UNIVERSITÀ DI PADOVA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA TRIENNALE IN INGEGNERIA INFORMATICA

PariCredits : Stallo ed Equa Distribuzione

Relatore: Ch.mo Prof. Enoch Peserico Stecchini Negri De Salvi

Correlatore: Paolo Bertasi

Laureando: *Mattia Ornamenti*

Anno Accademico: 2009-2010

Indice

Sommario

Questa tesi è rivolta a coloro che sono interessati a conoscere il nuovo sistema di gestione dei crediti interni nel modulo Crediti di PariPari, atto a risolvere il problema dello stallo delle richieste e a garantire un'equa distribuzione di risorse.
Nel Capitolo 1 verranno introdotti tutti quei concetti e strutture dati del progetto PariPari ed in particolare del modulo Crediti, necessari alla comprensione delle nuove funzionalità.
Nel Capitolo 2 verrà analizzato il problema dello stallo dei crediti e la sua soluzione mediante grafo tra plugin.
Nel Capitolo 3 saranno discusse tutte le tecniche introdotte che sfruttano il grafo per raggiungere un'equa distribuzione di crediti.
Nel Capitolo 4 saranno precisate alcune considerazioni di carattere implementativo sul progetto.
Nel Capitolo 5 si introdurranno gli sviluppi futuri a breve termine del progetto.

Capitolo 1

Introduzione

1.1 – Il progetto PariPari[1]

PariPari è un progetto, attivo presso il Dipartimento di Ingegneria dell'Informazione dell'Università degli Studi di Padova, con lo scopo di realizzare una rete P2P multifunzionale.
PariPari si basa, appunto, sul paradigma di rete *Peer-To-Peer* (P2P) in base al quale non esistono nodi della rete specializzati come Server (Serventi - fornitori di servizi) e Client (Clienti - utilizzatori di servizi), ma tutti i nodi fungono sia da cliente che da servente verso altri nodi della rete.
La *multifunzionalità* è un fattore caratterizzante del progetto. A partire dall'avvento della rete Internet si sono diffuse un numero incredibile di applicazioni che usano la rete per fornire un qualche servizio all'utente; chat, browser, posta elettronica, condivisione di file solo per citarne alcuni. Lo scopo finale del progetto PariPari è proprio quello di riunire tutti questi servizi in un'unica piattaforma accessibile da tutti, compresi quei servizi che sono sempre stati forniti, storicamente, solo mediante paradigma Client-Server. Ecco il motivo per cui si è scelto Java come linguaggio di programmazione; la possibilità, fornita da Java, di scrivere programmi specifici per la sola macchina virtuale JVM permette di astrarre le diverse architetture hardware e software focalizzando l'attenzione solo sui contenuti.

1.2 – Architettura a plugin

Per un progetto di tali dimensioni è fondamentale il concetto di modularità, ossia la possibilità di definire e sviluppare delle unità funzionali pressoché indipendentemente dalla restante parte del progetto. Questo concetto è radicato alla base dell'architettura di PariPari grazie all'imposizione di diversi livelli di astrazione nelle comunicazioni tra

[1] Per maggiori informazioni sulle caratteristiche e sulle peculiarità dell'intero progetto fare riferimento a [1].

unità. Le astrazioni di interesse per la nostra analisi saranno esposte nel paragrafo *1.4 – API e servizi*, insieme alle possibilità che una tale astrazione riesce a fornire.

Le unità funzionali di cui è composto PariPari vengono denominate *Plugin*. La figura 1.1 mostra una rappresentazione figurativa ma molto precisa dell'architettura del sistema[2].

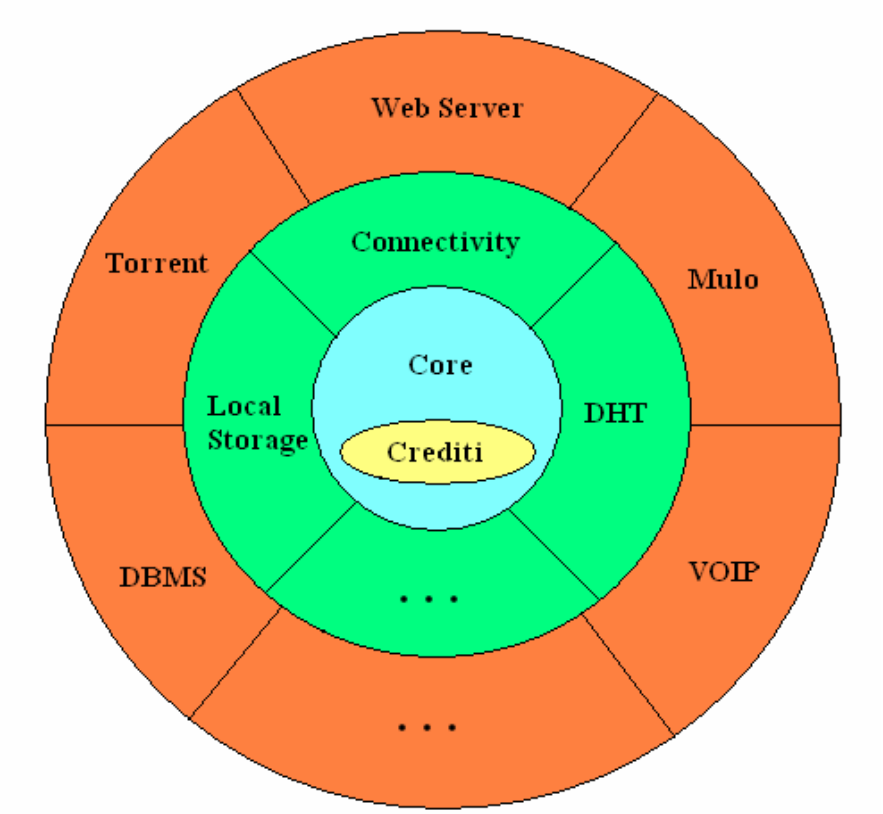

Figura 1.1 : Rappresentazione dell'architettura a plugin del sistema.

Al centro di tutto vi è il *Core*, di cui il modulo Crediti fa parte. Esso si occupa di gestire il caricamento e la disinstallazione dei plugin, l'utilizzo dell'unità di elaborazione ed è, inoltre, l'intermediario di qualsiasi comunicazione[3].
L'anello che circonda il Core viene denominato *cerchia interna*. I plugin che ne fanno parte (Connectivity, Local Storage, DHT, …[4]) forniscono principalmente servizi utili ad altri plugin e hanno quindi bassissima interazione diretta con l'utente. Il loro scopo è amministrare efficientemente le risorse della macchina locale; da notare come, sotto questo aspetto, anche il Core possa essere considerato facente parte della cerchia interna.

[2] Per una più dettagliata analisi delle funzionalità dei principali plugin fare riferimento a [1].
[3] Quest'ultima caratteristica verrà precisata nel paragrafo *1.4 – API e servizi*.
[4] Attualmente i tre elencati sono gli unici plugin della cerchia interna. Nulla vieta che in futuro ne possano essere sviluppati degli altri.

L'anello più esterno è denominato *cerchia esterna*. Ne fanno parte tutti quei plugin che forniscono uno o più servizi direttamente utili all'utente. Questi plugin sono quelli che tendono maggiormente ad utilizzare i servizi offerti dalla cerchia interna. La lista può essere davvero lunga, in quanto i servizi offerti all'utente tramite la rete Internet sono in continuo aumento e per ciascuno di essi è possibile realizzare un plugin in PariPari.

1.3 – Economia in PariPari

L'economia[5] in PariPari è un fattore estremamente importante, al punto che è stato dedicato un team di sviluppo apposito: il modulo Crediti. Lo scopo di questo modulo, di fatto parte integrante del Core, è quello di assistere il Core nelle decisioni intraprese nelle due grandi macro-economie presenti nel progetto e denominate crediti interni e crediti esterni.

- I *crediti interni* supervisionano l'utilizzo delle risorse locali, contese fra i plugin caricati nella macchina.
- I *crediti esterni* supervisionano lo scambio di risorse tra i nodi della rete.

Proponiamo ora una breve panoramica sui principali sistemi economici che è possibile utilizzare per reti P2P, alcuni dei quali sono effettivamente implementati nelle più note reti di File Sharing.[6]

- **Sistemi basati su reputazione**:
  - Livello di *partecipazione propria* (usato da Kazaa/FastTrack): ogni nodo calcola il proprio livello di partecipazione nella rete e lo comunica agli altri. È un sistema estremamente facile da eludere poiché nulla vieta di pubblicare un livello di partecipazione falso.
  - Livello di *partecipazione altrui* (usato da eMule/eDonkey): ciascun nodo calcola il livello di partecipazione dei nodi con cui ha avuto degli scambi. È un sistema non transitivo (caratteristica estremamente importante per economie di questo tipo). È facile imbrogliare ma non si ottiene nessun vantaggio nel farlo.
  - Algoritmo *EigenTrust* (cfr. [3]): ciascun nodo può calcolare il livello di partecipazione di altri nodi, anche sconosciuti, pesando i valori di fiducia forniti dai nodi conosciuti. È un sistema difficile da ingannare, ma tende a fornire più risorse a comunità grandi, non incentivando le piccole comunità.

- **Sistemi basati sul commercio di moneta**:
  - Usando una moneta virtuale, immagazzinabile e durevole, si può evitare il calcolo di livelli di partecipazione/fiducia. La moneta può essere

[5] Economia ha qui il significato di "aspetto economico" di un sistema. Non deve essere quindi inteso come sinonimo di "risparmio".
[6] Per un'analisi dettagliata dei sistemi economici per reti P2P fare riferimento a [2].

esogena al sistema (Euro, Dollaro, …) o endogena al sistema (valuta virtuale usata solo all'interno della rete).

- **Sistemi basati sul baratto**:
  - Lo scambio di un bene per un altro bene porta ad un numero considerevole di problematiche quali l'asimmetria di interesse[7], la divisibilità delle risorse scambiate e altre ancora. Questo sistema risulta comunque molto valido una volta risolti i suddetti problemi.

Sebbene auspicabile poter usare uno stesso sistema economico sia per la gestione dei crediti interni sia per quelli esterni, questo non sembra possibile a causa di alcune differenze sostanziali delle due economie. I principali fattori sono:

- *Risorse costanti*. L'aumento di nodi nella rete fa aumentare anche le risorse disponibili. L'aumento di plugin caricati nella macchina non aumenta il numero di risorse che la macchina può offrire, anzi, ogni plugin è costretto a lavorare con meno risorse.
- *Numero di nodi*. Il numero di plugin caricabili in una macchina non è confrontabile con il numero di nodi delle reti P2P di maggiore successo.
- *Specializzazione.* I nodi della rete possono essere fornitori di svariati servizi, eventualmente in tempi diversi. Un plugin, invece, fornisce sempre gli stessi servizi.
- *Controllo centralizzato.* Nella macchina locale è presente un'unità centrale di controllo: il Core. In una rete esclusivamente P2P come PariPari questo non è possibile (andrebbe contro il concetto stesso di P2P).

Da queste considerazioni è stato deciso di implementare un sistema economico basato su moneta virtuale, chiamata appunto "***crediti***", per i crediti interni. Un sistema basato sul baratto di promesse per beni futuri incentivando i nodi a mantenere le promesse tramite un surplus[8], è stato invece scelto per i crediti esterni.
Ai fini della comprensione degli argomenti trattati successivamente, saranno utili solo le caratteristiche dei crediti interni.

1.4 – API e servizi

Benché i plugin possano essere sviluppati in maniera indipendente gli uni dagli altri, la stessa cosa non si può dire della loro esecuzione a run-time. La maggior parte, per funzionare correttamente, ha bisogno di servizi e funzionalità che non si implementa in quanto non direttamente collegate allo scopo per cui il plugin è stato sviluppato. Chiarifichiamo con un esempio: se si sta sviluppando un plugin che implementa il

[7] Asimmetria di interesse. Se A vuole qualcosa da B non è detto che B, in quel preciso momento, sia interessato a qualcosa da A.
[8] Tecnica tratta da [4].

protocollo eDonkey[9] (in PariPari esso esiste e si chiama Mulo) per fornire all'utente un client per la rete in questione, non è desiderabile doversi sobbarcare il compito di gestire anche l'uso di socket[10] per la comunicazione con la rete Internet, i calcoli per la distribuzione di informazioni tramite DHT[11], etc. Queste funzionalità devono essere offerte dai plugin della cerchia interna che hanno proprio lo scopo di fornire servizi utili ad altri plugin gestendo nel migliore dei modi le risorse a disposizione della macchina in cui operano.

Poiché i servizi offerti sono molto eterogenei tra di loro, è stato sviluppato un sistema per rendere uniforme e sicura la modalità con cui un plugin interagisce con gli altri; questo si traduce in un vantaggio per lo sviluppatore che può richiedere/offrire un qualsiasi servizio con la stessa struttura di chiamate.[12]

- La *sicurezza* può essere garantita solo dal Core, per cui in PariPari non esiste alcuna comunicazione diretta tra plugin; ogni comunicazione (compravendita di un servizio) deve passare per il Core. Questo si traduce nel seguente schema logico, che verrà precisato ulteriormente nel paragrafo *1.6 – Transazioni e rinnovi: funzionamento di uno scambio*. La richiesta di un servizio deve essere mandata al Core, il quale provvederà a inoltrare la richiesta al plugin destinatario; l'offerente risponderà (sempre al Core) con il servizio richiesto, se disponibile.
- L'*uniformità* negli scambi è resa possibile grazie all'uso di interfacce; poiché i servizi sono rappresentati da oggetti, l'unico modo per renderli uniformi è imporre alcune interfacce/classi astratte da implementare. Le interfacce che rappresentano i servizi scambiabili tra plugin, vengono chiamate API[13].

Da notare come un sistema di questo tipo permetta l'implementazione multipla di una stessa API; infatti, è possibile in PariPari avere disponibile più di un plugin che fornisce lo stesso servizio semplicemente implementando la stessa interfaccia. È compito del Core, supportato dal modulo Crediti, gestire correttamente le implementazioni multiple fornendo ai plugin richiedenti il servizio migliore tra quelli disponibili.

Analizziamo più in dettaglio le caratteristiche di un servizio offerto. Il venditore necessita che vengano specificati uno o più parametri per poter offrire un servizio corretto: alcuni parametri identificheranno la tipologia del servizio, altri la qualità del servizio. Di questi parametri, però, solo un sottoinsieme è rilevante ai fini delle decisioni intraprese dal modulo Crediti.
Un esempio può chiarificare; se il servizio in questione è un socket per la comunicazione verso un processo residente in un'altra macchina della rete, sarà necessario specificare parametri identificatori come:

---

[9] Maggiori informazioni sul protocollo/rete/client in questione si possono trovare su: http://en.wikipedia.org/wiki/EDonkey_network.
[10] Socket. Astrazione software del punto di accesso di un canale di comunicazione verso un processo operante in una macchina distinta.
[11] DHT. Distributed Hash Table. Tabella di Hash Distribuita.
[12] Il sistema che permette di uniformare le richieste di servizi è stato denominato *T.A.L.P.A. framework*.
[13] API. Application Programming Interface. Interfaccia di Programmazione di un'Applicazione.

- Indirizzo IP di destinazione.
- Porta di destinazione.

Mentre parametri di qualità possono essere:

- Banda minima richiesta.
- Latenza massima.
- Livello di jitter.

È immediato notare come l'indirizzo IP e la porta non siano assolutamente rilevanti ai fini del calcolo del prezzo del servizio.
Viene quindi definita "*feature*" una caratteristica di un servizio rilevante al modulo Crediti ai fini del calcolo del prezzo.[14]

1.5 – Pubblicazione di servizi: Descriptor e GoodsTable

Analizziamo ora il meccanismo con cui un plugin può pubblicare le informazioni riguardanti i servizi offerti e i servizi richiesti. In questo contesto "pubblicare" ha il significato di fornire al Core e al modulo Crediti le informazioni necessarie per permettere un corretto utilizzo delle risorse.[15]

Poiché è necessario conoscere le caratteristiche dei servizi disponibili prima che essi vengano richiesti, le informazioni devono essere pubblicate dai plugin al momento del caricamento; per fare ciò è sufficiente predisporre, nell'archivio jar con cui viene distribuito il plugin, un file chiamato "***descriptor.xml***".
Come il nome suggerisce, il file è scritto in linguaggio XML[16] e viene analizzato da un parser XML[17] al momento del caricamento del plugin. La struttura del file deve seguire uno standard ben preciso che viene sempre controllato, bloccando il processo di caricamento nel caso ci siano errori nella sintassi e nella struttura. Questa verifica iniziale è obbligatoria per avere delle informazioni sintatticamente corrette e quindi leggibili dal parser.
Non è però possibile evitare, tramite una semplice validazione XML, la pubblicazione di informazioni false; dichiarare di implementare un servizio che invece non si implementa, dichiarare la possibilità di fornire una certa qualità al servizio che poi invece non si riesce a garantire. Per falsificazioni di questo tipo saranno gli stessi

[14] Poiché i dettagli sulle proprietà, sulle caratteristiche e sull'implementazione del concetto di *feature* non sono rilevanti per questa tesi, si è deciso di non approfondire questa sezione.
Un'analisi dettagliata può essere trovata in [5].
[15] Ricordiamo che non esiste mai comunicazione diretta tra plugin. Le informazioni in questione vengono fornire al Core, il quale le ridirige al modulo Crediti.
[16] XML. Extensible Markup Language. È un linguaggio che permette di definire un meccanismo sintattico per estendere o controllare il significato di altri linguaggi.
[17] Il parser XML utilizzato al momento è JAXP (https://jaxp.dev.java.net/), usato nella sua funzionalità di interfaccia DOM (http://en.wikipedia.org/wiki/Document_Object_Model).

plugin, utilizzatori del servizio, a comunicarlo al modulo Crediti tramite il meccanismo del *feedback*. Per un plugin che riceve un servizio non funzionante o non soddisfacente le caratteristiche pattuite, è possibile informare il modulo Crediti fornendo un feedback negativo sull'API ottenuta. Le giuste azioni posso quindi essere poste in essere, come impedire l'afflusso di crediti verso il plugin malevolo.

Facciamo ora una breve panoramica sulla struttura e sulle principali informazioni che si devono predisporre nel file *descriptor.xml* e che saranno utili per comprendere le novità introdotte nel progetto.[18]

```
<plugin>

    <dependencies>
        // Qui va predisposto l'elenco delle API da cui il
        // nostro plugin dipende. Vengono chiamate anche
        // dipendenze.
    </dependencies>

    <interfaces>
       // Qui va predisposto l'elenco dei servizi che si
       // implementa e, per ciascuno di essi, le
       // caratteristiche delle rispettive feature.
    </interfaces>

</plugin>
```

Figura 1.2 : Schema scheletro del file descriptor.xml.

Per ciascun servizio offerto deve essere indicato il tempo medio standard di utilizzazione da parte dei compratori (denominato *timeunit*). Per ciascuna *feature* possiamo trovare:

- Minimo e massimo valore comprabile.
- Quantità disponibile.
- Legge di calcolo del prezzo.

Gran parte di queste caratteristiche sono comunque modificabili a run-time da parte del plugin venditore. Le ragioni per questa possibilità sono principalmente due:

- Riflettere maggiormente la disponibilità di risorse della macchina. La quantità e la qualità dei servizi offerti dipendono strettamente dalla disponibilità di risorse offerta dalla macchina ospitante.
- Non sempre possono essere note, al momento del caricamento, le caratteristiche dei servizi che si possono offrire. Ad esempio, se si implementa il servizio di

[18] Un'analisi dettagliata sulla struttura del file XML e le motivazioni che hanno portato a decidere per una sintassi di questo tipo, si possono trovare in [5].

socket è impossibile sapere a priori quanta banda offre la connessione Internet della macchina ospite.

La possibilità di modifica a run-time è concessa per le caratteristiche delle singole feature. Il tipo di servizi offerti e richiesti, resta invece immodificabile.

Una volta caricate le informazioni di un nuovo plugin esse vengono mantenute in una struttura chiamata "***GoodsTable***". Questa non è altro che una struttura dati complessa, disponibile, per la consultazione, a tutti i plugin caricati.
La struttura è tabellare. Ad ogni riga corrisponde un certo servizio offerto da un certo plugin; ricordiamo che uno stesso servizio può essere implementato da più plugin contemporaneamente, quindi solo la coppia (API – plugin) è univoca in ogni istante. Ogni elemento di una riga rappresenta una feature del servizio considerato con tutte le proprie caratteristiche.
Tramite una tecnica di validazione delle chiamate ai metodi della *GoodsTable*, le informazioni sui servizi offerti sono disponibili per la consultazione da qualunque plugin. Le modifiche, invece, sono possibili solo per il plugin a cui la riga si riferisce.[19]

1.6 – Transazioni e rinnovi: funzionamento di uno scambio

Analizziamo ora nel dettaglio la compravendita di una generica risorsa.
Il procedimento è stato suddiviso in tre punti, poiché sono tre le fasi di interazione con il modulo Crediti per garantirsi l'utilizzo di un servizio: ricerca, richiesta, scadenza. Ciascuna fase è distaccata dalle altre nel tempo da intervalli non noti a priori, dipendenti dal servizio offerto.

Una rappresentazione grafica dell'interazione compratore/Core/venditore è descritta in figura 1.3.

- **Ricerca**. Il plugin che necessita un servizio, consulta la GoodsTable per sapere se esso è disponibile con le caratteristiche volute; di fondamentale importanza è il tempo per il quale lo si vuole utilizzare. Il modulo Crediti verifica la disponibilità e ne calcola il prezzo per il compratore, proponendogli tramite un oggetto di tipo `Agreement` l'offerta migliore.

- **Richiesta**. Se il plugin compratore accetta l'offerta, notifica al Core la volontà di comprare il servizio. Mentre il Core contatta il venditore per farsi dare il servizio richiesto, il modulo Crediti tiene traccia della compravendita tramite un oggetto di tipo `Transaction`. Una transazione rappresenta, per il modulo Crediti, un servizio in fase di utilizzo e viene eliminata alla scadenza del servizio stesso. Al momento dell'acquisto, il prezzo concordato per l'API viene detratto dal conto del plugin compratore[20] ma non viene immediatamente

[19] Maggiori informazioni sulla struttura della *GoodsTable* si possono trovare in [2].
[20] Il conto di un plugin è rappresentato da un oggetto di tipo `PluginAccount`. L'insieme dei conti è mantenuto in una struttura chiamata `Bank`.

depositato nel conto del venditore per evitare di fornire crediti a quei plugin malevoli che non forniscono correttamente i servizi dichiarati. Possiamo quindi correttamente immaginare che i crediti detratti dal compratore vengano, temporaneamente, depositati nella transazione stessa, fino alla sua scadenza.

- **Scadenza**. Poco prima della scadenza del tempo concordato di utilizzo, per il compratore sono possibili due scelte:

  - *Lasciar scadere il servizio*. Nel caso il plugin abbia terminato l'utilizzo dell'API non deve compiere nessuna azione particolare. La transazione scade e, in caso di feedback positivo, i crediti concordati vengono depositati nel conto del venditore.
  - *Rinnovare l'uso del servizio*. Se il plugin necessita l'uso del servizio più del tempo prestabilito, può effettuare un rinnovo (*Renew*). Un rinnovo deposita nel conto del venditore i crediti del servizio appena conclusosi, ed estende la possibilità di sfruttare il servizio stesso al compratore. Il prezzo viene ricalcolato secondo le condizioni attuali e viene detratto dal compratore.

    Benché i rinnovi possano sembrare, a prima vista, solo una complicazione nell'interazione tra plugin, essi sono fondamentali per evitare inutili sprechi di tempo. Consideriamo, ad esempio, la compravendita di un socket; spesso non è possibile sapere a priori per quanto si vuole instaurare un canale di comunicazione verso un altro nodo della rete. In queste condizioni raggiungere la scadenza senza aver completato la trasmissione di informazioni è assai frequente; è impensabile, a questo punto, costringere il plugin che gestisce i socket[21] a chiudere il canale per poi riaprirlo poco dopo. Ecco quindi che rinnovare l'uso del servizio risulta essere la tecnica migliore.

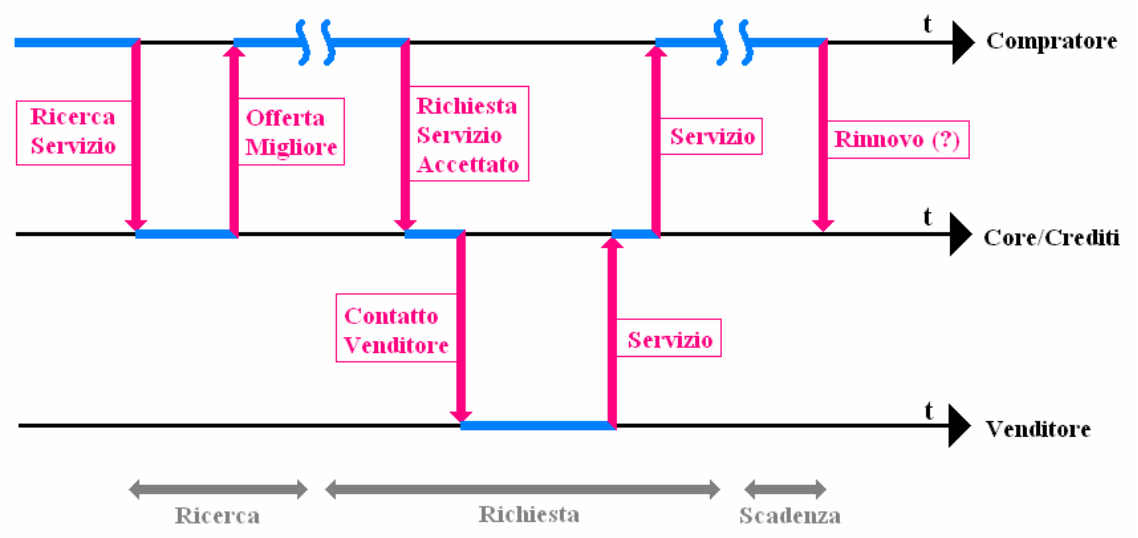

Figura 1.3 : Visualizzazione dell'interazione compratore/Core/venditore.

[21] Connectivity nel nostro caso.

Capitolo 2

Stallo del sistema e grafo tra plugin

2.1 – Il problema

Ora che è stato analizzato nel dettaglio il meccanismo e le strutture dati tramite le quali vengono gestite le compravendite di risorse tra plugin, una domanda sorge spontanea: il sistema, allo stato attuale, permette di gestire correttamente ed efficacemente le risorse del sistema? In altre parole, eseguendo PariPari con il modulo Crediti attivo[22] e utilizzandolo come un normale utente farebbe (caricare e chiudere plugin, mandare messaggi in chat, download e upload di file, etc) cosa succede? Un sistema così strutturato dopo un certo periodo di tempo, che dipende dalla quantità di risorse richieste, va in ***stallo***.
Come la definizione stessa di stallo suggerisce:

« *Un insieme di processi si trova in una situazione di stallo (deadlock) se ogni processo dell'insieme aspetta un evento che solo un altro processo dell'insieme può provocare* »[23]

Lo stallo può essere visto come un errore logico dell'intero programma. Non provoca alcun errore nell'esecuzione dei processi coinvolti che tentano di proseguire in attesa di risorse che non arriveranno mai. Nel nostro particolare caso, il concetto di "*processo*", nella definizione, va sostituito con il termine "*plugin*", mentre il concetto di "*evento*" va inteso come "*crediti*" nel senso di moneta.

Descriviamo ora il motivo per cui il sistema raggiunge la condizione di stallo. La ragione è insita[24] nella struttura del sistema stesso e dipende dall'estrema varietà di comportamento dei plugin, in termini dell'interazione reciproca.
Secondo questa regola (l'interazione che un plugin ha con gli altri) possiamo distinguere tre categorie:

[22] Per consentire un più semplice sviluppo e test dei plugin, è attualmente possibile eseguire PariPari senza bloccare le richieste di risorse in caso si possegga un account con un valore negativo di crediti. Questa funzione, presente esclusivamente per ragioni di test, viene chiamata disattivazione del modulo Crediti.
[23] http://it.wikipedia.org/wiki/Deadlock
[24] Irremovibile a meno di cambiare l'intero meccanismo di interazione tra plugin.

- **Plugin Iniziali**. Gran parte dei plugin appartiene a questa categoria. Sono plugin che utilizzano i servizi offerti da altri ma non implementano nessun servizio ulteriore. Possono essere denominati plugin end-user, nel senso che offrono una certa funzionalità all'utente. Mulo, Torrent, IRC, Web Server e altri ancora appartengono a questa categoria.
- **Plugin Intermedi**. Utilizzano i servizi offerti da certi plugin per fornire servizi più complessi ad altri plugin. DHT appartiene a questa categoria.
- **Plugin Finali**. Non dipendono da alcun servizio. Amministrano le risorse fisiche della macchina (porte di comunicazione, disco, etc) offrendo servizi ad altri plugin per un utilizzo semplice ed efficace delle risorse. Connectivity e Local Storage appartengono a questa categoria.

Analizziamo un semplice flusso di crediti tra le categorie appena elencate per comprendere il problema.

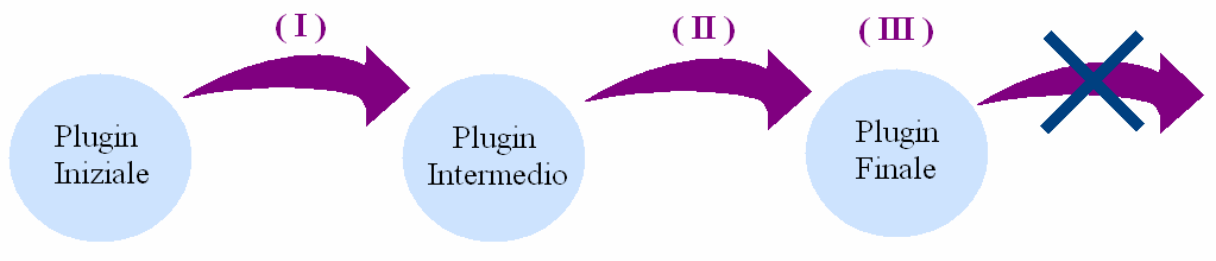

Figura 2.1 : Tipico flusso di crediti tra plugin.

I. Un plugin Iniziale richiede un servizio di un plugin Intermedio. I crediti vengono scalati dal plugin Iniziale per essere immagazzinati temporaneamente nella transazione.
II. Se tutto va a buon fine, alla scadenza della transazione i crediti vengono depositati nell'account del plugin Intermedio, che può utilizzarli per comprare risorse da un plugin Finale.
III. Se anche qui tutto va a buon fine, i crediti vengono trasferiti dal plugin Intermedio a quello Finale.

Ora sorge il problema: questi crediti diventano, di fatto, inutilizzabili! Poiché, per sua natura, un plugin Finale non richiederà mai alcun servizio, i crediti che gli vengono forniti resteranno bloccati per sempre.
In base alla velocità con la quale vengono effettuate le richieste di servizi, entro breve termine, uno dopo l'altro, tutti i plugin presenti resteranno senza crediti mandando in stallo l'intero sistema; ciascun plugin resta in attesa di nuovi crediti che nessuno gli può fornire.

2.2 – Grafo tra plugin

Per risolvere lo stallo e per fornire un'infrastruttura di base per gestire qualsiasi problema legato al flusso di crediti, gran parte dei quali verranno analizzati nel dettaglio nel prossimo capitolo, è stata introdotta una nuova struttura dati nel modulo Crediti: il ***grafo tra plugin***. Questa struttura deve poter tenere traccia dei trasferimenti in corso, dei trasferimenti conclusasi con successo e non, della situazione finanziaria (crediti in banca) di ogni plugin caricato e, in base a queste informazioni, prendere le decisioni opportune.

Il grafo è, ovviamente, formato da nodi e archi. Analizziamo nel dettaglio cosa essi rappresentano nel nostro contesto:

- Un **nodo** del grafo rappresenta un plugin correntemente caricato nella macchina locale. Ogni nodo è identificato univocamente dal nome del plugin, che ci viene fornito dal Core. Esso viene creato nel momento in cui un plugin viene caricato e viene eliminato dalla struttura nel momento in cui il plugin viene fermato.
- Ogni **arco orientato** del grafo rappresenta il percorso e la direzione di un possibile trasferimento di crediti nel corso dell'esecuzione del programma. È, infatti, possibile conoscere, sin dal momento dell'avvio di un plugin, quali saranno i percorsi possibili (non è detto che poi siano effettivamente utilizzati) dei crediti in uscita e dei crediti in ingresso. Tutte le informazioni sono disponibili indirettamente nel file *descriptor.xml*. La lista dei servizi richiesti per il funzionamento è contenuta tra i tag `<dependencies>` del file XML e verrà chiamata appunto *dependencies*. La lista dei servizi offerti è contenuta invece tra i tag `<interfaces>` e verrà chiamata *implementedAPIs*.[25]
  Una volta raccolte queste informazioni dal file, la costruzione degli archi è concettualmente semplice:
  - Per ogni elemento di *dependencies* devono essere creati uno o più archi verso quei nodi che dichiarano di implementare quel servizio nel loro *implementedAPIs*.
  - Per ogni elemento di *implementedAPIs* devono arrivare uno o più archi dai nodi che dichiarano quel servizio nel loro *dependencies*.

Per rappresentare i concetti appena esposti verrà utilizzata la raffigurazione presente in Figura 2.2. In 2.2 (a) è rappresentato il singolo nodo/plugin; sulla sinistra vengono elencati i servizi implementati, sulla destra i servizi da cui si dipende. In 2.2 (b) è invece rappresentato un arco tra i plugin P1 e P2 che lega la dipendenza API1 di P1 all'implementazione della stessa in P2.

[25] *Dependencies* ed *implementedAPIs* sono esattamente i nomi dati alle variabili nell'implementazione del grafo.

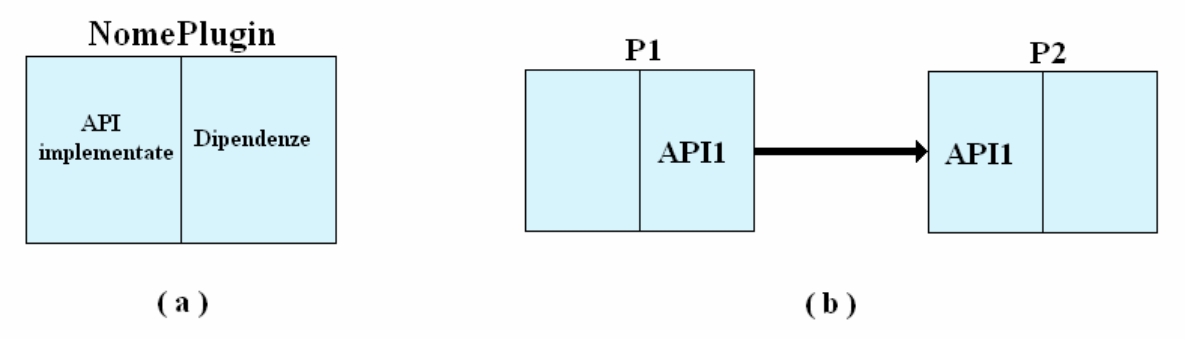

Figura 2.2 : Rappresentazione di nodi (a) e di archi (b) del grafo tra plugin.

Il grafo così costruito ha delle caratteristiche che meritano un approfondimento:

- Non è un grafo semplice ma un ***multigrafo***. Per definizione[26], un multigrafo è un grafo provvisto di cappi o multi-archi (più archi tra la stessa coppia di nodi). Nel nostro caso:
  - I *cappi* non sono possibili. Non è possibile per un plugin dipendere da un servizio che esso stesso implementa.
  - I *multi-archi* sono invece possibili, anzi sono estremamente comuni. Un plugin può dipendere, infatti, da più di un servizio offerto da uno stesso plugin.

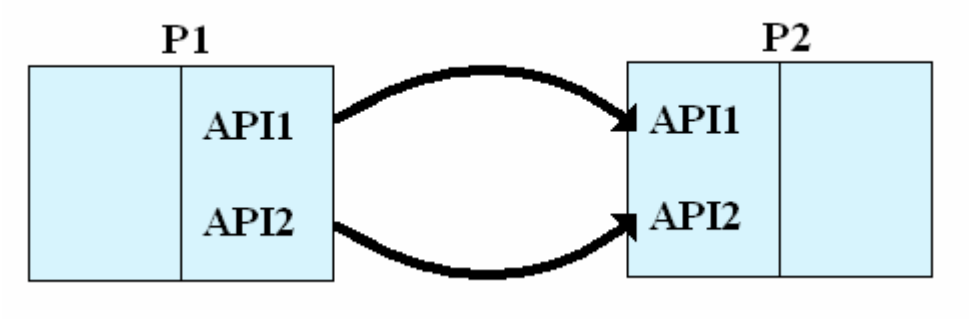

Figura 2.3 : Rappresentazione di un possibile multi-arco.

Una conseguenza di questa caratteristica è l'impossibilità di rappresentare il grafo tramite la matrice di adiacenza.

- È possibile la formazione di ***cicli***. Benché, a prima vista, la struttura del progetto possa fuorviare e far propendere nel dire che il grafo sia aciclico, sotto

[26] http://en.wikipedia.org/wiki/Multigraph

particolari condizioni si possono formare cicli di qualsiasi dimensione. La condizione principale è la presenza di plugin che implementano API totalmente indipendenti fra loro. Ad esempio, supponiamo di trovarci nella condizione più semplice illustrata in Figura 2.4.
Il plugin P1 dipende dall'API API1 che viene implementata dal plugin P2; nulla vieta, per come è strutturato il sistema, al plugin P1 di implementare l'API API2, totalmente indipendente dalla precedente, e al plugin P2 di servirsene. Aumentando la complessità del sistema si possono creare cicli di dipendenze fra 3, 4 o più plugin.

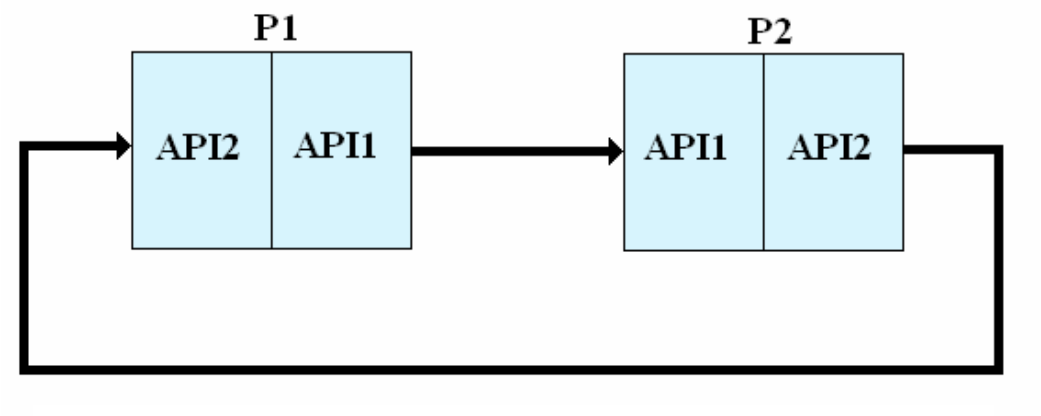

Figura 2.4 : Rappresentazione di una dipendenza ciclica tra due plugin.

La conseguenza di questa caratteristica è l'impossibilità di sfruttare la semplicità dei grafi orientati aciclici per sviluppare le tecniche di equa distribuzione analizzate nel Capitolo 3.
Notiamo inoltre come, allo stato attuale di sviluppo del progetto PariPari, non siano presenti dipendenze cicliche di alcun genere.

2.3 – Risolvere lo stallo

La Figura 2.5 permette di visualizzare in maniera chiara il problema dello stallo del sistema. I crediti, inizialmente presenti nella parte sinistra della figura, vengono trasferiti transazione dopo transazione verso la parte destra dello schema. La presenza di cicli non fa altro che ritardare questa inevitabile conseguenza, permettendo a solo una frazione di crediti di trasferirsi più volte tra i nodi formanti il ciclo.
Il modo più semplice di risolvere il problema è introdurre un sistema che permetta ai crediti giunti ad un plugin Finale, di trasferirsi ad un plugin Iniziale, ritornando quindi ad essere utilizzabili.

Per realizzare ciò si è fatto ricorso all'uso dei ***thread*** Java. Un unico thread viene attivato al momento dell'avvio del programma; ad intervalli stabiliti dalle condizioni di carico del sistema, esso preleva tutti i crediti bloccati nei plugin Finali e li trasferisce, secondo una precisa legge, nei plugin Iniziali.

È interessante notare come questa tecnica crei un *plugin virtuale* che, idealmente, collega tutti i plugin Finali a tutti gli Iniziali, realizzando così un unico grande ciclo nel quale i crediti non si possono più bloccare.

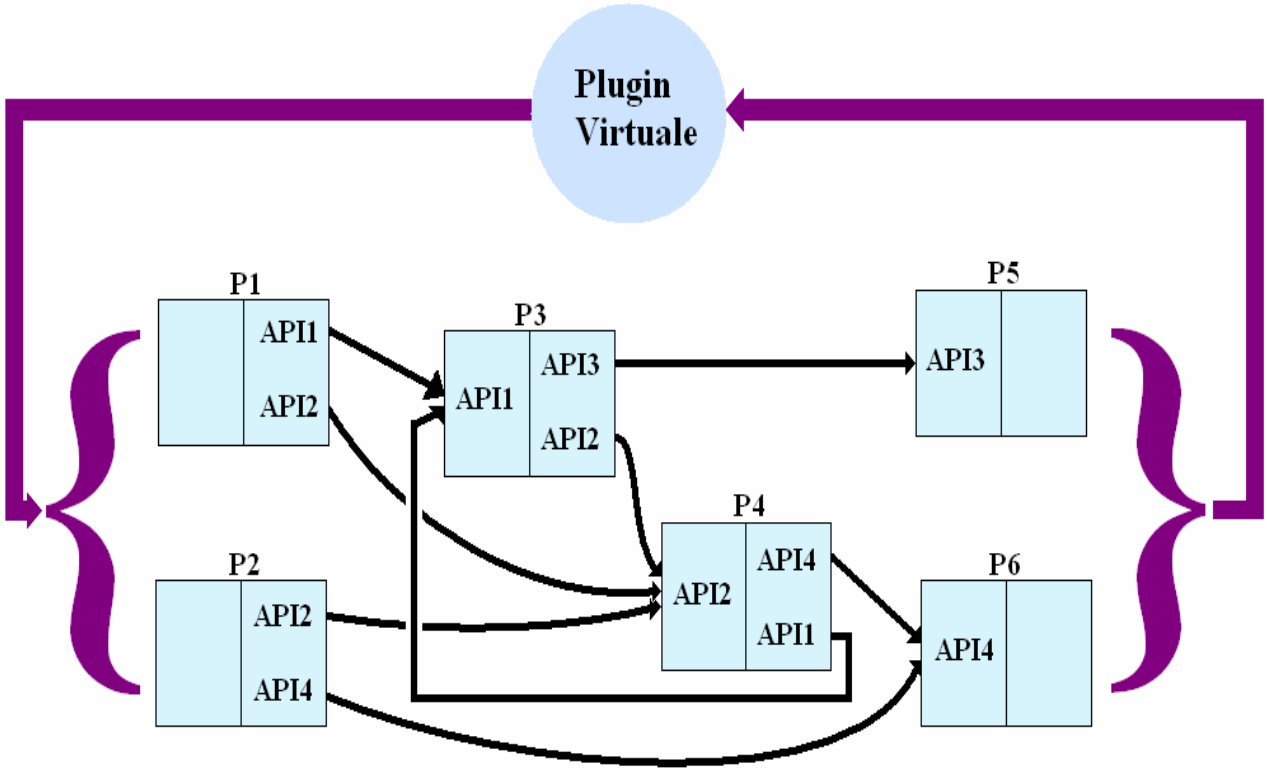

Figura 2.5 : Esempio di grafo completo tra 6 plugin e rappresentazione della tecnica per risolvere lo stallo.

Capitolo 3

Equa distribuzione

3.1 – I problemi

Ora che il problema dello stallo del sistema è stato risolto, vogliamo utilizzare la struttura del grafo per raggiungere un altro obiettivo: una logica e corretta distribuzione delle risorse disponibili tra i plugin che ne fanno richiesta.
Per comprendere a fondo questo, che, di fatto, può essere considerato un problema del sistema attuale, vanno considerati due aspetti fondamentali del funzionamento del modulo Crediti:

- Un plugin può avvalersi di un servizio pubblicato solo se ha crediti sufficienti per pagare il servizio.
- I plugin restano, comunque, delle unità funzionali indipendenti tra loro. Non hanno alcuna coscienza della presenza di altri plugin caricati contemporaneamente nella macchina, e perciò ciascuno considera le risorse presenti come se fosse l'unico ad utilizzarle.

Il fine ultimo e la vera novità del modulo Crediti è proprio questa: frenare la corsa alle risorse organizzando una politica di sfruttamento *equa* ed *efficace*. Questo comporta:

- ***Equità***. Permettere ad ogni plugin di rivaleggiare alla pari nell'acquisto delle risorse, nei limiti dei crediti disponibili.
- ***Efficacia***. Consentire sempre lo sfruttamento totale delle risorse disponibili, siano esse utilizzate da un solo plugin o da tutti i plugin.

Il grosso ostacolo da superare è il fatto che questa politica deve essere imposta indirettamente, ossia bilanciando la quantità di crediti che un plugin possiede non solo con le effettive richieste dello stesso ma anche basandoci sulle richieste degli altri plugin e sulla situazione globale del sistema.

Per raggiungere questi obiettivi iniziamo con l'analizzare i fattori chiave su cui focalizzeremo l'attenzione:

- *Tipologia di plugin*. A seconda della tipologia di plugin (Iniziale, Intermedio, Finale) cambia drasticamente l'interazione che si ha con il resto del sistema. Benché si sia tentato, senza successo, di sviluppare delle tecniche di equa distribuzione che potessero essere indipendenti dalla tipologia, alla fine si è deciso di optare per delle tecniche separate, più specifiche e quindi più efficaci per ciascuna tipologia.
- *Situazione globale del sistema*. L'intero sistema, come collezione di plugin, può trovarsi in varie situazioni nel corso della sua esecuzione che si possono riassumere in:
    - *Situazione Statica*. Nessun plugin fa richieste e non si sviluppa quindi nessun flusso di crediti. Attenzione: il fatto di non esserci richieste non significa che i plugin siano in standby senza fare nulla. Possono, infatti, essere in una fase in cui effettuano solo operazioni interne e non richiedono servizi altrui.
    - *Situazione Standard*. È la condizione di utilizzo più normale. Ciascun plugin (o comunque la maggior parte) effettua periodicamente delle richieste, il che comporta un flusso pressoché equilibrato di crediti in ogni direzione.
    - *Situazione di Picco Sparso*. Un numero limitato di plugin effettua elevate richieste, mentre i restanti plugin sono in situazione Statica. Questo comporta il dover convogliare la maggior parte dei crediti nei percorsi ad alto utilizzo, limitando al minimo i crediti inutilizzati.
    - *Saturazione*. Tutti (o quasi) i plugin effettuano un numero elevato di richieste saturando il sistema, ossia comprando tutte le risorse rese disponibili. Notiamo come in questa situazione la quantità di crediti che circola nel sistema è assai bassa (mentre si resta in Saturazione) e può essere confusa con la situazione Statica se si osserva solo il flusso di crediti generato.
- *Frequenza specifica*. Un plugin, indipendentemente dalla sua tipologia, può interagire con il resto del sistema ad una frequenza[27] molto varia. Certi plugin effettuano richieste molto frequentemente per realizzare i propri obiettivi, altri invece richiedono i servizi altrui molto raramente.

Sono state analizzate, sulla base dei parametri appena elencati, le situazioni che si possono venire a creare e i problemi legati. Sono state quindi sviluppate cinque importanti tecniche che, sfruttando il grafo ed ampliandone le funzionalità, permettono di garantire una buona equa distribuzione.

[27] Con il termine frequenza vogliamo indicare il numero di richieste al secondo che un plugin effettua in un certo intervallo di tempo; è quindi da considerarsi un valor medio. Non possiamo, infatti, prescindere sulla periodicità delle richieste.

3.2 – Frequenza di ridistribuzione

In questo paragrafo analizzeremo la durata dell'intervallo di tempo tra una ridistribuzione di crediti e la successiva, da parte del thread introdotto in *2.3 – Risolvere lo stallo*.
La proprietà che si desidera fornire a questo valore è: qualsiasi sia la situazione in cui si trova il sistema, l'attesa per la prossima ridistribuzione deve essere tale da evitare la mancanza di crediti per qualsiasi plugin di tipo Iniziale. Infatti, i plugin Iniziali per loro natura non hanno alcun modo per guadagnare crediti vendendo servizi; la loro possibilità di comprare le API da cui dipendono si basa solo ed esclusivamente sulla fornitura di crediti che il modulo Crediti concede.
Ecco quindi che rifornire i plugin Iniziali ad intervalli[28] ottimali affinché non restino privi di crediti, nonostante la presenza di crediti bloccati nei plugin Finali, è un parametro fondamentale per un efficiente utilizzo delle risorse.

La possibilità di definire una frequenza di distribuzione, a priori, unica che possa essere ottimale in qualsiasi situazione è un'ipotesi da scartare immediatamente. Analizziamo quindi una ***soluzione dinamica***, che calcoli di volta in volta il valore più adatto alla situazione.

Il problema di fondo, da risolvere, è il seguente: come si può sapere quando un plugin rimarrà senza crediti? In altre parole, com'è possibile conoscere in anticipo quale sarà l'evoluzione futura del sistema?
Il modo migliore per sapere cosà accadrà nel prossimo futuro è guardare al passato. Vengono quindi monitorati gli scambi effettuati dai plugin nel recente passato per ipotizzare come si comporteranno nel futuro.
Iniziamo con il considerare le seguenti informazioni/variabili[29]:

- $T_{exDistr}$ = Ultimo valore calcolato per il tempo di ridistribuzione. Quando la variabile viene analizzata, ha anche il significato di tempo trascorso dall'ultima ridistribuzione di crediti.
- C_{spesi} = Quantità di crediti spesi dal plugin nell'ultimo periodo. Si è scelto di considerare, in tutte le tecniche introdotte, l'andamento degli scambi solo dell'ultimo periodo poiché è fondamentale avere un sistema reattivo negli istanti in cui ci sono grosse variazioni nell'utilizzo delle risorse. Tenere in considerazione anche l'andamento passato, rende più lento il sistema nell'accorgersi dei cambiamenti.
- C_{acc} = Crediti disponibili nell'account del plugin al momento dell'analisi.

Il nostro scopo è determinare un corretto:

[28] Viene qui usata la parola "intervallo" e non "frequenza" poiché, come si vedrà, il valore viene modificato ogni volta che scade l'ultimo periodo calcolato.
[29] Nella trattazione teorica di questo capitolo possono essere considerati concetti base, ossia informazioni che "qualcuno" mi fornisce e che posso usare in ogni momento. Nella fase di implementazione, la maggior parte di questi concetti sono stati tramutati in variabili di programma.

- T_{distr} = Nuovo valore per il tempo di ridistribuzione.

Volendo esprimere in formule il nostro obiettivo di evitare che un plugin rimanga senza crediti prima della prossima ridistribuzione, possiamo legare le nostre variabili in questo modo:

(3.1)
$$C_{acc} \geq \frac{C_{spesi}}{T_{exDistr}} T_{distr}$$

Invertendo la formula, esplicitando l'incognita, otteniamo:

(3.2)
$$T_{distr} \leq C_{acc} \frac{T_{exDistr}}{C_{spesi}}$$

Volendo far valere la formula *(3.2)*, non solo per un singolo generico plugin, ma per tutti gli N_{Ini} plugin Iniziali, definiamo:

$$i = 1 , \dots , N_{Ini}$$

un indice per i suddetti plugin. Possiamo ora estendere i concetti espressi a tutti i plugin di questo tipo:

(3.3)[30]
$$T_{distr} = \frac{9}{10} \min_{i} \left(\frac{C_{acc}(i) \cdot T_{exDistr}}{C_{spesi}(i)} \right)$$

Dove si è sostituito il ≤ con un minimo e si è introdotto un fattore 9/10 al risultato finale per evitare di ottenere il valore limite, facilmente superabile nel caso in cui vengano effettuate più richieste di quelle previste.

[30] Benché non formalmente corretto, si è deciso in tutto il testo di usare una notazione con parentesi tonde per indicare la dipendenza di una variabile dagli indici, al posto della più classica notazione a pedici. Il motivo è, semplicemente, evitare un'eccessiva pesantezza delle formule con troppi pedici, e far risaltare la dipendenza di certe variabili dagli indici in questione.

Il valore così ottenuto ha un possibile range di variazione [0 , $+\infty$][31]. Introdurre un limite inferiore (T_{MIN}) e superiore (T_{MAX}) è necessario per i seguenti motivi:

- Introdurre T_{MIN} evita periodi troppo brevi che possono portare ad un eccessivo overhead nel sistema, impegnato a calcolare troppo spesso il modo migliore per ottenere un'equa distribuzione di crediti occupando quindi cicli macchina al posto dei vari plugin.
- Introdurre T_{MAX} ha lo scopo di evitare periodi troppo lunghi. Infatti, più aumenta l'intervallo di tempo più aumenta la probabilità che ci siano delle variazioni considerevoli nell'uso delle risorse che non possono essere riscontrate prima della prossima ridistribuzione. Per mantenere quindi una buona prontezza del sistema al variare, spesso molto rapido, nell'utilizzo delle risorse, risulta necessario ridistribuire, in certi casi, prima del previsto.

I valori di T_{MIN} e T_{MAX} qui introdotti andranno determinati sperimentalmente per permettere comunque un ampio range di variazione al valore dell'intervallo, eliminando le situazioni troppo eccessive.

Un'analisi più approfondita alla situazione di Saturazione del sistema, ha fatto emergere un comportamento non voluto: in condizioni di intenso utilizzo delle risorse può accadere, come caso limite, che nessun plugin Iniziale riesca a comprare risorse perché non ha crediti a sufficienza. In tale condizione, o comunque in una situazione simile in cui molti plugin sono in questo stato, il tempo di ridistribuzione resta fissato al valore massimo T_{MAX}. Questo comportamento risulta controproducente poiché, se ci sono pochi crediti che circolano[32], vorrei poter ridistribuire quelli arrivati al livello Finale il più spesso possibile.
Per gestire correttamente questa situazione è necessario introdurre una nuova informazione:

- Flag = Valore booleano che vale 1 se il plugin, nell'ultimo periodo, ha effettuato almeno una richiesta che non è andata a buon fine per mancanza di crediti. Vale 0 altrimenti.

Usando questa nuova variabile per calcolare:

- N_F = Numero di plugin Iniziali con Flag attivo.

È ora possibile aggiustare la formula *(3.3)* per abbassare il valore dell'intervallo in casi di Saturazione:

[31] E' un intervallo puramente teorico. In pratica il limite superiore è dato dal massimo valore rappresentabile per il tipo di dato scelto.
[32] Ricordiamo che, se siamo in condizione di Saturazione, la maggior parte dei crediti è immagazzinata nelle transazioni in corso. Per questo motivo risultano esserci pochi crediti liberi.

(3.4) $$T_{distr} = T_{distr} - \frac{\left(T_{distr} - T_{MIN}\right)}{N_{Ini}} N_F$$

Come si può notare, questo aggiustamento:

- Non modifica il valore precedentemente calcolato nel caso in cui nessun Flag sia attivo.
  $N_F = 0 \rightarrow T_{distr} = T_{distr}$
- Per ogni Flag attivo diminuisce proporzionalmente il tempo, per arrivare al caso limite in cui tutti i Flag sono attivi ed il tempo si riduce a T_{MIN}.
  $N_F = N_{Ini} \rightarrow T_{distr} = T_{MIN}$

3.3 – Ridistribuzione

In questo paragrafo analizzeremo la tecnica usata per ridistribuire equamente, tra i plugin Iniziali, i crediti bloccati nei plugin Finali. La formula finale che verrà proposta è calcolata dal thread di ridistribuzione ad ogni risveglio[33] ed è estremamente importante poiché, da essa, dipende la capacità dei plugin Iniziali di poter comprare risorse e quindi eseguire, il più velocemente possibile, i compiti impartiti dall'utente.

Approfondiamo il significato di "equa distribuzione" a questa particolare tecnica che ci apprestiamo a discutere. Per distribuire equamente i crediti dovrà essere fornita a ciascun plugin Iniziale, entro i limiti imposti dalla quantità di crediti bloccati al livello Finale, una quantità di denaro legata sia alle proprie esigenze sia alle richieste degli altri plugin coinvolti.
Di fondamentale importanza per i nostri calcoli è la seguente informazione:

- $C_{daDistr}$ = Crediti da distribuire. Rappresenta la quantità di crediti che, al momento dell'analisi, sono bloccati al livello Finale e che, quindi, sono disponibili per la ridistribuzione al livello Iniziale.

Inoltre, verranno riutilizzate alcune variabili già introdotte nel paragrafo precedente e che mantengono lo stesso significato:

- $C_{spesi}(i)$ con $i = 1 , \dots , N_{Ini}$.
- $Flag(i)$ con $i = 1 , \dots , N_{Ini}$.

[33] Quindi dopo T_{distr} secondi dall'ultima ridistribuzione di crediti.

Il nostro scopo è quello di determinare il miglior:

- $C_{asseg}(i)$ = Crediti assegnati. Quantità di crediti assegnati al plugin i-esimo.

Ogni elemento del vettore C_{asseg}[34] è compreso nel range [0 , $C_{daDistr}$] estremi compresi. Inoltre ha la proprietà:

(3.5)
$$\sum_{i=1}^{N_{Ini}} C_{asseg}(i) = C_{daDistr}$$

Partiamo, nello sviluppare la soluzione migliore per calcolare il generico elemento $C_{asseg}(i)$, da una semplice considerazione: per legare fra di loro i comportamenti del generico plugin i-esimo e i restanti ($N_{Ini} - 1$) possiamo assegnare i crediti in modo proporzionale all'utilizzo di risorse nel recente passato, supponendo che nel prossimo futuro i plugin mantengano l'andamento delle richieste.
La formula:

(3.6)
$$C_{asseg}(i) = \frac{C_{spesi}(i)}{\sum_{j=1}^{N_{Ini}} C_{spesi}(j)} C_{daDistr}$$

Esprime esattamente quanto richiesto. A ciascun plugin viene assegnato una porzione di $C_{daDistr}$ proporzionale ai crediti spesi (numeratore), ma contemporaneamente strettamente legata al comportamento degli altri plugin (denominatore).
Ovviamente la formula appena considerata verifica la proprietà *(3.5)*:

$$\text{Proof}: \sum_{i=1}^{N_{Ini}} C_{asseg}(i) = \sum_{i=1}^{N_{Ini}} \left(\frac{C_{spesi}(i)}{\sum_{j=1}^{N_{ini}} C_{spesi}(j)} C_{daDistr} \right) = C_{daDistr}$$

[34] Di fatto C_{asseg} può essere visto come un vettore di N_{Ini} elementi che fornisce tutte le informazioni su come sono stati suddivisi i crediti bloccati.

Consideriamo ora la situazione in cui uno o più plugin Iniziali hanno il Flag attivo. In tale condizione l'equazione *(3.6)* non fornisce alcun vantaggio ai plugin che sono rimasti senza crediti rispetto a quelli che, invece, sono riusciti a "cavarsela" con i crediti a disposizione. Se quindi un plugin ha necessità di più crediti rispetto a quelli che gli sono stati forniti all'ultima distribuzione, per esaudire le richieste dell'utente e, contemporaneamente, altri plugin possono accettare una diminuzione della loro parte (quelli che non hanno il Flag attivo sono perfetti per questo scopo) allora un surplus ai primi può essere concesso.
Per applicare questo concetto, riutilizziamo la variabile Flag ed estendiamo la formula *(3.6)* in questo modo:

(3.7)
$$C_{asseg}(i) = \frac{\left(1 + Flag(i)\cdot\alpha\right)\cdot C_{spesi}(i)}{\sum_{j=1}^{N_{Ini}}\left[\left(1 + Flag(j)\cdot\alpha\right)\cdot C_{spesi}(j)\right]} C_{daDistr}$$

Sfruttando il fatto che la variabile Flag è booleana[35], abbiamo fornito un surplus di crediti (fattore α) ai plugin che lo necessitano. Notiamo come, in condizione di Saturazione in cui tutti i Flag sono attivi, la presenza di questo fattore è irrilevante ai fini delle porzioni assegnate.

Un α ottimale andrà determinato sperimentalmente per evitare di fornire un eccessivo vantaggio ai plugin rimasti senza crediti.

Un'analisi ancora più approfondita della situazione a Flag attivi ha permesso di migliorare ulteriormente la formula *(3.7)*.
Consideriamo la condizione più semplice in cui due plugin hanno il Flag attivo ma un diverso andamento:

- Uno ha speso molto prima di esaurire i crediti e quindi attivare il Flag.
- L'altro ha speso molto meno del precedente.

Applicando in questa condizione la formula *(3.7)*, il surplus di crediti viene fornito in maniera proporzionale a quanto hanno speso prima di restare senza, quindi, di fatto, avvantaggiando sempre di più il plugin che aveva speso di più. Questo comportamento è esattamente l'opposto di quello voluto; se un plugin ha speso meno di un altro prima di attivare il Flag, questo significa che aveva meno crediti a disposizione e quindi dovrebbe essere avvantaggiato rispetto a quelli che ne avevano di più.

Per realizzare ciò è stato deciso di rendere il coefficiente α non più costante ma funzione dei crediti spesi. Definito:

- C_{min} = Minimo valore di crediti spesi tra plugin Iniziali con Flag attivo.

[35] Flag assume i valori { 0 , 1 } non { vero , falso }.

La nuova definizione di α risulta:

(3.8)[36]
$$\alpha\left(C_{spesi}\right)=\frac{C_{\min}}{C_{spesi}}\alpha'$$

Dove α**'** è il coefficiente costante che corrisponde al significato di α nella formula *(3.7)*.
Non si fa altro, quindi, che fornire a ciascun plugin con Flag attivo un coefficiente di surplus variabile:

- Per il plugin che ha speso di meno ($C_{spesi} = C_{min}$) viene utilizzato α**'.**
- Per tutti gli altri un coefficiente minore ($C_{spesi} > C_{min}$).

3.4 – Crediti iniziali e crediti totali

Un aspetto che non è stato ancora toccato da questa analisi, è l'assegnazione dei crediti iniziali ad un nuovo plugin caricato. Una trattazione approfondita e una gestione accurata di questi valori, è fondamentale per la riuscita di uno scopo chiave del modulo Crediti: permettere lo sfruttamento completo delle risorse disponibili.

Per capire al meglio questo aspetto è interessante conoscere come esso viene gestito attualmente. Poiché, finora, nessuno aveva analizzato la questione approfonditamente, è stata utilizzata la tecnica più semplice e banale: al caricamento di un plugin gli venivano assegnati un numero di crediti costante, 100 per la precisione. È facile accorgersi come questo, o qualsiasi altro valore fosse scelto, risulta inadeguato perché non legato in alcun modo alla situazione del sistema:

- In certe situazioni può risultare troppo elevato, in tal caso creando un'abbondanza di crediti e non frenando la corsa alle risorse.
- In altre situazioni può essere troppo basso, non permettendo ai plugin di comprare certe risorse. Ad esempio, un plugin che stabilisca un prezzo per i propri servizi nell'ordine delle migliaia, rispetto ai 100 crediti di default, non potrà mai vedere le proprie API venir comprate.

Strettamente legato a questo problema ve ne è un altro, di pari importanza: determinare la quantità di crediti che è necessario aggiungere al sistema, al caricamento di un nuovo plugin. Poiché le due problematiche possono sembrare simili, quasi ridondanti, è necessario chiarificare la fondamentale differenza: al momento del caricamento di un nuovo plugin è necessario, ovviamente, ***assegnargli*** un certo numero

[36] Attenzione: in questo caso la notazione con parentesi tonde ha il significato di "in funzione di" e non di indice.

di crediti iniziali affinché possa iniziare la compravendita di servizi ma, contemporaneamente, la presenza di un nuovo attore nel sistema può modificare la quantità di servizi disponibili e quindi richiedere la presenza di una diversa quantità di ***crediti globali***.

Come si può notare, quello che inizialmente appariva come un unico problema, ha fatto emergere una coppia di problemi strettamente legati che verranno ora analizzati e risolti nel dettaglio.

3.4.1 – Creazione di nuovi crediti e crediti totali

Un fattore importantissimo nell'economia dell'intero sistema è la quantità di ***crediti totali*** presenti nel sistema stesso, indipendentemente dalla loro ripartizione tra i plugin presenti. Determina la possibilità o meno dei plugin di comprare servizi e la capacità di sfruttare a pieno le risorse che la macchina mette a disposizione.
Il modo migliore per determinare questo valore è renderlo in qualche modo proporzionale alla totalità di risorse presenti nel sistema in ogni istante. Questo può essere realizzato in due fasi:

- Al momento della ***creazione*** di un nuovo plugin viene aggiunta al sistema una quantità di crediti proporzionale alle risorse che esso dichiara di poter offrire nel file "*descriptor.xml*".
- Durante l'***esecuzione***, un plugin può modificare i valori e le quantità delle risorse offerte. Una qualsiasi variazione si deve riflettere con una modifica dei crediti totali presenti nel sistema.

Il significato di "*proporzionale*" usato finora è meglio precisato: i crediti che vengono aggiunti al sistema per la presenza di un nuovo servizio offerto sono esattamente i crediti necessari per comprare l'intero servizio alla massima qualità[37], per un tempo pari ad un *timeunit*.

Quindi riassumendo. Alla creazione di un nuovo plugin vengono analizzati i servizi che il plugin dichiara di implementare. Nel caso in cui non dichiari nulla non vengono aggiunti nuovi crediti nel sistema. Nel caso, invece, dichiari uno o più servizi vengono aggiunti i crediti necessari a comprarli. Ogni qual volta, a run-time, vengono modificate le quantità dei servizi offerti, vengono immediatamente aggiornati anche i crediti totali di conseguenza.

3.4.2 – Assegnazione dei crediti iniziali

Vogliamo determinare ora quale sia il valore più corretto di crediti da assegnare ad un nuovo plugin appena caricato. Possiamo subito notare che:

[37] Massimo valore delle *feature* caratterizzanti il servizio.

- Per i plugin Finali questa tecnica ha ben poca importanza poiché essi non comprano nessuna risorsa.
- Assai più importante è, invece, per i plugin Intermedi e Iniziali i quali necessitano di crediti per poter iniziare a comprare servizi.

Poiché, come già accennato, fornire un valore costante predeterminato di crediti non è una scelta accettabile, decidiamo di sfruttare il valore dei crediti totali appena analizzato nel paragrafo precedente, d'ora in poi $\mathbf{C_{totali}}$, per fornire ai nuovi plugin una quantità di crediti legata strettamente a questa variabile.

Non sapendo nulla del nuovo plugin (è un buono o un cattivo plugin, richiede tante o poche risorse) la soluzione più giusta è quella di fornirgli una porzione equa dei crediti totali. Definito quindi:

- N_{II} = Numero di plugin di tipo Iniziale e Intermedio caricati.

Decidiamo di assegnare ai nuovi plugin esattamente:

(3.9)
$$C_{iniziali} = \frac{1}{N_{II}} C_{totali}$$

Volendo quindi analizzare l'interazione tra le due tecniche dei paragrafi *3.4.1* e *3.4.2* al momento della creazione di un nuovo plugin possiamo affermare:

- Al caricamento di un *plugin Finale* vengono aggiunti al sistema i crediti necessari per comprare le risorse dichiarate. Poiché l'assegnazione di crediti iniziali non è importante in questo caso, tutti i crediti aggiunti gli vengono anche assegnati. Sarà poi compito del thread ridistribuire questi nuovi crediti.
- A caricamento di un *plugin Intermedio* vengono aggiunti al sistema i crediti necessari per comprare le risorse dichiarate. Contemporaneamente gli vengono forniti $C_{iniziali}$ crediti. La differenza tra $C_{iniziali}$ e i crediti aggiunti al sistema[38] andrà scalata ai crediti da ridistribuire al prossimo risveglio del thread.
- Al caricamento di un *plugin Iniziale* gli vengono forniti $C_{iniziali}$ crediti ma non vengono aggiunti nuovi crediti al sistema. Questo comporta il dover scalare sempre in negativo, i crediti da ridistribuire al prossimo risveglio del thread.

[38] La differenza può essere sia positiva che negativa a seconda della situazione.

3.4.3 – Eliminazione di plugin

Un problema analogo a quello del paragrafo *3.4.2* accade alla chiusura di un plugin, ossia decidere quanti crediti togliere dall'ammontare totale C_{totali} e cosa farne dei crediti presenti nell'account al momento dell'eliminazione.
Iniziamo col fornire un formalismo per queste informazioni:

- C_{finali} = Crediti presenti nell'account del plugin al momento della chiusura.
- $C_{servizi}$ = Porzione dei C_{totali} dovuta ai servizi definiti dal plugin in fase di eliminazione. In altre parole, i crediti necessari a comprare i servizi offerti dal plugin (ovviamente 0 per i plugin Iniziali).

Leghiamo le variabili in questo modo:

(3.10) $$\Delta = C_{finali} - C_{servizi}$$

- $\Delta > 0$ significa che sono presenti nell'account più crediti di quanti ne avevo aggiunti al sistema nel momento della creazione[39]. I crediti in eccesso verranno ridistribuiti nel grafo al prossimo risveglio del thread.
- $\Delta < 0$ significa che sono presenti meno crediti di quelli che avevo aggiunto alla creazione. In tal caso dovranno essere eliminati Δ crediti dal sistema perché in eccesso; il thread si preoccuperà di eliminarli dai crediti bloccati al livello Finale.

3.5 – Pesatura richieste

Focalizziamo ora la nostra attenzione sull'unica tipologia di plugin non ancora analizzata: i plugin Intermedi. Sappiamo che questi plugin guadagnano crediti vendendo i servizi implementati e, contemporaneamente, spendono crediti per comprare altri servizi. Possiamo inoltre, con buona approssimazione, supporre che i servizi comprati vengano usati solo ed esclusivamente per implementare i servizi offerti. In altre parole, le richieste fatte per scopi interni[40] sono una parte irrilevante delle richieste totali. Questa supposizione può essere fatta poiché questa tipologia di plugin non ha contatto diretto con l'utente (non è un plugin end-user), quindi essi sono in esecuzione solo se altri plugin utilizzano i loro servizi.

La domanda da porsi a questo punto è: i plugin Intermedi riescono ad operare senza alcun intervento da parte del modulo Crediti? La risposta è fornita, indirettamente, dalla seguente considerazione:

[39] E modificati opportunamente a run-time.
[40] Intendiamo richieste in uscita non legate/dovute ad una richiesta in ingresso, ma ad operazioni interne del plugin che necessitano comunque di servizi esterni.

- Certi plugin, per implementare una singola richiesta in ingresso sono costretti a comprare più di un servizio. In tal caso i crediti tenderanno, progressivamente, a diminuire.
- Altri plugin, per implementare più richieste in ingresso riescono a fare solo una richiesta in uscita. In tal caso i crediti tenderanno ad aumentare.
- Anche supponendo, nel caso migliore, che ad una richiesta in ingresso ne corrisponda una in uscita, non è detto che il costo della seconda sia comparabile col guadagno della prima.

Questa breve analisi costringe ad introdurre una nuova tecnica per l'equa distribuzione di crediti applicabile alla sola categoria dei plugin Intermedi: lo scopo sarà tentare di equilibrare, in ogni condizione di utilizzo, i crediti in ingresso con i crediti in uscita.

Per fare ciò è stato deciso di calcolare un coefficiente **K**, specifico per ogni plugin Intermedio, che sarà usato per pesare il costo delle richieste in uscita tenendo conto anche del flusso di crediti in ingresso.

Il coefficiente è di tipo moltiplicativo e della durata di un intervallo di tempo; esso viene quindi ricalcolato all'inizio di ogni intervallo per essere sempre aggiornato con la storia più recente del plugin e modifica il prezzo ***X*** di un'ipotetica richiesta in uscita, al costo ***KX***. Poiché non è accettabile poter comprare servizi senza pagare e, a maggior ragione, non avrebbe senso guadagnare crediti comprando un servizio, il range di variazione per il coefficiente è **[** 0 **,** $+\infty$ **]** con zero escluso.

Analizziamo più in dettaglio il significato di **K** per un generico plugin Intermedio:

- **K = 1** implica che il plugin è in un ottimo equilibro tra costo dei servizi in uscita e ricavi derivanti dalla vendita.
- **K > 1** implica che il plugin tende ad accumulare crediti dai servizi offerti. Vogliamo quindi, con tale coefficiente, far pagare di più i servizi comprati per ristabilire l'equilibrio. Non è stato stabilito nessun limite superiore al valore assumibile.
- **K < 1** implica che il plugin tende a perdere crediti, non riuscendo a bilanciare con i servizi offerti. In tal caso il coefficiente diminuisce il costo dei servizi usati per ristabilire l'equilibrio. Per evitare di fornire praticamente gratis i servizi è stato deciso di dare un limite inferiore (K_{min}) al valore di **K** da determinare sperimentalmente. La variabilità del coefficiente si assesta quindi nel range **[** K_{min} **,** $+\infty$ **].**

Per determinare il valore corretto di **K** per ciascun plugin, basiamo la nostra analisi sull'interazione tra due variabili chiave:

(I) $$\Delta C = C_t - \frac{1}{N_{II}} C_{totali}$$

ΔC rappresenta lo scarto tra i crediti presenti nell'account del plugin al momento dell'analisi (C_t) e il valor medio di crediti/plugin[41] atteso in situazione Standard (C_{totali} / N_{II}). Questa variabile può risultare maggiore, minore o uguale a zero.

(II) $$C_{netti} = \sum C_{ingresso} - \sum C_{uscita}$$

C_{netti} rappresenta semplicemente un'estensione della variabile C_{spesi} già utilizzata più volte nel corso dei paragrafi precedenti, ma adattata a questa tipologia di plugin nei quali avviene un flusso di crediti sia in ingresso sia in uscita. Indica il bilancio dei crediti scambiati nell'ultimo periodo. Anche questa variabile può risultare maggiore, minore o uguale a zero.

Volendo equilibrare lo scarto ΔC entro un periodo pari all'ultimo intervallo di ridistribuzione, con un andamento recente di C_{netti} crediti, possiamo calcolare **K** per un generico plugin Intermedio in questo modo:

(3.11) $$K \cdot C_{netti} - C_{netti} = \Delta C$$

Esplicitando l'incognita:

(3.12) $$K = \frac{\Delta C}{C_{netti}} + 1$$

A questo punto possiamo estendere la formula *(3.12)* considerando la possibilità di bilanciare i crediti in più di un intervallo di ridistribuzione per evitare eccessive fluttuazioni di **K** (indichiamo questo numero di periodi con β). Consideriamo anche un surplus in caso di Flag attivo come già analizzato nel paragrafo 3.3 per i plugin Iniziali:

(3.13) $$K = \frac{\Delta C}{\beta \cdot \left(1 \pm Flag \cdot \alpha\right) \cdot C_{netti}} + 1$$

[41] Da leggere come "crediti per plugin".

Possiamo notare come, a differenza dell'analisi fatta nel paragrafo *3.3 – Ridistribuzione*, queste formule si basano esclusivamente sul comportamento del singolo plugin Intermedio; infatti, non viene fatto alcun riferimento agli indici. Questo deriva dal fatto che, mentre i plugin Iniziali erano strettamente legati fra loro poiché condividevano una fonte comune di approvvigionamento di crediti[42], i plugin Intermedi possono essere completamente indipendenti fra loro; in certi casi essi interagiscono con sezioni totalmente distinte del grafo rendendo quindi inutile un'analisi comune che leghi i loro comportamenti. Questo è anche il motivo per cui si è anche usato un coefficiente α costante.

Come si sarà notato in formula *(3.13)* è stato usato un ± che merita un'analisi più approfondita; a differenza dell'equazione *(3.7)* nella quale tutte le variabili potevano assumere solo valori positivi, in questo caso la presenza di possibili valori negativi in entrambe le variabili ΔC e C_{netti} comporta un più complesso andamento in caso di Flag attivi.
Analizziamo nel dettaglio i casi che si possono generare:

- $\Delta C > 0$ $C_{netti} > 0$. Ho più crediti della media e, inoltre, tendono ad aumentare nel tempo. Dalla formula otteniamo un coefficiente $\mathbf{K} > 1$ come, infatti, voluto. La presenza del Flag attivo in questa situazione è estremamente rara (ho $\Delta C > 0$ quindi dovrei avere crediti a sufficienza); in ogni caso diminuire il coefficiente, quindi far pagare meno, è la scelta migliore. Il segno $+$ nella formula realizza questo obiettivo.
- $\Delta C > 0$ $C_{netti} < 0$. Ho più crediti della media ma tendono a diminuire. $\mathbf{K}$ risulta essere minore di 1. La presenza del Flag attivo è, anche in questo caso, molto rara. Lo scopo resta comunque diminuire il coefficiente e questo può essere fatto ponendo il segno $+$ nella formula.
- $\Delta C < 0$ $C_{netti} > 0$. Ho meno crediti della media ma tendono, almeno recentemente, ad aumentare. Il valore di $\mathbf{K}$ ottenuto risulta negativo poiché è necessario continuare a far pagare meno del dovuto per raggiungere l'equilibrio. In caso di Flag attivo devo diminuire ancora più il coefficiente; questo è ottenuto ponendo un segno $-$ nella formula.
- $\Delta C < 0$ $C_{netti} < 0$. Ho meno crediti della media e tendono ancora a diminuire. In questa condizione ottengo un $\mathbf{K} > 1$ che porterebbe a far pagare più del voluto, il che non è ovviamente il risultato sperato. In questa particolare condizione non posso far altro che imporre $\mathbf{K} = K_{min}$ per limitare i costi. In queste condizioni un Flag attivo, per diminuire il costo delle risorse, dovrebbe intervenire con un segno $-$ nella formula.

[42] I crediti bloccati nel livello Finale e riassunti nella variabile $C_{daDistr}$.

Possiamo ora determinare il corretto segno da applicare alla formula (3.13); esso dipende dal segno di ΔC. Otteniamo quindi l'equazione finale desiderata:

$$K = \frac{\Delta C}{\beta \cdot \left(1 + sign\left(\Delta C\right) \cdot Flag \cdot \alpha\right) \cdot C_{netti}} + 1 \qquad (3.14)$$

3.6 – Idle Plugin

Un'ultima breve ma importante analisi può essere fatta considerando l'andamento dell'intero sistema e l'applicazione delle tecniche sviluppate finora, nella situazione di Picco Sparso[43].
In tale situazione, anche applicando tutte le tecniche esposte nei paragrafi precedenti, non si riesce a sfruttare a pieno le risorse del sistema. Il motivo è che i crediti presenti negli account dei plugin Statici non possono essere usati in alcun modo da quelli attivi, di fatto, diventando crediti bloccati ossia inutilizzabili per comprare ulteriori risorse, almeno fino a quando i plugin escono dalla fase Statica (il che può anche non accadere).

È necessaria quindi una tecnica che permetta di utilizzare, almeno in parte, i crediti bloccati nei plugin che recentemente non hanno fatto alcuna richiesta e distribuirli, se esistono, a quei plugin che invece hanno fatto così tante richieste da avere il Flag attivo.
Definiamo quindi un plugin in "***Idle Mode***" se ha queste caratteristiche:

- $C_{spesi} = 0$ per i plugin Iniziali o equivalentemente $C_{netti} = 0$ per i plugin Intermedi.
- $Flag = 0$ per evitare di rubare crediti a quei plugin che non hanno fatto richieste perché avevano pochi crediti.
- Type = Iniziale o Intermedio poiché i plugin Finali non partecipando attivamente negli scambi non devono essere contemplati in questa trattazione.

La tecnica introdotta prevede quindi di prelevare una certa percentuale di crediti (fattore γ da determinare sperimentalmente) dai plugin in *Idle Mode* e distribuirli, in maniera proporzionale, ai plugin con Flag attivo, se presenti.

[43] La definizione è presente nel paragrafo *3.1*.

Capitolo 4

Considerazioni implementative

4.1 – Casi particolari per *3.2 - Frequenza di ridistribuzione*

Analizziamo alcuni casi particolari della formula finale per la tecnica illustrata del paragrafo *3.2 – Frequenza di ridistribuzione*. La formula in questione è la *(3.3)*; osserviamo come essa si comporta in situazioni più o meno rare ma che meritano un approfondimento.

- Non ci sono plugin Iniziali.
  Questa condizione è comune nella fase di caricamento dei primi plugin, nella fase finale di chiusura del programma o comunque in ogni situazione in cui nessun plugin end-user è attivo. In tale condizione si è deciso di imporre il tempo di ridistribuzione a T_{MIN} in modo da essere consci, il prima possibile, dell'eventuale presenza di nuovi plugin.
- Uno o più plugin non hanno fatto richieste ($C_{spesi} = 0$) perché non ne avevano la necessità ($Flag = 0$).
  Non considerare la possibilità di avere $C_{spesi} = 0$ è un grave errore poiché provoca un'eccezione di tipo *ArithmeticException* a run-time bloccando l'esecuzione del programma. Il problema è stato risolto non considerando, preventivamente, nel calcolo tutti quei plugin in tale condizione.

4.2 – Casi particolari per *3.3 – Ridistribuzione*

Analizziamo ora alcuni casi significativi nell'utilizzo della formula *(3.7)* per la ridistribuzione di crediti.

- Non ci sono plugin Iniziali.
  In tale condizione non ha senso ridistribuire i crediti bloccati poiché non c'è alcuna destinazione valida. È stato deciso quindi di non effettuare la ridistribuzione e attendere il prossimo risveglio del thread che, in tale

situazione, è impostato seguendo il caso speciale omonimo in *4.1 – Casi particolari per 3.2 – Frequenza di ridistribuzione*.

- Nessun plugin ha fatto richieste ($C_{spesi} = 0$) perché non ne avevano la necessità ($Flag = 0$).
  Il problema di questa situazione è che me ne posso accorgere solo una volta analizzati tutti i plugin Iniziali. Poiché ci troviamo in situazione Statica molto probabilmente ci sono anche pochi crediti da ridistribuire. È stato quindi deciso di non effettuare la ridistribuzione poiché non ci sono dati a sufficienza per decidere in maniera corretta come suddividere i crediti.
- Uno o più plugin non hanno fatto richieste ($C_{spesi} = 0$) perché non avevano abbastanza crediti ($Flag = 1$).
  In questa situazione, estremamente eccezionale, alcuni plugin avevano così pochi crediti da utilizzare che non hanno potuto comprare nemmeno un servizio. Non avendo abbastanza notizie della loro storia precedente non si può far altro che fornire una porzione predeterminata dei crediti da ridistribuire ($C_{daDistr}$). Il modo migliore è quindi fornirgli esattamente:

(4.1)
$$C_{asseg}(i) = \frac{1}{N_{Ini}} C_{daDistr}$$

In modo tale che, nel caso peggiore in cui tutti i plugin Iniziali sono in questa situazione, i crediti bloccati vengono distribuiti equamente fra tutti.

4.3 – Casi particolari per *3.5 – Pesatura richieste*

Analizziamo ora alcuni casi interessanti della formula *(3.14)*, equazione finale del paragrafo *3.5 – Pesatura richieste*.

L'unico caso veramente di interesse è la possibilità di avere $C_{netti} = 0$; in tali condizioni la formula *(3.14)* non può ovviamente essere applicata poiché lancerebbe una *ArithmeticException*. Il modo migliore per risolvere la questione, però, non è unico. Dipende dalla condizione in cui mi trovo, determinando ben tre possibilità distinte:

- Non sono avvenute richieste né in uscita né in ingresso perché non era necessario, ossia ci troviamo in situazione Statica. Per determinare questo tipo di condizione è necessario introdurre una nuova informazione:
  - N_{rich} = Numero richieste in uscita eseguite dal plugin nell'ultimo intervallo.

La situazione appena descritta può essere identificata come:

$N_{rich} = 0$, Flag = 0

Non avendo alcuna informazione sulla situazione recente del sistema, si è deciso di mantenere il precedente coefficiente **K**, qualunque esso sia, in attesa di nuove informazioni.

- Non sono avvenute richieste perche non c'erano crediti a sufficienza.

  $N_{rich} = 0$, Flag = 1

  Non si può far altro che imporre $\mathbf{K} = K_{min}$ per far pagare il meno possibile al plugin in attesa che arrivino nuovi crediti.

- Le richieste si sono equilibrate autonomamente.

  $N_{rich} \neq 0$

  È una condizione estremamente rara ma da considerare. In tal caso:

  - Se $\Delta C > 0$ ho più crediti della media quindi decido di aumentare il coefficiente precedente da **K** a **(K + K/2)**.
  - Simmetricamente se $\Delta C < 0$ diminuisco il coefficiente precedente da **K** a **(K - K/2)**.

Capitolo 5

Sviluppi futuri

Come si sarà notato nella trattazione teorica (*Capitolo 3*), alcune costanti non sono state esplicitate poiché saranno determinate sperimentalmente. Il prossimo passo sarà, infatti, perfezionare il valore di queste costanti in casi reali di utilizzo del sistema.

Per realizzare ciò, è in corso di sviluppo un modulo grafico che permetta la visualizzazione del grafo tra plugin. Grazie a questo strumento si potrà analizzare in maniera visiva, quindi estremamente semplice, l'andamento dei crediti nel sistema e bilanciare, modificando le suddette costanti, il flusso di moneta nelle varie condizioni di utilizzo.

Riportiamo ora l'elenco delle costanti incontrate, il loro significato ed il valore da cui si partirà nell'analisi pratica:

- α o α' = Fattore di surplus per plugin Iniziali (α') e Intermedi (α). Rappresenta la frazione di crediti che fornisco in più a quei plugin con Flag attivo. Il valore di partenza sarà **0.1**.
- β = Numero di periodi entro i quali si desidera equilibrare il flusso di crediti nei plugin Intermedi. Si partirà da un'analisi con $\beta = 1$.
- γ = Frazione di crediti dei plugin in *Idle Mode* che si intende prelevare per distribuire ai plugin con Flag attivo. Si inizierà con **0.25**.
- T_{MIN} = Intervallo minimo di ridistribuzione del thread. Evita un eccessivo overhead del modulo Crediti rispetto ai plugin caricati. Un intervallo minimo di un paio di secondi sarà il valore iniziale.
- T_{MAX} = Intervallo massimo di ridistribuzione del thread. Evita attese troppo lunghe che diminuiscono la reattività del sistema ai cambiamenti nell'utilizzo delle risorse. Un intervallo massimo di 10-15 secondi potrebbe essere sufficiente.

- K_{min} = Coefficiente minimo di pesatura delle richieste per plugin Intermedi. Evita l'acquisto di risorse a prezzi troppo bassi. Si partirà da un limite di **0.1** rispetto al prezzo concordato senza alcuna pesatura.

Bibliografia

[1] Bertasi Paolo, 2005. *Progettazione e realizzazione in Java di una rete Peer To Peer anonima e multifunzionale*. Tesi di laurea.

[2] Andreon Mirco, 2009. *PariCredit 2009*. Tesi di laurea specialistica.

[3] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. *The eigentrust algorithm for reputation management in p2p networks*. In Proceedings of the Twelfth International World Wide Web Conference, 2003.

[4] Peserico Enoch, 2006. *P2P Economics*. In: ACM (Ed.), SIGCOMM '06: proceedings of the acm conference of the Special Interest Group on Data Communication.

[5] Degan Sebastiano, 2009. *PariCredit 2009: Guida dello sviluppatore*. Tesi di laurea triennale.