di guasto e la prevedibilità del programma.

L'affidabilità è definita come la capacità di rispettare le specifiche tecniche di funzionamento nel tempo[17].

Per quanto riguarda un software quanto detto si traduce statisticamente come la "probabilità di un software di operare senza guasti in un dato ambiente per un dato tempo" [18].

Prima di addentrarsi in concetti successivi è necessario chiarire cosa si intende per "guasti", "errori" e "avarie".

Una non conformità strutturale o logica di qualche componente del sistema (avaria) può provocare degli errori. Un errore è una transizione dello stato globale del sistema non prevista, che potrebbe portare il sistema definitivamente fuori delle specifiche di funzionamento, interrompendo la disponibilità dei servizi forniti e quindi generando guasti [19].

In un software modulare con le ambizioni di PariPari, è necessario che il codice sia strutturato in modo tale da massimizzare non solo l'affidabilità ma

[16] Pressman, Roger S., *Principi di Ingegneria del software*, 2004, p.255

[17] Affidabilità `http://it.wikipedia.org/wiki/Affidabilità`, (visitato il 17 febbraio 2013).

[18] Musa J. D., A. Ianninoe K. Okumoto, *Engeneering and Managing Software with Reliability Measure*, McGraw-Hill, 1987. in Pressman, Roger S., *Principi di Ingegneria del software*, 2004, Op.cit p. 651.

[19] Affidabilità `http://it.wikipedia.org/wiki/Affidabilità`, (visitato il 17 febbraio 2013).

anche la disponibilità, ossia la probabilità che un programma operi secondo i requisiti in un dato istante[20]. Questo è reso in parte possibile dalla natura modulare del progetto, ma specialmente dall'architettura Peer-to-Peer della rete su cui poggia.

Valori di disponibilità ed affidabilità elevati sono senza ombra di dubbio traguardi da perseguire e raggiungere per ogni software. La frammentazione temporale del gruppo di sviluppo di PariPari e l'inesperienza dei programmatori forza un nuovo approccio per la creazione del software.

Si è deciso pertanto di puntare essenzialmente su tre fasi chiave riprese da strategie e metodologie "agili"[21] come TDD, modelli evolutivi ed *Extreme Programming*[22]:

- progettazione di interfacce e documentazione
- scrittura del codice e *testing*
- collaudo e *testing*

È evidente quindi che un ruolo chiave nello sviluppo del software viene ricoperto proprio dal *testing*.

2.3 Perché il Testing

L'obiettivo di sviluppare un software corretto e senza errori viene perseguito seguendo un processo di sviluppo TDD, con programmazione a più mani, scri-

[20] Pressman, Roger S., *Principi di Ingegneria del software*, 2004, p.652

[21] Nell'ingegneria del software, per metodologia agile si intende un particolare metodo per lo sviluppo del software che coinvolge quanto più possibile il committente, ottenendo in tal modo una elevata reattività alle sue richieste. Tratto da Metodologia agile, `http://it.wikipedia.org/wiki/Metodologia_agile`, (visitato il 17 febbraio 2013).

[22] L'Extreme Programming è una "metodologia agile" e un approccio all'ingegneria del software formulato da Kent Beck, Ward Cunningham e Ron Jeffries. Tratto da Extreme Programming, `http://it.wikipedia.org/wiki/Extreme_Programming`, (visitato il 17 febbraio 2013).

vendo codice commentato e documentato. Inoltre sono previste riunioni frequenti, che aiutano a formalizzare le scelte di programmazione e progettazione individuando i punti deboli.

Questi accorgimenti portano anche ad un altro vantaggio: lo sviluppo rapido del software. Impiegare meno tempo nella scrittura del codice per dedicarlo al *testing* significa diminuire il tempo effettivo di sviluppo software a favore del processo di controllo, per assicurarsi che il codice scritto sia corretto.

A tal proposito una falsa credenza che si è cercati di sfatare (anche con delle apposite lezioni agli sviluppatori di PariPari) è quella che il *testing*[23] sia una perdita di tempo. Potremmo infatti (proprio tentando di riassumere i concetti presentati in queste lezioni) pensare erroneamente all'evoluzione del software come ad un processo lineare, senza badare troppo a *testing*, *refactoring* e controllo del codice scritto. Procedendo in questo modo si entrerà in una spirale senza uscita con termine ultimo il fallimento (vedi figura 2.7). Lo sviluppatore è in ritardo sulla tabella di marcia pianificata, quindi tende a ridurre il tempo riservato al *testing*. Questo con grande probabilità porterà ad un codice più fragile, che come unica conseguenza avrà la necessità di essere corretto. La correzione di un software non testato è complessa in quanto non si può a priori considerarne corretta una data porzione; pertanto la correzione di codice non testato interessa tutto il software. Il risultato è l'accumulo di un ulteriore ritardo sulla tabella di marcia. Per recuperare tempo si potrebbe scegliere di ridurre ulteriormente il *testing*, aggravando paradossalmente il ritardo.

Alla luce di quanto descritto, per il progetto PariPari si è scelto di enfatizzare il concetto di T.D.D. con riunioni e documenti che mettessero in evidenza il ruolo del *testing*.

[23] Procedimento utilizzato per individuare le carenze di correttezza, completezza e affidabilità delle componenti software in corso di sviluppo. Consiste nell'eseguire il software da collaudare, da solo o in combinazione ad altro software di servizio, e nel valutare se il comportamento del software rispetta i requisiti. Tratto da Testing, `http://it.wikipedia.org/wiki/Testing`, (visitato il 18 febbraio 2013).

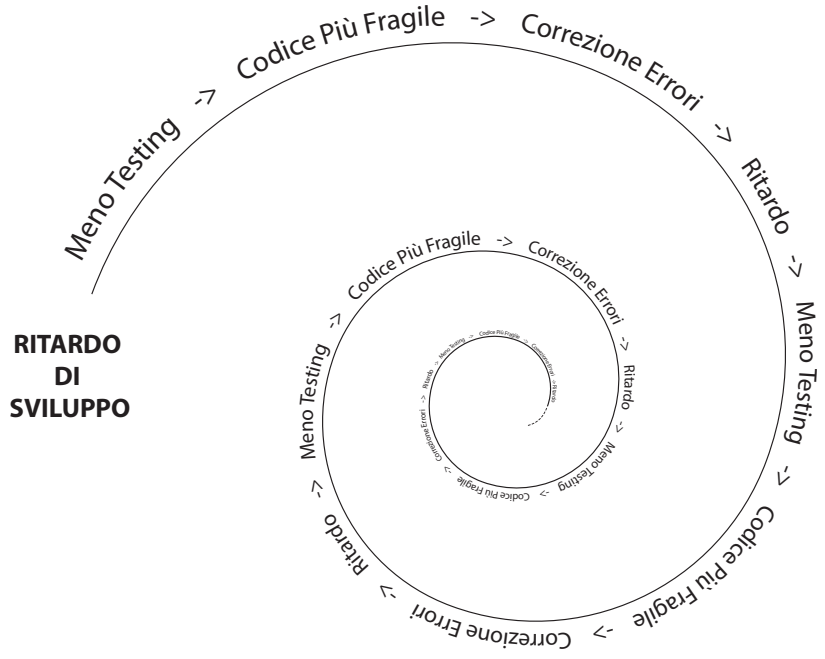


Figura 2.7: Spirale di recessione.

Capitolo 3

DHT

Una Tabella Hash Distribuita, DHT (*Distributed Hash Table*) identifica un particolare sistema di immagazzinamento e reperimento delle informazioni che sfrutta una rete di calcolatori. All'interno di una DHT ogni informazione è legata ad una chiave univoca. Quindi in una DHT qualsiasi informazione si trova all'interno di un pacchetto contenente una coppia <chiave,valore>. L'analogia con una tabella Hash convenzionale è forte. Una tabella Hash è una rappresentazione di una funzione *Hash*: una funzione non iniettiva che mappa gli elementi di un dominio finito (valori) in un codominio (chiavi) di cardinalità inferiore. La differenza sta appunto nel fatto che queste coppie <chiave,valore> sono distribuite il più possibile equamente tra tutti i partecipanti alla rete. In una DHT ogni nodo al momento dell'accesso è identificato in modo univoco da un ID. Il dominio di questo ID identificativo è lo stesso delle chiavi salvate nella rete e corrisponde al dominio della funzione Hash. Questo permette di individuare i nodi responsabili di una determinata risorsa, ovvero quei partecipanti aventi ID "più vicino" alla chiave memorizzata. Pertanto in una DHT è fondamentale definire una metrica interna per misurare la distanza tra due Hash: l'ID (hash di un nodo) e la chiave (hash di un valore).

3.1 Scopo della DHT

La nuova DHT dovrà garantire funzionalità di ricerca (search) e salvataggio (*store*) di informazioni in maniera analoga a quella prevista da Kademlia, e quelle previste dal precedente progetto di rete DHT per PariPari. In aggiunta la Nuova DHT per PariPari dovrà prevedere la gestione di moduli per garantire autenticità/incorruttibilità dei dati, nonché il controllo sull'accessibilità agli stessi. L'idea è quindi di creare un sistema di permessi che possa essere escluso completamente da chi non volesse usufruirne, in modo da azzerare l'*overhead* per "servizi pubblici" (es. file sharing).

3.2 Struttura

Il nuovo modulo DHT di PariPari, si struttura in 4 moduli principali con i quali ci si interfaccia con opportune API studiate e testate appositamente per rendere snello e sicuro l'utilizzo di DHT. I moduli principali sono classi create come istanze statiche all'interno del modulo DHT. Nello specifico sono *"Permissions Manager"*, *"Network Manager"*, *"Disk Manager"*, *"Asyncronous Request Manager"*. L'iterazione tra queste classi è visibile in figura 3.1 dove viene presentata la struttura del modulo DHT.

Prima di descrivere i principali moduli che compongono il plug-in DHT, si vuole presentare la struttura di una risorsa DHT, ossia l'oggetto che viene salvato nella Rete e che contiene tutti i dati e i metadati necessari per il funzionamento della stessa DHT. Si possono riconoscere due campi di dati fondamentali. Entrambi contengono dati e metadati. Il primo che identifichiamo come *DHTKey*, è appunto la chiave univoca contenente tutti i tag (chiavi di ricerca) attraverso i quali si può risalire alla *entry* quando la si ricerca nella DHT . Il secondo è DHT-Value, che contiene il valore in formato binario del dato da salvare (ad esempio il file in questione).

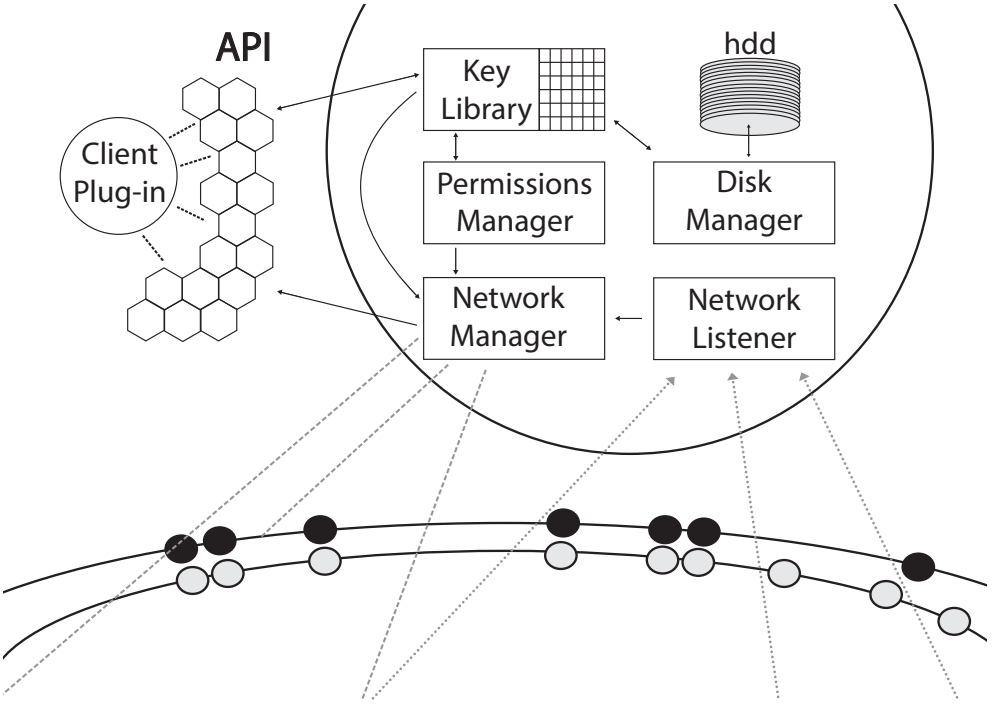


Figura 3.1: La struttura di DHT.

Questa descrizione è valida solo dal punto di vista dell'utente (o plug-in utilizzatore), sono infatti omesse le informazioni relative ai permessi, come le chiavi e le *challenge*[1]. In sezione 3.2.1 viene invece presentata nella sua interezza una *DHTResource* completa di tutte le informazioni, anche quelle necessarie alla gestione degli accessi che saranno ad uso esclusivo del nodo ospitante la risorsa.

3.2.1 Disk Manager

Il compito principale di questa classe è mantenere su disco una tabella con chiavi e le relative coordinate nella rete, in modo da poter recuperare i file. In realtà manterrà una serie di risorse *DHTResource* con la struttura visibile in figura 3.3, composte da:

DHTKey con le relative parole chiave di ricerca (*tag*), che rappresenta la chiave univoca della risorsa.

[1] Affrontate nella sezione 3.3

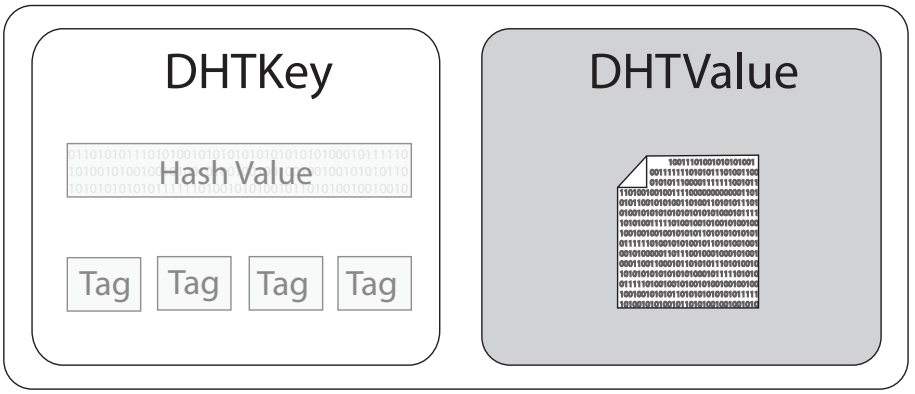


Figura 3.2: La struttura di una Entry DHT (lato nodo utilizzatore).

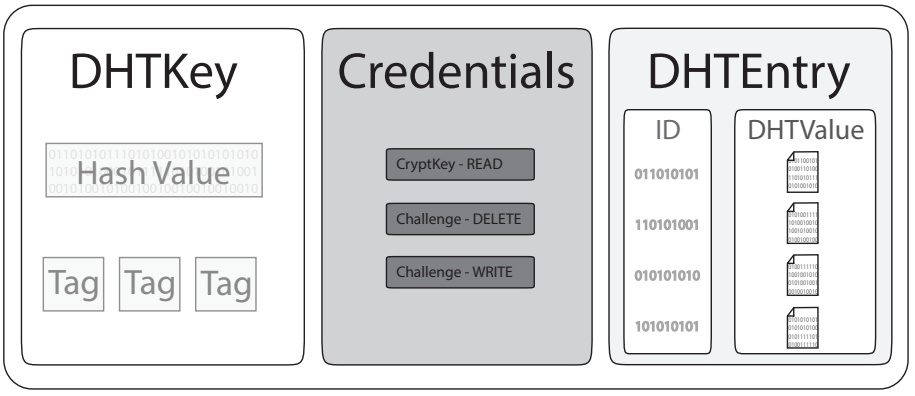


Figura 3.3: La struttura di una Entry DHT (lato nodo ospitante).

Credentials le eventuali *challenge* e chiavi di lettura/scrittura, in genere, se prevista, tutta l'implementazione del sistema di permessi.

DHTEntry uno o più oggetti di questo tipo, che a loro volta contengono informazioni sul nodo proprietario (*ID*) e il dato stesso (*DHTValue*).

Si può fare quindi una netta distinzione tra i metadati (*DHTKey*,*Credentials*) e i dati (*DHTValue*) trattati dal *Disk Manager*.

3.2.2 Resource Controller

Thread che controlla le varie risorse presenti nell'archivio di chiavi (*KeyLibrary*). Secondo un algoritmo di controllo si decide se salvare la risorsa su disco o meno. In prima analisi si è stimato che il controllo possa avvenire con cadenza 10, 20 e 30 minuti. Se alla fine di questo ciclo la risorsa non ha subito modifiche viene salvata su disco passando attraverso il *Disk Manager*. Il fine del salvataggio su disco è sicuramente quello di alleggerire il sistema occupando meno *RAM* possibile. Lo svantaggio inevitabile sarà la diminuzione dell velocità di risposta del sistema.

3.2.3 Permission Manager

Modulo di DHT che permette la gestione dei permessi. Si occupa di criptare, decriptare e condividere chiavi. È compito di questo modulo, attraverso le classi presenti nel package *"paripari.dht.security.cipher"* fornire la coppia di chiavi pubblica e privata da usare per la codifica a seconda dell'algoritmo scelto.

3.2.4 Key Library

Classe con una forte interazione con il *Disk Manager*. In particolare si riesce ad individuare un modulo che la compone: il *Key Storer*, che si occupa di tener

traccia delle coppie chiave, valore: <*K,V*>. La chiave univoca di un determinato oggetto è *"K"*, mentre *"V"* è l'oggetto stesso. La chiave non si limita a contenere solo i dati ma anche:

- i permessi associati alla chiave
- tempo di scadenza
- lista di parole chiave (*tag*)
- eventuali altre informazioni addizionali

Il *Key Library* contiene anche un altro modulo: *"ExpiryResourceService"*, che si occupa della eliminazione di chiavi con "data di scadenza" (*"expiry date"*) scaduta. L'eliminazione procede solo nel caso in cui l'ultimo valore rimasto sia scaduto.

3.2.5 Serializer

Questo modulo si occupa essenzialmente della decodifica dei messaggi in modo tale da tradurli in un flusso di *byte*. In realtà la serializzazione viene implementata a livello di classe dato che ogni valore nella DHT deve essere un oggetto che implementa `paripari.api.dht.DHTValue`. Infatti attraverso i metodi `byte[]toByteArray()` e `DHTValuefromByteArray(byte[])` dell'interfaccia *DHTValue* si riesce a convertire rispettivamente da oggetto ad *array* di *byte* e da *array* di *byte* ad oggetto.

3.2.6 Network Manager

Gestisce la comunicazione con gli altri nodi della DHT tenendo traccia dei plug-in che effettuano richieste o inviano dati sulla Rete. Prevede l'utilizzo di primitive di ricerca e salvataggio sull'intera DHT ed astrae l'implementazione effettiva

della particolare rete in uso utilizzando un oggetto di tipo *Protocol*, che contiene l'elenco completo delle primitive di rete che ogni DHT di PariPari dovrà realizzare.

3.2.7 DHT Layout

Modulo che permette di effettuare ricerche sincrone e asincrone nella rete DHT. Lo scopo non è ottenere oggetti o valori associati alle chiavi, ma piuttosto identificare i nodi responsabili di tali chiavi, o che per lo meno abbiano traccia di tali nodi nelle loro tabelle di *routing*.

3.3 Sistema di Permessi

Il sistema di permessi della DHT di PariPari è la vera e propria innovazione rispetto ai vecchi progetti. Il punto forte di questo sistema oltre alla semplicità vuole essere quello di essere completamente disabilitato. Il sistema di permessi di PariPari si colloca nel package ‘‘`paripari.dht.security`’’ ed è gestito dalla classe *"Permission Manager"*.

Ad ogni chiave nella DHT è associato un insieme di 3 permessi: *Read*, *Write*, *Delete*. Ognuno di questi permessi può assumero un valore che può essere *Restricted* o *Unrestricted*. Nel caso di scrittura(*"Write"*) o lettura(*"Read"*) per poter procedere si dovrà superare un controllo di accesso. Questo può essere la risposta ad una *Zero-Knowledge challenge*[2](nel caso di scrittura) o semplicemente una chiave asimmetrica (lettura) con la quale decriptare il contenuto. È compito della DHT creare nuove *challenge* o coppie di chiavi crittografiche per ogni nuova risorsa da salvare nella rete (vedi 3.2.3, rendendo completamente trasparente ai client l'intera complessità sottostante. Questo viene fatto per praticità (gli svi-

[2]Un sistema di verifica per concludere se un interlocutore è a conoscenza della risposta ad una domanda, senza mai fornire alcuna informazione sulla risposta stessa.

luppatori di client non devono preoccuparsi dei dettagli di funzionamento della crittografia), ma anche per diminuire eventuali minacce all'integrità del sistema.

Quando un nodo vuole effettuare uno *store* (scrittura) di una coppia <*K,V*> deve dichiarare il set di permessi che intende applicare alla chiave che vuole salvare. Se la chiave non è presente nella DHT la richiesta viene accettata e viene salvata l'intera coppia chiave-valore. Diversamente il valore "V" viene aggiunto alla tabella dei valori per quella chiave "*K*" solo se i permessi coincidono e solo se il nodo supera le *challenge* impostate. Nel caso di modifica, si procede allo stesso modo, ma il valore presente viene sovrascritto. Ogni nodo può sovrascrivere solo il proprio valore salvato per una determinata chiave, e mai quelli degli altri[3].

Read	**Write**	**Delete**	**Tipo di Chiavi**	**Esempio di Utilizzo**
R	R	R	K, CHS, CHC	Gruppo privato con amministratori
R	R	U	K, CHS	Gruppo privato
R	U	R	K, CHC	FTP dottorato
R	U	U	K	Voto segreto
U	R	R	CHS, CHC	Annunci cancellabili da moderatori
U	R	U	CHS	Annunci non cancellabili da moderatori
U	U	R	CHC	Forum con moderatori
U	U	U	-	File Sharing

R = Restricted
U = Unrestricted
K = Chiave Crittografica
CHS = Challenge di Scrittura
CHC = Challenge di Cancellazione

Tabella 3.1: Possibili Permessi di protezione.

Tabella tratta dalla *WIKI* di progetto alla pagina *Permessi*

Se i permessi non coincidono, o il nodo fallisce nel rispondere alle *challenge* la richiesta viene rifiutata.

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 una chiave, potrà eliminare la chiave e

[3]Come visibile in figura 3.3, per una chiave "K" possono esistere diversi valori salvati "V".

tutti i valori associati dalla DHT o aggiornarli. In ogni caso un nodo può sempre cancellare il proprio valore associato ad una chiave. Va notato che mentre per *Read* e *Write* il valore *Unrestricted* significa che chiunque può leggere o scrivere valori su quella chiave, per *Delete* significa che nessuno può cancellare la chiave altrui. Alcuni esempi di permessi configurabili si possono vedere in tabella 3.1. Nel caso di servizi che non richiedano l'applicazione di permessi particolari non verranno effettuate *challenge* di alcun tipo né crittazioni, per cui l'*overhead* di tutto questo meccanismo viene drasticamente contenuto.

In questa implementazione di DHT la coppia <*K,V*> è ben definita. In *"K"* è salvata la sola chiave in chiaro, con i relativi *Tag* su cui è possibile effettuare ricerche. La parte *"V"* invece può essere criptata. Tramite la classe *"Permission Manager"* è possibile decriptare questo valore per ottenere dati fruibili.

In PariPari i dati vengono elaborati da classi differenti in base al fatto che debbano o meno essere criptati. Il *"Network Manager"* è infatti responsabile di attuare la procedura per la codifica/decodifica oppure lasciare passare il pacchetto senza alcuna fase di cifratura. Quindi l'intero plug-in *"Permission Manager"* può essere attivato arbitrariamente (da *"Network Manager"*) a seconda dei casi. Questo si traduce in un funzionamento adattivo di DHT che può dinamicamente attivare il modulo di *security* o meno a seconda dell'esigenza.

3.4 Sicurezza

Durante lo studio di DHT si sono evidenziate problematiche riguardanti la sicurezza. Non si tratta di nuovi problemi ma situazioni conosciute che affliggevano anche i vecchi progetti DHT e riguardavano possibili vulnerabilità collegate all'accesso alla Rete e alla scelta della chiave identificativa per DHT.

Di seguito saranno presentati alcuni problemi con le possibili soluzioni formalizzate nella piattaforma *Redmine*. Le problematiche che verranno presentate

sono ancora aperte, di proposito non si è ancora definito le soluzioni da adottare per rispettare il modello di "programmazione agile". L'obiettivo infatti è produrre qualcosa di funzionante nel minor tempo possibile.

3.4.1 Login

L'accesso a PariPari è subordinato all'inserimento di un nome utente. Ogni nome utente corrisponde ad un *"ID utente"*. Il primo ostacolo sono le registrazioni fasulle. Senza un minimo di controllo per evitare registrazioni fasulle, si potrebbe compromettere il funzionamento dell'intera Rete: accedendo con un nome utente già in uso potrebbero venir meno tutte le garanzie del modulo permessi. Una possibile soluzione è una password che metta parzialmente al sicuro da furti di identità, anche se questo aprirebbe il problema del verificare la correttezza della password.

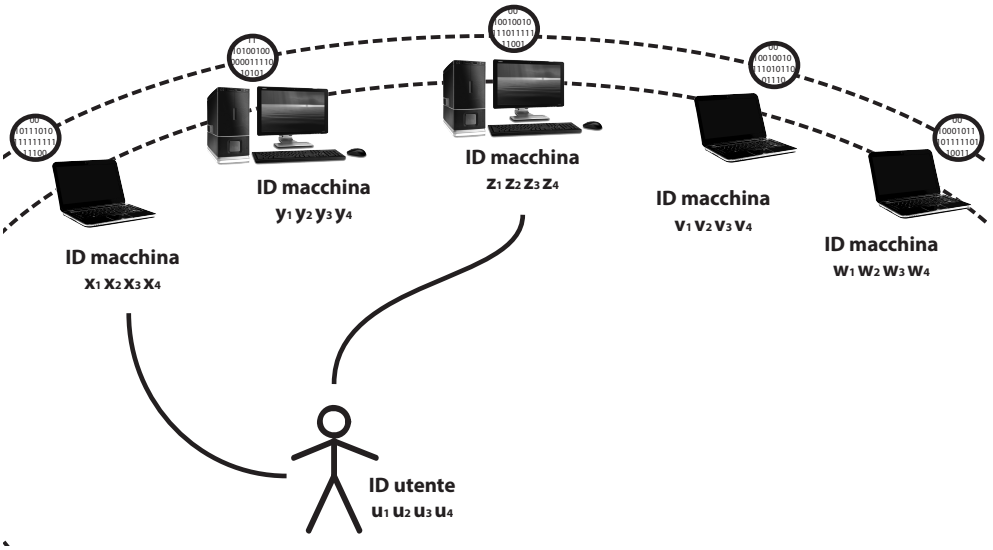


Figura 3.4: Esempio di un utente con più macchine.

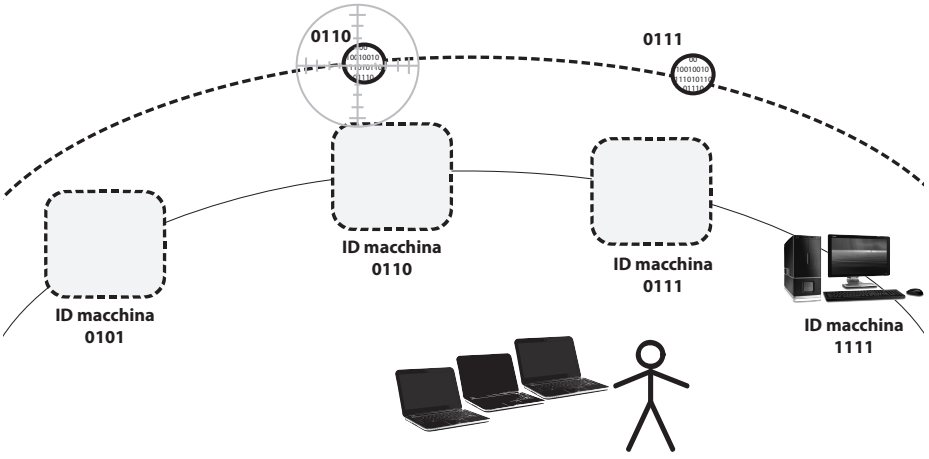
Ogni macchina che si collega a PariPari otterrà un codice identificativo chiamato *"ID macchina"*, quindi ad ogni utente (ID utente) possono corrispondere più ID macchina. Nell'istante in cui una macchina si collega alla Rete le vie-

ne assegnato in modo casuale un ID macchina: una chiave pubblica della DHT. Le risorse con valore *Hash* più vicino a questa chiave pubblica verranno prese in carico dalla macchina che diverrà pertanto un riferimento per tutti i nodi che chiedono di effettuare operazioni con tali risorse. Una concreta vulnerabilità è la possibilità di aggirare il sistema di gestione casuale dell'assegnazione ID macchina. Scegliere accuratamente un ID macchina permetterebbe di essere responsabile di determinate risorse: quelle con chiave *hash* più vicina all'ID macchina.

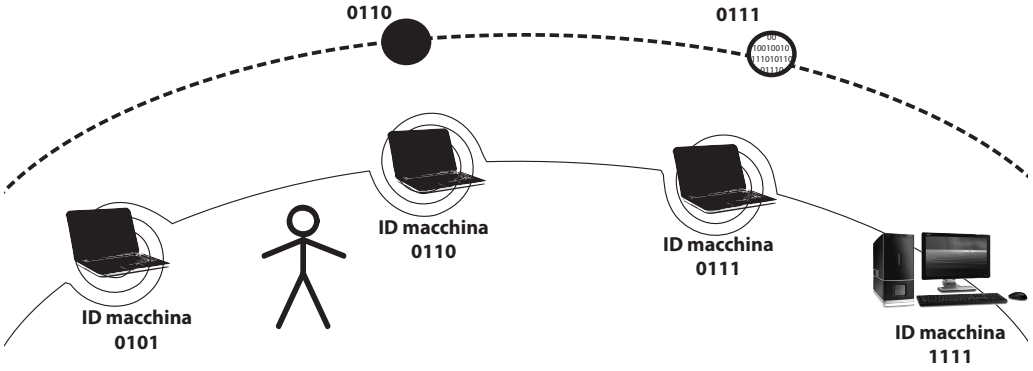
Il pericolo è che alcune macchine malintenzionate potrebbero inserirsi in punti specifici della DHT al fine di oscurare determinate risorse. Un esempio di quanto detto è descritto in figura 3.5, dove un utente malintenzionato, volendo oscurare la risorsa il cui hash è 0110, cercherà di inserirsi nella rete con più macchine, scegliendo come ID macchina valori vicini all'hash della risorsa (0110). Nel caso descritto, con tre macchine a disposizione si sceglieranno come ID macchina i valori 0110, 0111, 0101. Un primo modo per contrastare questa vulnerabilità potrebbe essere la replicazione delle chiavi. L'esistenza di più copie della risorsa sparse nella Rete renderebbe più difficile l'occultamento della risorsa, ma il problema non si eliminerebbe definitivamente: se la replicazione fosse a "X" livelli, "X" macchine malintenzionate sarebbero una concreta minaccia. La replicazione quindi non è la risposta a questo problema, ma è sicuramente necessaria per altri scopi (vedi p.33). Un sistema di rotazione delle chiavi potrebbe essere invece la risposta per eliminare tale vulnerabilità.

3.4.2 Rotazione delle chiavi

L'idea è molto semplice: rendere indipendente il posizionamento di un nodo nella DHT rispetto ad una data risorsa. Rotazione quindi si traduce in ri-mappare le risorse presenti nella rete con un *offset* da applicare all'*hash* di ricerca della risorsa.



(a) Individuazione dell'ID risorsa.



(b) Collocamento nella posizione di oscuramento.

Figura 3.5: Esempio di inserimento mirato ad oscurare delle risorse.

I punti chiave di questa procedura che bisogna definire sono:

- sincronizzazione tra i nodi della rete
- algoritmo di generazione dell'*offset*
- frequenza di rotazione

Dato che l'*offset* sarà sempre diverso e sarà cambiato con una data frequenza, i nodi devono essere sincronizzati. Infatti in un dato istante tutti i nodi devono essere d'accordo su quale valore di *offset* applicare a un eventuale ricerca. Una idea potrebbe essere di utilizzare una primitiva di *broadcast* per comunicare a

tutti i nodi della rete un *token*. Tale *token* dovrebbe essere generato solo da alcuni nodi della rete appositamente elencati nel *token* stesso. All'interno del *token* ogni nodo accreditato deve inserire il proprio valore (generato casualmente) di *offset* relativo da applicare. L'insieme di questi valori parziali viene interpretato come una stringa binaria, e risulta essere l'*offset* da applicare tramite XOR ad ogni ricerca o salvataggio.

3.4.3 Replicazione delle chiavi

Gli obiettivi della replicazione delle chiavi sono sicuramente aumentare la robustezza e le prestazioni. Si vuole evitare infatti che delle risorse possano scomparire dalla rete per la presenza di nodi malevoli o per semplice *churn*[4]. Con la replicazione delle risorse si pone il problema di formalizzare le procedure per mantenere le copie nella rete: *refresh*. Ci sono sostanzialmente due strade percorribili. Una prima in cui ogni nodo proprietario di una risorsa si assume tutte le responsabilità: tramite delle ricerche controlla il livello di replicazione, nel caso fosse necessario genera della nuove repliche e si occupa di salvarle nella Rete. La seconda opzione è di *autorefresh*: le repliche che definiamo vecchie prima della rotazione dovranno generare le repliche che indichiamo con nuove. Distinguiamo due fasi precedenti alla rotazione, una prima in cui le repliche "vecchie" contattano i futuri nodi ospitanti segnalando l'ID della risorsa che dovrà essere replicata. Successivamente ogni "nuovo" nodo ospitante effettuerà una ricerca della risorsa con l'ID pre-rotazione, ottenendo un certo numero di risultati (al più tutti). Il valore della risorsa corretto sarà quello che si presenta con maggiore frequenza (Tratto da *State of the art*, `http://verona.dei.unipd.it/redmine/projects/newdht/wiki/Stateoftheart`, visi-

[4]Si tratta del continuo ciclo di accesso e disconnessione dei nodi tipico di una rete *P2P*. Viene anche definito come il fenomeno di "partecipazione dinamica" dei pari, in una rete *P2P*. Tratto da Daniel Stutzbach, Reza Rejaie, University of Oregon - *Understanding Churn in Peer-to-Peer Networks*, 2006.

tato il 21 febbraio 2013). Questo per evitare che alcuni nodi malevoli possano inquinare tutte le repliche nell'intervallo successivo di rotazione semplicemente richiedendo il salvataggio per primi.

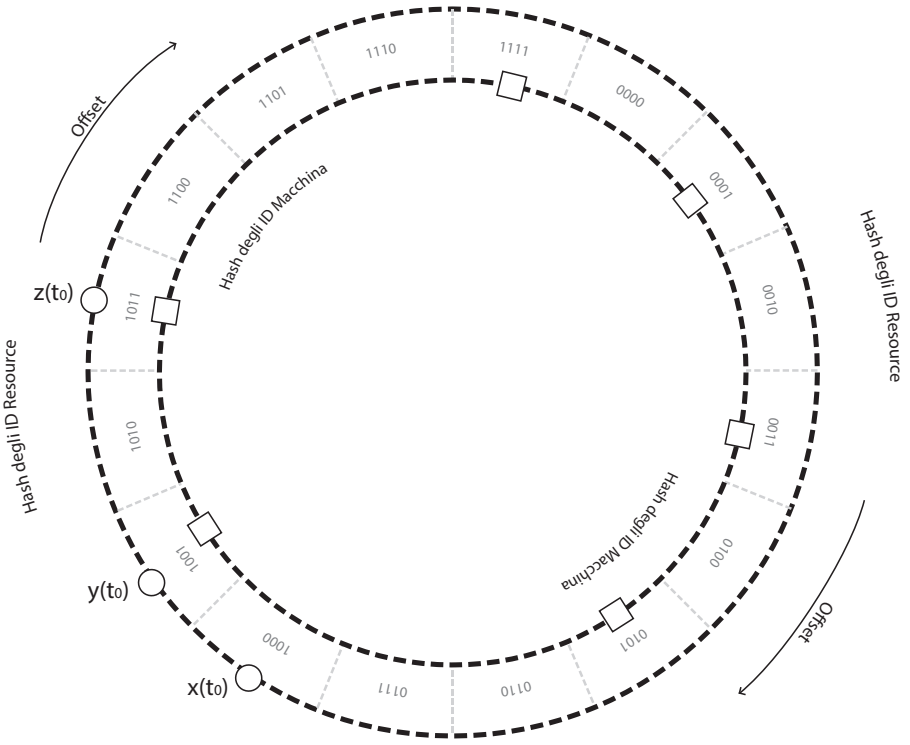
Infine per implementare la rotazione è necessario tener traccia non solo dell'ultimo *token* valido, e quindi dell'ultimo *offset* valido, ma anche del precedente. Questa è una conseguenza dello "sfasamento" della rotazione, necessario per ridurre il consumo di banda in corrispondenza della rotazione. Si è pensato infatti di non ruotare tutti gli *hash* di tutte le risorse contemporaneamente, ma di farlo in modo sfasato per non causare picchi di consumo di banda negli istanti di rotazione.

3.5 Progettazione delle API

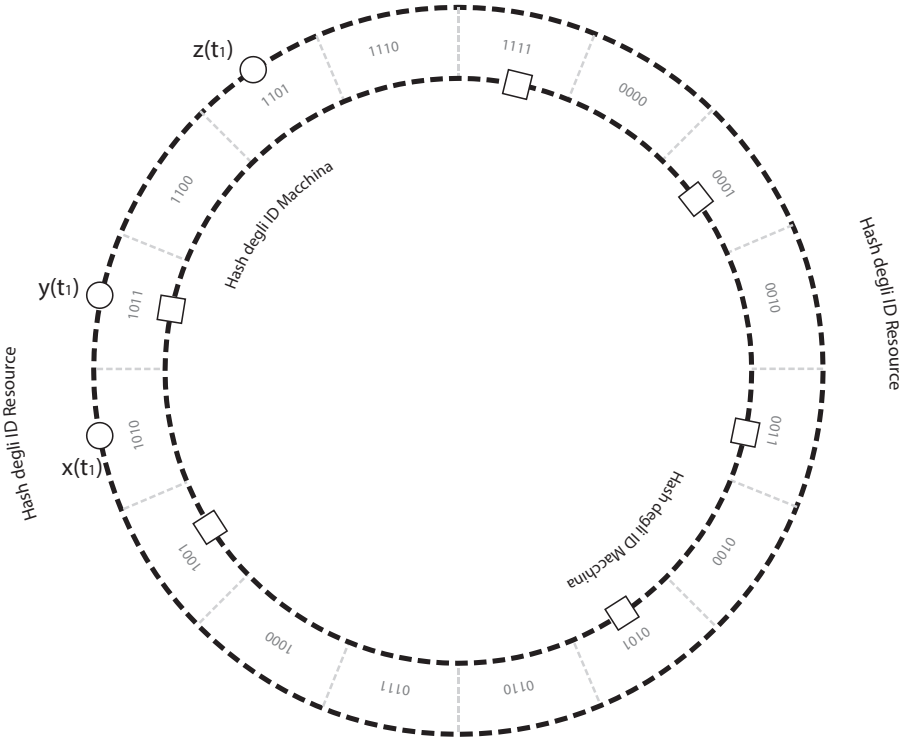
Le API di PariPari sono essenzialmente delle interfacce che a loro volta estendono l'interfaccia *"API<T extends API<T> >"*. Tale interfaccia contenuta nel *package "paripari.api"* è *l'astrazione di tutte le interfacce*. Ogni plug-in che voglia offrire un servizio (ad altri plug-in) deve prevedere almeno una sua API che estenda questa interfaccia. Si dovrà prevedere anche un metodo costruttore "builder"[5] che permetta di ottenere un istanza del plug-in richiesto e di utilizzare le sue primitive (API). Sempre allo stesso livello stanno le interfacce di *"Sender"* ed *"eventListener"* utilizzate rispettivamente per identificare plug-in che richiedono API e gestire scambi asincroni di risorse (fra plug-in).

La progettazione delle API è stato il primo passo per strutturare la DHT. Per la loro progettazione è stato necessario innanzitutto definire dei requisiti e delineare il funzionamento ad altissimo livello della DHT. Solo successivamente si è proceduto a scrivere la documentazione della API e appena è stata disponibile un'implementazione di *default* sono state testate e riesaminate.

[5]Per i dettagli sulla funzione "builder" vedi 5.1.1



(a) Prima della rotazione (istante t_0).



(b) Dopo la rotazione (istante t_1).

Figura 3.6: Esempio di rotazione.

Capitolo 4

Testing: basi teoriche

La programmazione estrema (XP, *Extreme programming*) è forse il metodo agile più diffuso e conosciuto. Il nome fu coniato da Beck poiché l'approccio fu sviluppato spingendo le normali procedure, come lo sviluppo ciclico e il coinvolgimento del cliente, a livelli "estremi"[1].

4.1 Filosofia di programmazione ed Extreme Programming

I punti chiave della programmazione estrema (XP) di Beck sono: pianificazione incrementale, progettazione semplice, test iniziali, *refactoring*. La pianificazione incrementale è decisa e rivalutata di volta in volta con riunioni del team di sviluppo. Come ogni altro aspetto in XP essa assume una forte caratteristica ciclica. In PariPari si è deciso di abbandonare l'idea di "progettazione delle modifiche" abbracciando quindi la progettazione semplice. Ad esempio lungo tutto lo sviluppo del modulo DHT si è deciso di non approfondire la gestione *"credits"*(vedi 5.3.1 a p.67), caratteristica che interesserà le future *release* di PariPari.

Un ruolo importante è rivestito dai test iniziali: questi permettono di produrre codice senza errori e senza perdere di vista l'obiettivo. La pratica di scrivere i

[1] Sommerville, I., *Ingengeria del software* , 8° ed. Pearson - Addison Wesley, 2007, p.388

test prima del codice stesso è alla base di TDD (vedi 2.2 a p.14). Un test è la base da cui partire per un qualsiasi intervento nel software e può anche suggerire una data scelta di programmazione. Nel caso di PariPari il test è anche usato come esempio di utilizzo delle API messe a disposizione. In tal modo PariPari, inteso come *framework* per lo sviluppo di applicativi, risulta intuitivo proprio grazie agli esempi forniti dai test.

È stato accennato che tutte le procedure che interessano il modello XP sono caratterizzate da una forte ciclicità. In particolare questo è molto evidente nel *refactoring* [2]. Un investimento nella cilicità del *refactoring* assicura migliore leggibilità del codice con conseguente riduzione della complessità di intervento, qualità fondamentali per un software dallo "sviluppo accademico" come PariPari.

4.2 Test

L'obiettivo del test è quello di realizzare un prodotto il più possibile esente da errori, ma come citato nella nota 12 a p.16, ad oggi non è possibile dare la garanzia di un software privo di errori. Sicuramente l'attività di *testing* aiuta ad eliminare gli errori presenti.

I test si possono dividere in tre grandi tipologie in base agli obiettivi e alle conoscenze richieste per completarli. Per ogni tipologia di test si possono utilizzare strumenti automatici o manuali che saranno presentati di seguito.

4.2.1 Black Box Test

Questa categoria di test è la prima che generalmente si va ad applicare. Per applicare tali test le conoscenze del codice possono essere molto ridotte. Il requisito

[2] In ingegneria del software, il refactoring (o code refactoring) è una tecnica strutturata per modificare la struttura interna di porzioni di codice senza modificarne il comportamento esterno, applicata per migliorare alcune caratteristiche non funzionali del software.
Tratto da `http://it.wikipedia.org/wiki/Refactoring`, (visitato il 21 febbraio 2013)

essenziale è che sia ben definito l'*input* e l'*output*. Si tratta quindi di considerare il modulo da testare come una "scatola nera" senza entrare nel dettaglio di come funzioni internamente. Lo scopo del Black Box Test è verificare che ad un determinato *input* corrisponda l'*output* previsto.

Come in ogni buon test l'*input* e l'*output* non vanno selezionati arbitrariamente, ma vanno studiati dominio e codominio della funzione "programma". In questo modo è possibile orientare la verifica verso valori critici che non vengono in genere presi in considerazione dal programmatore. Spesso si tratta di valori che non emergono naturalmente nella logica di programma, un esempio su tutti è il valore *"null"*.

Junit

Junit è un *framework* inventato da Kent Beck ed Erich Gamma per il test di unità relativo al linguaggio di programmazione Java. È il fondamentale strumento per lo sviluppo guidato da test (TDD)[3]. La versione utilizzata per il *testing* di DHT e a cui faremo riferimento è la 4 (Junit 4.8).

Con Junit si riesce ad automatizzare tramite facili primitive un insieme di test. In una classe di test si possono distinguere principalmente tre tipologie di metodi: inizializzazione, azzeramento, test. Le prime due comprendono i metodi: *setUpBeforeClass* e *tearDownAfterClass*. Si tratta di una coppia di metodi eseguiti una sola volta rispettivamente all'inizializzazione della classe (*setUpBeforeClass*) e una sola volta al termine di tutti i test (*tearDownAfterClass*). Il loro impiego può essere necessario in alcuni test parametrici o automatici (vedi p. 42), quando si vuole definire un certo valore iniziale per le variabili di classe. Ad ogni "inizio" dell'insieme di test compresi nella classe verrà eseguito il metodo *setUpBeforeClass* e ad ogni "fine" verrà eseguito il metodo *tearDownAfterClass*.

[3]Junit, `http://it.wikipedia.org/wiki/JUnit`, (visitato il 23 febbraio 2013)

I metodi utilizzati quindi risultano *setUp*, *tearDown* e *test*. Fanno sempre parte di "inizializzazione" e "azzeramento" i metodi *setUp* e *tearDown*. Generalmente con il *setUp* si inizializzano tutte le variabili di classe ad uno stato di *default* condivisibile per lo meno dalla maggioranza dei test che andranno implementati nella classe. Quello che si ritrova sempre è la variabile *cut*(*Class Under Test*) che rappresenta una istanza della classe da collaudare. Automaticamente prima di ogni test (appositamente identificato da *"@Test"* prima della dichiarazione del metodo) viene eseguito il *setUp* e successivamente, concluso il test viene eseguito il *tearDown*. Il metodo *tearDown* ha il compito di resettare ogni cambiamento effettuato all'interno delle variabili di classe, per tornare nel prossimo test ad una situazione completamente indipendente dalle precedenti chiamate. In ogni metodo di collaudo troveremo una clausola di asserzione, in cui si asserisce che un dato comportamento o stato di variabili è ciò che ci si aspetta per considerare superato il test. Una clausola di questo tipo può essere ad esempio:

```
AssertTrue( x < 3);
```

Che farà superare positivamente il test nel caso la variabile "x" sia minore al numero intero "3".

Nel corso del collaudo del modulo DHT si è scelto di utilizzare *Hamcrest* [4]. Ci si è orientati verso questo *framework* per osservare tutte le considerazioni e le linee guida sulla buona programmazione viste anche nel capitolo 2. Con *Harmcrest* infatti è possibile aumentare la leggibilità dei test. Ad esempio l'asserzione che in Junit restituisce "test superato" nel caso in cui la variabile "x" sia minore del numero intero "3" si può scrivere come:

```
assertTrue( x < 3);
```

con *Hamcrest* invece si può scrivere:

[4]Hamcrest è un ambiente di sviluppo per creare comparatori. (Tradotto da `http://en.wikipedia.org/wiki/Hamcrest`, (visitata il 10 marzo 2013) I comparatori sono particolari tipi di sintassi per mettere a confronto due oggetti. Essi sono alla base delle asserzioni di Junit.

```
assertThat(x,is(lessThan(3)));
```

risultando più vicina ad un linguaggio parlato e quindi più leggibile. Inoltre, cosa più importante, in caso di fallimento dell'asserzione si avranno maggiori dettagli sull'errore. Nel primo caso verrà segnalato semplicemente un errore (codice 4.1) da cui si può solo dedurre che "x" ha assunto un qualche valore maggiore di "3". Nel secondo caso, con *Hamcrest*, si avrà una spiegazione dettagliata dell'errore permettendo di capire direttamente quale valore di "x" abbia generato l'errore; nell'esempio visibile nel codice 4.2 il valore assunto da "x" era 2 147 483 647.

Codice 4.1: Falllimento del test (x<3). Log dell'errore

```
java.lang.AssertionError:
  at org.junit.Assert.fail(Assert.java:91)
  at org.junit.Assert.assertTrue(Assert.java:43)
  at org.junit.Assert.assertTrue(Assert.java:54)
  at paripari.myTest.test(AssertTrueExample.java:5)
```

Codice 4.2: Falllimento del test (x<3) con Hamcrest. Log dell'errore

```
java.lang.AssertionError:
Expected: is a value less than <3>
  got: <2147483647>

  at org.junit.Assert.assertThat(Assert.java:778)
  at org.junit.Assert.assertThat(Assert.java:736)
  at paripari.myTest.test(AssertThatExample.java:5)
```

Il framework Junit permette inoltre l'esecuzione di test automatici e parametrizzati semplicemente anteponendo il costrutto

```
@RunWith(value = Parameterized.class)
```

alla classe di test. Tramite un metodo statico appositamente costruito ed identificato da "`@Parameters`" è possibile restituire una lista di oggetti che saranno

poi passati uno ad uno come parametro ad un metodo ausiliario di "preparazione dei dati". Questo metodo, che a differenza degli altri non è identificato dal "`@Test`" iniziale, riceve in ingresso un oggetto di parametrizzazione con il quale ad esempio può andare ad aggiornare delle variabili di classe. Successivamente saranno eseguiti tutti i test della classe, per poi passare ad una nuova esecuzione del metodo di "preparazione dei dati" che riceverà in ingresso il successivo parametro della lista. Di conseguenza tutti i test della classe saranno eseguiti un numero di volte pari al numero di elementi nella lista di parametrizzazione.

EasyMock e PowerMock

Spesso il funzionamento di una classe è subordinato ai risultati provenienti da altre classi esterne; inoltre non è così raro che una classe per funzionare utilizzi istanze di altre classi. Questa dipendenza potrebbe compromettere la bontà del collaudo in quanto aggiunge variabili esterne non controllabili che potrebbero essere fonte di errori. Per mantenere il massimo livello di indipendenza dei test vengono simulate le istanze di queste classi esterne tramite degli oggetti *"mock"* che sono delle "finte copie" delle classi originali il cui comportamento può essere completamente programmato. *EasyMock* è una libreria Java che si occupa di implementare tali oggetti. Oltre ad alcune funzionalità aggiuntive rispetto ad *EasyMock*, *PowerMock* permette di creare oggetti *mock* anche di classi statiche. Senza la libreria *PowerMock* tutte le funzionalità di integrazione (vedi sezione 4.2.3) non sarebbero state collaudabili. Infatti ogni modulo di PariPari una volta in esecuzione ha necessariamente una parte statica come il *Core*, il modulo principale, il cui metodo di avvio *main* nella classe (paripari.core.Main) è statico (vedi codice 4.3).

Codice 4.3: Dichiarazione del metodo main in Main.java

```
/**
184 * Launches PariPari.
    * <p>
186 * This method throws an IllegalStateException for every
    * invocation following the first, as it is not meant to
188 * be called from within the code unless you're using
    * PariPari as a library.
190 *
    * @param args
192 *   command line arguments, as described in the help
    *   section run java -jar PariPari.jar --help to see
194 *   the full list, or refer to our wiki
    *
196 * @throws IllegalStateException
    *   in case this method has already been
198 *   called within this JVM instance
    */
200 public static synchronized void main(String[] args)
```

4.2.2 White Box Test

In contrapposizione al *Black Box*, il *White Box testing* non mira a verificare che siano rispettati i requisiti di *input* e *output* della funzione "programma". Con il *White Box Testing* infatti ci si orienta verso l'architettura interna del modulo e quindi sulla costruzione della "funzione programma". Lo scopo dei test *White Box* è verificare che tutte le righe di codice del modulo in esame siano effettivamente eseguite nei modi e nelle modalità previste dalla documentazione del programma. Un esempio classico sono i cicli (*while*, *for*) e le condizioni (*if-then-else*). La tecnica *White box* si occupa di far eseguire iterativamente ogni passo di ogni ciclo presente nel codice, e anche di soddisfare sequenzialmente tutte le possibili condizioni. L'obiettivo del *Withe Box Testing* è chiaramente quello di forzare l'esecuzione di tutto il codice eseguibile in tutte le sequenze di esecuzione possibili, in modo da far assumere al programma tutti gli stati possibili.

4.2.3 Integration Test

L'ultima tipologia di test è quella di integrazione. In realtà può venire considerata come un metodologia Black Box, ma dato che non interessa più un singolo modulo ma almeno due moduli merita una trattazione a parte. Possiamo considerare l'integrazione a diversi livelli di dettaglio in base al grado di astrazione nel considerare moduli e classi. Questa "granularità" dovrebbe essere ciclica, come tutti i processi di XP, e aumentare sempre più fino a collaudare (se possibile) il comportamento di ogni singola variabile. In genere ci sono due moduli da integrare, ma si può generalizzare anche nel caso in cui sia necessaria una integrazione a più moduli. Si parte da un interfaccia finta (*"mock"*) di un modulo e un interfaccia vera dell'altro, per poi invertire. Si verificano i funzionamenti Black Box della interazione delle rispettive interfacce. Superato questo primo livello, che in genere è quello critico, si estendono le due o più interfacce considerando le altre componenti dei moduli, e si itera il procedimento, allargando a macchia d'olio i componenti considerati.

Il termine ultimo del collaudo di integrazione è per costruzione il Beta Test dell'integrazione stessa.

L'integrazione del nuovo modulo DHT con gli esistenti Connectivity e Storage è un'attività partita recentemente nella cronologia di sviluppo di questa nuova DHT. Le interfacce di comunicazione che interessano i tre plug-in sono ben definite e vantano una precisa e minuziosa documentazione. Questo aiuterà il collaudo del funzionamento complessivo.

L'utilizzo di *Storage* da parte della DHT è limitato al processo di recupero e salvataggio di chiavi e valori da e verso il disco. Per minimizzare il numero di accessi al disco, in particolare per la ricerca, viene utilizzata l'implementazione

di un *BloomFilter* [5] che serve per conoscere in una ricerca se un dato elemento è possibile sia salvato nel disco oppure se è necessario attivare una procedura completa di ricerca attraverso la DHT.

Il modulo DHT utilizza *Connectivity* ogni qualvolta viene effettuata un'operazione esterna al modulo DHT stesso. Tenendo conto quindi della struttura di DHT tutte le operazioni che interessano *Search*, *Store*, *Read*, *Delete*, passano attraverso il modulo *Connectivity*.

[5] Si tratta di una struttura probabilistica inventata da Burton Howard Bloom (1970) che funge da filtro decisionale, basato su funzioni hash, per definire se un oggetto appartiene o meno a un insieme. La particolarità del BloomFilter è che le probabilità di falsi negativi, ossia di decidere erroneamente che un determinato oggetto non appartiene all'insieme, sono molto basse. Tratto da Bloom Filter, `http://en.wikipedia.org/wiki/Bloom_filter`, (visitato il 23 febbraio 2013)

Capitolo 5

Testing di DHT

I primi test eseguiti sulla DHT riguardano le implementazioni vuote di *default* delle interfacce. Questi sono test di unità che secondo la classificazione di XP possiamo considerare come *Acceptance test* piuttosto che *Unit test*, in quanto costruiti sulle API ossia su ciò che deve utilizzare direttamente un utente del *software* [1]. Ricordiamo infatti che PariPari ha come destinatari due tipi di utenza: un utente finale che semplicemente usufruisce dei servizi, e un utente programmatore, che anche grazie alla documentazione scritta potrà costruire i propri moduli e farli eseguire nella Rete.

5.1 Interfacce

I primi test presentati riguardano le interfacce utente. In DHT, come del resto in tutto il progetto PariPari, un plug-in riesce ad interagire con un altro tramite le API messe a disposizione. Un programmatore utilizzerà tali "interfacce"(API) all'interno del proprio codice per accedere a tutta una serie di funzionalità of-

[1] Per *Acceptance Test* si considerano quei test formali creati per determinare se il sistema soddisfa i requisiti di progettazione e per consentire all'utilizzatore finale di determinare con facilità se il sistema ha un funzionamento adatto o meno. Si differenzia dallo *Unit Test* proprio per il destinatario del test, ossia l'utilizzatore finale e non il programmatore. Per questo Kent Beck propose di riferirsi a questi test con il termine di *Customer test*. Tratto da Cunningham & Cunningham, Inc., `http://c2.com/cgi/wiki?AcceptanceTest` (visitato il 10 marzo 2013).

ferte dai diversi moduli; nel caso di DHT si può accedere a primitive di ricerca, cancellazione, salvataggio o condivisione.

Il compito del collaudo è anche quello di presentare ad un programmatore esterno un esempio di utilizzo del codice. In PariPari essendo il "programmatore esterno", un effettivo utilizzatore del *software* possiamo parlare di *Customer Test* con riferimento al test delle API.

Di seguito sarà presentato il lavoro di collaudo della classe *DHTStoreAPI*, ma il lavoro svolto riguarda nella stessa misura *DHTSearchAPI*, *DHTShareAPI* e *DHTDeleteAPI*.

L'interfaccia *DHTStoreAPI* comprende i seguenti metodi:

setAutoRefresh abilita o disabilita l'*autorefresh* per una risorsa nella DHT.

setExpiration imposta il momento dopo il quale la risorsa è da considerarsi non più valida.

setPrivileged imposta determinati permessi.

store salva la chiave e/o associa i relativi valori nella DHT.

storeAsync effettua una operazione di *"store"* in modo asincrono.

5.1.1 Builder

Ciò che accomuna tutte le interfacce di programmazione sono i costruttori (*Builder*). Infatti, prima di utilizzare un'interfaccia bisogna costruirla passando al metodo costruttore i giusti parametri. Questa soluzione permette di mantenere il controllo dei parametri necessari per l'istanziamento degli oggetti, direttamente in fase di compilazione: uno sviluppatore che utilizza un *Integrated Development Environment (IDE)* come Eclipse[2] può trarre notevoli vantaggi per evitare errori

[2]Progetto *Opensource* originariamente ideato da IBM alla fine del 2001 che attraverso la fondazione *not-for-profit* Eclipse (gennaio 2004) offre servizi di supporto allo sviluppo software, come

durante la scrittura del codice sorgente.

Come visibile nell'elenco dei metodi, l'interfaccia *DHTStoreAPI* non prevede un costruttore proprio, ma, come per tutte le API in PariPari, viene creata una sotto classe *Builder* che implementa *paripari.api.Builder<T>* (Codice 5.1).

Codice 5.1: DHTStoreAPI Builder

```
116 public class Builder implements
            paripari.api.Builder<DHTStoreAPI>
118   private final DHTKey key;
      private DHTValue value;
120   private FileInputStreamAPI fileValue;
```

Tale classe deve implementare un metodo che si occupi di creare un'istanza ben formata dell'oggetto *"Builder"*, in questo caso relativa a DHTStoreAPI, in altri relativa a DHTSearchAPI, DHTDeleteAPI o DHTShareAPI. Si dovranno pertanto fornire la chiave (*key*) identificativa su cui si vuole effettuare l'operazione *store* e il valore (*value*) da salvare. Quest'ultimo parametro può essere un oggetto *DHTValue* generico, ma anche una API di gestione file (*FileInputStreamAPI*) per permettere di estrarre il valore da salvare direttamente da un file. A questo scopo si prevedono due distinti costruttori descritti rispettivamente nel codice 5.2 e 5.3.

Come sarà approfondito nella sezione 5.2, la documentazione presentata è sufficiente ad implementare un buon collaudo. Ognuno dei due metodi di costruzione di *DHTStoreAPI* richiede una istanza di tipo *DHTKey*. Dato che si tratta di una classe *final* [3] non si può usare la libreria *EasyMock* convenzionale, bisogna usare la libreria *PowerMock*. In questo modo si riescono a simulare degli oggetti *"mock"* della classe *DHTKey* anche se non sarebbe permesso. La configurazione

IT Infrastructure, *IP Management*, *Development Process* e *Ecosystem Development*. Tratto da The Eclipse Foundation, *About the Eclipse Foundation*, `http://www.eclipse.org/org/` (visitato il 28 marzo 2013).

[3] In java una classe *final* è una classe che non permette sottoclassi. Non si può quindi estendere la classe *final* per generare una nuova classe con codice differente.

Codice 5.2: Costrutture di istanze DHTStoreAPI con DHTValue

```
122 /**
    * Creates a new builder.
124 *
    * @param key
126 *          the key for which a new value is to be stored,
    *          cannot be <code>null</code>
128 * @param value
    *      the value to be stored for the argument
130 *      <tt>key</tt>, cannot be <code>null</code>
    * @throws IllegalArgumentException
132 *          if <tt>key</tt> or <tt>value</tt> is
    *      <code>null</code>
134 */
    public Builder(DHTKey key, DHTValue value) {}
```

Codice 5.3: Costrutture di istanze DHTStoreAPI con valore da file

```
    /**
146 * Creates a new builder.
    *
148 * @param key
    *           the key for which a new value is to be stored,
150         cannot be <code>null</code>
    * @param value
152 *           the file handler from which the value to be
    *       stored for the argument <tt>key</tt> is to
154 *       be read, cannot be <code>null</code>
    * @throws IllegalArgumentException
156 *           if <tt>key</tt> or <tt>value</tt> is
    *       <code>null</code> and if <tt>value</tt>
158 *       is not a valid file descriptor
    *       (i.e. it can't be read)
160 */
    public Builder(DHTKey key, FileInputStreamAPI value) {}
```

iniziale avviene impostando come esecutore del test la classe *PowerMockRunner* (di default è *Junit*) e indicando quali classi necessitano di essere preparate inizialmente per il test. L'esempio è visibile nel codice 5.4. Successivamente è

Codice 5.4: DHTStoreBuilderTest - Dichiarazioni variabili di classe e configurazione

```
32 @RunWith(PowerMockRunner.class)
   @PrepareForTest({ DHTKey.class})
34 public class DHTStoreBuilderTest {
       /*
36      * final mock created with PowerMock
        */
38     private DHTKey testKeyMock;
       /*
40      * Class under Test
        */
42     private Builder cut;
       /*
44      * Next object are parameters necessary
        * for builder (cut)
46      */
       private Operation permission;
48     private FileInputStreamAPI fileInput;
       private DHTValue value;
50     private FileDescriptorAPI fd;
```

necessario procedere con la creazione effettiva delle istanze di test e degli oggetti *mock*, questo avviene chiamando il metodo *createMock* della classe *PowerMock*, e passando come parametro la classe che si vuole simulare. Questa inizializzazione è bene ripeterla ogni volta sia necessario azzerare eventuali problemi generati con l'esecuzione del test precedente. Come esposto nella sezione 4.2.1 saranno utilizzati i metodi *setUp* e *tearDown*.

Tenendo conto di quanto scritto nella *javadoc* del builder nel codice 5.3, si prevede che il costruttore debba ammettere due parametri in ingresso. Nel caso in cui il secondo parametro (*value*) sia un *FileDescriptor* non valido o nullo, viene lanciata un'eccezione del tipo *IllegalArgumentException*. Studiando il comporta-

Codice 5.5: DHTStoreBuilderTest - setUp

```
@Before
public void setUp() throws Exception {
  testKeyMock = PowerMock.createMock(DHTKey.class);
  value = PowerMock.createMock(DHTValue.class);
  fileInput = PowerMock.createMock(
            FileInputStreamAPI.class);
  fd = PowerMock.createMock(FileDescriptorAPI.class);
}
```

mento di un oggetto *FileDescriptor* si conclude che per controllare la validità del descrittore bisogna in sequenza:

recuperare il descrittore utilizzando il metodo `.getFD()`

verificarne la validità utilizzando il metodo `.valid()`

Verrà pertanto programmato l'oggetto *mock* di nome *fd* (in riferimento al codice 5.4) per verificare che siano effettuate tali chiamate. Per verificare che il comportamento del costruttore sia idoneo si dovrà configurare l'oggetto *mock FileDescriptor*, per restituire prima *false* e poi *true* quando interrogato sulla validità dell'oggetto. Nel primo caso il costruttore dovrà lanciare una *IllegalArgumentException*, nel secondo dovrà restituire un oggetto *DHTStoreAPI*. Quanto detto per il caso *false* è visibile nel codice 5.6, dove con i costrutti *expect*, *replay* si programma il comportamento del *FileDescriptor* simulato e con *verify* si verifica che effettivamente le chiamate vengano fatte. Con l'utilizzo di consuete asserzioni si controlla che il metodo *Builder* restituisca l'eccezione.

Questo è solo un esempio significativo dei test effettuati che, insieme agli altri ha condotto allo sviluppo del builder come visibile nel codice 5.7.

Gli *Acceptance Test* o *Customer Test* hanno potuto solo far capire se si stava percorrendo la strada giusta nella progettazione. Test più significativi per gli sviluppatori sono invece i test di unità. Il primo modulo di DHT sviluppato è stato quello di gestione della rete: *"Network Manager"*. Il *package* contenente le

Codice 5.6: DHTStoreBuilderTest - notValidBuilderTest_FileInputStream

```
@Test
public void notValidBuilderTest_FileInputStream()
                    throws IOException {
  EasyMock.expect(fileInput.getFD()).andReturn(fd);
  EasyMock.expect(fd.valid()).andReturn(false);
  EasyMock.replay(fileInput);
  EasyMock.replay(fd);
  try {
    cut = new Builder(testKeyMock, fileInput);
    fail("not exceptions even if fileInput is not valid");
  } catch (IllegalArgumentException e) {
  assertTrue("exceptions " + e + "correctly throw", true);
  };
  EasyMock.verify(fileInput);
  EasyMock.verify(fd);
}
```

Codice 5.7: DHTStoreAPI - Classe Builder, costruttore con FileInputStreamAPI

```
public Builder(DHTKey key, FileInputStreamAPI value) {
  if (key == null)
    throw new IllegalArgumentException(
      "null keys can't be stored in the DHT");
  if (value == null)
    throw new IllegalArgumentException(
      "null values can't be stored in the DHT");
  try {
    if (!value.getFD().valid())
      throw new IllegalArgumentException(
    "invalid file descriptor");
  } catch (IOException e) {
    throw new IllegalArgumentException
    ("invalid file descriptor", e);
    }
  this.key = key;
  this.fileValue = value;
}
```

classi che si occupano di queste funzionalità è "`paripari.dht.network`". Successivamente e quasi in contemporanea si è sviluppato il modulo di crittografia, situato nel *package* "`paripari.dht.security`".

5.2 Network Test

In ogni test di unità, per ogni classe collaudata, è stata data priorità al costruttore. Effettivamente in una classe di test, dopo i metodi di "`setup`" e "`teardown`", il primo metodo che si va a collaudare è il costruttore di classe.

Una delle classi principali del *package Network* è "`NetworkManager.class`". Il costruttore di questa classe non esiste esplicitamente, quindi (a meno di sviluppi futuri) si può considerare di utilizzare il costruttore di *default*. Si possono individuare altri due metodi di questa classe che potrebbero essere pensati come costruttori in quanto restituiscono un oggetto di tipo *NetworkManager*. Ognuno di essi infatti va ad impostare una variabile fondamentale per il funzionamento della classe stessa.

Il primo va ad impostare il *layout* della rete. In questo caso per *layout* si intende la struttura logica della rete ossia il modo di effettuare ricerche ed ottenere risultati. In particolare il parametro *layout* passato al costruttore è un'istanza che implementa l'interfaccia *NetworkLayout*. *NetworkLayout* è definita con il solo obiettivo di restituire tutti e soli i nodi che hanno delle conoscenze su una chiave cercata. Questi nodi saranno poi quelli che verranno contattati nel caso di operazioni di *store*, *read* o *search* sulla medesima chiave. Nel listato 5.8 è presentata la javadoc del metodo "`setLayout`".

Innanzitutto è importante verificare come non serva il codice scritto, ma sia sufficiente una documentazione ben scritta. In ingresso al metodo viene passato un parametro (*"layout"*). Ogni qualvolta ci sono dei parametri in ingresso un buon test deve verificare che il codice risulti solido a tutta una serie di input

Codice 5.8: NetworkManager Javadoc

```
/**
52 * Sets the layout of the network in use.
   *
54 * @param layout
   *            the DHT layout for the network in use
56 * @return a reference to this object,
   *       so calls can be chained
58 * @throws IllegalArgumentException
   *             if <tt>layout</tt> is <code>null</code>
60 */
public NetworkManager setLayout(NetworkLayout layout) {
62   return null;
}
```

non validi o comunque ai casi limite. Il valore più comune che viene provato è il valore *null*. Nel caso di *NetworkManager* la documentazione descrive con esattezza il comportamento che dovrà avere il metodo in caso di valore *null* in ingresso: dovrà essere lanciata un'eccezione del tipo *IllegalArgumentException*.

Un primo test sarà quindi verificare che il comportamento in caso di parametro *"null"* in ingresso sia effettivamente quello descritto nella *javadoc*. Una prima idea potrebbe essere forzare il codice all'errore, recuperando l'errore lanciato tramite un blocco *try-catch*. Se l'esecuzione del codice non lancia eccezioni il metodo di test dovrà giungere ad un fallimento sistematico. Viceversa se il blocco *catch* raccoglie la giusta eccezione il metodo di test dovrà fermarsi in uno stato di correttezza. L'implementazione di quanto detto è presentata nel codice 5.9, dove *"cut"* è la (*Class Under Test*) classe da testare, ossia l'istanza di *NetworkManager*.

Alcuni strumenti di Junit come "`org.junit.test.Expected`" permettono una leggibilità maggiore del codice. Lo stesso test presentato nel codice 5.9 lo si può scrivere senza utilizzare i costrutti *try-catch*, che ne diminuiscono la leggibilità, ma semplicemente anteponendo una clausola *expect* subito dopo la dichiarazio-

Codice 5.9: Metodo testSetNullLayout()

```
@Test
public void testSetNullLayout() {
  try{
    cut.setLayout(null);
    fail("expect an exception on setting null layout");
  }catch(IllegalArgumentException e){
  assertTrue(true);
}
```

ne del tipo di metodo, e specificando cosa effettivamente ci si aspetta. Il risultato è presentato nel codice 5.10, dove appunto viene esplicitato che dall'esecuzione di quel test ci si aspetta il lancio di un'eccezione.

Codice 5.10: Metodo testSetNullLayout() con uso di *expected*

```
@Test (expected = IllegalArgumentException.class)
public void testSetNullLayout() {
  cut.setLayout(null);
  fail("expect an exception on setting null layout");
}
```

Un'altra tipologia di test che si possono eseguire partendo dalla sola *javadoc* sono quelli che verificano il tipo di oggetto restituito dal metodo (se questo prevede un valore di ritorno). Nel caso di *NetworkManager* si andrà pertanto a verificare che venga restituito un riferimento all'oggetto stesso utilizzato per la chiamata al metodo. Possiamo per completezza mantenere due asserzioni consecutive: la prima che verifichi che l'oggetto restituito sia effettivamente una istanza della classe *NetworkManager.class*, la seconda che verifichi che si tratti proprio della *cut* con cui si effettua la chiamata al metodo. Il metodo di test risultante è visibile nel codice 5.11.

Inizialmente i test falliranno in quanto non esiste alcuna implementazione di

Codice 5.11: Metodo testSetProtocol()

```
@Test
public void testSetProtocol() {
  Protocol protocol = EasyMock.createMock(Protocol.class);
  Object ret = cut.setProtocol(protocol);
  assertThat(ret,instanceOf(NetworkManager.class));
  assertThat(cut,equalTo(ret));
}
```

NetworkManager. Questi saranno comunque d'aiuto per lo sviluppatore nell'implementazione della classe. Il codice 5.12 rappresenta una possibile implementazione di *NetworkManager* che tiene conto dell'esistenza di una variabile di classe *NetworkLayout layout*.

Codice 5.12: NetworkManager.java

```
/**
* Sets the layout of the network in use.
*
* @param layout
*            the DHT layout for the network in use
* @return a reference to this object,
*       so calls can be chained
* @throws IllegalArgumentException
*             if <tt>layout</tt> is <code>null</code>
*/
public NetworkManager setLayout(NetworkLayout layout) {
  if (layout == null)
    throw new IllegalArgumentException
            ("layout can't be null!");
  this.layout = layout;
  return this;
}
```

5.3 Security

Come nel *package Network* anche in *Security* è stato adottato lo stesso approccio iniziale con i test di unità. Alcune differenze che meritano di essere citate riguar-

dano la classe *RSACryptographicKey.class*, una implementazione dell'interfaccia *CryptographicKey.class*. Come visibile nel codice 5.13 questa classe si occupa di creare la coppia di chiavi pubblica e privata tipiche di un algoritmo di cifratura RSA [4].

Codice 5.13: CryptographicKey.class

```
/**
 * A key to be used to encrypt/decrypt
 * resources in the DHT.
 */
public interface CryptographicKey extends Serializable {

    /**
     * Return the "read part" of this CryptographicKey.
     *
     * @return a {@link ReadKey} object
     */
    public ReadKey getReadKey();

    /**
     * Return the "write part" of this CryptographicKey.
     *
     * @return a {@link WriteKey} object
     */
    public WriteKey getWriteKey();
}
```

Le chiavi generate sono di 2048 bit e sono rappresentate dagli oggetti del tipo *ReadKey* e *WriteKey*. Questi oggetti saranno trattati a pagina 61 e in particolare verrà analizzato l'oggetto *WriteKey*. Un collaudo appropriato per questa classe si basa sicuramente su un test di cifratura con il quale si verifica, in pura *Black Box*, il funzionamento complessivo della classe. Dato che non si tratta prettamente solo di un test *Black Box*, ma interessa diverse classi del *package Security*, nella documentazione questo test è descritto come un "Test di Integrazione Interna". La logica del test è semplice: un *array* di *byte* viene criptato tramite

[4] *In crittografia l'acronimo RSA indica un algoritmo di crittografia asimmetrica, utilizzabile per cifrare o firmare informazioni.* Tratto da RSA, `http://it.wikipedia.org/wiki/RSA`, (visitato il 16 marzo 2013)

la classe *WriteKey* e viene creato un nuovo oggetto da cui è possibile risalire all'*array* originale solo con la chiave di lettura contenuta nell'oggetto *ReadKey* associato alla classe *WriteKey* utilizzata inizialmente per cifrare. La prima cosa necessaria per il test è un array di *byte* significativo. Per fare questo si è scelto di utilizzare la classe "`java.util.Random`". Tale classe mette a disposizione un metodo "`nextBytes()`" che permette di ottenere un *array* di *byte* casuale della dimensione preferita. Si è utilizzato un *array* di prova di 128 *byte* ossia 1024 bit, come definito nelle specifiche. Utilizzando il metodo "`apply()`" della classe *WriteKey* si ottiene la codifica dell'*array* casuale che sarà possibile decodificare attraverso la classe *ReadKey*. Il collaudo di codifica e decodifica si conclude mettendo a confronto l'*array* originale con la copia codificata e poi decodificata. Nel codice 5.14 del test di cifratura, si assume che *"cut"* sia un'istanza della classe *RSACryptographicKey*, e che *rand* sia l'*array* casuale di 128 *byte*.

Codice 5.14: Test di Cifratura

```
92  @Test
    public void testCorrectlyCiphred(){
94    //encode a string
      byte[] encrypted = cut.getWriteKey().apply(rand);
96    //verify if it is not plaintext
      assertThat(encrypted, not(equalTo(rand)));
98    //try to decode it with the read key
      assertThat(cut.getReadKey().apply(encrypted),
100             equalTo(rand));
    }
```

Per assicurare ripetibilità e specialmente per non limitarsi ad eseguire il test su un solo *array* si è scelto di automatizzare il test e parametrizzarlo come descritto a pagina 42. Viene creato un metodo di parametrizzazione,"`data()`", eseguito prima dei test e della loro inizializzazione (*setUp*). Per programmare tale esecuzione in modo automatico è sufficiente anteporre "`@Parameterized.Parameters`" al metodo "`data()`". L'esecuzione avviene quindi secondo il seguente ordine:

1. **parametrizzazione:** esecuzione del metodo "`data()`"
2. **inizializzazione:** esecuzione del metodo "`setUp()`"
3. **test:** esecuzione del metodo di test contrassegnato con "`@Test`"
4. **azzeramento:** pone a *null* le variabili di test con esecuzione del metodo "`tearDown()`"

Costruendo il metodo "`data()`" in modo da restituire una lista, condizioneremo *Junit* va ad eseguire per ogni elemento della lista l'intero pacchetto di test presente nella classe. Quindi saranno eseguiti *inizializzazione, test, azzeramento* di ogni metodo di test un numero di volte pari agli elementi della lista. Strutturando la parametrizzazione come nel codice 5.15, otterremo una iterazione automatica dei test per dieci volte.

Codice 5.15: Parametrizzazione

```
//Easy way to repeat 10 times the class test
@Parameterized.Parameters
public static List<Object[]> data() {
  return Arrays.asList(new Object[10][0]);
}
```

L'*inizializzazione* è stata concepita per generare ogni volta un *array* casuale come visibile nel codice 5.16. In questo modo si ottiene la ripetizione del test ogni volta con un *input* diverso.

Codice 5.16: Inizializzazione

```
@Before
public void setUp() throws Exception {
  Random r = new Random();
  r.nextBytes(rand);
  System.out.println(rand);
  cut = new RSACryptographicKey();
}
```

5.3.1 Serializzazione

Le interfacce *WriteKey*, *ReadKey* e *CryptographicKey*, sono interfacce serializzabili perché a loro volta estendono la classe *Serializable*. In Java un oggetto o una classe viene progettata come serializzabile se deve essere trasmessa attraverso un flusso di dati. È compito del metodo di serializzazione clonare opportunamente la classe per renderla un oggetto inviabile attraverso un canale di trasmissione dati. Generalmente la serializzazione prevede una coppia di metodi: il metodo di scrittura e il metodo di lettura. Il primo permette di trasformare la classe stessa in un flusso di byte, il secondo permette di elaborare un flusso di byte ed eventualmente estrapolare da questo un istanza della classe di partenza. Di seguito sarà presentato il lavoro di serializzazione e collaudo delle classi relative alla cifratura RSA.

I metodi di scrittura convenzionalmente riconosciuti per serializzazione e deserializzazione sono rispettivamente *writeObject* e *readObject*. Per effettuare un collaudo significativo c'è stata inoltre la necessità di riscrivere il metodo *equals* in modo da permettere di verificare correttamente l'uguaglianza di due oggetti della stessa classe. Si è scelto di presentare il lavoro fatto con la classe *RSAWriteKey*, ma questo, con opportuni accorgimenti, è estendibile anche alle classi *RSACryptographicKey* e *RSAReadKey*.

RSAWriteKey

Nel codice 5.17 presentato è possibile distinguere la variabile di classe privata *internalKey* del tipo *PrivateKey*. Questa permette di ottenere modulo ed esponente della chiave RSA relativa. Per la serializzazione della classe *RSAWriteKey* è sufficiente trasmettere i due valori di modulo ed esponente: da questi infatti, conoscendo l'algoritmo RSA è possibile ottenere la chiave di partenza e quindi un'istanza uguale della classe *RSAWriteKey*.

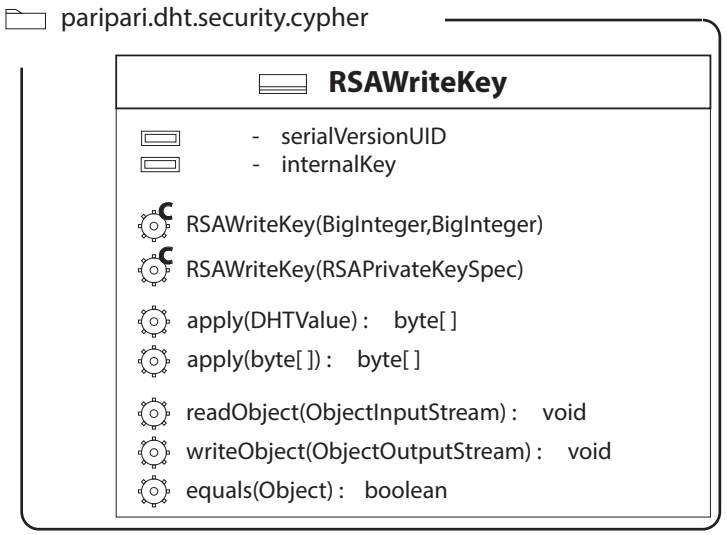


Figura 5.1: Schema UML della classe RSAWriteKey.

Codice 5.17: Metodo writeObject di serializzazione di RSAWriteKey

```
164 private void writeObject(ObjectOutputStream out)
                    throw IOException{
166   BigInteger modulus = ((RSAPrivateKey) internalKey)
              .getModulus();
168   BigInteger exponent = ((RSAPrivateKey) internalKey)
              .getPrivateExponent();
170   out.writeObject(modulus);
      out.writeObject(exponent);
172 }
```

Allo stesso modo la *deserializzazione* (codice 5.18) avviene con una sorta di *parser* [5] del flusso di dati in input, che tenta di individuare modulo ed esponente nell'ordine con cui una precedente serializzazione avrebbe dovuto porli.

Codice 5.18: Metodo readObject di deserializzazione di RSAWriteKey

```
private void readObject(java.io.ObjectInputStream in)
174       throws IOException, ClassNotFoundException {
  BigInteger modulus = (BigInteger) in.readObject();
176   BigInteger exponent = (BigInteger) in.readObject();
  RSAPrivateKeySpec key =
178           new RSAPrivateKeySpec(modulus, exponent);
  try {
180     initializeInternalPrivateKey(key);
  } catch (IllegalArgumentException e) {
182   e.printStackTrace();
  } catch (CryptographicKeyBuildException e) {
184   e.printStackTrace();
  }
186 }
```

Il collaudo della serializzazione deve iniziare verificando che la classe stessa sia serializzabile, l'idea è quindi di tentare di produrre uno *stream* della classe, creando un nuovo flusso di uscita di byte e andando a scrivere in esso l'istanza di *RSAWriteKey* tramite il metodo di serializzazione. Senza conoscere i dettagli di serializzazione, si potrà verificare la serializzabilità dell'oggetto controllando se il flusso generato è non nullo e ha lunghezza maggiore di zero. L'esempio descritto è visibile nel codice 5.19, dove vengono utilizzate due istanze della classe *RSAWriteKey*: *cut1* e *cut2* che rappresentano la stessa istanza di classe generata con costruttori diversi. L'istanza *cut1* è generata con il costruttore che riceve un solo parametro di ingresso di tipo *RSAPrivateKeySpec*, *cut2* invece è generata con due parametri di ingresso che rappresentano modulo ed esponente. In realtà le

[5] In informatica, il parsing o analisi sintattica è il processo atto ad analizzare uno stream continuo in input (letto per esempio da un file o una tastiera) in modo da determinare la sua struttura grammaticale grazie ad una data grammatica formale. Un parser è un programma che esegue questo compito. Tratto da Parsing, `http://it.wikipedia.org/wiki/Parsing`, (visitato il 12 marzo 2013)

due istanze sono identiche in quanto il parametro *RSAPrivateKeySpec* viene appositamente costruito partendo dagli stessi modulo ed esponente utilizzati per la costruzione di *cut2*. Si effettua il collaudo con due istanze tendenzialmente uguali per far emergere eventuali anomalie nei costruttori che non fossero emerse nei specifici *builder-test* (5.1.1).

Codice 5.19: Verifica se la serializzazione è implementata

```
@Test
public void isSerializableTest() throws IOException,
        ClassNotFoundException,
        CryptographicKeyExecException,
        CryptographicKeyBuildException{
  // consider cut2 ReadKey
  // serialize
  ByteArrayOutputStream out = new ByteArrayOutputStream();
  ObjectOutputStream oos = new ObjectOutputStream(out);
  oos.writeObject(cut2);
  oos.close();
  //verify
  assertTrue(out.toByteArray().length > 0);

  // consider cut1 ReadKey
  // serialize
  out = new ByteArrayOutputStream();
  oos = new ObjectOutputStream(out);
  oos.writeObject(cut1);
  oos.close();
  //verify
  assertTrue(out.toByteArray().length > 0);
}
```

Una volta collaudata la capacità di serializzazione della classe resta da verificare che questa venga fatta correttamente. Per entrambi i modi possibili di costruire una istanza di *RSAWriteKey*, viene proposto un test che prevede: una serializzazione, una deserializzazione e un confronto dell'istanza ottenuta con quella iniziale. Proprio questo confronto ha fatto emergere il bisogno di riscrivere il metodo *equals*. L'esempio posto nel codice 5.20 è relativo alla classe *cut1*, generata quindi con il costruttore con un solo parametro in ingresso, ma que-

sto non è vincolante: lo stesso test è stato fatto con *cut2*. In tale esempio viene serializzato l'oggetto *cut1* e, il flusso di *byte* ottenuto viene deserializzato e salvato come un oggetto di tipo *WriteKey* (*copy*). Infine tramite il metodo *equals* i due oggetti vengono confrontati. Il Test viene superato se i due oggetti risultano uguali.

Codice 5.20: Verifica la correttezza della serializzazione

```
348 @Test
    public void serializationCut1Test() throws IOException,
350         ClassNotFoundException,
            CryptographicKeyExecException,
352         CryptographicKeyBuildException{
    //      consider cut1 ReadKey
354 // serialize
    ByteArrayOutputStream out = new ByteArrayOutputStream();
356 ObjectOutputStream oos = new ObjectOutputStream(out);
    oos.writeObject(cut1);
358 oos.close();

360 //deserialize
    byte[] pickled = out.toByteArray();
362 InputStream in = new ByteArrayInputStream(pickled);
    ObjectInputStream ois = new ObjectInputStream(in);
364 Object o = ois.readObject();
    WriteKey copy = (WriteKey) o;
366
    //verify with equals method
368 assertThat(cut1,equalTo(copy));
    }
```

Capitolo 6

Evoluzioni Future

"Optimism is an occupational hazard of programming. Feedback is the treatment."

Kent Beck

L'obiettivo di questa tesi era descrivere il lavoro di affiancamento allo sviluppo del modulo DHT e le procedure per la qualità del software comuni a tutto il progetto PariPari. Si è descritta quindi la programmazione agile, lo sviluppo guidato dai test (TDD) e sono stati presentati alcuni esempi pratici di collaudo.

Seguendo lo spirito di programmazione agile, solo dopo aver realizzato un modulo DHT funzionante si implementeranno con il modello di sviluppo ciclico funzionalità aggiuntive e saranno definiti dettagli sulla sicurezza. Ad esempio un problema aperto è quello del controllo degli accessi alla Rete per evitare furti di identità; un altro riguarda la gestione di *black list* di nodi che tentano di accedere senza permesso alle risorse.

Attualmente DHT risulta ancora in fase di sviluppo, ma le fondamenta sono solide e soprattutto la pianificazione dello sviluppo è chiara.

6.1 Evoluzioni future del progetto PariPari

PariPari mira ad una release ufficiale prima della fine del 2013 e tenendo conto che Core, Storage, Connectivity sono già funzionanti l'attesa riguarda solo DHT che è vicino ad un primo rilascio.
Le potenzialità di PariPari sono ancora grandi rispetto alla concorrenza, nonostante la diffusione, di sistemi *cloud All-in-One*, che offrono tutta una serie di servizi desiderabili. La piattaforma *Google* su tutti permette attraverso *Drive* la scrittura e catalogazione di documenti, attraverso *Gmail* la ricezione e archiviazione della posta elettronica, attraverso *GoogleMusic* l'upload e l'ascolto in *streaming* di contenuti multimediali, ma poi ancora *google plus*, *google maps* e *google Calendar*. Tali servizi sono sicuramente distribuiti, ma la loro architettura interna non è di rete *peer-to-peer* (P2P) pura. Il vantaggio di *PariPari* è proprio questo specialmente per la recente diffusione di connessioni ad alta velocità come *4G*[1] e fibra ottica; il fitto scambio di messaggi, necessario in una rete P2P, risulta incidere molto meno nella banda della rete permettendo al contempo velocità di risposta della rete più elevate (recupero e salvataggio di pacchetti).

Una volta uscita una release funzionante, il compito dei "pariparisti"[2] sarà quello di continuare lo studio del progetto per aumentarne affidabilità, robustezza e prestazioni.

[1] *Nell'ambito della telefonia mobile con il termine 4G (acronimo di 4th (fourth) Generation) s'indicano relativamente a tale campo, le tecnologie e gli standard di quarta generazione successivi a quelli di terza generazione, che permettono quindi applicazioni multimediali avanzate e collegamenti dati con elevata banda passante. [...] I tecnici hanno dichiarato che i nuovi terminali sono in grado di ricevere fino ad un massimo di 100 Megabit/s in movimento e 1 Gigabit/s in posizione statica: in pratica il contenuto di un normale DVD video potrebbe essere scaricato in quasi un minuto da un terminale connesso a una rete 4G.* Tratto da 4G Telefonia, `http://it.wikipedia.org/wiki/4G_(telefonia)`, (visitato il 17 marzo 2013)

[2] Con tale nome vengono identificati gli sviluppatori di PariPari

6.2 Il futuro del testing

Seguendo un modello di sviluppo ciclico, si darà il via ad un nuovo *refactoring* dell'intero codice per risolvere tutte le problematiche specialmente di sicurezza non ancora completamente risolte, a tal proposito citiamo la prevenzione di attacchi Sibilla (*Sybil Attack*) [3]. La vulnerabilità (del sistema di permessi) a un attacco Sibilla è tanto maggiore quanto più facilmente si possono generare le utenze. In alcune ricerche, finanziate da *Microsoft*, sembra che l'unico modo per eliminare questa vulnerabilità sia l'uso di una Entità di certificazione centrale, ma questa scelta andrebbe in contraddizione con la scalabilità del sistema [4]. Il problema è ancora in fase di studio.

Il ruolo del testing in questo *refactoring* sarà più che mai fondamentale. Sarà necessario costruire un insieme di test per simulare le più comuni operazioni, i possibili attacchi, e un normale utilizzo. Sarà necessario entrare nel *beta testing* automatizzando tutta una serie di operazioni per verificare e monitorare i limiti della Rete. Per permettere di quantificare la bontà degli interventi eseguiti si dovranno costantemente controllare tutta una serie di parametri e valori significativi, uno su tutti è la scalabilità (che mira ad essere il "cavallo di battaglia" di PariPari). A questo proposito dovranno essere creati dei moduli di PariPari chiamati *dummy*[5] che rappresenteranno diverse istanze di PariPari. Queste verranno inserite in una rete di simulazione in cui verranno costantemente controllati parametri come tempo di risposta, banda passante e distribuzione dei nodi.

Il collaudo è considerato dai programmatori come la pena da scontare per

[3] Si tratta di attacchi che destabilizzano il sistema (*reputation system*) creando numerose entità/nodo con lo scopo di guadagnare credibilità e influenza nella rete P2P.

[4] Douceur, J.R., *The Sybil Attack*, Microsoft Research, Atti del 1 ° Workshop internazionale su reti peer-to-peer (IPTPS), 2002, pp.1,3

[5] Letteralmente "copia". Nel contesto ci si riferisce a delle copie del modulo originale. Generalmente questi vengono utilizzati nel testing di scalabilità di un software. Sono l'equivalente dei *mock object*.

non aver raggiunto la perfezione[6], in realtà si tratta dell'esaltazione della perfezione che viene dimostrata proprio attraverso il superamento del collaudo. Tuttavia, come dice Pressman [7], il collaudo non finisce mai, la sua responsabilità passa dal tecnico del software al cliente. Ogni volta che il cliente utilizza il programma, egli di fatto compie un collaudo.

Nel momento in cui si finisce il collaudo di un software si smette di dimostrarne la correttezza, ecco perché in PariPari il testing non dovrà mai fermarsi.

[6]Beizer, B. , *Software Testing Techniques*, 2° ed., Van Nostrand-Reinhold, 1990. in Pressman, Roger S., *Principi di Ingegneria del software*, 4° edizione McGraw-Hill, 2004, p. 424

[7]*Ivi* p. 462

Appendice A

Andamento di PariPari

Questo documento vuole essere una collezione di dati e previsioni con il fine di rendere l'idea dell'andamento di PariPari. Il periodo preso in considerazione, dove non diversamente indicato, va da Gennaio 2008 a Marzo 2013. Il modo di recupero delle informazioni e dei dati statistici utilizzato non sempre si può considerare affidabile. In particolare, come segnalato di seguito, questo è evidente nella raccolta di informazioni riguardanti la quantità di documentazione scritta.

I dati sprovengono dall'incrocio delle informazioni recuperate attraverso diverse fonti:

- SVNStat sul *repository* di PariPari;
- Mediawiki di progetto;
- Bugzilla: `http://www.paripari.it/bugzilla/`;
- Redmine: `http://verona.dei.unipd.it/redmine/projects/paripari/wiki`;
- Sistema di prenotazione aule del dipartimento: `http://prenota.dei.unipd.it`;
- Gruppo google di PariPari;
- Utenza nella macchina *Indy* che ospita PariPari tramite *Secure Shell* (SSH).

Proprio per le finalità di questo documento, non si riporteranno tutti i dati nel dettaglio[1], ma si cercherà di mantenere quanto più possibile una visione d'insieme.

A.1 Bug

Si è iniziato tenere traccia dei Bug solo a partire dalla seconda metà del 2008 quando si è attivata la piattaforma *Bugzilla*; successivamente molti dei servizi sono stati unificati in *Redmine*. Nell'andamento complessivo si notano picchi di "scoperta" di bug nei mesi di aprile e maggio ma, per non appesantire la trattazione, si è scelto di non indicare l'andamento mensile. Nel complesso si nota una progressiva diminuzione della scoperta di bug.

Anno	**2008**	**2009**	**2010**	**2011**	**2012**	**2013**
Numero di Bug	113	87	38	20	11	3

Tabella A.1: Andamento dei Bug

A.2 Linee di Codice

Le linee di codice sono da considerarsi in senso assoluto: non viene preso in considerazione il refactoring in queste righe di codice. Pertanto viene considerata la scrittura di nuove righe di codice e non l'eventuale cancellazione. L'approssimazione scelta è di $\pm$ 25 000 righe.

La terza riga dei dati indica le differenze rispetto all'anno precedente. Nel 2010 c'è stata la più importante differenza, poi fino al 2012 si evidenzia una diminuzione del codice scritto per anno. Nel 2013 sembra che si possa avere un inversione di tendenza.

[1]Infatti per alcune grandezze prese in considerazione è stato possibile scendere nel dettaglio dei mesi, settimane, giorni, ore.

Anno	2008	2009	2010	2011	2012	2013
Linee di Codice	825	990	1300	1525	1550	1575
Differenza dall'anno precedente	-	165	310	225	25	25

Tabella A.2: Righe di codice espresse in migliaia ($\times$1000)

A.3 Testing e Copertura del codice

È stata verificata la quantità di righe di testing rispetto a quelle di codice con la stessa metodologia della sezione A.2. Questo è stato possibile per l'organizzazione interna del *repository* che vede collocati l'insieme dei test in *package* e cartelle a se stanti.

Inoltre è stata verificata anche la copertura del testing con lo strumento di *coverage* (*Eclemma*) disponibile con Eclipse. L'approssimazione è a $\pm$1000 righe. Per copertura si intendono le righe di codice (e non di test) che vengono eseguite durante la fase di testing. Nella copertura il dato traparentesi è la copertura annuale media, questo dato è presente solo fino al 2010. Il dato copertura si riferisce al valore nel mese di dicembre dell'anno in questione approssimato all'intero più vicino. Negli anni 2011 e 2012 non sono presenti dati sulla copertura, il dato più aggiornato (quello presente nella colonna 2013), è stato calcolato nel mese di Aprile.

Anno	2008	2009	2010	2011	2012	2013
Numero di righe di test ($\times$1000)	825	990	1300	1525	1550	1575
Copertura dei Test (%)	25(22)	25(26)	27(29)	/	/	35

Tabella A.3: Copertura del testing

A.4 Quantità di dati generata - Documentazione

Questa grandezza è stata misurata tenendo in considerazione i documenti nella WIKI di progetto e in redmine. Questa è una delle misure più difficilmen-

te interpretabili e di cui si hanno meno dati realistici. Il calcolo è stato fatto direttamente sul repository con il comando ‘‘du’’ - *disk usage* di linux che nel caso esprime il risultato in blocchi di “kibibyte” (KiB)[2] , nella posizione /var/svn/repos/paripari/db/revs/del server *Indy* di PariPari. Si tratta del dato più impuro in quanto tiene conto anche delle righe di codice.

Nella tabella troviamo sia il valore totale della somma, sia il valore parziale rispetto all’anno precedente.

Anno	2008	2009	2010	2011	2012	2013
dati in KiB parziali	23544	37588	121556	303692	122620	2812
Totale dati in KiB	23544	61132	182688	486380	609000	611812

Tabella A.4: Quantità di dati generata

A.5 Riunioni e controllo

Queste grandezze sono state misurate tenendo in considerazione le riunioni (espresse in ore) e i singoli interventi nel forum o nel gruppo *GoogleGroup*. C’è da segnalare che mediamente un intervento su dieci prima del 2011 è un intervento senza alcun tipo di informazione aggiuntiva: si tratta infatti di messaggi di conferma per una riunione piuttosto che per un ordine[3]. Si è giunti a questo risultato tramite l’analisi in dettaglio di un campione rappresentativo di quattro mesi per anno: Gennaio, Febbraio, Marzo, Aprile.

[2]Il kibibyte è un’unità di misura dell’informazione o della quantità di dati, il termine deriva dalla contrazione di *kilo binary byte* ed ha per simbolo *KiB*. 1 kibibyte = 1 024 B = 2^{10} byte. Tratto da http://it.wikipedia.org/wiki/Kibibyte (visitato il 4 aprile 2013).

[3]Tra questi messaggi troviamo quelli che gli sviluppatori hanno chiamato in gergo *ack* (*acknowledgement*) per fare riferimento al tipico scambio di messaggi del protocollo TCP), ma anche messaggi di comunicazioni *off topic (OT)* non inerenti allo sviluppo del software, come ad esempio gli auguri per le festività.

Anno	2008	2009	2010	2011	2012	2013
Tempo di meeting (ore)	72	67	52	36	47	6
Numero interventi	3242	1275	886	277	540	-

Tabella A.5: Riunioni e controllo

A.6 Commenti e valutazioni

Il documento ha potuto fornire un andamento di Paripari dal 2008 ai primi mesi del 2013. Prima di qualsiasi valutazione è necessario ribadire che i dati raccolti danno solo un'indicazione, anche perché relativi alle sole grandezze rilevate e documentate. Da indagini eseguite all'interno di PariPari risulta infatti che non sempre gli strumenti convenzionali sono stati usati propriamente. Ad esempio non sempre i bug hanno attraversato un processo di segnalazione, assegnazione e risoluzione documentato attraverso *Redmine* o *Bugzilla*. Un'ulteriore esempio sono le riunioni, non sempre segnalate attraverso i canali convenzionali, oppure non sempre eliminate dalla programmazione quando annullate.

Dai dati raccolti si riscontra un calo nella scoperta di bug, ma un aumento della copertura del testing. Il tempo dedicato alle riunioni ha subito un calo nel 2011, ma grazie a Redmine è in aumento il numero e la qualità degli interventi (nel forum). Questi ultimi si possono considerare come uno scambio di messaggi di controllo. Tenendo conto di quanto detto in sezione A.5 il "controllo assoluto" sul progetto inteso come il valore stimato[4] del numero di interventi è in aumento. Infine, dato di grande rilievo, la copertura del testing è in crescita nonostante lo sia anche il codice prodotto. Questo significa che viene testato più codice di quanto non ne venga scritto dai programmatori. Un dato sicuramente positivo.

[4]In base a quanto detto i valori precedenti al 2011 vengono moltiplicati per 0,1.

Risulta comunque negativa la diminuzione della scoperta di bug, infatti solo raramente questo è sinonimo di "perfezione" del codice. Per invertire la tendenza un'idea potrebbe essere quella di anticipare una fase di *Beta Test*, ma questo andrebbe contro agli obiettivi della programmazione agile: avere qualcosa di funzionante nel più breve tempo possibile. La strada da percorrere risulta quindi quella già programmata: completare il prima possibile il plug-in DHT, al fine di procedere con le successive fasi di *test* e *refactoring*.

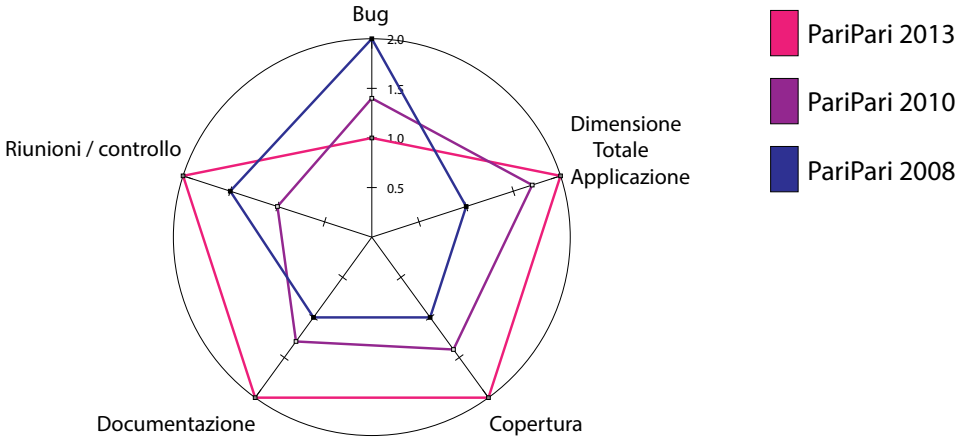


Figura A.1: Tendenza di PariPari dal 2008 al 2013
I valori sono calcolati sulla base delle tabelle precedenti normalizzando i valori su una scala da 1 a 2. In pratica si è utilizzata una funzione lineare biettiva che associa al minimo della riga il valore 1, e al massimo il valore 2.

Bibliografia

[1] Beck, k., *Extreme Programmming Explained: embrace change*, 12th printing, Addison Wesley, 2004.

[2] Beizer, B. , *Software Testing Techniques*, 2° ed., Van Nostrand-Reinhold, 1990.

[3] Bertasi, P. ,Relatore Peserico, E., *Progettazione e realizzazione in Java di una rete peer to peer anonima e multifunzionale*, Tesi di Laurea, Padova 2006.

[4] Daniel Stutzbach, Reza Rejaie, University of Oregon - *Understanding Churn in Peer-to-Peer Networks*, 2006.

[5] Documentazione di progetto DHT, `http://verona.dei.unipd.it/redmine/projects/newdht/wiki`, 2012.

[6] Douceur, J.R., *The Sybil Attack*, Microsoft Research, Atti del 1 ° Workshop internazionale su reti peer-to-peer (IPTPS), 2002.

[7] Kernighan, B.W. e Ritchie D.M., *The C Programming Language*, 1° ed., Prentice Hall Software Series, 1978.

[8] Musa J. D., A. Ianninoe K. Okumoto,*Engeneering and Managing Software with Reliability Measure*, McGraw-Hill, 1987.

[9] Phillips A., *Software Develop - Defensive Programming* `http://devmethodologies.blogspot.it/2012/05/defensive-programming.html` (visitato il 18 febbraio 2013).

[10] Pressman, Roger S., *Principi di Ingegneria del software*, 4° edizione McGraw-Hill, 2004.

[11] Sommerville, I., *Ingengeria del software* , 8° ed. Pearson - Addison Wesley, 2007.

[12] Simonetto, A.,Relatore Peserico, E., *Progettazione e realizzazione in Java di una rete P2P anonima e multifunzionale: connettività sicura e affidabile*, Tesi di Laurea, Padova, 2005.

[13] The Eclipse Foundation, *About the Eclipse Foundation*, `http://www.eclipse.org/org/`.

[14] `http://rubyonrails.org/`

[15] `http://wikipedia.org/`