

UNIVERSITÀ DEGLI STUDI DI TORINO

Facoltà di Scienze Matematiche Fisiche e Naturali



CORSO DI LAUREA IN INFORMATICA
ANNO ACCADEMICO 2005/2006

RELAZIONE DI TIROCINIO

Progettazione e implementazione di un sistema
di accounting per una rete peer to peer.

Relatore: Prof. Giancarlo Ruffo

Candidato: Fabio Varesano

Indice

0. Introduzione.....	5
1. Inquadramento generale.....	6
1.1 Architetture applicative classiche.....	6
1.2 Architetture Peer to Peer.....	7
1.3 Architetture Client/Server e Peer to Peer a confronto.....	7
1.4 Il problema del Free Rider.....	8
Egoismo.....	9
Maliziosità.....	9
Inaffidabilità.....	10
1.5 Incentivi in sistemi peer to peer.....	10
Schemi basati su fiducia e reputazione.....	10
Schemi basati su scambi.....	10
1.6 Scopo della tesi.....	11
2. Obiettivi di un sistema di accounting.....	12
2.1 Decentralizzazione.....	12
2.2 Efficienza.....	12
2.3 Scalabilità.....	12
2.4 Affidabilità.....	12
2.5 Sicurezza.....	12
2.6 Persistenza.....	13
2.7 Consistenza.....	13
2.8 Flessibilità.....	13
3. Overlay Network e Pastry.....	14
3.1 Proprietà delle DHT.....	15
3.2 Pastry.....	15
Stato dei nodi.....	16
Procedura di Routing.....	17
Auto adattamento della Rete.....	18
Località.....	19
4. Schema di Protocollo.....	21
4.1 Assunzioni Iniziali.....	21
4.2 Operazioni supportate.....	21
4.3 Convenzioni grafici.....	22
Propagazione messaggio e Response.....	23
4.4 Funzioni e convenzioni di utilità.....	24
4.5 Balance Request.....	25
4.6 Funding.....	27
4.7 Cash Flow.....	28
4.8 Withdrawal.....	30
4.9 Miglioramenti al protocollo.....	31
Affidabilità / Maliziosità dell'account root.....	31
Frode durante Cash Flow.....	33
4.10 Node join e Node leave.....	34
4.11 Creazione di un account.....	35
5. Implementazione.....	36
5.1 FreePastry.....	36
5.2 Approccio allo sviluppo.....	36
Continuation.....	36
5.3 COIN.....	38

Package e Classi.....	38
API.....	40
5.4 Protocollo e Implementazione.....	42
Pastry 2.....	42
Protocollo e Messaggi.....	43
5.5 Test.....	46
6. Miglioramenti e Sviluppi Futuri.....	47
6.1 Sicurezza.....	47
6.2 Persistenza.....	47
6.3 Dimensione Leafset.....	48
6.4 Formato dei messaggi.....	48
6.5 Affiancamento ad altro applicativo.....	48
6.6 Testing.....	48
7. Conclusioni.....	50
7.1 Ringraziamenti.....	50
8. Bibliografia.....	51
9. Curriculum Vitae.....	52

0. Introduzione

Questo testo descrive la mia esperienza come tirocinante presso il Dipartimento di Informatica dell'Università di Torino. Durante il mio tirocinio, supervisionato dal Professor Giancarlo Ruffo, ho lavorato ad un sistema di accounting (contabilità) per una rete peer to peer.

Il mio lavoro, durato, tra documentazione, progettazione ed implementazione, più di quattro mesi, mi ha permesso di entrare in contatto con tecnologie entusiasmanti quali, ad esempio, le Distributed Hash Tables (DHT) o Pastry.

Nei seguenti capitoli vengono presentate le motivazioni alla base della necessità di un sistema di accounting. Viene quindi descritto un protocollo applicativo da me ideato.

Successivamente è presentata una implementazione prototipale del protocollo presentandone le differenti scelte implementative e architetturali. Vengono suggeriti inoltre alcuni possibili sviluppi e miglioramenti ulteriori all'applicazione realizzata.

Capitoli e contenuti:

- 1. Inquadramento generale**
vengono presentate e analizzate le architetture client/server e peer to peer, viene analizzato il problema del Free Rider, vengono introdotti i sistemi ad incentivi ed infine inquadrato lo scopo della tesi.
- 2. Obbiettivi di un sistema di accounting**
vengono analizzati gli obbiettivi fondamentali alla base della progettazione di un sistema di accounting.
- 3. Overlay Network e Pastry**
vengono introdotte le Overlay Network, viene presentato nei dettagli Pastry.
- 4. Schema di protocollo**
viene presentato nei dettagli il protocollo applicativo progettato, dettagliando le varie interazione nei differenti stati del sistema.
- 5. Implementazione**
viene introdotta la fase implementativa del protocollo, vengono motivate alcune scelte implementative e presentate alcune soluzioni.
- 6. Miglioramenti e Sviluppi Futuri**
vengono presentati alcuni possibili miglioramenti apportabili alla implementazione ed alcuni interessanti sviluppi futuri.
- 7. Conclusioni**
vengono riportate le mie conclusioni personali sull'esperienza del tirocinio.

Ultima modifica: giovedì 7 dicembre 2006 13.28

1. Inquadramento generale

Internet rappresenta sicuramente una delle tecnologie che più hanno avuto impatto sulla vita delle persone nei nostri giorni. Ogni giorno milioni di utenti si scambiano email, chiacchierano nelle chat, partecipano ad aste online, etc. Recentemente, le tecnologie basate su Internet hanno addirittura fatto irruzione nella telefonia al punto che è possibile compiere delle telefonate esclusivamente utilizzando Internet.

Gran parte di questo sviluppo si deve alla buona progettazione delle tecnologie su cui si basa Internet. Queste tecnologie, denominate protocolli, sono state fondamentali alla nascita di Internet, ma hanno dimostrato la loro buona progettazione solo nell'ultimo decennio, rendendo possibile lo scalare dell'interrete da poche centinaia di host fino alle dimensioni planetarie attuali. Pensare che i protocolli su cui si basa tutta la comunicazione Internet, quali ad esempio TCP/IP, furono ideati e pubblicati quasi trenta anni or sono rende possibile avere un'idea di quale sia stata la cura nello sviluppo di tali protocolli.

1.1 Architetture applicative classiche

I servizi utilizzati ogni giorno da milioni di persone tramite Internet, quali ad esempio lo scambio di email, la fruizione di pagine web, lo scambio di messaggi istantanei tramite chat basano il loro funzionamento su diversi protocolli applicativi.

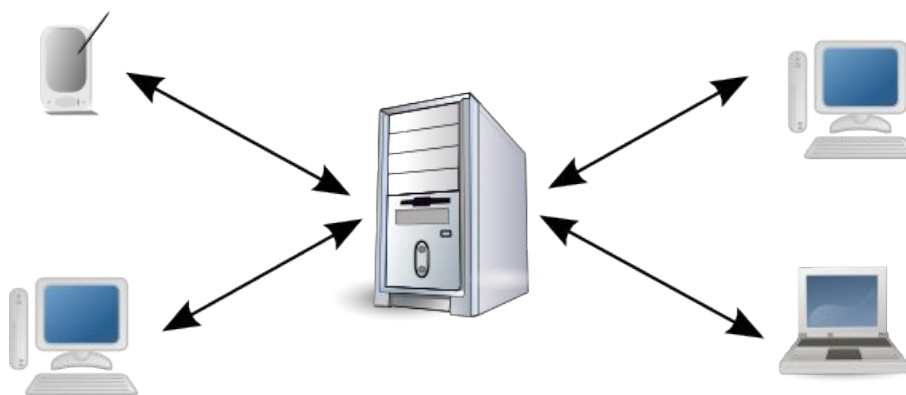


Immagine 1: architettura client/server

Questi protocolli applicativi sanciscono le modalità di interazione tra le varie entità applicative coinvolte nel delivering del servizio. Praticamente tutti questi protocolli presentano una architettura Client/Server.

Questo genere di architettura, semplificando, presenta un host detto server che eroga il servizio. Gli altri host della rete che accedono al servizio erogato dal server vengono detti client. Queste due tipologie di entità interagiscono secondo le modalità descritte dal protocollo applicativo permettendo la fruizione del servizio finale.

1.2 Architetture Peer to Peer

Da non moltissimo tempo su Internet sono comparsi servizi particolari che basano il loro funzionamento non su una architettura client/server classica bensì su una peer to peer (P2P). Questo genere di architetture sono oggi principalmente conosciute per servizi di file sharing che permettono lo scambio, molte volte illegale, di file tramite internet.

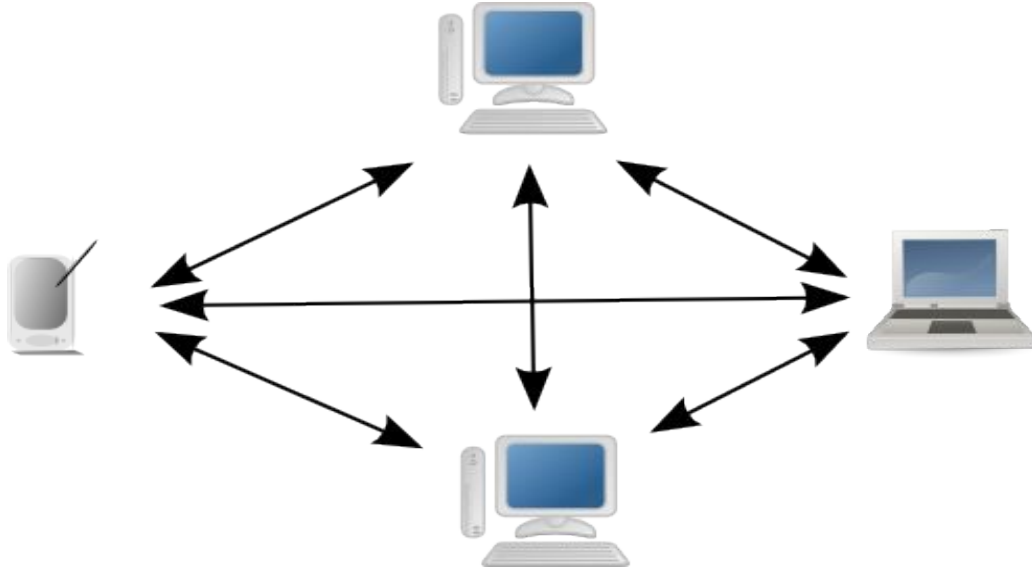


Immagine 2: architettura peer to peer

Questo genere di architetture sono parzialmente o completamente decentralizzate nel senso che limitano o eliminano del tutto la presenza di entità centrali (server) relegando tutta la gestione del protocollo agli host.

1.3 Architetture Client/Server e Peer to Peer a confronto

Le architetture Client/Server e Peer to Peer presentano entrambe vantaggi e svantaggi estremamente legati fra loro.

La tabella seguente presenta le proprietà delle due differenti architetture.

Proprietà	Architettura Client/Server	Architettura Peer to Peer
Amministrabilità	+	-
Consistenza dei Dati	+	-
Estensibilità	-	+
Tolleranza ai Guasti	-	+
Sicurezza	+/-	-
Resistenza alle denunce	-	+
Scalabilità	+/-	+

Chiaramente, un vantaggio chiave dei sistemi Client/Server è l'essere

generalmente facili da amministrare e mantenere consistenti e sicuri. Purtroppo, dato che le risorse di un Server sono solitamente difficili da estendere, le performance di un sistema Client/Server tipicamente diminuiscono all'aumentare dei Client che utilizzano il servizio. Quindi, per sistemi con un grandissimo numero di utenti, una architettura Client/Server potrebbe non essere scalabile o almeno essere necessariamente accompagnata da elevati costi di gestione e infrastrutturali per poter mantenere una buona Quality of Service (QoS).

Inoltre, dato che un server centrale rappresenta di fatto un single point of failure, le applicazioni basate su di una architettura Client/Server hanno una bassa tolleranza ai guasti in quanto, se un malfunzionamento impedisce il corretto funzionamento del server, l'intero sistema non sarà in grado di funzionare rendendo inutilizzabile il servizio. In aggiunta i sistemi Client/Server sono tipicamente anche vulnerabili ad alcune forme di attacchi, ad esempio i Denial-of-Service (DoS) o problemi di carattere giudiziario.

L'approccio Peer to Peer rimuove il potenziale collo di bottiglia rappresentato dal server centrale. Più risorse vengono distribuite e replicate fra i diversi peer, maggiormente il sistema risulta tollerante ai guasti. Inoltre, dato che ogni peer che partecipa al sistema compensa il carico di lavoro aggiuntivo rendendo disponibili le proprie risorse, un sistema Peer to Peer scala tipicamente bene all'aumentare del numero di utenti.

Purtroppo, l'amministrabilità, la consistenza dei dati e la sicurezza, che sono relativamente facili da ottenere in un sistema Client/Server, sono decisamente più difficili da raggiungere in un sistema Peer to Peer e richiedono complessi meccanismi di gestione.

1.4 Il problema del Free Rider

Le architetture Peer to Peer basano il loro funzionamento sull'assunzione che i peer collaborino al fine di ottenere uno scopo comune. Per collaborazione nei sistemi Peer to Peer si intende che ogni peer debba contribuire tante risorse proprie quante ne utilizza dal sistema.

Ad esempio, se un peer vuole scaricare da un sistema di file sharing peer to peer un file di 10MB, dovrebbe poi condividere questo determinato file permettendo agli altri peer di scaricare da lui almeno 10MB di dati.

In questo modo viene garantita la sostenibilità del sistema, uniformando e rendendo equo il rapporto risorse ottenute/risorse condivise tra i vari peer che partecipano al sistema.

Purtroppo però, dato che i peer sono entità autonome, non è possibile assumere che si comportino in un determinato modo. È quindi inevitabile che, senza un appropriato meccanismo sociale o economico, la collaborazione tra i peer non avvenga.

Ad esempio, è stato osservato che, nell'applicazione di filesharing Gnutella, senza incentivi per la cooperazione, solo pochi dei peer offrono risorse. Quasi il 70% dei peer non condividono nessun file mentre il 50% dei risultati delle query di ricerca vengono restituiti da solo l'1% dei peer. Questa mancanza di

collaborazione porta inevitabilmente al degradarsi delle prestazioni del sistema.

Per *free rider* si intende quindi un peer il quale trae beneficio dagli sforzi degli altri peer senza contribuire nessuna risorsa o senza avere un comportamento corretto per il raggiungimento del bene comune.

Nei sistemi Peer to Peer, due differenti tipi di free riding possono essere distinti:

- quello per la non condivisione di risorse, quali ad esempio file o risorse hardware
- quello per la non collaborazione nelle funzionalità base del sistema, quali ad esempio non eseguire il forwarding delle query di ricerca.

Mentre la prima tipologia di free riding è più semplice da individuare (basta chiedere al peer le risorse da lui condivise, la lista dei file condivisi per esempio), la seconda risulta assai più complessa da individuare.

Il fatto che i peer non collaborino o non si comportino correttamente è un problema decisamente grave nei sistemi peer to peer che è necessario affrontare per produrre una applicazione veramente efficace e affidabile.

Le ragioni di una non collaborazione da parte dei peer sono differenti e vengono analizzate di seguito.

Egoismo

Uno dei principali problemi e ragioni per il free riding è la razionalità e l'attenzione ai propri interessi dei peer. Senza un appropriato meccanismo economico o sociale un peer razionale non ha nessun incentivo per la collaborazione in quanto collaborare è costoso in termini di risorse e può facilmente ridurre l'esperienza dell'utente locale.

Ad esempio rendere disponibile un file ad altri peer in una applicazione di file sharing è costoso in quanto consuma la banda del peer locale e può rendere il sistema più lento in altri compiti.

Bisogna notare che, al fine di poter attuare questi meccanismi sociali ed economici, è necessario che un particolare comportamento sia registrabile, cosa non sempre possibile.

Maliziosità

Un altro importante problema è il comportamento malizioso dei peer. Ad esempio un peer potrebbe comportarsi deliberatamente in modo scorretto rispetto ai suoi potenziali competitori rispetto una risorsa al fine di aumentare il proprio profitto.

Bisogna notare che un peer potrebbe addirittura usare scorrettamente il sistema proprio al fine di renderlo inutilizzabile o comunque non affidabile.

Un meccanismo di incentivi non sarebbe utile per prevenire la maliziosità dei peer. L'unica alternativa possibile è quella di identificare univocamente i peer e i messaggi da loro generati utilizzando ad esempio uno schema di firma e/o

codifica di messaggi basato su algoritmi di encryption a chiave asimmetrica.

Inaffidabilità

Il fatto che un peer non segua un comportamento corretto rispetto ad certo protocollo, non significa necessariamente che si stia comportando maliziosamente od egoisticamente. Un problema che deve venire affrontato nelle reti Peer to Peer è, infatti, la non affidabilità dei peer.

I peer sono inaffidabili per costituzione e possono fallire sia per problemi loro (errori di configurazione o implementazione) che per problemi nella rete internet.

Il meccanismo ad incentivi, anche in questo problema, non può essere d'aiuto. L'unica soluzione è implementare le applicazioni in modo che possano tollerare i fallimenti dei singoli peer e che quindi siano il più possibile ridondanti.

1.5 Incentivi in sistemi peer to peer

Al fine di eliminare il problema del free riding è necessario dare ai peer degli incentivi al fine di convincerli a contribuire positivamente al sistema.

È possibile dividere i differenti approcci agli incentivi in una tassonomia che viene riportata nella figura sottostante.

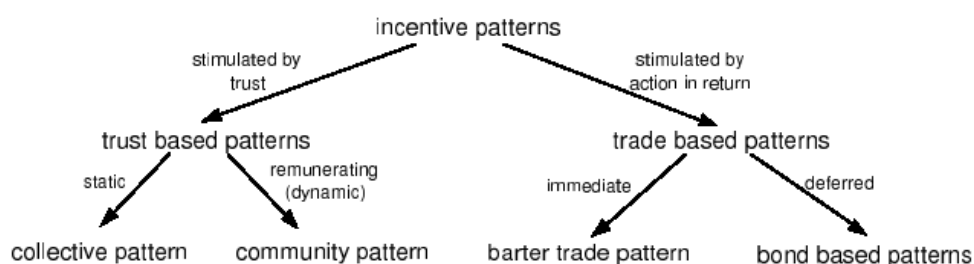


Immagine 3: tassonomia per incentivi in sistemi P2P

Schemi basati su fiducia e reputazione

Questo genere di schemi ad incentivi si basa sulla fiducia che viene stipulata tra i diversi peer del sistema. Viene infatti creata una sorta di comunità di peer, che possono accedere ai diversi servizi in base alla reputazione che hanno acquisito all'interno della community.

Schemi basati su scambi

I sistemi ad incentivi basati su scambi possono essere di due tipologie:

- barter-trade: vengono scambiati servizi tra i peer
- bond based: viene pagata una moneta virtuale durante l'acquisizione di un servizio

Gli schemi barter-trade vengono attualmente utilizzati in diverse applicazioni di file sharing peer to peer attuali. Alcuni esempi sono il meccanismo tit-for-tat di Bittorrent o il meccanismo a crediti di Emule. Purtroppo questi approcci sono

profondamente orientati al file sharing e poco si adattano a situazioni di utilizzo più generali.

La differenza principale tra le due tipologie di schema è illustrata nella figura seguente:

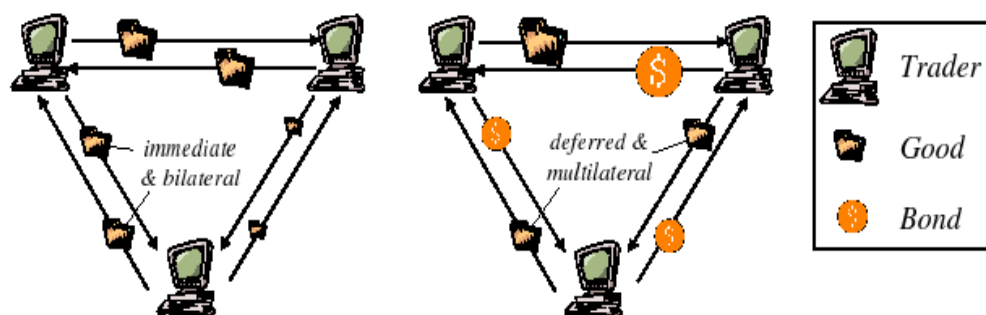


Immagine 4: schemi ad incentivi barter-based e bond-based

Mentre il modello barter-trade richiede la restituzione immediata di un servizio alla ricezione e può funzionare solamente in modo bilaterale, i modelli bond-based sono più flessibili in quanto permettono uno scambio multilaterale e ritardato delle risorse. Infatti un peer può compensare i servizi precedentemente ricevuti erogandone ad un altro peer più in là nel tempo.

Inoltre il modello barter-trade richiede che ogni peer condivida una quantità di risorse uguale a quella che riceve da altri peer.

I modelli bond-based, nei quali una metodologia di valutazione delle risorse condivise è implicitamente presente, semplificano lo scambio di risorse di diversa tipologia aventi valori differenti.

1.6 Scopo della tesi

Lo scopo di questa tesi e del progetto implementato è stato quindi quello di creare un protocollo che permettesse lo scambio di una moneta virtuale tra i peer partecipanti ad un sistema peer to peer. Un sistema di questo genere può essere utilizzato per implementare un modello ad incentivi bond-based.

L'utilizzo di un sistema di questo genere non si limita solo all'incentivazione. Basando l'erogazione della moneta virtuale in base all'effettivo pagamento di denaro reale è possibile estendere il modello ad incentivi permettendo di avere un sistema in cui è possibile vendere un determinato servizio.

In un sistema di questo tipo i peer possono addirittura guadagnare per le risorse condivise.

In questo modo si garantisce un sicuro interesse da parte dei peer nel comportarsi correttamente ai fini del raggiungimento dello scopo comune.

2. Obiettivi di un sistema di accounting

Durante la progettazione e la successiva implementazione del sistema di accounting sono stati individuati una serie di obiettivi che hanno guidato lo sviluppo in tutte le sue fasi.

2.1 Decentralizzazione

Considerando che il sistema sviluppato doveva essere utilizzato come componente aggiuntivo ad una rete peer to peer esistente era assolutamente necessario che il sistema fosse il più possibile decentralizzato al fine di eliminare completamente la presenza di single point of failure e permettere la scalabilità del sistema.

È stato quindi assunto un approccio completamente peer to peer eliminando, ove possibile, componenti centralizzati.

2.2 Efficienza

Il sistema realizzato non offre, di per se, un servizio desiderabile dall'utente finale. È qualcosa che viene utilizzato a fianco di altre tecnologie peer to peer al fine di garantire e incentivare la cooperazione fra i peer.

Era necessario quindi che il sistema fosse il più possibile leggero sotto tutti i punti di vista (banda, cpu, memoria) al fine di non incidere negativamente sulle prestazioni dell'eventuale applicazione peer to peer affiancata.

Inoltre si voleva che il sistema potesse essere utilizzato anche su host dalle risorse hardware non elevate al fine di essere utilizzato in componenti embedded o con poche risorse: palmari, cellulari etc..

2.3 Scalabilità

Il sistema deve poter essere utilizzato su reti peer to peer di dimensioni planetarie. Il sistema deve quindi poter scalare bene. Le performance non devono abbassarsi all'aumentare dei peer nella rete bensì rimanere costanti, garantendo la stessa Quality of Service.

2.4 Affidabilità

Un sistema di questo tipo deve essere assolutamente affidabile. Non possono essere tollerati interruzioni di servizio o degrado delle prestazioni fino a rendere inutilizzabile il sistema. A tal fine è necessario implementare sistemi di ridondanza dei dati e bilanciamento del carico di lavoro.

2.5 Sicurezza

Un sistema di questo tipo non può che avere la sicurezza come obiettivo centrale. Non si può permettere che il sistema possa essere vulnerabile. I peer devono accedere ad un servizio erogato solamente dopo aver pagato la loro quantità di moneta virtuale. La moneta virtuale non può essere spesa più volte o

creata in modo fraudolento.

2.6 Persistenza

I dati associati agli account dei vari peer non possono essere persi. Devono essere disponibili anche a distanza di mesi dall'ultimo utilizzo. Questo è un obiettivo chiave in quanto i peer entrano ed escono continuamente dal sistema e quindi gestire la persistenza è un compito complesso.

2.7 Consistenza

I dati forniti dal sistema devono assolutamente essere consistenti. Non è ammissibile che il sistema riporti dati errati o differenti da quelli previsti, anche in presenza di peer maliziosi o non affidabili.

2.8 Flessibilità

Il sistema sviluppato è stato pensato e progettato per permetterne l'utilizzo in diversi scenari. Si è quindi cercato di rendere il sistema il più possibile slegato dalle logiche dell'applicazione supportata.

3. Overlay Network e Pastry

Uno dei problemi chiave nelle applicazioni peer-to-peer su larga scala è quello di fornire efficienti algoritmi per il routing dei messaggi attraverso la rete.

Le Distributed Hash Tables (DHT) sono una classe di sistemi distribuiti decentralizzati che costituisce la base per una rete p2p robusta ed efficiente. Una DHT è costituita da un set di chiavi astratto piuttosto vasto chiamato keypace e da un insieme di nodi fra i quali il keypace viene partizionato secondo uno schema di mapping chiave-nodo specifico per ogni implementazione.

Tutti i nodi che fanno parte della DHT sono connessi tramite una rete di link logici detta overlay network poiché costruita a livello applicativo, al di sopra della rete Internet.

Una overlay network può essere sviluppata secondo diverse topologie, ma generalmente le reti di sovrapposizione presentano una struttura che connette direttamente, tramite link logici, ciascun nodo ad un ristretto insieme di nodi “vicini”.

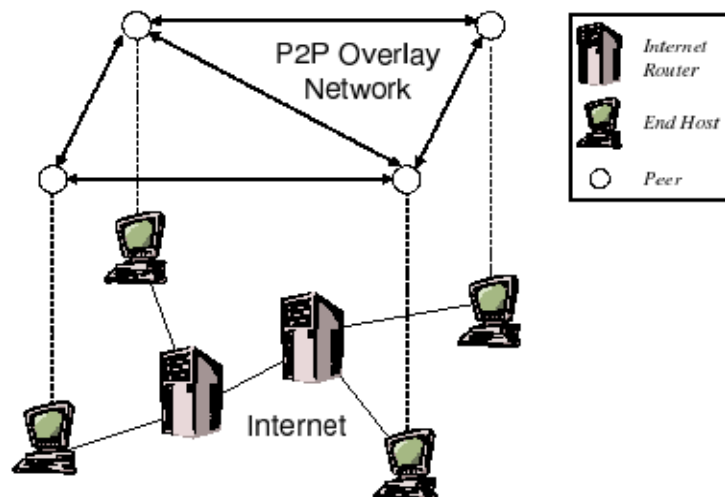


Immagine 5: overlay network

Ad ogni nodo è generalmente associata una chiave (identificatore) appartenente al keypace e i suoi vicini formano un insieme di nodi (a cardinalità fissa o variabile) i cui identificatori sono i più vicini all' identificatore del nodo in questione, secondo la relazione d'ordine definita sul keypace. Solitamente è sempre sulla base delle chiavi associate ai nodi che viene stabilita la partizione del keypace, ad esempio mappando su un nodo tutte le chiavi i cui valori sono più vicini al valore del suo identificatore piuttosto che a quelli degli identificatori di qualunque altro nodo.

L'overlay network fornisce una funzionalità di routing fra i nodi che permette di spedire un messaggio, contrassegnato da una chiave appartenente al keypace, garantendo che, dopo un certo numero di hops fra i nodi, questo verrà consegnato al nodo al quale è stata assegnata una partizione dello spazio delle chiavi all'interno della quale sia contenuta la chiave che identifica il messaggio; una

DHT si comporta quindi, in prima approssimazione, in modo simile ad una hash table basata su liste di collisione, considerando che ogni nodo può essere interpretato come un bucket al quale sono assegnati oggetti contrassegnati da un particolare insieme di valori di un codice di hash.

3.1 *Proprietà delle DHT*

Le applicazioni peer-to-peer a partire da quelle di “seconda generazione” sono basate su DHT perché queste enfatizzano particolari proprietà di efficienza fondamentali per la realizzazione di una buona applicazione p2p:

- **Decentralizzazione:** i nodi formano collettivamente il sistema senza alcun coordinamento centrale
- **Scalabilità:** il sistema è predisposto per un funzionamento efficiente anche con diversi milioni di nodi.
- **Tolleranza ai guasti e manutenibilità:** il sistema dovrebbe risultare affidabile anche in presenza di nodi che entrano, escono dalla rete o sono soggetti a malfunzionamenti con elevata frequenza.

La proprietà di scalabilità è diretta conseguenza dell'efficienza degli algoritmi di routing dei messaggi forniti dalle overlay network: meno cresce, al crescere del numero di nodi presenti sulla rete, il numero massimo di hop necessari per instradare un messaggio fino alla corretta destinazione, più sarà possibile aumentare il numero di nodi senza che le prestazioni del sistema degradino in modo sensibile.

Le più comuni topologie di DHT sono strutturate in modo tale che la dilazione, ovvero il numero massimo di hop nella procedura di routing, sia $O(\log n)$, con n pari al numero di nodi che costituiscono la rete. È altrettanto importante, ai fini della scalabilità, che la dilazione sia bassa anche per reti sulle quali i nodi mantengono un numero molto ridotto, ed eventualmente costante, di riferimenti ai nodi vicini.

La limitatezza della cardinalità dell'insieme dei vicini conosciuti da ogni nodo è fondamentale anche per garantire la proprietà di tolleranza ai guasti e manutenibilità: più ridotto è il numero dei link mantenuti da ogni nodo, meno sarà oneroso, soprattutto a livello di numero di messaggi scambiati, aggiornare la struttura della rete in occasione dell'arrivo o della dipartita di un nodo.

Se sfruttato opportunamente e se corredato di operazioni adatte implementate a livello della overlay network, il meccanismo delle DHT può essere utilizzato per costruire una vasta gamma di applicazioni p2p, a cominciare da quelle di storage distribuito e sharing di contenuti; nel seguito verrà esaminato approfonditamente il caso di Pastry, un substrato per applicazioni peer-to-peer basato su DHT, una implementazione del quale è stata utilizzata nella costruzione del prototipo che verrà descritto in questa trattazione.

3.2 *Pastry*

Pastry [6] è un efficiente substrato per applicazioni p2p sviluppato congiuntamente da gruppi di ricercatori del Microsoft Research di Cambridge

(UK) e Redmond (USA), della Rice University (USA) della Purdue University (USA) e della University of Washington (USA).

Pastry è definito come una self-organizing overlay network; ogni nodo Pastry inoltra dei messaggi di richiesta e interagisce con una o più istanze di applicazioni locali che la utilizzano; a ciascun nodo, all'atto dell'entrata nella rete, è assegnato un identificatore di 128 bit, generato casualmente o a partire da una chiave pubblica, detto nodeId: questo è utilizzato per indicare la posizione del nodo in uno spazio circolare di nodeId (il keyspace citato sopra) che va da 0 a 2^{128-1} .

Assumendo che la rete sia costituita da N nodi, Pastry è in grado di inoltrare un messaggio, in meno di $\log_2^b(N)$ passi, fino al nodo il cui nodeId è quello numericamente più vicino alla chiave del messaggio (dove b è un parametro di configurazione solitamente uguale a 4); questa funzionalità è resa disponibile grazie ad un particolare algoritmo greedy che instrada il messaggio verso il nodo con nodeId numericamente più vicino alla chiave del messaggio fra tutti i nodi conosciuti.

Nel seguito verranno illustrati alcuni aspetti del sistema Pastry utili anche ad approfondire i concetti generali sulle DHT e sulle overlay network esposti fino ad ora e ad apprezzare le caratteristiche per le quali Pastry è stato scelto come substrato per il prototipo oggetto di questa trattazione.

Stato dei nodi

Ogni nodo Pastry mantiene una serie di strutture dati (figura 1) necessarie per eseguire le principali funzionalità offerte dalla rete. L'insieme dei dati contenuti all'interno di queste strutture è chiamato stato del nodo.

La prima è la routing table, R, composta da righe di 2^b-1 entry ciascuna; ogni entry della generica n-esima riga si riferisce ad un nodo il cui nodeId condivide le prime n cifre con il nodeId del nodo su cui è mantenuta la tabella ma la cui n+1-esima cifra è differente. Ciascuna entry della tabella contiene l'indirizzo IP del nodo contrassegnato dall'Id al quale la entry si riferisce; se, per una data entry, non è conosciuto nessun nodo adatto, questa sarà lasciata vuota. Il numero di righe è, in media, « $\log_2^b(N)$ » poiché l'assegnazione casuale dei nodeId comporta una popolazione uniforme dello spazio dei nodeId.

La seconda struttura dati è il leaf set, L, un insieme di coppie <IP, NodeId>, solitamente con cardinalità pari a 2^b o a 2^{b+1} , che contiene i riferimenti agli $|L|/2$ nodi con i più piccoli nodeId maggiori e agli $|L|/2$ nodi con i più grandi nodeId minori dell'identificatore del nodo in questione.

L'ultima struttura dati è il neighborhood set, M, nel quale sono contenute coppie <IP, NodeId> relative ad un insieme di nodi (della stessa cardinalità del leaf set) che sono i più vicini, secondo una metrica di prossimità geografica, al nodo in questione.

È da notare come la scelta del parametro b implichi un contro-bilanciamento fra il numero massimo di passi per completare la procedura di routing e la dimensione delle strutture dati mantenute da ciascun nodo sulla rete.

Mentre la routing table ed il leaf set sono indispensabili per l'esecuzione dell'algoritmo di routing, il neighborhood set è utile per mantenere alcune buone proprietà di località che saranno discusse in seguito.

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

La figura adiacente, tratta da [6], rappresenta lo stato di un ipotetico nodo Pastry il cui nodeId è 10233102. Tutti i numeri sono rappresentati in base 4. La prima riga della routing table è la riga 0. Le caselle scurite nella generica n-esima riga contengono l'n-esima cifra del nodeId del nodo presente. I nodeId delle caselle sono stati suddivisi in una tripla <prefisso comune con 10233102 - cifra successiva - resto del nodeId>.

Gli indirizzi IP associati ai nodeId non sono mostrati.

Procedura di Routing

La procedura di instradamento in Pastry è definita come key-based routing perchè eseguita sulla base della chiave k che contrassegna il messaggio che deve essere instradato e degli identificatori dei nodi che determinano la partizione del keyspace. La figura mostra una tipica rappresentazione dello spazio circolare degli identificatori dei nodi: per questa ragione la rete Pastry, al pari di molte analoghe overlay network, viene anche denominata ring.

Se k cade nel range di nodeId coperto dal leaf set, il messaggio è consegnato direttamente al nodo destinazione, ovvero il nodo il cui Id è quello numericamente più vicino a k (possibilmente il nodo locale). Se la chiave non cade nel range del leaf set, allora viene consultata la routing table e il messaggio è inoltrato verso il nodo il cui nodeId condivide con k un numero maggiore di cifre rispetto al nodeId del nodo presente; nel caso in cui questo non fosse possibile (ad esempio perché le entry appropriate sono vuote oppure perché il nodo selezionato risulta irraggiungibile) il messaggio è inoltrato verso un nodo il cui identificatore condivide con la chiave almeno lo stesso numero di cifre rispetto al nodeId del nodo presente ma che rispetto a questo risulta numericamente più vicino a k; un nodo che soddisfi questa proprietà esiste sempre.

La procedura di routing converge sempre, perché ogni passo inoltra il messaggio verso un nodo il cui nodeId è sempre numericamente più vicino alla chiave del messaggio, ed è possibile dimostrare che, in condizioni “normali”,

ovvero assumendo tabelle di routing accurate e in assenza di recenti fallimenti dei nodi i cui riferimenti sono mantenuti in memoria, il numero massimo di hops nella procedura di routing è $\llcorner \log_2 b(N) \gg$, come riportato in precedenza. Per stime precise ed ulteriori considerazioni si rimanda a [6].

Auto adattamento della Rete

Pastry possiede la caratteristica di essere self-organizing perché definisce un protocollo di gestione dell'arrivo e della dipartita dei nodi che mantiene l'integrità della rete anche nel caso di un elevato churn rate (ossia la frequenza con cui avvengono cambiamenti nella sua struttura).

Un nuovo nodo al quale sia stato ordinato di unirsi alla rete esistente ha bisogno di contattare un bootstrap node (che verrà chiamato A nel seguito) geograficamente prossimo che faccia già parte della rete Pastry, magari parte di una core network composta da nodi sempre attivi; il riferimento a tale nodo può essere ottenuto ad esempio tramite multicast IP con la tecnica dell'expandig ring oppure attraverso differenti canali esterni al sistema.

Si assume che l'Id del nuovo nodo sia X: il nodo X richiede ad A di inoltrare uno speciale messaggio di join con chiave uguale a X; come ogni altro messaggio, Pastry inoltra il messaggio fino al nodo Z numericamente più vicino ad X.

Ciascun nodo coinvolto nella procedura di routing del messaggio di join invierà come risposta al nodo X la propria routing table. Dall'informazione ottenuta, il nodo X può inizializzare la propria tabella di routing nel modo descritto di seguito.

Sia N_i l'i-esima riga della tabella di routing di un nodo con Id pari a N, e si consideri il caso generale in cui X ed A non abbiano un prefisso comune: la riga X_0 viene inizializzata con i valori contenuti nella riga A_0 , i quali, essendo identificatori che non condividono alcun prefisso con A, non condividono, per transitività, alcun prefisso con X; la riga X_1 viene costruita invece sulla base dell'informazione contenuta nella riga B_1 , dove B identifica il primo nodo dopo A coinvolto nella procedura di routing. Infatti, poiché B condivide la prima cifra con X (dato che A instrada il messaggio verso un nodo il cui Id condivide almeno la prima cifra con la chiave X del messaggio), allora B_1 contiene riferimenti adatti per l'inizializzazione della riga X_1 . In generale la riga M_i , dove M è l'Id del nodo coinvolto nella procedura di routing al passo i, contribuisce alla costruzione della routing table di X inizializzando la riga X_i .

Infine il leaf set del nodo X viene costruito sulla base del leaf set del nodo Z, essendo Z il nodeId numericamente più vicino a X, e il neighborhood set di X è ricavato da quello di A, il quale è stato scelto con un principio di prossimità geografica.

Una volta inizializzato il suo stato, eventualmente ricorrendo a richieste addizionali rispetto a quelle definite dal protocollo di routing del messaggio di join, il nuovo nodo avvisa della sua presenza tutti i nodi che necessitano di essere avvertiti del suo arrivo.

Nel caso di dipartita di un nodo è invece necessario aggiornare opportunamente lo stato dei nodi che mantengono riferimenti a nodi non più attivi.

Ogni nodo Pastry invia periodicamente dei messaggi per verificare l'attività di tutti i membri del proprio leaf set; per rimpiazzare il nodo non più attivo in un leaf set (ovvero che non risponde ai messaggi di verifica per un determinato periodo di tempo), il nodo che riscontra la non validità del riferimento richiede il leaf set del nodo, il cui riferimento è memorizzato nella metà del leaf set alla quale apparteneva il nodo caduto, avente il nodeId numericamente più distante dal proprio Id; dall'insieme di riferimenti ottenuti (che sarà parzialmente sovrapposto al leaf set del nodo che ha effettuato la richiesta) il nodo sceglierà un riferimento adatto per rimpiazzare quello non più valido.

Per rimpiazzare una entry non valida nella routing table, il nodo interessato richiede il valore di quella stessa entry ad uno dei nodi i cui riferimenti sono memorizzati nella stessa riga di quello non valido; nel caso in cui questo non sia possibile, la stessa procedura di richiesta viene ripetuta nei confronti dei nodi i cui riferimenti sono memorizzati nella riga successiva.

Infine, gli elementi del neighborhood set, la cui validità è verificata periodicamente al pari degli elementi del leaf set, sono rimpiazzati, se non più validi, utilizzando l'informazione contenuta nei neighborhood set dei vicini, similmente a quanto accade per il leaf set.

Risultati sperimentali pubblicati su [6] dimostrano che il costo di riparazione (in termini di messaggi scambiati) dello stato di un nodo è molto ridotto anche in presenza di un'elevata percentuale di guasti dei nodi e che il protocollo di auto-mantenimento della rete risulta efficace anche in prolungate situazioni di massicci e repentini cambiamenti della propria struttura.

Località

Dato che gli identificatori dei nodi sono assegnati casualmente, non esiste nessuna relazione fra la prossimità numerica di due nodeId e la prossimità geografica delle corrispondenti macchine sulle quali vengono eseguite le applicazioni Pastry. Questa caratteristica conferisce alla rete Pastry (come, in generale, ai sistemi basati su DHT) una particolare robustezza, determinata dal fatto che un guasto esteso ad un'intera area geografica (ad esempio un blackout) si traduce a livello di overlay network in una serie di piccoli guasti distribuiti su tutta la rete e facilmente riparabili tramite l'algoritmo discusso nel paragrafo precedente.

Tale proprietà potrebbe però comportare significative inefficienze nella procedura di routing; basti pensare all'instradamento di un messaggio che deve essere consegnato ad un nodo il cui nodeId è numericamente molto distante dal nodeId del mittente nel caso in cui le macchine del mittente e del destinatario siano geograficamente prossime: il messaggio sarebbe inoltrato, con un elevato numero di IP hops, attraverso nodi dislocati nella Internet mondiale per giungere infine ad un nodo distante soltanto pochi IP hops dal mittente.

Pastry garantisce però il rispetto di un principio di località che rende efficiente

la procedura di routing anche riguardo ad una metrica di prossimità geografica.

Il concetto di “prossimità” in Pastry si basa su una metrica scalare, ad esempio sul numero di IP hops o sulla distanza geografica fra due nodi. Si assume che lo spazio definito dalla metrica di prossimità sia euclideo, per il quale quindi valga la disuguaglianza triangolare (ovvero quel principio per cui la distanza fra A e B sommata alla distanza fra B e C non è mai minore della distanza fra A e C).

Per far sì che l'algoritmo di routing sia realmente efficiente è necessario mantenere la proprietà per cui le entry all'interno di una generica riga della routing table si riferiscono a quei nodi che, rispetto a tutti gli altri nodi attivi aventi un identificatore con il prefisso adatto per quella riga, sono vicini al nodo locale secondo la metrica di prossimità geografica scelta; assumendo che la proprietà sia verificata per una rete già esistente, è possibile mostrare che, all'atto dell'aggiunta di un nuovo nodo, questa rimane valida.

Sia A il nodo che riceve il messaggio di join direttamente dal nuovo nodo X: A₀ contiene riferimenti a nodi che, per ipotesi, sono vicini ad A, e dato che X è vicino ad A (perché la procedura di avvio di un nodo seleziona un bootstrap node vicino al nodo locale) e nello spazio considerato vale la disuguaglianza triangolare, X e i nodi in A₀ saranno relativamente vicini. Si consideri ora la riga X₁, ottenuta dal nodo B: i nodi in B₁ sono vicini a B, tuttavia ad una distanza da esso sensibilmente maggiore rispetto alla distanza che intercorre tra A e B; questo accade perché le entry contenute in ogni successiva riga nella routing table sono selezionate da un insieme esponenzialmente decrescente, dato che il prefisso dei nodeId che deve essere in comune con il nodeId del nodo che mantiene la tabella aumenta di una cifra ad ogni riga. Perciò, essendo X e A vicini ed essendo B, in media, sensibilmente più vicino ad A che non ai nodi in B₁, è ragionevole affermare che i nodi in B₁ sono fra i più vicini ad X fra tutti i nodi che posseggono un prefisso adatto per la riga X₁ della routing table di X.

Le medesime considerazioni possono essere fatte per tutti i rimanenti passi dell'instradamento del messaggio di join, dimostrando che, all'aggiunta di un nuovo nodo nella rete, la proprietà iniziale rimane invariata.

Data questa caratteristica della tabella di routing, è possibile osservare che, ad ogni passo della procedura di instradamento, un messaggio viene inoltrato ad un nodo ad una distanza geografica sempre maggiore rispetto al nodo mittente e, sebbene l'algoritmo di routing non garantisca l'adozione del percorso minimo, l'allontanamento progressivo dalla fonte del messaggio dà origine a “buoni” instradamenti. È inoltre interessante notare che, ad ogni successivo hop nell'instradamento su overlay network corrisponde una distanza geografica esponenzialmente sempre più grande sulla rete Internet.

4. Schema di Protocollo

Descriviamo ora in modo dettagliato il protocollo applicativo progettato.

4.1 Assunzioni Iniziali

Ogni utente del sistema possiede una chiave asimmetrica composta da una chiave pubblica e da una chiave privata. La chiave asimmetrica deve venire firmata a sua volta con la chiave di una Certification Authority, garantendo così la bontà delle chiavi utilizzate.

Si presuppone che il protocollo qui analizzato venga implementato sopra ad una overlay network della tipologia di Pastry. Per ogni utente che voglia utilizzare il sistema viene creato un nodo. Il `nodeId` del nodo viene calcolato a partire dalla chiave pubblica associata all'utente.

Durante lo scambio di messaggi nel protocollo, i vari nodi firmano le informazioni contenute, per garantire la non contraffazione del messaggio.

4.2 Operazioni supportate

Il protocollo applicativo implementato supporta le seguenti operazioni:

- **balance request:** viene invocata quando un peer vuole richiedere lo stato del suo account.
- **cash flow:** permette di inviare della moneta virtuale da un peer ad un altro
- **funding:** aggiunge una quantità di denaro reale all'account virtuale di un peer
- **withdrawal:** rimuove una quantità di denaro dall'account virtuale di un peer e permette la ritrasformazione in moneta reale.

Inoltre, al fine di garantire la persistenza dei dati memorizzati dal sistema, i peer dovranno reagire alle modifiche del proprio leafset, reagendo quindi alle situazioni di entrata e uscita di un peer dal proprio leafset.

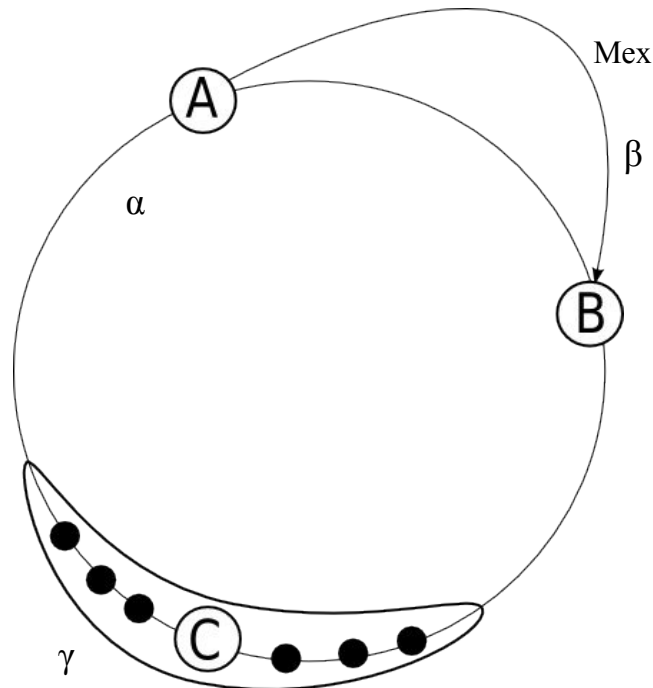
Dovranno quindi essere analizzate e implementate le operazioni di **node join** e **node leave**, attivate rispettivamente all'entrata e all'uscita di un peer dal sistema.

Nelle pagine seguenti verranno analizzate in modo dettagliato le operazioni del protocollo cercando di entrare in modo dettagliato nelle varie interazioni tra i peer coinvolti.

4.3 Convenzioni grafici

Al fine di migliorare la comprensione delle interazioni del protocollo progettato, ogni fase in ogni operazione è accompagnata da una serie di grafici.

Per migliorare la leggibilità di questi grafici vengono qui elencate le convenzioni utilizzate per la loro creazione.



La figura sopra rappresenta l'overlay network in uno degli stati dell'applicazione. Il cerchio α rappresenta il ring Pastry. Ovviamente viene assunto che il numero dei nodi partecipanti all'applicazione sia elevatissimo, nonostante non vengano visualizzati tutti.

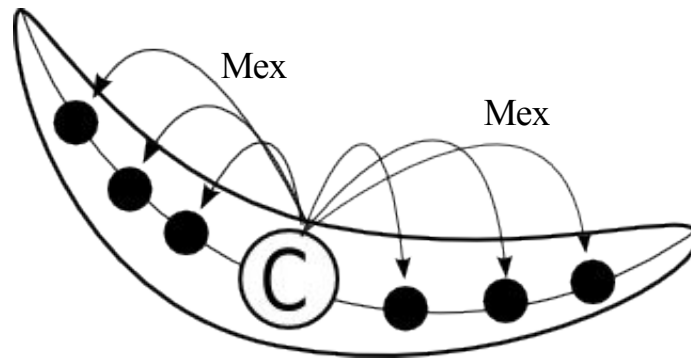
I nodi A, B, C sono peer che partecipano all'applicazione nel determinato stato. La freccia β rappresenta invece un messaggio che viene mandato da un nodo ad un'altro nel senso della freccia. Nella figura sopra il messaggio viene inviato dal nodo A al nodo B. Insieme ad ogni freccia viene riportato il nome del messaggio trasmesso (in questo caso *Mex*).

Il gruppo di nodi evidenziato dall'ellisse γ rappresenta invece un leafset, il quale nodo radice viene disegnato nel centro. In questo caso il nodo radice del leafset γ è C. In nero e senza etichetta vengono disegnati i nodi facenti parte del leafset.

Il root node svolge un compito molto importante all'interno di un leaf set, come verrà evidenziato successivamente nella trattazione.

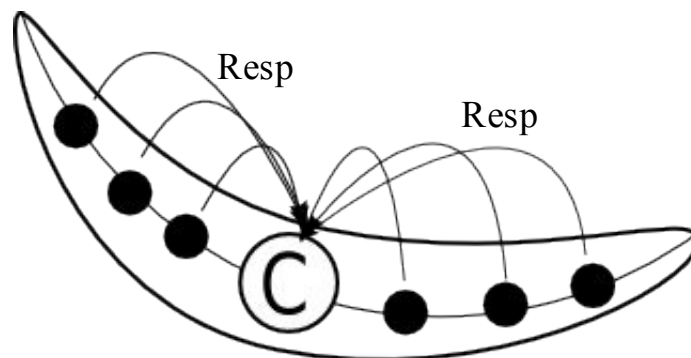
Vengono ora dettagliati lo scambio di messaggi tra il root node e i nodi appartenenti al leaf set.

Propagazione messaggio e Response



La figura sopra visualizza lo standard con cui è stata rappresentata la propagazione di un messaggio dal root node ai nodi del leafset. Le frecce rappresentano i messaggi inviati dal nodo C ai nodi appartenenti al proprio leafset.

Ad ogni messaggio di propagazione i nodi riceventi inviano un response al root node che lo ha generato. La figura sottostante mostra lo standard con cui viene rappresentato il recupero dei messaggi di response da parte del root node.



4.4 Funzioni e convenzioni di utilità

Al fine di migliorare la trattazione vengono ora definite una serie di funzioni che verranno utilizzate in tutta la spiegazione del protocollo.

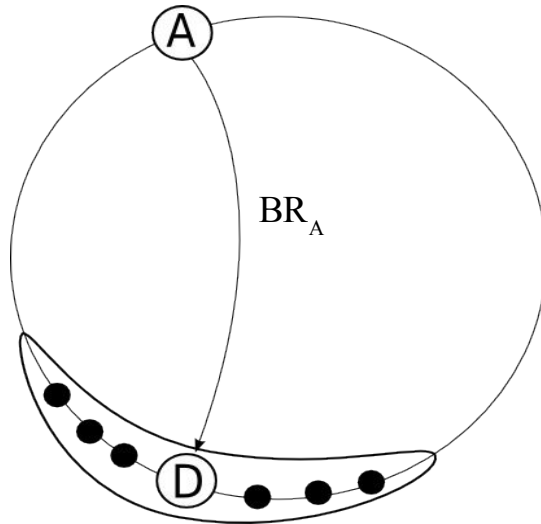
Definiamo:

- K_A chiave asimmetrica del nodo A: la chiave asimmetrica associata al nodo A. A partire da questa chiave vengono generate le rispettive chiavi pubbliche e private
- K_A^+ chiave pubblica del nodo A
- K_A^- chiave privata del nodo A
- $S_{K^-}(M)$ funzione firma: dato un messaggio M, restituisce la firma digitale calcolata a partire da M utilizzando la chiave privata K^- .
- $V_{K^+}(M, S)$ funzione verifica: dato un messaggio M ed una firma digitale S, restituisce *true* o *false* a seconda che la verifica della firma S a partire dal messaggio M sia andata a buon fine.
- \parallel funzione concatenazione: dati due componenti, siano essi messaggi o codici, ne restituisce la concatenazione, appende cioè al primo il contenuto del secondo.
- **m1 = “informazione base”** informazione base portata in un messaggio
- $Mex = \{m1, m2, \dots\}$ messaggio composto: dati diversi componenti base di un messaggio, questa scrittura rappresenta il messaggio che porta in se tutte le informazioni base definite
- Mex_A messaggio firmato digitalmente dal nodo A
- $Z(A)$ funzione accountRoot: dato un nodo A, restituisce il nodeId del root node responsabile per l'account di A.
- L_C leaf set associato al nodo C

4.5 Balance Request

Precondizioni: Il nodo A, vuole conoscere lo stato del suo account, cioè vuole avere dal sistema la quantità di denaro che è depositata sul suo account virtuale.

1. invio richiesta



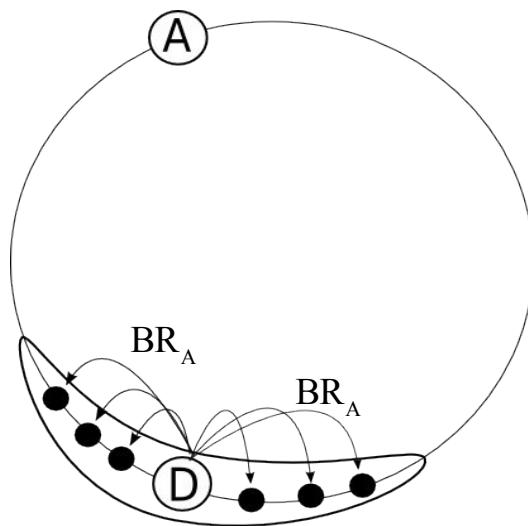
BR = "Balance Request"

$$BR_A = BR \parallel S_{Ka}-(BR)$$

$$D = Z(A)$$

Il nodo A vuole conoscere lo stato del suo account. Per prima cosa invia al nodo D, account root per l'account di A, una richiesta di balance request firmata.

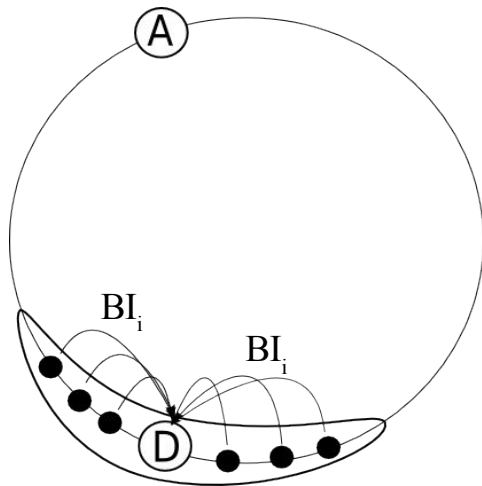
2. propagazione della richiesta di balance



Il nodo D riceve la richiesta di balance proveniente da A. D capisce di essere l'account root per il nodo A. D deve quindi inoltrare la richiesta a tutti i nodi appartenenti al proprio leaf set.

Ogni nodo che riceve la balance request inoltrata da D accede in memoria all'account di A prendendone le informazioni di balance.

3. raccolta informazioni di response



BI = “Balance Information”

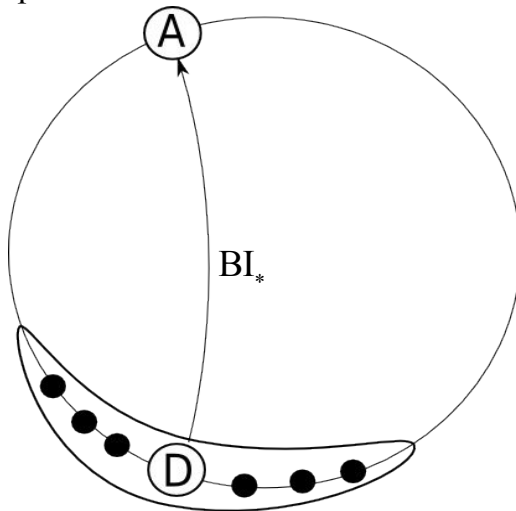
$$Bi_i = BI \parallel S_{ki}-(BI)$$

Dopo aver recuperato le informazioni dell'account del nodo A, ogni nodo i appartenente al leaf set di D le invia all'account root D.

Il nodo D raccoglie queste risposte (aspetterà fino allo scadere di un timer) memorizzandole.

Non appena ricevuti abbastanza response (la percentuale di successi è configurabile) procede al passo successivo.

4. risposta finale



$$BI_* = \{Bi_i \mid i \in \text{leafset } D\}$$

Il nodo D ha ricevuto abbastanza risposte, le concatena in un messaggio e invia la risposta finale al nodo A.

Alla ricezione di BI_* , il nodo A analizzerà i dati contenuti nel messaggio.

A dovrà attuare una decisione per maggioranza: assumerà come risultato corretto quello riportato da più nodi.

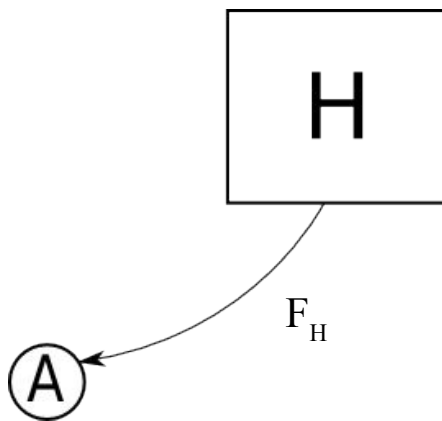
Postcondizioni: Il nodo A riceve le informazioni sul suo account o un messaggio di errore appropriato.

4.6 Funding

Precondizioni: L'utente proprietario del nodo A vuole aggiungere del denaro reale al suo account virtuale. Egli esegue una transazione, usando una carta di credito o PayPal, versando il denaro ad una banca centrale. Questo procedimento avviene al di fuori del sistema peer to peer, ad esempio tramite il sito web sicuro della autorità centrale / banca.

Alla ricezione del pagamento l'autorità centrale / banca rilascia un particolare messaggio, firmato con la propria chiave privata, che verrà utilizzato all'interno del sistema.

1. generazione messaggio funding



H autorità centrale / banca

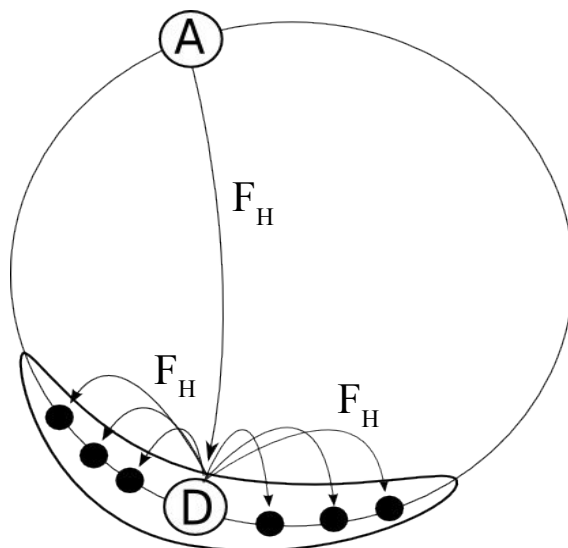
M = "Quantità di Denaro"

$F = \{M\}$

$F_H = F \parallel S_{KH}(F)$

L'utente associato al nodo A riceve dal sito web un messaggio firmato dall'autorità centrale che è una ricevuta dell'avvenuto pagamento di moneta reale.

2. aggiornamento del conto



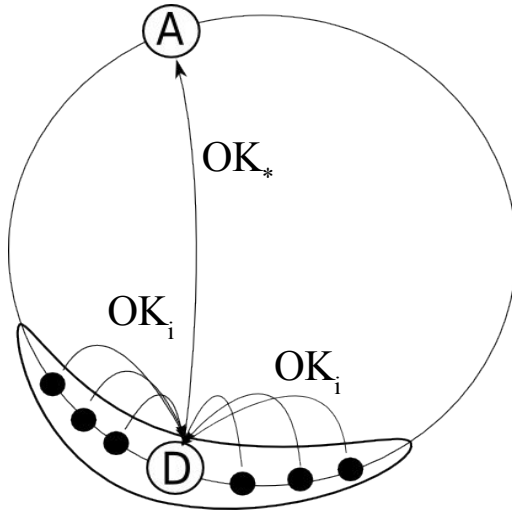
$D = Z(A)$

L'utente inserisce questo messaggio all'interno dell'applicazione peer to peer, tramite interfaccia grafica.

L'applicazione controlla la bontà del messaggio (la firma dell'autorità centrale) e, se corretta, lo invia al nodo root del proprio account.

L'account root D inoltra il messaggio a tutti i nodi del suo leafset. Alla ricezione del messaggio i nodi, dopo aver controllato la bontà del messaggio (firma), aggiornano il balance dell'account.

3. conferma deposito



$Ok_i =$ “Messaggio di conferma del nodo i ”

$$Ok_* = \{Ok_i \mid i \in \text{leafset } D\}$$

Dopo aver aggiornato il balance dell'account da incrementare, ogni nodo i appartenente al leafset di D invia un messaggio di conferma a D .

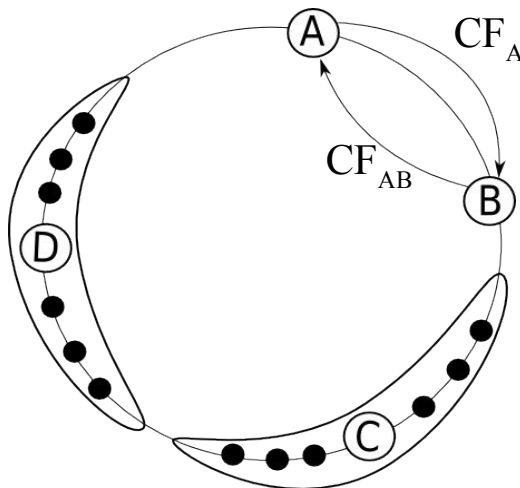
D memorizza tutti i messaggi di conferma ricevuti. Non appena raggiunta una soglia di successi (configurabile), tutti i messaggi vengono concatenati e inviati al nodo A .

Postcondizioni: L'account del nodo A è stato incrementato della quantità di denaro inserita nel messaggio generato dall'autorità centrale. In caso di errore questo viene segnalato.

4.7 Cash Flow

Precondizioni: Il nodo A vuole inviare una quantità di moneta virtuale al nodo B .

1. negoziazione della transazione



$M =$ “Denaro da trasferire”

$AID =$ “Identificativo del nodo A ”

$BID =$ “Identificativo del nodo B ”

$$CF = \{M, AID, BID\}$$

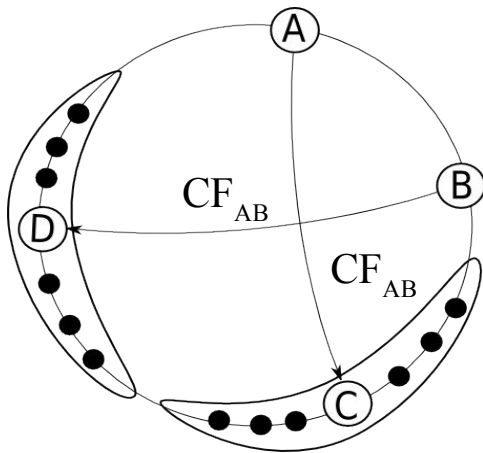
$$CF_A = CF \parallel S_{KA}-(CF)$$

$$CF_{AB} = CF \parallel S_{KA}-(CF) \parallel S_{KB}-(CF)$$

Il nodo A crea un messaggio di Cash Flow contenente la quantità di denaro da trasferire, la sorgente del denaro e la destinazione.

Alla ricezione del messaggio il nodo B aggiunge la propria firma digitale e invia il messaggio così composto ad A .

2. Aggiornamento del conto



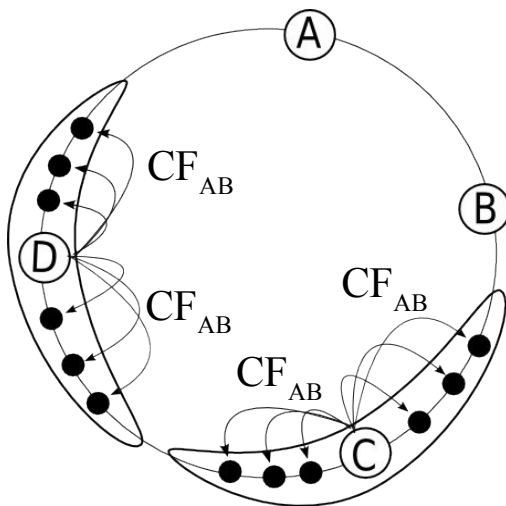
$$D = Z(A)$$

$$C = Z(B)$$

Una volta completata la negoziazione, i nodi iniziano la procedura di aggiornamento del conto. I nodi A e B invieranno il messaggio di Cash Flow firmato da entrambi all'account root del nodo opposto.

Quindi A contatterà l'account root di B e B contatterà l'account root di A.

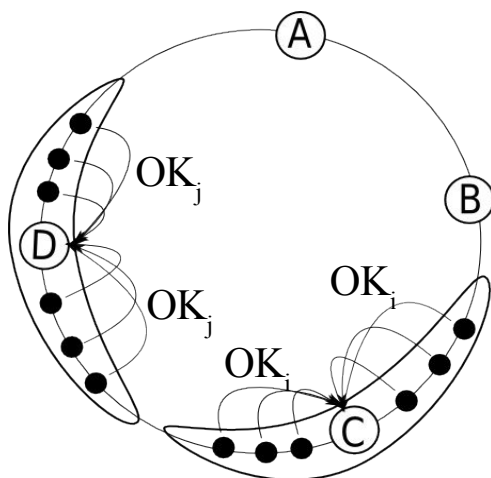
3. propagazione dell'aggiornamento



Alla ricezione del messaggio di cash flow da parte degli account root, il messaggio viene propagato ad ogni nodo i facente parte del rispettivo leafset.

Ogni nodo i nei rispettivi leafset controllano la bontà del messaggio propagato e, se il messaggio è "buono", aggiornano le il conto del nodo di cui sono responsabili.

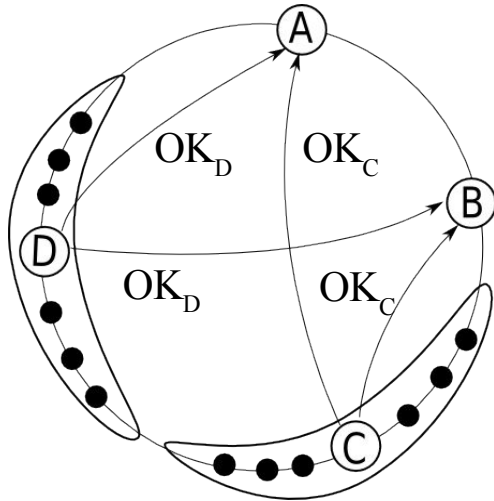
4. conferma della propagazione



Una volta aggiornato il conto, un messaggio di conferma viene restituito all'account root da ogni nodo i partecipante al leafset.

Gli account root memorizzano le risposte ricevute.

5. conferma del cash flow



$$Ok_C = \{Ok_i \mid i \in \text{leafset } C\}$$

$$Ok_D = \{Ok_j \mid j \in \text{leafset } D\}$$

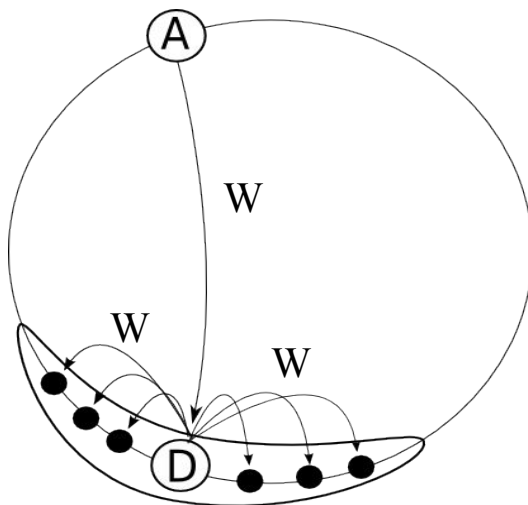
Dopo aver ricevuto sufficienti risposte dai nodi del leafset, gli account root creano un messaggio contenente tutte queste risposte. Inviano poi il messaggio ad entrambi i nodi coinvolti nel cash flow.

Postcondizioni: I rispettivi account dei nodi coinvolti nel CashFlow sono stati aggiornati. Dall'account del nodo A è stato rimosso del denaro, dall'account del nodo B è stato aggiunto del denaro. La quantità di denaro è M, specificata dal primo messaggio di CashFlow.

4.8 Withdrawal

Precondizioni: Il nodo A, che ha sul suo account una quantità di denaro virtuale, vuole riconvertire tutto o parte di questo denaro virtuale in denaro reale, che verrà poi depositato sul suo conto bancario reale.

1. richiesta di withdrawal e propagazione

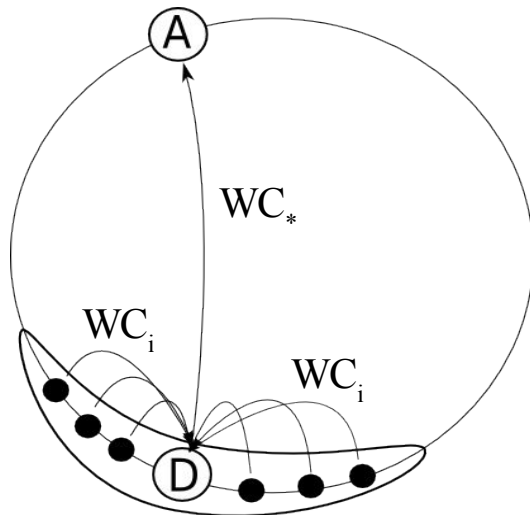


WC = "withdrawal confirm"

$$D = Z(A)$$

Il nodo A invia una richiesta di prelievo al proprio account root D. D una volta ricevuta la richiesta la propaga al proprio leafset.

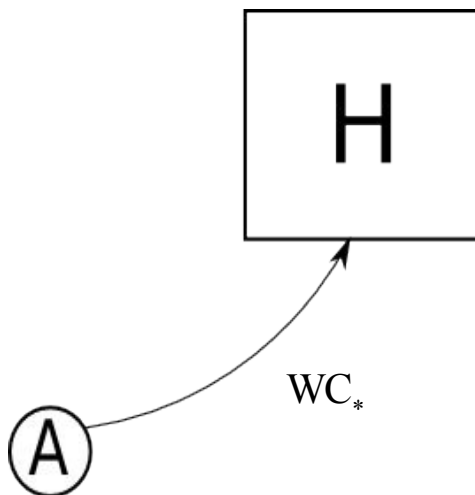
2. conferma withdrawal



WC = "withdrawal confirm"
 $WC_* = \{WC_j \mid j \in \text{leafset } D\}$

Il nodo D raccoglie le conferme al withdrawal generate dai nodi del proprio leafset concatenandole in un messaggio conferma complessiva che invierà poi al nodo A.

3. riconversione denaro



Per poter riconvertire il denaro virtuale in denaro reale l'utente associato al nodo A dovrà contattare l'autorità centrale, ad esempio tramite un sito web, e inviare il messaggio di conferma ricevuto dall'account root.

L'autorità centrale verifica le firme digitali incluse nel messaggio e può a questo punto procedere al deposito del denaro virtuale sul conto bancario dell'utente.

Postcondizioni: l'account virtuale del nodo A è stato effettivamente decrementato, mentre al conto bancario dell'utente associato al nodo A è stato aggiunto del denaro da parte dell'autorità centrale.

4.9 Miglioramenti al protocollo

Sebbene le modalità di interazione presentate fino ad ora possano sembrare ad una prima analisi ragionevoli e complete, è necessario complicare ulteriormente il protocollo al fine di evitare alcuni problemi riguardanti sicurezza ed affidabilità.

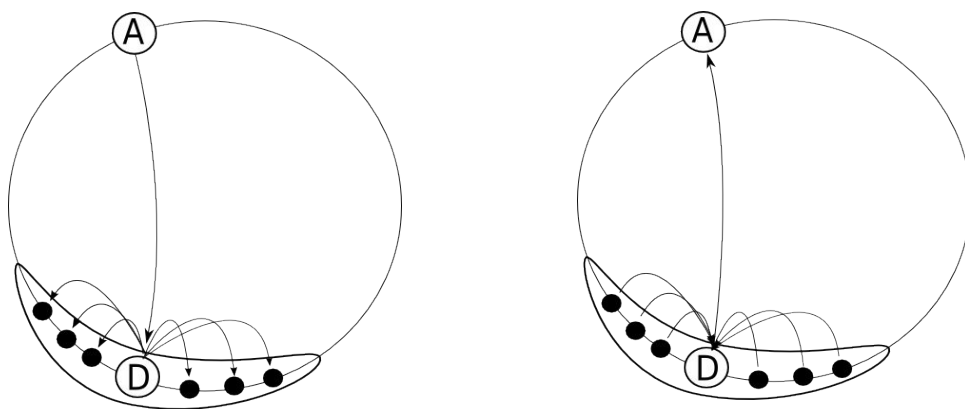
Affidabilità / Maliziosità dell'account root

Fino ad ora è stato assunto che l'account root associato ad un nodo segua

sempre il comportamento specificato dal protocollo. Questo approccio di “fiducia” ha permesso di implementare la propagazione dei messaggi al leafset in modo decisamente efficace.

Bisogna ricordare infatti che, in genere, i nodi del leafset associato ad un nodo sono già in contatto con il root node, in quanto appartenenti allo stesso neighborhood set, quindi lo scambio di messaggi con questi nodi non richiede ricerche o routing all'overlay network. In alcune implementazioni, come ad esempio FreePastry, non è neppure necessaria la creazione di una connessione tramite socket: infatti i vari nodi sono già in contatto tra loro per lo scambio di messaggi di manutenzione e verrà semplicemente riutilizzato il canale di comunicazione esistente.

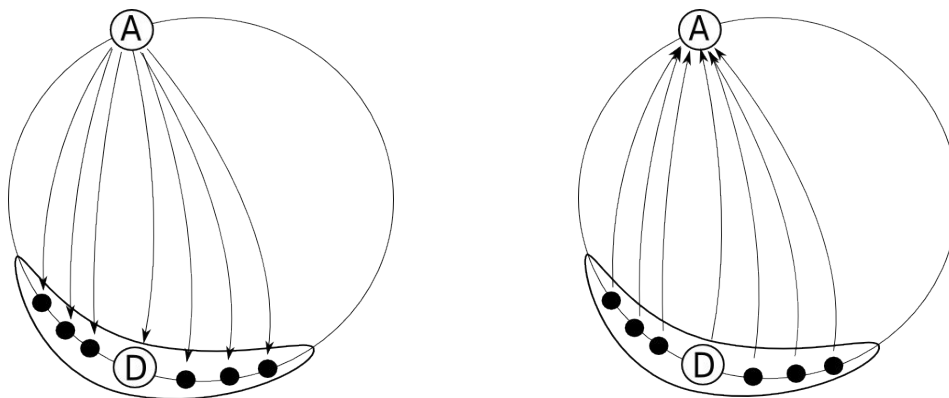
Purtroppo, però, bisogna osservare che, nelle interazioni rappresentate nella figura sottostante, l'account root D rappresenta, di fatto, un single point of failure.



Infatti, supponendo che il peer D fallisca o peggio ancora si comporti in modo malizioso, non possiamo garantire che l'interazione vada a buon fine e che l'account di A venga effettivamente modificato.

Se il peer D non inoltra il messaggio ricevuto dal nodo A, l'aggiornamento dell'account non può avvenire.

La soluzione al problema può essere solo quella di eliminare del tutto la dipendenza totale rispetto all'account root. Una soluzione possibile ed implementabile è rappresentata nella figura sottostante.



Il ruolo centrale dell'account root è stato eliminato, relegando il compito dell'invio/ricezione dei messaggi al nodo A.

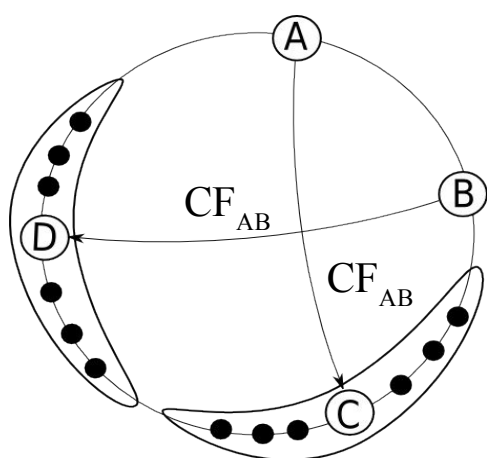
In questo modo però perderemmo tutti i vantaggi di prestazioni sopra descritti. Il procedimento di lookup handles (ricerca dei nodi responsabili per un account) è infatti decisamente costoso per l'overlay network.

Si è pensato quindi di utilizzare la soluzione sopra proposta solamente quando necessario, cioè quando la procedura standard, che fa affidamento al root node, fallisce. Così viene garantito il completamento dell'interazione anche in presenza di root node maligni o poco affidabili senza però perdere i vantaggi prestazionali della prima soluzione.

NOTA: La soluzione di fallback qui presentata, sebbene implementabile, non è stata considerata durante l'implementazione del progetto. Per coloro che intendono ampliare il progetto realizzato è consigliata l'analisi del metodo `getHandles()` all'interno della classe `PastImpl.java` facente parte del package `rice.p2p.past` delle librerie `FreePastry 2`. In questo metodo viene data una implementazione di questo tipo al problema della consegna dei messaggi di propagazione.

Frode durante Cash Flow

L'interazione di Cash Flow presentata ha un problema di sicurezza decisamente grave. La parte problematica è il passo 2 riportato qui sotto.

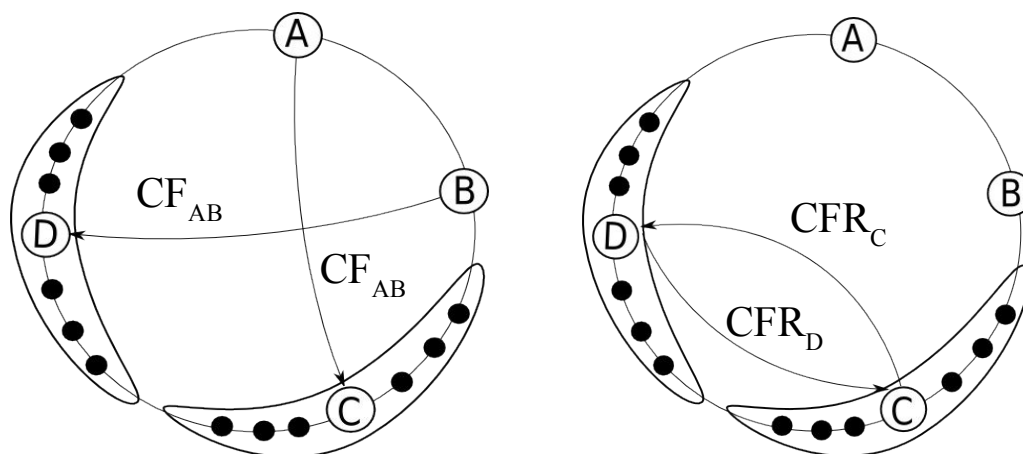


Se i nodi A e B sono maligni e vogliono frodare il sistema con la soluzione qui presentata lo possono fare abbastanza facilmente.

Il Cash Flow sta avvenendo nella direzione da A a B. Dopo aver negoziato la transazione (fase 1 Cash Flow) A aggiorna il conto di B, aggiungendo del denaro, mentre B non fa nulla.

In questo modo è stato aggiunto del denaro al conto del nodo B, senza che la stessa quantità di denaro venisse prelevata dal conto del nodo B. Un problema abbastanza serio!

La soluzione è data dal seguente raffinamento dell'interazione:



Un nodo che svolge la funzione di account root, alla ricezione di un messaggio di Cash Flow, prima di procedere alla propagazione della richiesta al proprio leafset, invia un messaggio di Cash Flow Received all'altro account root coinvolto nella transazione. Inoltre non procederà alla propagazione fino a che non avrà ricevuto l'analogo messaggio di Cash Flow Received da parte dell'altro account root.

In questo modo viene garantito l'effettivo decremento del bilancio del nodo che ha generato il Cash Flow.

NOTA: la soluzione qui presentata non è stata presa in considerazione durante l'implementazione del progetto, ma dovrebbe essere possibile implementarla senza troppe difficoltà.

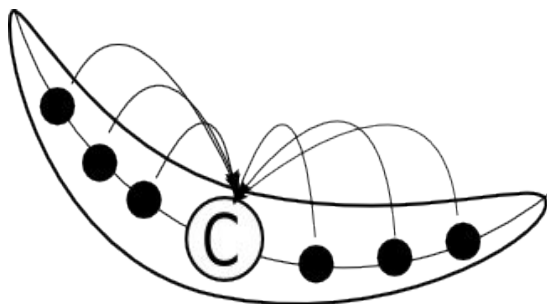
4.10 Node join e Node leave

I nodi del sistema sono, per definizione, non affidabili ed è necessario assumere che continuamente i nodi entrino ed escano dal sistema. Il substrato Pastry, come abbiamo già spiegato nei capitoli precedenti, reagisce alle modifiche che avvengono fra i nodi a lui “vicini”. Notiamo come, quando un nodo non risulta più disponibile, le strutture dati del leafset e del neighborhood set vengano automaticamente adattate alla situazione. Quando un nodo lascia un leafset automaticamente un'altro prende il suo posto, mantenendo l'invariante di struttura.

Le situazioni di node join e node leave producono quindi la stessa conseguenza sui nodi dello stesso leafset: un nuovo nodo ne entra a fare parte.

Il nodo appena entrato ovviamente non disporrà delle informazioni dei conti disponibili negli altri nodi del leafset. I nodi dovranno quindi comunicare al peer le informazioni da loro memorizzate. Il peer dovrà allocare nella propria memoria le informazioni ricevute dagli altri peer preparandosi quindi a far parte del sistema.

La situazione è rappresentata dalla seguente figura:



I nodi disegnati in nero vengono avvisati dal substrato della modifica del leafset e dell'arrivo del nodo C. Inviano quindi le informazioni possedute in memoria al nuovo arrivato. C alloca i nuovi account ed è pronto a contribuire al sistema.

4.11 Creazione di un account

Nel momento in cui un nodo entra nel sistema per la prima volta nessun account possiede le informazioni del conto del nuovo nodo.

Al fine di inizializzare il proprio account il nodo entrante nel sistema deve eseguire subito una invocazione dell'operazione di Balance Request.

I nodi facenti parte del leaf set associato all'account del nuovo nodo, vedendo arrivare una richiesta di balance per un account non disponibile, allocheranno lo spazio per il nuovo account impostandolo per default a zero.

5. Implementazione

In questo capitolo viene descritta l'esperienza implementativa del protocollo fino ad ora discusso.

5.1 *FreePastry*

Durante lo sviluppo è stato utilizzato FreePastry, una implementazione Open Source del protocollo Pastry. FreePastry è stato sviluppato presso la Rice University, utilizzando Java come linguaggio di programmazione.

La versione attualmente stabile di FreePastry è la 1.4.4, mentre il team di sviluppo è attualmente attivo nello sviluppo della versione 2, che si trova attualmente in stato di beta.

Il progetto può essere seguito tramite il sito <http://www.freepastry.org/>. Per coloro interessati ad avere maggiori informazioni è consigliato seguire lo sviluppo tramite la mailing list ufficiale dove ogni giorno vengono pubblicati gli aggiornamenti e gli sviluppi.

Chiunque si avvicini allo sviluppo con FreePastry può ottenere molte informazioni e una buona visione di insieme leggendo e seguendo passo passo il tutorial disponibile all'indirizzo <http://www.freepastry.org/FreePastry/tutorial/>. È disponibile anche una versione del tutorial specifica per la versione 2 beta di FreePastry all'indirizzo <http://www.freepastry.org/FreePastry/tutorial/preview/> (l'URL sarà probabilmente modificato all'uscita della versione 2 stabile).

Sebbene in principio fosse stato deciso di utilizzare la versione 1.4.4 per lo sviluppo dell'implementazione del protocollo di accounting, alcune delle caratteristiche innovative presenti nella versione 2 beta hanno convinto nel passaggio alla nuova versione.

La discussione di questa scelta viene rimandata successivamente nella trattazione.

5.2 *Approccio allo sviluppo*

La disponibilità del codice sorgente di applicazioni sviluppate sopra le API FreePastry, quali PAST, SCRIBE o SplitStream, ha permesso di prendere confidenza con tecniche di sviluppo di applicazioni distribuite decisamente entusiasmanti.

Le soluzioni adottate nei software sopra citati, in particolare PAST, sono state riutilizzate all'interno del progetto implementato. In questo modo è stato possibile usare tecniche altrimenti impensabili che hanno garantito una maggiore qualità dell'applicazione finale.

Continuation

Un esempio di soluzione estremamente funzionale utilizzata abbondantemente nel progetto è quella delle Continuation.

Una Continuation è simile ad un callback o, in Java, un Listener, ma viene utilizzata tipicamente solo una volta. L'utilizzo principale delle Continuation in FreePastry è quello di gestire la latenza della rete o altri tipi di operazioni di IO che siano bloccanti oppure possano richiedere molto tempo per l'esecuzione.

Per capire l'uso delle Continuation viene presentato un piccolo esempio di possibile utilizzo.

Durante l'implementazione del progetto avremmo potuto dichiarare il metodo per la richiesta di balance in questo modo:

```
public Result balanceRequest(NodeHandle nh)
```

Ma, dato che la richiesta potrebbe richiedere parecchio tempo a causa della latenza della rete, il programma si sarebbe bloccato fino al completamento dell'esecuzione del metodo. Nel frattempo sarebbe stato possibile eseguire altre chiamate alla rete ma, per come è strutturato questo metodo, non sarebbe stato possibile a meno di non ricorrere ad un utilizzo intenso dei Thread, sicuramente di difficile implementazione.

Le Continuation permettono di “continuare” ad eseguire mentre si sta aspettando il risultato di una chiamata bloccante.

Diamo uno sguardo all'interfaccia `rice.Continuation.java`:

```
/**
 * Asynchronously receives the result to a given method call, using
 * the command pattern.
 *
 * Implementations of this class contain the remainder of a
 * computation
 * which included an asynchronous method call. When the result to the
 * call becomes available, the receiveResult method on this command
 * is called.
 *
 * @version $Id: Continuation.java 3033 2006-02-06 15:32:48Z jstewart
 *
 * @author Alan Mislove
 * @author Andreas Haeberlen
 */
public interface Continuation {

    /**
     * Called when a previously requested result is now available.
     *
     * @param result The result of the command.
     */
    public void receiveResult(Object result);

    /**
     * Called when an exception occurred as a result of the
     * previous command.
     *
     * @param result The exception which was caused.
     */
    public void receiveException(Exception result);
}
```

Il metodo di richiesta di balance è stato quindi implementato così:

```
public void balanceRequest(NodeHandle nh, Continuation  
command)
```

Notiamo la differenza rispetto alla precedente implementazione: il metodo prende come argomento aggiuntivo una Continuation mentre non viene restituito nessun valore.

Quando il metodo `balanceRequest` avrà pronto un risultato invocherà `command.receiveResult()`. All'interno della classe che implementa Continuation sarà poi possibile gestire in modo appropriato il risultato del metodo. Altrimenti in caso di errore verrà chiamata `command.receiveException()` con una Exception appropriata.

In questo modo è possibile scrivere codice per l'esecuzione asincrona in modo estremamente semplificato, riducendo gli errori e i problemi riscontrabili in una implementazione a Thread.

5.3 COIN

Dimostrando una elevata fantasia da parte dell'autore, il progetto implementato è stato chiamato COIN.

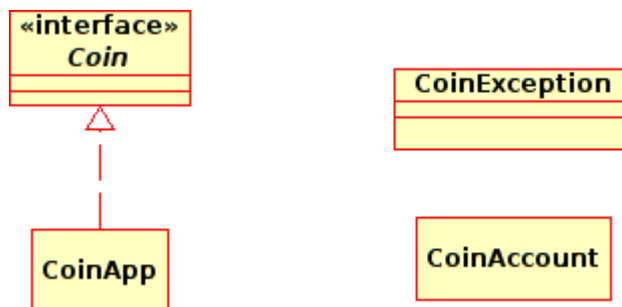
COIN è l'implementazione di un sistema di accounting per una rete distribuita che segue il protocollo descritto nei capitoli precedenti.

Package e Classi

Durante la stesura del codice si è cercato di isolare le varie aree funzionali dell'applicazione dividendo l'insieme delle classi in una serie di package Java. Verranno ora analizzati e rappresentati, dove utile, tramite diagramma UML.

```
unito.p2p.coin  
    Coin.java,          CoinAccount.java,      CoinApp.java,  
CoinException.java
```

Le classi contenute nel package coin rappresentano il Core dell'applicazione. Qui è stato implementato il protocollo.

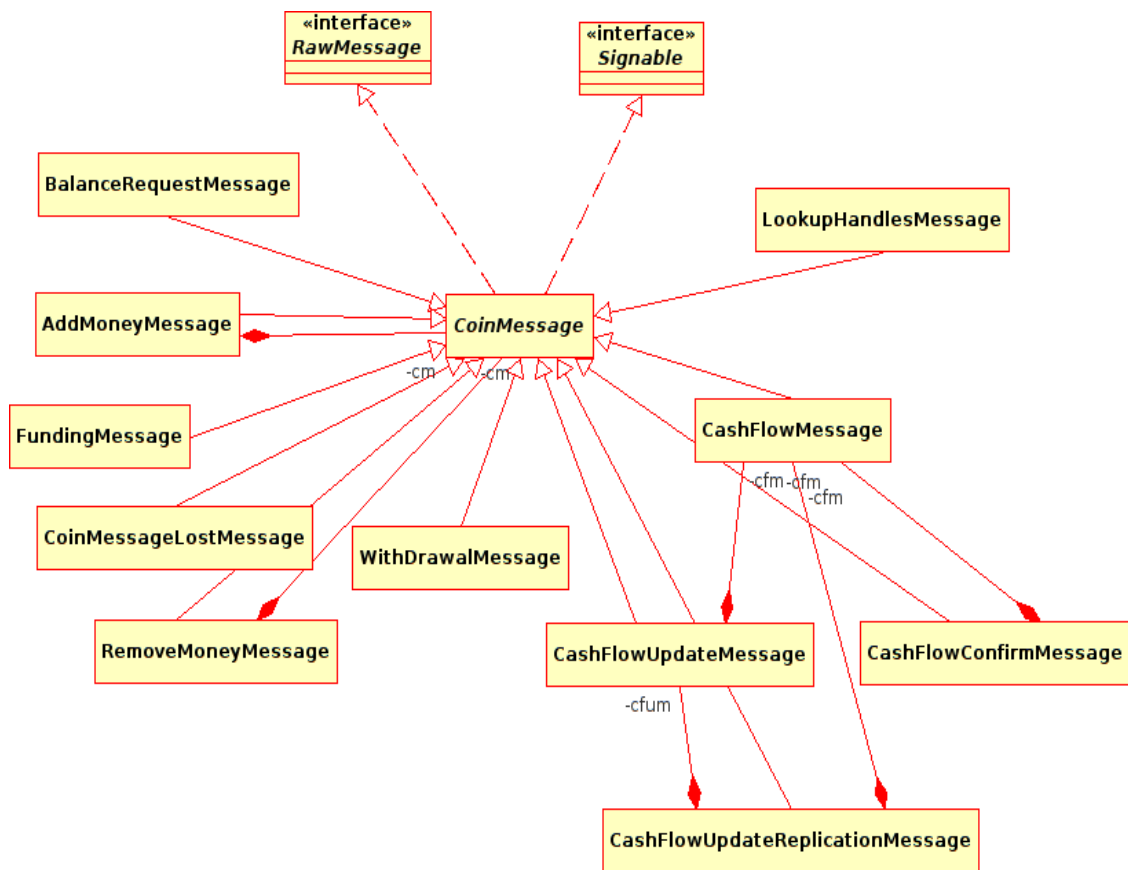


```

unito.p2p.coin.messaging
  AddMoneyMessage.java,   CashFlowUpdateMessage.java,
  LookupHandlesMessage.java, BalanceRequestMessage.java,
  CashFlowUpdateReplicationMessage.java,
RemoveMoneyMessage.java,
  BalanceRequestPropagatedMessage.java,
CoinMessage.java,
  WithdrawalMessage.java,   CashFlowConfirmMessage.java,
  CoinMessageLostMessage.java,   CashFlowMessage.java,
  FundingMessage.java

```

Queste classi implementano le varie tipologie di messaggi scambiati dai vari peer durante tutte le interazioni del protocollo.



```

unito.p2p.coin.security
  Signable.java

```

Nel package security sono implementate le logiche di sicurezza dell'applicazione.

```

unito.p2p.coin.testing
  DistCoin.java, DistInteractiveCoin.java

```

Infine nel package testing sono contenute le classi utilizzabili per il testing dell'applicazione.

API

Le applicazioni che necessitano dei servizi implementati da COIN possono utilizzare una serie di api esportate dall'interfaccia `unito.p2p.coin.Coin.java` ed implementate all'interno della classe `unito.p2p.coin.CoinApp.java`.

Riportiamo ora il codice dell'interfaccia delle API utilizzabile dalle applicazioni:

```

/**
 * @(#) Coin.java This interface is exported by all instances of Coin.
 * An instance of Coin provides a distributed accounting service.
 *
 * @author Fabio Varesano
 *
 */
public interface Coin {

    /**
     * Send amount of money to the given NodeHandle and return the
     * result to the given continuation. The result of this method
     * will be an acknowledgement of the completed result.
     *
     * @param nh the node wich will receive endpoint money
     * @param amount the amount of money to send
     * @param command Command executed when the result is received
     */
    public void doCashFlow(NodeHandle nh, int amount, Continuation
command);

    /**
     * Return the balance of the given NodeHandle to the Continuation
     *
     * @param nh the NodeHandle to get its balance
     * @param command the continuation which will receive the result
     */
    public void balanceRequest(NodeHandle nh, Continuation command);

    /**
     * Add money to a node account.
     * This operation is controlled by a central authority which
     * converts the amount of money into system virtual money.
     * The given Fundingessage will be signed by the central authority.
     *
     * @param fm the authority signed FundingMessage
     * @param command the continuation which will receive the result
     */
    public void funding(FundingMessage fm, Continuation command);

    /**
     * Used to trasform back system virtual money into real money.
     * A withdrawal message is sent to the account root then routed to
     * all the account holders which update the system account.
     * This will return back to the node who posses the account a list
     * of signed acknowledgment that he will then submit to the central
     * authority which then will add amount to the real account of the
     * node.
     *
     * @param amount the amount of money to convert back
     * @param command the Continuation which will recieve the result
     */
    public void withdrawal(int amount, Continuation command);
}

```

5.4 Protocollo e Implementazione

In alcune interazioni del protocollo presentato nei capitoli precedenti è assolutamente richiesta affidabilità nella consegna dei messaggi tra le varie entità coinvolte. Un esempio di interazione di questo tipo è la fase uno del Cash Flow dove viene negoziata l'operazione di trasferimento di moneta virtuale.

Il metodo `route()` esportato dall'interfaccia `rice.p2p.commonapi.Endpoint.java` attraverso il quale vengono inviati messaggi non garantisce di per se affidabilità. L'effettiva consegna di messaggi è infatti best-effort come avviene spesso in molti protocolli di rete.

Era necessario implementare una soluzione a questo problema al fine di rendere davvero utile l'applicazione sviluppata.

Al fine di garantire affidabilità una soluzione possibile sarebbe potuta essere implementare un complesso sistema di riscontri e ritrasmissioni, basato sul metodo di consegna dei messaggi Pastry. Questa soluzione, oltre che estremamente complessa dal punto di vista implementativo, avrebbe però appesantito l'overlay network in quanto i messaggi inviati sarebbero stati consegnati non in modo diretto bensì con la procedura di routing tipica di Pastry, con un overhead decisamente inaccettabile.

Altra soluzione possibile poteva essere implementare la consegna affidabile tramite l'utilizzo di socket creati ad hoc dall'applicazione. In questo modo l'affidabilità sarebbe stata relegata al protocollo TCP sul quale il socket avrebbe scambiato i dati.

Questa soluzione però, sebbene efficace, avrebbe complicato non poco il codice dell'applicazione. Al fine di rendere il codice scalabile sarebbe stato infatti necessario implementare un complesso meccanismo di Thread, sicuramente di difficile gestione. Inoltre sarebbero stati persi alcuni vantaggi del substrato Pastry, come ad esempio il richiedere solamente una porta TCP aperta nei firewall, la capacità di passare tramite NAT e la facilità di gestione di chiamate asincrone tramite Continuation.

Pastry 2

A questo punto dello sviluppo, dopo un'attenta valutazione del problema, è stato deciso il passaggio da FreePastry 1.4.4 a FreePastry 2 beta.

FreePastry 2 rende disponibili agli sviluppatori alcuni strumenti aggiuntivi decisamente utili.

In particolare ha avuto particolare rilievo la presenza nella versione 2 degli application socket (appsocket), una implementazione a livello applicativo del concetto di socket presente nei moderni sistemi operativi e linguaggi di programmazione.

Gli application Socket sono stati fondamentali nella risoluzione del problema di affidabilità descritto nel paragrafo precedente. È stato infatti possibile implementare una consegna di messaggi affidabile senza complicare eccessivamente il codice dell'applicazione.

Gli application Socket presentano, infatti, molti vantaggi:

- è possibile avere controllo completo su come i messaggi vengono serializzati
- si possono controllare meglio le risorse, in particolare si può implementare un controllo di flusso end to end
- si riesce ad utilizzare completamente la banda del peer senza interferire sul traffico di gestione del substrato FreePastry, cosa che una implementazione coi socket tradizionali non avrebbe garantito
- non è necessario aprire altre porte nei Firewall
- è possibile usare la soluzione anche in situazioni di connettività limitata, come ad esempio peer dietro a NAT
- si semplifica il codice tramite l'utilizzo di Continuation
- è possibile utilizzare il FreePastry Simulator, una utility per il testing di applicazioni P2P su larga scala.

Per ulteriori informazioni è possibile leggere la parte di tutorial specifica per gli application socket:

http://freepastry.org/FreePastry/tutorial/preview/tut_app_sockets.html

La classe `unito.p2p.coin.CoinApp.java` fa quindi un ampio uso degli application socket, in particolare il metodo `sendViaSocket()` realizza la consegna affidabile di messaggi tramite application socket.

Il passaggio a FreePastry 2 ha però provocato non pochi problemi di stabilità. Infatti al momento della implementazione FreePastry 2 era ancora in stato di beta e durante i test sono stati portati alla luce diversi bug. I problemi riscontrati sono stati tutti riportati al team di sviluppo che in modo molto professionale ha corretto prontamente tutti i bug segnalati.

Protocollo e Messaggi

Comprendere le logiche di protocollo implementate nella classe `unito.p2p.coin.CoinApp.java` è un compito non facile.

Al fine di rendere comprensibile un'analisi del codice in termini di messaggi di protocollo scambiati, vengono ora riproposti i grafici del protocollo in termini di Classi di messaggi.

Convenzione:

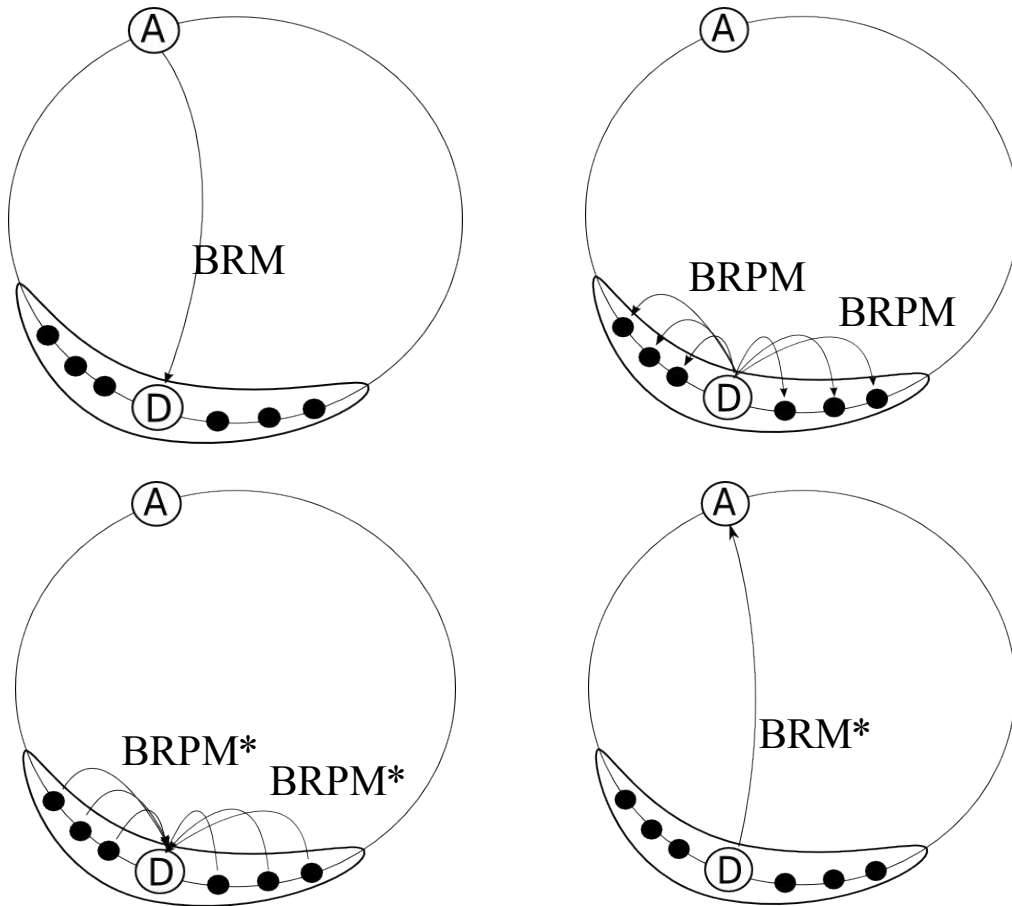
AMM = AddMoneyMessage.java,
CFUM = CashFlowUpdateMessage.java,
BRM = BalanceRequestMessage.java,
CFURM = CashFlowUpdateReplicationMessage.java,
RMM = RemoveMoneyMessage.java,
BRPM = BalanceRequestPropagatedMessage.java,
CM = CoinMessage.java,
WDM = WithdrawalMessage.java,

CFCM = CashFlowConfirmMessage.java,
 CMLM = CoinMessageLostMessage.java,
 CFM = CashFlowMessage.java,
 FM = FundingMessage.java

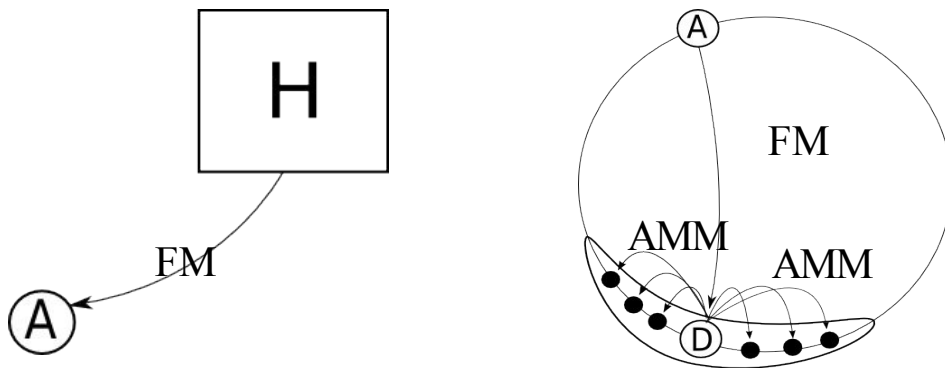
I messaggi contrassegnati con l'asterisco vengono intesi aventi il flag *response* a true.

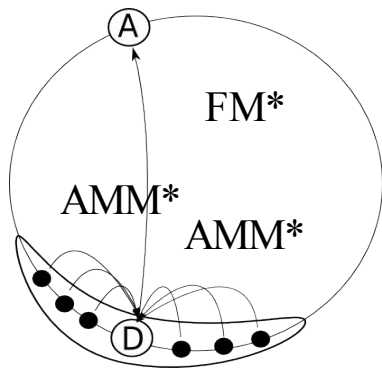
es: CFM* = messaggio di CashFlowMessage con *response* a true.

Balance Request:

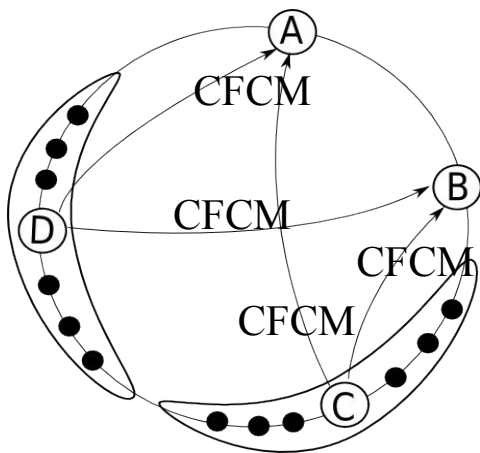
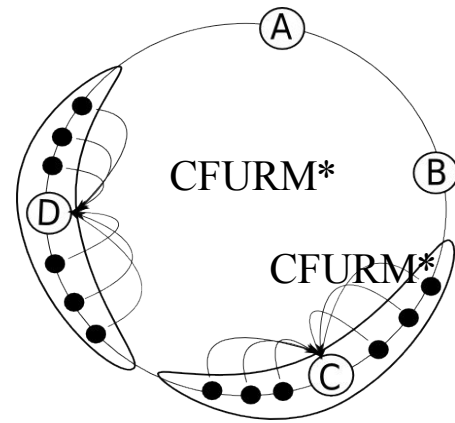
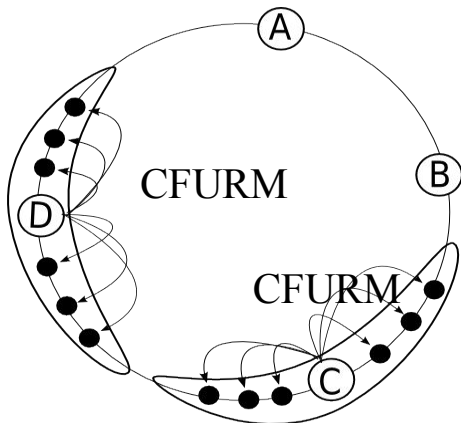
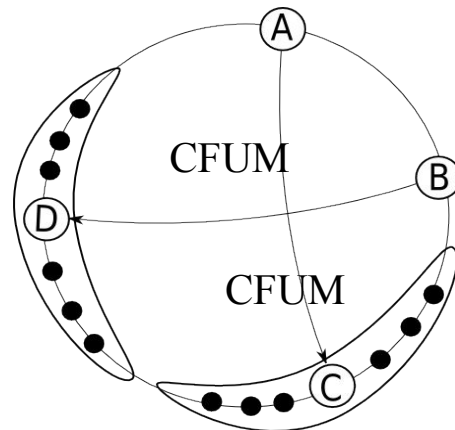
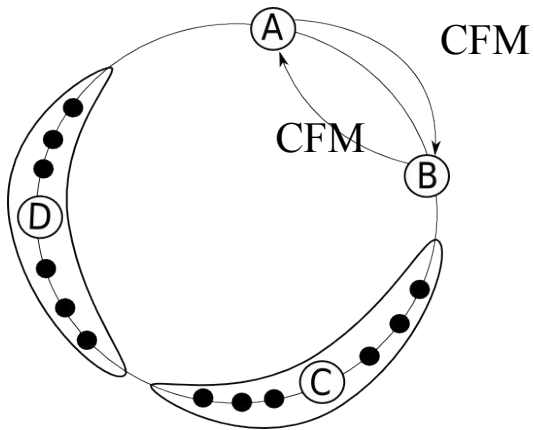


Funding:

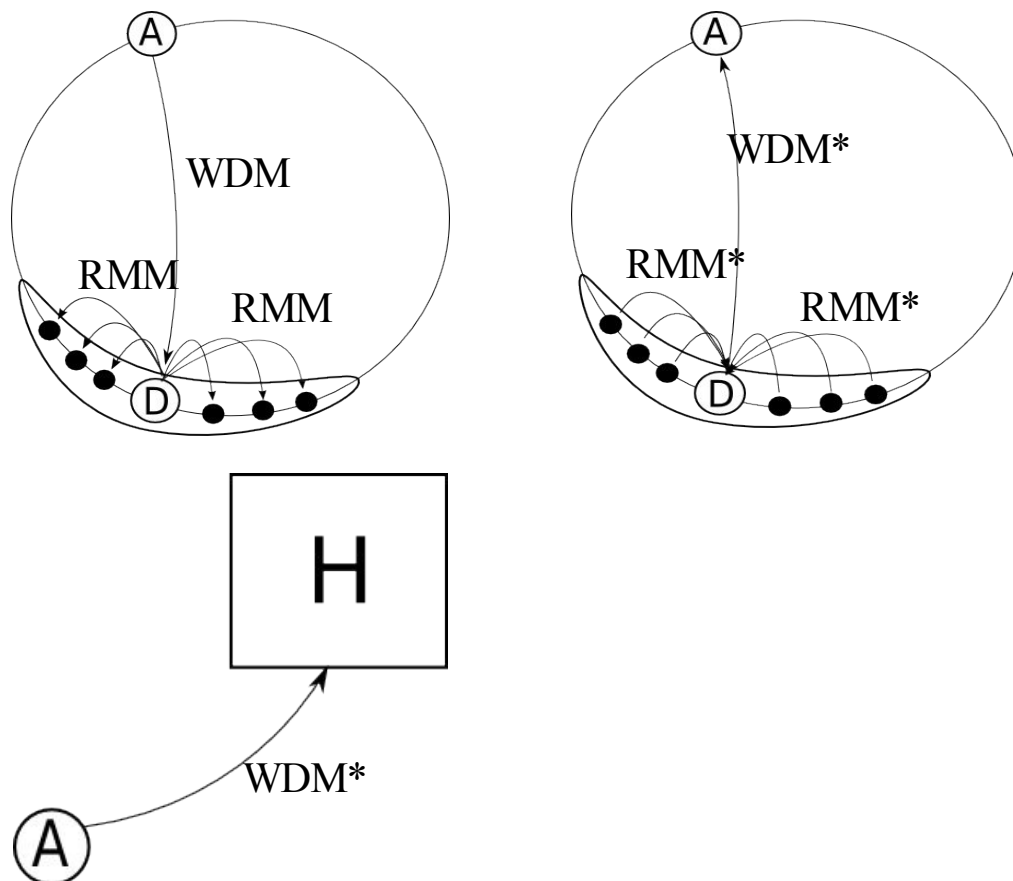




Cash Flow:



Withdrawal:



L'analisi del codice sorgente dell'applicazione dovrebbe risultare più semplice consultando i grafici proposti.

5.5 Test

La fase di testing si è compiuta durante tutto lo sviluppo dell'applicazione. Una delle prime problematiche affrontate è stata quella di creare ring di nodi sufficientemente grandi. Una soluzione è stata data tramite la classe `DistCoin.java` nel pacchetto `unito.p2p.coin.testing`.

Tramite questa classe è possibile creare ring Pastry/Coin ad una dimensione abbastanza elevata, circa una cinquantina di nodi, all'interno di un singolo PC. Automaticamente la classe esegue tutte le operazioni del protocollo, per facilitare il debugging.

Inoltre è stata sviluppata una classe per il testing interattivo del sistema. La classe `DistInteractiveCoin.java` presentando un menù all'utente permette di eseguire tutte le operazioni di COIN in modo interattivo.

Alla fine del progetto l'applicazione è stata anche testata su una piccola rete di PC all'interno del Dipartimento di Informatica, dimostrando la correttezza dell'implementazione anche in reti con maggiori quantità di nodi.

6. Miglioramenti e Sviluppi Futuri

Sebbene il lavoro fin qui descritto abbia richiesto, sia dal punto di vista progettuale che implementativo, una quantità di tempo molto elevata, non è stato possibile completare del tutto l'implementazione del protocollo.

Vengono qui dati una serie di spunti al miglioramento che potrebbero venire ripresi per dare origine a successive evoluzioni dell'applicazione.

6.1 Sicurezza

Il protocollo discusso nei capitoli precedenti ha nella sicurezza un componente fondamentale. L'implementazione, sebbene preveda già una interfaccia Signable, non si cura della gestione della sicurezza.

Sarebbe necessario implementare tutti quei meccanismi di firma digitale descritti nei capitoli precedenti. Bisognerebbe creare una logica di busta digitale per i messaggi trasferiti. A questo voleva servire l'interfaccia Signable.

Bisognerebbe creare una procedura di mapping da chiave pubblica a nodeId. La direzione da seguire è quella di creare una Node Factory basata, non su socket come quella attuale, bensì sulla chiave pubblica.

Bisognerebbe anche creare un efficace meccanismo di scambio delle chiavi tra i peer coinvolti nel sistema. Pensare semplicemente di scaricare le chiavi dei peer dall'autorità centrale non rispecchia le condizioni di scalabilità e tolleranza ai guasti che sono nostri obiettivi. Un'efficace soluzione per la memorizzazione delle chiavi pubbliche potrebbe essere l'utilizzo di PAST, l'applicazione di storage distribuito del package FreePastry.

È infine consigliabile tenere d'occhio le evoluzioni del package FreePastry, in quanto, a detta del team di sviluppo, le versioni successive alla 2 avranno nella sicurezza uno dei componenti fondamentali.

6.2 Persistenza

L'attuale implementazione non mantiene lo stato globale del sistema. All'uscita di un nodo ed alla conseguente modifica del leafset l'applicazione dovrebbe reagire per garantire persistenza. In particolare il nuovo nodo entrato nel leafset dovrebbe ricevere dai vicini gli account memorizzati fino ad ora come descritto nei capitoli precedenti.

L'attuale implementazione reagisce già alle modifiche del leafset. Il metodo update() nella classe CoinApp.java viene invocato ogni qualvolta il leafset subisce una modifica. Proprio in questo metodo bisognerebbe gestire l'aggiornamento del nuovo nodo nel leafset.

Per una implementazione più complessa potrebbe essere utile l'analisi delle classi ReplicationManager e ReplicationManagerClient entrambe utilizzate da PAST per garantire la replicazione dei contenuti.

6.3 Dimensione Leafset

Per semplicità si è impostata come dimensione del leafset 10. Questo valore ha semplificato notevolmente la programmazione in quanto minore del Neighborhood Set. In questo modo la scrittura del metodo `getHandles()`, utilizzato per ottenere i `nodeId` dei nodi facenti parti del leafset, è stata semplificata molto.

Un approccio più generale avrebbe dovuto prevedere la possibilità di impostare leafset più grandi della dimensione del Neighborhood Set, ma questo avrebbe provocato inevitabilmente un'aggiuntiva complessità dell'applicazione.

Per una soluzione a questo problema si può consultare, anche in questo caso, il codice di PAST, in particolare il metodo `getHandles()` della classe `PastImpl.java`

Sebbene dieci nodi possa sembrare ragionevole, in realtà la dimensione da dare al leafset è ancora oggetto di ricerca e sicuramente diversi test potrebbero aiutare nell'individuazione del valore corretto.

6.4 Formato dei messaggi

I messaggi scambiati tra i peer dell'applicazione sono stati implementati sotto forma di classi Java. Per poter essere inviati tramite la rete si è fatto uso della serializzazione disponibile in Java.

In questo modo non è possibile interagire con implementazioni scritte in altri linguaggi di programmazione.

Un approccio più generale sarebbe potuto essere quello utilizzato dalla versione 2 di FreePastry, cioè una sorta di serializzazione taggata dei dati trasmessi resa possibile dall'implementazione dell'interfaccia `RawMessage`. Con questa soluzione diventa possibile implementare metodi di codifica e di decodifica universali, riutilizzabili in diversi linguaggi di programmazione.

Per ulteriori informazioni è consigliabile l'analisi del package `rice.tutorial.rawserialization` disponibile in FreePastry 2.

Altri approcci ancora più generali potrebbero essere la codifica tramite XML o ASN.1/BER.

6.5 Affiancamento ad altro applicativo

Sarebbe interessante utilizzare COIN insieme ad un altro sistema basato su pastry. Una possibile applicazione da affiancare potrebbe essere il lavoro sviluppato da Luca Maria Aiello [3] nel suo tirocinio.

6.6 Testing

Infine sarebbe necessario sottoporre l'applicazione a diverse tipologie di test. Sicuramente interessante sarebbe il testing in una rete simile ad Internet come quella resa disponibile dal progetto PlanetLab.

Sarebbe interessante vedere come si comporta l'applicazione in presenza di peer maligni o fallibili per vedere come vengono gestite le situazioni di errore e

fallimento.

Inoltre sarebbe sicuramente utile proporre dei test di performance al fine di vedere quanto overhead COIN introduce se affiancato ad una applicazione già esistente.

Un ulteriore utile test potrebbe essere implementare il Simulator disponibile nella versione 2 di FreePastry. Che, a detta degli sviluppatori, permette il testing in locale di overlay network anche di dimensioni elevatissime in quanto simula la creazione di nodi, connessioni etc.

Per un approfondimento consiglio la lettura di

http://freepastry.org/FreePastry/tutorial/preview/tut_simulator.html

7. Conclusioni

L'attività svolta durante il mio tirocinio è stata entusiasmante. Mi sono scontrato con tecnologie davvero affascinanti.

Ideare, progettare ed implementare un protocollo applicativo di questo genere è stata una esperienza estremamente motivante e gratificante. Vedere per la prima volta due peer scambiarsi denaro tramite il sistema da me sviluppato è stato divertentissimo.

Ho potuto capire diverse problematiche della progettazione di protocolli di rete per applicazioni di sicurezza. Mi sono imbattuto in problemi complessi e concreti come la necessità di affidabilità, efficienza e scalabilità.

Ho capito molto delle logiche complesse nelle applicazioni distribuite, scontrandomi poi con l'implementazione delle stesse a livello pratico.

Per quanto riguarda il progetto sviluppato spero che il Dipartimento investa ulteriormente su ciò che è stato da me creato. Penso sia nel complesso un buon lavoro e vederlo continuato da qualcuno sarebbe davvero gratificante.

Infine consiglio di tenere d'occhio il progetto FreePastry. Durante lo sviluppo si è rivelato davvero completo ed efficace e sono sicuro che vedremo la nascita di diverse applicazioni basate su di esso.

7.1 Ringraziamenti

Ringrazio innanzitutto il Professor Giancarlo Ruffo, per essere stato la mia guida durante tutto il tirocinio. Ringrazio inoltre il Dottor Schifanella per il supporto nella fase di testing. Ringrazio infine il Dipartimento di Informatica per le infrastrutture messe a disposizione.

Mi preme infine ringraziare il team di sviluppo di FreePastry, in particolare Jeff Hoyer, per il supporto, la gentilezza e disponibilità dimostratami lungo tutta la fase di sviluppo dell'applicazione.

Ringrazio inoltre tutte le persone che mi sono state vicine durante questi anni di studio. La mia fantastica famiglia che mi ha dato un ambiente sereno e caloroso durante tutta la mia vita. I miei amici che mi hanno aiutato a “staccare” quando ero a pezzi dallo studio.

8. Bibliografia

1. David Hausheer "*PeerMart: Secure Decentralized Pricing and Accounting for Peer-to-Peer Systems*". Diss. ETH Zurich No. 16200, Shaker Verlag, ISBN 3-8322-4969-9, Aachen, Germany, March 2006.
2. Philipp Obreiter, Jens Nimis "*A Taxonomy of Incentive Patterns: The Design Space of Incentives for Cooperation*" (2003)
3. Luca Maria Aiello "*Fairpeers: progettazione ed implementazione di un servizio di file management tramite Pastry*". Università di Torino, Dipartimento di Informatica 2006
4. FreePastry Website <http://www.freepastry.org/>
5. FreePastry Tutorial 1.4 <http://freepastry.org/FreePastry/tutorial/>
6. A. Rowstron and P. Druschel, "*Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems*". IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, pages 329-350, November, 2001
7. FreePastry Tutorial 2 beta
<http://freepastry.org/FreePastry/tutorial/preview/> (probabilmente l'URL cambierà al rilascio di FreePastry 2 stabile)
8. P. Druschel and A. Rowstron, "*PAST: A large-scale, persistent peer-to-peer storage utility*", HotOS VIII, Schloss Elmau, Germany, May 2001.
9. A. Rowstron and P. Druschel, "*Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility*", ACM Symposium on Operating Systems Principles (SOSP'01), Banff, Canada, October 2001.

9. Curriculum Vitae

FABIO VAREANO



CURRICULUM VITAE

PERSONAL INFORMATIONS

Name	VAREANO, Fabio
Address	54, C.so Salvemini, ITALY – 10137 Torino (TO)
Mobile	(+ 39) 3331343352
Telephone	(+ 39) 0113099397
E-mail	fvaresano@yahoo.it
Nationality	Italian
Birthplace	Torino, 01.06.1984

EDUCATION AND TRAINING

- Dates 2003 – now
- Name and type of organisation providing education and training UNIVERSITY OF TORINO (TURIN)
Department of Computer Science www.di.unito.it
- Principal subjects/occupational skills covered Currently studying at the third year of the Computer Science Degree Course.
- Level in national or international classification Average 27/30

- Dates 1998 – 2003
- Name and type of organisation providing education and training SCIENTIFIC AND TECHNOLOGICAL HIGH SCHOOL
E. FERRARI, Torino (Turin) www.itisferrari.it
- Title of qualification awarded Graduation
- Level in national or international classification Mark 90/100

- Dates 2003
- Name and type of organisation providing education and training UNIVERSITY of CAMBRIDGE
(Course and Exams took place in Italy with teacher from the University of Cambridge)
- Title of qualification awarded Preliminary English Test (PET)
- Level in national or international classification Pass with Merit

<p>TECHNICAL SKILLS AND COMPETENCES Acquired during life and career</p>
--

PROGRAMMING LANGUAGES

- **(X)HTML, CSS, JavaScript**
Advanced knowledge. My goals are user centric websites focused on content clarity, easy to browse and W3 compatibility.
- **PHP**
4 years of experience. Deep knowledge of the better Open Source Content Management Systems available.
- **SQL**
Deep knowledge. Equally skilled in MySQL and PostgreSQL and proprietary DBMS, such as Oracle and MSSQL.
- **C**
Good knowledge. Developed multiprocess application under Unix/Linux.
- **JAVA**
Good knowledge. Excellent knowledge of Object Oriented Programming.
- **JSP**
Good knowledge. Developed web applications.

OPERATING SYSTEMS

- **Linux**
Advanced knowledge as both advanced user and administrator. Excellent knowledge of the main distributions. At the moment I'm using Slackware and ArchLinux on both workstations and servers.
- **OpenBSD**
Good knowledge. Used as testing server for some web applications.
- **Windows**
Advanced knowledge of all versions. Configured small size LANs.
- **Mac OS X**
Base knowledge and limited experience.

SYSTEM APPLICATIONS

- **Apache**
Advanced knowledge of the server. Configured for both testing and production environment. Excellent knowledge of configuration for multidomains settings, encrypted connection and scripting languages support.

- **Qmail - VPopMail**
Advanced knowledge. Successfully served 4 domains of users.
- **MySQL**
Good knowledge.
- **Samba**
Advanced knowledge. Configured small size LANs.
- **IpTables**
Good knowledge of the Linux firewall.

WORKING EXPERIENCES

- | | |
|--|---|
| <ul style="list-style-type: none"> • Period • Company Name • Type of Company • Type of work • Main tasks and activities | <p>Autumn 2006
UNIVERSITY OF TORINO (TURIN)
Department of Computer Science www.di.unito.it
University
Developer
Developed COIN, an accounting system for distributed P2P Networks</p> |
| <ul style="list-style-type: none"> • Period (from – to) • Company Name • Type of Company • Type of work • Main tasks and activities | <p>2004 – now
DRUPAL http://www.drupal.org
Open Source Content Management System
Developer
Maintainer and main developer of video module (video streaming from drupal)
Maintainer and main developer of css module (editing of cascading style sheets of contents)
Developer of some patches for other modules and core.</p> |

PERSONAL EXPERIENCES

- | | |
|--|--|
| <ul style="list-style-type: none"> • Period (from – to) • Company Name • Type of Company • Type of work • Main tasks and activities | <p>February – June 2006
Linux Course - 2006
http://i-teach.educ.di.unito.it/course/view.php?id=11
(login as "ospite")
UNIVERSITY OF TORINO (TURIN)
Department of Computer Science www.di.unito.it
Teacher
Organizing and teaching a course about Linux.</p> |
|--|--|

WEB SITES

- Websites developed
 - www.varesano.net
Fabio Varesano's Personal Website

- www.adrenalinteam.it
Adrenalin Team Freestyle Ski Team
- www.falcoarredamenti.it
Falco Furniture Store
- www.windsmpeg.tk
Windsurfing Video Site

<p>PERSONAL ABILITIES AND EXPERIENCES Acquired during life</p>

MOTHER TONGUE

ITALIAN

- Reading
- Writing
- Speaking

ENGLISH

Excellent
Very Good
Very Good

FRENCH

- Reading
- Writing
- Speaking

Good
Good
Good

DRIVER'S LICENSE

driver's license for cars

Last modified: 2006-12-07