Alma Mater Studiorum · Università di Bologna

CAMPUS DI CESENA
SCUOLA DI SCIENZE
Corso di Laurea in Ingegneria e Scienze Informatiche

Tecniche di Resource Discovery nel Grid Computing

Relatore:
Prof.
Alessandro Ricci

Presentata da:
Stefano Belli

Sessione III
Anno Accademico 2014/2015

PAROLE CHIAVE

Grid Computing

Resource Discovery

Agente

Introduzione

Il Grid Computing è un paradigma del calcolo distribuito divenuto particolarmente importante nei tempi moderni grazie alla sua implementazione presso numerosi enti scientifici, che vedono nella sua versatilità, potenza e scalabilità una preziosa risorsa di calcolo.
Grazie alla sua concezione di "Resource", intesa come un insieme di elementi atti a collaborare nell'elaborazione di determinati compiti, la rete Grid dispone di una potenza di calcolo proporzionale al numero di nodi che la compongono, offrendo difatti un elevata scalabilità.
Tuttavia, dato che le risorse di una rete Grid possono essere del tutto eterogenee tra loro e geograficamente disperse, si pone il problema di individuare un algoritmo di "Resource Discovery" (cioè, un algoritmo capace di individuare le risorse adatte a un determinato compito) in grado di rispondere in maniera ottimale alle necessità della griglia.
Pertanto, questo testo si occuperà di analizzare i principali algoritmi di *Resource Discovery* attualmente in uso nelle griglie computazionali, valutando i principali vantaggi e svantaggi di ogni soluzione.
Verrà posta inoltre una particolare attenzione verso il *Resource Discovery* ad agenti, presentato in questo testo come una valida soluzione per la maggior parte dei casi.

L'elaborato è organizzato come segue: nel primo capitolo verranno presentati alcuni elementi fondamentali di una rete Grid, soffermandosi anche sull'importanza del Grid Computing nei tempi moderni (che lo vede in contrasto con il crescente

successo del Cloud Computing nelle applicazioni di rete).

Successivamente nel secondo capitolo verranno introdotti alcuni modelli di *Resource Discovery*, analizzandone inoltre alcune implementazioni pratiche. A fine capitolo sarà presente una tabella riassuntiva che sintetizza i punti di forza di ogni variante.

Infine, verrà analizzato il *Resource Discovery* ad Agenti, proponendolo come valida alternativa ai modelli tradizionali a fronte dei suoni numerosi vantaggi. Come per il capitolo precedente, verrà anche presentato un esempio pratico di *Resource Discovery* ad agenti: l'implementazione Kang et al.

Indice

Capitolo 1

Introduzione al Grid Computing

Il termine "Grid" (usualmente tradotto in italiano come "Griglia") è stato utilizzato per la prima volta da Ian Foster e Carl Kesselman in "The Grid: Blueprint for a New Computing Infrastructure", una pubblicazione che già allora formalizzava le basi del Grid Computing odierno.
Tuttavia, bisogna considerare come il Grid Computing abbia trovato impiego ben prima di allora: sin dagli anni '90, infatti, assunse un ruolo centrale nella elaborazione di grandi quantità di dati, sopratutto in ambito scientifico.
Ma che cos'è esattamente il Grid Computing?
Una delle definizioni più comunemente utilizzate per descrivere cos'è il Grid Computing è la seguente [1]:

> "Coordinated resource sharing and problem solving in dynamic, multiinstitutional virtual organizations"

Il Grid Computing permette la creazione di infrastrutture di rete capaci di lavorare su un singolo obiettivo (chiamato Task), attraverso la condivisione di risorse, compiti(chiamati "job") e servizi tra i calcolatori presenti nella rete.
Ciò che differenzia il Grid Computing dai sistemi di calcolo ad alte prestazioni convenzionali (come ad esempio i cluster di calcolo) è principalmente l'eterogeneità dei suoi componenti: le reti che lo formano sono, in genere, debolmente accoppiate, formate da nodi molto diversi da loro e geograficamente dispersi.
Tuttavia, grazie all'utilizzo di precisi algoritmi, tecniche e servizi, da tale rete è

possibile ricavare una elevata potenza di calcolo, paragonabile a quella erogata da singoli supercomputer.
Per garantire diversi parametri, come tempi di risposta, sicurezza e perfomance, le reti Grid sono gestite da speciali software chiamati "Middleware", che integrano al loro interno servizi di assegnazione dei job (chiamati "Job Scheduling"), servizi di sicurezza e di accesso al sistema.

Un elemento che merita approfondimento è senza dubbio il concetto di VO (**V**irtual **O**rganization), che ricopre un ruolo fondamentale nelle Griglie computazionali.
Una VO è un insieme di enti, organizzazioni oppure singoli individui, che condividono le proprie risorse (in un contesto controllato) in modo tale che i membri di una Virtual Organization possano collaborare tra loro al fine di conseguire gli stessi obiettivi.
In generale, una VO può essere costituita da:

-un insieme di individui o istituzioni

-un insieme di risorse da condividere

-o un insieme di regole per la condivisione

In questo testo ci occuperemo principalmente di risorse ("Resources"), intese come entità condivise per le necessità della griglia computazionale, e dei relativi protocolli di ricerca ("Resource Discovery").

Concludendo, in questo capitolo verranno presentati alcuni aspetti chiave dei sistemi Grid, per poi soffermarsi sulle tecniche di ricerca delle *Resources* nella rete: il *Resource Discovery*.

1.1 Architettura di rete

Il modello di riferimento dei sistemi Grid è detto "a clessidra" ("Hourglass")[2], nome dovuto sicuramente alla particolare organizzazione che questa architettura dà ai suoi componenti.

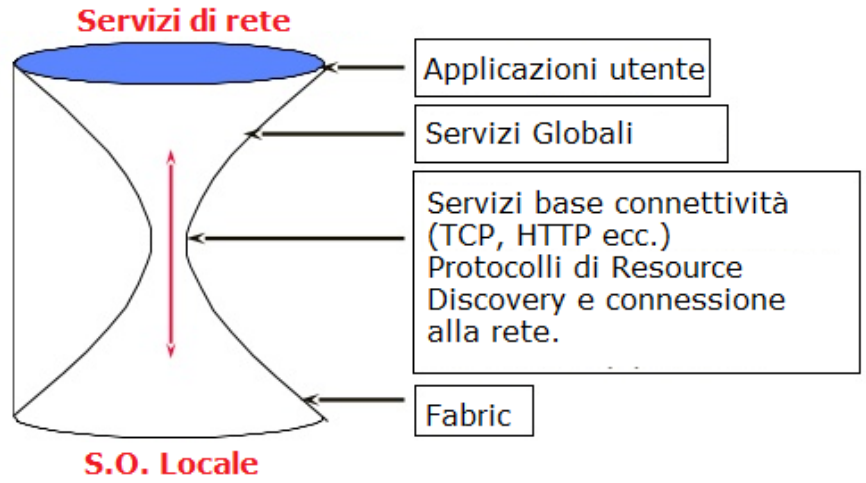

Figura 1.1: Architettura a clessidra

Come mostrato in figura, infatti, questo modello struttura i protocolli utilizzati nella rete attraverso un sistema a livelli, a cui corrisponde una diversa larghezza della clessidra.
Maggiore è la larghezza, migliore sarà l'estensibilità di tale protocollo nella rete: è possibile notare come questa architettura offra un alto grado di personalizzazione per quanto riguarda i tipi di resources (S.O. locale) e le applicazioni offerte dalla rete, ma sia molto limitato al centro, dove risiedono infatti alcuni protocolli fondamentali di internet.

Un altro approccio molto comune nello studio delle reti Grid è la suddivisione della griglia in livelli ben distinti, chiamati Fabric, Connectivity, Resource, Collective e Application.

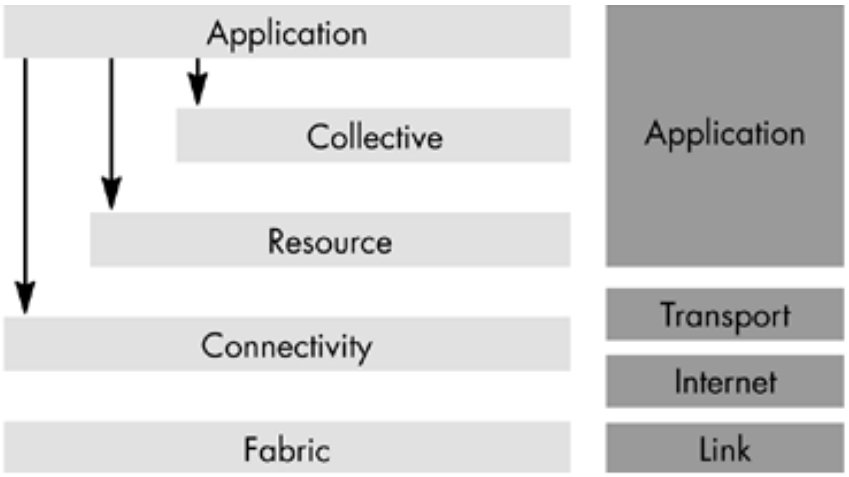

Figura 1.2: Livelli dell'architettura Grid

Il livello **Fabric** fornisce tutte le interfacce di accesso necessarie per le risorse della rete, come ad esempio computer, servizi di memorizzazione e database.

Il livello **Connectivity**, invece, definisce l'insieme dei protocolli di comunicazione e autenticazione che consentono transazioni Grid a livello di rete.
I protocolli di comunicazione garantiscono lo scambio di dati con le risorse definite nel livello Fabric, attraverso meccanismi di *naming*, *routing* e *transport*, convenzionalmente implementati attraverso il protocollo TCP/IP.
I protocolli di autenticazione sono invece fondamentali per garantire le corrette identità di risorse e utenti nello scambio dati, e devono fornire servizi di "Single Sign-on" (un utente deve loggarsi soltanto una volta per la sua sessione di lavoro, senza dover ri-effettuare il login ad ogni operazione), servizi di "Delegation" (il programma in uso dall'utente deve essere in grado di eseguire alcune operazioni automatizzate a suo nome, una volta autenticato) e "Integration" (deve essere garantita una certa interoperabilità tra protocolli di sicurezza locali, in uso dall'u-

tente, e di rete Grid.).

Il **Resource Layer** costruisce, sui protocolli di comunicazione e autenticazione del *Connectivity Layer*, API e SDK che si occupano del monitoraggio e del controllo delle operazioni effettuate su singole risorse computazionali.
Si divide in due protocolli principali: Information Protocol, adibito alla raccolta di informazioni delle risorse disponibili (stato, configurazione, politica di utilizzo ecc.) e Management Protocol, il cui compito è assicurarsi che tutte le politiche di accesso alla risorsa siano rispettate con quelle specificate.

Il livello **Collective** si occupa invece del coordinamento di un insieme di risorse, come quelle che si possono trovare all'interno di una *Virtual Organization*, e include servizi di Job Scheduling (assegnazione dei job pendenti agli elementi della rete), Resource Brokering (selezione della *Resource* più adatta per un determinato job) e Data Replication (distribuzione, in più copie, dei dati calcolati nei nodi).
Si noti che, a differenza del *Resource Layer*, il livello Collective permette un'elevata personalizzazione dei protocolli utilizzati, spesso portando a implementazioni fatte su misura per una specifica VO o comunità di utenti della Grid.

Infine, **Application Layer** riguarda tutte le applicazioni utente che forniscono un interfaccia di accesso alla griglia computazionale; si noti che queste applicazioni si affidano completamente ai servizi e le interfacce fornite dai livelli sottostanti per eseguire i propri compiti.

1.2 Resources

In questa sezione si approfondiranno gli elementi probabilmente più importanti di una griglia computazionale: le *Resources*.
Esse sono entità che determinano la vera potenza computazionale di una rete di Grid Computing: esse sono un insieme di entità fisiche come calcolatori, elementi di storage e software, ma anche unità distribuite nella rete, come ad esempio i DFS (**D**istributed **F**ile **S**ystem).
In altre parole, con questo termine si intende un elemento della griglia che fornisce alcune risorse (come dati, potenza di calcolo e servizi) a disposizione degli altri nodi della rete, per l'esecuzione solitamente di un *Task*.

Attualmente, esistono molte tipologie di *Resource*[3], create ognuna per rispondere a una determinata esigenza. Questo testo, tuttavia, si soffermerà soltanto sulle tipologie più comuni, che verranno elencate di seguito.

Computation Resource

La *Computation Resource* è una delle tipologie di risorsa più comuni all'interno di una rete Grid, e rappresenta un insieme di risorse di calcolo raggruppate logicamente (ad esempio per motivi di vicinanza geografica).
Dato che rappresentano la potenza di calcolo dell'intera griglia computazionale, la loro presenza è fondamentale per l'assegnazione (e lo svolgimento) dei *task* correnti.
Si noti come le CR siano presenti in tutte le implementazioni conosciute di reti grid, e pertanto comporre un quadro generale degli elementi che le compongono è un compito sostanzialmente impossibile. Pertanto, a titolo esemplificativo, si prenderà in esame come le *Computation Resources* vengono implementate nel *Middleware* "gLite", sotto il nome di "Computing Element"(CE).

Data questa breve premessa, si deve considerare come ogni CE sia costituito da

alcuni elementi fondamentali che ne determinano il funzionamento:

- un Grid Gate (GG), che rappresenta l'interfaccia di accesso del nodo, regolando le comunicazioni in entrata e in uscita tra la rete e il nodo stesso.
- un Local Resource Management System (LRMS), il cui compito è quello di assegnare i *job* arrivati tramite il *GG* agli elementi del sistema.
  Ovviamente, questo si rende particolarmente utile in caso in cui un Computing Element sia formato da una moltitudine di calcolatori.
- Una serie di Workers Nodes (WNs), con la quale si identificano i singoli calcolatori del CE, il cui compito è quello di svolgere direttamente i job loro assegnati.

Storage Resource

Le *Storage Resources* sono elementi di rete dedicati all'immagazzinamento e gestione dei dati processati dalla griglia computazionale, anche solo per periodi di tempo limitati: non è inusuale, infatti, utilizzare delle *Storage Resources* come cache per le computazioni in corso.
Come per i *Computing Resource*, ogni SR può essere composto da una moltitudine di elementi fisici, come dischi di rete o datacenter, accomunati da un nome logico. Anche in questo caso per descrivere la struttura interna tipica di un SR ci si affida alla sua implementazione secondo il *Middleware* gLite, con il nome di "Storage Element" (SE).
Ogni SE all'interno di una griglia computazionale è formato da due elementi fondamentali:

- Lo Storage Management (SRM), un *Middleware* capace di mascherare la grande eterogeneità dei sistemi di archiviazione presenti nella griglia, consentendo l'accesso ai dati archiviati in modo uniforme e garantendo servizi di trasferimento file, allocazione su disco e ridenominazione.
  Esistono varie tipologie di SRM, che variano in base alla quantità di dati fornita dallo Storage Element e alla tipologia di archiviazione: per piccoli

SE, ad esempio, l'SRM viene affiancato da un Disk Pool Manager (DPM), un gestore di archiviazione adatto a quantità di dati modeste.

– Un protocollo di trasferimento, solitamente GridFTP, che gestisce il trasferimento dati tra un Storage Element e un altro nodo della rete.
Sviluppato dall' *Open Grid Forum*, una delle community più attive in ambito Grid Computing, GridFTP consente di ottenere alcune funzionalità aggiuntive rispetto a FTP (da cui deriva) per l'invio di dati, che lo rendono particolarmente adatto ad ambienti Grid.

Communication Resource

Questa tipologia di *Resource* trova il suo miglior impiego in strutture di rete Grid particolarmente grandi, dove la comunicazione tra nodi può diventare molto complessa per essere gestita autonomamente.
In quest'ambito, le *Communication Resources* spesso fungono da "tramite" tra due nodi in comunicazione, fornendo servizi capaci di ridurre considerevolmente la latenza di rete.
Ad esempio, si consideri che le *Communication Resources* offrono vere e proprie "Routing tables" per gli altri nodi della rete, utili per minimizzare l'utilizzo di algoritmi di *Resource Discovery* (che si ricordano essere piuttosto dispendiosi in termini di tempo e di traffico generato).
Come ultima nota è bene precisare che non tutte le implementazioni di griglia computazionale prevedono l'utilizzo di *Communication Resources*: il modello centralizzato, infatti, non ne prevede l'esistenza.

1.3 Middleware

Nell'ambito del Grid computing, il Middleware è un software di supporto che ne facilita la gestione, l'utilizzo e il funzionamento.
Il Middleware si interpone infatti tra il livello Hardware (le singole risorse) e Software (le applicazioni utente), garantendo l'astrazione dei servizi offerti e facilitando la comunicazione tra dispositivi diversi.
Nel dettaglio, le principali mansioni del Middleware sono:

Sicurezza: tutte le operazioni che richiedono un certo livello di sicurezza necessitano un passaggio tramite il Middleware. A questo software, ad esempio, è assegnato il compito di gestire tutte le comunicazioni criptate della rete, che sono codificate per motivi di sicurezza.
Solitamente, inoltre, nel middleware viene confermata la validità dei certificati digitali con la quale i nodi si presentano alla rete.

Information Management: dato il suo ruolo centrale di gestore della rete Grid, il Middleware necessita di una conoscenza completa e approfondita dei nodi presenti all'intero della griglia.
Pertanto, il Middleware è appositamente equipaggiato con una serie di algoritmi e servizi che consentono di ottenere lo stato attuale delle *Resources*; tali informazioni verranno poi riutilizzate per le altre funzionalità fornite dal Middleware, come ad esempio il Job Sheduling.

Resoruce Management: potendo accedere allo stato generale della griglia, al Middleware viene assegnato il *Job Scheduling* della rete, che ricordiamo essere fondamentale per l'assegnazione dei job correnti.

Illustrate le caratteristiche generali di un software Middleware, di seguito verranno presentate due varianti di questo software molto comuni: gLite e Globus.

1.3.1 gLite

gLite è stato il principale middleware per la rete europea di grid computing EGEE, per un periodo compreso tra il 2006 e il 2013.
Sviluppato dalla collaborazione di più di 80 persone provenienti da centri di ricerca europei, gLite è stato utilizzato con successo da più di 15000 ricercatori in svariati ambiti scientifici.
Le principali componenti da cui è formato sono:

User Interface: punto di accesso alla griglia gestita da gLite, prevede un'autenticazione tramite account certificato.
La UI permette all'utente verificato di consultare la lista di risorse disponibili, di sottomettere i job o di cancellarli, oltre che di vedere lo stato attuale degli stessi.

Computing Element e Storage Element: componenti che sono stati già introdotti nelle sezioni precedenti, il Computing Element contiene il Grid Gate (qui chiamato *GateKeeper*), il Local Resource Management System e i Worker Nodes. Lo Storage Element, come già specificato, fornisce un accesso uniforme alle risorse di data storage. Al suo interno contiene un Storage Resource Manager, che viene utilizzato per gestire la grande eterogeneità dei sistemi di archiviazione presenti nella griglia.

Information Service: questo componente fornisce informazioni sulle risorse del Grid e sul loro stato. Queste sono essenziali per il funzionamento dell'intera rete in quanto è attraverso l'IS che le risorse sono individuate. Le informazioni pubblicate sono anche usate per scopi di monitoraggio e accounting.

Workload Management System: infine, questa componente ha il compito di accettare i job immessi dagli utenti ed assegnarli ai vari Computing Element, per memorizzare il loro stato e richiamare il loro output.

Si tenga presente che gLite adotta il linguaggio JDL (**J**ob **D**escription **L**anguage)[4]

per descrivere i job da assegnare al WMS.
Esso è un linguaggio molto simile allo standard JSDL (che vedremo a breve), caratterizzato da una forte estensibilità, tanto che è possibile per l'utente finale aggiungere ogni attributo che ritiene necessario, senza compromettere il corretto funzionamento del job descritto.

Lo scheduling dei job (chiamato "match-making" in gLite) è un altra funzione del WMS, che può adottare diverse politiche in base alle necessità attuali.
Ad esempio, qualora la latenza di esecuzione del job sia un fattore importante da minimizzare, verrà adottata la politica detta "Eager scheduling", che assegna il job corrente alla prima risorsa disponibile in rete.
Se nessuna risorsa verrà trovata disponibile, sarà effettuato un *polling* per soddisfare nel più breve tempo possibile la richiesta.
Il "Lazy scheduling", invece, si affida alle notifiche di disponibilità inviate dalle Resources per l'esecuzione di una richiesta, ed è pertanto una politica che fa della latenza un fattore non determinante.

1.3.2 Globus Toolkit

Globus Toolkit (GTK) è un software middleware sviluppato dalla "Globus Alliance", associazione che riunisce vari enti universitari e di ricerca nel mondo per la creazione e la gestione di sistemi Grid.
Nato con l'intento di creare un middleware in grado di risolvere i problemi reali che si affrontano nell'installazione e gestione di una rete grid, Globus Toolkit si basa principalmente su standard internazionali per il suo funzionamento, come ad esempio lo standard SOAP definito dalla W3C.
La sua distribuzione open source, inoltre, gli ha permesso di riscuotere una certa popolarità in ambito scientifico.
Globus Toolkit è composto da quattro componenti principali:

Globus Resource Management: GRM è il gestore delle risorse della rete Glo-

bus, strutturato su un architettura a livelli: al livello più alto risiedono tutti i servizi per l'allocazione di insiemi di risorse, mentre al più basso vi sono i servizi di allocazione della singola risorsa.
Un esempio è sicuramente il Globus Resource Allocation Manager (GRAM), che corrisponde al già citato Grid Gate di un CE.

Grid Security Infrastructure: questo componente si occupa di fornire servizi di sicurezza ai membri della rete, come servizi di crittografia, autenticazione ecc.
Il GSI viene invocato, ad esempio, ogniqualvolta due entità vogliano comunicare tra loro, verificando l'autenticazione di ognuno attraverso lo standard dei certificati X.509.

Grid Information Service: infine, GIS è il componente di Globus che si incarica di raggruppare le informazioni di stato delle varie risorse.
Suddiviso a sua volta in tre componenti principali, GIS ottiene gli stati delle risorse attraverso degli invii periodici effettuati dalle risorse stesse della Grid.
Il destinatario di tali informazioni è un insieme di Web Server denominati Grid Index Information Service (GIIS), generalmente posti in una struttura a più livelli per questioni di scalabilità.

Globus può essere affiancato da un insieme di altre applicazioni di utilità, come ad esempio servizi di data management, *Resource discovery* e monitoring, sempre basati su interfacce standard.
Si tenga conto, inoltre, che questo middleware fornisce anche un estesa API per lo sviluppi di applicativi personalizzati in Java e C++.

1.3.3 Job Submission Description Language

JSDL[5] è una variante XML utilizzata per la specifica di *job* da assegnare ai worker nodes della griglia.
Introdotto nel 2005 dall'OGF (**O**pen **G**rid **F**orum), JSDL è attualmente utilizzato nella maggior parte dei middleware presenti sul mercato, come Unicore e gLite.
La sua distribuzione nelle varie architetture, infatti, permetterebbe di fare coesistere vari sistemi di grid computing attraverso un solo linguaggio, abbandonando tutti i vecchi linguaggi di job scheduling proprietari.
Qui di seguito viene mostrata la struttura tipica di un job descritto in JSDL:

```
<JobDefinition>
    <JobDescription>
        <JobIdentification.../>?
        <Application.../>?
        <Resources.../>?
        <DataStaging.../>*
    </JobDescription>
</JobDefinition>
```

Il primo tag presente, *JobDefinition*, è quello di apertura tipico di JSDL. Può contenere il parametro opzionale *id*, utile per assegnare a tale job un identificativo.

JobDescription invece funge da contenitore per quelli che sono gli elementi più comuni nella descrizione di un job, come *Application*, *Resources* ecc., che vedremo di seguito.

JobIdentification contiene alcuni elementi opzionali come ad esempio *JobName* e *JobDescription*. Questi sono utili per inserire informazioni aggiuntive in formato stringa.

Application è il tag che descrive l'applicazione a cui il job verrà assegnato, tramite il suo nome e versione (rispettivamente *ApplicationName* e *ApplicationVersion*).

Si noti che JSDL permette l'implementazione di estensioni atte ad inserire nuove funzionalità. In questo caso è utile segnalare l'estensione POSIX utilizzabile nel suddetto tag, che permette l'assegnazione di applicazioni in ambienti UNIX.

Il tag *Resources* descrive le risorse impiegate nello svolgimento del job, specificando ad esempio su quali host deve essere svolto, l'indirizzo di archiviazione temporaneo dei dati calcolati e il tempo massimo disponibile per l'esecuzione.
In sostanza, si tratta di una parte fondamentale della descrizione di un job, in quanto ne specifica gli elementi essenziali e indica come eseguirli.

Infine, *DataStaging* fornisce alcune informazioni sull'invio dei dati calcolati, ad esempio specificando l'indirizzo del destinatario e come tali dati debbano essere trattati ad invio terminato ecc.

1.4 Grid e Cloud a confronto

È lecito domandarsi se è ancora utile soffermarsi sullo studio di sistemi Grid nei tempi moderni, sopratutto con il crescente successo del paradigma Cloud in internet.
Per poter rispondere adeguatamente a questa domanda, è bene esaminare la definizione [6] di Cloud Computing:

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction."

Il Cloud computing si pone, in sostanza, come fornitore di tecnologia di informazione e comunicazione (ICT) attraverso dei servizi pubblici, e deve gran parte del suo successo alla sua elevata efficienza.
La sua fornitura di servizi attraverso un uso intensivo (e preciso) della virtualizzazione permette infatti di creare services mirati, con un allocazione di risorse adeguata alle esigenze richieste.
Inoltre, dato lo sfruttamento di server dedicati, le performance ottenute dall'utilizzo di questi sistemi sono particolarmente buone, e generalmente migliori di una rete Grid con un pari numero di nodi.
Tuttavia, questo paradigma manca di alcuni aspetti fondamentali per potersi sostituire completamente al Grid Computing in ambito HTC (**H**igh **T**roughput **C**omputing):

Affidabilità: sebbene un assioma del cloud stabilisca che la capacità delle risorse dei provider siano infinite, bisogna riconoscere che in ambito Grid la richiesta di potenza computazionale può diventare molto elevata.
Affidare completamente il calcolo dei job immessi nella rete ai servizi cloud potrebbe mettere in difficoltà il provider in caso di un carico di lavoro eccessivo: che succederebbe, infatti, se non fossero disponibili abbastanza server per i job asse-

gnati?
Il tal caso, il provider dovrebbe provvedere all'installazione (o virtualizzazione) di nuovi datacenter, con conseguente perdita di tempo; sebbene questa problematica sia presente anche in ambito Grid (se una *Request* non può essere soddisfatta data la mancanza di *Resources* idonee, essa rimane pendente), la sua risoluzione è affidata alla *VO* della Grid, e non a un gestore esterno con le proprie politiche in fatto di assistenza, tempi di risoluzione e necessità.

Sicurezza: L'architettura tipica del cloud computing prevede uno o più server reali fisicamente collocati presso il data center del fornitore del servizio, ed accessibili tramite interfacce create appositamente.
Premesso che questa architettura offre alcuni vantaggi (ad esempio le prestazioni fornite, che si ricordano essere ottime), la rete così formata è esposta ad inevitabili problemi di sicurezza, dato che vi è un singolo punto di accesso per i servizi, che quindi la rendono facilmente attaccabile da terzi.
Sebbene esistano contromisure efficaci, infatti, alcuni attacchi (ad esempio, del tipo DDoS) sono *mitigabili*, e non schermabili: è importante capire quindi come il Cloud possa al massimo limitare i danni di un attacco, ma non esserne del tutto immune.
Fortunatamente, lo stesso non si può dire del Grid computing che adotta modelli di implementazione del tipo *Distribuito* e ad *Agenti*: essendo la rete composta da N *Resources* debolmente distribuite, un attacco capace di paralizzare la rete dovrebbe effettuare almeno N DDoS, sebbene indubbiamente di entità minori.

Un altro aspetto che è bene approfondire è la completa allocazione delle risorse ai servizi Cloud: dato che in ambito HTC vi è spesso necessità di salvare in memoria i dati elaborati, è triviale notare come essi risiederanno completamente presso il gestore del servizio.
Non si avrà, quindi, una completa gestione dei dati mantenuti nella rete, dato che sarà discrezione del provider cancellarli ad elaborazione terminata.

Sostenibilità: l'ambito Grid si è reso famoso anche grazie all'apporto del cosiddetto "CPU Scavenging".
Il *CPU Scavenging* (anche chiamato "Cycle Scavenging") è un meccanismo che sfrutta i cicli di clock del processore che non vengono utilizzati pienamente: un esempio può essere l'attesa da parte del sistema di un input utente, che solitamente si risolve in un sotto-utilizzo notevole del processore.
Dallo sfruttamento di questo meccanismo nacque il "Volounteer Computing", una variante del Grid Computing che prevede l'utilizzo di tali cicli del processore in ambito *consumer* tramite internet; tra i Middleware più famosi che implementano questo meccanismo, si ricorda BOINC (**B**erkeley **O**pen **I**nfrastructure for **N**etwork **C**omputing).

Considerando che l'intera rete BOINC nel 2015 ha raggiunto picchi di 139 Biliardi di operazioni in virgola mobile al secondo (mentre l'attuale supercomputer più potente al mondo, il *Tianhe-2*, ha registrato picchi di 33 Biliardi) si può capire come i pc consumer nel mondo possano offrire una valida piattaforma di calcolo e fungere da *Resources* di una rete Grid.
Spostare le operazioni HTC completamente nel Cloud significherebbe perdere questa enorme potenza di calcolo, che dovrebbe essere sostituita con l'aggiunta di datacenter sempre più potenti da parte del provider, con conseguente dispendio di risorse.

Capitolo 2

Protocolli di Resource Discovery

Il capitolo precedente ci ha mostrato come le reti Grid siano un complesso insieme di elementi, ognuno con uno specifico compito per il raggiungimento di un obiettivo comune.
L'utilizzo di griglie computazionali con un elevato numero di nodi ha portato, negli anni, alla necessità di ottimizzare le performance generali della griglia, per ridurre al minimo il tempo impiegato per l'assegnazione (e l'esecuzione) di un job.
A tal proposito, è necessario notare come le tecniche di *Resource Discovery* abbiano un costo computazionale tipicamente molto alto, che inevitabilmente si ripercuote sulla latenza generale del sistema; per questo motivo, l'individuazione di una tecnica di *Resource Discovery* ottimale è diventata oggetto di studio per molti ricercatori del mondo.
Sarebbe un errore, tuttavia, sottintendere che le performance siano l'unica problematica da risolvere per questi algoritmi: si vedrà, infatti, che alcune implementazioni promettono un costo computazionale notevolmente ridotto rispetto ad altre, ma espongono l'intero sistema a numerose vulnerabilità.

Questo capitolo avrà pertanto l'obiettivo di introdurre i principali algoritmi di *Resource Discovery* in uso nel mondo, valutando i pro e i contro di ogni implementazione; a fine capitolo, sarà presente una tabella riassuntiva di ogni modello presentato.

2.1 Modello Centralizzato

Il modello centralizzato è uno dei più vecchi modelli di allocazione delle risorse, e consiste nell'affidare staticamente a un singolo nodo la gestione di un intera rete. Nell'ambito delle reti di Grid Computing, tale modello consiste nell'affidare il ruolo di Resource manager a un singolo calcolatore, in modo che ogni worker node possa fare riferimento ad esso per l'assegnazione dei *jobs* della griglia.
Solitamente questi nodi vengono implementati tramite l'installazione di un web service Sebbene questa soluzione possa sembrare ragionevole e semplice da implementare, soffre di numerosi problemi strutturali:

Single-point-failure: la centralizzazione del sistema in un singolo nodo lo rende largamente dipendente da esso, esponendo l'architettura al rischio di essere inutilizzabile qualora si verifichi un errore.
Sebbene il problema sia mitigabile dall'utilizzo di sistemi di backup, il loro uso comporterebbe un costo di mantenimento non trascurabile, sopratutto qualora la rete risulti molto estesa.

Scalabilità: in caso di aumento di dimensioni della griglia, questa struttura non sarebbe scalabile, dato ci si affiderebbe sempre a un singolo nodo per poter gestire tutto lo scheduling della rete.
Pertanto, l'infrastruttura di accesso allo scheduler e la sua velocità computazionale potrebbero fungere da collo di bottiglia, con evidenti conseguenze sulle performance generali.

Nonostante questa serie di problemi architetturali, il modello centralizzato ebbe un certo campo di applicazione nelle griglie di calcolo distribuito: le prime versioni di UNICORE, l'architettura MDS2 e GMD ne sono un esempio.

2.1.1 Grid Market Directory

In questa sezione approfondiremo il modello GMD [7], un architettura per reti Grid Computing presentata nel 2006 da Jia Yu et al.
GMD è innanzitutto un architettura a modello centralizzato, che comunica con i worker nodes attraverso un singolo Web Server, oltre che basarsi su un singolo database (GSI) per l'archiviazione delle risorse.
Nella seguente figura è illustrato lo schema architetturale di GMD.

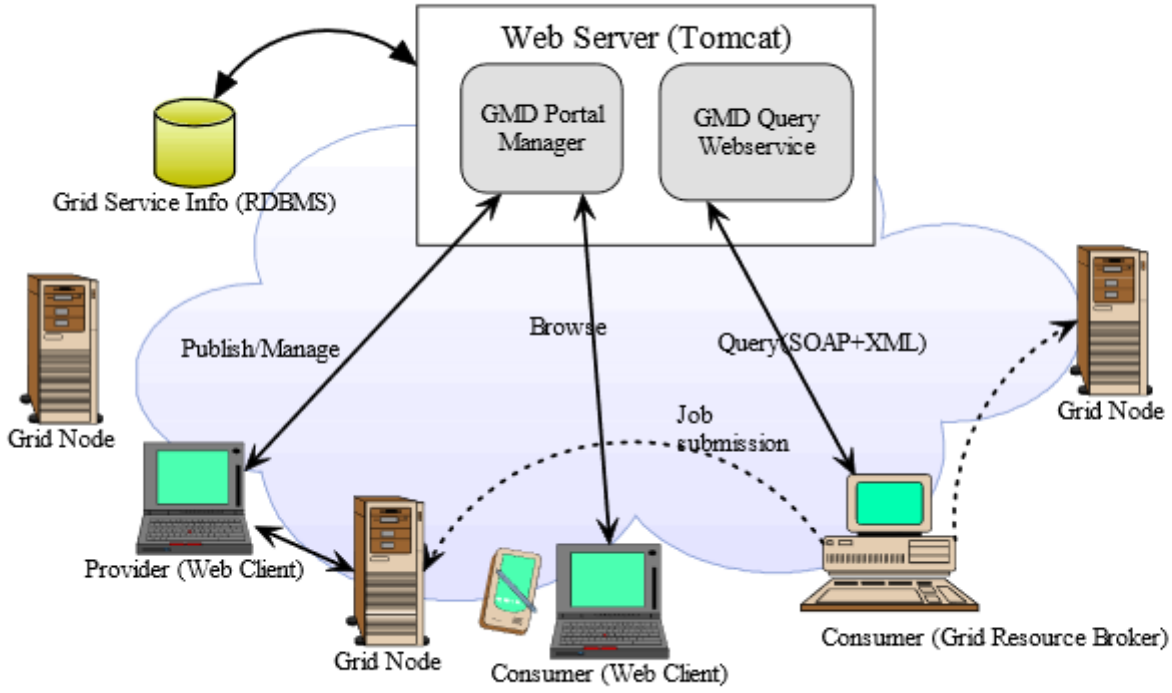

Figura 2.1: Schema di funzionamento di GMD

Come si può evincere dalla figura, di primaria importanza nel sistema è il Web Server, sul quale dipende il funzionamento dell'intera griglia; al suo interno, possiamo trovare due componenti fondamentali dell'architettura: GPM e GQWS.

GPM (GMD Portal Manager) è l'interfaccia di controllo con la quale i client possono gestire le funzionalità offerte da GMD.
Tale interfaccia comprende un accesso tramite autenticazione username/password,

un form di controllo dei servizi attualmente presenti divisi per area applicativa e altre funzionalità amministrative.

GQWS (GMD Query Web Service) è il modulo di GMD che si occupa dello scheduling a livello più alto: tramite questa interfaccia è possibile interrogare il sistema per individuare, ad esempio, il servizio migliore per l'esecuzione di un dato job, il costo stimato ecc.
La comunicazione tra client e GQWS, tuttavia, può avvenire solo attraverso una precisa codifica dei messaggi in XML, come illustrato di seguito:

```
<query_service>
   <service_type>CPU-SERVICE</service_type>
   <provider_name>HOST105</provider_name>
</query_service>
```

Nell'esempio sopra riportato, il client sta richiedendo informazioni circa un certo provider (cioè un nodo) presente nel servizio 'CPU-SERVICE'.
Il server, alla ricezione del suddetto messaggio, farà un operazione di parsing tramite il modulo 'Query Processor', il cui compito è proprio quello di interpretare le richieste dei client, e interrogherà il database per ottenere i dati da inviare in risposta.

Da qui, è triviale notare la forte dipendenza che tutti gli utilizzatori della griglia hanno nei confronti del web service: da esso deriva la conoscenza di altri nodi, l'archiviazione dei dati elaborati e molto altro ancora, evidenziando il fattore single-point-failure tipico di questa architettura.
Inoltre, la presenza di un singolo database a cui fare riferimento semplifica sicuramente le query necessarie per l'ottenimento dei dati, ma implica anche un aggiornamento continuo degli stessi ad ogni interazione della griglia; questo fattore potrebbe portare a problemi di scalabilità in caso di griglie molto estese.

2.2 Modello Gerarchico

Il modello Centralizzato presentato nella sezione precedente ha dimostrato come l'insorgenza di single-point-failure e bottlenecks possa compromettere il normale funzionamento della rete.
Nel tentativo di risolvere questi problemi, nei primi anni 2000 nacque il modello Gerarchico, presentato come la naturale evoluzione del precedente: attraverso un indicizzazione delle *Resources* secondo una precisa gerarchia, questo modello prometteva di limitare notevolmente l'insorgenza di bottlenecks all'interno della rete. Il meccanismo che lo regolava era piuttosto semplice: attraverso una distribuzione equa delle risorse sotto le dipendenze di alcuni nodi di grado "superiore", le *Resource* della rete risultavano meno dipendenti nei confronti di un singolo punto, come invece avveniva nel modello centralizzato; la creazione di tali strutture, inoltre, permetteva la creazione di piccole VO locali, che riducevano la propagazione delle query di ricerca dei nodi della rete.

In termini di prestazioni, l'adozione di questo modello comporta numerosi vantaggi, tra le quali si ricorda una minore esposizione a problemi di single-point-failure e la riduzione di bottlenecks.
Inoltre, questa architettura garantisce anche un buona latenza (anche se non pari al livello della centralizzata) in quanto la dimensione dei gruppi locali è prefissata, e non può superare limiti che degraderebbero sicuramente le performance generali. Tuttavia, questo modello non riesce a risolvere la presenza di single-point-failure all'interno della rete, sebbene alcune varianti riescano a limitare in gran parte questa vulnerabilità; per giunta, le implementazioni tipiche di questa architettura sono molto complesse, e richiedono una precisa installazione dei suoi componenti.

Date queste premesse, approfondiamo una delle varianti di questo modello più famose: l'indicizzazione a tre livelli, presentata nella prossima sezione.

2.2.1 Indicizzazione a tre livelli

Nel 2007, Yulan Yi et al. proposero un modello gerarchico [8] per reti grid suddiviso in livelli.

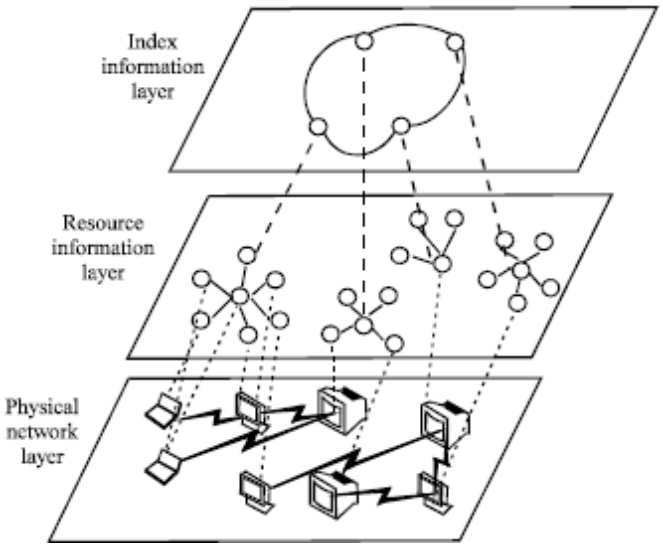

Figura 2.2: Schema di funzionamento dell'indicizzazione a tre livelli

Il primo livello (Physical Network Layer) è composto dalle resources stesse, interconnesse tra di loro tramite un collegamento di rete.

Il secondo livello (Resource Information Layer) è composto da nodi logici riferiti direttamente ai nodi del primo livello, con una cardinalità n-1; tali nodi contengono alcune informazioni come ad esempio indirizzo ip e stato delle *Resources*.

Il terzo livello, infine, è composto dai soli supernodi, raggruppati in una architettura ad anello che viene ciclata nel caso di *Resource Discovery*.

Grazie a questo livello di astrazione, è possibile creare delle piccole Virtual Organization amministrate da un supernodo; si noti che la dimensione massima di tali VO è definita staticamente in modo da assicurare alla griglia una certa scala-

bilità (il raggiungimento della dimensione massima ne comporterebbe una nuova creazione).

Oltre ad un intelligente organizzazione di rete, questa variante implementa un meccanismo di *Resource Discovery* molto efficiente e diviso in 4 fasi distinte, che verranno illustrate di seguito:

Submission phase: La fase di submission consiste nella ricerca locale di una certa tipologia di nodo, che d'ora in poi chiameremo 'type_a'. Se un nodo di tipologia 'type_a' non è presente nella Virtual Organization locale, allora l'utente invia la query all' Index Information layer.

Sorting phase: L'arrivo della query nell'Index Information layer comporta l'ordinamento dei supernodi attraverso un indicatore w, calcolato come segue:

$$w = \begin{cases} p_1 \times TotalCnt + (1 - p_1) \times MaxCnt) \\ \qquad\qquad\qquad\qquad T/F = F \\ p_2 \times TotalCnt + (1 - p_2) \times MaxCnt) \\ \qquad\qquad\qquad\qquad T/F = T \end{cases}$$

Questo valore indica la presenza di nodi del tipo richiesto (che in questo esempio sono 'type_a') in un certo supernodo: maggiore sarà il valore di w, maggiore sarà la presenza dei nodi richiesti nella sua VO.
La formula per il calcolo di w presenta alcuni parametri fondamentali: $TotalCnt$ è il numero di nodi del tipo 'type_a' trovati nella VO, $MaxCnt$ è il numero totale di nodi che invece il supernodo può contenere.
p1 e *p2*, invece, sono dei valori empirici di "aggiustamento" inseribili dall'utente.

Searching phase: Una volta ottenuta una lista ammissibile di supernodi, essi vengono scansionati sequenzialmente per poter trovare dei nodi adatti come Resources.

In altre parole, questa fase si occupa trovare la migliore *Resource* che possa soddisfare la richiesta corrente, attraverso un'analisi del suo stato e la sua disponibilità in termini di risorse fornite (ad esempio, quanto tempo mette a disposizione per la computazione).

Locating phase: Questa fase, infine, si occupa di fornire l'utente delle informazioni necessarie per localizzare la *Resource* individuata nella fase *Searching*, solitamente tramite la fornitura del suo indirizzo IP.

Le prestazioni di questa forma di *Resource Discovery* sono ben documentate [8], e mostrano come questa architettura riesca ad avere dei tempi di latenza significativamente più bassi rispetto alle metodologie Exhaustive e Lumped.
Inoltre, bisogna considerare come questa architettura sia altamente scalabile per le dimensioni della griglia: l'introduzione di supernodi che fungono da VO locali, infatti, permette di circoscrivere le query di ricerca localmente, evitando nella maggior parte dei casi di propagarle nel resto della rete.
Tuttavia, bisogna considerare come questa struttura soffra di problemi di sicurezza: nella sua forma proposta, infatti, non viene inclusa nessun tipo di autenticazione in fase di *Resource Discovery*.
La sua complessità infine è piuttosto elevata, fattore che comporta un alto costo di mantenimento della rete: oltre alle difficoltà di installazione, è necessario anche costantemente assicurarsi che i supernodi siano connessi alla griglia, in quanto non è previsto nessun algoritmo di sostituzione dinamica del supernodo.

2.3 Modello Distribuito

I capitoli precedenti hanno mostrato come i modelli centralizzati e gerarchici non riescano ad agire con efficienza in caso di griglie molto estese e dinamiche, dati i loro noti problemi di bottlenecks e single-point-failure.
Per ovviare a questi difetti si è cercato di implementare nelle griglie computazionali il modello distribuito, consistente in un insieme dinamico di nodi equivalenti connessi via internet.
In questo modello, infatti, i peer appartenenti alla griglia condividono un insieme di informazioni e servizi senza gerarchia di sorta, fattore che comporta numerosi vantaggi:

Dinamicità: caratteristica dei sistemi p2p è la grande dinamicità della rete che formano, in quanto i nodi possono connettersi e disconnettersi dalla griglia in qualsiasi momento, continuando comunque a garantire il corretto funzionamento del sistema.
Lo stesso non si può dire dei precedenti modelli: la disconnessione di un supernodo o del web server nel GMD paralizzerebbe, anche temporaneamente, la griglia computazionale.

Scalabilità: particolarmente vero nei sistemi distribuiti strutturati, i sistemi distribuiti offrono una grande scalabilità in termini di dimensioni della griglia.
Grazie alla decentralizzazione dell'architettura, infatti, è possibile ridurre in maniera significativa il rischio di bottlenecks, distribuendo le query di ricerca in modo uniforme tra i vari peers.

Prestazioni: generalmente i sistemi p2p offrono delle ottime prestazioni per quanto riguarda il *Resource Discovery*, utilizzando vari algoritmi di ricerca ottimizzati come DHT e Chord. Quest'ultimo verrà opportunamente approfondito successivamente.

Purtroppo, è difficile delineare una struttura generale di Griglia computazionale a

modello distribuito, dato che nel corso del tempo il suddetto modello ha ricevuto numerose implementazioni molto diverse tra loro; per semplicità, in questo testo, tali implementazioni verranno suddivise in quattro categorie principali: Classico, Strutturato, Ibrido e Super to Peer.

Classico

Con modello distribuito classico, nelle reti grid, si intende una architettura di rete p2p non strutturata, in cui ogni nodo(chiamato anche peer) è connesso in modo casuale a un numero fisso di altri nodi.
Essendo la loro scelta del tutto casuale, la struttura non risente in alcun modo dell'eventuale connessione e disconnessione di nuovi peers, e non vi sono, pertanto, problemi di single-point-failure.
Si noti che, data mancanza di un entità di livello più alto capace di amministrare tutti i nodi della griglia, la ricerca di una Resource è affidata al cosiddetto *query flooding*, cioè alla propagazione ricorsiva di una query da peer a peer fino al raggiungimento della risorsa.
Questa dinamica, sebbene sia efficace per reti Grid medio/piccole, mal si adatta all'utilizzo in griglie di grandi dimensioni, perché tante query simultanee di ricerca potrebbero saturare rapidamente la rete e renderla inutilizzabile.
Inoltre, dato che ogni query generata dal meccanismo di query flooding ha un numero massimo di "salti" (TTL, Time To Live) definito, questa architettura potrebbe generare dei risultati falsi-positivi, cioè non arrivare a destinazione sebbene la Resource cercata esista nella rete.

Questa variante presenta comunque alcune famose implementazioni, come LARD (Learning Automata-based Resource Discovery), che utilizza un meccanismo ad agenti per identificare le *Resources* richieste. Si tenga nota che si è deciso di menzionare LARD nel modello distribuito (e non nel capitolo successivo, dedicato appositamente ai sistemi di questo tipo) perché gli agenti di questa variante hanno il solo compito di trovare, in modo del tutto automatico, nuovi *path* per una certa richiesta.

Non hanno, quindi, una vera e propria autonomia come i modelli che verranno presentati nel capitolo successivo.

Strutturato

Le griglie computazionali che implementano questa variante si basano sulla distribuzione in tutti i nodi di strutture dati indicizzate che "mappano" l'organizzazione della rete.
Tale meccanismo viene chiamato DHT (**D**istribuited **H**ash **T**able) che, nelle sue varie implementazioni, si è dimostrato piuttosto efficiente nella ricerca di Resources all'interno di una rete Grid, avendo un costo computazionale per ogni query pari a $O(logN)$, dove N è il numero di nodi.
Inoltre, data la distribuzione omogenea della struttura dati in tutta la rete, questa architettura non incorre in potenziali bottlenecks nella ricerca di *Resource*, dato che il carico di lavoro è distribuito equamente in tutti i nodi.
Tuttavia, questa variante necessita di un costante mantenimento delle hash table presenti all'interno di ogni nodo, il che genera molto traffico di rete per ogni aggiornamento.
Inoltre, questa variante non è adatta per griglie di calcolo distribuito con una forte dinamicità delle *Resource* archiviate, in quanto provocherebbero un continuo cambiamento delle Hash Key presenti nelle Table, e quindi produzione di traffico di rete.
Fra le sue applicazioni più note ricordiamo CAN (**C**ontent **A**ddressable **N**etwork), ACO (**A**nt **C**olony **O**ptimization) e Chord; quest'ultimo verrà analizzato nelle sezioni seguenti.
Per ulteriori approfondimenti su questa tipologia di *Resource Discovery*, si rimanda al riferimento [9].

Super to Peer

La tipologia Super to Peer è un nuovo approccio al modello distribuito che migliora l'integrazione tra griglie computazionali e sistemi p2p.
Questo approccio infatti prevede la creazione di super nodi all'interno della rete,

in modo da formare per ognuno di essi una Virtual Organization.
All'interno di una VO, un super nodo funge da server per tutti i peer client a lui connessi, oltre che essere responsabile per alcuni compiti come il routing o la ricerca di altri peers.
Si viene così a creare una rete formata da una serie di VO amministrate da dei singoli super peer, una struttura di rete del tutto simile alla struttura gerarchica descritta nei capitoli precedenti.
Tuttavia, bisogna considerare come questa variante permetta di integrare la scalabilità tipica dei sistemi distribuiti all'interno di un architettura gerarchica, permettendo quindi di adottarla in griglie computazionali molto estese.
L'efficienza è generalmente buona: si è stimato [10] che tutte le implementazioni di questa variante abbiano una complessità di messaggio pari a $O(S^2)$ e una latenza di $O(S)$, dove S è il numero di super nodi nella rete.
L'architettura comunque soffre di problemi di bottlenecks in caso di un vasto numero di query broadcast (in quanto si potrebbero propagare per l'intera rete, generando molto traffico), ed inoltre alcune implementazioni possono essere esposte al rischio di single-point-failure nell'evenienza in cui un super peer si sconnetta.
Tra le implementazioni degne di nota, ricordiamo KaZaA, Gnutella2 e alcuni modelli proposti da Mastroianni et al.

Ibrido

Le varianti proposte precedentemente presentano vulnerabilità più o meno gravi che possono compromettere la giusta funzionalità della rete Grid.
Per ovviare a questi problemi si è cercato negli anni di trovare soluzioni alternative capaci di integrare tutti i punti di forza dei modelli precedenti, creando soluzioni ibride di grid computing distribuite.
Tuttavia, è difficile tracciare uno schema generale capace di raffigurare questa variante di modello distribuito, data la sua grande eterogeneità; questo testo, pertanto, si limiterà a presentare alcune sue implementazioni.
DirectConnect, ad esempio, è un variante che combina un architettura centralizzata con un meccanismo super to peer per il *Resource Discovery*, tuttavia presenta

problemi di bottleneck dato il suo centralismo e ha una scalabilità mediocre.
L'implementazione Moreno-Vozmediano, invece, combina un sistema di *Resource Discovery* peer to peer "classico" con un organizzazione tipica dei web clusters; sebbene offra delle buone prestazioni e sia altamente scalabile, questo meccanismo rischia di sovraccaricare troppo la rete in caso di query broadcast simultanee.

2.3.1 Implementazione Iamnitchi

Questa applicazione del modello distribuito "classico" prevede la creazione di una rete di peer senza gerarchia, basata sul query flooding per l'ottenimento delle *Resources* necessarie.
La suddetta rete si forma tramite la connessione dei peer a un nodo statico, a cui si richiede la locazione di eventuali altri peer nella rete.
La tecnica di *Resource Discovery* è formata da quattro algoritmi fondamentali:

Random Walk: consiste nella selezione casuale di un peer tra quelli a conoscenza del nodo. Sebbene non sia una logica particolarmente efficiente, è utile in casi di griglie computazionali piccole, in quanto viene evitato l'overhead derivante dagli altri algoritmi di selezione.

Learning: i nodi mantengono alcune informazioni sui peer vicini che hanno risposto positivamente a una certa tipologia di query. In questo modo, tutte le query simili verranno mandate di volta in volta al nodo più appropriato.
Qualora queste informazioni non siano presenti, si utilizza l'algoritmo Random Walk.

Best Neighbor: questo algoritmo si basa invece sulla quantità di query risolte piuttosto che sulla loro tipologia.
Pertanto, questo algoritmo tende a inviare le query ai nodi con la percentuale di risoluzione maggiore rispetto ad altri.
In modo analogo a Learning, viene mantenuto un archivio locale per poter usare

efficacemente l'algoritmo.

Learning+Best Neighbor: identico a Learning, con l'eccezione di usare Best Neighbor in caso di mancanza di informazioni.

Le prestazioni raggiunte dall'utilizzo di questo algoritmo sono generalmente buone, sopratutto grazie all'utilizzo di *Best Neighbor* per griglie particolarmente grandi. Uno schema generale delle prestazioni raggiunte dall'utilizzo degli algoritmi di *Resource Discovery* è presente nell'immagine successiva.

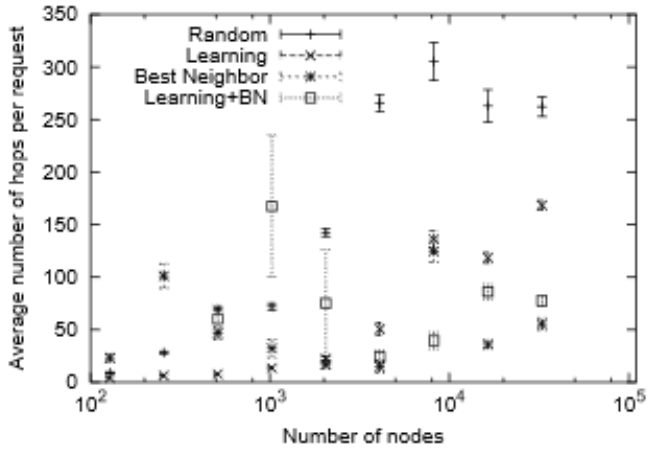

Figura 2.3: Prestazioni degli algoritmi di Resource Discovery presentati

Tuttavia, basandosi sul modello distribuito non strutturato, questa implementazione eredita tutti i problemi tipici del modello, come ad esempio la presenza di falsi negativi (implementando il TTL nelle query c'è il rischio che esse vengano scartate prematuramente) e problemi di scalabilità (il traffico generato dal query flooding può essere eccessivo in caso di richieste simultanee multiple).
Per ulteriori informazioni su questa implementazione, si rimanda a [11].

2.3.2 Chord

Chord [12] è un esempio di modello distribuito strutturato, studiato per adattarsi efficacemente ad ambienti P2P con una elevata dinamicità.
Essendo una variante di DHT, questo algoritmo permette di costruire una p2p con alcune caratteristiche:

Load Balance: Chord distribuisce uniformemente le chiavi di indicizzazione tra i nodi, create tramite una funzione di hash (SHA-1) che include anche le *Resource*.

Scalabilità: Come premesso precedentemente, questo algoritmo permette di costruire una rete di nodi con un elevata scalabilità e dinamicità, dato che le operazioni di ricerca (lookup) sono piuttosto efficienti, stimate con un costo computazionale pari a $O(logN)$.

Robustezza: l'eventuale rimozione o entrata di un nodo nella rete non comporta problemi di bottlenecks o single-point-failure.
Nello specifico, Chord utilizza il principio del "Consistent Hashing", assegnando ad ogni nodo i tutte le chiavi comprese tra i e $i-1$, permettendo la creazione di una rete logica ad anello.

Si noti inoltre che le prestazioni di lookup sono particolarmente buone perché Chord implementa anche la cosiddetta "Finger table", cioè una tabella di routing distribuita basata su intervalli di nodi nel quale una Resource è presente.
Più "lontana" è una Resource richiesta, e più ampio sarà l'intervallo di indirizzamento contenuto nel nodo, come mostrato in figura.

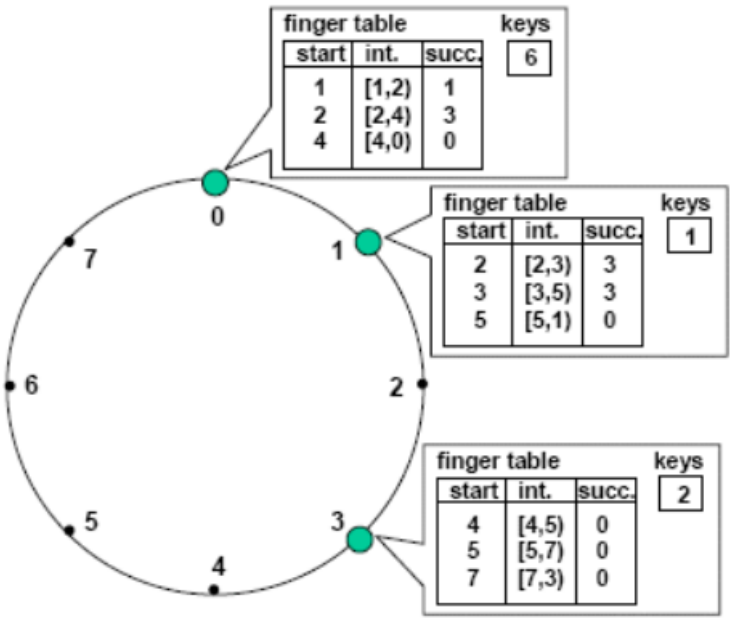

Figura 2.4: Distribuzione delle finger table nei nodi

Dato che ogni peer contiene una finger table, la disconnessione o connessione di nuovi peer deve comportare un aggiornamento delle stesse, cosa che può comportare la generazione di molto traffico di rete.
Inoltre, si noti che Chord non permette l'implementazione di una query basata su un range di Resource. Pertanto, l'eventuale necessità di un elevato numero di Resource da parte di un peer porterebbe a un costo computazionale dell'operazione pari a $O(RlogN)$, dove R è il numero di richieste.

2.4 Confronto tra modelli presentati

Dopo una breve panoramica dei possibili protocolli di *Resource Discovery* implementabili nelle reti di Grid Computing, questa sezione offrirà un riepilogo dei pro e contro di ogni modello.
La tabella sottostante permette di confrontare direttamente le funzionalità di ogni modello presentato precedentemente.

Modelli Resource Discovery	Scalabilità	Dinamicità	Presenza errori falsi-positivi	Presenza di Single-point-failures	Presenza di Bootlenecks
Centralizzato	No	Si	No	Si	Si
Gerarchico	Si	Si	No	Limitata	No
Distribuito non strutturato	No	Si	Si	No	No
Distribuito strutturato	Si	Si	No	Si	No
Distribuito Super to Peer	Limitata	Si	No	Si	No

Figura 2.5: Confronto tra i modelli presentati

Appare evidente che la soluzione centralizzata sia la più sconveniente per l'implementazione in reti grid, data sopratutto la sua vulnerabilità a problemi di single-point-failure (come abbiamo già visto nell'implementazione GMD, infatti, qualora il web server subisca una disconnessione l'intera rete non potrebbe funzionare correttamente).

La soluzione gerarchica, invece, pare presentare pochi problemi strutturali, che rendono adatta la sua implementazione in reti grid. Il rischio di single-point-failure, infatti, è limitato solamente ai nodi di livello più alto, e non è invalidante per l'intera rete ma soltanto per la VO locale.

Tuttavia, questa caratteristica è presente soltanto nell' Indicizzazione a tre livelli, mentre in altre architetture del modello gerarchico il single-point-failure è ancora un problema considerevole, capace di paralizzare l'intera rete.
Inoltre, come abbiamo visto nel esempio di Yulan Yi et al., la sua implementazione è particolarmente complessa, fattore che comporta un elevato costo di manutenzione in reti particolarmente grandi: è necessario tenere presente, infatti, che l'installazione di ogni supernodo nella rete richiede l'utilizzo di un calcolatore con elevate prestazioni ed ad alta affidabilità.

Il modello distribuito, infine, mostra alcune vulnerabilità in termini scalabilità che lo rendono poco adatto all'implementazione nei Grid Computing Networks.
L'unica eccezione pare essere il modello distribuito strutturato, che come abbiamo visto nell'implementazione Chord, ha un costo computazionale contenuto che si mantiene tale qualsiasi sia il numero di peers.
Si noti che tale variante sia stata segnalata come soggetta a problemi di single-point-failure. L'algoritmo tipico di *Resource Discovery* in questi sistemi, infatti, segue quello che è un percorso definito staticamente all'interno dei nodi (come ad esempio gli intervalli contenuti all'interno della finger table).
Sebbene esistano vari algoritmi correttivi in tal senso, la disconnessione improvvisa di un nodo comporterebbe vari problemi al routing delle query, in quanto verrebbero a mancare alcuni intervalli di indirizzamento.

Risulta evidente quindi come nessuna delle soluzioni proposte precedentemente possa definirsi decisiva per l'implementazione in reti di Grid computing.
Tuttavia, la continua ricerca di nuove soluzioni di *Resource Discovery* ha portato all'ideazione di un innovativo modello che fa ben sperare nella risoluzione delle problematiche appena esposte: si tratta del modello ad Agenti, che verrà introdotto nel prossimo capitolo.

Capitolo 3

Resource Discovery ad Agenti

Le difficoltà dimostrate nel capitolo precedente hanno portato, negli anni, all'ideazione di un modello di *Resource discovery* ad agenti, nella speranza di individuare una soluzione definitiva ai problemi di performance, affidabilità e sicurezza tipici di queste architetture.
Essendo un approccio al *Resource discovery* relativamente nuovo ed eterogeneo, è difficile individuare alcuni aspetti generali di questo modello.
Per tale motivo, il presente capitolo si occuperà nel dettaglio delle varianti che implementano il concetto MAS (**M**ulti **A**gent **S**ystem), cioè quei sistemi che utilizzano una pluralità di agenti all'interno della rete per determinati obiettivi.
La scelta di focalizzare l'attenzione sulle architetture multi-agente è dovuta principalmente alla loro maggiore flessibilità (che vedremo nel dettaglio nell' Implementazione Kang et al.) che ben si adatta alle complesse reti Grid.
A seguire, verrà presentata una breve introduzione sugli agenti e su alcune loro caratteristiche, per poi approfondire l'utilizzo di tali entità intelligenti all'interno delle reti di Grid Computing.

3.1 Agenti software

Dare una definizione di cosa sia esattamente un Agente non è un compito facile, data la grande varietà di definizioni che nel corso del tempo sono state presentate, anche in netto contrasto tra loro [13].
In linea generale, tuttavia, un agente è un sistema computazionale, costituito da un programma software ed eventualmente un supporto hardware, che:

- interagisce con l'ambiente circostante ed è reattivo agli stimoli di tale ambiente;
- è capace di prendere decisioni, e di conseguenza di agire, in modo autonomo, con il fine di raggiungere un obiettivo, chiamato generalmente "goal" (che può essere predefinito o negoziato);
- è in grado di comunicare (coordinarsi, cooperare, negoziare) con altri agenti.

Generalmente, gli agenti che operano in ambito puramente software vengono chiamati "agenti software", che è la tipologia di agenti di cui ci interesseremo in questo testo.

Essi possono essere classificati in varie tipologie, in base ai loro criteri comportamentali o al contesto applicativo in cui vengono implementati:

Agenti Mobili: hanno l'abilità di spostarsi da un host all'altro all'interno della rete, in modo da eseguire il codice ritenuto più opportuno in quella precisa locazione.
Dato che la selezione di nuovi host, l'esecuzione di codice e lo spostamento sono azioni eseguite autonomamente, l'uso di questa tipologia di agenti permette di alleggerire notevolmente il carico della rete, e apre la via a nuovi sistemi di distribuzione dati, come vedremo.

Agenti di Interfaccia: sono predisposti all'interfacciamento con l'utente e forniscono inoltre assistenza nell'esecuzione di determinati compiti.

A differenza degli agenti mobili sono tipicamente statici, e risiedono in modo permanente all'interno di una macchina fisica.

Agenti Cooperativi: questi tipi di agenti riescono a collaborare con l'ambiente circostante o con altri Agenti per il raggiungimento di un obiettivo che può essere sia comune che proprio di ogni singolo Agente.
Requisito fondamentale per il loro funzionamento è l'implementazione di un sistema di comunicazione tale da permettere lo scambio di informazioni, come ad esempio l'utilizzo del linguaggio FIPA-ACL.

In ambito Grid computing, tutte e tre le tipologie vengono utilizzate nelle varie implementazioni del modello ad agenti.
A titolo di esempio, si veda la sezione " Implementazione Kang et al." per un esempio di utilizzo degli agenti cooperativi e di interfaccia, rispettivamente i "Broker agents" e i "Trading agents".

3.1.1 Sistemi Multi-Agente

In alcuni casi è necessario costruire architetture composte da più agenti, basate su una stretta collaborazione per il raggiungimento di uno scopo comune.
Questa tipologia di sistemi viene chiamata "Sistema Multi-Agente", e si basano generalmente sui concetti di "Agente" e "Container" (un insieme degli stessi); il raggruppamento di tutti i container viene invece chiamato "Piattaforma", ed è rappresentato solitamente come una macchina virtuale in cui tutti gli agenti possono operare.
La realizzazione di queste complesse architetture richiede molto spesso l'implementazione di una gerarchia tra di agenti: nel caso in cui esista una netta distinzione tra un agente di grado più alto ("master") e uno di grado più basso ("slave"), allora l'organizzazione del sistema verrà ad "organizzazione verticale", mentre qualora un agente possa essere sia *master* che *slave* il sistema verrà detto ad "organizzazione orizzontale".

I MAS (**M**ulti **A**gent **S**ystems) differiscono molto dai sistemi ad agenti singoli in quanto devono tenere in considerazione alcuni fattori fondamentali di coordinazione:

- Ambiente: gli agenti del sistema devono prepararsi ad eventi esterni che possono interferire col normale funzionamento della rete, ed eventualmente essere in grado di armonizzarsi per la loro risoluzione.
  Un esempio di tali eventi nel grid computing può essere la improvvisa disconnessione di un nodo a causa di un errore imprevisto.
- Conoscenza: le informazioni del sistema devono essere distribuite tra gli agenti, così come le loro abilità.
- Eterogeneità: gli agenti possono essere implementati attraverso architetture e linguaggi differenti, senza alcuna omogeneità obbligatoria.
  Questo prerequisito ben sia adatta alle reti Grid, che sono eterogenee per definizione.
- Interazione: data la loro varietà di implementazione, è necessario che gli agenti siano conformi ad alcuni standard di comunicazione fondamentali come ACL, per garantire la comunicazione con tutti i nodi della rete.

Come già predetto, ACL (**A**gent **C**ommunication **L**anguage) è uno standard di comunicazione per agenti software utilizzato sopratutto in ambito MAS. Questo standard ha come scopo la creazione di una sintassi che permetta di facilitare l'interoperabilità tra agenti anche molto diversi tra loro, oltre che specificare una semantica standard per le comunicazioni.

Una delle implementazioni più famose di ACL è sicuramente FIPA-ACL, una variante del linguaggio standard introdotto dalla FIPA (**F**oundation for **I**ntelligent **P**hysical **A**gents).

```
(request
:sender (:name exampleAddress.com:8080)
:receiver (:name exampleReceiver:6600)
:ontology exampleOntology
:language FIPA-SL
:protocol fipa-request
:content
(action pickhotel@tcp://pick.com:6600
(reservation (:arrival 25/11/2000)
  (:departure 05/12/2000) ...
)))
```

Un esempio di FIPA-ACL

In questo linguaggio, un messaggio è composto da una serie di parametri, che ne stabiliscono alcune caratteristiche fondamentali come il mittente, il destinatario e l'azione che si vuole eseguire.
Sender e *Receiver*, ad esempio, indicano rispettivamente il mittente e il destinatario di un messaggio, espresso tramite un indirizzo IP; si noti come il parametro *Sender* è opzionale, in quanto è permesso per ACL inviare messaggi in forma anonima.
Ontology è un campo che indica alcuni parametri opzionali di un messaggio, utili sopratutto per modificare il contenuto semantico del campo *content*.
Quest'ultimo invece rappresenta il vero corpo del messaggio, scritto in un formato comprensibile per il ricevente: tale formato è indicato dal parametro *language*.

3.2 Modello ad Agenti

Il modello ad agenti è una recente proposta di architettura di *Resource Discovery*, presentata in varie forme come alternativa ai modelli tradizionali.
Come il nome suggerisce, questo modello fa un uso intensivo di agenti software all'interno della rete per finalità di *Resource discovery*.
Grazie alle loro ben note qualità di autonomia, infatti, gli agenti consentono di distribuire gli algoritmi di selezione delle risorse all'interno dei nodi stessi, rendendo la rete altamente dinamica.
Date queste premesse, è del tutto lecito trovare delle similitudini tra il modello ad agenti e quello distribuito, tanto da chiedersi quale sia la sostanziale differenza.
A titolo di esempio, si prenda l'architettura Chord: entrambi i modelli distribuiscono nei nodi un informazione per accedere alle varie *Resources* della rete, ma se da un lato *Chord* si limita a distribuire delle *finger tables* per indirizzare i clients, il modello ad agenti utilizza delle entità intelligenti *in loco* per trovare la migliore *Resource* disponibile, ad esempio fornendo un indicatore di qualità della stessa.
Si noti che molte implementazioni di questo modello utilizzano sistemi multi agente per le comunicazioni tra le varie entità del sistema, come introdotto nella sezione precedente.

In linea generale, il modello ad agenti presenta alcuni punti di forza rispetto ai modelli già visti:

Dinamicità: data la particolare gestione delle *Resources* tipica di questa rete (un agente software gestisce un insieme di N nodi, formando una piccola *VO*), l'eventuale connessione o disconnessione di *Resources* all'interno della rete è ben tollerata, dato che viene gestita unicamente dall'agente responsabile di quella *VO*, riducendo inoltre notevolmente il traffico immesso nella rete.

Scalabilità: la scalabilità del sistema è assicurata dall'inserimento di nuovi agenti software mano a mano che la la rete cresce di dimensione, in modo da evitare che un singolo agente sia sovraccaricato e funga quindi da collo di bottiglia.

Data inoltre l'autonomia degli agenti (e la loro adattabilità ai cambiamenti della rete) la rete richiede una minima manutenzione rispetto ad altri modelli, come ad esempio quello gerarchico.

Flessibilità: l'inserimento di agenti intelligenti all'interno del sistema permette di espandere quella che è la classica concezione di Grid computing, dato che si può introdurre un elemento di controllo intelligente all'interno delle *VO* locali.
Ne è un esempio il già citato *indicatore di qualità*, che avrà un utilizzo pratico nell'implementazione presentata da Kang et al., presente nella prossima sezione.

Tuttavia, il modello ad agenti non è esente da alcuni difetti strutturali, primo fra tutti l'alto overhead, dovuto perlopiù al tempo necessario per gli agenti di effettuare decisioni autonomamente.
Se infatti si considera che ogni agente, alla ricezione di una *Request*, debba avviare algoritmi specifici per l'individuazione di una risorsa, allora ci si accorgerà che i tempi di latenza sono piuttosto elevati.
Inoltre, questa architettura soffre di elevata complessità: l'implementazione di una sistema a modello ad agenti è un compito molto complesso, che richiede anche la conoscenza approfondita delle interazioni dei sistemi multi agente.

Per questi motivi, la ricerca nel campo del Grid computing ad agenti è tuttora molto attiva, e fino ad oggi sono state prodotte alcune architetture interessanti: tra queste, ricordiamo il lavoro svolto da Tan et Al. [14] nell'utilizzare efficacemente un sistema multi agente in ambito grid computing; una proposta di ricerca semantica di *Resources* da parte di Han and Berry [15] e infine l'implementazione Kang et al. [16] che illustreremo nella sezione successiva.

3.2.1 Implementazione Kang et al.

Questa applicazione del modello ad agenti è un valido esempio di come tali architetture siano altamente scalabili e adattabili.
Innanzitutto, è bene premettere che questa implementazione si basa sul paradigma MAS già introdotto nella sezione precedente, e che quindi è formata da un ecosistema di agenti in comunicazione tra loro.

Successivamente, gli agenti all'interno della rete vengono suddivisi in tre tipologie principali: User agent, Provider agent e Broker agent.
Un User agent agisce direttamente per conto dell'utente e fornisce un interfaccia per facilitare l'utilizzo della griglia computazionale.
Il suo compito principale, tuttavia, è quello di comunicare con i Broker per ottenere nuove Resources dalla rete.
Esse sono invece fornite dai Provider agents, che hanno dei compiti del tutto simili agli User agents (solitamente, infatti, vengono soprannominati "Trading agents").
Il Broker agent infine viene posto come strato intermedio di comunicazione tra l'utente e la risorsa ricercata, allocando la Resource ritenuta più adatta per un dato job all'interno della rete.

Tuttavia, il lettore più attento avrà notato che l'utilizzo di una gerarchia di agenti espone la rete al rischio di single-point-failure (ad esempio, se il Provider agent non fosse più raggiungibile, l'allocazione di una *Resource* sarebbe impossibile).
Per tutelare il sistema da questo problema, i Broker agent effettuano dei controlli periodici sui Provider per verificarne il loro stato; qualora un Provider agent fosse irraggiungibile, infatti, verrebbe escluso dalla lista di possibili fornitori di *Resources*.

È bene osservare, inoltre, come in questa organizzazione l'utilizzo di un singolo Broker Agent per una moltitudine di *Resource* porterebbe ad inevitabili problemi di bottlenecks.

Per ovviare a questo problema, ogni richiesta superiore a un certo limite di gestione verrà indirizzata agli altri Broker agent presenti nella rete, rendendo di fatto il sistema altamente scalabile.

Ora che gli elementi fondamentali del sistema sono stati introdotti, viene fornito un semplice esempio di come effettivamente si svolga una comunicazione tra gli agenti di questa architettura.

Comunicazione tra agenti

Uno schema riassuntivo di come la comunicazione tra gli agenti del sistema avvenga è presente nell'immagine sottostante, e verrà opportunamente approfondita a seguire.

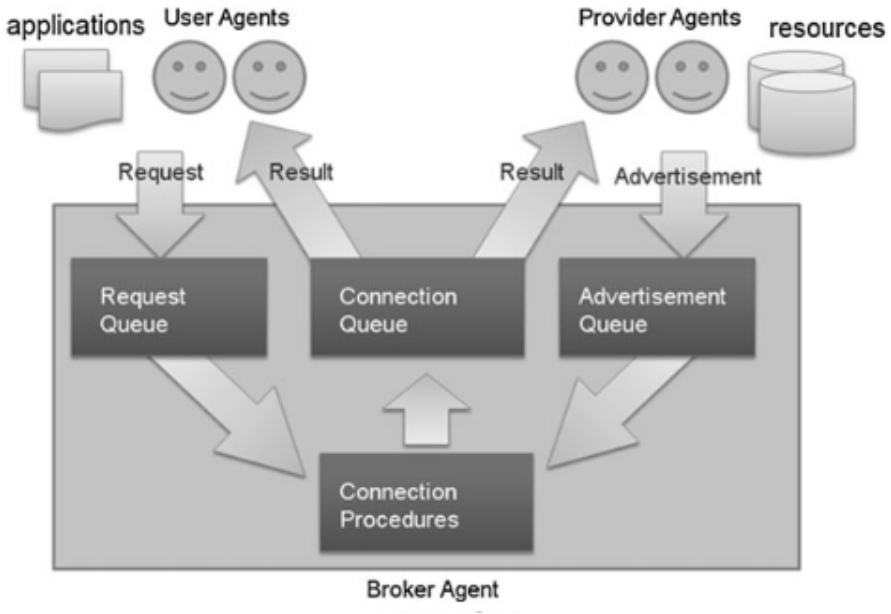

Figura 3.1: Comunicazione tra agenti nella variante Kang

Innanzitutto, un User agent invia una richiesta di Resource (solitamente chiamata "Request") al Broker agent. Come già specificato precedentemente, qualora la *Request* corrente sia oltre il limite gestibile per quell'agente, allora essa verrà indirizzata a un altro Broker agent.

All'arrivo presso un *Broker agent* con disponibilità nel processarla, la *Request* entra nella fase di "Connection", cioè una valutazione di quanto essa possa essere soddisfatta dalle *Resource* disponibili.
Questa fase si compone di 4 fasi principali, illustrate di seguito:

Selection: in questa fase, ogni risorsa di uno specifico Broker agent viene analizzata per capire se può soddisfare in maniera ottimale la *Request* corrente.
Tale analisi viene svolta su molteplici fattori, primo fra tutti l'effettiva disponibilità della *Resource*: dato che questa architettura si adatta anche a Grid molto dinamiche, una *Resource* potrebbe essersi disconnessa o essere già occupata in altri compiti.
Un altro fattore fondamentale di cui si tiene conto è lo stato della *Resource* stessa: fattori come la velocità del processore, la RAM libera rimasta e capacità del disco devono soddisfare il minimo garantito specificato nella *Request*.
Infine, viene valutato anche il tempo stimato di calcolo ("timeslot") per la *Request*, in modo da selezionare soltanto le *Resources* con un tempo minimo di elaborazione garantito.

Evaluation: una volta ottenuta una lista di risorse che soddisfano i requisiti minimi di computazione, il sistema si occupa di calcolare uno speciale indicatore di qualità della risorsa, che viene generalmente chiamato "Utility". La formula per calcolare tale indicatore è la seguente:

$$U_i = w_1 \times U_P^i + w_2 \times U_{TS}^i$$

dove i indica la connessione corrente (cioè la coppia *Request-Resource*).
U_P^i è un indicatore di "costo" della connessione, che tiene in considerazione il massimo ammissibile per l'utente e il minimo richiesto dalla risorsa, con un valore totale che oscilla tra 0 e 1 (dove più alto è il valore, più conveniente sarà la connessione).
U_{TS}^i rappresenta l'indicatore di *timeslot* della connessione, tenendo conto della disponibilità del *Provider* e dello *User agent*. Anche qui, maggiore sarà il valore e più margine di tempo sarà disponibile per l'elaborazione.

Infine, *w1* e *w2* sono dei coefficienti scelti direttamente dall'utente che specificano quanto gli indicatori di *timeslot* e di costo debbano pesare nel calcolo di *U*.

Filtering: questa fase si occupa di filtrare (e quindi scartare) ogni possibile *Resource* che non superi un certo valore di *U*.
Tale valore viene anche detto " Treshold" e viene specificato direttamente dalla *Request* in fase iniziale.
Si noti Dopo l'operazione di filtraggio, le *Resource* rimaste vengono inviate allo *User agent*, che deciderà autonomamente quale scegliere.

Recommendation: qualora le fasi precedenti non abbiano individuato nessuna *Resource* candidata (ad esempio, a causa di un valore di *treshold* troppo alto), il sistema entrerà nella fase *Recomendation*, atta ad indirizzare lo *User agent* verso altri *Broker agent*.
La selezione del *Broker agent* può avvenire tramite un algoritmo ad approccio circolare, in cui ogni provider viene interrogato in successione fino a che una *Resource* disponibile non viene individuata, oppure tramite un approccio "multicast", in cui multipli *Resource broker* vengono interrogati contemporaneamente.

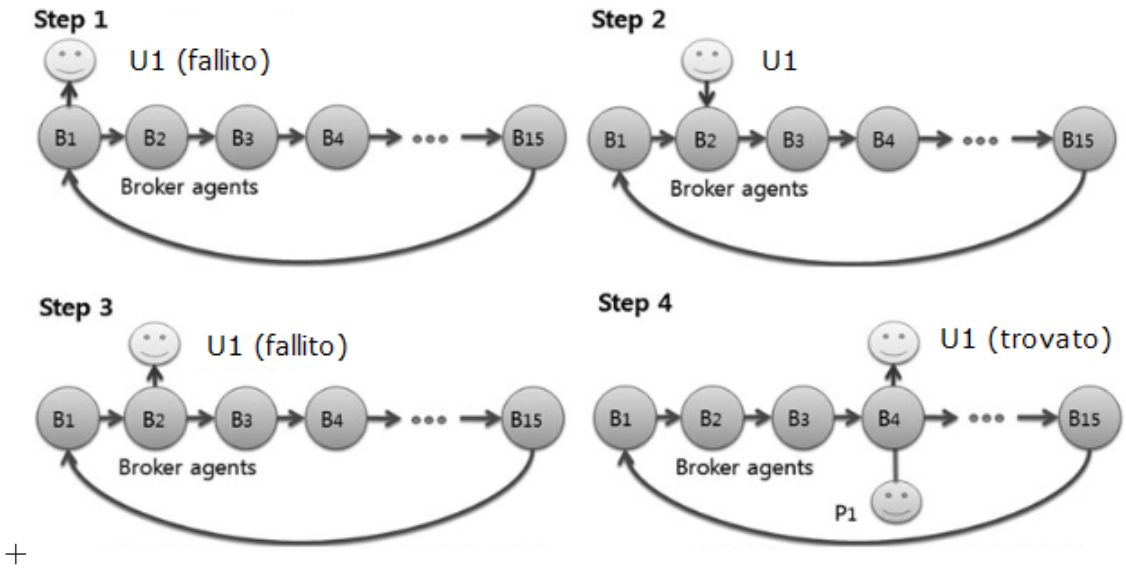

Figura 3.2: *Reccomendation* con algoritmo circolare

Le prestazioni raggiunte da questa variante sono piuttosto buone per qualsiasi dimensione della griglia: l'architettura ha infatti dimostrato di reagire bene in termini di scalabilità, il che la rende adatta all'utilizzo in reti di grandi dimensioni. Inoltre, come dimostrato nelle pagine precedenti, questa variante ha una buona tolleranza nei confronti di eventi improvvisi, quali la disconnessione di peers: il sistema, infatti, è del tutto immune a problemi di single-point failure, in quanto esistono dei meccanismi dinamici di sostituzione dei broker agents.

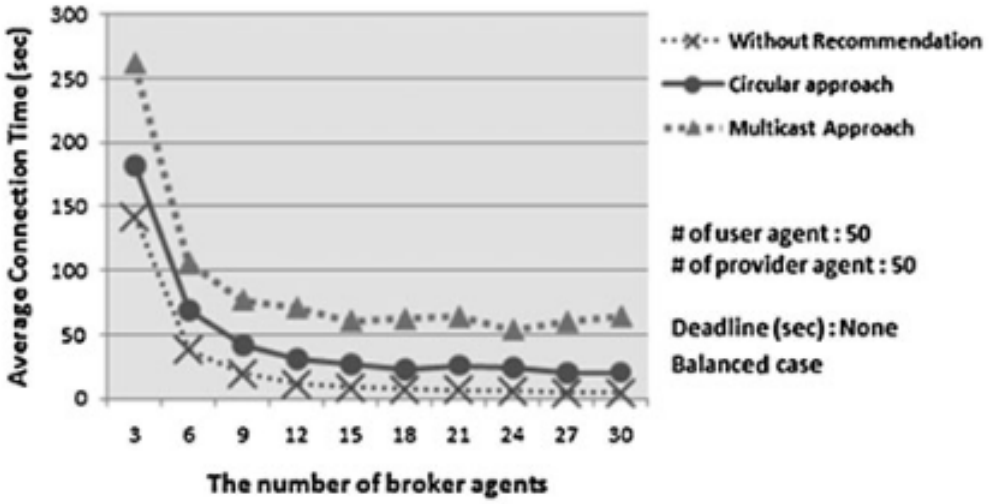

Figura 3.3: Tempi di connessione dell'implementazione

Tuttavia, anche questa variante presenta alcune significative debolezze strutturali, prima fra tutti l'alta latenza di connessione che gli agenti provocano a causa dei loro algoritmi di *Resource Discovery*.
Sebbene infatti il sistema non risenta in maniera significativa di un aumento di dimensione della griglia (le prestazioni, infatti, rimangono pressoché costanti), questa architettura ha un tempo di latenza generalmente più alto rispetto ad altri modelli già presentati.
La causa di tale latenza è sicuramente da imputare, come già premesso, ai vari algoritmi di *Resource Discovery* quali *Selection*, *Evaluation*, *Filtering* e *Reccomendation*: sebbene forniscano un indicatore di qualità chiamato *Utility*, conveniente

per fini di localizzazione di risorse, il costo computazionale necessario per calcolarlo è sicuramente non trascurabile.
La complessità di implementazione di un sistema ad agenti, inoltre, è un altro fattore da non sottovalutare: oltre a scoraggiare implementazioni per grid di modeste dimensioni (che possono essere gestite con varianti più semplici e ugualmente performanti) questo sistema richiede anche un alto livello di conoscenza dei sistemi MAS.

Conclusione

L'utilizzo di un corretta architettura di *Resource Discovery* in ambito Grid è fondamentale, perché essa è responsabile del funzionamento della griglia computazionale: come i capitoli precedenti hanno dimostrato, infatti, fattori come le performance e l'affidabilità della rete stessa dipendono in buona misura dalle tecniche di *Resource Discovery* utilizzate.
Non stupisce, quindi, che attualmente ricoprano un ruolo di primo piano in ambito di ricerca del *Grid Computing*: l'ambiente accademico presenta periodicamente nuove soluzioni che promettono di risolvere in maniera definitiva i già noti problemi di bottlenecks e single-point-failure, oltre che garantire una certa affidabilità della rete.
In questo testo si è cercato di fornire una visione generale delle principali tecniche di *Resource Discovery* utilizzate, valutando i vantaggi e gli svantaggi di ogni applicazione; conseguentemente, tale analisi permette di trarre le seguenti conclusioni:

- Il modello Centralizzato è adatto a griglie di modeste dimensioni, che non sono soggette ad un futura espansione: le sue già citate debolezze in termini di scalabilità, infatti, scoraggiano la sua implementazione in reti con un elevato numero di nodi.
  Il fattore single-point-failure inoltre è risolvibile tramite l'adozione di sistemi di backup, che anche qui sono ragionevoli fintanto che le dimensioni della griglia sono modeste.

– Il modello Gerarchico si presenta come un valido candidato qualora la rete cresca in modo considerevole, dato che quest'architettura si è dimostrata efficace anche per griglie particolarmente estese.
Tuttavia, pur mantenendo delle buone prestazioni generali, questa architettura soffre della presenza di single-point-failure nei nodi di gestione, che potrebbero paralizzare parte della rete per un tempo indefinito.

– L'utilizzo di modelli Distribuiti sembra essere particolarmente incoraggiante, considerando la loro indipendenza a qualsiasi forma di gerarchia della rete: la loro forma distribuita permette infatti di escludere problemi di single-point-failure, dato che nessun nodo è capace di paralizzare l'intera griglia computazionale.
Tuttavia, questa architettura è molto sensibile agli eventi della rete: data la particolare forma distribuita, ogni evento che modifichi la sua morfologia in maniera sensibile deve essere notificato agli altri peer, in modo da essere pronti alle successive comunicazioni.
Questo meccanismo può comportare la generazione di un elevata quantità di traffico non necessario, che porterebbe alla formazione di bottleneck per alcune connessioni di rete; in qualche variante, inoltre, è possibile la generazione di segnali falsi-positivi per quanto riguarda il *Resource Discovery*.

– Infine, il modello ad Agenti sembra riesca a risolvere i problemi sopra citati: l'utilizzo di software intelligenti permette infatti di affrontare gli eventi della rete con dinamicità, fornendo un alto grado di affidabilità.
La loro implementazione, inoltre, permette di estendere il concetto di nodo tipico delle altre reti: ogni *Resource* in questa variante è infatti capace di ricavare informazioni in completa autonomia, fornendo parametri utili come il già visto *Utility*.
Tuttavia, l'adozione di questo modello comporta un degrado sensibile delle prestazioni rispetto ad altre architetture: l'overhead provocato dagli agenti può diventare rilevante qualora la dimensione della rete diventi considerevole.

Alla luce di queste considerazioni, è bene osservare come i classici problemi di bottlenecks e single-point-failure possano essere mitigati considerevolmente, ma mai esclusi del tutto: sebbene le suddette varianti infatti reagiscano bene alla disconnessione improvvisa di un nodo o a un suo sovrautilizzo, lo sviluppo di algoritmi risolutivi per queste problematiche sarà sempre necessario.
La ragione è da ricercare su molteplici fattori: talvolta la topologia della rete rende la griglia naturalmente esposta a problemi di single-point-failure, mentre altre volte i nodi, per la loro elevata complessità, apportano un peso eccessivo su di essa.

Si può concludere pertanto che, allo stato attuale, non esista una soluzione che sia efficace per tutti i casi d'uso del Grid Computing: ogni modello deve essere scelto in base a delle precise esigenze e prospettive di sviluppo, per poter minimizzare le debolezze di ogni architettura.
Ovviamente, la ricerca in ambito Grid è ancora molto attiva, e l'ideazione di un modello di *Resource Discovery* esente da problemi rilevanti è un obiettivo molto sentito dalla comunità scientifica: è pertanto fondamentale rimanere aggiornati su queste tecnologie qualora vi sia la necessità di implementare un sistema Grid Computing.

Bibliografia

[1] Ian Foster, Carl Kesselman e Steven Tuecke.
The Anatomy of the Grid: Enabling Scalable Virtual Organizations
International Journal of Supercomputer Applications 15: 200-222, 2001.

[2] Ian Foster, Carl Kesselman.
The Grid 2: Blueprint for a New Computing Infrastructure.
Morgan Kaufmann Publishers, 2003.

[3] Bart Jacobm, Michael Brown, Kentaro Fukui e Nihar Trivedi.
Introduction to Grid Computing.
IBM International Technical Support Organization: 20-23, 2005.

[4] JDL (Job Description Language).
http://www.dcc.fc.up.pt/ ines/aulas/0910/CG/jdl.pdf

[5] JSDL (Job Submission Description Language).
https://www.ogf.org/documents/GFD.56.pdf

[6] Peter Mell, Timothy Grance.
The NIST Definition of Cloud Computing.
National Institute of Standards and Technology, 2011.

[7] Jia Yu, Srikumar Venugopal, Rajkumar Buyya.
A Market-Oriented Grid Directory Service for Publication and Discovery of Grid Service Providers and their Services.
Springer Science, 2006

[8] Yulan Yin, Huanqing Cui e Xin Chen.
The Grid Resource Discovery Method Based on Hierarchical Model.
Information Technology Journal 6: 1090-1094, 2007.

[9] P. Trunfio, D. Talia, H. Papadakis, P. FragoPoulou, M. Mordacchini, M.Pennanen ...
Peer-to-Peer resource discovery in Grids: Models and systems.
Future Generation Computer Systems, 2007

[10] Nima Jafari Navimipour, Amir Masoud Rahmani, Ahmad Habibizad Navin e Mehdi Hosseinzadeh.
Resource discovery mechanisms in grid systems: A survey.
Journal of Network and Computer Applications 41: 389-410, 2014.

[11] Adriana Iamnitchi, Ian Foster e Daniel C. Nurmi.
A peer-to-peer approach to resource location in grid environments.
11th IEEE International Symposium, 2002.

[12] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek e Hari Balakrishnan.
Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications.
MIT Laboratory for computer science, 2001.

[13] Stan Franklin e Art Graesse.
Is it an Agent, or just a Program?.
Third International Workshop on Agent Theories 21-36, 1997.

[14] Yunsong Tan, Jianjun Han e Yuntao Wu
A Multi-agent Based Efficient Resource Discovery Mechanism for Grid Systems.
Journal of Computational Information Systems 6: 3623-3631, 2010.

[15] Liangxiu Han e Dave Berry.
Semantic-supported and agent-based decentralized grid resource discovery.
Future Generation Computer Systems 24: 806-812, Elsevier 2008.

[16] Jaeyong Kang e Kwang Mong Sim.
A multiagent brokering protocol for supporting Grid resource discovery.
The International Journal of Artificial Intelligence, Neural Networks... 37: 527-542, Springer 2012.