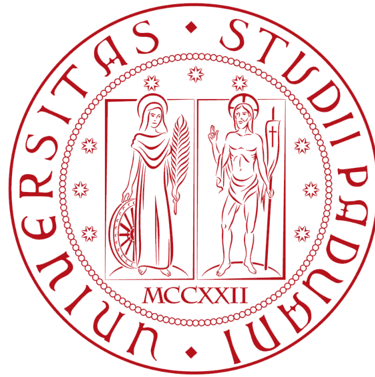

Università degli Studi di Padova
Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in
Ingegneria Informatica

Tesi di laurea magistrale

Allocazione di task basata su DHT nel Calcolo Distribuito Volontario

Laureando: Sebastian Daberdaku
Relatore: Ch.mo Prof. Carlo Ferrari

15 Ottobre 2012

A mio padre, a mia madre e a mia sorella,
ancora una volta...

Indice

Sommario v

Introduzione vii

1. Il Calcolo Distribuito Volontario 1

1.1. Gli inizi . . . 1
1.2. L'importanza . . . 1
1.3. Uno sguardo sul funzionamento . . . 2
1.4. I framework . . . 4
1.5. BOINC . . . 4
1.5.1. Architettura di un sistema BOINC . . . 5
1.5.1.1. Il software client . . . 6
1.5.1.2. Il server . . . 8
1.5.1.3. Ulteriori funzionalità dei server BOINC . . . 9
1.6. Grid vs. Volunteer Computing . . . 10
1.7. Punti deboli e problemi . . . 10

2. Lookup, Routing, e Storage Distribuito 13

2.1. Pastry . . . 13
2.1.1. Introduzione . . . 13
2.1.2. Design di Pastry . . . 14
2.1.3. Stato del nodo Pastry . . . 15
2.1.4. Routing . . . 16
2.1.4.1. Performance del Routing . . . 17
2.1.5. API di Pastry . . . 18
2.1.6. Auto-organizzazione e adattamento . . . 18
2.1.6.1. Arrivo di un nodo . . . 18
2.1.6.2. Allontanamento di un nodo . . . 19
2.1.7. Località . . . 20
2.1.7.1. Località nella tabella di routing . . . 20
2.1.7.2. Località nel routing . . . 22
2.1.7.3. Localizzare il più vicino tra k nodi . . . 22
2.1.8. Fallimenti arbitrari dei nodi e partizionamento della rete . . . 24
2.2. PAST . . . 25
2.2.1. Introduzione . . . 25
2.2.2. Design di PAST . . . 26
2.2.3. Sicurezza . . . 27
2.2.3.1. Generazione dei nodeId . . . 27
2.2.3.2. Generazione dei certificati dei file e delle ricevute di salvataggio . . . 27
2.2.3.3. Generazione dei certificati e delle ricevute di reclamo . . . 28
2.2.3.4. Quote di storage . . . 28
2.2.3.5. Integrità del sistema . . . 28
2.2.3.6. Persistenza . . . 28

2.2.3.7. Privacy ed integrità dei dati 29
2.2.3.8. Pseudonimia 29
2.2.3.9. Smartcard 29
2.2.3.10. Schema di routing di PAST 29
2.2.4. Storage management 30
2.2.4.1. Cause dello sbilanciamento del carico 30
2.2.4.2. Deviazione delle repliche 31
2.2.4.3. Deviazione dei file 31
2.2.4.4. Mantenimento delle repliche 31
2.2.5. Caching 32
2.3. Erasure Coding 32

3. Un nuovo approccio al calcolo distribuito volontario **33**
3.1. Panoramica sull'architettura del sistema 33
3.1.1. Il funzionamento 33
3.1.2. I nodi dedicati 34
3.1.3. I nodi volontari 34
3.2. Caratteristiche principali di un progetto di ricerca 35
3.2.1. Numero di nodi volontari e dedicati 35
3.2.2. Caratteristiche delle unità di lavoro 35
3.3. Unità di lavoro e risultati calcolati nello storage distribuito 36
3.3.1. Operazioni sulle unità di lavoro 37
3.3.1.1. Distribuzione delle unità di lavoro 37
3.3.1.2. Ricerca delle unità di lavoro 38
3.3.2. Operazioni sui risultati calcolati 39
3.3.2.1. Distribuzione dei risultati calcolati 39
3.3.2.2. Ricerca dei risultati calcolati 39
3.3.3. Considerazioni sulla correttezza 40
3.4. Altri aspetti importanti 41
3.4.1. Utilizzo dello storage distribuito 41
3.4.2. Sicurezza 41
3.4.3. k-replicazione 42

4. Implementazione di un modello di simulazione **45**
4.1. OverSim 45
4.1.1. Introduzione 45
4.1.2. Struttura di OverSim 46
4.1.2.1. Underlay 46
4.1.2.2. Overlay 48
4.1.2.3. Tier Application 48
4.1.2.4. Global Observer 49
4.1.2.5. Churn Generators 49
4.2. L'implementazione del modello 50
4.2.1. Overlay - Pastry 50
4.2.2. Tier 1 Application - SimplePAST 51
4.2.3. Tier 2 Application - DHTVolComp 53
4.2.4. Global Observer - GlobalDHTVolComp 54

5. Risultati e conclusioni **55**
5.1. Simulazioni per valutare il modello di calcolo distribuito volontario . . . 55
5.2. Configurazione delle simulazioni 56
5.3. Risultati 57

5.4. Conclusioni . . . 63

A. Appendice - Calcolo Distribuito Volontario per la Bioinformatica 65
A.1. Rosetta@home . . . 65
A.2. Folding@home . . . 66

B. Appendice - Hash Table, DHT e SHA-1 69
B.1. Hash Table . . . 69
B.1.1. Hash Function . . . 69
B.1.2. Collisioni . . . 69
B.2. DHT - Distributed Hash Table . . . 70
B.2.1. Proprietà . . . 70
B.2.2. Struttura . . . 70
B.2.2.1. Un semplice esempio di funzionamento . . . 71
B.2.2.2. Partizionamento dello spazio delle chiavi . . . 71
B.2.2.3. Rete Overlay . . . 71
B.2.3. Implementazioni . . . 72
B.3. SHA-1 . . . 72

C. Appendice - OMNeT++ 75

Bibliografia 77

Ringraziamenti 81

Sommario

I sistemi di calcolo distribuito volontario vengono utilizzati da molti progetti di ricerca, e, poiché è in crescita il numero di calcolatori a disposizione delle persone, questi sistemi saranno sempre più gettonati. L'architettura su cui essi si basano attualmente è quella della rete a stella: un server centrale distribuisce il lavoro e raccoglie i risultati dai computer dei volontari. Sono noti i problemi di robustezza e scalabilità che una simile architettura di rete comporta. In questo lavoro viene proposta una nuova architettura atta a risolvere le problematiche e le limitazioni degli attuali sistemi di calcolo distribuito volontario, utilizzando le ormai ben consolidate reti paritarie DHT. Le reti paritarie sono note per la loro robustezza e scalabilità, anche se composte da nodi altamente volatili, tutte caratteristiche che vengono in contro alle esigenze del calcolo distribuito volontario.

Introduzione

Il calcolo distribuito volontario è un sistema in cui volontari mettono i processori e la memoria dei loro PC a disposizione di progetti di ricerca scientifica affinché possano raggiungere i loro obiettivi. Sono molti i progetti di ricerca che utilizzano tali sistemi, progetti che spaziano dalla fisica delle alte energie, la biologia molecolare, fino alle previsioni climatiche. Attualmente, tali sistemi sono organizzati secondo il paradigma client/server in senso stretto: un server centrale distribuisce il lavoro ai computer dei volontari; quest'ultimi, dopo aver terminato l'elaborazione, restituiscono al server i risultati.

I computer dei partecipanti ed il server formano una rete a stella, con il server al centro. Questa architettura soffre del problema principale di tutti sistemi centralizzati - il server è il collo di bottiglia di tutto il funzionamento del sistema: le sue prestazioni possono limitare fortemente l'andamento del lavoro di tutta la rete; eventuali crash o guasti al server possono creare situazioni di stallo; il numero di volontari che possono aderire al progetto è limitato dalle capacità di memoria e processore del server stesso. Inoltre, in questa architettura, i volontari non comunicano tra di loro in alcun modo, rendendo il calcolo distribuito volontario utilizzabile solamente da applicazioni CPU intensive altamente parallelizzabili.

In questo lavoro viene proposta una nuova architettura decentralizzata basata sulle ormai ben consolidate reti paritarie DHT, atta a risolvere le problematiche e le limitazioni di quelle attualmente in uso nei sistemi di calcolo distribuito volontario. Le reti paritarie sono note per la loro robustezza e scalabilità, anche se composte da nodi altamente volatili: tutte caratteristiche che vengono in contro alle esigenze del calcolo distribuito volontario, e non solo. Idealmente, un'architettura decentralizzata potrebbe rendere disponibile il calcolo distribuito volontario ad applicazioni più complesse, che necessitano di comunicazione tra i volontari per essere svolte, e non solo di semplice *number-crunching*.

Questo elaborato segue a grandi linee questa struttura:

- Il primo capitolo introduce il calcolo distribuito volontario, fornisce una panoramica sullo stato dell'arte e identifica le principali problematiche legate all'architettura esistente.
- Nel secondo capitolo vengono introdotti la DHT Pastry ed il file system distribuito PAST, utilizzati come base per la nuova architettura di calcolo distribuito volontario.
- Il terzo capitolo introduce il nuovo modello di calcolo distribuito volontario, analizzando nel dettaglio tutto il suo funzionamento.
- Nel quarto capitolo viene descritta l'implementazione del modello proposto in un ambiente di simulazione per reti overlay.
- Infine, nel quinto capitolo vengono presentati i risultati ottenuti durante le sperimentazioni e le relative conclusioni.

1. Il Calcolo Distribuito Volontario

1.1. Gli inizi

L'avvento del calcolatore ha rivoluzionarizzato la ricerca scientifica. Gli scienziati hanno sviluppato modelli matematici accurati dell'universo fisico, e calcolatori programmati con tali modelli possono approssimare la realtà su diverse scale: il nucleo di un atomo, una molecola proteica, la biosfera della Terra o l'intero Universo. Usando questi programmi, si può "predire il futuro", si possono convalidare o confutare teorie, o addirittura operare "laboratori virtuali" che studiano le reazioni chimiche senza doverle per forza realizzare fisicamente.

In genere, una maggiore potenza di calcolo permette una migliore approssimazione della realtà. Questo ha spinto lo sviluppo di calcolatori che operano il più velocemente possibile. Un modo per velocizzare un calcolo è quello di parallelizzarlo, dividerlo in parti che possono essere elaborate da processori separati allo stesso tempo. La maggior parte dei supercomputer moderni funziona in questo modo, usando molti processori contemporaneamente.

Le forze economiche che plasmano la tecnologia favoriscono la larga scala. Una compagnia può spendere più risorse per sviluppare un chip CPU se poi ne potrà vendere a milioni di copie. Così, chip usati nei personal computer (come l'Intel Pentium) si sono sviluppati molto in fretta; infatti, la loro velocità raddoppiava ogni 18 mesi, un andamento noto come "Legge di Moore"[1].

Negli anni '90 ci furono due avvenimenti importanti. Primo, come conseguenza della "Legge di Moore", i PC divennero molto veloci – tanto veloci quanto un supercomputer di pochi anni prima. Secondo, l'Internet diventò disponibile al vasto pubblico. Improvvisamente c'erano milioni di calcolatori veloci connessi alla rete. L'idea di usare questi calcolatori come un supercomputer parallelo la ebbero indipendentemente molti gruppi di ricerca. Nel 1997 emersero due progetti di questo tipo: GIMPS[1], il quale cercava numeri primi molto grandi, e Distributed.net[2], che decifrava messaggi criptati. Questi progetti attirarono migliaia di partecipanti.

Nel 1999 venne lanciato un terzo progetto, SETI@home[3], con lo scopo di scoprire segnali radio emessi da forme di vita intelligente extraterrestre. SETI@home si comporta come uno "screensaver", viene eseguito soltanto quando il PC non è in uso, e fornisce una visualizzazione grafica dell'elaborazione in corso. Oggigiorno questo approccio viene utilizzato in molte aree di ricerca, come la fisica delle alte energie, la biologia molecolare, la medicina, l'astrofisica e la previsione delle dinamiche climatiche ed è noto come *calcolo distribuito volontario*.

1.2. L'importanza

Il calcolo distribuito volontario (Volunteer Computing) è un tipo di calcolo distribuito dove volontari possessori di calcolatori collegati ad Internet forniscono risorse (come i cicli del loro processore o lo spazio sul loro disco) per aiutare progetti di ricerca scientifica.

[1] In elettronica e informatica è indicato come prima legge di Moore il seguente enunciato: "Le prestazioni dei processori, e il numero di transistor ad esso relativo, raddoppiano ogni 18 mesi".

Sebbene le prestazioni dei calcolatori vadano aumentando di anno in anno, la necessità in termini di potenza di calcolo risulta spesso maggiore di ciò che le possibilità economiche degli enti di ricerca offrono. Per ovviare a questo problema gli scienziati dei vari gruppi di ricerca si sono avvalsi dell'aiuto del calcolo distribuito volontario. Dislocati su tutto il pianeta ci sono oltre un miliardo di personal computer, le CPU dei quali vengono generalmente sfruttate solamente per una minima percentuale del tempo in cui essi sono accesi. Da qua l'idea alla base del calcolo distribuito volontario: unire le forze di tanti calcolatori, sfruttando le loro risorse inutilizzate, per ottenere potenze di calcolo molto grandi.

Secondo tale stima, il calcolo volontario distribuito potrebbe, almeno in linea di principio, fornire più potenza di calcolo di qualsiasi altro calcolatore. Esso, infatti, fornisce altissime prestazioni in termini di *numero di operazioni in virgola mobile al secondo* (FLOPS - FLoating point Operations Per Second)[2], confrontabile a quello di un moderno supercomputer. Questa potenza di calcolo rende possibile progetti di ricerca che altrimenti non verrebbero svolti. Inoltre, l'utilizzo di una simile architettura di calcolo è favorito anche dal numero sempre maggiore di calcolatori nelle mani delle persone, non solo PC, ma anche smartphone e console di videogiochi.

Per poter usufruire del calcolo distribuito volontario, i progetti di ricerca devono attirare l'attenzione del vasto pubblico. Un progetto di ricerca con finanziamenti limitati, ma che attrae molti volontari può ottenere grandi potenze di calcolo (i tradizionali supercomputer sono estremamente costosi, e sono a disposizione di chi se li può permettere). Il calcolo distribuito volontario incoraggia l'interesse del pubblico nella scienza[4], dando ad ogni volontario la possibilità di determinare la direzione della ricerca scientifica, scegliendo di sostenere un progetto piuttosto che un altro.

1.3. Uno sguardo sul funzionamento

Un elemento essenziale del calcolo distribuito in generale è l'abilità di suddividere i problemi di calcolo che si intende risolvere in problemi più piccoli e più gestibili, ottenendo così "porzioni" di lavoro da svolgere. Se il problema è di una natura tale che non esiste un modo chiaro per poterlo suddividere in "porzioni", allora il calcolo distribuito non può essere utilizzato per la soluzione di tale problema.

Nel contesto del calcolo distribuito volontario, viene detta *unità di lavoro* (work unit) una certa quantità di dati che il progetto scientifico deve far analizzare dai calcolatori dei partecipanti. L'unità di lavoro viene replicata in un certo numero di istanze di calcolo, dette *risultati* (results), le quali vengono poi inviate ai calcolatori dei partecipanti per essere elaborate[3].

Le dimensioni (oltre che i contenuti) delle unità di lavoro dipendono esclusivamente dal progetto stesso, e possono variare da dimensioni relativamente piccole (da 200 a 500 Kilobyte) a molto grandi (diversi Megabyte). Il tempo necessario all'elaborazione è a sua volta dipendente dal progetto, variando da pochi minuti fino a mesi interi.

[2] I moderni processori includono una *floating point unit* (FPU), componente specializzata nel calcolo delle operazioni in virgola mobile. Quindi il FLOPS è un'unità di misura delle prestazioni della FPU.

[3] L'intuito induce a pensare che i volontari ricevano le unità di lavoro, e spesso è questo il senso che viene dato al termine. In realtà è il risultato (che è una istanza o copia dell'unità di lavoro) ad essere inviato al PC del volontario partecipante al progetto. Il risultato descrive un'istanza di calcolo che può essere non ancora avviata, in esecuzione o completata. Un'altra possibile interpretazione è pensare all'unità di lavoro come ad un "pacchetto" contenente diversi risultati, insieme all'informazione che verrà prodotta dalla computazione di quest'ultimi. Quando un volontario riceve una "work unit", in realtà sta ricevendo un risultato con la porzione "risultato calcolato" vuota (la risposta verrà creata come conseguenza diretta del calcolo svolto sui dati).

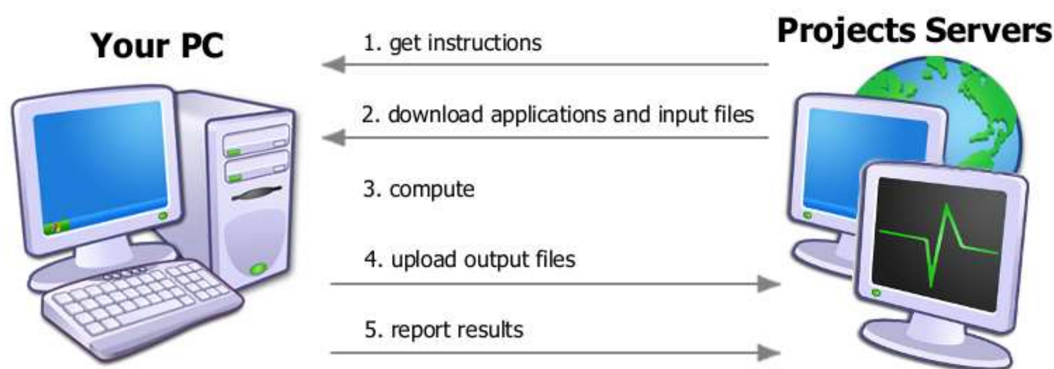

Figura 1.1.: Schema di funzionamento di un generico sistema di volunteer computing

I sistemi di calcolo distribuito volontario hanno praticamente tutti la stessa struttura di base: un programma client viene eseguito sul calcolatore del volontario. In base alle proprie preferenze personali, un volontario decide a quale progetto di ricerca afferire, e scarica ed installa sul proprio calcolatore il relativo software client.

Quando il software client sul PC del volontario non ha lavoro da svolgere, invia una richiesta al proprio server. Il server cerca tra le unità di lavoro disponibili ad essere inviate uno o più risultati che corrispondano alle capacità di elaborazione e di memoria del PC del volontario: il server non invierà file che richiedono più risorse di quelle disponibili[4].

Un singolo progetto può supportare diverse applicazioni, ed il server può inviare ai volontari i file di ciascuna applicazione. Oltre ai file da elaborare, i server dei progetti scientifici inviano anche le nuove versioni delle proprie applicazioni con eventuali aggiornamenti, i quali vengono scaricati automaticamente sui PC dei volontari.

Dopo aver eseguito le elaborazioni necessarie ed aver così ottenuto i rispettivi "risultati calcolati", il software client invia al server del progetto i file di output, e notifica il server del loro completamento. A questo punto il client può richiedere una nuova serie di file da elaborare. Questo ciclo avviene in modo automatico indefinitamente.

Perché funzioni correttamente, un sistema di calcolo di questo genere deve avere dei meccanismi che permettano di:

- tracciare l'andamento delle unità di lavoro ed assicurino che vengano calcolati tutti i risultati che le compongono;
- validare i risultati, per avere la certezza che l'elaborazione ottenga dei risultati corretti;
- integrare i risultati calcolati per ottenere il prodotto finale del lavoro, detto anche *Risultato Canonico*;
- gestire efficientemente le connessioni per le comunicazioni, in modo da distribuire il lavoro e raccogliere i risultati calcolati.

[4] In questi casi il software client, dopo che è stato installato sul PC del volontario, esegue un benchmark sulle prestazioni del calcolatore (CPU, memoria, spazio su disco, velocità della connessione al server, ecc.). In base al risultato del benchmark viene poi assegnato il tipo di job da eseguire.

1.4. I framework

I primi software client dei progetti di calcolo distribuito volontario consistevano in singoli programmi sviluppati dagli scienziati dei corrispettivi progetti di ricerca. Questi programmi combinavano il calcolo scientifico con l'architettura di calcolo distribuito in un'unica struttura monolitica, il che le rendeva poco flessibili. Infatti, risultava molto difficile lanciare nuove versioni delle applicazioni e/o eventuali aggiornamenti parziali.

Di recente sono state introdotte diverse piattaforme software specifiche per il calcolo distribuito volontario. Queste piattaforme sono dei middleware che permettono, a chiunque lo desideri, di creare un nuovo progetto di calcolo distribuito volontario. Infatti, mirano a disaccoppiare il più possibile le problematiche legate al sistema distribuito da quelle del calcolo scientifico vero e proprio, permettendo così agli scienziati di potersi concentrare di più su quest'ultimo.

Tra questi framework possiamo elencare:

- La Berkeley Open Infrastructure for Network Computing (BOINC)[5, 6, 7, 8] è la più diffusa tra le piattaforme per il calcolo distribuito volontario. Offre software client per Windows, Mac OS X, Linux, ed altre varianti di Unix.
- XtremWeb[9] viene usato principalmente come strumento di ricerca ed è stato sviluppato da un gruppo di ricerca presso la Université Paris-Sud 11.
- Xgrid[10] è stato sviluppato da Apple. Le sue componenti client e server supportano solamente il Mac OS X. Attualmente Apple ha interrotto il progetto con l'uscita dell'OS X v10.8 (Mountain Lion).
- Grid MP[11] è una piattaforma commerciale sviluppata da United Devices la quale veniva usata in progetti di calcolo distribuito volontario come grid.org, World Community Grid, Cell Computing, e Hikari Grid.

BOINC è il framework più utilizzato dai progetti di ricerca per creare applicazioni di calcolo volontario distribuito, tanto da farne senza alcun dubbio l'esponente più rappresentativo della categoria. Per questo motivo, nella parte che segue, verranno descritte nel dettaglio le caratteristiche di quest'ultimo.

1.5. BOINC

La piattaforma Berkeley Open Infrastructure for Network Computing (BOINC)[5, 6, 7, 8] è un middleware open source per il calcolo distribuito volontario. BOINC è stato creato da un team di sviluppatori dello Space Sciences Laboratory (SSL) presso la University of California, Berkeley.

Figura 1.2.: Logo del progetto BOINC

Il framework BOINC semplifica molto il processo di creazione e di utilizzo di un progetto di calcolo distribuito volontario. Applicazioni esistenti scritte nei linguaggi più comuni (C, C++, Fortran) possono essere eseguite come applicazioni scientifiche BOINC (porting) con pochissime modifiche. Un'applicazione può essere composta da diversi file (programmi multipli e script che li coordinano). Le nuove versioni delle applicazioni possono essere lanciate senza necessità di intervento da parte del partecipante.

Il sistema BOINC offre diversi livelli di sicurezza. Ad esempio, dà la possibilità di utilizzare firme digitali basate su crittografia a chiave pubblica per proteggere dalla distribuzione di virus. Inoltre esiste la possibilità di eseguire il software client in modalità sandbox[5], per proteggere i dati dell'utente volontario.

I progetti BOINC-based possono avere server multipli, nonché server dati e di schedulazione dei job separati. I client provano automaticamente a connettersi a server alternandoli; se i server sono tutti inaccessibili, i client usano il back-off esponenziale per evitare il flooding dei server quando questi torneranno attivi.

Tutto il software BOINC viene distribuito sotto una licenza pubblica che ne permette l'utilizzo libero e gratuito per progetti pubblici o privati, con la restrizione di non poter essere utilizzato come base di un prodotto commerciale. Le applicazioni che usano BOINC non devono necessariamente essere open-source. Ogni progetto deve installare e mantenere i propri sistemi server: questi sistemi si possono installare facilmente usando componenti open-source (MySQL, PHP, Apache).

BOINC può essere utilizzato da molti progetti diversi. I progetti sono indipendenti; ciascuno opera separatamente sui propri server e database. Tuttavia, i progetti possono condividere risorse nel seguente senso: i partecipanti installano il software client BOINC come un qualsiasi altro programma, il quale scarica ed esegue le applicazioni specifiche dei progetti ai quali il partecipante decide di afferire. I partecipanti possono scegliere a quanti e quali progetti partecipare, e come ripartire le proprie risorse tra questi ultimi. Quando il server di un progetto ha problemi tecnici o termina il lavoro da inviare, le risorse dei suoi partecipanti vengono suddivise fra gli altri progetti a cui i partecipanti afferiscono.

Dal punto di vista dei partecipanti ai progetti, BOINC offre diversi vantaggi. Il software client BOINC è disponibile per tutti i sistemi operativi più comuni (Mac OS X, Windows, Linux ed altri sistemi Unix), e può utilizzare CPU multiple. Inoltre fornisce interfacce web per la creazione di account per i volontari, la modifica delle preferenze e la visualizzazione dello stato dei partecipanti. Le preferenze di un partecipante possono essere facilmente propagate a tutte le sue macchine con cui partecipa ai progetti di ricerca, rendendo così facile la gestione di molteplici host.

In genere, i server dei progetti tengono traccia di quanto lavoro ha svolto ciascun partecipante; questo viene chiamato credito. Il credito indica il contributo di un utente ad un certo progetto BOINC a cui afferisce. Serve ai partecipanti come metrica per valutare il loro contributo individuale. Per assicurare che il credito venga assegnato in maniera giusta, la maggior parte dei progetti BOINC-based lo assegna secondo lo schema che segue:

- Ogni lavoro può essere assegnato a due computer.
- Quando un calcolatore riporta un risultato, richiede una certa quantità di credito, basandosi su quanto tempo CPU ha impiegato per l'elaborazione.
- Quando almeno due risultati vengono riportati, il server li confronta tra di loro. Se i risultati combaciano, allora agli utenti viene assegnata il minimo tra i valori di credito richiesto.

1.5.1. Architettura di un sistema BOINC

Un sistema BOINC consiste in due strati (layer) che operano secondo l'architettura client-server. Una volta che il software client BOINC viene installato su una macchina,

[5]Nella sicurezza informatica il "sandbox" indica un meccanismo di sicurezza per separare i programmi in esecuzione. Viene spesso usato per eseguire codice non testato, oppure programmi non affidabili, con provenienza da fonti non fidate o terze parti non certificate.

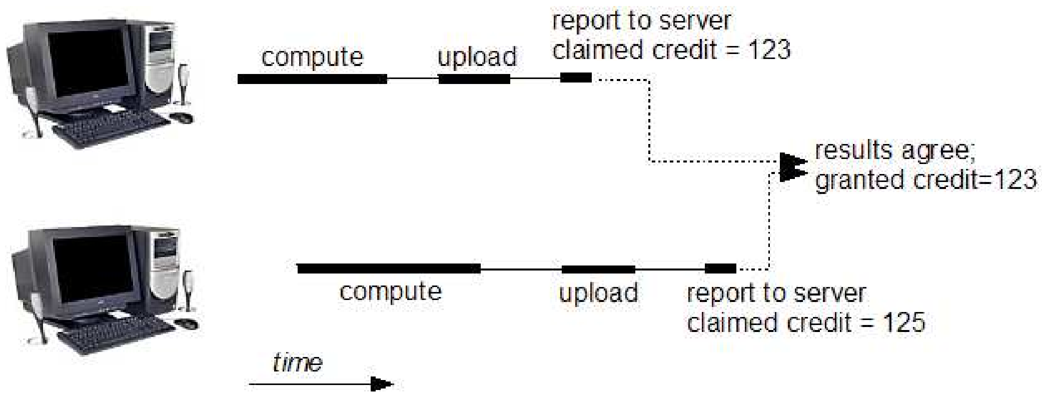

Figura 1.3.: Assegnazione del credito in un progetto BOINC-based

il server inizia ad inviare i file di lavoro al client. Il calcolo avviene al lato client, invece l'output viene inviato al lato server.

1.5.1.1. Il software client

Il software client BOINC è strutturato in diverse applicazioni. Queste comunicano tra di loro usando il meccanismo di chiamata di procedura remota (RPC) di BOINC. Le applicazioni che lo compongono sono:

Il *core* (nocciolo) del client che consiste nel programma *boinc* (*boinc.exe*). Il core si occupa delle comunicazioni tra client e server. Scarica le applicazioni scientifiche, fornisce un meccanismo unificato di login per la gestione dei profili utente (impostazioni e credito), assicura che i file eseguibili siano aggiornati alle ultime versioni e spartisce se risorse CPU tra le diverse applicazioni scientifiche (se ce ne sono più di una installate). Su Unix il core viene in genere eseguito come un daemon, ed occasionalmente come uno job cron. Su Windows invece si può scegliere durante l'installazione se il client sarà eseguito come un servizio di sistema, o come semplice applicativo. I meccanismi di gestione degli aggiornamenti e di ricezione automatica del lavoro da svolgere forniti dal core semplificano moltissimo la programmazione delle applicazioni scientifiche.

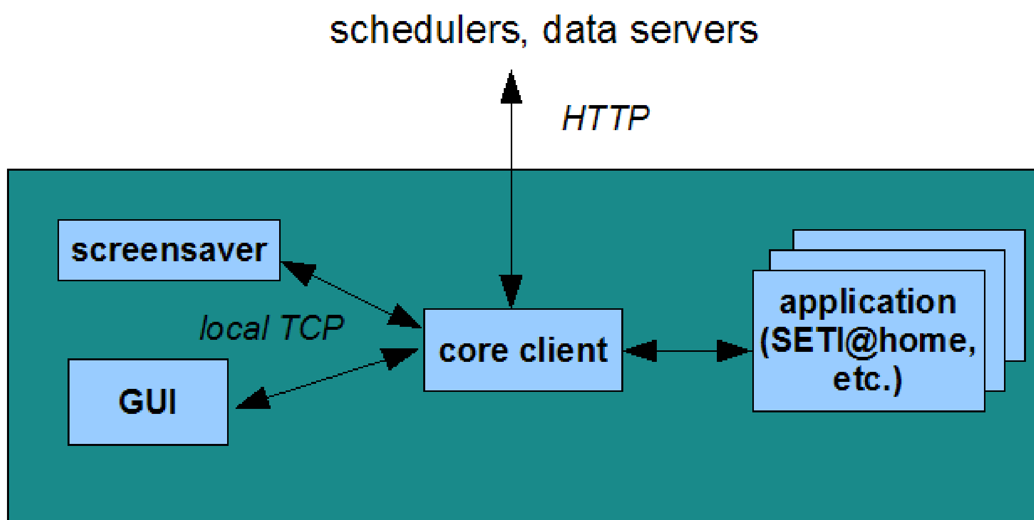

Figura 1.5.: Struttura sel software client BOINC

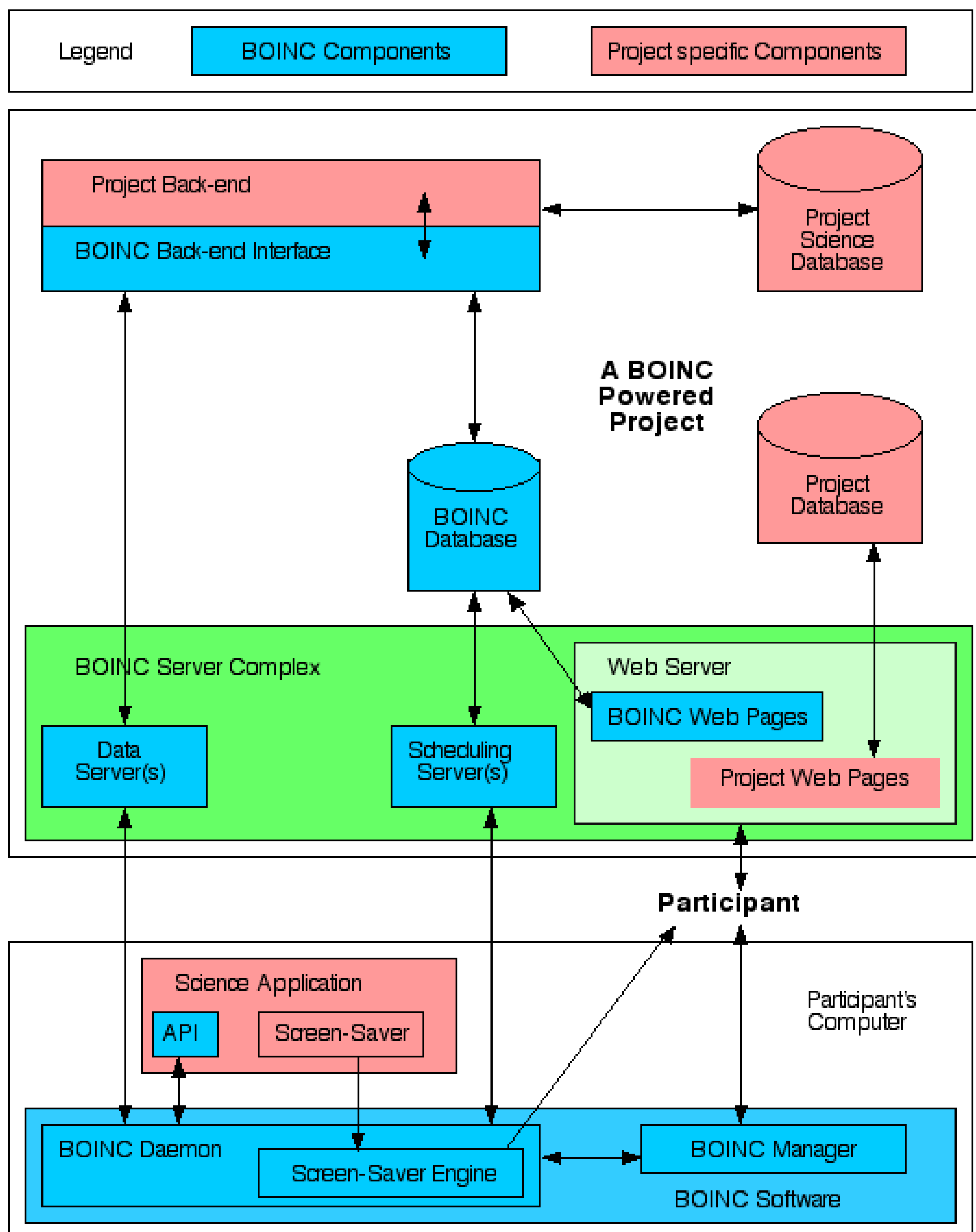

Figura 1.4.: Architettura di un sistema BOINC-based

Una o più *applicazioni scientifiche* che eseguono i calcoli scientifici veri e propri. Esiste una specifica applicazione scientifica per ciascun progetto di calcolo distribuito volontario che utilizza il framework BOINC. Le applicazioni scientifiche usano il core del client per scambiare lavoro, risultati e statistiche con i propri server.

Una *GUI* (Graphical User Interface - Interfaccia Grafica Utente), *boincmgr* (o *boincmgr.exe*), che comunica con il core dell'applicazione usando chiamate di procedura remota. Di default, il core permette solo connessioni dallo stesso computer, però può essere configurato per riceverne anche da altri computer (tramite meccanismo di autenticazione con password); questo meccanismo permette ad una persona di gestire molteplici installazioni del software client BOINC da una singola workstation. La GUI è stata scritta usando il toolkit WxWidgets, in modo da fornire la stessa user experience anche su piattaforme diverse. Tramite la GUI gli utenti si possono connettere ai core dei client, possono istruire i client ad installare nuove applicazioni scientifiche, possono monitorare il progresso dei calcoli in corso, e possono visualizzare i messaggi log del sistema BOINC.

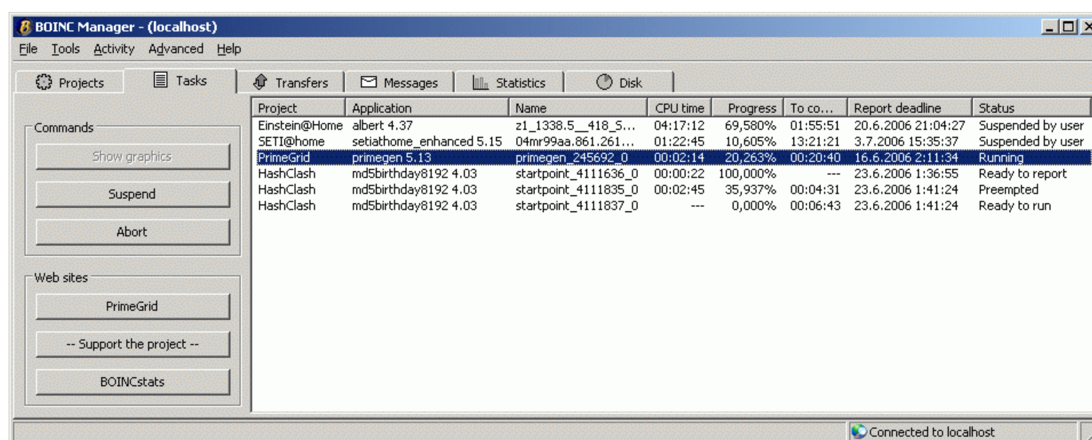

Figura 1.6.: La GUI del sofware client BOINC

Lo *screensaver*, il quale fornisce la possibilità alle applicazioni scientifiche di usare un ambiente di visualizzazione grafico. Gli screensaver BOINC vengono programmati usando la BOINC graphics API, Open GL, ed il GLUT toolkit. Tipicamente mostrano animazioni grafiche che rappresentano l'elaborazione in corso.

1.5.1.2. Il server

Il server nei sistemi BOINC può essere eseguito su una o più machine per permettere una certa scalabilità ai progetti di calcolo distribuito volontario. I server BOINC girano su sistemi operativi Linux ed usano Apache, PHP e MySQL come base dei loro sistemi web e di database.

Il server consiste in due programmi CGI[6] e (normalmente) cinque daemon, scritti in C++. Le computazioni che devono essere eseguite dai PC dei partecipanti volontari vengono chiamate unità di lavoro (work units). Un risultato descrive una istanza di una unità di lavoro, anche se non è stata ancora avviata e/o completata. Un progetto non crea i risultati esplicitamente; essi vengono creati dalle unità di lavoro in modo automatico dal server.

[6] Common Gateway Interface (interfaccia comune, nel senso di standard, per gateway), è una tecnologia standard usata dai web server per interfacciarsi con applicazioni esterne. Ogni volta che un client richiede al web-server un URL corrispondente ad un documento in puro HTML gli viene restituito un documento statico (come un file di testo); se l'URL corrisponde invece ad un programma CGI, il server lo esegue in tempo reale, generando dinamicamente informazioni.

Il programma CGI scheduler gestisce le richieste dai client, ricevendo i risultati completati ed inviando nuovo lavoro da eseguire. Lo scheduler non prende i risultati disponibili direttamente dal database; esiste un daemon feeder che carica i task dal database, e li mantiene su un blocco di memoria condivisa, dove lo scheduler va a leggere. Il feeder riempie periodicamente gli "slot" che si liberano in questo blocco di memoria condivisa dopo che lo scheduler ha inviato quei risultati ai client.

Quando tutti i risultati di una unità di lavoro vengono completati e riportati al server, il validatore li confronta. Il validatore può avere codice dipendente dallo specifico progetto scientifico per eseguire confronti "fuzzy" tra i risultati, oppure può semplicemente eseguire un confronto bitwise. Se i risultati combaciano, l'unità di lavoro viene segnata come valida, gli utenti ottengono il loro credito, e viene creato un risultato canonico.

Dopodiché il daemon assimilatore elabora i risultati canonici usando codice dipendente dallo specifico progetto scientifico. Per esempio, alcuni progetti analizzano i file e salvano le informazioni in un database, altri semplicemente copiano i file da qualche altra parte. Un assimilatore può anche generare altre unità di lavoro in base ai risultati ottenuti fino a quel momento.

Il file_deleter daemon cancella i file di output dopo che l'assimilatore li ha analizzati, e cancella i file di input non più necessari.

Il daemon delle transizioni gestisce le transizioni di stato delle unità di lavoro e dei risultati. Inoltre genera i risultati dalle unità di lavoro quando queste vengono create la prima volta, o quando c'è bisogno di averne altri (per esempio, quando si ottiene un risultato invalido).

1.5.1.3. Ulteriori funzionalità dei server BOINC

I server BOINC forniscono ulteriori funzionalità che verranno di seguito esplicate.

La *ridondanza omogenea*. Molte applicazioni numeriche producono risultati diversi dato lo stesso input, in base all'architettura della macchina utilizzata, al sistema operative, al compilatore e a dirittura ai flag di compilazione. Per alcune applicazioni queste discrepanze producono solo piccole differenze sull'output finale, e i risultati possono essere validati tramite una funzione di confronto "fuzzy" che permette piccole deviazioni. Altre applicazioni sono "divergenti" nel senso che piccole differenze numeriche portano a differenze imprevedibilmente grandi nell'output finale. Per tali applicazioni è difficile distinguere tra risultati corretti ma differenti per causa di discrepanze numeriche, e risultati errati. L'approccio del confronto "fuzzy" non è più applicabile.

BOINC fornisce una funzionalità chiamata *ridondanza omogenea* per gestire le applicazioni divergenti. La ridondanza omogenea suddivide gli host partecipanti in "classi di equivalenza numerica": due host sono nella stessa classe se e solo se essi restituiscono risultati identici datogli lo stesso input. Lo scheduler di BOINC invierà i risultati corrispondenti ad una unità di lavoro data solamente a host che si trovano nella stessa classe di equivalenza numerica; questo permette di usare l'uguaglianza stretta per confrontare i risultati.

Workunit trickling (letteralmente far sgocciolare le unità di lavoro). Unità di lavoro molto grandi impiegano ovviamente molto tempo per essere completate. I progetti che hanno simili unità di lavoro potrebbero voler dare l'opportunità ai propri partecipanti di guadagnare credito più frequentemente, cosicché i volontari non debbano aspettare troppo a lungo perché il calcolo giunga a temine, come avviene con le unità di lavoro comuni che impiegano soltanto alcune ore ad essere elaborate. Quindi i progetti possono sviluppare le loro applicazioni scientifiche in modo che le unità di lavoro di questo genere possano riportare quanto lavoro è stato svolto da un dato partecipante anche durante l'elaborazione. Questo avviene tramite i messaggi trickle. Questi messaggi possono anche essere utilizzati per controllare l'avanzamento parziale del lavoro, e abortire il

calcolo in caso vengano rilevati errori nei risultati parziali, in modo da non dover rifare una lunga ed onerosa computazione da capo.

Localality scheduling. Questa funzionalità permette di inviare determinati file da elaborare a determinati host, sapendo che questi ultimi possiedono già risultati di computazioni precedenti, necessari allo svolgimento di quella corrente, al fine di minimizzare le comunicazioni e gli spostamenti di dati.

Distribuzione mirata del lavoro. La distribuzione del lavoro avviene in base ai parametric del calcolatore partecipante. Per esempio, computazioni richiedenti 512 MB di RAM verranno inviati soltanto agli host aventi almeno quella quantità di memoria.

1.6. Grid vs. Volunteer Computing

Il grid computing è una forma di calcolo distribuito dove una organizzazione (azienda, università, ecc.) utilizza i propri computer esistenti per eseguire calcoli che necessitano prestazioni elevate in termini di potenza di calcolo. Questo differisce dal calcolo volontario distribuito sotto diversi aspetti:

- Le risorse (i calcolatori) sono affidabili; si può assumere con parecchia tranquillità che i risultati ottenuti dai propri PC non sono intenzionalmente sbagliati, ed il credito richiesto non è falsificato. Quindi non c'è bisogno di replicazione.
- Non c'è bisogno di usare gli screensaver; anzi potrebbe essere desiderabile fare in modo che il calcolo avvenga in modo totalmente invisibile ai semplici utilizzatori dei computer e fuori dal loro controllo.
- L'installazione del software client è tipicamente automatizzata.
- Il numero di calcolatori da gestire è generalmente più piccolo in un sistema grid, anche di interi ordini di grandezza.
- In un sistema grid i calcolatori non sono volatili (a meno di guasti).

Quindi complessivamente possiamo affermare che un sistema grid risulta essere molto più semplice da gestire rispetto ad un sistema di calcolo volontario distribuito.

1.7. Punti deboli e problemi

Come si può chiaramente notare dagli esempi precedenti, il volunteer computing allo stato dell'arte è organizzato secondo il classico paradigma client-server. Nodi volontari e server formano una rete a stella, con il server al centro. Le comunicazioni avvengono esclusivamente tra i nodi volontari ed il server: non c'è comunicazione tra i volontari. Il server gestisce lo scheduling decidendo ogni volta quale job verrà assegnato a quale volontario.

In questa architettura il server è il collo di bottiglia di tutto il funzionamento del sistema: tutte le comunicazioni devono passare per il server. Questa caratteristica limita le possibili applicazioni del calcolo distribuito volontario principalmente a problemi altamente parallelizzabili di *number-crunching*[7]. Con questa architettura è quasi impensabile affrontare problemi che necessitano di scambi continui di dati anche tra i nodi volontari. Inoltre, anche limitando il campo ai soli problemi CPU-intensive facilmente

[7]Dette anche applicazioni CPU-intensive, sono applicazioni che ricevono pochi dati in input, sui quali eseguono una elevatissima quantità di calcoli, e di conseguenza, impiegano molto tempo per completare l'elaborazione. Si contrappongono alle applicazioni data-intensive, che ricevono grandi quantità di dati in input, che però vengono elaborati velocemente.

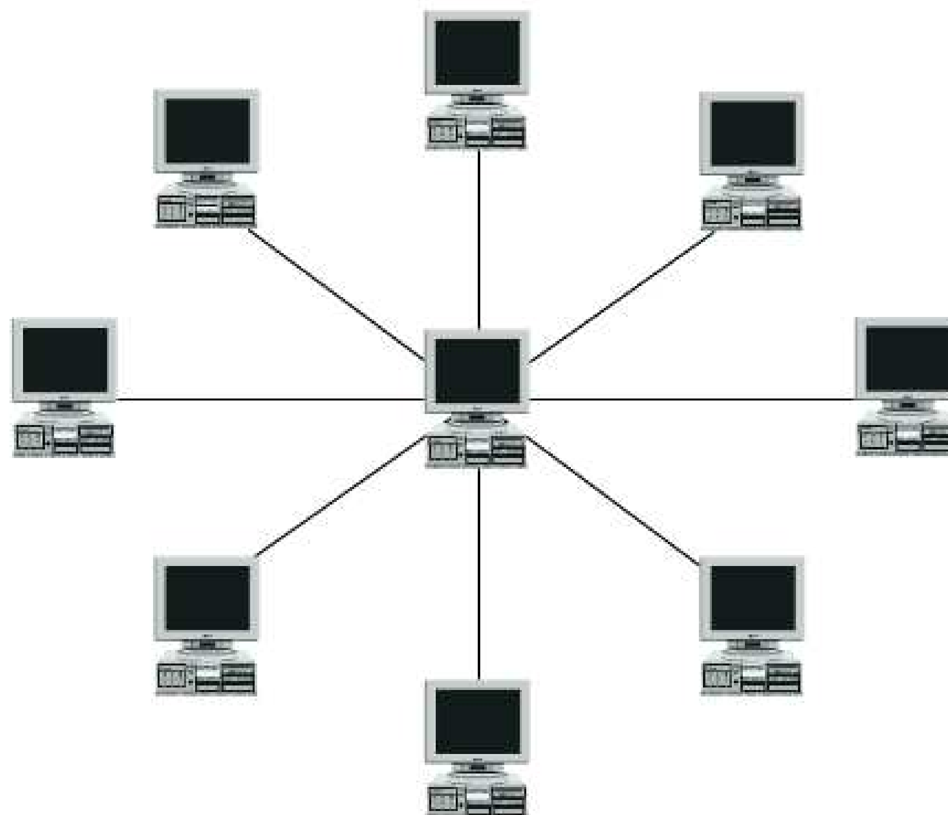

Figura 1.7.: Rete a stella

parallelizzabili, le necessità di maggiori prestazioni del server (in termini di CPU e memoria, ma anche di banda) si può facilmente presentare a fronte di un incremento del numero di volontari o della quantità di dati da elaborare. Inoltre, se il server è inattivo per qualche motivo, l'intera rete di calcolo distribuito potrebbe rimanere inutilizzata per giorni.

Per ovviare a questo problema si vuole introdurre una nuova architettura basata sulle reti paritarie strutturate. Con un'architettura decentralizzata, che permette le comunicazioni tra i client volontari (ad esempio con una API simile alle direttive MPI), sarà possibile estendere il calcolo volontario distribuito a nuovi scenari di applicazioni data-intensive, le quali attualmente sono proibitive data la loro necessità di grossi trasferimenti di dati.

Le reti peer-to-peer sono una tecnologia ormai ben collaudata, scalano bene al crescere del numero di partecipanti, sono robuste anche in presenza di nodi fortemente volatili (che si connettono e si disconnettono molto frequentemente) e non necessitano di un server centrale per funzionare. In questo lavoro viene proposta un'architettura basta su DHT, più precisamente su PAST e Pastry, che non necessita di un server centrale di grandi prestazioni e sempre attivo per funzionare.

PAST è un sistema di storage distribuito a larga scala su Internet, che fornisce scalabilità, alti livelli di disponibilità e sicurezza. È un'applicazione Internet peer-to-peer completamente auto-organizzabile, dove i nodi non sono affidabili, possono connettersi, sconnettersi o malfunzionare in qualsiasi momento senza avvertimento. Comunque, il sistema è in grado di fornire garanzie, accesso efficiente allo storage, bilanciamento del carico e scalabilità.

PAST utilizza lo schema di localizzazione e routing di Pastry, il quale esegue il routing delle richieste dei client tra i nodi Pastry in modo affidabile ed efficiente, ha buone

proprietà di località sulla rete e risolve automaticamente i fallimenti e le aggiunte di nuovi nodi.

Nel Capitolo 2 di questo elaborato vedremo come si può migliorare l'architettura esistente dei sistemi di calcolo distribuito volontario sfruttando la rete PAST/Pastry. Prima però dobbiamo capire come sono costruite e come funzionano quest'ultime.

2. Lookup, Routing, e Storage Distribuito

Come già accennato, questo lavoro ha come obiettivo principale quello di introdurre una nuova architettura di calcolo distribuito volontario, con una serie di miglioramenti rispetto a quella attuale, come: la decentralizzazione del sistema di calcolo, migliore scalabilità e robustezza, nonché la possibilità di eventuali comunicazioni tra i nodi volontari. In questo capitolo descriveremo nel dettaglio alcune delle tecnologie e degli strumenti con i quali sarà possibile costruire la nostra architettura di calcolo.

2.1. Pastry

Nelle sezioni che seguono, descriveremo Pastry[12, 13, 14, 15, 16], un substrato di routing e localizzazione, scalabile e distribuito, per le applicazioni peer-to-peer su larga scala. Pastry esegue il routing e la localizzazione di oggetti a livello di applicazione su reti overlay di nodi connessi ad Internet, potenzialmente molto grandi. Può essere usato per costruire una varietà di applicazioni peer-to-peer, inclusi lo storage distribuito, il data sharing, le comunicazioni di gruppo e i servizi di naming.

Ogni nodo nella rete Pastry ha un identificatore unico (nodeId). Quando ad un nodo Pastry si presenta una coppia (chiave, messaggio), esso esegue il routing efficiente del messaggio al nodo con il nodeId numericamente più vicino alla chiave, tra tutti i nodi attualmente attivi di Pastry. Ogni nodo Pastry tiene traccia dei suoi più immediati vicini nello spazio dei nodeId, e notifica le applicazioni di eventuali arrivi di nuovi nodi, malfunzionamenti o recuperi di nodi esistenti. Pastry tiene presente la località nella rete, e mira a minimizzare le distanze percorse dai messaggi, secondo una metrica scalare di prossimità come il numero di hop durante il routing IP o il delay dei messaggi ICMP.

Pastry è completamente decentralizzata, scalabile ed auto-gestita; si adatta automaticamente agli arrivi, agli allontanamenti e ai malfunzionamenti dei nodi.

2.1.1. Introduzione

Le applicazioni peer-to-peer sono state rese popolari dalle applicazioni di file-sharing. Nonostante gran parte dell'attenzione sia stata posta su questioni di copyright alzate da queste applicazioni, i sistemi peer-to-peer hanno molti aspetti tecnici interessanti, come il controllo decentralizzato, l'auto-organizzazione, l'adattabilità e la scalabilità. I sistemi peer-to-peer possono essere definiti come sistemi distribuiti nei quali i nodi hanno capacità e responsabilità identiche, e tutte le comunicazioni sono simmetriche.

Uno dei problemi principali nelle applicazioni peer-to-peer su larga scala è quello di riuscire a fornire algoritmi efficienti di routing e di localizzazione di oggetti sulla rete. Pastry propone uno schema generico di routing e localizzazione, basato su una rete overlay auto-organizzante di nodi connessi ad Internet. Pastry è completamente decentralizzato, resistente ai guasti, scalabile ed affidabile, ed inoltre ha buone proprietà di routing.

Pastry è inteso come un substrato generico per la costruzione di una varietà di applicazioni Internet peer-to-peer come il file sharing, lo storage distribuito, le comunicazioni di gruppo e i sistemi di naming. Fino ad oggi sono state costruite diverse applicazioni

basate su Pastry, incluso un sistema di storage distribuito chiamato PAST[17, 18], e un sistema scalabile di tipo publish/subscribe chiamato SCRIBE[19, 20, 21].

Pastry fornisce le seguenti funzionalità. Ogni nodo nella rete Pastry ha un unico identificatore numerico (nodeId). Quando ad un nodo si presenta un messaggio ed una chiave numerica, esso effettua efficientemente il routing del messaggio al nodo con nodeId numericamente più vicino alla chiave, tra tutti i nodi Pastry correntemente attivi. Il numero atteso di passi nel routing è $O(\log N)$, dove N è il numero di nodi Pastry che compongono la rete. Ad ogni nodo Pastry che il messaggio incontra durante il tragitto verso la sua destinazione, viene notificato il livello applicativo, il quale può eseguire computazioni specifiche all'applicazione in base al messaggio.

Pastry prende in considerazione la località della rete; mira a minimizzare la distanza che i messaggi percorrono, secondo una metrica scalare di prossimità come il numero di hop nel routing IP del messaggio. Ogni nodo Pastry tiene traccia dei suoi immediati vicini nello spazio dei nodeId, e notifica le applicazioni di eventuali arrivi di nuovi nodi, malfunzionamenti o recuperi di nodi esistenti. Poiché i nodeId vengono assegnati in maniera aleatoria, con altissima probabilità, l'insieme di nodi con nodeId adiacenti è diverso geograficamente, dal punto di vista proprietario, della giurisdizione, ecc. Le applicazioni possono sfruttare questa caratteristica, poiché Pastry può effettuare il routing ad uno dei nodi numericamente più vicini alla chiave. Un'euristica assicura che, tra i nodi con nodeId numericamente più vicini alla chiave data, il messaggio viene inoltrato, con molta probabilità, ad un nodo vicino in termini della metrica di prossimità, al nodo da cui il messaggio ha avuto origine.

Le applicazioni usano queste funzionalità in modi diversi. PAST, per esempio, usa un fileId come chiave Pastry per un file, calcolato come l'hash del nome del file. Le repliche del file vengono salvate sui k nodi Pastry con nodeId numericamente più vicini al fileId. Un file può essere richiesto con l'invio di un messaggio tramite Pastry, usando il fileId come chiave. Per definizione, la ricerca garantisce di arrivare ad un nodo che contiene il file, finché almeno uno dei k nodi è ancora attivo. In più, segue che il primo nodo che il messaggio raggiunge, tra i k che contengono il file, sarà, molto probabilmente, un nodo vicino al client; quel nodo consumerà il messaggio e risponderà con il file desiderato. Il meccanismo di notifica di Pastry permette a PAST di mantenere repliche di un file sui k nodi con nodeId più vicini alla chiave, indipendentemente da malfunzionamenti, nuovi arrivi o allontanamenti dei nodi, usando solamente coordinamento locale tra nodi adiacenti.

2.1.2. Design di Pastry

Un sistema Pastry è una rete overlay di nodi che si auto-organizzano, dove ognuno dei nodi effettua il routing delle richieste dei client, ed interagisce con le istanze locali di una o più applicazioni. Qualsiasi calcolatore che si connette ad Internet ed usa il software del nodo Pastry può agire come tale, soggetto solo alle politiche di sicurezza specifiche dell'applicazione stessa.

Ad ogni nodo nella rete overlay peer-to-peer Pastry viene assegnato un identificatore da 128 bit (nodeId). Il nodeId viene usato per indicare la posizione del nodo su uno spazio circolare di nodeId, il quale spazia da 0 a 2^{128}–1. Il nodeId viene assegnato casualmente quando un nodo si collega al sistema. Si assume che i nodeId vengano generati con distribuzione uniforme sull'insieme $\{0, 1, 2, \ldots, 2^{128} - 1\}$. Per esempio, i nodeId potrebbero essere generati calcolando l'hash crittografico della chiave pubblica del nodo o del suo indirizzo IP. Come risultato di un'assegnazione casuale dei nodeId, con alta probabilità, i nodi con nodeId adiacenti sono diversi geograficamente, dal punto di vista proprietario, giuridico, ecc.

Supponendo che una rete consista di N nodi, Pastry può eseguire il routing di una data chiave al nodo col nodeId numericamente più vicino in meno di $\lceil \log_{2^b} N \rceil$ in condizioni operative normali (b è un parametro di configurazione con valore tipicamente 4). Anche a fronte di continui malfunzionamenti dei nodi, l'eventuale inoltro del messaggio è garantito a meno che $\lfloor |L|/2 \rfloor$ nodi con nodeId adiacenti falliscano contemporaneamente. ($|L|$ è un parametro di configurazione con valore tipicamente 16 o 32). Segue lo schema di funzionamento di Pastry.

A scopo di routing, i nodeId e le chiavi vengono interpretate come sequenze di cifre con base 2^b. Pastry effettua il routing dei messaggi verso il nodo il cui nodeId è quello numericamente più vicino alla data chiave. Questo avviene come segue. Consideriamo il generico i-esimo passo di routing, dove il messaggio si trova all'i-esimo nodo incontrato a partire dall'origine verso la destinazione, e deve andare all'$(i+1)$-esimo nodo. La chiave utilizzata ha in comune col nodeId dell'i-esimo nodo un prefisso lungo l_i. Il nodo $(i+1)$-esimo deve avere un nodeId con prefisso comune con la chiave più lungo di l_i di almeno una cifra (b bit), cioè $l_{i+1}–l_i \geq 1$. Se non si conosce un nodo simile, il messaggio viene inoltrato ad un nodo il cui nodeId ha in comune con la chiave un prefisso lungo quanto il prefisso comune del nodo corrente, cioè $l_{i+1} = l_i$, ma che è numericamente più vicino alla chiave di quanto non lo sia il nodeId del nodo corrente ($\|nodeId_i - chiave\| > \|nodeId_{i+1} - chiave\|$). Per realizzare questa procedura di routing, ogni nodo mantiene alcune informazioni di stato, che verranno descritte di seguito.

2.1.3. Stato del nodo Pastry

Ognuno dei nodi Pastry mantiene una tabella di routing, un insieme di vicinanza (neighborhood set) e un insieme di foglie (leaf set). Iniziamo con la descrizione della tabella di routing. La tabella di routing R di un nodo è organizzata in $\lceil \log_{2^b} N \rceil$ righe con $2^b - 1$ entry ciascuna. Ciascuna delle $2^b - 1$ entry alla riga n della tabella punta ad un nodo il cui nodeId ha in comune col il nodeId del nodo presente le prime n cifre, differendo però alla $(n+1)$-esima.

Ogni entry della tabella di routing contiene gli indirizzi IP di uno dei (potenzialmente tanti) nodi i cui nodeId hanno il prefisso appropriato; in pratica, viene scelto un nodo che è vicino al nodo corrente secondo la metrica di prossimità. Se non si conoscono nodi con nodeId adeguato, allora l'entry della tabella di routing viene lasciata vuota. La distribuzione uniforme dei nodeId assicura una popolazione equa dello spazio dei nodeId; quindi, in media, solo $\lceil \log_{2^b} N \rceil$ righe della tabella di routing sono popolate.

La scelta di b implica un trade-off tra la dimensione della porzione popolata della tabella di routing (approssimativamente $\lceil \log_{2^b} N \rceil \times \left(2^b - 1\right)$ entry) ed il numero massimo di hop richiesti per effettuare il routing tra qualsiasi coppia di nodi ($\lceil \log_{2^b} N \rceil$). Con un valore di $b = 4$ e 10^6 nodi, una tabella di routing contiene in media 75 entry ed il numero medio di hop è 5, mentre con 10^9 nodi, la tabella di routing contiene in media 105 entry ed il numero medio di hop è 7.

Il neighborhood set M contiene i nodeId e gli indirizzi IP degli $|L|$ nodi più vicini (secondo la metrica di prossimità) al nodo corrente. Il neighborhood set viene normalmente usato nel routing dei messaggi; è utile nel mantenere le proprietà di località. Il leaf set L è l'insieme dei nodi con i $|L|/2$ nodeId numericamente più vicini più grandi, ed i $|L|/2$ nodeId numericamente più vicini più piccoli, relativamente al nodeId del nodo corrente. Il leaf set viene usato nel routing dei messaggi, come descritto di seguito. Valori tipici per $|L|$ ed $|M|$ sono 2^b oppure 2×2^b.

NodeId 10233102

Leaf set

SMALLER		LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table

-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Figura 2.1.: Stato di un ipotetico nodo Pastry con nodeID 10233102, $b = 2$, ed $l = 8$. Tutti i numeri sono in base 4. La riga in cima alla tabella di routing è la riga zero. Le celle in grigio in ciascuna riga della tabella di routing mostra la cifra corrispondente del nodeId del nodo corrente. Le nodeId di ciascuna entry sono state spezzate per mostrare il prefisso in comune con 10233102 - prossima cifra - resto del nodeId. Gli indirizzi IP associati non sono mostrati.

2.1.4. Routing

L'algoritmo 2.1 mostra in pseudo-codice il meccanismo di routing di Pastry. La procedura viene eseguita quando un messaggio con chiave D arriva al nodo con nodeId A. Prima definiamo alcune notazioni.

R_l^i: l'entry nella tabella di routing R alla colonna i, $0 \leq i < 2^b$, e riga l, $0 \leq l < \lfloor 128/b \rfloor$.

L_i: l'i-esimo nodeId più vicino nel leaf set L, $-\lfloor |L|/2 \rfloor \leq i \leq \lfloor |L|/2 \rfloor$, dove l'indice positivo/negativo indica rispettivamente i nodeId più grandi/piccoli del nodeId corrente.

D_l: valore dell'l-esima cifra della chiave D.

$shl(A, B)$: la lunghezza del prefisso comune delle chiavi A e B in cifre.

Dato un messaggio, il primo nodo controlla se la chiave cade nel range di nodeId coperti dal suo leaf set (riga 1). Se è così, il messaggio viene inoltrato direttamente al nodo di destinazione, ossia il nodo del leaf set il cui nodeId è il più vicino numericamente alla chiave (possibilmente anche il nodo corrente) (riga 3).

Se la chiave non è coperta dal leaf set, allora viene utilizzata la tabella di routing e il messaggio viene inoltrato al nodo che ha in comune un prefisso lungo almeno una cifra in più (righe 7-9). In certi casi è possibile che l'entry corrispondente nella tabella di routing sia vuota o che il nodo puntato non sia raggiungibile (righe 12-15), nel qual caso il messaggio viene inoltrato ad un nodo che ha in comune un prefisso lungo almeno quanto quello attuale, ed è numericamente più vicino alla chiave del nodeId corrente. Tale nodo deve trovarsi nel leaf set a meno che il messaggio sia già giunto al nodo col nodeId numericamente più vicino. E, salvo che $\lfloor |L|/2 \rfloor$ nodi adiacenti nel leaf set falliscano simultaneamente, almeno uno di questi nodi deve essere funzionante.

Algoritmo 2.1 Pseudo-codice dell'algoritmo di routing di Pastry

```
(1)   if (L_{-⌊|L|/2⌋} ≤ D ≤ L_{⌊|L|/2⌋}) {
(2)       // se D è nel range del nostro leaf set
(3)       inoltra a L_i tale che |D − L_i| è minimo;
(4)   }
(5)   else {
(6)       // usa la tabella di routing
(7)       sia l = shl(D, A);
(8)       if (R_l^{D_l} ≠ null) {
(9)           inoltra a R_l^{D_l};
(10)      }
(11)      else {
(12)          // caso raro
(13)          inoltra a T ∈ L ∪ R ∪ M, tale che
(14)              shl(T, D) ≥ l,
(15)              |T − D| < |A − D|;
(16)      }
(17)  }
```

Questa semplice procedura di routing converge sempre, perché ad ogni passo porta il messaggio ad un nodo tale che (1) ha in comune con la chiave un prefisso più lungo del nodo corrente, oppure (2) ha in comune un prefisso ugualmente lungo, ma che è numericamente più vicino alla chiave di quanto non lo sia il nodo corrente.

2.1.4.1. Performance del Routing

Si può far vedere che il numero atteso di passi di routing è $\lceil \log_{2^b} N \rceil$, supponendo di avere tabelle di routing accurate e che non ci siano fallimenti recenti nei nodi. In breve, consideriamo i tre casi nella procedura di routing. Se un messaggio viene inoltrato usando la tabella di routing (righe 7-9), allora l'insieme dei nodi, i cui nodeId hanno un prefisso comune con la chiave più lungo, si riduce di un fattore di 2^b ad ogni passo, il che significa che la destinazione viene raggiunta in $\lceil \log_{2^b} N \rceil$ passi. Se la chiave è nel range del leaf set (riga 3), allora la destinazione dista al più un hop.

Il terzo caso avviene quando la chiave non è coperta dal leaf set (è ancora lontano più di un hop dalla destinazione), però non ci sono le corrispondenti entry nella tabella di routing. Supponendo di avere tabelle di routing accurate e di non avere fallimenti recenti, questo significa che non esiste un nodo con il prefisso appropriato (righe 12-15). La probabilità che avvenga questo caso dipende da $|L|$. Le analisi dimostrano che con $|L| = 2^b$ e $|L| = 2 \times 2^b$ la probabilità che avvenga il suddetto caso è meno di 0.02 e 0.006 rispettivamente. Anche quando questo caso avviene, ha come effetto, con alta probabilità, di incrementare di un solo passo il processo di routing.

Nel caso in cui ci siano molti fallimenti di nodi simultanei, il numero di passi per il routing del messaggio piò essere al peggio lineare in N, mentre i nodi stanno aggiornando i loro stati. Questo è un upper-bound molto lasco; in pratica, le prestazioni del routing degradano gradualmente con l'aumento dei fallimenti recenti. Eventualmente, la ricezione del messaggio è garantita a meno che $\lfloor |L|/2 \rfloor$ nodi con nodeId consecutivi non falliscano simultaneamente. Data l'attesa diversità dei nodi con nodeId adiacenti, e con una scelta ragionevole di $|L|$ (per esempio 2^b), la probabilità di un simile evento può essere resa molto piccola.

2.1.5. API di Pastry

Di seguito viene definite la Application Programming Interface (API) di Pastry.

`nodeId = pastryInit(Credentials, Application)` permette al nodo locale di unirsi ad una rete Pastry esistente (o di crearne una nuova), inizializza tutto lo stato rilevante, e restituisce il nodeId del nodo corrente. Le credenziali specifiche dell'applicazione contengono informazioni necessarie per autenticare il nodo. L'argomento `Application` è un handle all'oggetto applicazione che fornisce al nodo Pastry le procedure da invocare quando avvengono certi eventi (per esempio quando arriva un messaggio).

`route(msg, key)` permette a Pastry di eseguire il routing del messaggio al nodo con nodeId numericamente più vicino alla chiave tra i nodi attualmente attivi nella rete.

Le applicazioni costruite sopra lo strato Pastry devono implementare le seguenti operazioni:

`deliver(msg, key)` chiamato da Pastry quando il nodo riceve un messaggio ed il nodeId del nodo corrente è effettivamente quello numericamente più vicino alla chiave, tra tutti i nodeId dei nodi attivi.

`forward(msg, key, nextId)` chiamato da Pastry appena prima di inoltrare un messaggio al nodo con `nodeId = nextId`. Il livello applicativo può decidere di cambiare il contenuto del messaggio o il valore di `nextId`. Impostare `nextId` a `NULL` termina il messaggio al nodo locale.

`newLeafs(leafSet)` chiamato da Pastry quando ci sono cambiamenti nel leaf set del nodo corrente. Questo fornisce al livello applicativo la possibilità di settare impostazioni specifiche dell'applicazione in base al leaf set.

2.1.6. Auto-organizzazione e adattamento

In questa sezione vengono descritti i protocolli di Pastry per la gestione degli arrivi e degli allontanamenti dei nodi nella rete Pastry. Iniziamo con l'arrivo di un nuovo nodo che si collega al sistema.

2.1.6.1. Arrivo di un nodo

Quando arriva un nuovo nodo, deve innanzitutto inizializzare le sue tabelle di stato, e poi deve informare gli altri nodi della sua presenza. Assumiamo che il nuovo nodo conosca inizialmente un nodo Pastry vicino, il nodo A, secondo la metrica di prossimità, che è già parte del sistema. Tale nodo lo si può trovare automaticamente, per esempio, usando un multicast IP "expanding ring", oppure può essere ottenuto dall'amministratore di sistema tramite altri canali.

Supponiamo che il nodeId del nuovo nodo sia X. (L'assegnazione dei nodeId viene fatta secondo criteri specifici dell'applicazione; di solito si calcola il hash SHA-1 (vedi B.3 a pagina 72) dell'indirizzo IP del nodo, o della sua chiave pubblica). Il nodo X chiede ad A di inoltrare un messaggio speciale di "join" con chiave uguale a X. Come qualsiasi altro messaggio, Pastry inoltra il messaggio di join al nodo esistente Z il cui id è numericamente il più vicino ad X.

In risposta alla richiesta di "join", i nodi A, Z, e tutti gli altri nodi incontrati nel cammino da A a Z, inviano le loro tabelle di stato a X. In nuovo nodo X ispeziona queste informazioni, potrebbe richiedere informazioni di stato da altri nodi, e poi inizializza le proprie tabelle di stato, usando la procedura descritta qua sotto. Infine, X informa i nodi che necessitano di sapere del suo arrivo. Questa procedura assicura che X possa inizializzare il proprio stato con valori appropriati, e che vengano aggiornati gli stati di tutti i nodi affetti.

Dato che il nodo A viene assunto essere nelle vicinanze del nuovo nodo X, il neighborhood set di A viene usato per inizializzare il neighborhood set di X. In più, Z ha

il nodeId esistente più vicino ad X, quindi il suo leaf set sarà la base del leaf set di X. In seguito, consideriamo la tabella di routing a partire dalla riga zero. Consideriamo il caso più generale, dove i nodeId di A e X non hanno prefissi in comune. Sia A_i la riga della tabella di routing di A corrispondente al livello i. Quindi A_0 contiene il valore appropriato per X_0. Gli altri livelli della tabella di routing di A non possono servire a X, poiché gli id di A e X non hanno prefisso in comune.

Comunque, il valore appropriato per X_1 può essere ottenuto da B_1, dove B è il primo nodo incontrato sul cammino da A a Z. Per chiarire questo punto, si osservi che le entry B_1 e X_1 hanno lo stesso prefisso, perché X e B hanno la stessa cifra iniziale nei loro nodeId. In modo simile, X ottiene l'entry per X_2 dal nodo C, il prossimo nodo incontrato, dopo B, nel cammino da A a Z, e così via.

Infine, X trasmette una copia del suo stato risultante ad ogni nodo nei suoi neighborhood set, leaf set e tabella di routing. Quei nodi aggiornano a turno i loro stati in base alle informazioni ricevute. Si può dimostrare che a questo punto, il nodo X è in grado di inviare e ricevere messaggi, e di partecipare nella rete Pastry. Il costo totale per il join di un nodo, in termini di numero di messaggi scambiati, è $O(\log_{2^b} N)$.

Pastry usa un approccio ottimistico per controllare gli arrivi e gli allontanamenti concorrenti dei nodi. Dato che l'arrivo/allontanamento di un nodo ha effetto solo un piccolo numero di nodi esistenti nel sistema, la concorrenza è rara e un approccio ottimistico è appropriato. In breve, quando un nodo A fornice informazioni di stato al nodo B, allega un *timestamp* al messaggio. B aggiusta il suo stato in base a queste informazioni ed invia eventualmente un messaggio di aggiornamento (update) ad A (es., informando A del suo arrivo). B allega il *timestamp* originale, che permette ad A di controllare se il suo stato è cambiato da allora. Nel caso in cui il suo stato sia cambiato, risponderà con lo stato aggiornato, e B riavvierà le sue operazioni.

2.1.6.2. Allontanamento di un nodo

I nodi nella rete Pastry possono malfunzionare o allontanarsi senza preavviso. Di seguito discuteremo come vengono affrontati questi eventi dalla rete Pastry. Un nodo Pastry viene considerato fallito (failed) quando i suoi vicini immediati nello spazio dei nodeId non possono più comunicare con quest'ultimo.

Sia X un nodo fallito, e sia A un nodo che contiene X nel suo leaf set. Se il nodeId di A è maggiore del nodeId di X, allora A richiede il leaf set del nodo con nodeId minore tra quelli attivi nel suo leaf set. Altrimenti, se il nodeId di A è minore del nodeId di X, allora A richiede il leaf set del nodo con nodeId maggiore tra quelli attivi nel suo leaf set. Per esempio, se $nodeId_X < nodeId_A$, e cioè se è fallita una entry L_i del leaf set di A, con $-\lfloor |L|/2 \rfloor < i < 0$, allora A richiederà il leaf set di $L_{-\lfloor |L|/2 \rfloor}$. Questo leaf set si sovrappone parzialmente al leaf set L del nodo attuale, e contiene nodi con nodeId non attualmente presenti in L. Tra questi nuovi nodi viene scelto quello corretto da inserire in L, verificando prima che il nodo sia attualmente attivo contattandolo. Questa procedura garantisce che ciascun nodo ripari il proprio leaf set in modo pigro (lazy) a meno che $\lfloor |L|/2 \rfloor$ nodi con nodeId adiacenti non falliscano contemporaneamente. Questo è abbastanza improbabile anche con valori modesti di$|L|$.

Il fallimento di un nodo che appare nella routing table di un altro nodo viene rilevato quando quel nodo tenta di contattare il nodo fallito e non vi è risposta. Come spiegato nella sezione 2.1.4 a pagina 16, questo evento normalmente non ritarda il routing del messaggio, dato che il messaggio può essere inoltrato ad un altro nodo. Bisogna però trovare una entry di ricambio per preservare l'integrità della tabella di routing.

Per riparare una entry R_l^d fallita della tabella di routing, si deve contattare il primo nodo riferito da un'altra entry R_l^i, $i \neq d$ della stessa riga, e bisogna chiedere l'entry del nodo R_l^d. Nell'eventualità che nessuna delle entry della riga l abbia un puntatore ad

un nodo attivo con il prefisso appropriato, il nodo contatterà una entry R^i_{l+1}, $i \neq d$, lanciando a questo punto una ricerca più ampia. Questa procedura ha molte probabilità di trovare un nodo adatto se ne esiste uno.

Il neighborhood set non viene di norma usato nel routing dei messaggi, però è comunque fondamentale tenerlo aggiornato, poiché il set gioca un ruolo importante nello scambio di informazioni sui nodi vicini. A tale scopo, un nodo cerca di contattare periodicamente ciascun membro del proprio neighborhood set per vedere se essi sono ancora attivi. Se un membro non risponde, il nodo chiede agli altri membri del suo neighborhood set le loro tabelle di neighborhood, controlla le distanze di ciascun nuovo nodo scoperto nelle tabelle ricevute, ed in base a questo aggiorna il proprio neighborhood set.

2.1.7. Località

Nelle sezioni precedenti abbiamo visto le proprietà basilari del routing in Pastry e abbiamo discusso le sue performance in termini di numero atteso di hop di routing e di messaggi scambiati come parte di una operazione di join. Questa sezione è centrata su un altro aspetto della performance del routing in Pastry, ossia le sue proprietà riguardanti la località. Mostreremo che il cammino scelto da un messaggio ha un'alta probabilità di essere "buono" secondo la metrica di prossimità scelta.

In Pastry, la nozione di prossimità nella rete è basata su una metrica scalare di prossimità, come il numero di hop nel routing IP o la distanza geografica. Si assume che il livello applicativo fornisca una funzione che permetta ad ogni nodo Pastry di determinare la "distanza" da un nodo con un dato indirizzo IP. Un nodo con un valore di distanza piccolo si assume essere più desiderabile. Ci si aspetta che un'applicazione implementi questa funzione in base alla sua scelta riguardo alla metrica di prossimità, usando servizi di rete come *traceroute* o le *mappe di sottorete di Internet*, e tecniche di caching ed approssimazione per minimizzare l'overhead.

In questa descrizione, assumeremo che lo spazio di prossimità definite dalla metrica di prossimità scelta sia Euclideo; che significa che vale la disuguaglianza triangolare[1] per le distanze tra nodi Pastry. Questa ipotesi non è veritiera in pratica per alcune metriche di prossimità, come il numero di hop necessari per effettuare il routing IP da un nodo all'altro. Se l'uguaglianza triangolare non sussiste, il routing di base di Pastry non ne risente; però, le sue proprietà di località potrebbero soffrirne.

Di seguito viene descritto come viene estesa la procedura precedente per la gestione degli arrivi dei nodi, con un'euristica che assicura che le entry delle tabelle di routing vengano scelte per fornire buone proprietà di località.

2.1.7.1. Località nella tabella di routing

Nella sezione 2.1.6 a pagina 18 abbiamo descritto il procedimento col quale un nuovo nodo che si college alla rete inizializza la propria tabella di routing. Supponiamo come prima che il nuovo nodo X chieda ad un nodo esistente A di inoltrare un messaggio di join usando X come chiave. Il messaggio segue un cammino percorrendo i nodi A, B, ecc., ed eventualmente arriva al nodo Z, il quale è il nodo attivo col nodeId numericamente più vicino ad X. Il nodo X inizializza la sua tabella di routing ottenendo la i-esima riga della sua tabella di routing dallo i-esimo nodo incontrato durante il cammino da A a Z.

La proprietà che vogliamo mantenere consiste nel volere che tutte le entry della tabella di routing puntino a nodi vicini al nodo corrente, secondo la metrica di prossimità, tra tutti i nodi attivi con un prefisso adatto per l'entry. Assumiamo che questa proprietà

[1] La disuguaglianza triangolare è una proprietà matematica. Essa è una delle proprietà caratterizzanti una distanza in uno spazio metrico. Formalmente tale proprietà afferma che, dato lo spazio metrico (X, d), allora: $\forall x, y, z \in X \ : \ d(x, y) \leq d(x, z) + d(z, y)$.

sia valida per tutti i nodi che partecipano alla rete prima dell'arrivo di X, e dimostriamo che possiamo mantenere la proprietà anche quando X si unisce alla rete.

Per prima cosa, richiediamo che il nodo A sia vicino ad X, secondo la metrica di prossimità. Dato che le entry alla riga zero della tabella di routing di A sono vicine ad A, A è vicino ad X, ed assumiamo che la disuguaglianza triangolare sia valida nello spazio delle prossimità, segue che le entry sono relativamente vicine ad X. Quindi, la proprietà desiderata è preservata. Analogamente, anche il procedimento col quale si ottiene il neighborhood set di X da A è appropriato.

Consideriamo ora la riga uno della tabella di routing di X, la quale è stata ottenuta dal nodo B. Le entry in questa riga sono vicine a B, comunque, non è chiaro come B possa essere vicino ad X. Intuitivamente, sembrerebbe che prendere la riga uno della tabella di routing di X dal nodo B non preservi la proprietà desiderata, dato che le entry di B sono sì, vicine a B, ma non necessariamente ad X. In realtà, le entry tendono ad essere ragionevolmente vicine ad X. Ricordiamo che le entry in ciascuna riga successiva vengono scelte da un insieme con dimensione che decresce esponenzialmente. Quindi, la distanza attesa tra B e le sue entry della riga uno (B_1) è molto più grande della distanza percorsa dal nodo A al nodo B. Come risultato, B_1 è una scelta ragionevole per X_1. Lo stesso ragionamento si applica per ogni livello e passo di routing successivo, come descritto nella figura 2.2.

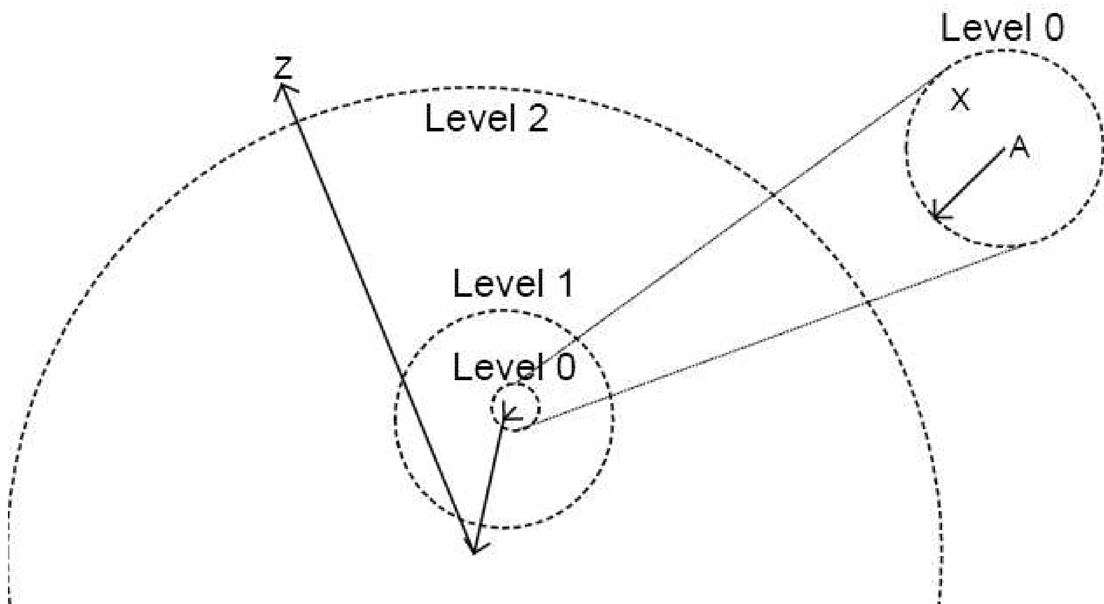

Figura 2.2.: Distanza del passo di routing a confronto con le distanze dei rappresentanti ad ogni livello (in base a dati sperimentali). Il cerchio intorno all'n-esimo nodo nel cammino da A a Z indica la distanza media dei rappresentanti del nodo al livello n. Da notare che X si trova all'interno di ciascuno dei cerchi.

Dopo che X ha inizializzato il suo stato secondo la procedura descritta, la sua tabella di routing e il suo neighborhood set approssimano la proprietà desiderata. Comunque, la qualità di questa approssimazione deve essere migliorata per impedire errori a cascata che potrebbero portare ad una cattiva proprietà di località nel routing. A questo proposito, esiste una seconda fase nella quale X richiede gli stati da ciascuno dei nodi nella sua tabella di routing e dal suo neighborhood set. Dopodiché confronta le distanze delle entry corrispondenti trovate in queste tabelle ed aggiorna il proprio stato con gli eventuali nodi più vicini che riesce a trovare. Il neighborhood set contribuisce con

informazioni preziose in questo procedimento, perché mantiene e propaga informazioni sulla vicinanza dei nodi indipendentemente dai prefissi dei nodeId.

Intuitivamente, uno sguardo alla figura 2.2 nella pagina precedente mette luce sul perché utilizzare gli stati dei nodi presenti nelle tabelle di routing e di neighborhood nella fase uno, fornisce buone entry rappresentative per X. I cerchi indicano la distanza media della entry da ciascuno dei nodi incontrati lungo il percorso, corrispondenti alle righe nella tabella di routing. Si osservi che X si trova all'interno di ciascun cerchio, anche se fuori centro. Nella seconda fase, X ottiene lo stato delle entry scoperte nella fase uno, che si trovano ad una distanza media pari al perimetro di ogni rispettivo cerchio. Questi stati devono includere le entry appropriate per X, ma che non sono state scoperte da X nella prima fase, a causa della sua posizione decentrata.

Segue la discussione su come la località nelle tabelle di routing di Pastry influisce sui cammini dei messaggi.

2.1.7.2. Località nel routing

Le entry nelle tabelle di routing di ciascun nodo Pastry vengono scelte in modo da essere vicine al nodo corrente, secondo la metrica di prossimità, tra tutti i nodi con il prefisso del nodeId desiderato. Come risultato, in ciascun passo di routing, un messaggio viene inoltrato verso un nodo relativamente vicino, con un nodeId che ha un prefisso comune più lungo o che è numericamente più vicino alla chiave di quanto non lo sia il nodo locale. Quindi ad ogni passo il messaggio è sempre più vicino alla destinazione nello spazio dei nodeId, mentre percorre la distanza minima nello spazio delle prossimità.

Poiché si usano solo informazioni locali, Pastry minimizza la distanza del prossimo passo di routing senza conoscenza globale. Questo procedimento chiaramente non garantisce di trovare il cammino minimo dalla sorgente alla destinazione; comunque, produce cammini relativamente buoni. Due fatti sono rilevanti a questa affermazione. Primo, dato un messaggio inoltrato dal nodo A al nodo B che dista d da A, il messaggio non può successivamente essere inoltrato ad un nodo che si trova a distanza minore di d da A. Questo deriva direttamente dalla procedura di routing, assumendo di avere tabelle di routing accurate.

Secondo, la distanza attesa, che il messaggio percorre durante ciascun passo di routing successivo, cresce esponenzialmente. Per vedere questo si osservi che una entry nella tabella di routing alla riga l viene scelta da un insieme di nodi di taglia $N/2^{bl}$. Questo significa che entry su righe successive vengono scelte da un numero di nodi esponenzialmente decrescente. Data la distribuzione aleatoria e uniforme dei nodeId sulla rete, questo significa che la distanza attesa della entry più vicina in ogni riga successiva cresce esponenzialmente.

Insieme, questi due fatti implicano che, anche se non si può garantire che la distanza di un messaggio dalla sua sorgente cresca in maniera monotòna ad ogni passo, il messaggio comunque tende a percorrere salti via via più grandi, senza possibilità di andare ad un nodo, che si trova entro un raggio d_i da un nodo i, già incontrato durante il cammino, dove d_i è la distanza percorsa dal passo di routing preso uscendo dal nodo i. Quindi, il messaggio non può andare altrove se non verso la sua destinazione. La figura 2.3 nella pagina successiva illustra questo effetto.

2.1.7.3. Localizzare il più vicino tra *k* nodi

Alcune applicazioni peer-to-peer che utilizzano Pastry replicano le informazioni sui k nodi Pastry con nodeId, nello spazio dei nodeId di Pastry, numericamente più vicini alla chiave. PAST, per esempio, replica i file in modo da assicurare un alto grado di disponibilità anche a fronte di eventuali fallimenti dei nodi. Pastry esegue naturalmente

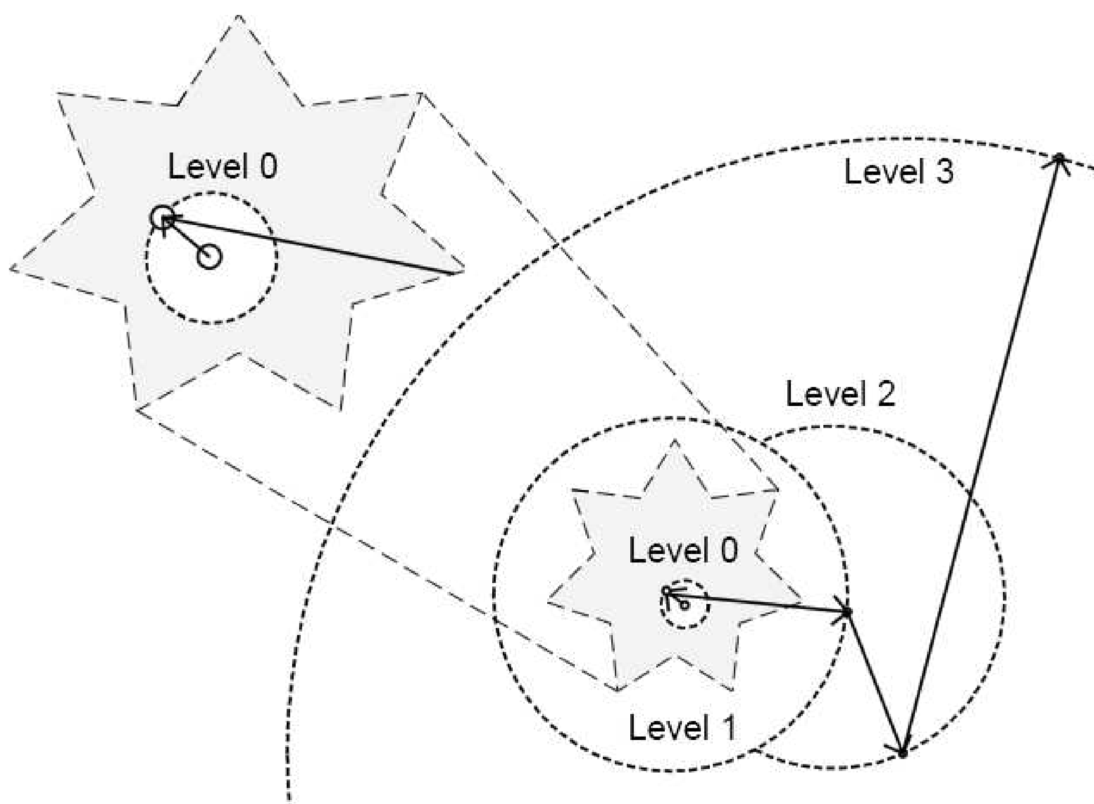

Figura 2.3.: Esempio di traiettoria di un tipico messaggio nella rete Pastry, basato su dati sperimentali. Il messaggio non piò ritornare in uno dei cerchi rappresentanti le distanze percorse in ciascun passo di routing uscendo dai nodi intermedi. Anche se il messaggio può tornare parzialmente "indietro" durante i suoi passi iniziali, le distanze crescenti esponenzialmente che percorre in ogni passo lo obbligano a muoversi velocemente verso la sua destinazione.

il routing dei messaggi con la chiave data al nodo con nodeId numericamente più vicino a quest'ultima, assicurando che il messaggio raggiunga uno dei k nodi finché almeno uno di essi è ancora attivo.

In più, le proprietà di Pastry fanno sì che, nel tragitto da un client al nodo numericamente più vicino, il messaggio raggiunga prima un nodo vicino al client, in termini di prossimità metrica, tra i k nodi numericamente più vicini. Questo è utile in applicazioni come PAST, perché ottenere un file da un nodo vicino minimizza la latenza del client ed il carico della rete. In più, si osservi che data l'assegnazione casuale dei nodeId, i nodi con nodeId adiacenti sono probabilmente molto dispersi nella rete. Quindi è importante dirigere una query di lookup verso un nodo che si trova relativamente vicino al client.

Ricordiamo che Pastry effettua il routing dei messaggi verso il nodo con il nodeId più vicino alla chiave, mentre cerca di compiere la minore distanza possibile in ciascun passo. Quindi, tra i k nodi numericamente più vicini ad una chiave data, il messaggio tende a raggiungere per primo un nodo vicino al client. Ovviamente, questo procedimento è un'approssimazione del routing al nodo più vicino. Primo, come discusso prima, Pastry prende solo decisioni di routing locali, minimizzando la distanza percorsa al passo successivo senza un'idea globale della direzione da seguire. Secondo, dato che Pastry esegue il routing principalmente in base ai prefissi dei nodeId, potrebbe perdersi qualche nodo vicino con un prefisso diverso da quello della chiave. Nel caso peggiore sono $k/2 - 1$ le repliche salvate su nodi i cui nodeId differiscono dalla chiave nei loro

domini al livello zero. Come risultato, Pastry inoltrerà inizialmente verso il più vicino tra i $k/2+1$ nodi rimanenti.

Pastry usa una euristica per superare il problema del mismatch del prefisso che abbiamo appena descritto. L'euristica si basa sulla stima della densità di nodeId nello spazio dei nodeId usando informazioni locali. In base a questa stima, l'euristica capisce quando un messaggio si avvicina all'insieme dei k nodi numericamente più vicini, e poi passa al routing in base all'indirizzo numericamente più vicino per trovare la replica più vicina. Risultati sperimentali dimostrano che Pastry è in grado di trovare il nodo più vicino in più del 75% di tutte le query, ed uno dei due nodi più vicini in più del 91% di tutte le query.

2.1.8. Fallimenti arbitrari dei nodi e partizionamento della rete

Fino ad ora abbiamo supposto che i nodi in Pastry fallissero silenziosamente. Qui, discuteremo brevemente come una rete Pastry può gestire nodi che falliscono arbitrariamente, quando un nodo anche se fallito continua a rispondere, però si comporta in modo non corretto o anche malevolo. Lo schema di routing di Pastry che abbiamo mostrato finora è troppo deterministico. Quindi, è vulnerabile a nodi malevoli o falliti che accettano i messaggi ma non li inoltrano. Query ripetute possono dunque fallire ogni volta, dato che percorrono sempre lo stesso percorso.

In applicazioni dove devono essere tollerati i fallimenti arbitrari dei nodi, il routing può essere randomizzato. Ricordiamo che, per evitare i cicli infiniti nel routing, un messaggio deve sempre essere inoltrato ad un nodo che ha un prefisso comune con la destinazione più lungo, o ha la stessa lunghezza di prefisso del nodo corrente ma è numericamente più vicino, nello spazio dei nodeId, del nodo corrente. La scelta tra diversi nodi che soddisfano questi criteri può essere resa aleatoria. In pratica, la distribuzione di probabilità dovrebbe tendere verso la scelta migliore per assicurare un basso delay medio. Nel caso di nodi malevoli o falliti durante il cammino, la query potrebbe dover essere ripetuta diverse volte dal client, finché non viene scelto un cammino che evita il nodo cattivo. Inoltre, anche i protocolli per il join e l'allontanamento dei nodi possono essere estesi per tollerare i nodi che si comportano male.

Un'altra sfida è composta dalle anomalie nello routing IP su Internet, che causano la non-raggiungibilità di alcuni host da certi IP ma non da altri. Il routing di Pastry tollera questo tipo di anomalie; i nodi Pastry si considerano attivi e rimangono raggiungibili nella rete overlay finché sono in grado di comunicare con i loro vicini più immediati nello spazio dei nodeId. Però, il protocollo di auto-organizzazione di Pastry potrebbe la creazione di overlay Pastry multiple ed isolate durante periodi di problemi col routing IP. Dato che Pastry si affida quasi esclusivamente allo scambio di informazioni dentro la rete overlay per auto-organizzarsi, tali overlay isolate potrebbero sopravvivere anche dopo che la piena connettività IP è stata ripristinata.

Una soluzione a questo problema consiste nell'utilizzo del multicast IP. I nodi Pastry possono periodicamente eseguire un multicast di espansione dell'anello per cercare altri nodi Pastry nelle loro vicinanze. Se esistono overlay Pastry isolate, queste verranno eventualmente scoperte, e reintegrate. Per minimizzarne il costo, questa procedura può essere eseguita in modo aleatorio ma non frequentemente dai nodi Pastry, con solo un limitato range di hop IP dal nodo, e solo se non è stata avviata nessuna ricerca da un nodo vicino di recente. Come beneficio aggiuntivo, il risultato di questa ricerca può essere sfruttato per migliorare la qualità delle tabelle di routing.

2.2. PAST

Di seguito descriveremo PAST[17, 18], un sistema di storage distribuito a larga scala su Internet, che fornisce scalabilità, alti livelli di disponibilità e sicurezza. PAST è un'applicazione Internet peer-to-peer completamente auto-organizzabile. I nodi PAST servono da access point per i client, partecipano nel routing delle richieste dei client, e contribuiscono al sistema di storage. I nodi non sono fidati, possono collegarsi al sistema in qualsiasi istante, come lo possono lasciare senza avvertimento. Comunque, il sistema è in grado di fornire garanzie, accesso efficiente allo storage, bilanciamento del carico e scalabilità.

2.2.1. Introduzione

Tra gli aspetti più interessanti del design di PAST troviamo (1) lo schema di localizzazione e routing di Pastry, il quale esegue il routing delle richieste dei client tra i nodi Pastry in modo affidabile ed efficiente, ha buone proprietà di località sulla rete e risolve automaticamente i fallimenti e le aggiunte di nuovi nodi; (2) l'utilizzo della randomizzazione per assicurare la diversità nell'insieme dei nodi che contengono la replica di un file e di assicurare il bilanciamento del carico; e (3) l'utilizzo opzionale di smartcard, le quali sono mantenute dagli utenti e rilasciate da una terza parte chiamata broker: le smartcard supportano un sistema di quote che bilancia le risorse e le richieste di storage nel sistema.

Il sistema PAST è composto da nodi connessi ad Internet, dove ciascun nodo è in grado di iniziare e inoltrare le richieste dei client per inserire o reperire file. Opzionalmente, i nodi possono anche contribuire con lo storage al sistema. I nodi PAST formano una rete overlay auto-organizzata. I file inseriti vengono replicati su più nodi per assicurare la persistenza e la disponibilità. Con altissima probabilità, l'insieme dei nodi su cui viene replicato un file è diversificato in termini di posizione geografica, appartenenza, amministrazione, connettività alla rete, leggi vigenti, ecc. Copie ulteriori di file popolari possono essere salvate nella cache di qualsiasi nodo PAST per bilanciare il carico delle query.

Un'utilità di storage come PAST attrae per diverse ragioni. Primo, sfrutta la moltitudine e la diversità (posizione geografica, appartenenza, amministrazione, connettività alla rete, leggi vigenti, ecc.) dei nodi su Internet per realizzare una persistenza forte e fornire alti livelli di disponibilità. Questo rende non necessaria la protezione dei dati di backup o archiviati nei mezzi di storage; similmente, rende non necessario il mirroring esplicito dei dati condivisi per questioni di prestazioni e disponibilità. Un'utilità di storage globale facilità la condivisione dello storage e della banda, permettendo quindi a un gruppo di nodi di salvare o pubblicare contenuti che eccedono le capacità di ciascun singolo nodo preso individualmente.

Mentre PAST offre servizi di storage persistente, le sue semantiche di accesso differiscono da quelle di un file system convenzionale. I file salvati in PAST vengono associati ad un fileId quasi-unico che viene generato al momento dell'inserimento del file nel sistema PAST. Quindi, i file salvati in PAST sono immutabili dato che un file non può essere inserito più volte con lo stesso fileId. I file possono essere condivisi a discrezione del proprietario, il quale decide di distribuire il fileId (potenzialmente in modo anonimo) e, se necessario, una chiave di decriptazione. Il proprietario di un file può anche reclamare lo storage ad esso associato, senza garantire però la completa ed immediata cancellazione del file. Queste semantiche più deboli evitano protocolli di agreement tra i nodi che mantengono un dato file.

Lo schema di routing di Pastry assicura che le richieste dei client vengano inoltrare correttamente ai nodi. Le richieste dei client di ottenere un file vengono inoltrare ad un

nodo che si trova vicino sulla rete al client che ha inviato la richiesta. Il numero di nodi PAST incontrati eseguendo il routing della richiesta di un client è al più logaritmico nel numero totale di nodi PAST che compongono il sistema sotto condizioni normali di operatività. Uno schema di gestione dello storage in PAST assicura che l'utilizzazione globale dello storage del sistema possa arrivare vicino al 100%, a dispetto della mancanza di un controllo centralizzato e della grande varietà nelle dimensioni dei file o nelle capacità individuali dei nodi. In un sistema di storage centralizzato, è richiesto un ulteriore meccanismo per assicurare il bilanciamento delle richieste e delle offerte di storage. Su questo fronte, PAST include un sistema sicuro di quote. In casi semplici, agli utenti vengono assegnate quote fisse, oppure gli viene permesso di usare tanto storage quanto quello che essi stessi forniscono. Opzionalmente, alcune organizzazioni chiamate broker possono scambiare lo storage e rilasciare smartcard agli utenti, le quali controllano quanto storage l'utente deve contribuire o può utilizzare. Il broker non è direttamente partecipe delle operazioni nella rete PAST, e la sua conoscenza del sistema è limitata al numero di smartcard che ha messo in circolazione, le loro quote e le date di scadenza.

Un altro problema dei sistemi peer-to-peer, ed in particolare dei sistemi di storage e di file-sharing, è la privacy e l'anonimato. Un utente che fornisce spazio per lo storage potrebbe non voler rischiare di essere perseguito legalmente per i contenuti salvati, ed i client che inseriscono e richiedono file potrebbero non voler dichiarare la loro identità. I client di PAST e gli utenti che forniscono storage non devono per forza fidarsi gli uni degli altri, e possono anche fidarsi in parte dei broker. In particolare, tutti i nodi si fidano dei broker per facilitare le operazioni di una rete PAST sicura bilanciando le risorse e le richieste di storage tramite un uso responsabile del sistema di quote. Dall'altra parte, gli utenti non devono rivelare ai broker (o nessun altro) la loro identità, i file che stanno richiedendo o inserendo. Ogni utente detiene uno pseudonimo inizialmente non-collegabile (initially unlinkable pseudonym)[22] sotto forma di chiave pubblica. Lo pseudonimo non è facilmente collegabile all'identità dell'utente, a meno che l'utente stesso non lo riveli volontariamente. Se lo si desidera, si possono usare più pseudonimi diversi dallo stesso utente per nascondere il fatto che, certe operazioni, siano state svolte da un unico individuo.

2.2.2. Design di PAST

PAST è composto da nodi connessi ad Internet. Ciascuno di essi si può comportare come nodo di storage e/o come access point per i client. A ciascun nodo viene assegnato un nodeId da 128 bit, che deriva dal hash crittografico della chiave pubblica del nodo stesso. Ad ogni file che inserito in PAST viene assegnato un fileId da 160 bit, corrispondente al hash crittografico del nome testuale del file, della chiave pubblica del proprietario ed un sale casuale[2]. Prima che un file venga inserito, viene generato un certificato per quel file, contenente il fileId, il suo fattore di replicazione k, la data di inserimento ed un hash crittografico del contenuto del file. Il certificato del file viene firmato dal proprietario del file.

Quando un file viene inserito in PAST, Pastry esegue il routing del file ai k nodi i cui identificatori sono i più vicini numericamente ai 128 bit più significativi dell'identificatore del file (fileId). Ognuno di questi nodi mantiene una copia del file. Il fattore di replicazione k dipende dai requisiti di persistenza e disponibilità del file e può variare da un file all'altro. Una richiesta di lookup per un file viene inoltrata verso il nodo attivo il cui nodeId è numericamente il più vicino al fileId richiesto.

[2] In crittografia, un sale è una sequenza casuale di bit utilizzata assieme ad una password come input a una funzione unidirezionale, di solito una funzione hash, il cui output è conservato al posto della sola password, e può essere usato per autenticare gli utenti.

Questa procedura assicura che (1) un file rimanga disponibile finché almeno uno dei k nodi che lo contengono sia ancora attivo e raggiungibile attraverso Internet; (2) con alta probabilità, l'insieme di nodi che contengono un file è diverso in termini di posizione geografica, appartenenza, amministrazione, connettività alla rete, leggi vigenti, ecc.; e, (3) il numero di file assegnati ad ogni nodo è relativamente bilanciato. La (1) deriva dalle proprietà dell'algoritmo di PAST descritte nella sezione 2.2.3.10 a pagina 29. La (2) e la (3) derivano dalla distribuzione uniforme e quasi-casuale degli identificatori assegnati a ciascun file e nodo.

2.2.3. Sicurezza

Il modello di sicurezza di PAST si basa sulle seguenti supposizioni:

1. È computazionalmente impossibile violare il sistema crittografico a chiave pubblica e/o la funzione crittografica di hash usati in PAST;

2. Anche se i client, gli operatori dei nodi o i loro software non sono fidati, e attaccanti possono controllare il comportamento di nodi PAST individuali, si suppone che la maggior parte dei nodi PAST si comporti correttamente;

3. Un attaccante non può controllare il comportamento delle smartcard.

Nella trattazione che segue assumeremo l'utilizzo delle smartcard. Come viene spiegato più avanti in questa sezione, è possibile operare una rete PAST senza l'ausilio di smartcard. Ogni nodo PAST ed ogni utente del sistema ha una smartcard. Una coppia (chiave pubblica, chiave privata) è associata a ciascuna smartcard. Ogni chiave pubblica delle smartcard è firmata dalla chiave privata dell'ente rilasciante per questioni di certificazione. Le smartcard generano e verificano i vari certificati usati durante l'inserimento e di reclamo dello spazio, e mantengono le quote di storage. Di seguito tracceremo le principali funzioni di sicurezza.

2.2.3.1. Generazione dei nodeId

Una smartcard fornisce il nodeId per il nodo PAST che la detiene. Il nodeId si basa su un hash crittografico della chiave pubblica della smartcard. Questa assegnazione probabilistica dei nodeId assicura una copertura uniforme dello spazio dei nodeId e la diversificazione dei nodi con nodeId adiacenti in termini di posizione geografica, appartenenza, amministrazione, connettività alla rete, leggi vigenti, ecc. Inoltre, i nodi possono verificare l'autenticità di ciascun nodeId.

2.2.3.2. Generazione dei certificati dei file e delle ricevute di salvataggio

La smartcard dell'utente che vuole inserire un file in PAST rilascia un *certificato del file*. Il certificato contiene un hash crittografico del contenuto del file (calcolato dal client del nodo), il fileId (calcolato dalla smartcard), il fattore di replicazione, il sale, e viene firmato dalla smartcard. Durante un'operazione di inserimento, il certificato del file permette, a ciascun nodo che detiene il file, di verificare (1) che l'utente è autorizzato ad inserire il file nel sistema, il che impedisce ai client di eccedere le loro quote di storage; (2) che il contenuto del file arrivato al nodo che lo deve contenere non sia stato corrotto durante il tragitto dal client da nodi malfunzionanti o malevoli; e, (3) che il fileId sia autentico, impedendo l'avvenimento di attacchi di tipo denial-of-service dove client malevoli tentano di esaurire lo storage di un sottoinsieme dei nodi PAST della rete scegliendo fileId con valori vicini tra di loro. Ogni nodo di storage, dopo aver salvato con successo una copia del file, rilascia ed invia al client una *ricevuta di salvataggio*, la

quale permette al client di verificare che le k copie del file siano state create su nodi con nodeId adiacenti, il che impedisce ad un nodo malevolo di sopprimere la creazione di k repliche diverse. Durante un'operazione di recupero, il certificato del file viene restituito insieme al file, permettendo così al client di verificarne l'autenticità e l'integrità.

2.2.3.3. Generazione dei certificati e delle ricevute di reclamo

Prima di iniziare un'operazione di reclamo, la smartcard dell'utente genera un certificato di reclamo. Il certificato contiene il fileId, viene firmata dalla smartcard e viene inclusa nella richiesta di reclamo che viene inoltrata ai nodi detentori del file. Quando viene elaborata una richiesta di reclamo, la smartcard di un nodo di storage verifica prima che la firma nel certificato del reclamo sia coerente con quella nel certificato del file salvato. Questo impedisce che altri utenti al difuori del proprietario del file ne possano richiedere il reclamo dello storage da esso occupato. Se l'operazione di reclamo viene accettata, la smartcard del nodo di storage genera una ricevuta di reclamo. La ricevuta contiene il certificato di reclamo e la quantità di storage reclamata; viene firmata dalla smartcard ed inoltrata al client.

2.2.3.4. Quote di storage

Le smartcard mantengono le quote di storage. La smartcard di ciascun utente viene rilasciata con una quota di utilizzo, in base a quanto storage si vuole concedere all'utente. Quando viene rilasciato il certificato di un file, una quantità pari alla dimensione del file moltiplicata per il fattore di replicazione viene detratta dalla quota. Quando un client presenta una ricevuta di reclamo appropriata, rilasciata da un nodo di storage, la quantità reclamata viene accreditata alla quota del client. Questo previene i client dall'eccedere le proprie quote. La smartcard di un nodo specifica la quantità di storage con la quale il nodo contribuisce alla rete (possibilmente anche zero). I nodi vengono controllati casualmente per vedere se possono fornire i file che dovrebbero contenere, scovando così eventuali nodi che cercano di barare offrendo meno storage di quello mostrato nella loro smartcard.

2.2.3.5. Integrità del sistema

Sono diverse le condizioni che assicurano l'integrità di base del sistema PAST. Primo, per mantenere un bilanciamento del carico approssimativo tra i nodi di storage, i nodeId ed i fileId dovrebbero essere uniformemente distribuiti. La procedura per generare e verificare i nodeId ed i fileId ci assicura di questo. Secondo, deve esserci un equilibrio tra la somma di tutte le quote dei client (richiesta) e lo storage totale disponibile (offerta). Il broker assicura il bilanciamento, in linea di principio usando il prezzo monetario dello storage per regolare richiesta e offerta. Terzo, singoli nodi malevoli devono essere incapaci di negare il servizio ad un client in modo persistente. Un protocollo di routing randomizzato, descritto nella sezione 2.1.8 a pagina 24 garantisce che una operazione ripetuta eventualmente verrà instradata aggirando il nodo malevolo.

2.2.3.6. Persistenza

La persistenza dei file in PAST dipende principalmente da tre condizioni. (1) Gli utenti non autorizzati non possono reclamare lo storage di un file, (2) un file viene salvato su k nodi di storage diversi, e (3) c'è sufficiente diversità nell'insieme dei nodi di storage che detengono un determinato file. Rilasciando e richiedendo certificati di reclamo, le smartcard assicurano la condizione (1). La (2) viene resa possibile dall'utilizzo delle ricevute di storage e la (3) dalla distribuzione quasi-casuale dei nodeId, i quali non

possono essere manipolati da un attaccante. La scelta del fattore di replicazione k deve tenere presente la frequenza dei fallimenti dei nodi di storage per assicurare un livello di disponibilità sufficiente. Nel caso in cui il fallimento di un nodo di storage comporti la perdita di file salvati, il sistema ristabilisce automaticamente k copie come parte della procedura di recupero.

2.2.3.7. Privacy ed integrità dei dati

Gli utenti possono usare la cifratura per proteggere la privacy dei loro dai, usando sistemi di cifratura di loro scelta. Questi sistemi di cifratura non dipendono dalle smartcard. L'integrità dei dati viene assicurata tramite i certificati dei file rilasciati dalle smartcard.

2.2.3.8. Pseudonimia

La firma della smartcard di un utente è l'unica informazione che associa un file salvato o una richiesta all'utente responsabile. L'associazione tra una smartcard e l'identità di un utente è nota solo all'utente, a meno che l'utente divulghi questa informazione volontariamente. La pseudonimia dei nodi di storage viene assicurata in modo simile dato che la firma della smartcard del nodo non viene collegata all'identità dell'operatore del nodo. In più, lo schema di routing di Pastry previene la disseminazione sparsa di informazioni riguardanti le associazioni tra nodeId ed indirizzi IP.

2.2.3.9. Smartcard

L'utilizzo delle smartcard e/o la presenza dei broker come terze parti fidate non sono fondamentali per l'architettura di PAST. Primo, le smartcard possono essere rimpiazzate da servizi on-line sicuri di quote eseguiti da altri broker. Secondo, è possibile eseguire PAST senza l'ausilio di terze parti. Comunque, data la tecnologia odierna, le smartcard ed i broker risolvono diversi problemi in modo efficiente:

1. Le smartcard ed i broker assicurano l'integrità dell'assegnazione dei nodeId ed i fileId. Senza una terza parte, sarebbe molto difficile e costoso prevenire gli attaccanti dallo scovare, tramite prove ripetute, fileId e nodeId che cadono tra due nodeId adiacenti esistenti in PAST.

2. Le smartcard mantengono le quote di storage in maniera sicura ed efficiente. Raggiungere la stessa scalabilità ed efficienza con un servizio on-line di quote è difficile. Senza l'ausilio di terze parti, implementare un sistema di quote richiederebbe protocolli di accordo molto complessi.

3. Le smartcard sono un mezzo conveniente tramite il quale un utente può ottenere le credenziali necessarie per unirsi al sistema in maniera anonima. Un utente può ottenere una smartcard con la quota desiderata in maniera anonima senza dover rivelare la propria identità a terze parti.

2.2.3.10. Schema di routing di PAST

PAST si basa su Pastry per il routing e la localizzazione dei file, il cui funzionamento è descritto nella sezione 2.1.4 a pagina 16. Dato un fileId, Pastry inoltra il messaggio associato al nodo il cui nodeId sia il più vicino numericamente ai 128 bit più significativi del fileId (ricordiamo che i fileId sono lunghi 160 bit). Dato che un file viene salvato su k nodi i cui nodeId sono numericamente vicini ai 128 bit più significativi del fileId, segue che il file può essere ottenuto a meno che tutti e k i nodi falliscano simultaneamente.

Dato che Pastry sceglie ripetutamente un passo di routing localmente "breve" (si veda la sezione 2.1.7.2 a pagina 22), i messaggi tendono a raggiungere per primo uno, dei k nodi contenenti il file desiderato, che si trova vicino al client, secondo la metrica di prossimità, come già discusso nella sezione 2.1.7.3 a pagina 22.

2.2.4. Storage management

Lo storage management di PAST mira a rendere possibile un alto livello di utilizzazione dello storage globale, ed ottenere una lenta degradazione delle prestazioni del sistema mentre questo si avvicina alla massima utilizzazione. La quantità di storage occupata dai file (incluse le repliche) in PAST dovrebbe poter crescere fino a raggiungere una grossa porzione della capacità totale di storage del sistema, prima che gran parte delle richieste di inserimento di file venga respinta, o soffra di prestazioni ridotte.

In linea con l'architettura generale decentralizzata di PAST, un obiettivo importante di progettazione per lo storage management è quello di basarsi solamente sul coordinamento locale tra nodi con nodeId vicini. In questo modo si mira ad integrare pienamente lo storage management con l'inserimento dei file ed avere overhead legati allo storage management modesti.

Le responsabilità dello storage management consistono in (1) bilanciare il rimanente spazio di storage libero tra i nodi nella rete PAST mentre l'utilizzazione generale del sistema si avvicina al 100%; e, (2) mantenere invariato il fatto che copie di ciascun file si trovino sui k nodi attivi con nodeId i più vicini al fileId. Gli obiettivi (1) e (2) sembrano essere conflittuali, poiché richiedere che un file venga salvato sui k nodi più vicini al suo fileId sembra non lasciare spazio a nessun bilanciamento del carico esplicito. PAST risolve questo conflitto in due modi.

Primo, PAST permette anche ad un nodo che non è tra i k nodi numericamente più vicini al fileId di contenere il file, se il nodo si trova nel leaf set di uno dei k nodi. Questo processo viene chiamato deviazione delle repliche, ed ha come scopo quello di appianare le differenze nelle capacità di storage e nell'utilizzo di nodi nel leaf set. La deviazione delle repliche deve essere effettuata con cura, per assicurare che la disponibilità del file non ne risenta. Secondo, la deviazione dei file viene eseguita quando l'intero leaf set di un nodo sta raggiungendo l'utilizzo massimo. Il suo scopo è quello di ottenere un bilanciamento del carico più globale su grosse porzioni dello spazio dei nodeId. Un file viene deviato ad una diversa porzione dello spazio dei nodeId scegliendo un sale diverso nella generazione del suo fileId.

2.2.4.1. Cause dello sbilanciamento del carico

Ricordiamo che ogni nodo nella rete PAST mantiene uno leaf set, il quale contiene gli l nodi con nodeId numericamente più vicini al nodo dato ($l/2$ con nodeId più grande e $l/2$ con nodeId più piccolo). Normalmente, le repliche dei file vengono salvate su k nodi che sono i più vicini numericamente al fileId (k non può essere più grande di $(l/2) + 1$).

Consideriamo il caso in cui non tutti i k nodi più vicini possano contenere una replica a causa di storage insufficiente, però esistono k nodi nei leaf set dei k nodi originali che possono contenere i file. Tale sbilanciamento nello storage tra i $l+k$ nodi nell'intersezione dei k leaf set possono originare da diverse cause:

- A causa di variazioni statistiche nell'assegnazione dei nodeId e fileId, il numero di file che viene assegnato a ciascun nodo può differire.

- La distribuzione delle dimensioni dei file inseriti può avere varianza elevata ed essere a coda pesante[3].
- Le capacità individuali di storage dei nodi PAST sono diverse.

La deviazione delle repliche mira a bilanciare lo spazio vuoto rimanente tra i nodi in ciascun leaf set. In più, mentre l'utilizzazione globale del sistema PAST aumenta, la deviazione dei file potrebbe diventare necessaria per bilanciare il carico dello storage tra le diverse porzioni dello spazio dei nodeId.

2.2.4.2. Deviazione delle repliche

Quando una richiesta di inserimento raggiunge un nodo con nodeId tra i k numericamente più vicini al fileId, il nodo controlla se può contenere una copia del file nel suo disco locale. Se sì, salva il file, crea una ricevuta di salvataggio, e inoltra il messaggio agli altri $k-1$ nodi con nodeId più vicini al fileId (dato che questi nodi devono esistere nello leaf set del nodo in questione, il messaggio può essere inoltrato direttamente). Ciascun nodo a turno proverà a salvare una replica del file e restituirà una ricevuta di salvataggio.

Se un nodo A non può contenere un messaggio nella sua memoria locale, considera la deviazione delle repliche. A questo scopo, A sceglie un nodo B dal suo leaf set che non è tra i k nodi più vicini e non contiene già una replica deviata del file. A chiede a B di mantenere una copia del file per conto di A, poi mantiene una entry per il file nella sua tabella con un puntatore a B, e restituisce una ricevuta di salvataggio come al solito. Diciamo che A ha deviato una copia del file a B. In realtà questo meccanismo è più complesso, e la descrizione dettagliata (che si può trovare in [18]) esula dagli scopi di questo lavoro.

2.2.4.3. Deviazione dei file

Lo scopo della deviazione dei file è quello di bilanciare lo storage libero tra le diverse porzioni dello spazio dei nodeId in PAST. Quando l'operazione di inserimento di un file fallisce perché i k nodi più vicini al fileId non sono in grado di contenere il file o di deviarne le repliche localmente nei loro leaf set, un messaggio di fallito tentativo viene inviato al nodo client richiedente l'inserimento. Il nodo client, allora, genera un fileId diverso usando un sale diverso, e riprova l'operazione di inserimento.

Un nodo client riprova questo procedimento per altre tre volte. Se, dopo quattro tentativi, l'inserimento fallisce ancora, l'operazione viene abortita ed il fallimento viene riportato all'applicazione. Tale fallimento implica che il sistema non è stato in grado di localizzare lo spazio necessario per salvare k copie del file. In tali casi, un'applicazione può decidere di riprovare l'operazione con una dimensione del file più piccola (per esempio frammentando il file) e/o diminuire il numero di repliche.

2.2.4.4. Mantenimento delle repliche

PAST mantiene l'invariante che k copie di ciascun file inserito devono essere mantenute su nodi diversi dello stesso leaf set. Questo avviene come segue.

Ricordiamo come Pastry gestisce l'aggiunta di nodi nuovi, il fallimento ed il ritorno di nodi esistenti (si vedano le sezioni 2.1.6.1 a pagina 18 e 2.1.6.2 a pagina 19). Come parte di questi aggiustamenti, un nodo può diventare uno dei k nodi più vicini per

[3]Nella teoria della probabilità, le distribuzioni a coda pesante sono distribuzioni di probabilità le cui code non sono limitate esponenzialmente: cioè, le loro code sono più pesanti della distribuzione esponenziale.

alcuni file; l'invariante dello storage richiede che tale nodo acquisisca una replica di ciascun file che verifichi questa evenienza, ed in caso ricreare le repliche mantenute da nodi eventualmente falliti in precedenza. In maniera simile, un nodo può smettere di essere tra i k nodi più vicini per certi file; l'invariante permette al nodo di sbarazzarsi delle copie dei file in questione. Gli ulteriori dettagli di questo procedimento (che si può trovare in [18]) sono tralasciati.

2.2.5. Caching

Ciascun nodo PAST può mantenere copie aggiuntive di un file. In questo modo mira ad ottenere bilanciamento del carico delle query e throughput alto per i file popolari, riducendo le distanze ed il traffico nella rete. I dettagli vengono tralasciati, ma si possono trovare in [18].

2.3. Erasure Coding

Durante la descrizione di PAST abbiamo introdotto il concetto di ridondanza come tecnica per assicurare un elevato livello di disponibilità e persistenza dei file in un sistema di storage distribuito. Ora introdurremo un'altra tecnica, chiamata *erasure coding*[23, 24, 25], che può essere usata in alternativa alla semplice ridondanza, come anche in combinazione con quest'ultima.

Un erasure code fornisce ridondanza senza l'overhead della replicazione in senso stretto. Gli erasure code dividono un oggetto in m frammenti, e li ricodificano in n frammenti, con $n > m$. Chiamiamo rate of encoding la quantità $r = m/n < 1$, che rappresenta una misura dell'efficienza del codice. Un rate di r aumenta il costo in termini di storage di un fattore $1/r$.

La proprietà principale degli erasure code è che si può ricostruire il messaggio originale da qualsiasi combinazione di m frammenti dagli n frammenti iniziali. Non importa quali blocchi arrivano al richiedente, ma quanti ne arrivano, per ricomporre il messaggio originale. Un erasure code che presenta tali caratteristiche prende il nome di *(n,m)*-codice e permette di ricostruire i dati originali nell'ipotesi in cui siano avvenute al massimo *n-m* cancellazioni in un gruppo di n frammenti codificati.

Non approfondiamo oltre questi algoritmi, riportati qua per completezza, dato che nella nostra trattazione ci limiteremo a considerare soltanto la replicazione come tecnica per assicurare la disponibilità e la persistenza dei file nello storage distribuito.

3. Un nuovo approccio al calcolo distribuito volontario

In questo capitolo descriveremo una nuova architettura per il calcolo distribuito volontario. Come abbiamo già accennato, utilizzeremo un modello molto simile ad uno storage distribuito PAST, che a sua volta si basa su Pastry per le operazioni di routing e di ricerca. In questo modo si mira a far avere al sistema di calcolo distribuito volontario le proprietà di scalabilità, robustezza ed efficienza che contraddistinguono la rete PAST/Pastry.

3.1. Panoramica sull'architettura del sistema

Nel modello proposto possiamo identificare due tipi di partecipanti: i nodi volontari, costituiti dai calcolatori dei volontari partecipanti ai progetti di ricerca, ed i nodi dedicati, costituiti dai calcolatori dei centri di ricerca, i quali avranno un ruolo molto simile a quello dei server degli attuali progetti di calcolo volontario distribuito.

I nodi volontari sono costituiti dai calcolatori dei volontari partecipanti ai progetti di calcolo distribuito volontario, i quali hanno come scopo principale quello di eseguire i calcoli scientifici per i fini del progetto di ricerca a cui i volontari afferiscono.

I nodi dedicati dei progetti di calcolo non eseguono calcoli scientifici. La loro mansione principale è quella di occuparsi della generazione delle unità di lavoro, e della successiva raccolta dei risultati calcolati dai volontari. Essi hanno un ruolo molto simile agli attuali server dei progetti di calcolo distribuito volontario, anche se, nella nostra trattazione non chiamiamo questi nodi server, poiché un tale termine non sarebbe corretto. Come vedremo, questi nodi non vengono contattati dai nodi volontari per ricevere il lavoro, come accade nei attuali progetti di calcolo distribuito volontario. In generale, ci possono essere diversi nodi dedicati, ma in linea di principio ne potrebbe bastare anche uno solo.

3.1.1. Il funzionamento

I nodi volontari insieme ai nodi dedicati formano una rete overlay PAST-simile, cioè non solo un sistema di storage distribuito: i volontari non donano semplicemente il loro storage libero, ma anche il tempo libero delle loro CPU. A grosse linee, il sistema opera secondo lo schema seguente:

1. I nodi dedicati dei progetti di ricerca, di volta in volta, creano e pubblicano sullo storage distribuito le unità di lavoro.

2. I nodi volontari scaricano i file da elaborare dallo storage distribuito.

3. I nodi volontari compiono le elaborazioni necessarie e creano i file contenenti i risultati calcolati, i quali vengono a loro volta caricati sullo storage distribuito.

4. I nodi dedicati controllano periodicamente lo storage distribuito e raccolgono eventuali risultati calcolati. Eventualmente, il ciclo riparte dal primo punto.

3.1.2. I nodi dedicati

Il nodo dedicato del progetto di calcolo è dotato di un software client PAST, che gli permette di collegarsi alla rete di calcolo distribuito volontario. I nodi dedicati del progetto insieme ai nodi client formano un sistema simile storage distribuito, sul quale sarà possibile salvare le unità di lavoro egli eventuali risultati calcolati.

I server ricoprono un ruolo centrale nel funzionamento negli attuali sistemi di calcolo distribuito volontario. Oltre a generare le unità di lavoro e a raccogliere i risultati calcolati dai client, prendono anche decisioni sullo scheduling e sull'assegnazione del lavoro.

Anche nel modello proposto, i nodi del progetto creano le unità di lavoro. Questo procedimento dipende dalla particolare applicazione scientifica che si intende risolvere col calcolo distribuito volontario. Una volta generate, le unità di lavoro vengono distribuite sulla rete PAST, dalla quale potranno essere scaricate dai nodi volontari. I volontari a loro volta, terminata l'elaborazione dei risultati, creano i file contenenti i risultati completati e li pubblicano sullo storage distribuito. Questi risultati vengono raccolti periodicamente dai nodi dedicati.

Come partecipante ad una rete di calcolo distribuito volontario PAST-simile, al nodo dedicato viene assegnato un nodeId Pastry da 128 bit, che lo identifica univocamente. Come già visto nelle sezioni precedenti, il nodeId viene generato dall'applicazione di una funzione hash crittografico alla chiave pubblica del nodo. In questo modello possiamo immaginare i nodi dedicati dotati di smartcard (coerentemente con quanto già visto durante la trattazione di PAST) le quali contengono una coppia (chiave pubblica, chiave privata), che si può usare a tale scopo.

I nodi dedicati (ce n'è può essere anche uno solo) sono quelli che danno inizio alla overlay PAST/Pastry. Inizialmente la overlay del progetto di ricerca è composta solamente dai suoi nodi dedicati, ed i nodi volontari che si vogliono aggiungere devono usare quest'ultimi come nodi di bootstrap.

3.1.3. I nodi volontari

Come al solito, supponiamo che i nodi volontari siano semplici calcolatori (PC, game console, smartphone, ecc.) collegati ad Internet, proprietà di persone (volontari) che vogliono donare le risorse non utilizzate per svolgere elaborazioni atte ad aiutare progetti di ricerca. Sul nodo volontario supponiamo in esecuzione il software client, il quale può essere scaricato ed installato dal proprietario del calcolatore in questione.

Dal punto di vista del volontario, i cambiamenti del modello appaiono abbastanza trasparenti. Come sempre, il volontario deve scegliere uno o più progetti di ricerca ai quali intende donare le risorse del proprio calcolatore, deve installare un software client (simile al software client BOINC), e gli eventuali programmi di calcolo scientifico specifici dei progetti di ricerca.

Il software client si comporta in parte come un client PAST. Infatti, permette al nodo volontario di connettersi ad una rete PAST-simile, dove nodi volontari e dedicati formano un sistema simile ad uno storage distribuito. Da questo sistema, il generico nodo volontario scarica i risultati[1], li elabora (cioè produce i "risultati calcolati"), e mette i risultati calcolati nello storage distribuito.

L'elaborazione avviene sempre durante i tempi morti, in cui l'utente volontario non sta utilizzando il proprio calcolatore (ad esempio quando parte lo screensaver). Lo scambi dei dati invece, può avvenire anche durante il normale funzionamento del calcolatore.

[1] Il temine risultato, come abbiamo spiegato in precedenza nella sezione 1.3 a pagina 2 , si riferisce ad una istanza dell'unità di lavoro, o una copia dell'unità di lavoro, con il campo "risultato calcolato" lasciato vuoto.

Infatti, possiamo pensare al client di calcolo volontario, come ad un programma peer-to-peer di file-sharing, il quale esegue download ed upload di dati senza interferire con il lavoro dell'utente. Opzionalmente, si può dotare il client di un meccanismo che limita la banda da esso utilizzata, in caso in cui l'utente ne senta la necessità, funzionalità questa, già presente sia nel client BOINC, sia nei più comuni programmi di file-sharing.

Anche ai nodi volontari viene assegnato un nodeId, come accade per i nodi dedicati. Possiamo pensare il nodo volontario dotato a sua volta di una smartcard, contenente la coppia (chiave pubblica, chiave privata). Coerentemente con quanto abbiamo visto in precedenza, il nodeId del nodo volontario viene generato applicando una funzione di hash crittografico alla sua chiave pubblica.

Quando un nodo cerca di connettersi per la prima volta alla overlay di calcolo distribuito volontario, lo deve fare tramite dei nodi di bootstrap. Le informazioni sui nodi di bootstrap si possono ottenere facilmente da un web server, simile a quello da cui l'utente scarica il software client (o anche lo stesso). I nodi di bootstrap possono inizialmente essere alcuni dei nodi dedicati, ma in seguito si possono usare anche nodi volontari a questo scopo.

3.2. Caratteristiche principali di un progetto di ricerca

Un progetto di ricerca di norma produce unità di lavoro di taglie e tipologie diverse, in modo da poter sfruttare al meglio le risorse di una varia gamma di calcolatori volontari. Senza perdere generalità, possiamo caratterizzare un generico progetto di ricerca definendo alcuni parametri, come vedremo in seguito. Nel modello proposto è assolutamente indifferente la tipologia di calcolo scientifico che si vuole risolvere, caratteristica che gli conferisce validità generale.

3.2.1. Numero di nodi volontari e dedicati

Il *numero di nodi volontari* che il progetto di calcolo distribuito volontario ha a sua disposizione è un parametro importante da determinare, poiché vi dipendono praticamente tutti gli altri che seguono. Un modo semplice per avere una stima approssimata di tale valore potrebbe essere quello di contare il numero di download del software di calcolo scentifico per un dato progetto di calcolo volontario distribuito.

Un altro parametro interessante è *il numero di nodi dedicati* di cui il progetto di ricerca dispone. In questo modello un nodo dedicato ha due mansioni: creare e distribuire le unità di lavoro e raccogliere i risultati calcolati dai nodi volontari.

3.2.2. Caratteristiche delle unità di lavoro

Quanti tipi di unità di lavoro produce il dato progetto di ricerca, che equivale al numero di categorie diverse di nodi volontari (classi di equivalenza) per il dato progetto di ricerca. Coerentemente con i modelli attuali di calcolo distribuito volontario, un'unità di lavoro appartenente ad una certa tipologia, può essere svolta solamente da un nodo volontario appartenente alla stessa tipologia. Per esempio, supponiamo di avere n tipologie diverse di unità di lavoro. Avremmo n tipologie di nodi volontari, ciascuna corrispondente ad un'unica tipologia di unità di lavoro. Il volontario appartenente alla tipologia j può elaborare solamente istanze di unità di lavoro di tipo j $(1 \leq j \leq n)$. Viceversa, un'unità di lavoro di tipo k può essere elaborata solamente da un nodo volontario di tipologia k $(1 \leq k \leq n)$.

È chiaro che questo parametro dipende sia dallo specifico progetto di ricerca (dall'applicazione scientifica, se vogliamo), sia dalle caratteristiche del pool di nodi volontari

a disposizione del progetto. In base alle caratteristiche dell'applicazione scientifica data, bisogna decidere quali differenze nelle caratteristiche hardware e/o software, ed in quale misura, ha senso considerare per determinare la relazione di equivalenza tra nodi volontari. In base a questa relazione di equivalenza sarà possibile determinare se due volontari appartengono alla stessa tipologia, o se appartengono a due tipologie diverse; in poche parole, la relazione di equivalenza permette di partizionare l'insieme dei volontari in classi di equivalenza, dove ad ogni classe di equivalenza possiamo associare una tipologia di nodo volontario. Due nodi sono equivalenti se appartengono entrambi alla stessa tipologia (classe di equivalenza).

Sì noti anche che la determinazione della relazione di equivalenza può dipendere anche da considerazioni simili a quelle fatte nella sezione 1.5.1.3 sulla *ridondanza omogenea*.

In questo parametro possiamo includere anche le *dimensioni delle unità di lavoro*, poiché in genere un'unità di lavoro più grande comporta più dati da elaborare. Dalle dimensioni delle unità di lavoro dipenderanno anche le *dimensioni dei risultati calcolati*.

Quante unità di lavoro, in media per ciascun tipo, vengono create ed immesse sulla rete di calcolo in un dato intervallo temporale. Definire questa quantità, in un certo senso, equivale a definire *le proporzioni dei tipi diversi di nodi volontari rispetto al numero totale di nodi volontari* a disposizione del progetto. Nella realtà si cerca di determinare il numero di volontari per tipologia, o perlomeno, avere una buona stima di tale parametro. Il numero di unità di lavoro per tipologia lo si deduce da questo parametro.

Quanto tempo necessitano, in media, le unità di lavoro per essere completamente elaborate. Quando si decide il numero di tipologie di unità di lavoro, e le caratteristiche di ciascuna tipologia, si tiene conto anche del tempo necessario che impiega, in media, un nodo volontario per elaborare un'istanza di un'unità di lavoro. Infatti, è buona norma che tutti i volontari, indipendentemente dal tipo, impieghino in media lo stesso tempo per completare le istanze delle corrispettive tipologie di unità di lavoro. Questo anche per una questione legata al credito che viene assegnato ai volontari: il meccanismo di assegnazione del credito non tiene conto in genere della velocità del calcolatore, ma piuttosto del tempo CPU che esso dedica al calcolo distribuito volontario.

3.3. Unità di lavoro e risultati calcolati nello storage distribuito

La generazione delle unità di lavoro avviene ai nodi dedicati dei progetti di ricerca. I procedimenti con i quali questo avviene sono fortemente *project-dependant*, e non saranno trattati in questo lavoro. Abbiamo già accennato al fatto che, dopo essere state create, le unità di lavoro vengono salvate sullo storage distribuito, per poi essere scaricate dai nodi volontari.

Un volontario deve poter cercare sullo storage distribuito una o più unità di lavoro appartenenti alla sua stessa tipologia; a sua volta, un nodo dedicato deve poter raccogliere in maniera efficiente i risultati calcolati dai nodi volontari. Come abbiamo già discusso, per localizzare una risorsa su una rete di questo tipo bisogna conoscere l'identificatore univoco di quella risorsa. Infatti, sulle DHT non esistono modi semplici per eseguire ricerche di risorse se non si conosce l'identificatore univoco (o un modo per ottenere tale identificatore, come ad esempio il nome del file). Quando si vogliono eseguire ricerche con nomi di file parziali bisogna ricorrere agli indici inversi [26] o ad altre tecniche di ricerca su DHT al quanto complesse. Il problema fortunatamente diventa affrontabile nel nostro caso, come presto vedremo.

3.3.1. Operazioni sulle unità di lavoro

Descriveremo di seguito le operazioni di distribuzione e di ricerca delle unità di lavoro sullo storage distribuito.

3.3.1.1. Distribuzione delle unità di lavoro

I nodi volontari e quelli dedicati formano una overlay PAST-simile. Le unità di lavoro non sono altro che file, e quindi, per poterle salvare sullo storage distribuito è necessario assegnare loro un fileId da 160 bit; quando un nodo dedicato genera un'unità di lavoro, applica una funzione di hash crittografico (ad esempio lo SHA-1) al nome testuale di quest'ultima per determinare il corrispondente fileId. Dopodiché salva, seguendo il meccanismo di PAST, l'unità di lavoro sullo storage distribuito.

L'unità di lavoro, oltre ai dati da elaborare, deve contenere alcune informazioni come: un numero identificatore che la contraddistingue da tutte le altre unità di lavoro; un intero che identifica il tipo di unità di lavoro; la deadline o scadenza massima entro la quale l'unità di lavoro deve essere completata – superata la deadline l'unità di lavoro scade ed i nodi che la mantengono in memoria la cancellano.

Insieme ad una data unità di lavoro, il nodo dedicato crea anche un file di tipo *riferimento ad un'unità di lavoro*. Un file di riferimento deve contenere il fileId della corrispettiva unità di lavoro. Quando un volontario cerca del lavoro da svolgere, in realtà prima cerca un riferimento ad un'unità di lavoro appartenente alla sua stessa tipologia; soltanto dopo scarica il lavoro vero e proprio. Come vedremo i riferimenti sono necessari per permettere la ricerca delle unità di lavoro.

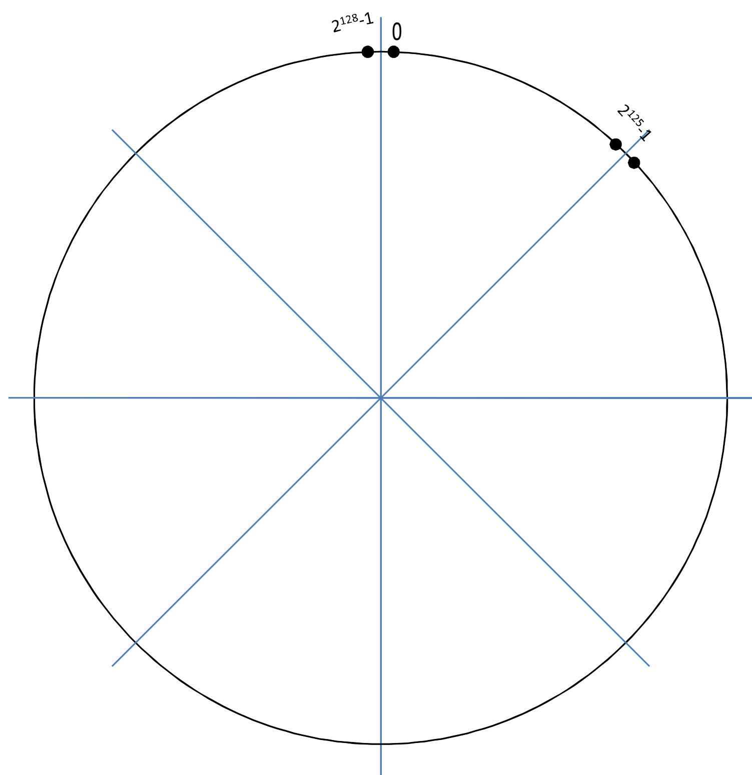

Figura 3.1.: Possibile partizionamento dello spazio circolare dei nodeId in $n = 8$ intervalli uguali. Come sappiamo, lo spazio degli nodeId in Pastry ha cardinalità 2^{128}. Ciascun intervallo conterrà $2^{128}/8 = 2^{125}$ nodeId consegutivi.

Supponiamo che il nostro progetto di calcolo volontario distribuito abbia n tipi diversi di unità di lavoro (nodi volontari). Possiamo immaginare di partizionare lo spazio degli nodeId in tanti intervalli quanti sono i tipi diversi di unità di lavoro (nodi volontari), e cioè n, dove ciascun intervallo è responsabile di una data tipologia di unità di lavoro.

Questi intervalli si possono costruire semplicemente dividendo lo spazio dei nodeId in n porzioni uguali, oppure partizionando tale spazio in modo che i rapporti tra le cardinalità degli intervalli rispecchino i rapporti tra le cardinalità delle corrispettive tipologie di unità di lavoro prodotte. La figura 3.1 nella pagina precedente illustra un possibile partizionamento.

Quando un nodo dedicato genera un'unità di lavoro, genera anche il corrispettivo riferimento. L'unità di lavoro viene salvata sullo storage distribuito nel modo usuale, invece il suo riferimento viene salvato sulla porzione dello spazio degli nodeId che corrisponde alla tipologia dell'unità di lavoro. Infatti, il nodo dedicato assegna al riferimento un fileId generato, non più dall'applicazione di una funzione hash al nome testuale del file, bensì casualmente, con distribuzione uniforme sull'intervallo degli nodeId responsabili per la tipologia dell'unità di lavoro a cui il riferimento punta[2]. In questo modo si crea una corrispondenza biunivoca tra gli intervalli dello spazio dei nodeId e le tipologie di unità di lavoro.

Un modo semplice per ottenere un partizionamento dello spazio dei nodeId è quello di distinguere gli intervalli a partire dai prefissi dei nodeId in essi contenuti. Ad esempio, se vogliamo partizionare lo spazio degli nodeId in 8 parti con cardinalità uguale, si procede nel seguente modo:

tutti i nodeId che iniziano per "000" appartengono al primo intervallo;
tutti i nodeId che iniziano per "001" appartengono al secondo intervallo;
tutti i nodeId che iniziano per "010" appartengono al terzo intervallo;
⋮ ⋮ ⋮
tutti i nodeId che iniziano per "111" appartengono all'ottavo intervallo.

3.3.1.2. Ricerca delle unità di lavoro

La ricerca delle unità di lavoro avviene secondo il seguente schema:

Il software client del nodo volontario, alla sua prima esecuzione, e poi eventualmente quando rileva cambiamenti nella configurazione hardware e/o software del sistema in cui risiede, esegue un benchmark delle prestazioni di quel nodo. Dal risultato del benchmark viene determinata la tipologia di appartenenza del volontario, che a sua volta determina il tipo di unità di lavoro che può essere svolta dal nodo volontario (ricordiamo la corrispondenza biunivoca tra tipologie di nodi volontari e tipologie di unità di lavoro).

Determinata la tipologia di appartenenza, il nodo volontario deve procurarsi del lavoro da svolgere. Per poter trovare unità di lavoro adeguate, il volontario deve procurarsi dei riferimenti a quel tipo di unità di lavoro.

Dalla discussione fatta nella sezione 3.3.1.1 nella pagina precedente segue che tutti i riferimenti, relativi ad una data tipologia di unità di lavoro, si trovano nei nodi con nodeId nell' intervallo responsabile per quella particolare tipologia di unità di lavoro. Viceversa, dato un particolare intervallo di nodeId, responsabile per un certo tipo di unità di lavoro, nei nodi della rete con nodeId in esso contenuti si possono trovare soltanto riferimenti ad unità di lavoro di quel tipo.

[2]Eventualmente si può usare il padding per far arrivare il fileId alla lunghezza di 160 bit, dato che i nodeId sono lunghi solo 128 bit. Si possono aggiungere 32 zeri a destra del fileId così calcolato, in quanto sappiamo che solo i 128 bit più significativi del fileId vengono usati per il routing.

Se vogliamo utilizzare il partizionamento per prefissi descritto alla fine della sezione 3.3.1.1, si ottiene una corrispondenza biunivoca tra la tipologia di unità di lavoro a cui punta il riferimento ed il prefisso del fileId che gli viene assegnato.

Per ottenere un riferimento ad un'unità di lavoro di un certo tipo il nodo client deve generare un nodeId casuale (cioè una sequenza di 128 bit) compreso nell'intervallo di nodeId a cui corrisponde la tipologia dell'unità di lavoro desiderata. Con questo nodeId (eventualmente con 32 zeri di padding a destra, per portare la sequenza a 160 bit) come chiave, invia una richiesta di tipo "ottieni riferimento" sulla rete PAST/Pastry. Per le proprietà della overlay PAST/Pastry, sappiamo per certo che il messaggio giungerà ad un nodo della rete, più precisamente, al nodo con nodeId numericamente più vicino alla chiave data. Il nodo, una volta ricevuta la richiesta di riferimento, guarda nella sua memoria locale in cerca di riferimenti, e restituisce una lista, eventualmente limitata ad un massimo di entry, contenente i riferimenti in suo possesso. Dato il partizionamento dello spazio dei nodeId e la distribuzione dei riferimenti descritto in precedenza, il nodo ricevente il messaggio contiene solo ed esclusivamente riferimenti ad un solo tipo di unità di lavoro, quello corrispondente all'intervallo in cui cade il nodeId del nodo in questione.

Ottenuta la lista di riferimenti, il nodo volontario sceglie quali unità di lavoro scaricare, secondo un qualche criterio, o anche in modo aleatorio, e successivamente le esegue sfruttando i periodi di non-utilizzo del calcolatore.

Se un nodo volontario riceve una lista di riferimenti vuota, ripete il procedimento precedente, generando un nuovo nodeId, sempre appartenente allo stesso intervallo, però questa volta diverso dal nodeId usato in precedenza.

3.3.2. Operazioni sui risultati calcolati

Segue la descrizione delle operazioni di distribuzione e di ricerca dei risultati calcolati sullo storage distribuito.

3.3.2.1. Distribuzione dei risultati calcolati

Dopo aver completato l'elaborazione di un'istanza di unità di lavoro, un nodo volontario crea un file contenente i risultati di tale elaborazione e lo salva sullo storage distribuito. Per fare ciò, deve prima calcolare il fileId del risultato creato. Il fileId si ricava nel modo usuale, ad esempio applicando una funzione di hash crittografico al nume testuale del file contenente il risultato. A questo punto il nodo invia il risultato sullo storage distribuito PAST/Pastry.

Risulta molto importante il nome testuale che viene assegnato a ciascun risultato. Infatti, scegliendo un formato standard per i nomi dei risultati, come ad esempio "Risultato_" + <nome testuale dell'unità di lavoro corrispondente> si possono ricavare i fileId di quest'ultimi molto facilmente, come vedremo nella sezione che segue.

3.3.2.2. Ricerca dei risultati calcolati

I nodi dedicati controllano periodicamente il completamento delle unità di lavoro da loro pubblicate sullo storage distribuito. Per cercare i risultati, i nodi dedicati hanno bisogno dei rispettivi identificatori. Poiché il nome testuale di un risultato è composto da una stringa standard del topo "Risultato_" + <nome testuale dell'unità di lavoro corrispondente>, e poiché i nodi dedicati conoscono i nomi testuali delle unità di lavoro che loro stessi hanno pubblicato sullo storage distribuito, questo procedimento risulta abbastanza facile.

Per ogni unità di lavoro generata e salvata sullo storage distribuito, il nodo dedicato calcola il nome testuale del corrispondente risultato calcolato. A questo nome applica la

solita funzione di hash crittografico ed ottiene il fileId del risultato calcolato. A questo punto invia una richiesta di tipo "GET(fileId)" sulla rete PAST/Pastry per ottenere il risultato desiderato.

Si noti che è molto importante utilizzare la stessa convenzione sui filename testuali su tutti i nodi, sia volontari che dedicati, affinché questo procedimento funzioni correttamente.

3.3.3. Considerazioni sulla correttezza

La correttezza delle operazioni sulle unità di lavoro, descritte nella sezione 3.3.1 a pagina 37, si basa due ipotesi fondamentali. La prima, è che il numero di unità di lavoro sia molto più grande del numero stesso di volontari. Il numero delle unità di lavoro deve essere tale da poter sfruttare al meglio tutti i nodi volontari disponibili, e quindi possiamo assumere come verificata questa prima ipotesi.

Anche nella realtà, nei sistemi di calcolo distribuito volontario attualmente in uso, questa ipotesi è verificata. Se prendiamo in esame un qualsiasi progetto BOINC potremmo da subito constatare che, appena installato il corrispettivo client, questo comincia a ricevere lavoro da svolgere. Questo significa che il numero delle unità di lavoro è in genere molto più grande del numero di volontari.

La seconda ipotesi è che la distribuzione dei fileId dei riferimenti è uniforme sui relativi intervalli di appartenenza, e sull'intero spazio dei nodeId. Questa ipotesi dipende dalla bontà del generatore di numeri casuali utilizzato per generare i fileId, e dal corretto partizionamento dello spazio dei nodeId descritto nelle sezioni precedenti. Il partizionamento deve far sì che le proporzioni tra gli intervalli di nodeId corrispondano alle proporzioni tra le cardinalità delle corrispettive tipologie di unità di lavoro prodotte.

Queste ipotesi fanno sì che ciascun nodo abbia mediamente m riferimenti ad unità di lavoro, con m = *numero di unità di lavoro / numero di nodi volontari*. In questo modo ci aspettiamo che le richieste di riferimento senza risposta siano relativamente poche. Generare una nuova richiesta con un nuovo fileId in genere risolve il problema. La seconda ipotesi garantisce anche la distribuzione uniforme dei riferimenti alle unità di lavoro, in modo da non sovraccaricare di richieste porzioni isolate dello spazio dei nodeId.

Per evitare situazioni in cui si hanno riferimenti ad unità di lavoro non più esistenti, bisogna assegnare a quest'ultimi un periodo di validità pari alla deadline delle unità di lavoro a cui puntano. Quando una data unità di lavoro scade, e viene di conseguenza cancellata dai nodi che la contengono, scade e viene cancellato anche il corrispettivo riferimento.

Per gestire correttamente situazioni in cui si presentano poche unità di lavoro sullo storage distribuito, bisogna integrare un meccanismo di backoff esponenziale quando si ripetono le richieste di tipo "ottieni riferimento". Un meccanismo simile si può adoperare anche per limitare il numero di volte che una data unità di lavoro viene richiesta ed eseguita.

Infatti, se per qualche motivo la distribuzione dei riferimenti o dei nodeId non è propriamente uniforme, e ci sono delle unità di lavoro che tendono ad avere più richieste delle altre, bisognerà limitare in qualche modo il numero di volte che esse vengono scaricate. Un modo può essere quello di permettere il download di un dato riferimento solo per un massimo numero di volte. Questo approccio potrebbe però soffrire di attacchi di tipo Denial-of-Service da parte di nodi malevoli, i quali, richiedendo sempre la stessa unità di lavoro, le farebbero raggiungere in fretta il limite massimo di download consentiti.

Un altro modo è quello di permettere il download di una data unità di lavoro ad intervalli temporali via via crescenti, in combinazione con il limite massimo di download

consentiti. Dopo che un nodo invia una data unità di lavoro ad un eventuale richiedente, aspetterà intervalli via via crescenti (secondo uno schema di backoff esponenziale) prima di restituire di nuovo la stessa unità di lavoro allo stesso richiedente. Il limite massimo di download consentiti permette di non impegnare più volontari del necessario con la stessa unità di lavoro.

Vale la pena notare che non c'è nessuna correlazione apparente tra la tipologia dell'unità di lavoro e il nodeId del nodo su cui viene salvata. Questo assicura una distribuzione uniforme del carico su tutta la rete, il che deriva direttamente dalle proprietà di PAST/-Pastry. L'utilizzo dei riferimenti serve proprio ad evitare situazioni di sbilanciamento del carico in cui parti dello spazio dei nodeId vengono sfruttate più di altre. I riferimenti alle unità di lavoro hanno tutti la stessa dimensione, non occupano molto spazio, e cosa più importante, vengono distribuiti uniformemente sullo spazio dei nodeId.

Lo stesso risultato non sarebbe ottenibile invece, utilizzando le unità di lavoro direttamente, al posto dei riferimenti. Anche riuscendo ad avere una distribuzione abbastanza equa per quanto riguarda il numero di unità di lavoro per nodo volontario, le unità di lavoro appartenenti a tipologie diverse hanno comunque dimensioni diverse (i riferimenti sono tutti uguali), e questo porterebbe ad un notevole sbilanciamento del carico sullo spazio dei nodeId.

Infine, osserviamo che, se un'unità di lavoro raggiunge la sua deadline senza mai essere stata elaborata, essa viene ripubblicata sullo storage distribuito con un nuovo fileId, come se si trattasse di una nuova unità di lavoro.

3.4. Altri aspetti importanti

3.4.1. Utilizzo dello storage distribuito

I nodi volontari che si uniscono alla rete di calcolo distribuito volontario forniscono, oltre al tempo CPU, parte della loro banda di comunicazione e del loro spazio di storage locale. I volontari possono scegliere, coerentemente con quanto abbiamo discusso nella sezione 2.2 a pagina 25 riguardo a PAST, di contribuire allo storage distribuito con una quantità di memoria a loro scelta. Per incentivare questo comportamento, i progetti di ricerca potrebbero introdurre una misura di credito che tenga conto anche dello spazio donato allo storage distribuito.

Per quanto riguarda i nodi dedicati valgono le stesse considerazioni: essi possono partecipare con una quantità di memoria locale a scelta (anche nulla) allo storage distribuito.

Nella sezione 2.2.3.4 a pagina 28 abbiamo discusso la possibilità di introdurre delle quote di storage in PAST. Questa possibilità risulta molto utile nel modello di calcolo distribuito volontario proposto, in quanto impone dei limiti di storage ai partecipanti. In sostanza, un nodo volontario ha un limite massimo di storage che non può eccedere, che è utile per prevenire situazioni in cui nodi malevoli cerchino di occupare lo storage distribuito generando risultati calcolati fittizi, atti solamente ad occupare spazio.

3.4.2. Sicurezza

Le quote di storage sono definite nelle smartcard di PAST. Possiamo immaginare che i progetti di ricerca mettano a disposizione un broker per la generazione delle smartcard, come discusso nella sezione 2.2.1 a pagina 25. Poiché un utente potrebbe esaurire la sua quota di storage producendo risultati calcolati, deve di volta in volta, reclamare lo spazio dai risultati più vecchi, oppure, impostare una durata di vita massima dei file contenenti i risultati. Tale time-out, una volta scaduto, permette di cancellare i

file vecchi e ripristinare le quote di storage utilizzate. Ovviamente, il valore del time-out deve essere tale da permettere ai nodi dedicati di raccogliere in tempo i risultati calcolati. Si osservi che questo meccanismo è necessario, in quanto solo in nodo che crea un file, ne può richiedere la cancellazione, o può impostare il suo tempo massimo di vita nello storage distribuito.

Come già discusso in precedenza nella sezione 2.2.3.2 a pagina 27, le smartcard permettono la verifica dell'autenticità dei file e delle richieste. Un nodo volontario è sicuro di eseguire le unità di lavoro rilasciate dai nodi dedicati e non da nodi malevoli grazie alle firme digitali. I nodi dedicati, possono firmare le unità di lavoro prima di pubblicarle sullo storage distribuito, in modo che ogni nodo volontario ne possa accertare l'originalità.

Un nodo volontario può rifiutare richieste di salvataggio di unità di lavoro che non presentano la firma dei progetti di ricerca a cui il volontario afferisce. Questa è una protezione in più da attacchi dove eventuali nodi malevoli si fingono nodi dedicati, e cercano di esaurire le risorse di storage della rete generando unità di lavoro fittizie.

Un nodo volontario può anche rifiutare richieste per il download dei risultati calcolati se queste non provengono dai nodi persistenti. Un'altra misura di sicurezza, per impedire il furto dei risultati già calcolati (ad esempio a scopo di ottenere credito in modo disonesto) consiste nel cifrare i risultati ottenuti con la chiave pubblica dei nodi dedicati. In questo modo solo chi possiede la chiave privata, e cioè il vero nodo dedicato, potrà ottenere i risultati.

I nodi volontari possono firmare i file contenenti i risultati calcolati prima di salvarli sullo storage distribuito. Questo meccanismo permette ai nodi dedicati di verificare l'identità dell'autore di un dato risultato, ed eventualmente afferire l'adeguata quantità di credito per il lavoro svolto. In questo modo, nessun nodo partecipante può rubare il lavoro di altri nodi volontari.

Le firme possono essere usate per verificare anche l'integrità dei dati, sia di quelli provenienti dai nodi dedicati, sia di quelli prodotti dai nodi volontari.

3.4.3. *k*-replicazione

Per garantire la persistenza[3] e la disponibilità dei file sullo storage distribuito, PAST mette a disposizione un meccanismo di replicazione con fattore k di tutti i file in esso contenuti. Come già discusso nella sezione 2.2.3.6, il sistema garantisce anche il ripristino delle copie dei file perse a causa dei fallimenti dei nodi. Questa è una proprietà desiderabile per il modello di calcolo volontario distribuito proposto, dove i nodi volontari sono assolutamente inaffidabili.

La k-replicazione può portare a problemi di consistenza nella distribuzione dei riferimenti alle unità di lavoro sullo spazio degli nodeId. Più precisamente, questo problema si presenta nelle situazioni in cui il fileId di un riferimento venga inoltrato ad un nodo con nodeId vicino al confine tra due intervalli diversi di nodeId. In questa situazione, la k-replicazione potrebbe far sì che alcune copie di un riferimento vadano su nodi con nodeId in un intervallo che non corrisponde alla tipologia di unità di lavoro puntata dal riferimento.

Per far fronte a questo problema si possono utilizzare le tecniche di deviazione delle repliche e dei file discusse nelle sezioni 2.2.4.2 e 2.2.4.3 a pagina 31. Un nodo deve controllare il fileId dei riferimenti ed assicurarsi che essi siano inclusi nel suo stesso intervallo di nodeId. Nel caso in cui un nodo riceva un riferimento con fileId non incluso nel suo intervallo di appartenenza, allora lo può tranquillamente rifiutare come se non

[3] In alternativa, o anche in combinazione con la ridondanza, si possono usare gli erasure coding descritti nella sezione 2.3 a pagina 32.

avesse abbastanza storage libero per contenerlo. A questo punto verranno attuati i meccanismi di deviazione delle repliche o dei file.

4. Implementazione di un modello di simulazione

4.1. OverSim

OverSim[27, 28] è un framework per la simulazione di reti overlay basato su OMNeT++[29, 30, 31], un ambiente di simulazione ad eventi discreti. OverSim è stato sviluppato presso l'Institute of Telematics della Karlsruhe Institute of Technology (KIT) ed è un progetto open source. I suoi punti di forza sono la flessibilità garantita dai numerosi protocolli di overlay attualmente implementati, la scalabilità (si possono effettuare simulazioni di reti composte da centinaia di migliaia di nodi) e la possibilità di riutilizzo del codice grazie alla modularità del C++.

4.1.1. Introduzione

OverSim è stato progettato, secondo il lavoro di Dabek et al. [32], per fornire una serie di funzionalità, molto utili per realizzare simulazioni di reti, che di seguito elencheremo:

Scalabilità il simulatore è in grado di effettuare simulazioni con overlay di dimensioni estese impiegando tempi ragionevoli.

Flessibilità il simulatore facilità la simulazione di reti overlay sia strutturate che non. L'utente è in grado di specificare tutti i parametri relativi alla simulazione tramite un file di configurazione umanamente comprensibile. Il simulatore dovrebbe anche fornire la possibilità di simulare la mobilità e i fallimento dei nodi, come anche comportamenti malevoli.

Modellazione della rete sottostante il simulatore fornisce modelli della rete fisica sottostante intercambiabili. Da una parte ci sono modelli che simulano una topologia di rete pienamente configurabile con bande realistiche, ritardo e perdita di pacchetto. Dall'altra ci sono modelli alternativi per la simulazione di reti con tanti nodi.

Riutilizzo del codice di simulazione il simulatore è realizzato in modo tale da permettere l'intercambiabilità di parti del codice con altre piattaforme e questo garantisce ampi margini di crescita e di sviluppo.

Statistiche il simulatore è in grado di collezionare dati statistici come: il traffico di rete inviato, ricevuto o instradato da ogni nodo, il numero di pacchetti consegnati con o senza successo, il numero di hop effettuati durante il routing, ecc.

Documentazione il simulatore è supportato da un'ampia documentazione riguardante i vari spezzoni di codice e le varie funzionalità.

Visualizzatore interattivo il simulatore fornisce una GUI che permette di visualizzare le topologie di overlay e della rete sottostante, per facilitare la validazione ed il debugging dei protocolli sviluppati. La GUI di OverSim è mostrata nella figura 4.1 nella pagina successiva.

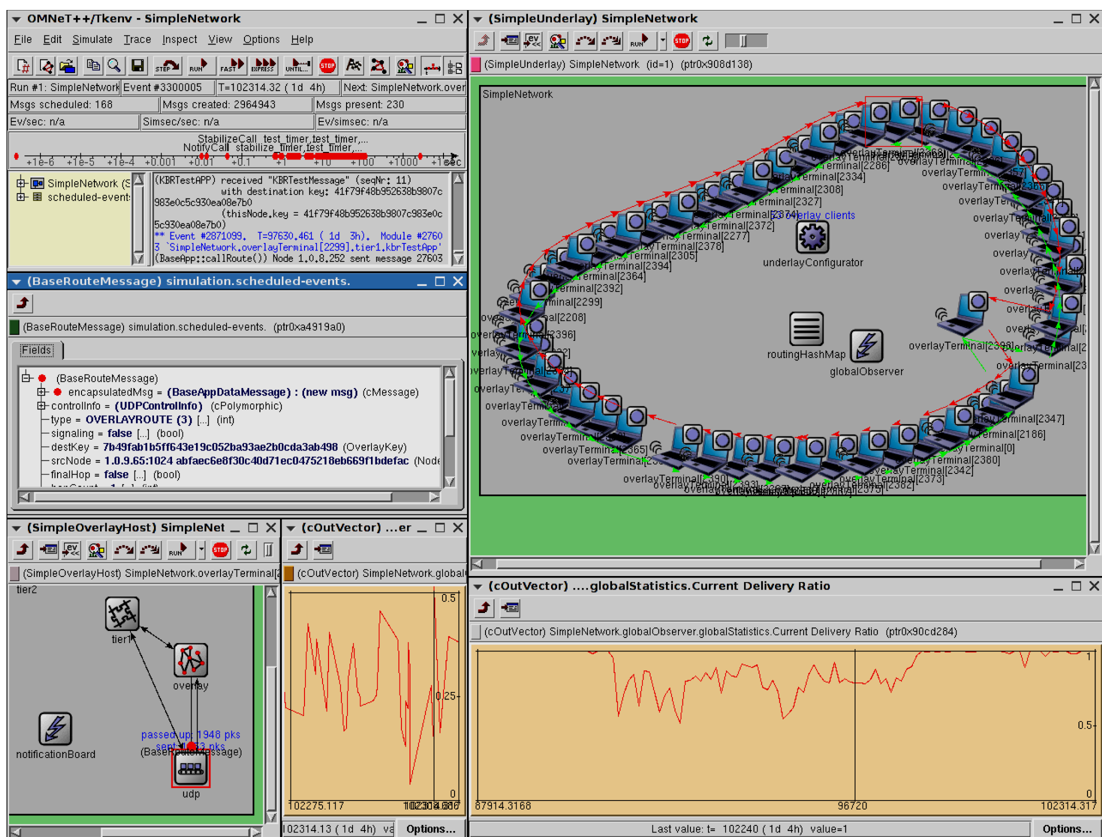

Figura 4.1.: Screenshot della GUI di OverSim.

4.1.2. Struttura di OverSim

OverSim adotta una struttura modulare a strati, secondo lo schema riportato nella figura 4.2 a fronte.

4.1.2.1. Underlay

Alla base dell'architettura di OverSim troviamo i modelli della rete fisica (underlay) sui quali si sviluppano le reti overlay. In OverSim ci sono tre modelli della rete fisica sottostante; a seconda del grado di dettaglio della simulazione che si vuole ottenere, l'utente può scegliere quale utilizzare.

Simple Underlay Il più scalabile dei tre modelli è il Simple Underlay. In questo modello i pacchetti vengono inviati da un nodo dell'overlay all'altro usando una tabella di routing globale. I pacchetti vengono ritardati tra un nodo e l'altro con un ritardo costante, oppure, per simulazioni più realistiche, con un ritardo calcolato dalla distanza tra i nodi. A questo scopo, ogni nodo viene posizionato su uno spazio euclideo bidimensionale. In più, ai nodi viene assegnato un accesso logico alla rete caratterizzato da una larghezza di banda b_n, un ritardo di accesso d_n ed una probabilità di perdita del pacchetto, così da poter simulare reti eterogenee. Il ritardo d_e che subisce il pacchetto P con lunghezza l_P tra i nodi A e B viene calcolato come: $d_e = d_A + \frac{l_P}{b_A} + c \cdot \|A - B\|_2 + d_B + \frac{l_P}{b_B}$, dove c è una costante.

In questo modo, tutti i parametri importanti della rete sottostante possono essere simulati con un solo evento di simulazione. Sì può simulare anche la mobilità dei nodi cambiando le coordinate, i parametri che caratterizzano l'accesso alla rete e l'indirizzo

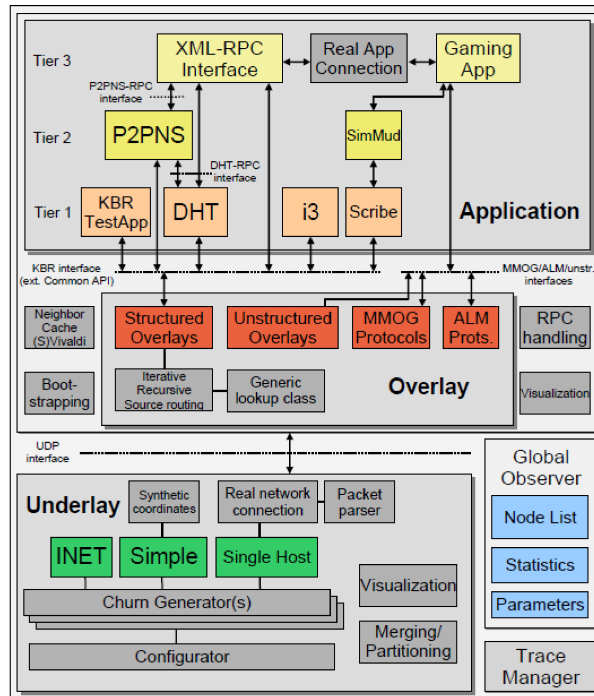

Figura 4.2.: Architettura modulare di OverSim.

IP di un nodo. Dato il poco overhead che introduce, questa tecnica permette di ottenere alti livelli di accuratezza insieme alla possibilità di simulare un gran numero di nodi.

Single Host Underlay Il modello SingleHost permette di poter riutilizzare le implementazioni dei protocolli su reti reali, senza dover modificare il codice sorgente. Con questo modello di underlay OverSim emula un singolo host, il quale si può connettere ad altre istanze su reti esistenti, anche su Internet.

Per permettere all'host emulato di comunicare su reti reali, OverSim introduce uno scheduler di eventi diverso da quello standard di OMNeT++. Diversamente dallo scheduler di default, il quale mira a produrre simulazioni le più veloci possibile, questo scheduler rallenta la velocità di simulazione a tempo reale. Se non ci sono eventi di simulazione da elaborare, legge i dati in ingresso da un dispositivo di rete TUN che permette la recezione e l'invio di pacchetti. I pacchetti ricevuti vengono convertiti in pacchetti OMNeT++ e poi passati alla "scheda di rete" del nodo simulato.

INET Underlay Il modello INET di underlay è stato creato dal framework INET[33] per OMNeT++, il quale contiene modelli di simulazione di tutti i livelli della rete a partire dal livello MAC. È utile per la simulazione di strutture contenenti sia router di backbone che di accesso. INET contiene anche un complesso modello di rete fisica per la simulazione delle reti IEEE 802.11. Dato che la INET framework originale non è ottimizzata per reti di larga scala, nel modello di underlay contenuto in OverSim sono state apportate delle modifiche per rendere più veloce l'instradamento dei pacchetti.

Il modello INET, tramite tecniche simili a quelle usate nel modello SingleHost, permette di aggiungere un modulo speciale di tipo "router" per la simulazione di scenari dove esso fa da tramite a reti reali. Questo può essere utilizzato per dimostrare la correttezza di protocolli overlay con un numero limitato di apparecchi fisici, connettendoli con un gran numero di nodi simulati in OverSim.

Tutti i modelli di rete fisica presentati hanno delle interfacce UDP/IP consistenti verso il protocollo dell'overlay. Quindi è possibile scambiare i modelli di rete sottostante in modo completamente trasparente all'overlay.

ReaSE Underlay ReaSE - Realistic Simulation Environments for IP-based Networks[34, 35] è un framework di simulazione per OMNeT++, in grado di creare ambienti di simulazione realistici con topologie di rete gerarchiche, traffico di background, e traffico di attacco.

Per realizzare queste funzionalità ReaSE implementa un'estensione dell'INET framework. I modelli di underlay generati da ReaSE sono molto realistici, ma allo stesso tempo troppo complessi per poter essere utilizzati in simulazioni con molti nodi.

Il supporto ReaSE non è incluso in OverSim, ma lo si può aggiungere come mostrato in [36].

4.1.2.2. Overlay

Sopra lo strato di underlay troviamo il *Key-Based Routing Layer* (KBR), che implementa le funzionalità di routing basato su chiave, offrendo una API comune per le diverse overlay secondo il modello presentato in [32]. Le funzionalità di base della KBR possono essere naturalmente estese o re implementate da ciascun protocollo di overlay in base alle proprie esigenze specifiche. Dal punto di vista implementativo ciò si ottiene facendo in modo che le classi che definiscono nuovi protocolli di overlay ereditino dalla classe madre BaseOverlay, i cui metodi principali sono definiti come virtuali. In particolare ciascun nuovo protocollo dovrà ridefinire i seguenti metodi:

```
NodeVector* findNode(const OverlayKey& key,
                     int numRedundantNodes, int numSiblings,
                     BaseOverlayMessage* msg = NULL);
```

il quale restituisce i `numSiblings` nodi più vicini alla chiave key nella overlay, e

```
bool isSiblingFor(const NodeHandle& node, const OverlayKey& key,
                  int numSiblings, bool* err);
```

il quale determina se il nodo `node` è fra i `numSiblings` nodi più vicini alla chiave `key`. Deve essere consistente con i risultati forniti da `findNode()`.

Queste funzioni, che caratterizzano la topologia dell'overlay in uso, vengono richiamate dalla funzione di routing basato su chiave offerta dalla KBR Common API, `sendToKey()`.

Altre funzioni da re-implementare per la realizzazione di un nuovo protocollo sono quelle relative all'inizializzazione (`initializeOverlay()`), il join (`joinOverlay()`) e la distruzione dell'overlay (`finishOverlay()`).

Si noti che, oltre alle primitive per lo scambio di messaggi, OverSim mette a disposizione meccanismi per l'invocazione di RPC, sia tramite udp (`sendUdpRpcCall()`) che attraverso la overlay (`sendRouteRpcCall()`); è altresì possibile usare le RPC per la comunicazione fra tier diversi appartenenti ad uno stesso modulo (`sendInternalRpcCall()`).

4.1.2.3. Tier Application

Al di sopra dello strato di overlay si collocano le astrazioni del Tier1, che nell'approccio proposto in Dabek et al. [32] dovrebbero fare da interfaccia fra il KBR e le applicazioni degli strati superiori, offrendo a queste ultime una serie di funzionalità aggiuntive.

OverSim implementa alcuni dei candidati al Tier1 proposti nell'articolo di Dabek et al.[32], tra cui una Distributed Hash Table (DHT), un'applicazione di test volta a

verificare il corretto funzionamento della KBR Common API (KBRTestApp), Scribe - un sistema scalabile di comunicazione di gruppo, alcune implementazioni semplificate di casi d'uso reali (RealWorldTestApp, SimpleGameClient), ecc.

Naturalmente è possibile realizzare applicazioni più complesse che facciano uso di Tier superiori.

4.1.2.4. Global Observer

All'esterno di questa struttura gerarchica troviamo alcuni moduli globali che offrono funzioni d'utilità relative alla raccolta di dati statistici e al bootstrap dei nodi della overlay. In particolare OverSim mette a disposizione un BootstrapOracle da contattare al momento del join di un nuovo nodo per ottenere l'indirizzo di un peer già appartenente alla overlay (tramite la funzione `getBootstrapNode()`); sono inoltre presenti funzioni per l'aggiunta o l'eliminazione di un nodo dalla overlay e per impostare alcuni flag speciali sui nodi della overlay (quali ad esempio `setMalicious()` per la simulazione di fault bizantini).

4.1.2.5. Churn Generators

OverSim utilizza i churn generator per simulare reti non-statiche. I diversi churn generator simulano diversi comportamenti degli utenti e possono essere configurati in modo individuale.

Tutti i churn generator usano una *init phase* (fase iniziale) dove i nodi vengono aggiunti alla rete finche non si raggiunge il limite `**.targetOverlayTerminalNum`. Mediamente ogni `**.initPhaseCreationInterval` secondi viene aggiunto un nodo.

Dopo che sono stati aggiunti tutti i nodi, inizia la cosiddetta *transition phase* (fase di transizione), la quale viene utilizzata per permettere alla rete di stabilizzarsi prima che inizino le misurazioni. La durata della transition phase è specificata dal parametro `**.transitionTime`.

Dopo la transition phase, inizia la *measurement phase* (fase di misurazione), nella quale vengono raccolte le informazioni statistiche. La durata di questa fase viene specificata nel parametro `**.measurementTime`. Alla fine della measurement phase la simulazione termina.

OverSim fornisce quattro tipi diversi di churn generator:

NoChurn I nodi vengono aggiunti alla rete finché non si raggiunge il limite dettato dal parametro `targetOverlayTerminalNum`; dopodiché la rete rimane statica.

Parametri: Nessuno.

LifetimeChurn Alla creazione di un nodo viene impostata la sua aspettativa di vita, la quale si determina in base ad una data funzione di distribuzione di probabilità. Quando un nodo raggiunge la sua aspettativa di vita, esso viene rimosso dalla rete. Un altro nodo viene in seguito creato dopo un tempo morto, estratto aleatoriamente dalla stessa funzione di distribuzione di probabilità.

Parametri: `lifetimeMean`: Aspettativa di vita media (in secondi).

`lifetimeDistName`: La funzione di distribuzione di probabilità utilizzata (default: Weibull).

ParetoChurn Simile al LifetimeChurn. L'aspettativa di vita di un nodo viene generata aleatoriamente alla sua creazione. Questo però avviene in un processo composto da due passi. Per i dettagli si veda il lavoro di Yao et al. [37].

Parametri: `lifetimeMean`: Aspettativa di vita media (in secondi).

`deadtimeMean`: Tempo morto medio (in secondi).

RandomChurn Viene generato un numero casual ad intervalli temporali fissi. In base a questo numero, viene aggiunto, rimosso o spostato un nodo a caso.

Parametri: `targetMobilityDelay`: Intervallo temporale tra due azioni consecutive (in secondi).

`creationProbability`: Probabilità di creare un nodo nuovo.

`migrationProbability`: Probabilità che un nodo venga spostato.

`removalProbability`: probabilità che un nodo venga rimosso.

Restrizioni ai parametri: la somma delle tre probabilità non deve superare 1.

Inoltre, è possibile utilizzare più churn generator in un'unica simulazione, ad esempio per simulare il comportamento di nodi che svolgono funzioni diverse all'interno della overlay.

4.2. L'implementazione del modello

Al fine di studiare il funzionamento dell'architettura descritta nel capitolo precedente, è stata realizzata un modello con OverSim. Questo modello fornisce gran parte delle funzionalità descritte in precedenza, con il pregio di non dover costruire fisicamente una rete overlay con decine di migliaia di nodi per poterne testare il comportamento.

Come ogni altro modello in OverSim, anche quello implementato in questo lavoro ha struttura modulare, la quale verrà descritta nel dettaglio nelle sezioni che seguono.

4.2.1. Overlay - Pastry

La nostra implementazione utilizza l'overlay Pastry del pacchetto OverSim. Questo modello simula Pastry nei minimi dettagli, è stato più e più volte testato per la sua correttezza, e viene utilizzato anche in altre simulazioni di applicazioni peer-to-peer Pastry-based (ad esempio possiamo citare un modello SCRIBE in OverSim).

Di seguito descriveremo brevemente come sono state implementate le componenti principali di Pastry, lasciando i dettagli al codice sorgente reperibile liberamente alla homepage di OverSim [28].

Pastry.ned Come ogni modello di OverSim, anche l'overlay Pastry viene definita in NED, come possiamo vedere nella figura 4.3 nella pagina successiva. A parte `PastryModules` (rappresentato dal contenuto del rettangono grigio in figura), che è un CompoundModule, tutti gli altri oggetti in figura sono dei SimpleModule. I loro comportamento viene definito tramite classi C++. `PastryModules` realizza l'interfacciamento dell'overlay Pastry con gli altri livelli (underlay e tier 1) definendo delle specifiche porte e connessioni, da e verso, ciascun livello.

BasePastry e Pastry Questi due moduli vengono definiti nelle corrispettive classi omonime in C++. Pastry estende BasePastry ed insieme definiscono il comportamento dell'overlay secondo le specifiche dettate in [13]. Viene implementata la API di Pastry descritta nella sezione 2.1.5 a pagina 18.

PastryRoutingTable, PastryNeighborhoodSet e PastryLeafSet Questi tre moduli implementano le tre strutture dati che definiscono lo stato del generico nodo Pastry. Tutti e tre le classi C++ corrispettive estendono la classe `PastryStateObject` che ne definisce gli elementi comuni insieme ad alcune classi di utilità.

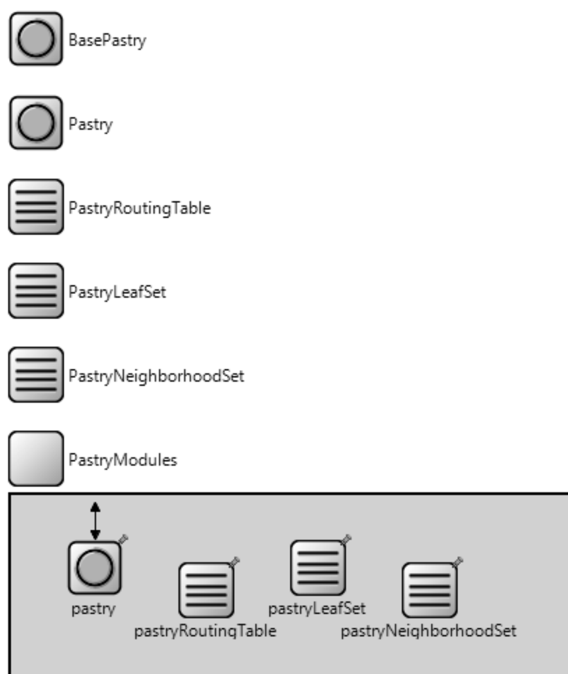

Figura 4.3.: Struttura dell'overlay Pastry in NED - Network Definition Language.

PastryMessage.msg Definisce i messaggi che il generico nodo Pastry utilizza per comunicare con gli altri nodi o con il livello superiore (tier 1). Il framework OMNeT++ permette di generare automaticamente tutte le classi C++ relative, facilitando di molto il lavoro del programmatore.

4.2.2. Tier 1 Application - SimplePAST

SimplePast costituisce un'implementazione semplificata del client di storage distribuito PAST. È composto come descritto nella figura 4.4 nella definizione NED.

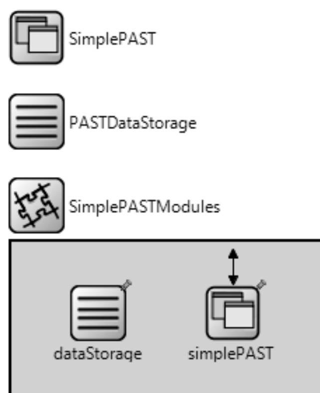

Figura 4.4.: Struttura di SimplePAST in NED - Network Definition Language.

`SimplePASTModules` è il CompoundModule che fornisce la connettività del modulo con il livello inferiore, l'overlay Pastry, ed il livello superiore, l'applicazione DHTVolComp che modella il client di un sistema di calcolo distribuito volontario.

PASTDataStorage Questo modulo implementa lo storage locale di un nodo della rete. Viene definito nella classe C++ `PASTDataStorage`, ed offre strutture dati e metodi per

la gestione dei diversi aspetti dello storage distribuito.

Ogni nodo ha una struttura dati `multimap<OverlayKey, PASTDataEntry>` nella quale vengono inserite le entry di cui il nodo è responsabile. La classe `PASTDataStorage` fornisce una serie di metodi per la gestione di questa struttura dati e delle relative entry in essa contenute. Tutti le operazioni sulle entry dello storage locale di un nodo vengono richieste esclusivamente dal modulo `SimplePAST`.

SimplePAST Il modulo SimplePAST implementa la versione semplificata di un client PAST. Funge in tutto e per tutto da client PAST, permettendo di salvare, richiedere o cancellare entry dallo storage distribuito, utilizzando il substrato Pastry per le operazioni di lookup e di routing dei messaggi.

Comunica con il substrato Pastry e con le applicazioni del tier 2 tramite RPC. Le RPC vengono gestite dai tre metodi di seguito elencati:

handleRpcCall() Metodo che gestice le richieste RPC, sia tra nodi diversi che quelle interne. In base al tipo di messaggio ricevuto il metodo può chiamare:

- **handlePutRequest()** Gestisce una richiesta di tipo PUT da un'altro nodo. Avvia i meccanismi necessari per l'eventuale inserimento di una nuova entry nello storage locale del nodo.
- **handleGetRequest()** Gestisce una richiesta di tipo GET da un'altro nodo. Avvia i meccanismi necessari per l'eventuale recupero di una entry contenuta nello storage locale del nodo.
- **handlePutCAPIRequest()** Gestisce una richiesta di tipo PUT ricevuta da un altro tier. Questo metodo gestisce RPC interne tra diversi tier dello stesso nodo.
- **handleGetCAPIRequest()** Gestisce una richiesta di tipo GET ricevuta da un altro tier. Questo metodo gestisce RPC interne tra diversi tier dello stesso nodo.
- **handleDumpPASTRequest()** Gestisce una richiesta di tipo DUMP. Questo metodo risponde con un vettore contentente tutte le entry contenute nello storage locale del nodo PAST corrente.

handleRpcResponse() Metodo che gestice le risposte RPC.

- **handlePutResponse()** Gestisce la risposta ad una richiesta di tipo PUT che abbiamo mandato ad un altro nodo in precedenza.
- **handleGetResponse()** Gestisce la risposta ad una richiesta di tipo GET che abbiamo mandato ad un altro nodo in precedenza.
- **handleLookupResponse()** Gestisce la risposta ad una LOOKUP da noi mandata in precedenza.

handleRpcTimeout() Metodo che gestice i timeout delle RPC. Opera le ritrasmissioni delle RPC non riuscite per un determinato numero di volte e notifica gli eventuali i fallimenti.

Oltre a questi metodi principali, SimplePAST implementa anche il metodo `update()` il quale aggiorna il contenuto dello storage locale di un nodo in caso di cambiamenti nei suoi vicini. Come sappiamo, l'aggiunta o il fallimento di un nodo con nodeId vicino al nodo corrente potrebbe invalidare alcune delle entry presenti nello storage locale, in quanto gli oggetti in questione potrebbero aver cambiato nodo responsabile. Per fronteggiare queste situazioni `update()` può chiamare a sua volta il metodo `sendMaintenancePutCall()` il quale invia una chiamata PUT di manutenzione.

Non sono invece implementati nel modello i meccanismi per la deviazione delle repliche e la deviazione dei file che abbiamo visto nelle sezioni 2.2.4.2 e 2.2.4.3. Essendo molto complessi e non è stato ritenuto opportuno implementarli.

Per quanto rigaurda la sicurezza, il modello non implementa il meccanismo di generazione e distribuzione delle smartcard, con tutti i meccanismi che ne derivano. Questi meccanismi impediscono ad un eventuale nodo malevolo di attuare un qualsiasi tipo di attacco, che non sia la semplice cancellazione dei dati contenuti nella propria memoria locale. Questo equivale in un certo senso al fallimento del nodo, che viene gestito dal modello.

SimplePASTMessage.msg Definisce i messaggi che il generico modulo SimplePAST utilizza per comunicare con gli altri nodi o con il tier 2.

4.2.3. Tier 2 Application - DHTVolComp

DHTVolComp è il modulo che simula il comportamento del client di calcolo distribuito volontario. Il modulo determina la funzione del nodo (se sarà volontario o dedicato) in base alla configurazione impostata e ne implementa tutte le funzionalità descritte in precedenza. Per simulare correttamente gli overhead di comunicazione tra i nodi e possibile impostare la dimensione delle unità di lavoro e dei file di risultati secondo i valori tipici che possiamo trovare nei progetti attualmente attivi di calcolo distribuito volontario. In questo simulatore, il contenuto dei pacchetti (work unit e risultati) è costituito da semplice padding. I nodi volontari simulano l'esecuzione dei task tramite attese, alla fine delle quali il lavoro viene considerato completato.

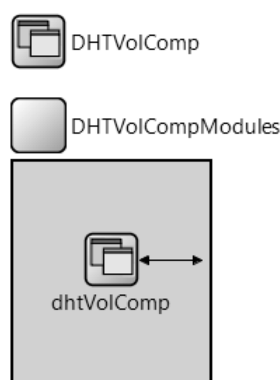

Figura 4.5.: Struttura di DHTVolComp in NED - Network Definition Language.

I metodi principali di questo modulo sono:

handleRpcResponse() Metodo che gestice le risposte RPC, tramite le quali avviene la comunicazione tra i diversi tier dello stesso nodo.

handleGetResponse() Metodo chiamato quando il nodo riceve una GET response a fronte di una richiesta di tipo GET da esso inviata in precedenza. Gestisce i dati ottenuti in risposta come segue: (1) se la GET response contiene una unità di lavoro, essa viene inserita in coda alle altre unità di lavoro presenti sul nodo. Se il nodo è idle, viene eseguita l'unità di lavoro appena ottenuta, altrimenti essa dovrà attendere in coda finché non arriverà il suo torno; (2) se la GET response contiene un riferimento, allora essa viene messa in coda ai riferimenti non ancora utilizzati. Quando il nodo termina i task da eseguire sfrutterà i riferimenti che ha già in memoria prima di richiederne altri; (3)

infine, se la GET response contiene un risultato (avviene solo quando il nodo che fa la richiesta è un nodo dedicato), questo viene rimosso dalla lista dei risultati pendenti.

handlePutResponse() Metodo chiamato quando il nodo riceve una PUT response a fronte di una richiesta di tipo PUT da esso inviata in precedenza, determinando se l'operazione ha avuto buon fine.

handleDumpResponse() Questo metodo gestisce le risposte alle richieste di DUMP che il modulo DHTVolComp invia al modulo SimplePAST. Le richieste di DUMP permettono di ottenere informazioni sui dati contenuti nella porzione del file system distribuito PAST contenuta nella memoria locale del nodo. DHTVolComp utilizza le DUMP request verso il tier sottostante per vedere se esso contiene unità di lavoro o riferimenti ad unità di lavoro che il client può eseguire, prima di inviare richieste di riferimenti ad altri nodi.

handleTimerEvent() Questo metodo è il cuore del funzionamento del modulo DHTVolComp. Gestisce tramite messaggi e auto-messaggi eventi come l'invio delle unità di lavoro, la raccolta dei risultati, l'esecuzione dei task, la generazione dei risultati, la ricerca di riferimenti a unità di lavoro e le richieste di unità di lavoro.

4.2.4. Global Observer - GlobalDHTVolComp

Il modulo Global Observer funge da osservatore globale dello stato della rete. Fornisce informazioni dettagliate su molti aspetti del funzionamento della rete nel suo complesso. Permette di avere stime attendibili sul numero di pacchetti scambiati, su eventuali errori di comunicazione, sullo stato globale dello storage PAST e sui tempi impiegati per lo svolgimento delle diverse fasi di funzionamento del sistema.

Questo modulo non ha un riscontro nel modello reale, in quanto sarebbe molto difficile da realizzare un osservatore onnivedente, però si rivela molto utile ai fini della raccolta dati per la valutazione delle prestazioni del modello simulato. Al suo interno vengono implementate diverse strutture dati, i contenuti delle quali possono essere analizzati tramite appositi strumenti forniti dal framework OverSim per l'ottenimento di informazioni statistiche.

5. Risultati e conclusioni

In questo capitolo presentiamo i risultati delle sperimentazioni atte a dimostrare le proprietà ed il corretto funzionamento del modello di calcolo distribuito volontario proposto. Al fine di ottenere dei risultati attendibili, per le simulazioni sono stati utilizzati parametri tratti da statistiche sui progetti di calcolo distribuito volontario esistenti. Di seguito vengono elencati i principali test eseguiti con i paramentri utilizzati ed i relativi risultati.

5.1. Simulazioni per valutare il modello di calcolo distribuito volontario

Per valutare il modello di calcolo distribuito volontario proposto nei capitoli precedenti si possono utilizzare diverse tecniche, anche se in questo lavoro si è ricorso principalmente alle simulazioni. Si potevano utilizzare modelli analitici o addirittura l'implementazione di un progetto di calcolo distribuito volontario vero e proprio. Sono state preferite le simulazioni perché comportano dei benefici significativi rispetto agli altri due metodi, e gli svantaggi sono più tollerabili rispetto a quelli che si avrebbero con gli altri due metodi.

Ci sono diversi vantaggi nell'utilizzo delle simulazioni. Le simulazioni portano a risultati facilmente riproducibili, mentre l'implementazione di un sistema di calcolo distribuito volontario vero e proprio di certo non garantisce questa proprietà. Inoltre, le simulazioni richiedono molta meno cooperazione da altri rispetto all'implementazione di un progetto di calcolo distribuito volontario. Mentre le simulazioni possono facilmente essere eseguite più e più volte, è praticamente impossibile far sì che i volontari partecipanti ad un progetto di ricerca scarichino ed installino un nuovo client per ciascuna configurazione da testare.

Ovviamente ci sono degli svantaggi che comporta l'utilizzo delle simulazioni. Esse non possono catturare i dettagli di basso livello, facilmente rilevabili da una reale implementazione. Vengono omessi fattori come la congestione della rete oppure gli effetti derivanti dall'interazione del sistema operativo con il software client di calcolo distribuito. Quindi, i risultati delle simulazioni non potranno mai essere accurati come quelli ottenuti implementando un progetto reale di calcolo distribuito volontario.

I modelli analitici spesso utilizzano distribuzioni che non riflettono la realtà in modo accurato. Inoltre, le simulazioni possono avere un alto grado di dettaglio senza diventare inpraticabili come i modelli analitici.

Uno svantaggio delle simulazioni consiste nel fatto che esse possono richiedere un tempo molto lungo per essere eseguite. Mentre con un modello analitico, inserendo i valori desiderati nelle corrispettive variabili, si potrebbero eseguire velocemente i calcoli desiderati. Le simulazioni possono introdurre molte possibilità di commettere errori durante la loro implementazione, cosa che avviene più difficilmente nella creazione di un modello analitico.

Per rendere un modello analitico il più possibile vicino alla realtà è necessario che esso includa informazioni sui calcolatori che partecipano ai progetti di calcolo distribuito volontario, e questo è molto difficile da realizzare in pratica. Infatti, l'elevato numero

di variabili in gioco non permette di creare modelli che sono allo stesso tempo affidabili e maneggevoli.

Per questi motivi in questo lavoro sono state utilizzate le simulazioni per valutare la qualità del modello proposto. Tutte le simulazioni all'interno di questo lavoro sono state eseguite su una macchina con processore Intel® Core™ i7-930 (8M Cache, 2,80 GHz, 4,80 GT/s Intel® QPI) e 6GB di RAM (3×2GB Kingston® HyperX Blu DDR3-1600 Tripple-channel), con sistema operativo Ubuntu 12.04 (Precise Pangolin) - Desktop Edition da 64-bit appositamente impostato (sono stati disattivati la maggior parte dei processi che eseguono in background).

5.2. Configurazione delle simulazioni

Come abbiamo già visto, il framework OverSim permette di simulare la volatilità dei nodi partecipanti ad una rete con diversi modelli probabilistici. Il pool degli host per i progetti di calcolo distribuito volontario è molto dinamico: gli host si aggiungono e lasciano la rete in continuazione. In più gli utenti occasionalmente reimpostano il software dei loro calcolatori, il che equivale alla creazione di un nuovo client dal punto di vista dei server. Il "tempo di vita" (l'intervallo compreso dal momento della creazione del client fino all'ultima comunicazione con i server del progetto) misura la durata della validità dell'Id che identifica univocamente i client dal punto di vista del server. La re-installazione del software client comporta la creazione di un Id nuovo, e, per il server questo equivale all'adesione di un nuovo client al sistema. La re-installazione del software client comporta la creazione di un Id nuovo, e, per il server questo equivale all'adesione di un nuovo client al sistema. Gli host dei progetti di calcolo distribuito volontario hanno un tempo di vita medio di 91 giorni con distribuzione esponenziale [38] (vedi figura 5.1). Questa statistica è molto importante, soprattutto per i progetti che producono unità di lavoro che necessitano di periodi di tempo molto lunghi per essere elaborate.

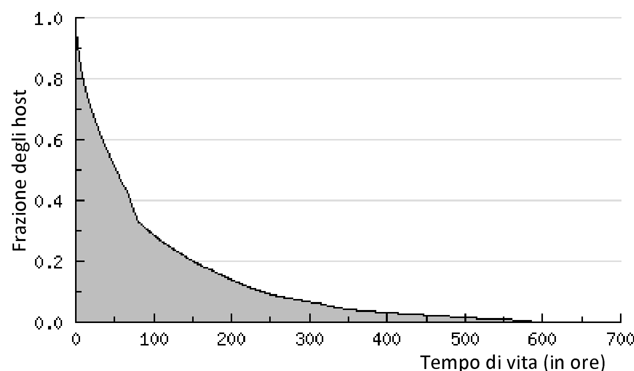

Figura 5.1.: Distribuzione del tempo di vita degli host.

OverSim non fornisce direttamente la possibilità di utilizzare variabili aleatorie esponenziali per modellare la volatilità dei nodi. Permette invece di utilizzare una variabile aleatoria di Weibull per modellare tale fenomeno. La variabile aleatoria di Weibull ha la seguente funzione di densità di probabilità: $f(x) = \frac{k}{\lambda^k} x^{k-1} e^{-(\frac{x}{\lambda})^k}$. Si nota immediatamente la somiglianza con la densità di probabilità della variabile aleatoria esponenziale: $f(x) = \gamma e^{-\gamma x}$. Infatti, impostando il parametro $k = 1$ e $\lambda = \frac{1}{\gamma}$ la variabile aleatoria di Weibull coincide con una variabile aleatoria esponenziale di parametro $\frac{1}{\lambda}$. Quindi OverSim è in grado di simulare correttamente questo comportamento. Il tempo di vita

dei nodi è un parametro invariante nelle nostre simulazioni, in quanto rappresenta una caratteristica invariante del sistema.

La velocità di comunicazione tra i nodi è determinata dal modulo Simple Underlay Configurator del simulatore. Poiché questo modulo simula le tipiche velocità di comunicazione e probabilità d'errore che si hanno in Internet, si è ritenuto opportuno non apportare modifiche ai suoi parametri. I parametri variabili delle simulazioni sono:

- il tempo di completamento dei task;
- la dimensione delle unità di lavoro (in MB).

Il tempo di completamento dei task è stato modellato come una variabile aleatoria normale (come suggerito in [39]) troncata ai valori positivi, con parametri (media e deviazione standard) tratti da progetti di calcolo distribuito volontario esistenti [40, 41].

Nella tabella 5.1 vengono elencati i parametri utilizzati:

Progetto	Durata media dei task (in ore)	Deviazione standard (in ore)
WCG FightAIDS@Home	7,7	4,6
WCG Genome Comparison	1,2	0,9
Poem@Home MFold	2,1	1,4
Poem@Home CHARMM	1,6	1,0

Tabella 5.1.: Durata media e deviazione standard dei task nei progetti utilizzati in questo lavoro.

Sono state utilizzate diverse dimensioni per le unità di lavoro, anch'esse tratte da progetti esistenti [41]. Tali valori variano da poche centinaia di kilobyte a qualche decina di megabyte.

Dai test eseguiti è emerso che le variazioni nelle dimensioni delle unità di lavoro secondo i limiti descritti in [41] non influiscono in modo percepibile sulle prestazioni del sistema. Questo è un risultato abbastanza ragionevole dato che: (1) tutte le unità di lavoro hanno dimensioni piccole (10MB o meno); (2) le velocità di download dei nodi sono relativamente elevate; (3) i task impiegano diverse ore per essere completati; ed infine (4) i calcolatori sono costantemente connessi ad Internet. In base a queste assunzioni notiamo che per scaricare un'unità di lavoro è necessario un tempo significativamente minore a quello necessario per calcolare il relativo risultato, e che quindi, un client che utilizza il buffering dei task, o che inizia a scaricarne uno mentre ne sta già eseguendo un altro, avrà sempre unità di lavoro disponibili da elaborare.

Inoltre si assume che il grado di parallelismo dei calcolatori moderni permetta di poter compiere dell'altro lavoro mentre si scarica un file. Se così non fosse, allora un calcolatore si congelerebbe e non permetterebbe nessun tipo di operazione ogniqualvolta si scarichi un file. Si assume che la frazione di tempo della CPU necessario per realizzare il download di un file sia molto piccola rispetto al tempo totale, e che quindi il download di un file non tolga molti cicli di CPU dalla unità di lavoro correntemente in elaborazione.

Questo è coerente con quanto osservato nei progetti di calcolo distribuito volontario attualmente in uso, e con i risultati delle simulazioni eseguite.

5.3. Risultati

Di seguito vengono riportati alcuni dei risultati ottenuti dai test eseguiti col simulatore. Sono stati eseguiti diversi test, modificando di volta in volta i parametri più significativi, che sono la durata media dei task con la relativa deviazione standard, ed il numero

medio di nodi volontari. La variazione delle dimensioni delle unità di lavoro, entro i limiti descritti in precedenza, non ha mostrato un impatto percepibile sulle prestazioni del sistema.

Inoltre, non sono state prese in considerazione le deadline attualmente vigenti per il completamento delle unità di lavoro, in quanto esse sono un parametro che dipende sia dalla particolare applicazione scientifica, sia dalle caratteristiche del sistema, ed in quanto tale, va tarato di conseguenza. Utilizzare delle deadline appositamente pensate per un sistema centralizzato sul modello proposto sarebbe poco significativo.

Sono state eseguite diverse simulazioni, con durata dei task e relativa deviazione standard come descritte nella tabella 5.1. Durante la simulazione vengono misurati diversi parametri, la maggior parte dei quali serve per controllare la correttezza del procedimento e dell'implementazione (debugging). I due parametri più significativi sono: il numero di task completati in un intervallo di durata Δ (il throughput del sistema); ed il numero di task in esecuzione misurato ogni Δ. Δ è stato scelto pari alla durata media dei task. Questo perché il numero di task completati in un intervallo di lunghezzaΔ, con $\Delta = durata\,media\,dei\,task$, fornisce una stima dello speed-up del sistema ad ogni istante di campionamento. Inoltre, dato che Δ dipende dalla durata media dei task, le due misure sopra indicate dipenderanno poco dalla tipologia di task scelti per l'esperimento. Infatti, anche passando da task con durata media di 1,2 ore e deviazione standard 0,9 ore, a task con durata media di 7,7 ore e deviazione standard 4,6 ore, la forma del grafico che si ottiene rimane pressoché invariata.

Per avere una stima delle prestazioni del sistema a regime, i nodi non iniziano a cercare e/o svolgere i task finché quest'ultimi non sono stati completamente distribuiti sulla rete dal nodo dedicato. Inoltre, le simulazioni vengono fermate quando il 90% dei task viene completato. Questo perché, a questo punto delle simulazioni, il numero di task disponibili diventa piccolo rispetto al numero di nodi presenti nel sistema, e di conseguenza i nodi avranno più difficoltà a trovare i task da eseguire. Notiamo però che una situazione simile è molto rara in pratica, dato che il numero di task da eseguire può essere mantenuto più o meno costante dai nodi dedicati, i quali, ad ogni risultato che raccolgono, potrebbero immettere nel sistema un nuovo task. Inoltre, il numero di task da eseguire è praticamente sempre maggiore del numero di nodi volontari disponibili. Una situazione diversa da questa porterebbe ad un utilizzo non ottimale di qualsiasi sistema di calcolo distribuito volontario, non solo del modello proposto in questo lavoro, ma anche di quelli attualmente in uso. Quindi le misurazioni sono state fatte in condizioni di carico massimo del sistema.

Nel grafico 5.2 a fronte viene mostrato il numero di task in esecuzione ed il numero di task completati con un sistema composto da 1 000 nodi che devono svolgere 50 000 task.

Dal grafico 5.2 nella pagina successiva si può notare che il numero di task in esecuzione in ogni momento è vicino al numero medio di nodi, cosa che indica un'alta utilizzazione del sistema. Il numero di task completati in ogni intervallo è di poco inferiore al numero di task in esecuzione all'istante di misurazione precedente. Questo perché la durata dei task viene modellata come una variabile aleatoria normale (ovviamente troncata ai valori positivi) con una deviazione standard abbastanza ampia rispetto alla media, e quindi le durate dei task hanno valori abbastanza sparsi.

Se si eseguono delle simulazioni con altri parametri (ad esempio con durata media dei task di 7,7 ore e deviazione standard di 4,6 ore) il grafico ottenuto non cambia in maniera apprezzabile. Invece, se eseguiamo dei test scegliendo una deviazione standard molto più piccola della durata media dei task (ad esempio di un ordine di grandezza più piccola), otteniamo un risultato un po' diverso dal precedente, come mostrato nel grafico 5.3 a fronte.

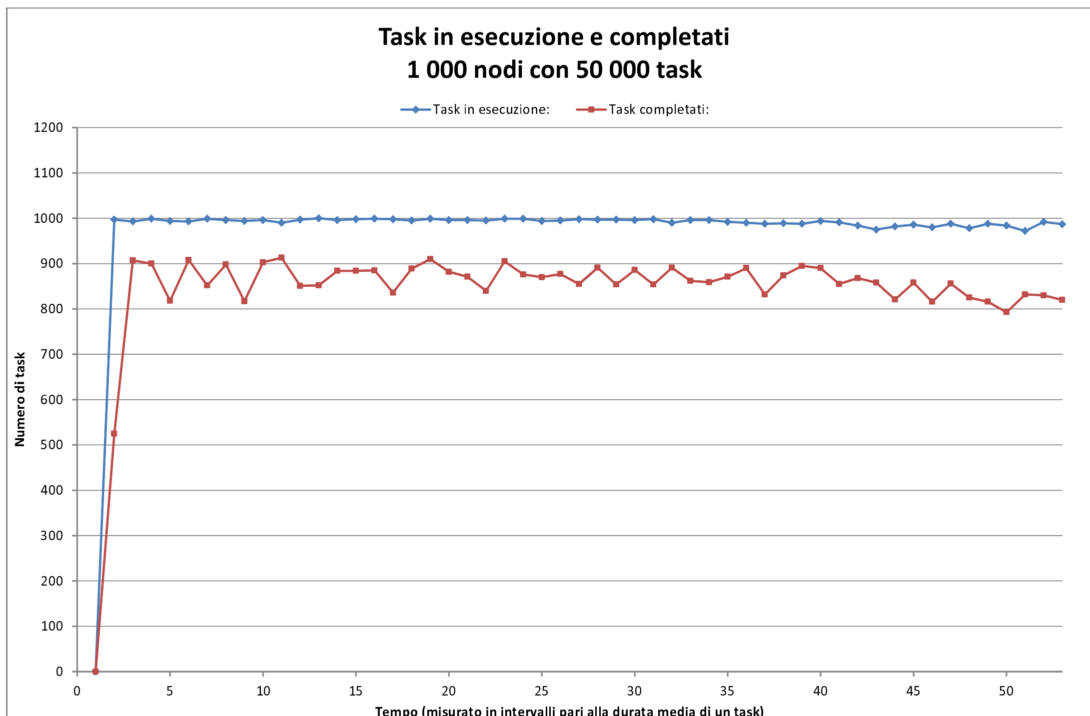

Figura 5.2.: Task in esecuzione e completati. Il sistema è composto da 1 000 nodi. I task hanno durata media di 1,2 ore e deviazione standard di 0,9 ore.

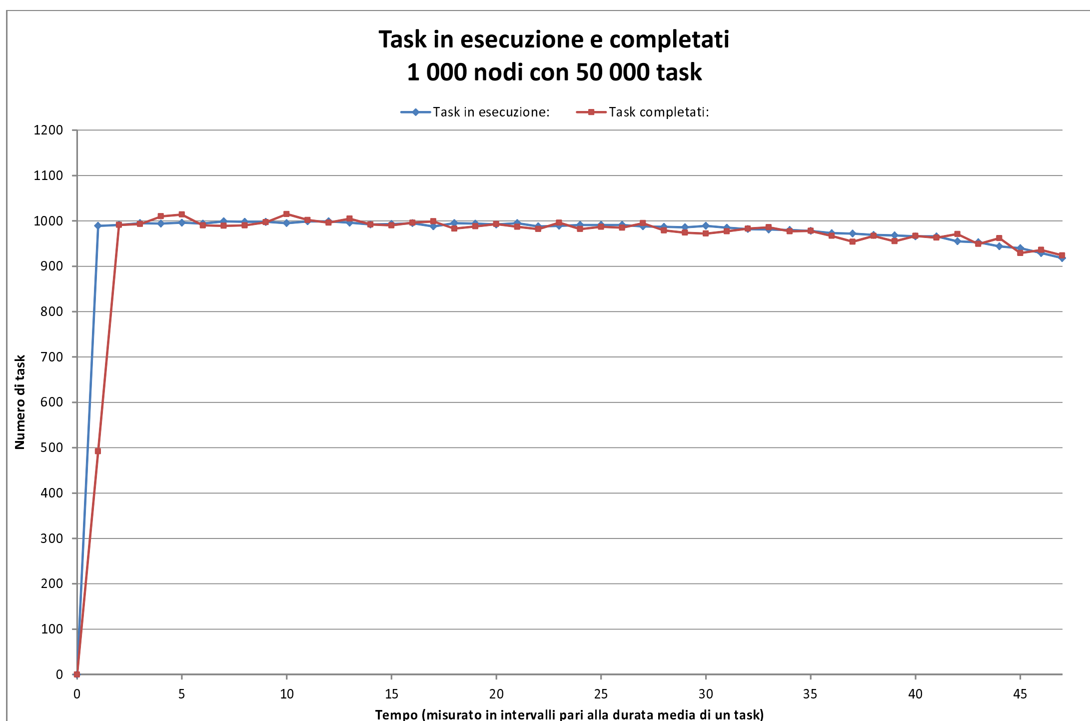

Figura 5.3.: Task in esecuzione e completati. Il sistema è composto da 1 000 nodi. I task hanno durata media di 2 ore e deviazione standard di 0,2 ore.

Notiamo immediatamente che la curva che descrive i task completati è, a meno di piccolissime differenze, la traslazione di un intervallo in avanti della curva dei task in esecuzione. Questo è facilmente spiegabile dal fatto che abbiamo scelto una deviazione standard molto piccola rispetto alla durata media dei task, e quindi, la maggior parte dei task in esecuzione ad un certo punto verrà completata entro un intervallo di durata pari alla durata media dei task.

Va precisato però, che una situazione simile non si presenta mai in pratica. Nessuno tra i progetti di ricerca che utilizzano il calcolo distribuito volontario ha task con durata caratterizzabile come una variabile aleatoria normale con una deviazione standard di un ordine di grandezza più piccola rispetto alla media. Questo particolare esempio è stato riportato per dimostrare che il numero di task completati è più piccolo del numero di task in esecuzione all'intervallo precedente per effetto della deviazione standard. Con una deviazione standard piccola, le due curve sono di fatto molto simili.

Portando a 2 000 il numero di nodi volontari aumenta anche il numero di task in esecuzione in ogni istante e il numero di task completati in ogni intervallo (chiaramente quando il sistema è a regime). Nei grafici 5.4, 5.5 e 5.6 a fronte vediamo chiaramente questo andamento. Poiché nella simulazione abbiamo mantenuto fisso il numero di task a 50 000, il sistema raggiunge la soglia del 90% di task completati più velocemente, e di conseguenza il grafico è composto da meno campioni rispetto al precedente.

Anche aumentando ulteriormente il numero di nodi e di task l'andamento del sistema è analogo. Aumenta il numero di task in esecuzione e di task completati ad ogni intervallo, proporzionalmente al numero di volontari. Il numero di task in esecuzione, durante il funzionamento a regime, è molto vicino al numero di nodi volontari che compongono il sistema. Chiaramente ci sono delle piccole fluttuazioni come conseguenza della volatilità dei nodi volontari. Ricordiamo che il tempo di vita dei nodi è una variabile aleatoria, e che quindi, ad ogni istante, con una certa probabilità, si aggiungono e si allontanano nodi dal sistema.

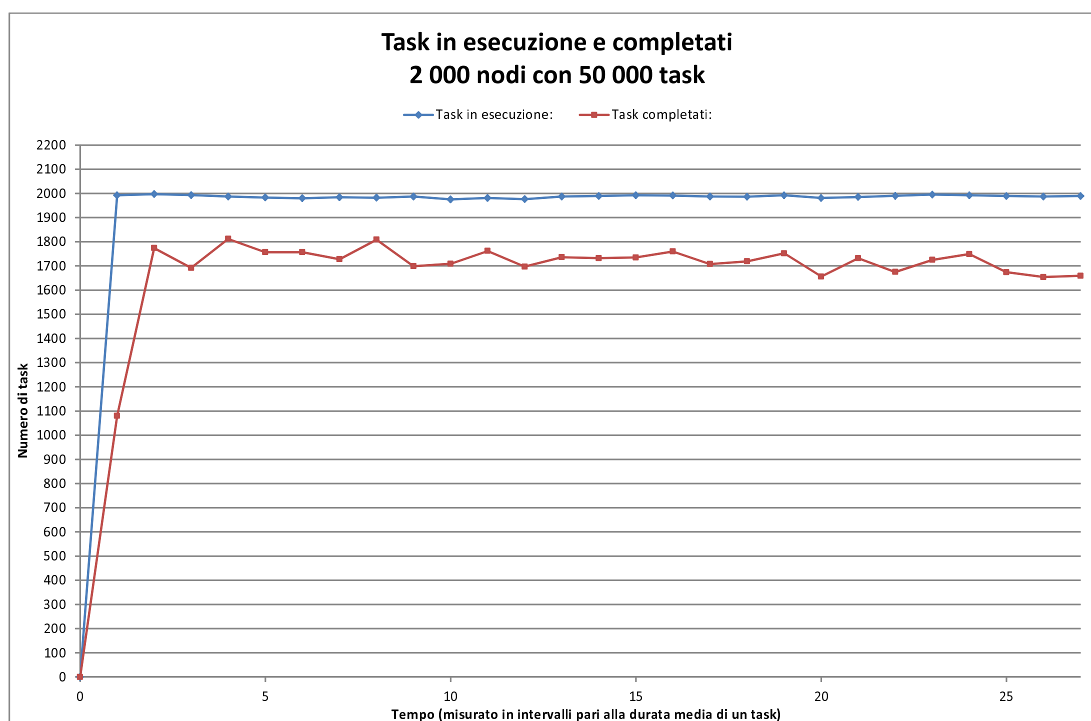

Figura 5.4.: Task in esecuzione e completati. Il sistema è composto da 2 000 nodi. I task hanno durata media di 1,2 ore e deviazione standard di 0,9 ore.

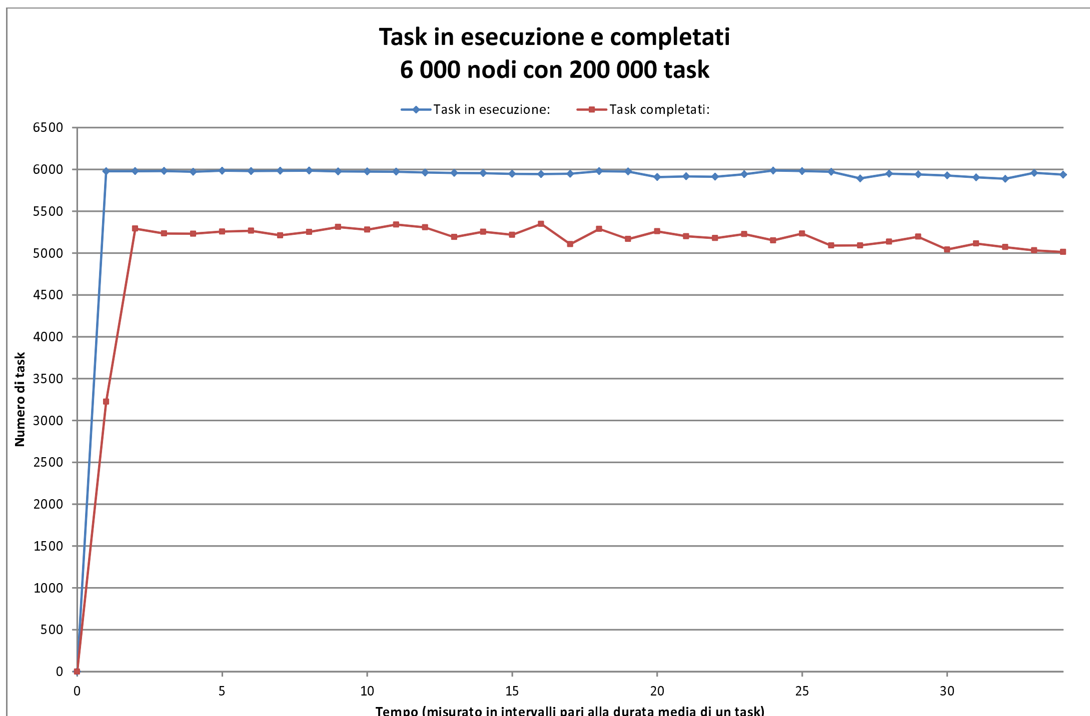

Figura 5.5.: Task in esecuzione e completati. Il sistema è composto da 6 000 nodi. I task hanno durata media di 1,2 ore e deviazione standard di 0,9 ore.

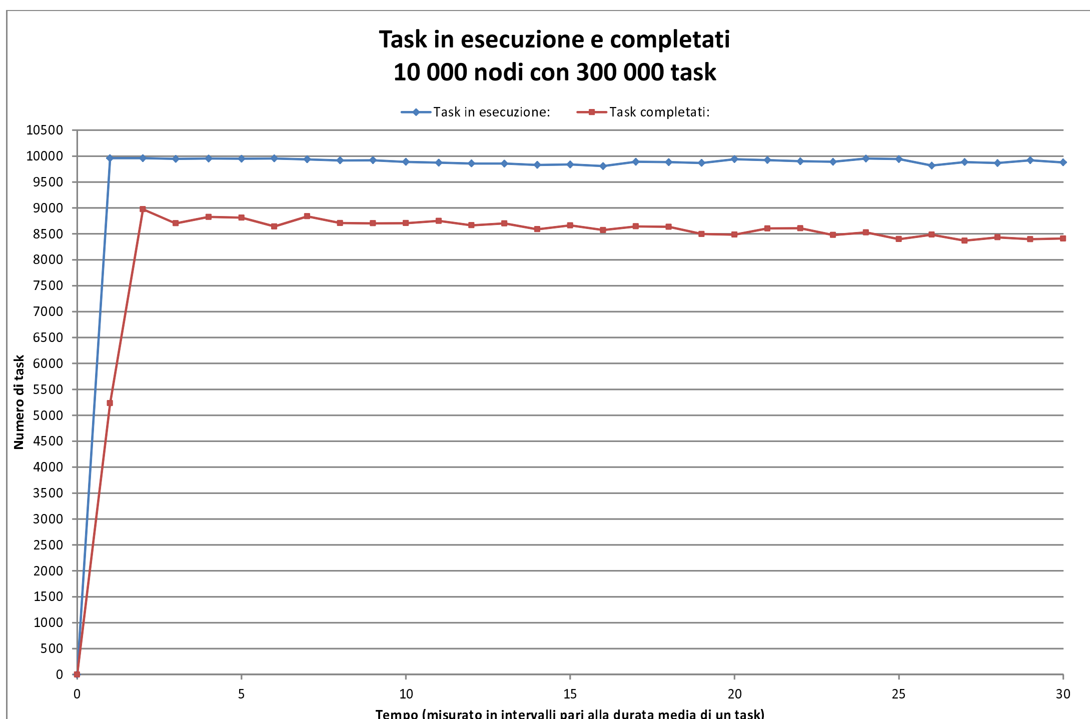

Figura 5.6.: Task in esecuzione e completati. Il sistema è composto da 10 000 nodi. I task hanno durata media di 1,2 ore e deviazione standard di 0,9 ore.

Nel grafico 5.7 possiamo vedere una misura dello speed-up del sistema (i valori esatti sono sulla tabella 5.2). Lo speed-up è stato calcolato come numero medio di task completati in un intervallo Δ (con $\Delta = durata\,media\,dei\,task$) quando il sistema è a regime. Lo speed-up aumenta linearmente con l'aumentare del numero medio di nodi volontari presenti nel sistema. Questo indica che l'overhead di comunicazione introdotto dall'overlay Pastry/PAST non ha un grosso impatto sulle prestazioni del sistema. Chiaramente questo è vero finché la durata media dei task è relativamente lunga rispetto ai tempi di ricerca/download delle unità di lavoro, ma fortunatamente questo sembra essere vero in praticamente tutti i progetti di calcolo distribuito volontario attualmente esistenti.

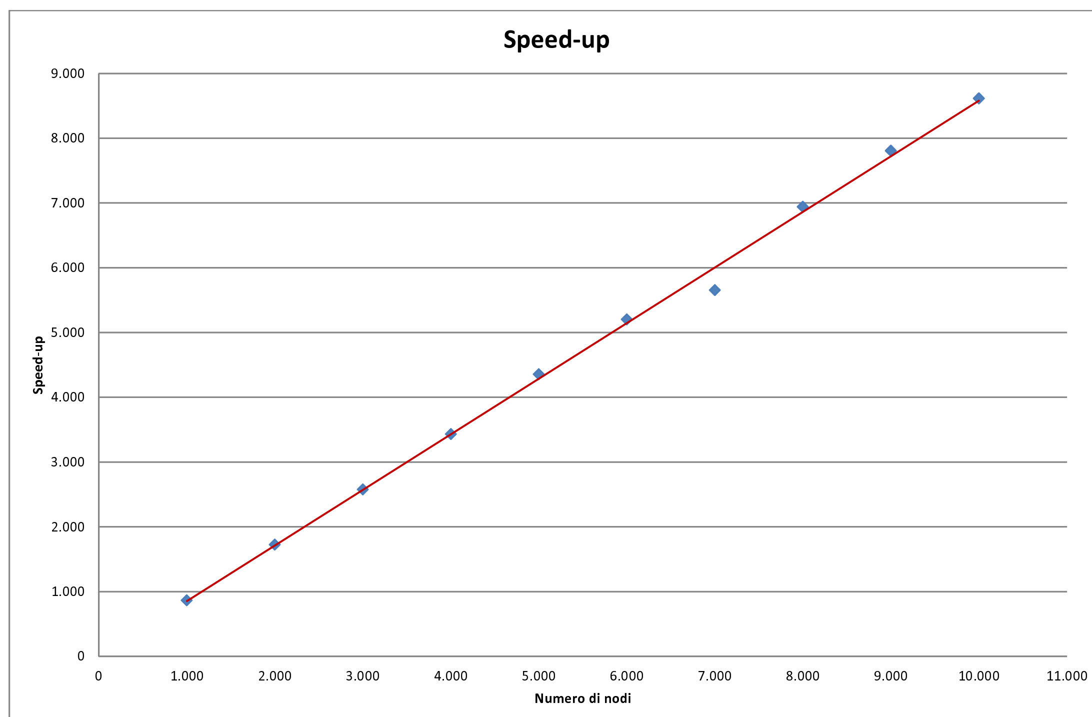

Figura 5.7.: Speed-up in funzione del numero di nodi del sistema.

Numero di nodi	Speed-up
1 000	864
2 000	1 725,5
3 000	2 577,2
4 000	3 430
5 000	4 356,4
6 000	5 200,9
7 000	5 653,6
8 000	6 941,8
9 000	7 808,3
10 000	8 614,8

Tabella 5.2.: Speed-up in funzione del numero di nodi del sistema.

Sono stati eseguiti anche alcuni test di integrità per garantire che il simulatore venisse eseguito correttamente. I test di integrità sono stati progettati per mostrare che la regolazione dei parametri ha avuto un effetto corretto sui risultati della simulazione.

Sono state eseguite alcune simulazioni con diverso numero di nodi, task e tempo di completamento dei task. Poi sono state ri-eseguite le simulazioni raddoppiando il tempo di completamento dei task, ottenendo, nello stesso intervallo temporale, il completamento di circa metà dei task rispetto alle prime simulazioni. Come atteso, le differenze nei tempi di completamento dei task producono grandi differenze nel throughput del sistema, inteso come numero di task completati in un certo intervallo temporale. I risultati dei test di integrità non sono stati riportati.

5.4. Conclusioni

I risultati ottenuti potrebbero non valere per parametri al difuori dei limiti testati. Si è assunto che i partecipanti ai progetti di calcolo distribuito volontario avessero connessioni veloci, invece di connessioni con modem a 56 kbps o con linee cellulari a bassa velocità. L'utilizzo di connessioni lente potrebbe cambiare i risultati delle simulazioni, però è improbabile che reti cellulari e modem a 56 kbps vengano utilizzati per partecipare ai progetti di calcolo distribuito volontario. In genere, le persone non sono disposte a scaricare le batterie delle loro apparecchiature wireless, come i cellulari o i PDA, e quindi non ci si aspetta che le reti cellulari vengano utilizzate in questo ambito. Inoltre, la maggior parte dei partecipanti ai progetti di calcolo distribuito volontario è composta da persone relativamente esperte nell'utilizzo della tecnologia, le quali hanno connessioni Internet ad alta velocità.

Le simulazioni sono state realizzate con un numero di nodi variabile dai mille ai diecimila. Oggigiorno alcuni progetti di calcolo distribuito volontario contano centinaia di migliaia di nodi partecipanti. Sfortunatamente, simulare una rete di tali dimensioni richiede un calcolatore molto potente, in termini di velocità del processore, ma soprattutto in termini di memoria RAM disponibile. Anche se la macchina utilizzata per eseguire i test presenta buone caratteristiche prestazionali, per simulare reti composte da più di 10 000 nodi serve molta più memoria RAM. Ogni nodo conserva un suo stato, e questo ha un certo costo in termini di memoria.

Anche con molteplici ottimizzazioni del codice, non si è stati in grado di superare il limite dei 10 000 nodi per simulazione, dato che il calcolatore andava in paginazione. Una simulazione con 10 000 nodi necessità di circa 7 ore per essere completata, e viene simulata con una velocità di circa 350 secondi di simulatore al secondo. Una rete con 100 000 nodi, prima che il calcolatore vada in paginazione, viene simulata con una velocità di circa 7-8 secondi di simulatore al secondo, per diminuire durante la paginazione a 0,001 secondi di simulatore al secondo. Diventa chiaramente impossibile portare a termine una simulazione in tempi ragionevoli in queste condizioni.

Inoltre, la mancanza di informazioni statistiche sullo speed-up di progetti esistenti di calcolo distribuito volontario ha reso impossibile il confronto prestazionale del modello proposto con questi ultimi. Infatti, le uniche misure di prestazioni che si trovano in letteratura fanno riferimento al numero stimato di FLOPS di cui un dato progetto dispone, dalla quale è impossibile trarre informazioni di confronto che non siano fortemente project-specific.

Per questi motivi non è stato possibile fornire delle misure di confronto tra i due modelli. Inoltre, sarebbe stato molto significativo un confronto sul numero di unità di lavoro e di risultati che vengono perduti come conseguenza della volatilità dei nodi volontari, sia nei modelli esistenti che in quello proposto. Ma tale scopo è difficilmente raggiungibile, poiché bisognerebbe implementare lo stesso progetto di calcolo distribuito volontario su entrambi i modelli, e fare in modo che i client di entrambe le implementazioni vengano installati da (più o meno) lo stesso numero di volontari, e tutto a scopo di test. Chiaramente, questa possibilità è infattibile, sia per il tempo necessario all'im-

plementazione di un progetto di calcolo distribuito volontario, sia per la difficoltà nel trovare l'adeguato numero di volontari disposti a partecipare all'esperienza.

Le prestazioni delle DHT in generale, e di Pastry in particolare, dipendono dal numero di nodi partecipanti alla rete. Le prestazioni in questo caso sono intese come numero di hop di routing che un messaggio deve percorrere per essere inoltrato ad un nodo con una data chiave. Chiaramente, sono affette da questo overhead solo le richieste di lookup, invece i trasferimenti dei file non risentono variazioni. Questo perché, una volta trovato il nodo destinazione per una data richiesta (che sia di tipo GET o PUT), il nodo richiedente inizializzerà una connessione diretta col nodo destinazione, ed il trasferimento dei dati avverrà direttamente tra questi due. Quindi possiamo affermare che l'overhead che nasce come conseguenza della crescita di dimensioni della rete affligge solo le richieste di lookup.

Fortunatamente, i messaggi di questo tipo sono piccoli, dato che contengono soltanto poche informazioni come in nodeId del mittente, il suo indirizzo IP, il nodeId del destinatario e qualche altro campo, sempre di poco peso in termini di dimensioni del pacchetto. Di conseguenza questi pacchetti viaggiano veloci, non soffrono di problemi come la frammentazione IP, e l'overhead generato dal loro inoltro è generalmente molto piccolo. Sicuramente molto più piccolo del tempo necessario per trasferire un'unità di lavoro da un nodo sorgente ad una certa destinazione.

Poiché abbiamo visto che i tempi di trasferimento sono in pratica trascurabili se confrontati con i tempi di esecuzione dei task, possiamo tranquillamente tralasciare anche i ritardi che subiscono le richieste di lookup come conseguenza della crescita della rete. Sotto questa plausibile ipotesi, possiamo confidare sul fatto che l'overhead introdotto dalla overlay Pastry/PAST sarà trascurabile rispetto al tempo di completamento dei task, anche quando si hanno a disposizione moltissimi nodi.

Le simulazioni svolte in questo lavoro dimostrano la correttezza del modello proposto, e danno dei risultati incoraggianti sotto ipotesi ragionevoli. Il passo successivo a questo lavoro consiste nell'implementazione di un progetto di prova basato sul modello proposto, e nella sua successiva distribuzione su internet, in modo che utenti volontari lo possano installare. Questo è un obiettivo molto ambizioso, che necessiterà sicuramente grandi moli di lavoro e tempo per la parte implementativa, ma soprattutto perché venga installato dai volontari e raggiunga una diffusione tale da fornire informazioni sperimentali significative.

A. Appendice - Calcolo Distribuito Volontario per la Bioinformatica

L'evoluzione storica della bioinformatica, che inizialmente si occupava principalmente dello studio del DNA e RNA, ha portato ad un vasto uso dell'informatica in molti settori della biologia. Gli attuali ambiti di ricerca includono l'allineamento di sequenze, la predizione genica, l'allineamento di sequenze proteiche, la predizione di struttura proteica, l'espressione genica e l'interazione proteina-proteina: tutte applicazioni che possono richiedere grande potenza di calcolo in caso ci si proponga di analizzare istanze particolarmente complesse. Per ovviare a questo problema gli scienziati dei vari gruppi di ricerca si sono avvalsi dell'aiuto del calcolo distribuito volontario. In questo lavoro vengono elencati gli attuali progetti di ricerca nell'ambito della bioinformatica che si avvalgono dell'aiuto di tale strumento.

A.1. Rosetta@home

Figura A.1.: Logo di Rosetta@home

Rosetta@home[42] è un progetto di calcolo distribuito per la previsione della struttura delle proteine sulla piattaforma BOINC, svolto al Baker laboratory all'Università di Washington. Rosetta@home si propone di prevedere le interazioni proteina-proteina e di progettare nuove proteine con l'aiuto di 343 422 volontari, 1 067 924 computer, per una potenza di calcolo totale di 119,331 TeraFLOPS in media (alla data del 29 giugno 2012)[42].

Benché il grosso del progetto sia orientato verso la ricerca di base per migliorare la precisione e la robustezza dei metodi di proteomica, Rosetta@home fa anche ricerca applicata sulla malaria, il morbo di Alzheimer e altre patologie.

Oltre alla ricerca legata alle malattie, la rete di Rosetta@home funge da quadro di test per nuovi metodi di bioinformatica strutturale. Questi nuovi metodi sono poi utilizzati in altre applicazioni basate su Rosetta, come RosettaDock e il progetto Human Proteome Folding, dopo essere stati sufficientemente sviluppati e giudicati stabili sull'ampio e diversificato gruppo di utenti di Rosetta@home. Due prove particolarmente importanti per i nuovi metodi sviluppati con Rosetta@home sono il Critical Assessment of Techniques for Protein Structure Prediction (CASP) e il Critical Assessment of Prediction of Interactions (CAPRI), esperimenti biennali che valutano rispettivamente lo stato dell'arte nella previsione della struttura delle proteine e dell'interazione proteina-proteina. Rosetta@home si classifica tra i principali programmi di simulazione delle interazioni tra proteine ed è uno dei migliori metodi di previsione della struttura terziaria disponibili[43][44].

La piattaforma di Rosetta prende il suo nome dalla Stele di Rosetta, poiché tenta di decifrare il "significato" strutturale delle sequenze proteiche di aminoacidi.

Al nocciolo di Rosetta stanno le funzioni dei potenziali per calcolare le energie delle interazioni all'interno e tra le macromolecole, ed i metodi per trovare le strutture ad

energia minima per le sequenze di aminoacidi (predizione delle strutture proteiche), oppure per trovare la sequenza di aminoacidi a energia minima per una proteina (ingegnerizzazione di proteine)[45]. Viene utilizzato continuamente il feedback dalle predizioni e dai test di design per migliorare le funzioni dei potenziali e gli algoritmi di ricerca.

Lo sviluppo di un solo programma che tratta questi problemi abbastanza diversi porta a considerevoli vantaggi: le diverse applicazioni forniscono test complementari del modello fisico sottostante (la fisica/chimica di base, ovviamente, è sempre la stessa in ogni caso); molti problemi di interesse, come la progettazione di proteine con dorsale (backbone) flessibile ed il docking proteina-proteina con flessibilità delle dorsali, necessitano dell'utilizzo combinato di diverse tecniche di ottimizzazione.

I file che contengono dati sulle singole proteine, sono distribuiti dai server situati nel laboratorio Baker all'Università di Washington ai computer dei volontari, i quali calcolano una previsione della struttura per la proteina assegnata. Per evitare previsioni di struttura duplicate su una data proteina, ogni Workunit viene inizializzata con dei numeri casuali. Questo dà ad ogni previsione una traiettoria unica di discesa lungo il landscape energetico della proteina[46]. Le previsioni di struttura su Rosetta@home sono approssimazioni di un minimo globale nel landscape energetico di una data proteina. Questo minimo globale rappresenta la conformazione più favorevole dal punto di vista energetico della proteina, cioè il suo stato nativo.

Figura A.2.: *Il target T0281 del CASP6, la prima previsione ab initio di una struttura proteica che si è avvicinata ad una risoluzione a livello atomico. Rosetta ha prodotto un modello per T0281 (sovrapposto in magenta) con un RMSD di 1,5 Å dalla struttura cristallina (blu).*

Il laboratorio di Baker ha pubblicamente rilasciato Foldit[47], un gioco online per la previsione della struttura delle proteine basato sulla piattaforma di Rosetta che mira a raggiungere gli obiettivi del progetto con un approccio di "crowdsourcing"[48]. Infatti, strategie di gioco adottate dai giocatori di Foldit possono essere codificate in algoritmi per migliorare i meccanismi di previsione del folding. Per fare un esempio, tramite l'analisi dei dati dei giocatori di Foldit sono state raccolte circa 5 400 strategie diverse di gioco, dalle quali sono stati estratti due algoritmi particolarmente buoni. Esaminando questi algoritmi si è riusciti ad ottenere un algoritmo per il folding, molto simile ad un altro algoritmo, non ancora pubblicato, sviluppato in quel periodo da un altro gruppo di ricerca indipendente. Quindi, gli ambienti di giochi scientifici online possono potenzialmente risolvere problemi difficili e scoprire e formalizzare nuove strategie ed algoritmi per tali problemi.

A.2. Folding@home

Folding@home[49] è un progetto di calcolo distribuito che ha come scopo lo studio del folding delle proteine e della relativa incidenza sulle malattie. È stato lanciato il 1º ottobre 2000, e da allora è gestito dal Pande Group, nel dipartimento di chimica dell'Università di Stanford.

Come già accennato in precedenza, il folding delle proteine è una proprietà fondamentale delle biomolecole, che gioca un ruolo critico nella stabilità della proteina. Inoltre è la causa che sta dietro a molte malattie. L'obiettivo della ricerca svolta nel progetto Folding@home è quello di definire un modello quantitativo e predittivo del processo di folding[50].

Figura A.3.: Logo di Folding@home

Un tale modello aiuterebbe a colmare le attuali lacune nella conoscenza di tale processo in modo da poter migliorare le attuali tecniche di progettazione delle proteine, fornendo gli strumenti per creare proteine dalle proprietà desiderate.

Il progetto utilizza simulazioni di dinamica molecolare, poiché permettono di capire i meccanismi di folding delle proteine a livello atomico. Storicamente non è mai stato possibile eseguire simulazioni di dinamica molecolare con dettaglio a livello atomico: una proteina si può ripiegare in alcuni decimi di microsecondo, le simulazioni, d'altro canto, sono tipicamente limitate ad una scala temporale con granularità al nanosecondo, introducendo una differenza dai 3 ai 4 ordini di grandezza. Quindi, simulare il folding di una proteina a livello atomico è computazionalmente infattibile per un singolo processore.

Il folding delle proteine non avviene in un passo unico. Una buona porzione del tempo impiegato per ripiegarsi viene passato "in attesa" in varie conformazioni intermedie, corrispondenti ad un minimo locale nel landscape energetico della proteina. Folding@home fa ripartire le simulazioni da questi stati, e calcola le probabilità di transizione tra insiemi di conformazioni in parallelo. Questo approccio esplora l'insieme delle fasi della proteina, senza fermarsi troppo negli stati di minimo locale, e raggiunge una parallelizzazione quasi lineare, portando una riduzione significativa del tempo di esecuzione[51]. Le brevi simulazioni degli stati di conformazioni vengono poi trasformati in modelli markoviani di stato, che essenzialmente servono come mappe per l'energia libera della proteina, quella cinetica, e per l'equilibrio termodinamico. Una volta costruito, ognuno di questi modelli markoviani illustra gli eventi di folding ed i cammini intrapresi, e può rappresentare gli stati delle conformazioni a risoluzioni arbitrarie. Può anche rivelare quali transizioni stanno limitando l'accuratezza del modello, permettendo così la costruzione di simulazioni più accurate. Usando la tecnica del campionamento adattativo, il tempo impiegato per costruire un modello markoviano dello stato è inversamente proporzionale al numero di simulazioni parallele, cioè il numero di processori disponibili.

Folding@home usa questi modelli per simulare il folding a scale temporali biologicamente rilevanti, per scoprire come le proteine si ripiegano male, e per confrontare qualitativamente le simulazioni con gli esperimenti.

Con una potenza elaborativa di circa 6 PetaFLOPS al Giugno 2012, Folding@home sviluppa una potenza di calcolo confrontabile con quella dei più veloci supercomputer al mondo[52].

B. Appendice - Hash Table, DHT e SHA-1

B.1. Hash Table

Una hash table[53] è una struttura dati usata per mettere in corrispondenza una data chiave con un dato valore. L'hash table viene utilizzata nei metodi di ricerca denominati hashing. Una ricerca basata su hashing è completamente diversa da una basata su confronti: invece di muoversi nella struttura dati in funzione dell'esito di confronti tra chiavi, si cerca di accedere agli elementi nella tabella in modo diretto tramite operazioni aritmetiche che trasformano le chiavi in indirizzi della tabella. Una proprietà importante è che in una tabella di hashing ben dimensionata il costo medio di ricerca di ogni elemento è indipendente dal numero di elementi.

B.1.1. Hash Function

l primo passo per realizzare algoritmi di ricerca tramite hashing è quello di determinare la funzione di hash. La funzione hash è una funzione non iniettiva che mappa una stringa di lunghezza arbitraria in una stringa di lunghezza predefinita. Molto spesso si richiede che la funzione hash abbia le seguenti proprietà:

- *resistenza alla preimmagine*: sia computazionalmente intrattabile la ricerca di una stringa in input che dia un hash uguale a un dato hash;
- *resistenza alla seconda preimmagine*: sia computazionalmente intrattabile la ricerca di una stringa in input che dia un hash uguale a quello di una data stringa;
- *resistenza alle collisioni*: sia computazionalmente intrattabile la ricerca di una coppia di stringhe in input che diano lo stesso hash.

Il dato da indicizzare viene trasformato da un'apposita funzione di hash in un intero compreso tra 0 ed $m - 1$ che viene utilizzato come indice in un array di lunghezza m. Supponendo che U sia l'universo delle chiavi e $T[0...m - 1]$ una tabella hash, una funzione hash h, stabilisce una corrispondenza tra U e le posizioni nella tabella hash, quindi: $h : U \to \{0, 1, ..., m - 1\}$

B.1.2. Collisioni

Idealmente, chiavi diverse dovrebbero essere trasformate in indirizzi differenti, ma poiché non esiste la funzione di hash perfetta, ovvero totalmente iniettiva, è possibile che due o più chiavi diverse siano convertite nello stesso indirizzo. Il caso in cui, la funzione hash genera un medesimo indirizzo anche se applicata a due chiavi diverse, viene chiamato collisione e può essere gestito in vari modi.

La scelta di una buona funzione di hash è indispensabile per ridurre al minimo le collisioni e garantire prestazioni sempre ottimali. Il risultato migliore si ha con funzioni pseudo-casuali che distribuiscono i dati in input in modo uniforme. Molto spesso però, una buona funzione di hash può non bastare: infatti le prestazioni di una hash table sono fortemente legate anche al cosiddetto *fattore di carico* (load factor) calcolato come

Celle libere/Elementi presenti e che dice quanta probabilità ha un nuovo elemento di collidere con uno già presente nella tabella.

Questa probabilità, in realtà, è più alta di quanto si possa pensare, come dimostra il paradosso del compleanno[1]. È bene dunque mantenere il load factor il più basso possibile (di solito un valore di 0.75 è quello ottimale) per ridurre al minimo il numero di collisioni. Ciò può essere fatto, ad esempio, ridimensionando l'array ogni volta che si supera il load factor desiderato.

B.2. DHT - Distributed Hash Table

Una tabella di hash distribuita (DHT) [54, 16, 55] è una classe di sistemi distribuiti decentralizzati che fornisce un servizio di ricerca (lookup service) simile a quello fornito da una tabella di hash; nella DHT vengono salvate coppie (chiave, valore) ed ogni nodo partecipante può estrarre efficientemente il valore corrispondente ad una chiave data.

L'onere del mantenimento della mappatura dalle chiavi ai valori è distribuito tra tutti i nodi, in modo che un eventuale cambiamento nell'insieme dei partecipanti abbia un impatto minimo sui dati. Questo permette ad una DHT di scalare ad un numero estremamente grande di nodi, e di gestirne i continui arrivi, allontanamenti ed eventuali fallimenti.

B.2.1. Proprietà

Le caratteristiche fondamentali di una DHT sono:

- Decentralizzazione: i nodi formano un sistema senza nessuna coordinazione centralizzata;
- Tolleranza ai guasti: il sistema deve essere affidabile (secondo una qualche misura di qualità) anche se molti nodi entrano, escono dalla rete o sono soggetti a malfunzionamenti con elevata frequenza.
- Scalabilità: il sistema deve funzionare efficientemente anche con un numero molto elevato di nodi (anche milioni).

Una tecnica chiave usata per ottenere queste caratteristiche è fare in modo che ogni nodo richieda di coordinarsi solamente con pochi altri nodi – molto comunemente con $O(\log N)$ degli N partecipanti alla rete – così da limitare il lavoro necessario a gestire eventuali cambiamenti nella struttura della rete.

Le DHT sono sistemi distribuiti, e in quanto tali, devono affrontare le tradizionali problematiche di questi ultimi come il bilanciamento del carico, l'integrità dei dati e le prestazioni, in particolar modo devono garantire che operazioni come il routing, l'inserimento e l'estrazione dei dati vengano eseguite velocemente.

B.2.2. Struttura

Una DHT ha diverse componenti fondamentali. La più importante è lo *spazio delle chiavi* (ad esempio l'insieme delle stringhe da 128 bit). Una *partizione dello spazio delle chiavi* divide quest'ultimo in parti, l'appartenenza delle quali viene suddivisa tra i nodi partecipanti alla rete. Una *rete di overlay* connette i nodi, permettendo loro di trovare il proprietario di qualsiasi chiave nello spazio delle chiavi.

[1] Il paradosso afferma che la probabilità che almeno due persone in un gruppo compiano gli anni lo stesso giorno è largamente superiore a quanto potrebbe dire l'intuito: infatti già in un gruppo di 23 persone la probabilità è circa 0,51; con 30 persone essa supera 0,70, con 50 persone tocca addirittura 0,97, anche se per arrivare all'evento certo occorre considerare un gruppo di almeno 367 persone (per il principio dei cassetti e la possibilità di anni bisestili).

B.2.2.1. Un semplice esempio di funzionamento

Il tipico funzionamento di una DHT per immagazzinare e poi recuperare un dato avviene nel seguente modo. Si supponga che lo spazio delle chiavi sia un set di stringhe di 128-bit. Per immagazzinare nella DHT un file caratterizzato dai parametri `filename` e `data`, viene inizialmente calcolato lo SHA1 hash del filename, producendo così una chiave `key` da 128 bit. In seguito, un messaggio `put(key, data)` può essere inviato ad un qualsiasi nodo della rete DHT. Il messaggio è inoltrato di nodo in nodo attraverso l'overlay network fino a che esso non raggiunge il singolo nodo che è responsabile per la chiave `key` in accordo alle regole per il partizionamento dello spazio delle chiavi, laddove la coppia `(key, data)` è immagazzinata. Qualsiasi altro client può quindi successivamente recuperare i contenuti del file calcolando a sua volta l'hash del filename per ottenere `key` e chiedere ad un qualsiasi nodo della rete DHT di trovare il dato associato a `key` tramite un messaggio `get(key)`. Il messaggio verrà quindi inoltrato attraverso l'overlay verso il nodo responsabile per `key`, che risponderà con una copia del dato immagazzinato.

B.2.2.2. Partizionamento dello spazio delle chiavi

Molte DHT usano alcune varianti del consistent hashing per mappare le chiavi ai nodi. Questa tecnica impiega una funzione $\delta(k_1, k_2)$ che definisce la nozione astratta di distanza tra la chiave k_1 e la chiave k_2. A ciascun nodo è assegnata una chiave che è detta identificativo (ID). Ad un nodo con ID i appartengono tutte le chiavi per cui i è l'ID più vicino, misurando in base aδ.

Esempio L'algoritmo Chord considera le chiavi come punti su una circonferenza, e $\delta(k_1, k_2)$ è la distanza percorsa (in senso orario) sul cerchio tra k_1 e k_2. Perciò, lo spazio delle chiavi circolare è diviso in segmenti contigui i cui punti terminali sono gli identificativi di nodo. Se i_1 e i_2 sono due ID adiacenti, allora il nodo con ID i_2 è proprietario di tutte le chiavi che cadono tra i_1 e i_2.

Il consistent hashing ha la caratteristica fondamentale che la rimozione o l'addizione di un nodo modifica solo il set di chiavi posseduti dai nodi con ID adiacenti, senza coinvolgere tutti gli altri nodi. Tutto ciò al contrario di una hash table tradizionale, nella quale l'addizione o la rimozione di un bucket causa un rimappaggio di quasi tutto l'intero spazio delle chiavi. Dal momento che ogni modifica nel set di chiavi di cui un nodo è responsabile corrisponde tipicamente ad un intenso (per quanto riguarda la larghezza di banda) movimento da un nodo all'altro di oggetti immagazzinati nella DHT, una minimizzazione di questi movimenti di riorganizzazione è necessaria al fine di far fronte in maniera efficiente a peer che hanno un comportamento molto dinamico (elevato numero di ingressi, uscite o malfunzionamenti).

B.2.2.3. Rete Overlay

Ciascun nodo mantiene un set di link agli altri nodi (i suoi vicini). Tutti insieme questi nodi formano l'overlay network, e vengono organizzati in modo strutturato, dando così forma alla topologia della rete.

Tutte le topologie di DHT condividono qualche variante della proprietà più essenziale: per ciascuna chiave k, o il nodo possiede k oppure ha un collegamento ad un nodo che è più vicino a k in termini di distanza nello spazio delle chiavi, come definito precedentemente. A questo punto risulta quindi facile inoltrare un messaggio al proprietario di una qualsiasi chiave k utilizzando il seguente algoritmo greedy: ad ogni passo successivo, inoltra il messaggio al vicino il cui ID è più vicino a k. Quando non esiste un vicino con queste caratteristiche, allora si è giunti al nodo effettivamente più vicino, il quale deve

essere il proprietario di k in accordo con quanto definito sopra. Questo tipo di routing viene talvolta definito come key-based routing.

Al di là della correttezza di fondo di questo tipo di routing, vi sono due vincoli chiave nella topologia al fine che il numero massimo di passi successivi in un qualsiasi percorso (dilazione) sia basso, in modo tale che la richiesta sia soddisfatta velocemente, e che il numero massimo di vicini di ciascun nodo (il grado del nodo) sia basso, al fine di mantenere basso l'overhead di mantenimento. Vi è un compromesso tra questi due criteri che è fondamentale nella teoria dei grafi. Alcune scelte comuni sono fra quelle illustrate qui di seguito, dove N è il numero dei nodi nella DHT:

- Grado $O(1)$, dilazione $O(\log N)$
- Grado $O(\log N)$, dilazione $O(\log N/\log\log N)$
- Grado $O(\log N)$, dilazione $O(\log N)$
- Grado $O(N^{1/2})$, dilazione $O(1)$

La terza scelta è la più comune, sebbene non sia l'ottima in termini di compromesso grado/dilazione, poiché tale topologia tipicamente consente una maggiore flessibilità in termini di scelta dei vicini. Ciò è utile, ad esempio, per scegliere dei vicini che siano conformi alla topologia e al tempo stesso simili in termini di latenza nella sottostante rete fisica.

B.2.3. Implementazioni

Le principali differenze che si incontrano nelle istanze pratiche di implementazioni di DHT possono includere le seguenti:

Lo spazio delle chiavi è un parametro della DHT. Diverse DHT esistenti usano chiavi da 128 bit o da 160 bit.

Alcune DHT usano funzioni di hash diverse dalla SHA-1.

Spesso la chiave k è in realtà il hash del contenuto di un file, e non del suo nome, in modo da poter ritrovare una risorsa anche se questa viene rinominata.

Alcune DHT pubblicano diversi tipi di oggetti. Per esempio, la chiave k potrebbe essere il node ID e il dato associato potrebbe descrivere come contattare questo nodo. ID può essere un numero casuale che viene usato direttamente come chiave k (quindi una ID di una DHT da 160 bit sarà semplicemente una sequenza casuale di 160 bit).

Spesso si usa la ridondanza per migliorare l'affidabilità. La coppia (k, *dato*) può essere salvata in più nodi corrispondenti alla stessa chiave. Di solito, invece di scegliere solamente un nodo, le DHT scelgono i nodi diversi, con i parametro specifico dell'implementazione. In alcune implementazioni, i nodi si accordano per decidere i range di chiavi di cui sono responsabili in modo dinamico.

B.3. SHA-1

Lo Secure Hash Algorithm (SHA-1)[56, 57] ha come scopo quello di calcolare una rappresentazione condensata di un messaggio o di un file dati. Questo algoritmo riceve in ingresso un qualsiasi messaggio di lunghezza minore di 2^{64} bit e restituisce un output da 160-bit chiamato digest del messaggio.

Lo SHA-1 viene detto sicuro perché è computazionalmente impossibile trovare un messaggio il quale corrisponda ad un dato digest, oppure trovare due messaggi i quali producono lo stesso digest. Un hash crittograficamente sicuro non dovrebbe permettere di risalire, in un tempo confrontabile con l'utilizzo dell'hash stesso, ad un messaggio o

file che possa generarlo. Ogni cambiamento nel file oppure nel messaggio implicherà, con altissima probabilità, un cambiamento nel digest risultante.

La specifica originale dell'algoritmo fu pubblicata nel 1993 come Secure Hash Standard, FIPS PUB 180, dal NIST. Ci si riferisce spesso a questa versione come SHA-0 per distinguerla dalle successive versioni. Fu ritirata dalla NSA breve tempo dopo la pubblicazione e fu soppiantata da una versione rivista, pubblicata nel 1995 (FIPS PUB 180-1) e solitamente nota come SHA-1. Lo SHA-1 differisce dallo SHA-0 unicamente per una sola rotazione di bit nel processo di preparazione del messaggio della sua funzione di compressione ad una via; ciò fu fatto, secondo la NSA, per correggere un difetto nell'algoritmo originale, il quale riduceva la sicurezza crittografica di SHA-0. Ad ogni modo, la NSA non fornì nessuna ulteriore spiegazione chiarificante. Sono state in seguito riportate debolezze sia nel codice dello SHA-0 sia in quello dello SHA-1. Lo SHA-1 appare offrire maggiore resistenza agli attacchi, a supporto dell'asserzione della NSA che il cambiamento aumentò la sicurezza.

Lo SHA-1 (così come lo SHA-0) produce un digest di 160 bit da un messaggio con una lunghezza massima di $2^{64} - 1$ bit ed è basato su principi simili a quelli usati da Ronald L. Rivest del MIT nel design degli algoritmi MD4 e MD5.

C. Appendice - OMNeT++

OMNeT++[29, 30, 31] è un simulatore open-source ad eventi discreti, sviluppato presso la Budapest University of Technology and Economics. Scritto interamente in C++, è stato progettato per essere scalabile, modulare e semplice da estendere.

L'unità fondamentale di un modello in OMNeT++ è il SimpleModule (modulo semplice), che rappresenta un generico elemento della rete. Più SimpleModule possono essere raccolti in un CompoundModule (modulo composto), una struttura gerarchica che può essere arbitrariamente estesa più volte. I diversi moduli comunicano fra loro mediante primitive di scambio di messaggi; ciascun modulo è dotato di una o più gate (porte), i cui collegamenti possono essere definiti in maniera statica o parametrica.

I moduli di OMNeT++ definiscono implicitamente dei "tipi", ossia delle interfacce che possono essere associate a implementazioni distinte, in una sorta di polimorfismo. Il simulatore non fa distinzione da questo punto di vista fra l'interfaccia di un SimpleModule e quella di un CompundModule; è dunque possibile accorpare o scomporre queste strutture in una fase successiva, ad esempio in caso di re-utilizzo di interfacce preesistenti.

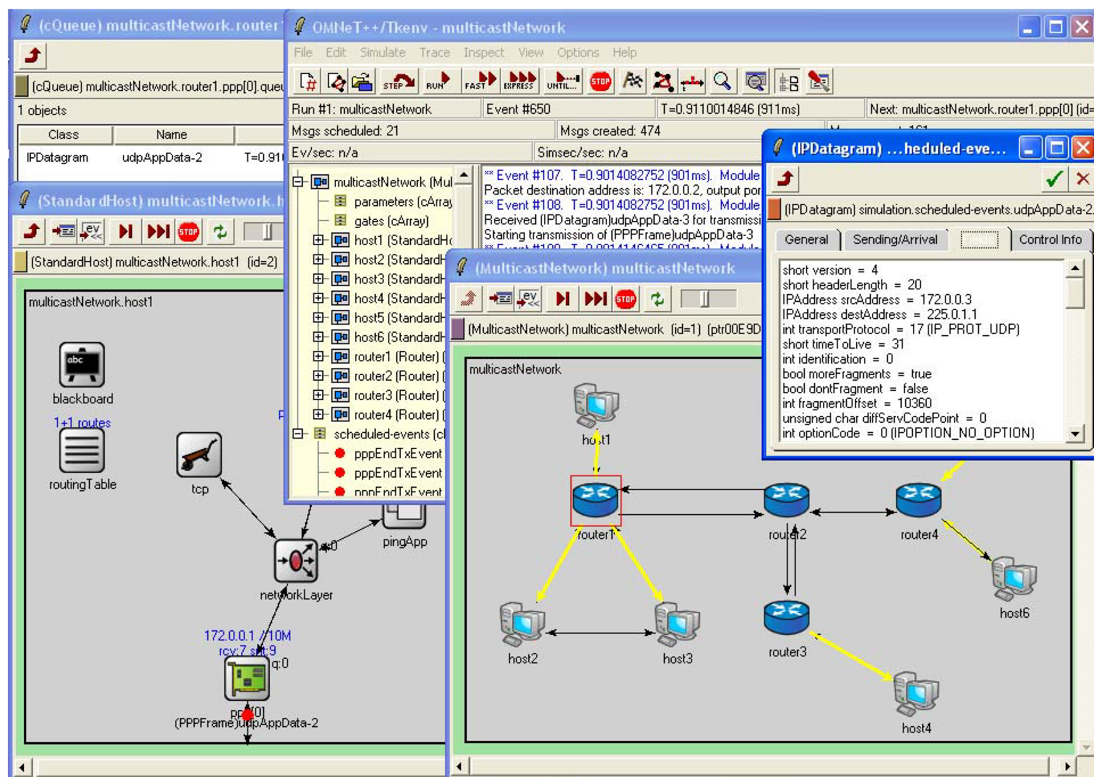

Figura C.1.: Screenshot di una simulazione in OMNeT++; si possono notare la struttura interna di un modulo host, la topologia di rete simulata e l'intestazione di un pacchetto IP in attesa nella coda di un router.

Inoltre, ciascun modulo può essere dotato di un numero arbitrario di parametri, i cui valori possono essere assegnati attraverso opportuni file di configurazione o al momento dell'esecuzione della simulazione. I parametri possono essere di tipo numerico, booleano

o stringa, e possono anche contenere alberi di dati in xml. Essi saranno poi accessibili al momento dell'esecuzione della simulazione dal codice C++ che implementa le funzionalità del modulo in questione; in questo modo è possibile modificare parametri n maniera dinamica, senza dover ricompilare i sorgenti ad ogni simulazione.

La struttura gerarchica dei moduli, i relativi parametri e le connessioni fra gli stessi vengono definiti in OMNeT++ attraverso uno specifico linguaggio, che va sotto il nome di NED (Network Definition Language). Si tratta di un linguaggio dichiarativo, di semplice utilizzo ma piuttosto flessibile. Esso consente, ad esempio, di definire in modo arbitrario le caratteristiche dei messaggi scambiati dai moduli della simulazione, permettendo la creazione di eventuali campi o strutture dati che andranno ad aggiungersi ad uno scheletro di base, attraverso meccanismi di ereditarietà. Il procedimento è particolarmente semplice, in quanto non è necessario riscrivere alcuna riga di codice C++; il compilatore NED di OMNeT++ provvederà a generare tutto il codice necessario a partire dalla dichiarazione dei campi del messaggio.

OMNeT++ è dotato di un'interfaccia grafica sviluppata in Tcl/Tk, attraverso la quale è possibile seguire in maniera dettagliata lo svolgimento della simulazione. Oltre allo scambio di messaggi fra i diversi componenti del modello di simulazione, la GUI consente di esaminare in ogni istante lo stato interno di ciascuno dei moduli, facilitandone il debugging. Naturalmente è possibile disabilitare l'uso dell'interfaccia grafica per l'esecuzione di simulazioni in remoto o di sistemi di grosse dimensioni.

Sono inoltre presenti tool di raccolta e di elaborazione di dati statistici ottenuti durante simulazioni (plove e scalar); è possibile ad esempio registrare valori scalari e vettoriali definendo una specifica finestra d'osservazione, processarli mediante filtri definiti dall'utente ed infine utilizzarli per la creazione di grafici di diverso tipo.

Utilizzando il framework di OMNeT++ sono state realizzate molte librerie che modellano il comportamento di diversi elementi in una rete. Vale la pena citare la libreria INET[33], che implementa numerosi protocolli di rete quali UDP, TCP, SCTP, IP, IPv6, Ethernet, PPP, 802.11, MPLS, OSPF, e molti altri. Sulla base di INET è stata realizzata infine OverSim, una libreria per la simulazione di reti overlay.

Bibliografia

[1] Great Internet Mersenne Prime Search Homepage. http://www.mersenne.org/.

[2] Distributed.net Homepage. http://www.distributed.net/.

[3] SETI@home Homepage. http://setiathome.berkeley.edu/.

[4] David P. Anderson. Public Computing: Reconnecting People to Science. In *Conference on Shared Knowledge and the Web*, Madrid, Spain, November 2003.

[5] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[6] BOINC homepage: Software open-source per il calcolo distribuito volontario e per il grid computing. http://boinc.berkeley.edu/.

[7] BOINC Wiki. http://boinc.berkeley.edu/wiki/Main_Page/.

[8] Unofficial BOINC Wiki. http://www.boinc-wiki.info/.

[9] Cécile Germain, Vincent Néri, Gilles Fedak, and Franck Cappello. XtremWeb: Building an Experimental Platform for Global Computing. In *Proceedings of the First IEEE/ACM International Workshop on Grid Computing*, GRID '00, pages 91–101, London, UK, UK, 2000. Springer-Verlag.

[10] Xgrid Homepage. http://support.apple.com/kb/PH11016?viewlocale=en_US/.

[11] Grid MP Homepage. http://www.univa.com/products/grid-mp/.

[12] Miguel Castro, Peter Druschel, Y.Charlie Hu, and Antony Rowstron. Topology-Aware Routing in Structured Peer-to-Peer Overlay Networks. In André Schiper, AlexA. Shvartsman, Hakim Weatherspoon, and BenY. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 103–107. Springer Berlin Heidelberg, 2003.

[13] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[14] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowstron. Exploiting network proximity in distributed hash tables. In Ozalp Babaoglu, Ken Birman, and Keith Marzullo, editors, *International Workshop on Future Directions in Distributed Computing (FuDiCo)*, pages 52–55, June 2002.

[15] Ratul Mahajan, Miguel Castro, and Antony Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *IPTPS'03*, February 2003.

[16] Ayalvadi Ganesh Antony Rowstron Miguel Castro, Peter Druschel and Dan S. Wallach. Security for structured peer-to-peer overlay networks. In *5th Symposium on Operating Systems Design and Implementaion (OSDI'02)*, December 2002.

[17] Peter Druschel and Antony Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, May 2001.

[18] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles (SOSP'01)*, pages 188–201, October 2001.

[19] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In Jon Crowcroft and Markus Hofmann, editors, *Networked Group Communication, Third International COST264 Workshop (NGC'2001)*, volume 2233 of *Lecture Notes in Computer Science*, pages 30–43, November 2001.

[20] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), October 2002.

[21] Miguel Castro, Michael B. Jones, Anne-Marie Kermarrec, Antony Rowstron, Marvin Theimer, Helen Wang, and Alec Wolman. An Evaluation of Scalable Application-level Multicast Built Using Peer-to-peer overlays. In *Infocom'03*, April 2003.

[22] Andreas Pfitzmann and Marit Köhntopp. Anonymity, Unobservability, and Pseudonymity - A Proposal for Terminology. In Hannes Federrath, editor, *Designing Privacy Enhancing Technologies*, volume 2009 of *Lecture Notes in Computer Science*, pages 1–9. Springer Berlin / Heidelberg, 2001.

[23] Hakim Weatherspoon and John Kubiatowicz. Erasure Coding Vs. Replication: A Quantitative Comparison. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 328–338, London, UK, UK, 2002. Springer-Verlag.

[24] Rodrigo Rodrigues and Barbara Liskov. High Availability in DHTs: Erasure Coding vs. Replication. In Miguel Castro and Robbert Renesse, editors, *Peer-to-Peer Systems IV*, volume 3640 of *Lecture Notes in Computer Science*, pages 226–239. Springer Berlin Heidelberg, 2005.

[25] Ros G. Dimakis, P. Brighten Godfrey, Yunnan Wu, Martin O. Wainwright, and Kannan Ramch. Network Coding for Distributed Storage Systems. In *In Proc. of IEEE INFOCOM*, 2007.

[26] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. One ring to rule them all: Service discover and binding in structured peer-to-peer overlay networks. In *SIGOPS European Workshop*, September 2002.

[27] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. OverSim: A Flexible Overlay Network Simulation Framework. In *Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007, Anchorage, AK, USA*, pages 79–84, May 2007.

[28] OverSim: The Overlay Simulation Framework. `http://www.oversim.org/`.

[29] András Varga and Rudolf Hornig. An overview of the OMNeT++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, Simutools '08, pages 60:1–60:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[30] OMNeT++ Homepage. `http://www.omnetpp.org/`.

[31] OpenSim Ltd., Budapest, Hungary. *OMNeT++ User Manual*, 2011.

[32] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. 2003.

[33] INET Framework Homepage. `http://inet.omnetpp.org/`.

[34] Thomas Gamer and Michael Scharf. Realistic simulation environments for ip-based networks. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, Simutools '08, pages 83:1–83:7, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[35] ReaSE - Realistic Simulation Environments for OMNeT++ Homepage. `http://projekte.tm.uka.de/trac/ReaSE/`.

[36] How to build OverSim with ReaSE support. `http://www.oversim.org/wiki/OverSimReaSE/`.

[37] Zhongmei Yao, Derek Leonard, Xiaoming Wang, and Dmitri Loguinov. Modeling Heterogeneous User Churn and Local Resilience of Unstructured P2P Networks. In *In ICNP*, pages 32–41, 2006.

[38] David P. Anderson and Gilles Fedak. The computational and storage potential of volunteer computing. In *Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '06, pages 73–80, Washington, DC, USA, 2006. IEEE Computer Society.

[39] Bahman Javadi, Derrick Kondo, Jean-Marc Vincent, and David P. Anderson. Discovering statistical models of availability in large distributed systems: An empirical study of seti@home. *IEEE Transactions on Parallel and Distributed Systems*, 22:1896–1903, 2011.

[40] Trilce Estrada, Michela Taufer, and Kevin Reed. Modeling job lifespan delays in volunteer computing projects. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 331–338, Washington, DC, USA, 2009. IEEE Computer Society.

[41] Unofficial BOINC Wiki: Projects, Science Applications, and Platforms.

[42] Rosetta@home Homepage. `http://boinc.bakerlab.org/`.

[43] Marc F. Lensink, Raúl Méndez, and Shoshana J. Wodak. Docking and scoring protein complexes: CAPRI 3rd Edition. *Proteins: Structure, Function, and Bioinformatics*, 69(4):704–718, 2007.

[44] Rosetta@home: Quanto sono accurate le nostre previsioni?. `http://boinc.bakerlab.org/rosetta/rah_about.php#accuracy/`.

[45] Rosetta@home: Presentazione della ricerca. http://boinc.bakerlab.org/rah_research.php/.

[46] Rosetta@home forums: Rosetta@home: Random Seed (message 3155). http://boinc.bakerlab.org/forum_thread.php?id=391&nowrap=true#3155/.

[47] Seth Cooper, Firas Khatib, Adrien Treuille, Janos Barbero, Jeehyung Lee, Michael Beenen, Andrew Leaver-Fay, David Baker, Zoran Popovic, and Foldit Players. Predicting protein structures with a multiplayer online game. *Nature*, 466(7307):756–760, August 2010.

[48] Firas Khatib, Seth Cooper, Michael D. Tyka, Kefan Xu, Ilya Makedon, Zoran Popović, David Baker, and Foldit Players. Algorithm discovery by protein folding game players. *Proceedings of the National Academy of Sciences*, 108(47):18949–18953, November 2011.

[49] Folding@home Homepage. http://folding.stanford.edu/.

[50] Stefan M. Larson, Christopher D. Snow, Michael Shirts, and Vijay S. Pande. Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology. 2002.

[51] Vijay S. Pande, Ian Baker, Jarrod Chapman, Sidney P. Elmer, Siraj Khaliq, Stefan M. Larson, Young Min Rhee, Michael R. Shirts, Christopher D. Snow, Eric J. Sorin, and Bojan Zagrovic. Atomistic protein folding simulations on the submillisecond time scale using worldwide distributed computing. *Biopolymers*, 68(1):91–109, 2003.

[52] Folding@home: Client statistics by OS. http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats/.

[53] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[54] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, August 2001.

[55] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371, December 2004.

[56] National Institute of Standards and Technology. *FIPS PUB 180-1: Secure Hash Standard*. Gaithersburg, MD, USA, 1995.

[57] D. Eastlake. RFC 3174: US Secure Hash Algorithm 1 (SHA1), September 2001.

Ringraziamenti

Volge a termine un percorso durato cinque anni, per certi versi faticoso, ma perlopiù bello e vissuto con intensità. Per questa fantastica esperienza sono moltissime le persone che devo ringraziare, e sarà certamente difficile elencarle tutte, ma ci proverò lo stesso, scusandomi da subito con chi non è stato menzionato.

Questo lavoro di tesi non sarebbe stato possibile senza una persona in particolare: il mio relatore - il Prof. Carlo Ferrari. È stato Lei professore ad incoraggiarmi a portare avanti questo lavoro, che mi ha dato molte soddisfazioni (e non poche notti insonni, ma se fosse stato facile non ne sarebbe valsa la pena!). La ringrazio per aver creduto in me, per il suo aiuto e la fiducia che mi ha dato.

Un ringraziamento particolare va alla mia famiglia: a mamma Çiljeta, a papà Vinçens e alla mia sorellina Regjina. Ringrazio i miei genitori per avermi dato, con moltissimi sacrifici, la possibilità di studiare a Padova – spero di non avervi mai deluso! Ringrazio Regjina per l'affetto che solo una sorella può dare al proprio fratello, anche quando il mio pH era sotto il 2 (. . . tristissimo tentativo di fare una battuta da chimico, spero che venga apprezzata lo stesso).

Se oggi sono qui a Padova lo devo anche a Don Mario Manara, e quindi un grosso ringraziamento va anche a lui. Mi ha voluto dare fiducia, e mi ha fatto venire in Italia senza neanche conoscermi. Spero di essere stato all'altezza delle sue aspettative!

Ringrazio i miei primissimi compagni di stanza: Denis Pontini e Matteo Romano. La mitica A307 rimane tuttora imbattuta! Grazie a Denis per gli scleri e le risate, e grazie a Matteo per averci sopportato! A entrambi auguro sempre e solo il meglio dalla vita e dalla Ferrari!

Ringrazio Pierandrea Magagna, detto anche Lupetto, Fulvino, Pjerin Ujka, Scout feccia e Piééééééérrr. È stato bello averti in camera, davi quel tocco di vitalità alla camera nei periodi di sessione che altrimenti non avrebbe avuto! Insieme abbiamo infranto il limite della birra a colazione. Sei stato e sarai sempre una distinta presenza nel Serenissimo Piano. Spero tu decida finalmente di rinunciare agli scout e a quella vita dissoluta alla quale ti sei abbandonato in questi anni! Ah, grazie a te ODIO "Sweet home Alabama" dal profondo del mio essere – e di conseguenza sono passato a musica molto più cattiva!

Grazie anche ad Andrea Mocellin, che ha resistito per un anno intero in camera con me! Ti faccio i miei migliori auguri per tutto!

Un ringraziamento immenso va alla mitica A305! Ringrazio TUTTI i miei compagni di stanza dell'anno scorso: Enrico "Biro" Biroli, Giuseppe "Beppe" Gobbetti, Angelo "Angel" Canal, Mohamed "Momi" Chalal, Giovanni "il Brune" Brunelli, Andrea "Padre" Marolla e Vincenzo "il Vince" Morello. Stare in ottupla è stata un'esperienza mistica!

In particolare un ringraziamento davvero sentito va ad Enrico, per l'allegria che ha portato sia in camera che in piano. Enrico, hai il potenziale per raggiungere qualsiasi vetta, non ti arrendere mai!

Il buon Vincenzo Morello ha moltissimi meriti, tra i quali quello di avermi sopportato durante l'ultimo mese. So di essere stato un vero "cacacazzu" in questo periodo, e a breve ti arriverà una medaglia al valore militare! Inoltre grazie per avermi invogliato ad esigere sempre di più da me stesso, non più solo sul piano accademico, ma anche quello fisico: se sono riuscito a fare le serie con i 100kg in panca piana è merito tuo!

Grazie al Beppe per le jam session in camera ad orari improbabili, per le kebabbate di mezzanotte, per le bevute e le feste di piano!

Grazie ad Angel per la sua bontà d'animo! Lo dico a nome di tutti gli ingegneri del collegio: se fossi l'unico filosofo in collegio, nessuno crederebbe Filosofia una facoltà burla (e invece c'è anche Masarà...). Grazie per tutto, per la compagnia, per le maratone di zombi-movie e per le bevute (soprattutto per due domeniche fa).

Ringrazio le colonne portanti del Serenissimo Piano: Mohamed "Momi" Chalal, Ezio "Ezius" Minnicelli, Luca "Mollo" Enrico Ferrari e Nicolò "Giane" Giannesini. Siete l'anima e il corpo di questo piano. Grazie a voi ho potuto essere il Doge di un grande Piano! Luca hai delle grosse responsabilità quest'anno: ricordati di dare il giusto peso anche alla goliardia e agli eventi ludici! Giane, tieni d'occhio il Mollo e non farlo studiare troppo! Ezio il piano sentirà tanto la tua mancanza dato che parti per l'erasmus tra non molto.

Momi sei stato un vero amico per cinque lunghi anni, grazie di tutto! Ti faccio i migliori auguri per tutto, e spero tra non molto di partecipare alla tua laurea.

Ringrazio anche tutti gli altri ragazzi del Serenissimo Terzo Piano, che hanno contribuito a fare del mio quinto anno il migliore in assoluto! Grazie a Riccardo Laterza, Francesco Zanette, Daniele Gerosa, a Matteo "Suspe" "Papa" Sfragara, a Davide "Pinky" Rosi, a Luigi "Bagigi" Guarato, a Piersaaaaaavino Lichinchi e a Francesco Cutore!

Grazie ad Andrea Martorana per la sua allegria, per la costante voglia di far festa e soprattutto per avermi "trasportato" durante un evento che alla lettura del papiro probabilmente verrà reso noto.

Grazie a Matteo "Valli" Vallar per la sua simpatia! Valli sei uno dei migliori in assoluto, continua così. Però devi prendere esempio da Vincenzo e andare in palestra!

Grazie ai dottori Massimiliano "Max" Carrozzini e Andrea Guttilla. Ragazzi siete davvero forti! È stato un piacere condividere momenti di quotidianità e di festa con voi!

Ringrazio tutti gli altri vecchi-vecchi del collegio (ormai siamo rimasti in pochi butei): Michele Zilocchi, Remon Atfy, Daniele Grossule, Paolo Macaccaro, Andrea Pomaré Montin, Francesco Romano, Eduardo Bloise, Matteo Parolin, Alessandro Dal Maso e "il Conte" Pierpaolo Martini.

Grazie a Michele "Michael" (e non Maicooooool, per non far confusione!) per le partite a scopone, per la goliardia che abbiamo saputo attuare insieme, e per i bei momenti passati con un sigaro in una mano e il brandy nell'altra!

Grazie a Eduardo per aver studiato Fondamenti di Comunicazioni con me! Siamo stati grandi vecchio, passato al primo colpo con voti più che decenti!

Grazie a Francesco Romano per l'ilarità che hai sempre portato! Se eviti di suonare l'organo di domenica mattina ti sarei ancora più grato! (ho fatto anche una pessima rima - non era assolutamente voluta, lo giurò :P)

Grazie a Paolo "Pol" Macaccaro per le belle serate in compagnia. Si partiva da casa sani per tornare pieni come lupi, e tu furbacchione ti portavi la carrozzina... me ne devo prendere una anch'io... (p.s. Cosimo deve ringraziare anche te mi sa per quella memorabile notte!)

Grazie a Riccardo "Gaspa" "Satana" Gasparetto Stori e Andrea "Vuda" Battaggia, che fanno da dissidenti politici/religiosi e rompono la monotonia di questo posto. Da Gaspa mi aspetto tante botte per la laurea, dato che ti ho fatto il sedere a strisce alla tua. Butei, anche se non siete più al quinto, voi siete il quinto, tenete duro!

Grazie a Damiano Duci per la sua gentilezza e la bontà d'animo! Grazie per tutti i vassoi che mi tenevi e per aver sopportato le polemiche sul tuo orientamento gastronomico.

Ringrazio Mauro "Dodoge" Pozzi e Enrico "Soviet-man" Maso per tutti gli anni che mi hanno sopportato in piano! Mauro ho ancora una bellissima foto del mio capola-

voro fatto di carta e ketchup con il quale hai fatto outing! A proposito, come sta tua morosa? Enrico ormai starai producendo sigarette in Afghanistan o qualche altro paese scandinavo, mi raccomando, testa bassa quando esci a fare la spesa... Scherzi a parte, grazie di tutto ragazzi!

Grazie a Damiano Marcazzan che è stato un buon vicino di stanza e anche un buon Doge! Grazie a te non mancavano mai le risate e la festa in piano!

Ringrazio Francesco "Ciccio" Locascio per tutti i consigli dati in questi anni! Ho potuto prendere esempio dal migliore! Auguri per tutto Ciccio!

Devo ringraziare sicuramente anche il buon Michele Fadone, che mi ha permesso di stare tranquillo le due settimane precedenti alla laurea. Grazie vecchio, mi hai salvato da Mannai, te ne sono grato!

Un grazie sentito a Marco De Mitri per tutti i momenti trascorsi insieme, sia nei piani che in palestra. Ti faccio i migliori auguri per il tuo nuovo percorso universitario!

Grazie a Simone Platania e Antonino Lamia, che si sono dimostrati due personaggi inimitabili! Simone in bocca al lupo per il trasferimento! Antonino grazie per lo scorso Mazzurro!

Ringrazio Xhon "Joy" "Gionna-boy" Çoba, Daniele Buonuomo e tutti gli altri butei del quinto! Siete un po' fiacchi quest'anno, mi raccomando con le compilation su Morgan! Il collegio ha bisogno di nuovi brani!

Grazie a Stefano, la mia matricola "provvisoria" e a Christian il mio vicino di stanza "provvisorio". Grazie di avermi tenuto compagnia in questi giorni ragazzi! Auguri per il vostro percorso universitario!

Grazie a Stefania Presta per il tempo passato in portineria a ciacolare (hai ancora il foglio?)!

Ringrazio anche Gianluca D'Incà! Come facevamo senza di te in collegio prima?

Grazie alla nostra cuoca preferita Diana Shima per la quale normale significa abbondante e abbondante significa un altro piatto!

Uno speciale ringraziamento va sicuramente a Denise De Zanet che è stata tra le prime persone che ho conosciuto in collegio. Sei una forza! Porti allegria d'ovunque tu vada! Grazie di tutto, anche per gli sbrandi su commissione che hai perpetuato per conto mio dalle vostre parti in questi anni!

Grazie a Giulia Garziera e Linda Da Rugna, compagne di bevute affidabili ed esperte dispensatrici di coccole! Grazie per tutto ragazze, siete davvero uniche! Giulia grazie (forse?) per il disegno del papiro, so che lo hai fatto tu! Grazie (forse?) a Linda per il papiro, spero tu non sia stata troppo cattiva...

Grazie a Federica Bloise per aver smesso di intasarmi la homepage di facebook con link da 14-enni, scherzo... Ti ricordi all'inizio quanto ti avevo detto su per questa cosa? Comunque grazie per questi anni, saranno sicuramente indimenticabili! E grazie anche per il papiro, mi dicono che se non fosse per voi donne, a quest'ora starei leggendo le etichette delle birre sui muri del Le Royale.

Grazie a Laura "Lauri" Diena e ad Angelica "Picci" Mascitti! Ragazze siete una forza! Mi dispiace che non siate in collegio anche quest'anno, ma probabilmente a voi no. Vi verrò presto a trovare, Picci confido in te per poter magari mangiare qualcosa che non sia un intruglio vegano. Lauri grazie per le ore di studio insieme e per i pranzi col sushi.

Grazie a Valentina Macchia, a Valentina Quadrio, a Sofia Silvestri e a Silvia Emmanuele! Sofia e Silvia mi mancherete da morire! Grazie di tutto ragazze, dobbiamo assolutamente rifare una serata devasto come quelle degli anni passati!

Grazie a Federica Fraccaroli per avermi distratto dalla tesi in questi giorni, anche se in modo leggermente molesto... scherzo, grazie Fede, sei una forza!

Ringrazio Elisa Saler per l'indimenticabile serata passata a vedere "Notre Dame De Paris". E questo è il tempo delle cattedrali...

Grazie a Roberta Messina che accompagna col sorriso tutti i mazziani che entrano ed escono dal collegio!

Ringrazio Rossella Petrucci per tutte le ore di studio e sclero che abbiamo passato insieme in questi cinque anni! Hai reso sopportabile il DEI, grazie davvero :)!

Grazie a Lucia Petterle che tra alti e bassi mi è sempre stata vicina. Luci anche tu sei prossima alla fine ormai, tieni duro e finisci questa benedetta tesi! Auguri per tutto!

Ringrazio Alessandro Di Pieri per i bei momenti trascorsi insieme, in mensa, e soprattutto a casa sua a Mestre. Che festoni che abbiam fatto!

Un ringraziamento va a Federica Moro per le infinite ore di studio, soprattutto sul progetto delle formiche. Grazie per non avermi sparato quella volta, davvero...

Grazie a Daniele Barilaro! Se avrò un futuro come taglialegna sarà solo merito tuo!

Ringrazio Francesco "Sisko" Carbone per le ore passate a preparare RO2. Che palle quel corso... da solo non avrei mai avuto abbastanza voglia di studiare quella roba la. Grazie!

Grazie a Luca "Pea" Pellegrini per essere il mona che sei! Anche a te non manca molto, quindi attento che da me ne prenderai tante di botte!

Ringrazio Alvise "Figo" Rigo per la simpatia e la gentilezza che ha sempre saputo dimostrare in questi anni! Quand'è che mi fai fare un giro sulla tua Lamborghini?

Ringrazio Mattia Ornamenti per i pomeriggi trascorsi d'avanti a n birre e a discutere delle verità dell'universo.

Grazie anche a Mattia Samory per il conforto morale e alcolico che mi ha saputo dare in questi anni! Domani tocca a te caro, mi raccomando ti voglio bello ciucco alla tua laurea!

Concludo qua ringraziando tutte le persone non menzionate in queste poche pagine (che sono un'infinità) e che hanno contribuito a rendere questi cinque anni meravigliosi. Grazie di tutto!

Padova, lì 11 ottobre 2012