UNIVERSITÀ DELLA CALABRIA

**Dipartimento di Elettronica,
Informatica e Sistemistica**

**Dottorato di Ricerca in
Ingegneria dei Sistemi e Informatica
XX ciclo**

Tesi di Dottorato

Super Peer Models for Public Resource Computing

Pasquale Cozza

UNIVERSITÀ DELLA CALABRIA

Dipartimento di ELETTRONICA, INFORMATICA E SISTEMISTICA

UNIVERSITÀ DELLA CALABRIA

Dipartimento di Elettronica,
Informatica e Sistemistica

Dottorato di Ricerca in
Ingegneria dei Sistemi e Informatica
XX ciclo

Tesi di Dottorato

Super Peer Models for Public Resource Computing

Pasquale Cozza

Pasquale Cozza

Coordinatore e supervisore
Prof. Domenico Talia

[illegible]

DEIS

DEIS – DIPARTIMENTO DI ELETTRONICA, INFORMATICA E SISTEMISTICA
Novembre 2007

Settore Scientifico Disciplinare: ING-INF/05

A mia moglie e a mia figlia

Preface

This thesis deals with the study and the evaluation of Peer-to-Peer distributed models and algorithms for public resource computing. In particular, in this work we present the design of a super peer model for public resource computing and its validation by simulation techniques.
The model and its simulator have been developed in the Grid Research Group of University of Calabria in collaboration with CNR-ICAR and University of Cardiff.
The first chapter gives and overview of distributed systems starting from traditional systems to more recent distributed systems; some comparison is discussed.
In the second chapter the Peer-to-Peer model is detailed, starting from a comparison with the client server model, then describing the peer-to-peer architecture,the super peer architecture, the discovery techniques used and the most common applications.
Special attention is given to a distributed computing application named Public Resource Computing, detailed in the third chapter.
The strength of this paradigm is emphasized by showing a successful project like SETI@home, and the environment widely used to support the implementation of such project, the BOINC framework.
In the fourth chapter, after a detailed analysis of reference models, it is proposed a new Super Peer model for Public Resource Computing which functionalities and performance have been tested by using a simulator.
Finally, in the fifth chapter, is discussed how the model has been extended to support a file sharing application represented by the Mrs DART use case.

Prefazione

Questo lavoro di tesi consiste nello studio e la valutazione di modelli e algoritmi distribuiti di tipo Peer to Peer ed in particolare nella progettazione di un modello a super peer per il public resource computing e sua validazione per mezzo di tecniche di simulazione. Il modello e il suo simulatore nascono dal lavoro del Gruppo di Ricerca di Griglie dell'Universita' della Calabria in collaborazione con CNR-ICAR e Universita' di Cardiff.
Nel primo capitolo e' fornita una breve panoramica dai sistemi distribuiti piu' tradizionali ai piu' recenti, con relativi confronti.
Nel capitolo 2 e' approfondito il modello Peer-to-Peer partendo da un confronto con l'architettura client-server, e poi fornendo una descrizione della architettura Peer-to-Peer, dell'architettura a Super Peer, del protocollo di comunicazione, delle tecniche di discovery utilizzate e delle applicazioni piu' diffuse.
Particolare attenzione e' rivolta ad una applicazione di calcolo distribuito che prende il nome di Public Resource Computing, descritta nel capitolo 3. Le potenzialita' di questo paradigma sono descritte fornendo un esempio di progetto di successo, SETI@home, e l'ambiente piu' utilizzato per il supporto all'implementazione di tali progetti, il framework BOINC.
Nel capitolo 3, dopo aver accuratamente dettagliato i modelli di riferimento, viene proposto un nuovo modello a super peer per il public resource computing, le cui funzionalita' e potenzialita' sono state testate mediante l'uso di un Simulatore.
Infine, nel capitolo 5, viene mostrato come il modello possa essere esteso al fine di supportare applicazioni di tipo file sharing, il caso di Mrs DART.

Contents

Part I Distributed computing: state of the art

1 Distributed systems 3
1.1 Cluster computing 4
1.2 Grid computing 6
1.3 Grids vs. Clusters 8
1.4 P2P systems 9
1.5 Grid vs. P2P 11

2 Peer-to-Peer 15
2.1 Historical 15
2.2 P2P networks: comparing to client-server model 16
2.3 P2P protocol definitions 17
2.4 P2P architectures 18
2.5 Discovery mechanisms for P2P systems 20
2.5.1 Centralized and decentralized indexes 20
2.5.2 Discovery techniques in unstructured P2P systems 22
2.6 Uses 24
2.6.1 Distributed Computing 25
2.6.2 File sharing 26
2.6.3 File sharing: P2P example architectures 26
2.6.4 Collaboration 32
2.6.5 Platforms 34
2.7 Bittorrent 34
2.8 Super Peer 36
2.8.1 Peer discovery and network creation 37
2.8.2 Resource discovery and querying inside a SP network . . 38
2.8.3 SP networks optimizations: redundancy 38

3 Public-Resource Computing 41
3.1 PRC model 42
3.2 PRC problem features 42
3.3 PRC projects 43
3.4 SETI@home 45
3.5 BOINC 50
3.5.1 BOINC project implementation 52
3.6 PRC social aspects 57

Part II A Super-peer model for Public Resource Computing

4 Super peer model and super peer model simulator 61
4.1 Job Submission protocol 62
4.1.1 Job Assignment and Data Download 63
4.1.2 Dynamic Caching 66
4.2 Discovery mechanism 68
4.3 Simulator description 68
4.3.1 Simulator messages 69
4.3.2 Membership protocol 75
4.3.3 Simulation Parameters 77
4.3.4 Performance Indices 78
4.4 Performance Evaluation 78

5 Model analysis by simulation techniques 81
5.1 Base scenario 82
5.2 Redundant submission of job scenario 84
5.3 With data caching scenario 88
5.4 Peers disconnection scenario 92
5.5 Scalability analysis 101
5.6 Multiple jobs assigner scenario 106

6 P2P networks for music information retrieval 109
6.1 DART scenario 110
6.2 Distributed Simulations 114

References 119

Part I

Distributed computing: state of the art

1 Distributed systems

Computer technology has revolutionized science. Scientists have developed accurate mathematical models of the physical universe, and computers programmed with these models can approximate reality at many levels of scale: an atomic nucleus, a protein molecule, the Earth's biosphere, or the entire universe. Using these programs, we can predict the future, validate or disprove theories, and operate "virtual laboratories" that investigate chemical reactions without test tubes [56].
Computer technology also has revolutionized itself to better support scientific required tasks and thanks to the electronic innovation.

Since Internet diffusion many desktop computers, laptop, play stations and so on were linked to the network to share data and to intercommunicate, as side effect these computers constitute a huge virtual computer made by computation resource and data spread over all the network. Distributed systems is exactly the technology that incorporate this concept to use shared unused resources.
There are several definitions and view points on what distributed systems are. Coulouris defines a distributed system as "a system in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing" [1]; and Tanenbaum defines it as "A collection of independent computers that appear to the users of the system as a single computer" [2]. Leslie Lamport once said that "A distributed system is one on which I cannot get any work done because some machine I have never heard of has crashed" reflecting on the huge number of challenges faced by distributed system designers. Despite these challenges, the benefits of distributed systems and applications are many, making it worthwhile to pursue.
The main features of a distributed system include [1, 2]:

- *Functional separation:* It's based on functionality/services provided, capability and purpose of each entity in the system.

- *Inherent distribution:* Entities such as information, people, and systems are inherently distributed. For example, different information is created and maintained by different people. This information could be generated, stored, analysed and used by different systems or applications which may or may not be aware of the existence of the other entities in the system.
- *Reliability:* Long term data preservation and backup (replication) at different locations is managed.
- *Scalability:* It consists in the addition of more resources to increase performance or availability. In the opposite in centralize systems scalability is usually restricted by the amount of centralized operation necessary and such system largely avoid central instances or servers.
- *Economy:* Resources are shared by many entities to help reduce the cost of ownership. As a consequence of these features, the various entities in a distributed system can operate concurrently and possibly autonomously. Tasks are carried out independently and actions are co-ordinated at well-defined stages *by exchanging messages.*

As a consequence of these features, the various entities in a distributed system can operate concurrently and possibly autonomously. Tasks are carried out independently and actions are co-ordinated at well-defined stages by exchanging messages. Also, entities are heterogenous, and failures are independent. Generally, there is no single process, or entity, that has the knowledge of the entire state of the system.

So far various types of distributed systems and applications have been developed and are being used extensively in the real world.
Inside of the various types of distributed systems, we analyze Clusters [3], Grids [5], and $P2P$ (Peer-to-Peer) networks [27].

1.1 Cluster computing

A cluster is a dedicated group of interconnected computers that appears as a single super-computer. It's possible classify clusters depending from possible applications to do with. *High availability clusters* have a single virtual host on the network interface where a very big number of users can connect; an example is a website as a websearch engine accessed bye many people frequently or a huge database with a huge frequence of query access.
High reliability clusters are generally made bye to twins machine, one the mirror of the other, that are opportunaly configured then once one is not available the other starts work in sake of. *Load condivision clusters* are arranged in a way that different kind of concurrent users can access available resources thanks the mediation of a *Resource Manager. Load balance cluster* are arranged in a way that a resource request can be forwarded to any available node depending from the load, think for example about a cluster that

must manage a growing number of users in an online game, a scientific data management and business applications. *High performance cluster* are special for process that generally can split in small subprocess and they are weakly dependent as in scientific engineering, robot management, medical imagines analysis, military control systems. Finally *Cluster grid* are clusters geographically distributed in stand of being inside a room, we will refer to them with the name *grid computing* in the rest of this work.
Since they can support real time applications as well, cluster systems should solve temporal constraints well defined to acceding resources of one application. From software point of view they need operating systems ad hoc or conventional operating systems with a software support for communication and data sharing, also to perform their task they need again specific scheduling algorithms.
Realizing a cluster means to build a "parallel computer", you need N computing nodes ($N > 1$) linked by a communication network to allow the data exchange among these nodes. No special requirements about net topology, speed and if dedicated or not. About software, a cluster generally uses opensource programs. Beowulf project [4] is the referement design model for many cluster realized like this and in figure 1.1 there is an architectural schema of it.
Beowulf clusters are tipically of computing centers where the issue is high

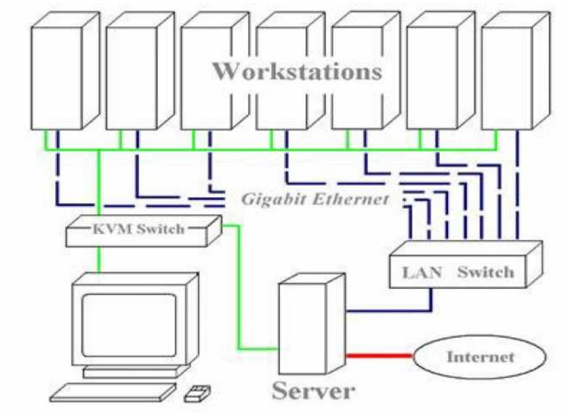


Fig. 1.1. Beowulf: the schema

performance computing, cpu intensive and storage intensive.
Thanks opensource software and reusing available unused old computers were operating system linux works well, clusters can offer high reliablity and at the same time not high costs.

1.2 Grid computing

The older definition of what a grid is how it's intended nowadays, it's from Ian Foster and Carl Kessellman that, in 1999, defined computation grids in The Grid: Blueprint for a New Computing Infrastructure as "A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive and inexpensive access to high-end computational capabilities". In this vision, the Grid will be to *all* computational resources what the World Wide Web presently is to documents containing information.
Grid users will have in fact at their disposal distributed high performance computers able to access and process terabytes of data stored in global databases, plus the appropriate tools to control and to share these resources.
In [17] Foster again try to define grids introducing the concept of virtual organizations. In this work the grid concept appears as the *coordinated resources sharing and problem solving in dynamic, multi-institutional virtual organizations (VO).* With VO is intended a set of individuals and/or institutions defined by such sharing rules. Later Foster synthesize a grid by listing three

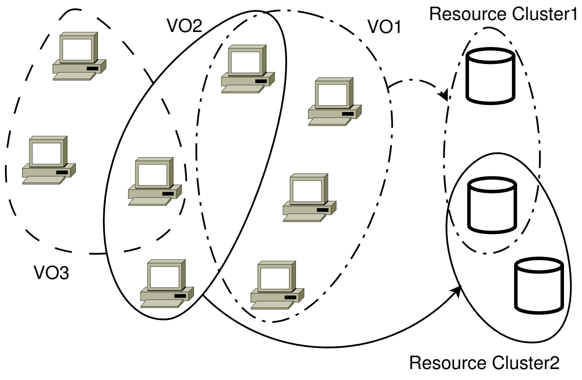


Fig. 1.2. Virtual Organizations accessing different and overlapping sets of resources

primary attributes [6]:

1. Computing resources are not administered centrally.
2. Open standards are used.
3. Non-trivial quality of service is achieved.

Plaszczak/Wellner define grid technology as "the technology that enables resource virtualization, on-demand provisioning, and service (resource) sharing between organizations". IBM defines grid computing as "the ability, using a set of open standards and protocols, to gain access to applications and data, processing power, storage capacity and a vast array of other computing resources over the Internet. A grid is a type of parallel and distributed system

that enables the sharing, selection, and aggregation of resources distributed across 'multiple' administrative domains based on their (resources) availability, capacity, performance, cost and users' quality-of-service requirements" [9]. Buyya defines a grid as "a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed autonomous resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements" [10] CERN, one of the largest users of grid technology, talk of The Grid: "a service for sharing computer power and data storage capacity over the Internet" [11]. A grid can be also intended as a type distributed system that enables coordinated sharing and aggregation of distributed, autonomous, heterogeneous resources based on users' QoS (Quality of Service) requirements.
Conceptually, the Grid can be thought of in terms of three layers. Underlying everything is the computational and data grid: the computer hardware and data networks upon which the work will be conducted. Above this is the 'information grid': the databases of information to be accessed by the hardware, and systems for data manipulation. On top is the 'knowledge grid', where high-level applications will mine the data for the knowledge that can form the basis of semantic understanding and intelligent decision making.
Middleware acts to interface between three types of entity: the users, the data they wish to process, and the computational resources required for this processing. Central to these interactions are metadata, that is to say descriptive information about these three types of entity, organized in a systematic manner that makes automated interactions possible.
Individual agents can continuously represent the users, the data and the resources on the Grid by presenting their metadata, while who offer a grid service must deal with authentication and authorization for manage payment, scheduling of activities, task monitoring.
A key point for successful Grid computing, is in fact to provide security access policies. When a single user account with a single log-on procedure must be sufficient for pervasive (any time, any place) access to all the computational resources required, with security permissions being handled automatically between the separate systems in a manner transparent to the user. This requires the development of certification systems employing modern public key encryption technology to establish the identity and trustworthiness of the user *(user authentication)*, and then to grant him/her access to those computational facilities and databases that the user has the right to use, perhaps by virtue of institutional membership or subscription to a database service *(user authorization).*
For the Grid to become a reality, there is the strict requirement that the applications used for information processing on both local and distant computers become integrated and truly interoperable. New software developments such as XML (eXtensible Markup Language), RDF (Resource Description Framework) and CORBA (Common Object Request Broker Architecture), together with systems for load balancing between distant computers, and task integra-

tion toolkits such as Globus [16], are likely to form the basis of the Grid tools that will be central to future interoperability, providing homogeneous access to heterogeneous data and resources [7].
So far grids are commonly used to support applications emerging in the areas of e-Science and e-Business, which commonly involve geographically distributed communities of people who engage in collaborative activities to solve large scale problems and require sharing of various resources such as computers, data, applications and scientific instruments.

1.3 Grids vs. Clusters

At first look it can seems that Grids and clusters can solve the same kind of computations, in fact both may satisfy high-performance or high-throughput requirements by enabling distributed computing. The point is that grids solve the more complicated problem of providing computing resources to groups that span organizational boundaries, where resources are spread among the groups. In fact, a grid may marshal numerous clusters from different organizations into a logical set of computational resources available to a group of authorized users. By definition, grid services must live in more a complex environment where resources must be shared and secured according to policies that may differ from organization to organization. In contrast, cluster computing evolved simple as a way of solving computationally demanding problems by using large numbers of commodity CPUs together with commodity networking technology.
Over the last part of the 20th century as computing power increased and prices dropped, it became clear that if large numbers of low-cost computers, for example computer connected to Internet, could provide supercomputing power at a much lower cost than purpose-built, high-performance supercomputers.

Because processes must communicate with other processes via the network, rather than hardware on the motherboard, communication is much slower. Also, high-speed RAM availability is limited by the amount of memory available to hosts in the cluster. Given these constraints, clusters have still proven invaluable in high-performance computing for solving problems that can easily be broken into many smaller tasks and distributed to workers. Ideal problems require little communication between workers, and their work product can be combined or processed in some way after the tasks have been completed.
Grids can certainly solve these sorts of problems, but they might have a supercomputer available for tasks that cannot be broken up so easily. The grid would provide a way to match this supercomputer with your problem, reserve it, authenticate your task and authorize its use of the supercomputer. It

Fig. 1.3. Columbia Super Cluster of NASA from 10.240 processors

would execute the task and provide a way to monitor progress on that supercomputer. When the supercomputer completes your task, it would send the results to you. This supercomputer might even be in a different hemisphere and owned by a different institution. Finally, the grid might even debit your account for using this service.
By way of contrast, a cluster might provide some of these services. It might even be a cluster of supercomputers, but the cluster would probably belong entirely to your institution, and it probably wouldn't bill you. In addition, your institution probably would have a consistent policy and method of authenticating your credentials and authorizing your use of the cluster. More important, the cluster would probably exercise complete and centralized control over its resources [12]. Finally we must considerate that Grids emerged to solve resource-sharing problems across academic and research institutions, where funding for researching a broad topic might be distributed across a variety of institutions that employed researchers focusing on particular aspects of that topic. Sharing this experimental data among many geographically distributed research organizations and researchers requires sophisticated resource-sharing technology that can expose those resources in an open, standard, and secure way. These requirements far exceed those of a cluster intended to provide high-performance computing for a given institution.

1.4 P2P systems

The *P*2*P* idea is based on the notion of increasing the decentralization of systems, applications, or simply algorithms. It is based on the principles that the world will be connected and widely distributed and that it will not be possible or desirable to leverage everything off of centralized, administratively managed infrastructures. *P*2*P* is a way to leverage vast amounts of computing power, storage, and connectivity from personal computers distributed around

the world.
A $P2P$ system then is one in which autonomous peers depend on other autonomous peers. Peers are autonomous when they are not wholly controlled by each other or by the same authority, e.g., the same user. Peers depend on each other for getting information, computing resources, forwarding requests, etc. which are essential for the functioning of the system as a whole and for the benefit of all peers. As a result of the autonomy of peers, they cannot necessarily trust each other and rely completely on the behaviour of other peers, so issues of scale and redundancy become much more important than in traditional centralized or distributed systems.
As with any computing system, the goal of $P2P$ systems is to support applications that satisfy the needs of users.

Centralized systems that serve many clients typically bear the majority of the cost of the system. When that main cost becomes too large, a $P2P$ architecture can help spread the cost over all the peers. Much of the cost sharing is realized by the utilization and aggregation of otherwise unused resources which results both in net marginal cost reductions and a lower cost for the most costly system component, then $P2P$ model brings a *cost sharing/reduction.*

Either such system can give an *improved scalability/reliability* respect the situation to have a strong central authority thanks autonomous peers.

A decentralized approach lends itself naturally to *resource aggregation and interoperability.* Each node in the $P2P$ system brings with it certain resources such as compute power or storage space. Applications that benefit from huge amounts of these resources, such as compute-intensive simulations or distributed file systems, naturally lean toward a $P2P$ structure to aggregate these resources to solve the larger problem.

In many cases, users of a distributed system are unwilling to rely on any centralized service provider. Instead, they prefer that all data and work on their behalf be performed locally. $P2P$ systems support this level of *autonomy* simply because they require that the local node do work on behalf of its user. Related to autonomy is the notion of *anonymity and privacy.* A user may not want anyone or any service provider to know about his or her involvement in the system. With a central server, it is difficult to ensure anonymity because the server will typically be able to identify the client, at least by Internet address. By employing a $P2P$ structure in which activities are performed locally, users can avoid having to provide any information about themselves to anyone else. $P2P$ systems assume that the computing environment is highly *dynamic.* That is, resources, such as compute nodes, will be entering and leaving the system continuously. When an application is intended to support a highly dynamic environment, the $P2P$ approach is a natural fit. $P2P$ system even is linked to dynamism concept, that is *enabling ad-hoc communication and collaboration*, where members come and go based perhaps on their current physical location or their current interests.
A more detailed description of current $P2P$ systems is given in next chapter.

1.5 Grid vs. P2P

Generally moderate-sized communities use Grid services specially when it's mandatory a trusted information exchange. In contrast, current *P2P* systems deal with many more participants and offer limited and specialized services, have been less concerned with qualities of service and trust. Anyway the point is that both two environments deal with the same general problem, namely, *resource sharing within VOs that may not overlap with any existing organization.*
Grid systems integrate resources that are more powerful, more diverse, and better connected than the typical *P2P* resource. Anyway the two types of system have both conceptual and concrete distinction about resources definition, target communities, connectivity, scalability, reliability and so on.
A Grid resource in fact might be *a cluster, storage system, database, or scientific Instrument* of considerable value that is administered in an organized fashion according to some well defined policy, while *home computers* arguably represent the majority of *P2P* resources.
P2P has been popularized by file sharing and public resource computing applications, the latter is a high performance conputing based on volunteers who donate their personal computers' unused resources to a computationally intensive research project.
In contrast with Grids, public resource computing involves an asymmetric relationship between projects and participants. Projects are typically small academic research groups with limited computer expertise and manpower. Most participants are individuals who own Windows, Macintosh and Linux PCs, connected to the Internet by telephone or cable modems or DSL, and often behind network-address translators (NATs) or firewalls. The computers are connected intermittently to the network, remaining available for a limited time with reduced reliability. Anyway the number of nodes connected in a *P2P* network at a given time is much greater than in a grid. Thus, participants are not computer experts, and participate in a project only if they are interested in it and receive "incentives" such as credit and screensaver graphics. Projects have no control over participants, and cannot prevent malicious behaviour [72, 19].
Grids generally include powerful machines that are statically connected through high performance networks with high levels of availability. On the other hand, the number of accessible nodes is generally low because access to grid resources is bounded to rigorous accounting mechanisms.

About applications, *P2P* systems tend to be vertically integrated solutions to specialized resource-sharing problems: currently deployed systems share either compute cycles or files while Grid systems more data intensive. For example, a recent analysis of Sloan Digital Sky Survey SDSS data [15] involved, on average, 660 MB input data per CPU hour; In contrast, SETI@home [67] moves at least four orders of magnitude less data: a mere 21.25 KB data per

CPU hour. The reason is presumably, in part at least, better network connectivity, which also allows for more flexibility in Grid application design: in addition to loosely coupled Applications, Grids have been used, for example, for numerical simulation and branch-and-bound based optimization problems. Grid systems must address the problem of scalability and deal with failures

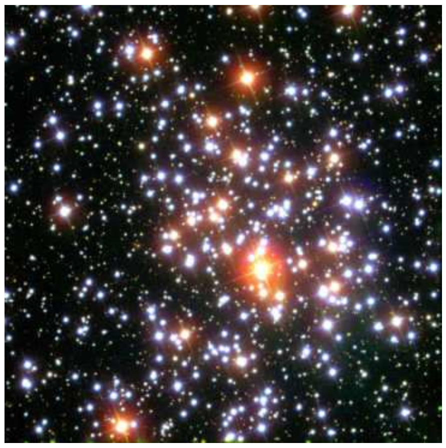


Fig. 1.4. The constellation Auriga, picture for Mapping the Universe project [15]

in terms of amount of activity, while *P*2*P* in terms of participating entities. Early Grid implementations did not address scalability and self-management as priorities. Thus, while the design of core Grid protocols does not preclude scalability, actual deployments often employ centralized components. One example of that is the Globus toolkit [16] where you can find central repositories for shared data, centralized resource management components, and centralized and/or hierarchical information directories. This situation is changing, with much work proceeding on such topics as reliable and scalable management of large job pools, distributed scheduling, replica location, and discovery.

Far larger *P*2*P* communities exist: millions of simultaneous nodes in the case of file-sharing systems and several million total nodes in SETI@home. The amount of activity is also significant, albeit, surprisingly, not always larger than in the relatively smaller-scale Grids (1-2 TB per day in file sharing systems). This large scale has emerged from robust self-management of large numbers of nodes.

The technologies used to develop Grid and *P*2*P* applications differ both in the specific *services* provided and in the emphasis placed on persistent, multipurpose *infrastructure.* Grid provide to creating and operating persistent, multipurpose *infrastructure* services for authentication, authorization, discovery, resource access, data movement, and so forth.

Many Grid communities use the open source Globus Toolkit [16] as a technology base. Significant effort has been channeled toward the standardization of protocols and interfaces to enable interoperability between different Grid deployments. $P2P$ systems have tended to focus on the integration of simple resources (individual computers) via protocols designed to provide specific vertically integrated functionality. Such protocols do, of course, define an infrastructure, but in general the persistence properties of such infrastructures are not specifically engineered but are rather emergent properties [18].

2

Peer-to-Peer

2.1 Historical

Peer-to-Peer ($P2P$) is the model of earlier distributed applications and it aims to employ distributed resources to perform function in a decentralized manner; in $P2P$ applications, resource can be computing, storage and bandwidth, while function can be computing, data sharing, collaboration.
E-mail systems built on the SimpleMail Transfer Protocol (SMTP) and Usenet News are first examples of such applications. The main idea is that there are local servers that received a message then they built connections with *peer (P)* servers to deliver messages into a user's mail file or into a spool file containing messages for the newsgroup. The File Transfer Protocol (FTP) currently is a client-server application, in the beginning it was very common for individuals to run FTP servers on their workstations to provide files to their Ps for such reason it's can be intended as the precursor to today's file sharing $P2P$ systems. Even an indexing system named Archie, was developed to provide a central search mechanism for files on FTP servers. This structure with central search and distributed files is exactly replicated in a very popular $P2P$ system: Napster.

At the time decentralized dial-up networks as UUNet and Fidonet were used, they were composed of a collection of machines that made periodic dial-up connections to one another. On a typical connection, messages (again, typically e-mail or discussion group entries) were transferred bi-directionally. Often, a message would be routed through multiple dial-up hops to reach its destination. This multi-hop message routing approach can be seen in current $P2P$ systems such as Gnutella.

At the very beginning of Internet all the content was provided by machines at similar levels (ignoring routing intermediaries that provide services like DNS and DHCP) with Arpanet. If someone wanted to publish something, they published it on their machine (or one within their physical location).

Things were balanced because equal amounts of traffic flowed both directions - all the nodes were Ps. An historical example of $P2P$ application was the Internet Relay Chat (IRC). The origin of IRC can be traced back to the Department of Information Processing Science at the University of Oulu[1], Finland during the latter part of August 1988. IRC consists of distributed servers that relay chat information between each other. A set of these servers is called a net. A user, or client, connects to one of these servers and joins a channel (like a chat room). Once they are in a channel their chat is relayed to every other client in that channel on that net. There are hundreds of established nets available and a given net can have thousands of channels.
Gradually with time at the late-80's and early- 90's more transients users connected to the network. This moved to more of a server model, with multiple clients connecting to it. It also allowed the use of less expensive and less functional computers such as desktop PCs. Today again with faster always on connections a new type of connectivity between the individual clients is on the rise.
The first wide use of $P2P$ seems to have been in instant messaging systems such as AOL Instant Messenger. These are typically hybrid $P2P$ solutions with discovery and brokering handled by a central server followed by direct communication between the P messaging systems on the PCs. The current phase of interest and activity in $P2P$ was driven by the introduction of Napster [Napster 2001] in 1999. It came at a time when computers and their network connections were nearing the level found previously in technical and academic environments, and re-created earlier approaches with an interface more suitable to a non-technical audience.

2.2 P2P networks: comparing to client-server model

Network configurations mainly include server and P networks. Typically in server (or Client / Server) networks there are one or more central servers that choreograph all networking activity between all the machines, and thanks to technologies like DHCP assign network address to them, then technologies like DNS allow one client to connect to the other using a common name without needing to know the machines actual address. In the opposite in a P network all the machines are equal, with each machine discovering what other machines are on the network and obtaining an address without the help of a server.
The presence of servers aims to provide a central location allowing the entire network to be administered from one location (or at least that is the theory.) and it symbolizes a source that can spread contents by request.
The downside is that server and administrator do the extra work to make it easier on the clients and end users. Servers, representing the single point of

[1] IRC: http://www.oulu.fi/english/index.html

failure of the whole network, are costly, both in time and money, so the more computers to be serviced by a server, the more cost effective it is to have one.

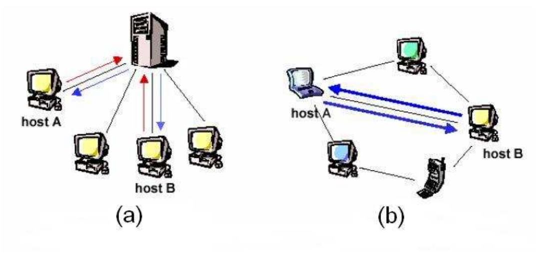


Fig. 2.1. Communication Systems: (a)Client-Server, (b)Peer-to-Peer

A P network does not require a special central server; this makes it more appealing to small networks, especially ones found in the home.
The downside is that it can take a while for a machine to find another machine on the network, and the burden of connecting correctly and $P2P$obtaining an address is placed on each client. If a P network is going to be very large then an administrator may be required to keep things running smoothly, unfortunately there is no central server for them to administer, so they will be required to administer each machine on the network.
A P can be seen as a client, where client indicates subservient to a server. Instead of calling the model client to client we call it *peer to peer* since for these connections they are equal or Ps.
The Ps may have to handle a limited connectivity, support possibly independent naming, and be able to share the role of the server.
The fact that a central server may have made the initial connection to the Internet (or some other network) possible is irrelevant in where the final activity takes place. The outcome is that all the action is in the Ps, or the fringes of the network as having all entities being client and servers for the same purpose.
If you want to publish something, you do so on your own machine, instead of on an external server. $P2P$ is a new abstraction on top of the current Internet structures.

2.3 P2P protocol definitions

To explain what $P2P$ protocol aims to do, it's suitable starting from common definitions of it. $P2P$ is defined by online dictionaries in the following ways: "$P2P$ is a communications model in which each party has the same capabilities and either party can initiate a communication session" by Whatis.com, "A type of network in which each workstation has equivalent capabilities and responsibilities" by Webopedia.com, "A $P2P$ computer network refers to any network that does not have fixed clients and servers, but a number of P nodes that function as both clients and servers to other nodes on the network" by Wikipedia.org.

Either there are several of the definitions of $P2P$ that are being used by the $P2P$ community. The Intel $P2P$ working group defines it as "the sharing of computer resources and services by direct exchange between systems" [25]. Alex Weytsel of Aberdeen defines $P2P$ as "the use of devices on the internet periphery in a nonclient capacity" [21]. $P2P$ are a "Class of systems and applications that employ distributed resources to perform a function in a decentralized manner" according to Vana Kalogeraki, Riverside. Ross Lee Graham defines $P2P$ through three key requirements: a) they have an operational computer of server quality; b) they have an addressing system independent of DNS; and c) they are able to cope with variable connectivity [29]. Clay Shirky of O'Reilly and Associate uses the following definition: "$P2P$ is a class of applications that takes advantage of resources - storage, cycles, content, human presence - available at the edges of the Internet. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses, $P2P$ nodes must operate outside the DNS system and have significant or total autonomy from central servers" [26]. Finally, Tim Kindberg of HP Labs defines $P2P$ systems as those with independent lifetimes [22].

2.4 P2P architectures

Decentralization is one of the major concept of $P2P$ systems. This includes distributed storage, processing, information sharing and also control information. Based on the degree of decentralization in a $P2P$ system, we can classify them into two categories: *Purely Decentralized* and *Hybrid Architecture*. Recently a new architecture has been successfully proposed: the *Super Peer (SP)* model [83], for details look at last section in this chapter.
Purely Decentralized seems to be the best but at the same time the hardest to realize, it can't rely on a always on servers, the case of Hybrid Architecture, or on a P with extra features, the case of SP.

An architecture overview is depicted in the following figure:

Following a description of each architecture according the way P node are linked and how they intercommunicate.

As well a valuation is done respect of main distributed application characteristic: management extendibility scalability, security, fault-tolerance and load balancing.
Lookup and discovery phases are detailed respect to $P2P$ architecture.

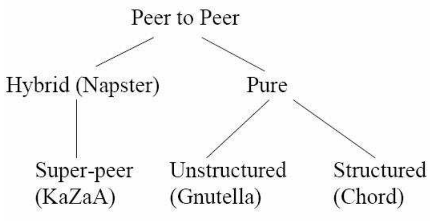


Fig. 2.2. Architecture

A pure $P2P$ system is a distributed system without any centralized control. In such systems all nodes are equivalent in functionality. In such networks the nodes are named as *servent (SERver+cliENT)*, the term servent represents the capability of the nodes of a peer-to-peer network of acting at the same time as server as well as a client.

Pure $P2P$ systems are inherently scalable, they are inherently fault-tolerant too, since there is no central point of failure and the loss of a P or even a number of Ps can easily be compensated. They also have a greater degree of autonomous control over their data and resources. On the other hand such systems present slow information discovery infact a query of the network must make many hops to reach many nodes, which takes much longer and there is no guarantee about quality of services. If the network is very large then you will most likely never query the entire network. Also because of the lack of a global view at the system level, it is difficult to predict the system behaviour.

Gnutella [39], Freenet [41], Chord [44] and CAN [46] are instances of such purely decentralized systems.

Hybrid or Brokered $P2P$ systems make use of a *Central Server or Broker* that maintains directories of information about registered users to the network, in the form of meta-data. This server may provide other services to aid in the matching of the Ps. Once the match is made then the end-to-end interaction is directly between two P clients.

Each node only ever knows about the central server and any other nodes that the central server introduces it too. The advantage of brokered in fact is that it has the performance of centralized but also allows direct P connections. Later in this section is detailed how the indexing phase happen in such systems, it can be centralized or decentralized as well.

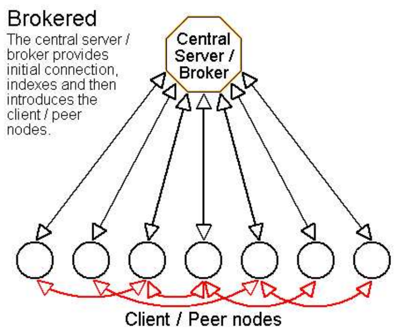

Brokered
The central server / broker provides initial connection, indexes and then introduces the client / peer nodes.
Central Server / Broker
Client / Peer nodes

Fig. 2.3. Brokered architecture

2.5 Discovery mechanisms for P2P systems

Distributed $P2P$ systems often require a discovery mechanism to locate specific data within the system. In general $P2P$ systems have evolved from first generation centralized structures to second generation flooding-based and then third generation systems based on distributed hash tables (chord, can).
First taxonomy of search methods can be done according to the location of the metadata , if *centralized* or *decentralized*, in informed approaches.

2.5.1 Centralized and decentralized indexes

Centralized indexes and repositories is the mechanism used in hybrid $P2P$ systems.

Centralized indexing scenario is the case of a central server maintains an index with meta data (file name, time of creation etc.) of files that are currently being shared by active Ps, a table of registered user connection information (IP addresses, connection speeds etc.), a table listing the files that each user holds and shares in the network. Each P maintains a connection to the central server, which store all information regarding location and usage of resources. Upon request from a P, the central index will match the request with the best P in its directory that matches the request. The best P could be the one that is cheapest, fastest, nearest, or most available, depending on the user needs. Then the data exchange will occur directly between the two Ps.

The user then opens a direct connection with the P that holds the requested file, and downloads it. This architecture is used by Napster [51].

The disadvantage is that such systems are vulnerable to censorship and malicious attack. Because of central servers they have a single point of failure. They are not inherently scalable, because of limitations on the size of the

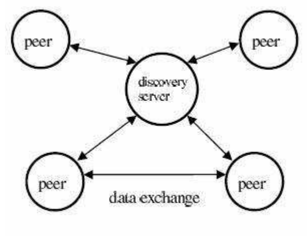


Fig. 2.4. Centralized indexing

database and its capacity to respond to queries. As central directories are not always updated, they have to be refreshed periodically.

To deals with this problems the decentralized indexing model comes up, in this case the central server task is to register the users to the system and facilitates the P discovery process. The server doesn't represent the single point

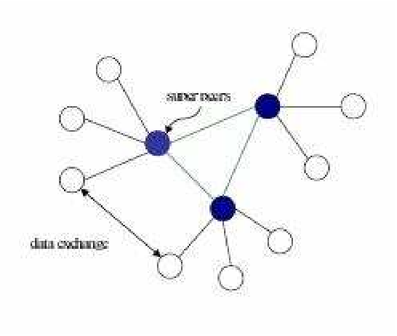


Fig. 2.5. Distributed indexing

of failure in fact some of the nodes assume a more important role than and they are called "supernodes" [55]. These nodes maintain the central indexes for the information shared by local Ps connected to them and proxy search requests on behalf of these Ps so queries are therefore sent to SuperNodes, not to other Ps, this happens in Kazaa [49, 37] and Morpheus [52].

To be designed as SP, a P must have sufficient bandwidth and processing power and a central server provides new Ps with a list of one or more SuperNodes with which they can connect. Such node turns into regular P if it doesn't receives at least the required number of connections to client nodes within a specified time, then it tries to become a SuperPeer again for another probation period. The concept of Super Nodes is common in Gnutella [38].

Respect to purely decentralized systems, decentralized indexing hybrid architectures reduce the discovery time and also they reduce the traffic on messages exchanging between nodes. Respect to centralized indexing, they reduce the workload on central server but they present slower information discovery.

2.5.2 Discovery techniques in unstructured P2P systems

Discovery techniques in unstructured $P2P$ systems, according to the information they utilize to locate objects, include *Blind methods*, the case of Gnutella, Random Walks, and *Informed methods*, the case of Napster, Routing Indices. Pure $P2P$ model belong to unstructured $P2P$ systems, this is the case in which each P does not maintain any central directory and each P publishes information about the shared contents in the $P2P$ network. Since no single P knows about all resources, Ps in need for resources flood an overlay network queries to discover a resource, each request from a P is flooded (broadcasted) to directly connected Ps, which themselves flood their Ps etc., until the request is answered or a maximum number of flooding steps occur. Flooding based search networks are built in an ad hoc manner, without restricting a priori which nodes can connect or what types of information they can exchange [31]. Different broadcast policies have been implemented to improve search in $P2P$ networks [36, 34, 35].

Flooding broadcast of queries algorithms include *simple flooding*, *iterative deepening*, *Random walk*, *Informed search.* The base flooding algorithm is the *Simple Flooding*; according this algorithm a P sends the query to all of its neighbour nodes, in the case a neighbour has the result, it will notify the query initiator; and the query initiator can get the result directly from it. In the case the neighbour hasn't the result this will decrease TTL (Time To Live) and forward the query to its neighbours. The forward of this query halts when TTL becomes 0. It means that a query initiator can get redundant results, coming from different nodes having the resource asking for, or no result even if data exists in network.

The simple flooding generates a lot of network traffic, to deal with this, in the Iterative Deepening the idea is that the search is started using flooding with small TTL. In this way, in the case the result is close to the requiring P, the message doesn't congest all the network.

If no result is found, new search is started with larger TTL. Then the algorithm halts when result is found or limit of TTL is reached. To allow this a policy array indicating for each search iteration the TTL, must be used. Otherwise the TTL can change according a choosed math function. An other flooding algorithm example is the *Unstructured: Random Walk (Blind Search).* In this case the query initiator selects only one neighbor to send the query according some heuristics. For example, if a neighbor always returns satisfactory results, it might be selected more often. This is possible if it receives feedback of whether neighbor was able to provide result, if the neighbor doesn't have the result, it will select one of its neighbors based on the heuristics. This pro-

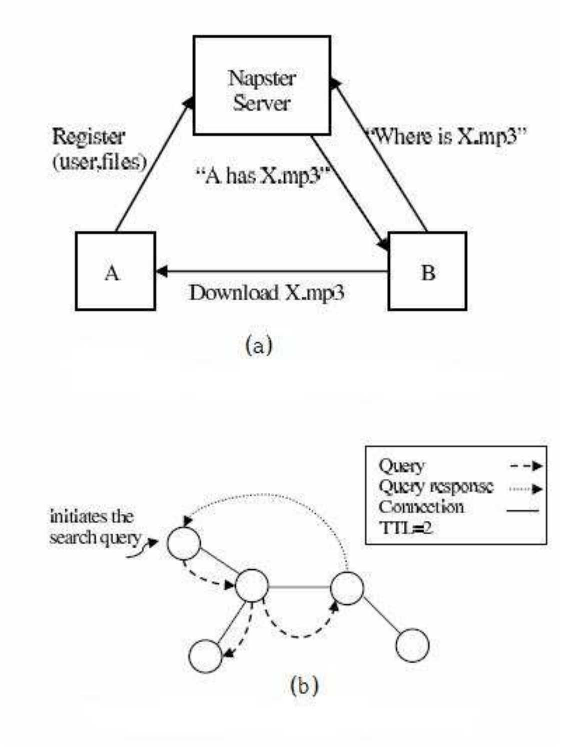


Fig. 2.6. Flooding algorithm: (a)Resource discovery in Napster, (b) Flooding-based broadcast

cess will repeat until result is found or TTL is met.
The opposite of the Blind Search is the *Informed Search* approach, according this in fact each P has a lookup index, neither complete nor accurate, storing file locations which have been searched previously.
If a P finds the location for a file in the index, it will directly contact the file holder and get the file. otherwise, it uses flooding for search. Once the file is found, the reverse path of query path is used to inform the query initiator about the location, it's this the way so Ps on the query path can update their indices speeding up next queries.
A Informed Search variation is that instead of storing the the file location the file will be replicated along the reversed query path. This combined with lookup indices is called *Replication Search* algorithm.
All the mechanisms analyzed so far, typicall of unstructured $P2P$ systems have some drawbacks as a the large amount of messages travelling inside the network, the need of duplicate queries and the hard task to opportunely set the TTl value, keeping under consideration that a too high TTL value brings a high load in network, a too low value could bring to not found any result.

Search in structured $P2P$ systems: the routing model The routing model adds structure to the way information about resources are stored using distributed hash tables and in some special case content based hash table. This protocol provide a mapping between the resource identifier and location, in the form of a distributed routing table, so that queries can be efficiently routed to the node with the desired resource.
Data items are distributed over Ps according to a well defined algorithm. Ps choose their data items using additional replication mechanism to check for availability. Each node has a unique identifier (Hash of IP) and each data item (e.g. file) that must be assigned has a key (Hash of title, author etc). Each node is responsible for storing files that have a key that is similar to the node identifier: given a key, a node efficiently routes the query to the node with an ID closet to the key.

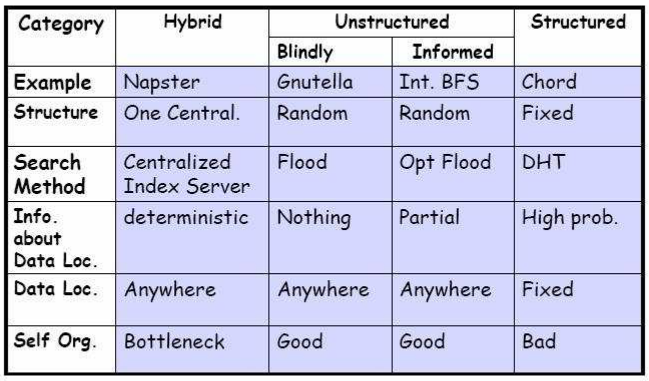
This protocol reduces the number of $P2P$ hops that must be taken to locate a resource. The look-up service is implemented by organizing the Ps in a structured overlay network, and routing a message through the overlay to the responsible P [23]. Starting from infomation stored in a DHT many services can be implemented as File sharing, Archiving, Database, names Directory, cat services, publish/subscribe systems, distributed Cache, streaming audio/video systems. Example of Structured $P2P$ systems that implement *Distributed Hash Table DHT* are Chord [MIT], Pastry [Microsoft Research UK, Rice University], Tapestry [UC Berkeley], Content Addressable Network (CAN) [UC Berkeley], SkipNet [Microsoft Research US, University of Washington], Kademlia [New York University], Viceroy [Israele, UC Berkeley], P-Grid [EPFL Ginevra].

An alternative of hash table structure is the bloom filter. A Bloom filter is an ingenious randomized data-structure for concisely representing a set in order to support approximate membership queries. The space efficiency is achieved at the cost of a small probability of false positives. It was invented by Burton Bloom in 1970 for the purpose of spell checking and for many years it was widely mentioned in a variety of large-scale network applications such as shared web caches, query routing, and replica location. For more details about refers to [77].

Search algorithms analyzed so far are classified inside the following comparison table 2.7.

2.6 Uses

$P2P$ networks enable applications such as file-sharing, instant messaging, online multiuser gaming and content distribution over public networks. Distributed storage systems such as NFS (Network File System) provide users



Category	Hybrid	Unstructured		Structured
		Blindly	Informed	
Example	Napster	Gnutella	Int. BFS	Chord
Structure	One Central.	Random	Random	Fixed
Search Method	Centralized Index Server	Flood	Opt Flood	DHT
Info. about Data Loc.	deterministic	Nothing	Partial	High prob.
Data Loc.	Anywhere	Anywhere	Anywhere	Fixed
Self Org.	Bottleneck	Good	Good	Bad

Fig. 2.7. Search in *P2P* comparison

with a unified view of data stored on different file systems and computers which may be on the same or different networks. The domains of *P2P* applications can be subdivided into four categories, particularly distributed computing, file sharing, collaboration, platforms, as summarized in figure 2.8 [27].

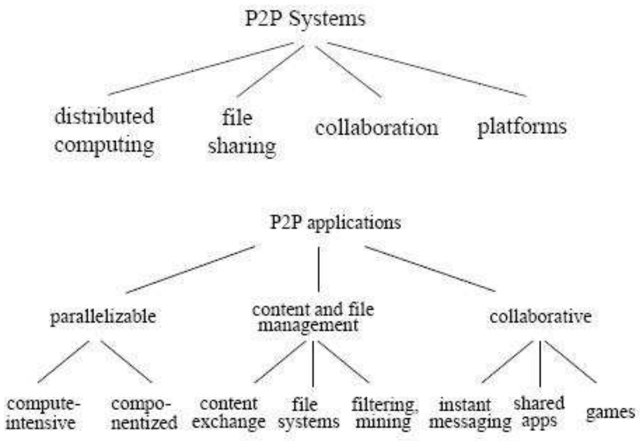


Fig. 2.8. Taxonomy of *P2P* systems and applications

2.6.1 Distributed Computing

These applications use resources from a number of networked computers. The general idea behind these applications is that idle cycles from any computer connected to the network can be used for solving the problems of the other computers that require extra computation. Some distributed computing applications are chat systems (like ICQ, IRC, Jabber, etc.) and PRC applications, see at next chapter.

2.6.2 File sharing

Content storage and exchange is one of the areas where $P2P$ technology has been most successful. File sharing applications [30, 32, 33] focus on storing information on and retrieving information from various Ps in the network. Distributed storage systems based on $P2P$ technologies are taking advantage of the existing infrastructure to offer the following features [27]:

- *File exchange areas:* some file sharing systems, such as Freenet, Gnutella, and Kazaa provide the user with a potentially unlimited storage area by taking advantage of redundancy. A given file is stored on some nodes in the $P2P$ community, but it is made available to any of the Ps. A P requesting a given file just has to know a reference to a file, and is able to retrieve the file from the community by submitting the file reference.
- *Highly available safe storage* The duplication and redundancy policies in some projects, as in Chord, offer virtual storage places where critical files get replicated multiple times, which helps ensuring their availability.
- *Manageability:* $P2P$ systems, as Freenet, enable easy and fast retrieval of the data by distributing the data to caches located at the edges of the network. The location of the data is not known by the retriever, perhaps not even after the data is retrieved.

2.6.3 File sharing: P2P example architectures

One of the best-known example of $P2P$ systems is Napster, it became famous as a music exchange system. Other instances are Gnutella, Freenet, Kazaa, Chord, etc.. $P2P$ systems, using a discovery mechanism based on hash tables, are Chord, CAN and Pastry too. Following each system short description.

Napster

Napster was originally developed to defeat the copying problem and to enable the sharing of music files over the Internet. Napster is a Brokered system, Although search mechanism is centralized, the file sharing mechanism is decentralized. Everyone connected to the central Napster server (may it rest in peace) and told the server what files it had available and how other Ps could reach it. Then when a node is looking for a specific file or P it asks the central server which responds with a listing of files and/or server connection information. At that point the two Ps connect to each other to transfer the file. Napster uses the centralized directory model to maintain a list of

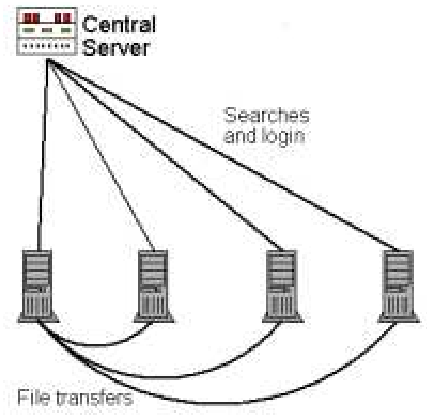


Fig. 2.9. Napster architecture

music files, where the files are added and removed as individual users connect and disconnect from the system. Users submit search requests based on keywords such as "title," "artist,"etc., the actual transfer of files is done directly between the Ps. Napster's centralized directory model inevitably yields scalability limitations and loss in performance.

Gnutella

Gnutella [38] was introduced in March of 2000, by two employees of AOL's Nullsoft division. It is an opensource file sharing program with functionality similar to that of Napster. Gnutella originally was an Equal P Decentralized architecture. Any new node could connect to any existing P in the network and then have access to the network. Once connected to a P it then had access to every P connected to that P, which continues out in a ripple effect until the TTL expires. Such decentralized nature of Gnutella provides a level of anonymity for users, but also introduces a degree of uncertainty.
With time Gnutella evolved to the SP Decentralized architecture. This helped answer the issues of scalability and slow search speeds. With this system a new node connects to one or more SPs and once connected has access to all Ps (and SPs) connected to that SP, continuing out in a ripple patter until the TTL expires. For a user to connect to a Gnutella network they only need to know the address of one other machine on the network. Once that connection is made then that node will discover other nodes on the network until a few connections are made. The Gnutella discovery Protocol include four types of messages:

- *Ping:* a request for a certain host to announce itself.

- *Pong:* reply to a Ping message. It contains the IP and port of the responding host and number and size of files shared.
- *Query:* a search request. It contains a search string and the minimum speed requirements of the responding host.
- *Query hits:* reply to a Query message. It contains the IP and port and speed of the responding host, the number of matching files found and their indexed result set.

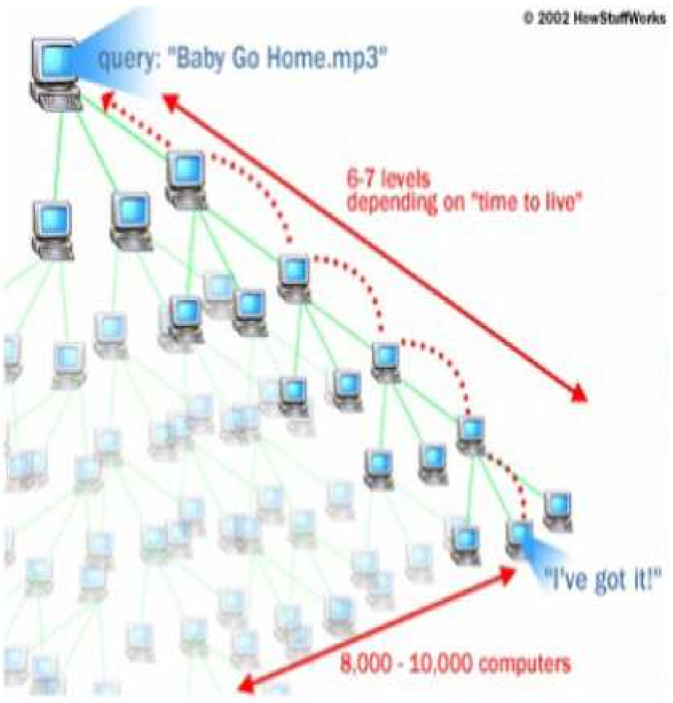


Fig. 2.10. Gnutella Protocol

After joining the Gnutella network(by using hosts such as gnutellahosts.com), a node sends out a Ping message to any node it is connected to. The nodes send back a Pong message identifying themselves, and also propagate the ping to their neighbours. Gnutella originally uses TTL-limited flooding (or broadcast) to distribute Ping and Query messages. At each hop the value of the field TTL is decremented, and when it reaches zero the message is dropped. In order to avoid loops, the nodes use the unique message identifiers to detect and drop duplicate messages.
This approach improves efficiency and preserve network band width. Once a node receives a QueryHit message, indicating that the target file has been

identified at a certain node, it initiates a direct out-of-network download, establishing a direct connection between the source and target node. Although the flooding protocol might give optimal results in a network with a small to average number of Ps, it does not scale well. Furthermore, accurate discovery of Ps is not guaranteed in flooding mechanisms. Also TTL effectively segments the Gnutella network into subsets, imposing on each user a virtual horizon beyond which their messages cannot reach. If on the other hand the TTL is removed, the network would be swamped with requests. One more problem is that the Gnutella protocol itself does not provide a fault tolerance mechanism. The hope is that enough nodes will be connected to the network at a given time such that a query will propagate far enough to find a result.

Freenet

The Freenet system [40, 41] was conceptualized by Ian Clarke in 1999 while at the University of Edinburgh and its implementation began in early 2000. The primary mission of Freenet is to make use of the system anonymous that means provide storage and use the system without being possible identify who determined who placed a file into the system and who made a request. The idea is that FreeNet is creating a network where individuals can post their opinion without the fear of their identity being revealed. And once their opinion is posted, it will remain available as long as people are downloading it. In Communication architecture Freenet is similar to Gnutella in communication architecture, it's a completely decentralized system and it represents the purest form of $P2P$ system. One optimization is that each time a file is requested a copy is made on the nodes closer to the requesting node. This makes it more convenient the next time it is requested from the same location, in fact Freenet is considered responsive as possible. In this system each P from the network is assigned a random ID and each P also Every node in the Freenet network maintains a set of files locally up to the maximum disk space allocated by the node operator. When all disk space is consumed, files are replaced in accordance with a least recently used (LRU) replacement strategy. Freenet's basic unit of storage is a file. Each file shared on such system, is identified by an ID. These are typically generated using the hash SHA-1 Function [43]. Each P will then route the document towards the P with the ID that is most similar to the document ID. This process is repeated until the nearest P ID is the current $P's$ ID. Each routing operation also ensures that a local copy of the document is kept. When a P requests the document from the $P2P$ system, the request will go to the P with the ID most similar to the document ID. This process is repeated until a copy of the document is found. Then the document is transferred back to the request originator, while each P participating the routing will keep a local copy.

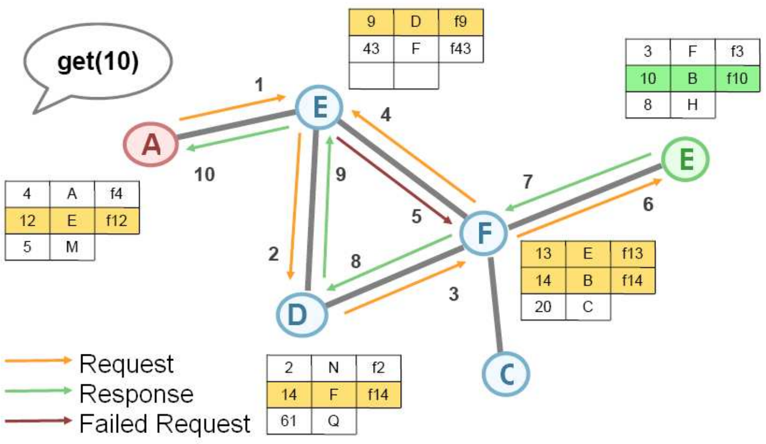


Fig. 2.11. Freenet: searching for data

The scalability of Freenet has been studied by its authors using extensive simulation studies [42]. Their studies support the hypothetical notion that route lengths grow logarithmically with the number of users.

Chord

Chord [44] uses a decentralized $P2P$ lookup protocol that stores key/value pairs for distributed data items. Given a key, it maps key a node responsible for storing the key's value. In the steady state, in an N-node network, each node maintains routing information about $O(logN)$ other nodes, and resolves all lookups via $O(logN)$ messages to other nodes. Updates to the routing information for nodes leaving and joining require only $O(log^2N)$ messages.

CAN: Content Addressable Networks

CAN [46] is a mesh of N nodes in virtual d-dimensional dynamically partitioned coordinate space. Each P keeps track of its neighbours in each dimension. When a new P joins the network, it randomly chooses a point in the identifier space and contacts the P currently responsible for that point. The contacted P splits the entire space for which it is responsible into two pieces and transfers responsibility of half to the new P, the new P also contacts all of the neighbours to update their routing entities. The CAN discovery mechanism consists of two core operations namely, a local hash-based look-up of a pointer to a resource, and routing the look-up request to the pointer. The CAN algorithm guarantees deterministic discovery of an existing resource in $O(N^{\frac{1}{d}})$ steps.

Pastry

An approach similar to Cord was also used in Pastry [45]. In the Pastry each node network has a unique identifier (nodeId) from a 128-bit circular index space. The pastry node routes a message to the node with a nodeId that is numerically closest to the key contained in the message, from its routing table of O(logN), where N is the number of active Pastry nodes. The expected of routing steps is O(logN). Pastry takes into account network locality; it seeks to minimizes the distance messages travel, according to a scalar proximity metric like the number of IP routing hops.

Kazaa (FastTrack)

Kazaa [49] is a Hybrid system that uses SuperNodes as local search hubs for lookup. Each SP role reminds to the role of the central server in Napster, except that here it is limited to a small part of the network. SP are nodes chosen inside the network because better in elaboration power, bandwidth, average time being connected on the net, respect others. Kazaa uses an intelligent download system to improve download speed and reliability. Each user send its files list to a SP node then SPs periodically send lists each others. The advantage is that when there is a file request, the system automatically finds and downloads files from the fastest connections, failed transfers are automatically resumed, and files are even downloaded from several sources simultaneously to speed up the download. When files are imported, the system automatically extracts meta-data from the contents of the files (such as ID3 tags for mp3 files). This makes for much faster andmore accurate searches.Kazaa also uses a technique called MD5 hashing to make sure the contents of multi-sourced files are identical.

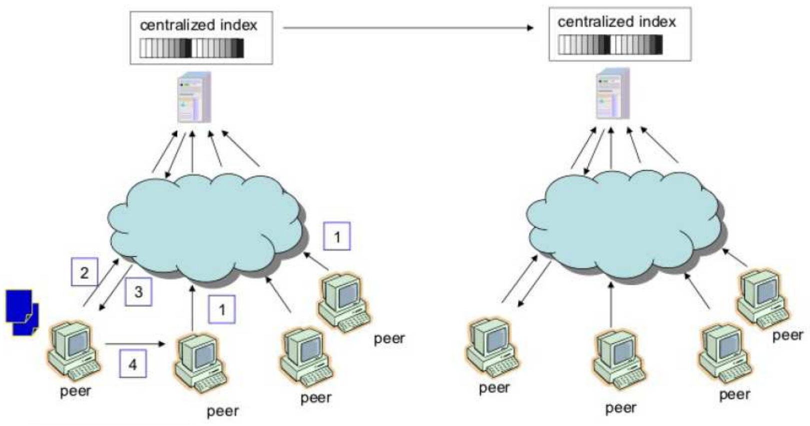


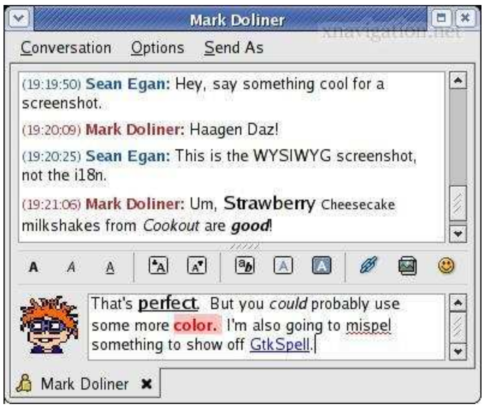
Fig. 2.12. Kazaa hybrid model

WASTE

WASTE [50] is a mesh-based workgroup tool that allows for RSA encrypted communication between small groups workgroups of users. The network is actually a partial mesh, with every possible connection made, limited by firewalls and routers. Communication is then routed over the network along the route of lowest latency, which allows communication between firewalled Ps via a non firewalled P.

2.6.4 Collaboration

Intuitively *Collaboration* is a new way to intend communication. If you wish chat with your friends over the network you shouldn't need to subscribe to a central server. The regulation of members, content and connections has to be determined by the members, instead of a service provider. Whenever you want start a project with a few friends, sharing project files, discussion boards, white boards, chat sessions and other necessities inside a collaborative environment, you should have the chance to communicate directly. Collaborative $P2P$ applications then aim to allow *application level collaboration* between users.



(19:19:50) **Sean Egan:** Hey, say something cool for a screenshot.

(19:20:09) **Mark Doliner:** Haagen Daz!

(19:20:25) **Sean Egan:** This is the WYSIWYG screenshot, not the i18n.

(19:21:06) **Mark Doliner:** Um, **Strawberry** Cheesecake milkshakes from *Cookout* are ***good***!

That's **perfect**. But you *could* probably use some more **color.** I'm also going to mispel something to show off GtkSpell.

Fig. 2.13. Gaim: a client for instant messaging

Collaborative applications are generally event-based. Ps form a group and begin a given task. The group may include only two Ps collaborating directly, or may be a larger group. When a change occurs at one P (e.g., that P initiates sending a new chat message), an event is generated and sent to the rest of the group. At the application layer, each P's interface is updated accordingly. There are a number of *technical challenges* that make implementation of this type of system difficult. Like other classes of $P2P$ systems, *location* of other Ps is a challenge for collaborative systems. Many systems rely on centralized directories that list all Ps who are online, refers to figure 2.14. To form a new group, Ps consult the directory and select the Ps they wish to involve.

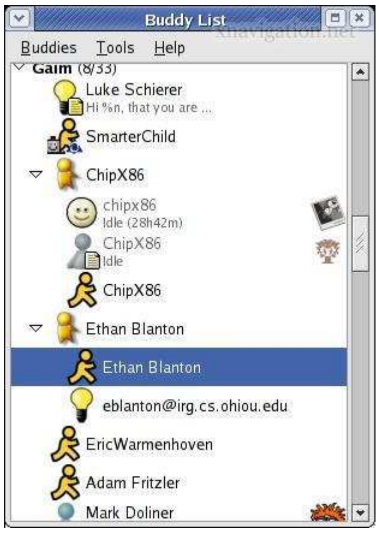


Fig. 2.14. Buddy list

Other systems, like Microsoft's NetMeeting, can require that Ps identify one another by IP address. This is much too restrictive, especially in environments where groups are large.

Fault tolerance is another challenge. In shared applications, messages often must be delivered reliably to ensure that all Ps have the same view of the information. In some cases, message ordering may be important. While many well-known group communication techniques address these challenges in a non-$P2P$ environment, most $P2P$ applications do not require such strict guarantees. The primary solution employed in $P2P$ applications is to queue messages that have been sent and not delivered (i.e., because a given P is down or offline). The messages can then be delivered to the offline P when it comes back online. *Realtime constraints* are perhaps the most challenging aspect of collaborative implementations. Users are the ultimate end points in a collaborative environment. As such, any delay can be immediately perceived by the user. Unfortunately, the bottleneck in this case is not the $P2P$ technology, but the underlying network. While many collaborative applications may work well in a local- area systems, wide-area latencies limit $P2P$ applications just as they limit client-server applications. Generally applications range from instant messaging and chat, to on line games, to shared applications that can be used in business, educational, and home environments. The gaming environment is the one in which real time is the a constraint; The game DOOM is a so-called First Person Shooter (FPS) game in which multiple players can collaborate or compete in a virtual environment. DOOM uses a $P2P$ structure in which each player's machine sends updates of the state of the environment (such as the player's movement) to each of the other machines. Only when all updates have been received does the game update the view. This was marginally viable in local-area, small-scale games, but did not scale to wide-area games. Long latencies and uneven computing power at the various players machines made this lock-step architecture unusable. All FPS games since DOOM have used amore standard client-server architecture for communication. Jabber [47] is a set of streaming XML protocols and technologies that enable any two entities on the Internet to exchange messages, presence, and other structured information in close to real time. Groove [48] provides a variety of applications for communication, content sharing (files, images and contact data), and collaboration (i.e. group calendaring, collaborative editing and drawing, and collaborative Web browsing).

2.6.5 Platforms

In terms of development, platforms such as JXTA [73], XtremWeb [74], Microsoft's .NET My Services and BOINC provide an infrastructure to support $P2P$ applications. For example JXTA(TM) is based on Java technology and it is a set of open protocols that enable any connected device on the network, ranging from cell phones and wireless PDAs to PCs and servers, to communicate and collaborate in a $P2P$ manner. JXTA Ps create a virtual network where any P can interact with other Ps and resources directly, even when some of the Ps and resources are behind firewalls and network address translations (NATs) or on different networks. A detailed description about the platform BOINC is given in next chapter.

2.7 Bittorrent

BitTorrent is a $P2P$ application that was invented because to facilitate fast downloads of popular files from the Internet in opposite then centralized download systems as in figure 2.15.
Thanks BitTorrent mechanism, each client can allow other clients to download from it, data that it had already downloaded. Clients are available to share their just downloaded files because they are foster to do that *(+upload = +download !)* and in such way the *free-riding* is discouraged.
To understand how BitTorrent works, we describe the scenario of how it operates when a single file is downloaded by many users. Typically the number of simultaneous downloaders for popular files could be of the order of a few hundreds while the total number of downloaders during the lifetime of a file could be of the order of several tens or sometimes even hundreds of thousands.

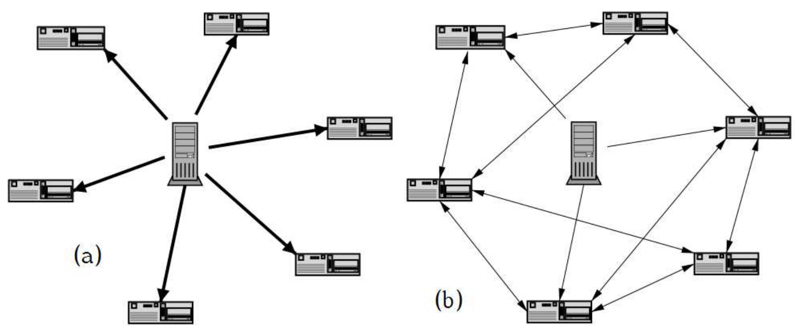


Fig. 2.15. Centralized download (a) vs. BitTorrent mechanism (b)

The basic idea in BitTorrent is to divide a single large file (typically a few 100 MBytes long) into pieces of size 256 KB each. For each file there is a corresponding description file, with *.torrent* extension, with segments number, segments hash code and the server Tracker address. The Tracker is a centralized software that store all active Ps on a file, since the set of Ps attempting to download the file do so by connecting to several other Ps simultaneously and download different pieces of the file from different Ps. Clients report information to the tracker periodically and in exchange receive information about other clients that they can connect to. The tracker is not directly involved in the data transfer and does not have a copy of the file.

In a BitTorrent network, a P that wants to download a file first connects

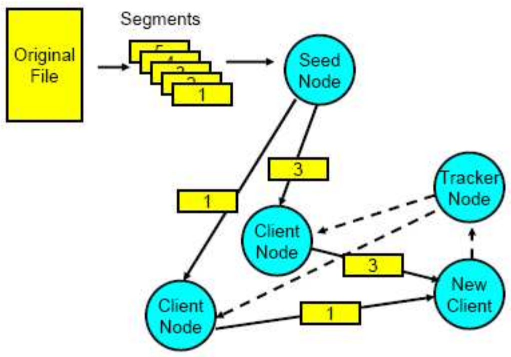


Fig. 2.16. BitTorrent segment files flow

to the tracker of the file. The tracker then returns a random list of Ps that have the file. The downloader then establishes a connection to these other Ps and finds out what pieces reside in each of the other Ps. A downloader then requests pieces which it does not have from all the Ps to which it is connected. But each P is allowed to upload only to a fixed number (default is four) at a given time. Uploading is called unchoking in BitTorrent. Which Ps to unchoke is determined by the current downloading rate from these Ps, i.e., each P uploads to the four Ps that provide it with the best downloading rate even though it may have received requests from more than four downloaders.

To allow each P to explore the downloading rates of other Ps, BitTorrent uses a process called optimistic unchoking. Under optimistic unchoking, each P randomly selects a fifth P from which it has received a downloading request and uploads to this P. Thus, including optimist unchoking, a P may be uploading to five other Ps at any time. Optimistic unchoking is attempted once every 30 seconds and to allow optimistic uncloaking while keeping the maximum number of uploads equal to five, an upload to the P with the least downloading rate is dropped.

BitTorrent distinguishes between two types of Ps, namely downloaders and seeds. Downloaders are Ps who only have a part (or none) of the file while seeds are Ps who have all the pieces of the file but stay in the system to allow other Ps to download from them. Thus, seeds only perform uploading while downloaders download pieces that they do not have and upload pieces that they have. Ideally, one would like an incentive mechanism to encourage seeds to stay in the system. However, BitTorrent currently does not have such a feature. We simply analyze the performance of BitTorrent as is.
In practice, a BitTorrent network is a very complicated system. There may be hundreds of Ps in the system. Each P may have different parts of the file. Each P may also have different uploading/downloading bandwidth. Globally this allows a better utilization of the available bandwidth (even download at 7MB/sec). At the same time this approach is not suitable for small dimension files and the Tracker represent the single point of failure of the network and it limits scalability. One more drawback is about security in fact inside this download system, all client nodes are visible and they can't hide their identity. Further, each P only has partial information of the whole network and can only make decisions based on local information. Anyway to deal with this BitTorrent has a protocol (called the rarest-first policy) to ensure a uniform distribution of pieces among the Ps and protocols (call the endgame mode) to prevent users who have all but a few of the pieces from waiting too long to finish their download [53, 54].

2.8 Super Peer

A special $P2P$ architecture is the *Super Peer (SP)* model [55]. It corresponds to a hybrid between brokered and decentralized topology, but would still be considered a sub-type to decentralized.
In this network there are P and SP nodes. A SP is a node that operates

both as a server to a set of clients. A node to be designed as *SP*, typically must have some special environmental advantages, it's always connected and specially it must have good connection speed, high visibility and long uptime.

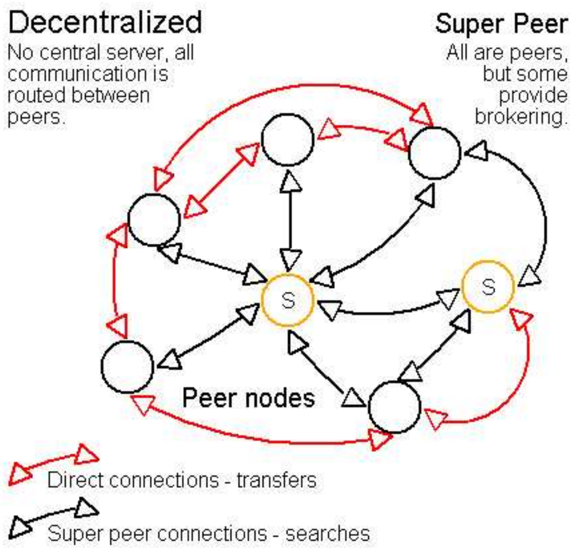

Decentralized
No central server, all communication is routed between peers.
Super Peer
All are peers, but some provide brokering.
S
S
Peer nodes
Direct connections - transfers
Super peer connections - searches

Fig. 2.17. Decentralized and super peer models

The network resulting from the connection of all *SP* nodes operates exactly like a pure *P2P* network, that catchs both the inherent efficiency of centralized search, and the autonomy, load balancing and robustness to attacks provided by distributed search.

We call a *SP* and its clients a cluster, where cluster size is the number of nodes in the cluster, including the *SP* itself.

To better understand how this protocol works, we analyze *P* discovery phase and resource discovery and querying, then some optimization introduced in the model.
P2P systems like KaZaA and Gnutella are adopting *SP*s in their design.

2.8.1 Peer discovery and network creation

The startup phase of such network generation is quite easy. From the *P* point of view it must know just about the address of the *SP* of the cluster it will be added to, after it connects it will share with its cluster's *SP* all information

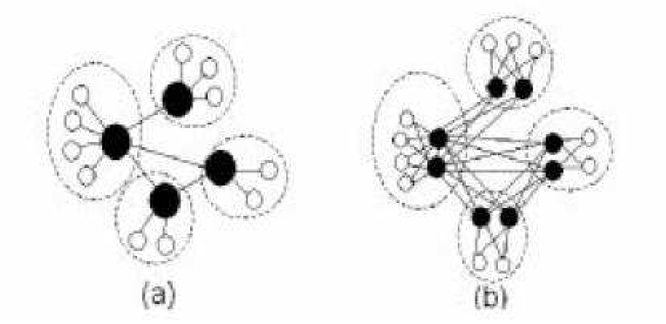


Fig. 2.18. Figure 1: Illustration of a SP network (a) with no redundancy, (b) with 2-redundancy. Black nodes represent SPs, white nodes represent clients. Clusters are marked by the dashed lines.

related to resource it offers for sharing.

The SP otherwise must know the address of its neighbour SPs to join the SPs net. Then he must discover its neighbour SP, every typical mechanism of discovery inside pure network can be used for this purpose.

2.8.2 Resource discovery and querying inside a SP network

Since each client belongs just to a unique SP, once clients submit queries to their SP they receive results just from it, as in a hybrid system.
When a SP receives a query from a neighbour, it will process the query on its clients' behalf, rather than forwarding the query to its clients, thanks a SP keeps an index over its clients' data.

As soon as the SP finds any results, it will return one *Response message.* This Response message contains the results, and the address of each client whose collection produced a result. When a client wishes to submit a query to the network, it will send the query to its SP only. The SP will then submit the query to its neighbours as if it were its own query, and forward any *Response messages* it receives back to the client. Outside of the cluster, a client's query is indistinguishable from a SP's query. All the query processing and traffic involves just SP nodes then Ps processing power is all dedicated at the jobs execution and the whole system runs efficiently.

2.8.3 SP networks optimizations: redundancy

Even if the depicted scenario seems to be efficient, in the case a SP fails or simply leaves, all its clients become temporarily disconnected until they can find a new SP to connect to, this means that a SP can be a potential bottleneck for its cluster, in fact it represents the single point of failure.

The natural idea, detailed in [55], to provide reliability and performance to the cluster and decrease the load on the SP, it's to introduce redundancy into the design of the SP.

A SP is defined to be k-redundant if there are k nodes sharing the SP load, forming a single virtual SP.
Figure 2.19 illustrates a SP network topology with redundancy $k = 2$.

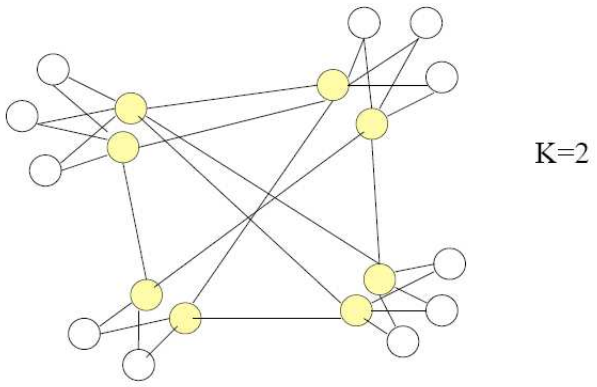


Fig. 2.19. A SP network topology with redundancy $k = 2$

In short every node in the virtual SP is a *partner* with equal responsibilities: each partner is connected to every client and has a full index of the data of clients, as well as the data of other partners. Clients send queries to each partner in a round-robin fashion; similarly, incoming queries from neighbours are distributed across partners equally. Hence, the incoming query rate on each partner is a factor of k less than on a single SP with no redundancy, though the cost of processing each query is higher due to the larger index.

Since all partners can respond to queries, if one partner fails, the others may continue to service clients and neighbours until a new partner can be found. Obviously the probability that all partners will fail before any failed partner can be replaced is much lower than the probability of a single SP failing.

The drawback of this approach is that:

- A client must send metadata to each of these partners when it joins, in order for each partner to have a full index with which to answer queries.
- The aggregate cost of a client join action is k times greater than before.
- Neighbours must be connected to each one of the partners, so that any partner may receive messages from any neighbour.
- The number of open connections amongst SPs increases by a factor of k^2, in the case that every SP in the network is k2-redundant.

At first glance, SP redundancy seems to trade off reliability for cost, anyway SP redundancy actually has the surprising effect of reducing load on each SP, in addition to providing greater reliability [55].

3 Public-Resource Computing

Nowadays scientific supercomputing requires high computer power and disk space for massive data storage that even can't find inside supercomputer centers and institutional machine rooms.
In this chapter a new distributed approach is analyzed: *Public-resource computing (PRC)*(also known as Global Computing or Peer-to-peer computing). It mainly consists in using the idle CPU time of hundreds of millions of personal computers currently spread all over the world and, in a few more years, from other consumer devices like game consoles and television set-top boxes. The idea of using those unused resources was first proposed in 1978 by the Worm computation project at Xerox PARC. They used 100 computers to measure the performance of Ethernet there. Many academic projects followed to explore this approach including Condor, a toolkit developed at the University of Wisconsin for writing programs that run on unused workstations, typically within a single organization.
Public-resource computing emerged in the mid-1990s with two projects, GIMPS [61]and Distributed.net [62], after the Internet expanded to the consumer market and there were millions of fast computers connected by a network In 1999, a scientific search project, SETI (Search for Extraterrestrial Intelligence)@home [67], was launched, with the goal of detecting radio signals emitted by intelligent civilizations outside Earth [68]. SETI@home acts as a "screensaver", running only when the PC is idle, and providing a graphical view of the work being done. SETI@home's appeal extended beyond hobbyists; it attracted millions of participants from all around the world. It inspired a number of other academic projects, as well as several companies that sought to commercialize the public computing paradigm.

Large-scale public-resource projects can't work without a middleware providing necessary functionality as client and server software, management tools, user-centered web features, and so on, for this purpose in last section in this chapter is detailed the middleware Berkeley Open Infrastructure for Network Computing (BOINC) [72] that solves or helps solve most of these problems.

Finally social implication of *PRC* are analyzes, specially its positive effects to encourage private user awareness of current scientific research and it catalyzes global communities centered around scientific interests, and it gives the public a measure of control over the directions of scientific progress.

3.1 PRC model

In *PRC* scenario a large numbers of computers, volunteered by members of the general public, provides computing and storage resources [57]. Inside a *PRC* application, scientific project jobs are executed by privately-owned and often donated computers offering their unused idle CPU time. The point is to get anybody with an Internet connection and spare compute power to donate CPU cycles on their computer. This leads to a very heterogeneous distributed model, because the network donator users and the donating machine types from the performance and the hardware architecture point of view heterogeneity. The mechanism of using compute power that would otherwise go to waste is often called cycle-scavenging or cycle-harvesting [82].

3.2 PRC problem features

To be amenable to public computing, a task must be divisible smaller independent pieces whose ratio of computation to data is high. A long computation causes low network traffic. This is necessary to keep server traffic at a manageable level. A critical point is to coordinate the work of different clients with possible frequent disconnection; many data dependencies in fact prevent an efficient and self-sufficient work of the client so applications should be capable of independent parallelism. Tasks should also be capable of tolerating errors. A client may produce an error and return a wrong result or a malicious user sends wrong results however the project should not be negatively affected by this [57]. Anyway for this purpose the solution can be calculating redundant results. Many types of computations have these properties:

1. Complex physical systems have a random and chaotic component. Their outcome is probabilistic, not exact. Studying the statistics of this outcome requires running large numbers of simulations with different random initial and boundary conditions. These simulations can be run in parallel.
2. There is an evolving field of "random algorithms" that provide approximate solutions to exact problems. These often involve random trials that can run in parallel.
3. "Genetic algorithms" are applicable to many areas. This approach involves creating a population of approximate solutions to a problem, and using the mechanisms of natural selection to approach an optimal solution.

4. Models of physical systems often have large numbers of underlying parameters whose optimal values are not known, and which combine nonlinearly. Exploring such parameter spaces requires large numbers of independent simulation runs. More generally, "Monte Carlo" algorithms involve large numbers of independent computations, corresponding to sampling in a high-dimensional space.
5. Applications that involve analyzing large amounts of data, such as data from a radio telescope (e.g., SETI@home) or from a particle accelerator, have inherent parallelism. The limiting factor is the computation-to-data ratio.
6. Some medical projects involve searching a set of millions or billions of molecules (for example, searching for potential drugs). These tasks are easily parallelized. Similarly some genetics projects involve matching a set of proteins with a DNA sequence; again, this is easily parallelized.

3.3 PRC projects

Large-scale public-resource computing became feasible with the growth of the Internet in the 1990s. Examples of volunteer computing paradigma implementations include searching for extraterrestrial life), high-energy physics, molecular biology, medicine, astrophysics, climate study, and other areas [57]. Early projects emerged in 1997: Great Internet Mersenne Prime (GIMPS) [61], which searched for large prime numbers, and Distributed.net (d.net) [62], which deciphers encrypted messages. These project attracted thousands of participants. Anyway the pioneer project in this real is SETI@home, which has attracted millions of participants wishing to contribute to the digital processing of radio telescope data in the search for extra-terrestrial intelligence. In 1999, SETI (Search for Extraterrestrial Intelligence)@home [67], aimed at building a huge virtual computer based on the aggregation of the computer power offered from internetconnected computers during their idle periods, has attracted millions of participants worldwide. The project uses two major components: the database server and the client. Clients can help with search for extra-terrestrial life by running the search program for a specified portion of the universe. This project strongly relies on its server to distribute jobs to each participating peer and to collect results after processing is done.
More recent projects include Folding@home and the Intel-United Device Cancer Research Project.
Both belong to genetic field, especially The Folding@home project [63] is dedicated to understanding protein folding, the diseases which result from protein misfolding and aggregation, and novel computational ways to develop new drugs in general.
A number of similar projects has supported today by the BOINC [72] sw infrastructure. The range of scientific objectives amongst these projects is very diverse, ranging from Einstein@home's [64], aiming at the detection of certain

types of gravitational waves to Climate@home's [65], which focuses on long-term climate prediction

How one might conduct massively distributed search for gravitational waveforms produced by binary stars orbiting one around the other is the scenario defined for the GridOneD project [66].
In this case we have a data-intensive Grid application that in general can require the distributed execution of a large number of jobs with the goal to analyze a set of data files. In this scenario, a data file of about 7.2 MB of data is produced every 15 minutes and it must be compared with a large number of templates (between 5,000 and 10,000) by performing fast correlation. Data can be analyzed in parallel by a number of Grid nodes to speed up computation and keep the pace with data production.
A new opensource project is PS3GRID, it is a volunteer computing project based on the PlayStation3 and BOINC for full-atom molecular dynamics simulations and other scientific applications specially optimized for the Cell processor [14].

The success of these projects is a proof that *PRC* can provide more computing power than any supercomputer, cluster, or grid, and the disparity will grow over time. The most important project, SETI@home, currently runs on about 1 million computers. This provides a processing rate of 60 TeraFLOPS (trillion floating-point operations per second). In contrast, the largest conventional supercomputer, the IBM ASCI White, provides about 12 TeraFLOPs. SETI@home's 1 million computers represents a tiny fraction of the approximately 150 million Internet-connected PCs worldwide. The latter number is projected to grow to 1 billion by 2015. Thus public computing has the potential to provide many PetaFLOPs of computing power.
Moore's Law asserts that the speed of CPU chips doubles about every 18 months. The rate of progress is even faster for "graphics coprocessors", the chips that handle 3D graphics in PCs and game consoles. Their doubling time is about 8 months, and current graphics chips have a raw floating-point arithmetic speed many times that of their host CPU. These graphics chips are becoming more programmable and flexible, and researchers are actively investigating their use for scientific computing. Because graphics chips are integrated in modern personal computers, this trend favors public computing over other paradigms.
Most computational tasks require storage (disk space) as well as computing. Here also, public resources can provide unprecedented capacity. Today, a typical PC provides about 80 Gigabytes of storage space, which in most cases is more than is used the PC owner. If 100 million computer users were each to provide 10 Gigabytes of storage, the total would be an Exabyte (10 to the 18th power) - greater than the capacity of any centralized storage system.

3.4 SETI@home

Currently several programs are looking for the evidence of life elsewhere outside of earth, the way is looking for some radio signals coming from our alien neighbours trying to contact us. Collectively, these programs are called SETI (the Search for Extra-Terrestrial Intelligence.).
One earlier, known as radio SETI, uses radio telescopes to listen for narrow-bandwidth radio signals from space. Such signals are not known to occur naturally, so a detection would provide evidence of extraterrestrial technology [69].
Radio telescope signals consist primarily of noise (from celestial sources and the receiver's electronics) and man-made signals such as TV stations, radar, and satellites. Modern radio SETI projects analyze the data digitally. This analysis generally involves three phases:

1. Compute the data's time-varying power spectrum.
2. Find candidate signals using pattern recognition on the power spectra.
3. Eliminate candidate signals that are probably natural or man-made.

More computing power enables searches to cover greater frequency ranges with more sensitivity. Radio SETI, therefore, has an insatiable appetite for computing power.
Radio SETI projects, as most of the SETI programs in existence today, build large computers able to analyze data from a telescope in real time. None of these computers look very deeply at the data for weak signals nor do they look for a large class of signal types. The reason for this is because they are limited by the amount of computer power available for data analysis.
To tease out the weakest signals, a great amount of computer power is necessary. It would take a monstrous supercomputer to get the job done. Rather than a huge computer to do the job, they could use a smaller computer but just take longer to do it. But then there would be lots of data piling up.
The solution ideally take the best of both approaches, in *PRC* manner: use LOTS of small computers, all working simultaneously on different parts of the analysis. This is possible because fortunately, the data analysis task can be easily broken up into little pieces that can all be worked on separately and in parallel.
In 1995 David Gedye the project manager at Starwave Corp. proposed doing radio SETI using a virtual supercomputer consisting of large numbers of Internet-connected PCs. SETI@home was born to explore this *PRC* idea.
The first challenge for SETI@home was to find a good radio telescope. The choose was Arecibo, Puerto Rico, the world's largest and most sensitive radio telescope. Arecibo is used for various astronomical and atmospheric research, but it's not possible to obtain its long-term exclusive use. However, in 1997 the U.C. Berkeley SERENDIP project developed a technique for piggybacking a secondary antenna at Arecibo [70]. As the main antenna tracks a fixed point in the sky (under the control of other researchers), the secondary antenna

traverses an arc that eventually covers the entire band of sky visible to the telescope. This data source can be used for a sky survey that covers billions of stars.
Unlike SERENDIP, SETI@home project need to distribute data through the Internet. Since in the beginning Arecibo's Internet connection was a 56 Kbps modem, U.C. Berkeley decided to record data on removable tapes (35GB DLT cartridges, the largest available at the time), mailed them from Arecibo to their laboratories and distribute data from servers there.
In short the SETI@home software makes use of a computationally intensive algorithm called "coherent integration" and Volunteer user computers perform fast fourier transforms on the data, looking for strong signals at various combinations of frequency, bandwidth, and chirp rates.
SETI@home collected data recording them at 5 Mbps from Arecibo telescope. It used 35 GB digital linear tapes. The recording time per tape is 16 hours. With one-bit complex sampling this yields a frequency band of 2.5 MHz which is enough to handle doppler shifts for relative velocities of up to 260 km/sec (or about the rate of the Milky Ways galactic rotation). The frequency-band is like many other SETI-projects centered at the Hydrogen-line (1.42 GHz) because man-made transmissions are forbidden here by an international treaty.

SETI@home computational model

SETI@home fits well the *PRC* paradigma. SETI@home data have a high computing-to-data ratio: each unit takes 3.9 trillion floating-point operations, or about 10 hours on a 500 MHz Pentium II, yet involves only a 350KB download and a 1 KB upload. This high ratio keeps server network traffic at a manageable level, and imposes minimal load on client networks. Applications such as computer graphics rendering require large amounts of data per unit computation, perhaps making them unsuitable to public-resource computation. However, reductions in bandwidth costs will allay these problems, and multicast techniques can reduce cost when a large part of the data is constant across work units.
Secondly, tasks with independent parallelism are easier to handle. SETI@home work unit computations are independent, so participant computers never have to wait for or communicate with one another. If a computer fails while processing a work unit, the work unit is eventually sent to another computer. Applications that require frequent synchronization and communication between nodes have been parallelized using hardware-based approaches such as shared-memory multiprocessors, and more recently via software-based cluster computing, such as PVM [76]. Public-resource computing, with its frequent computer outages and network disconnections, seems ill-suited to these applications. However, scheduling mechanisms that find and exploit groups of LAN-connected machines may eliminate these difficulties.
Thirdly, tasks that tolerate errors are more amenable to public-resource com-

puting. For example, if a SETI@home work unit is analyzed incorrectly or not at all, it affects the overall goal only slightly. Furthermore, the omission is remedied when the telescope scans the same point in the sky.
Theorically SETI@home's computational model is simple. The signal data is divided into fixed-size work units distributed via the Internet to the clients. The client program computes a result (a set of candidate signals) and returns it to the server. There is no communication between clients.
SETI@home employs redundant computation. Each work unit is processed multiple times to compensate the detection and discard of results of faulty processors or malicious users. A redundancy level of 2 or 3 is adequate for this.
Work units are formed by dividing the 2.5 MHz signal into 256 frequency bands. Each band is then divided into 107-second segments overlapping in time by 20 seconds. As SETI@home looks for continues signals with a length of up to 20 seconds the overlapp ensures that each signal is contained in at least one work unit. The resulting work units are 350 KB. Enough to keep a client busy for a while and small enough to be transfered in a matter of minutes even by a 56K Modem.
The resulting architecture system keeps under consideration that the source of failure for the project must be minimized so the distribution of data must be allowed even when the database is down and dependencies between server subsystem must be minimized, the resulting architecture is the one in figure 3.1.
The task of creating and distributing work units is done by a *server complex* located in U.C. Berkley laboratorier.

The server complex contains a relational database which stores information about tapes, work units, results, users and other aspects of the project. A multi-threaded data/result server handles distribution of work units to clients. HTTP is used so that users can participate in SETI@home even when connecting from behind a firewall. A garbage collector program removes work units from disk clearing an on-disk flag in their database records. Work units are flaged depending on how many correct results were received from the clients versus the intended level of redundancy.

The client program repeatedly gets a work unit from the data/result server, analyzes it, and returns the result (a list of candidate signals) to the server. It needs an Internet connection only while communicating with the server. The client can be configured to compute only when its host is idle, or to compute constantly at a low priority. The program periodically writes its state to a disk file, and reads this file on startup; hence it makes progress even if the host is frequently turned off.
Computational work of the clients deals with analyzing a work unit. This involves computing signal power as a function of frequency and time, then looking for several types of patterns in this power function: spikes (short bursts), Gaussians (narrow-bandwidth signals with a 20-second Gaussian envelope,

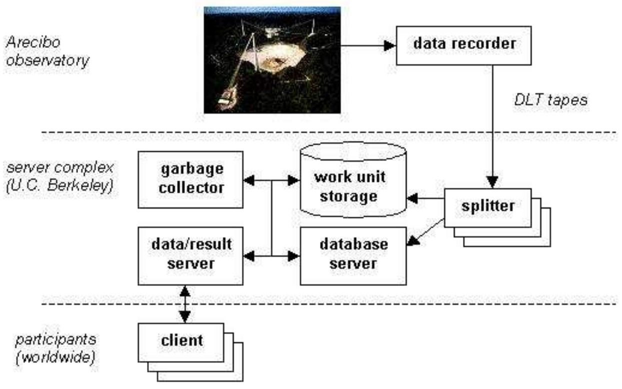
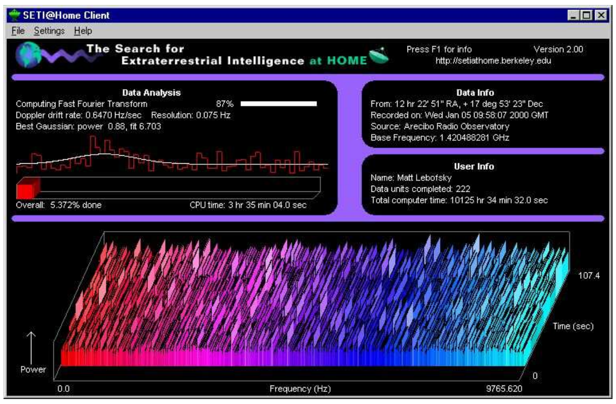


Fig. 3.1. The distribution of data



Data Analysis
Computing Fast Fourier Transform 87%
Doppler drift rate: 0.6470 Hz/sec Resolution: 0.075 Hz
Best Gaussian: power 0.88, fit 6.703
Overall: 5.372% done CPU time: 3 hr 35 min 04.0 sec

Data Info
From: 12 hr 22' 51" RA, + 17 deg 53' 23" Dec
Recorded on: Wed Jan 05 09:58:07 2000 GMT
Source: Arecibo Radio Observatory
Base Frequency: 1.420488281 GHz

User Info
Name: Matt Lebofsky
Data units completed: 222
Total computer time: 10125 hr 34 min 32.0 sec

Fig. 3.2. The SETI@home display, showing the power spectrum currently being computed (bottom) and the best-fit Gaussian (left).

corresponding to the telescope's beam movement across a point), pulsed signals (Gaussian signals pulsed with arbitrary period, phase, and duty cycle), and triplets (three equally-spaced spikes at the same frequency; a simple pulsed signal). Signals whose power and goodness-of-fit exceed thresholds are recorded in the output file.

The SETI@home client program is written in C++. The code consists of a platform-independent framework for distributed computing (6,423 lines), components with platform-specific implementations, such as the graphics library (2,058 lines in the UNIX version), SETI-specific data analysis code (6,572 lines), and SETI-specific graphics code (2,247 lines).
The client has been ported to 175 different platforms. The GNU tools, including gcc and autoconf, have greatly facilitated this task.

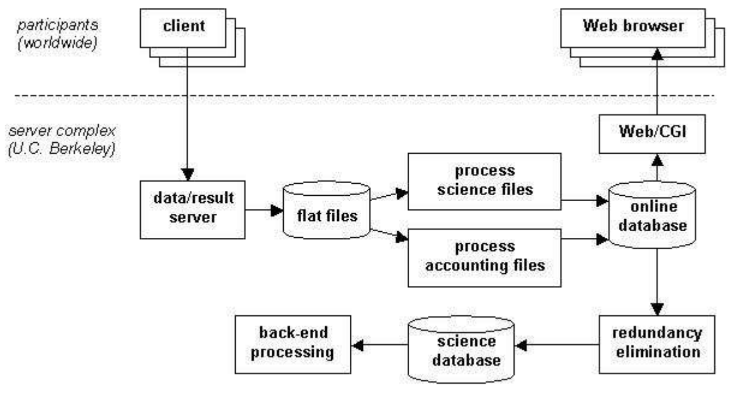


Fig. 3.3. The collection and analysis of results.

The client can run as a background process, as a GUI application, or as a screensaver. To support these different modes on multiple platforms, it used an architecture in which one thread does communication and data processing, a second thread handles GUI interactions, and a third thread (perhaps in a separate address space) renders graphics based on a shared-memory data structure.
Results are returned to the SETI@home server complex, where they are recorded and analyzed (see Figure 3.3).

About Scientific data, the data/server writes the result to a disk file. A program reads these files, creating result and signal records in the database. To optimize throughput, several copies of this program run concurrently. About Accounting information, for each result collected by the web browser, the server writes a log entry describing the result's user, its CPU time, and so on. A program reads these log files, accumulating in a memory cache the updates to all relevant database records (user, team, country, CPU type, and so on). Every few minutes it flushes this cache to the database.
A redundancy elimination program examines each group of redundant results and choose a "canonical" result for each work unit and they are copied to a separate database.
The final phase, back-end processing, consists of several steps. To verify the system, there is a check for the test signals injected at the telescope. Man-made signals (RFI) are identified and eliminated. The system look for signals with similar frequency and sky coordinates detected at different times. These "repeat signals", as well as one-time signals of sufficient merit, are investigated further, potentially leading to a final cross-check by other radio SETI projects according to an accepted protocol [59, 68].

Project results

In 1998 plans for SETI@home were announced. About 400.000 people pre-registered in the following year. In May 1999 Windows and MAC version were released and within a week 200.000 people downloaded and installed the client. In August 2002 SETI@home counted 3.91 million participants in 226 countries. Back then 50% of the users were from the USA and 71% described themselves as home users.
In the 12 months beginning July 2001 221 million work units were processed with an average throughput in that period of 27.35 Teraflops.
On the 25-Oct-2005, according to statistics of a
http://www.boincsynergy.com/stats/index.php BOINC website, SETI@home (running on BOINC, not counting the classic client) had 232.420 users with 494.090 computers. There is still a migration from the classic client (as described here) to the new BOINC client going on. Therefor right now it is hard to determine the real number of users and the real computating power of SETI@home.
SETI@home did not find signs of extraterrestrial life until now, but together with related projects it established the viability of *PRC*. Nevertheless *PRC* is not limitless or free. Huge computing power causes huge data traffic which is either expensive or limited or both. This limits the frequency range searched by SETI@home, greater range means more bits per second. Compared to other radio SETI projects, SETI@home covers a narrow frequency range, but does a more through search within that range.

3.5 BOINC

In spite of global resource, and an abundance of promising applications, relatively few large-scale public-resource projects have emerged so far.
Conducting a public computing project in fact requires adapting an application program to various platforms, implementing server systems and databases, keeping track of user accounts and credit, dealing with redundancy and error conditions, and so on.
Some open-source systems have been developed, such as Cosm, jxta [73], and XtremWeb [74],but these systems provide only part of the necessary functionality. Commercial systems such as Entropia [75] and United Devices are more full-featured but not free. For this purpose the Space Sciences Laboratory at the University of California, Berkeley, developed a Middleware projects named BOINC (Berkeley Open Infrastructure for Network Computing) . It is released under the GNU Lesser Public License, which allows the usage even with proprietary software.
BOINC makes it easy for scientists to create and operate *PRC* projects. It supports different applications, including those with large storage or communication requirements.

PC owners can participate in multiple BOINC projects, and can specify how their resources are allocated among these projects. BOINC projects are autonomous; each one maintains its own servers and databases, and does not depend on others. Participants can register with multiple projects, and can control how their resources are shared.

BOINCs general goal is to advance the public resource computing paradigm: to encourage the creation of many projects, and to encourage a large fraction of the worlds computer owners to participate in one or more projects. Specific goals include *Reduce the barriers of entry to public-resource computing*, in fact research scientist with moderate computer skills can create and operate a large public-resource in few because of BOINC; BOINCbased project can use single machine configured with common open-source software as server.
BOINC goal is Share resources among autonomous projects too, BOINC-based projects are autonomous, not centrally authorized or registered. Each project operates its own servers and stands completely on its own. PC owners can participate in multiple projects, and can assign to each project a ”resource share” determining how scarce resource (such as CPU and disk space) are divided among projects.
By using BOINC is possible to *Support a wide range of applications*; it provides flexible and scalable mechanism for distributing data, and its scheduling algorithms intelligently match requirements with resources. Existing applications in common languages can run as BOINC applications with little or no modification.
According *PRC* paradigma BOINC must take in consideration the goal to *Reward participants* that represent the main resources of each project: it provides a *credit accounting* system according how much computation participants have contributed and that reflects usage of multiple resource types (CPU, network, disk); BOINC also makes it easy for projects to add visualization graphics to their applications, which can provide screensaver graphics.

Current projects running on BOINC

The intention behind BOINC development was to provide the knowledge of public resource computing from the SETI project to the public. It focus lies on simple setup and less maintenance combinated with ïncentivesf̈or the participants to encourage the creation of and the participation in projects, actually *PRC* projects, moved on BOINC platform, are some very important scientific projects about different topics. About climate change prediction there are both *Climateprediction.net* and *Climate@home*. The aim of the first project (based at Oxford University) is to quantify and reduce the uncertainties in long-term climate prediction based on computer simulations. This is accomplished by running large numbers of simulations with varying forcing scenarios (initial and boundary conditions, including natural and manmade components) and internal model parameters. Climate@home is a collaboration of researchers at

NCAR, MIT, UCAR, Rutgers, Lawrence Berkeley Lab, and U.C. Berkeley. Its scientific goals are similar to those of Climateprediction.net, but it will be using the NCAR Community Climate System Model (CCSM). It will collaborate with Climateprediction.net to maximize compatibility and minimize redundant effort, and to enable a systematic comparison of different climate models.
The Einstein@home project search for gravitational signals, it involves researchers from University of Wisconsin, U.C. Berkeley, California Institute of Technology, LIGO Hanford Observatory, University of Glasgow, and the Albert Einstein Institute. Its purpose is to detect certain types of gravitational waves, such as those from spinning neutron stars, that can be detected only by using highly selective filtering techniques that require extreme computing power. It will analyze data from the Laser Interferometry Gravitational Observatory (LIGO) and the British/German GEO6000 gravitational wave detector.
From the collaboration between researchers at the U.C. Berkeley Computer Sciences Department and the Intel Berkeley Research Laboratory, *UCB/Intel study of Internet resources* project was born. It seeks to study the structure and performance of the consumer Internet, together with the performance, dependability and usage characteristics of home PCs, in an effort to understand what resources are available for peer-to-peer services. *LHC@home: CERN LHC particle accelerator* are CERN (in Geneva, Switzerland) project deploying a BOINC-based project on 1,000 in-house PCs, and plans to launch the project publicly in coordination with its 50th anniversary in October 2004. The projects current application is a FORTRAN program that simulates the behaviour of the LHC (Large Hadron Collider) as a function of the parameters of individual superconducting magnets. *Predictor@home* project is about protein-related diseases, based at The Scripps Research Institute, studies protein behavior using CHARMM, a FORTRAN program for macromolecular dynamics and mechanics. A similar project is *Rosetta@home.* A BOINC-based project has been implemented starting from *Cell Computing* project about biomedical research. Finally *Folding@home* [63], based at Stanford University studies protein folding, misfolding, aggregation, and related diseases. It uses novel computational methods and distributed computing to simulate time scales thousands to millions of times longer than previously achieved.
Each of these projects using BOINC platform must be identified with a single master URL, which describes the project and provides registration, client download and the URL to scheduling servers. BOINC provides tools that let participants remotely install the client software on large numbers of machines, and attach the client to accounts on multiple projects.

3.5.1 BOINC project implementation

BOINC consists of a client which is shared among all BOINC projects, and a project specific server complex and application program(s). Figure 3.4 shows an overview of the BOINC system. BOINC has a central work queue, depicted as the BOINC database in the figure, and clients connect to the scheduler.

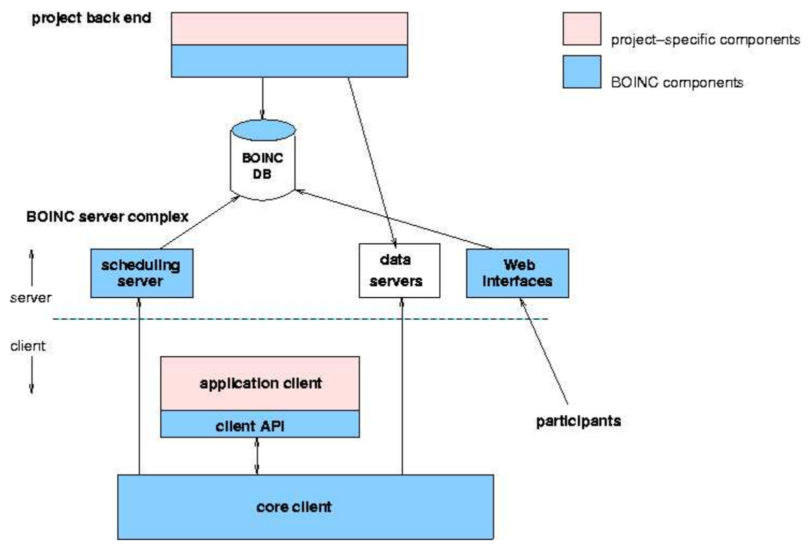


Fig. 3.4. BOINC overview

Client side includes interfaces for participants and developers providing the master URL, current project results and other information. The BOINC client software appears monolithic to participants but actually consists of several components. The *core client* performs network communication with scheduling and data servers, executes and monitors applications, and enforces preferences. It is implemented as a hierarchy of interacting finite-state machines, can manage unlimited concurrency of network transfers and computations, and is highly portable. The core client can operate in several modes: as a screensaver that shows the graphics of running applications; as a Windows service, which runs even when no users are logged in and logs errors to a database; as an application that provides a tabular view of projects, work, file transfers, and disk usage, and as a UNIX command-line program that communicates through stdin, stdout and stderr, and can be run from a cron job or startup file. A *client GUI* provides a spreadsheet-type view of the projects to which the host is attached, the work and file transfers in progress, and the disk usage. It also provides an interface for operations such as joining and quitting projects. It communicates with the core client via XML/RPC over a local TCP connection. The *Client API* provides information about utilization of resources and an interface for result visualisation. Finally the application client does the work on the *work units*.

A workunit represents the inputs to a computation: the application (but not a particular version) a set of references input files, and sets of command-line arguments and environment variables. Each workunit has parameters such as compute, memory and storage requirements and a soft deadline for completion. A result is the output of computation with files and a reference to a work unit. Files (associated with application versions, workunits, or results) have project-wide unique names and are immutable. Files can be replicated: the description of a file includes a list of URLs from which it may be downloaded or uploaded. Files contain attributes, like a list of URLs for down-/uploading and perhaps a compression flag.

When the BOINC client communicates with a scheduling server it reports completed work, and receives an XML document describing a collection of the above entities. The client then downloads and uploads files and runs applications; it maximizes concurrency, using multiple CPUs when possible and overlapping communication and computation.

The server complex of a BOINC project is centered around a relational database that stores descriptions of applications, platforms, versions, workunits, results, accounts, teams, and so on. Server functions are performed by a set of web services and daemon processes: *Scheduling servers* handles RPCs to communicate with partecipant hosts; it issues work and handles reports of completed results. *Data servers* handles input and output files distribution, using a certificate-based mechanism to ensure that only legitimate files, with prescribed size limits, can be uploaded. A relational database stores information about work, results, and participants. File downloads are handled by plain HTTP. BOINC provides tools (Python scripts and C++ interfaces) for creating, starting, stopping and querying projects; adding new applications, platforms, and application versions, creating workunits, and monitoring server performance.

The BOINC server consists of at least one web server that handles up- and downloads and a database server that keeps track of the state of the WorkUnits and their associated results. As shown in figure there is a *work generator* that firstly creates new jobs and their input files and a *scheduler*, a CGI program that is run whenever a client connects to the project and asks for work. Furthermore five different daemons periodically check the state of the database and perform any needed tasks within their area of responsibility. All these programs can be run on the same machine or they can be distributed to different servers for performance reasons.

The *transitioner* examines jobs for which a state transition has occurred and handles this change. The *validator* compares the instances of a work unit to verify whether the returned results are valid or not.

The *database purger* removes jobs and instance database entries that are no longer needed. The *assimilator* handels tasks which are done. The *file deleter* deletes input and output files that are no longer needed. The feeder

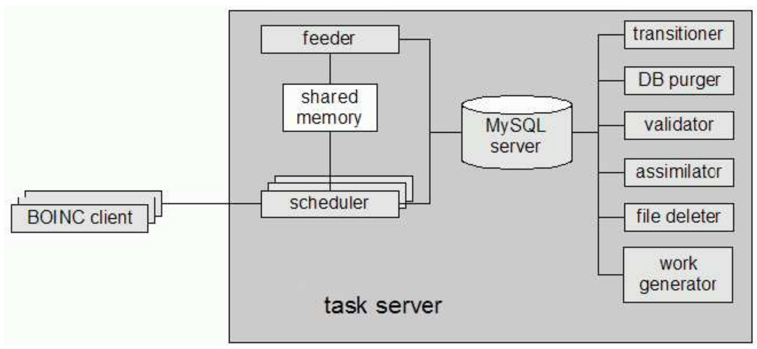


Fig. 3.5. BOINC task server

caches jobs which are not yet transmitted.

In this architecture servers and daemons can run on different hosts and can be replicated, so BOINC servers are scalable. Availability is enhanced because some daemons can run even while parts of the project are down (for example, the scheduler server and transitioner can operate even if the science database is down).

Summarizing the client application download work units from the Internet and allow to share execution time among various project. The BOINC client can be on execution on participant machine with the aspect of low priority process, as screensaver at the same way it happened for SETI@home.

The server side includes some special function as homogeneous redundancy, work unit preview and local scheduling. BOINC uses a simple but rich set of abstractions for files, applications, and data and a project defines application versions for various platforms (Windows, Linux/x86, Mac OS/X, etc.). For participants who, for security reasons, want to only run executables they have compiled themselves, whose computers have platforms not supported by the project, and who want to optimize applications for particular architectures, BOINC provides an anonymous platform mechanism. Such mechanism is usable with projects that make their application source code available. Participants can download and compile the application source code (or obtain executables from a third-party source) and, via an XML con- figuration file, inform the BOINC client of these application versions. Then, when the client communicates with that projects server, it indicates that its platform is anonymous and supplies a list of available application versions; the server supplies workunits (but not application versions) accordingly.

Redundant computing

BOINC supports redundant computing, a mechanism for identifying and rejecting erroneous computational results arising from malfunctioning comput-

ers/participants. Redundant because project can specify that N results should be created for each workunit. Once M over N of these have been distributed and completed, an application-specific function is called to compare the results and possibly select a canonical result. If no consensus is found, or if results fail, BOINC creates new results for the workunit, and continues this process until either a maximum result count or a timeout limit is reached.
BOINC implements redundant computing using several server daemon processes. First the *transitioner* implements the redundant computing logic in fact it generates new results as needed and identifies error conditions; the *validater* examines sets of results and selects canonical results. It includes an application-specific result-comparison function. The *assimilater* handles newly-found canonical results. Includes an application-specific function which typically parses the result and inserts it into a science database. Some numerical applications produce different outcomes for a given workunit depending on the machine architecture, operating system, compiler, and compiler flags. In such cases it may be difficult to distinguish between results that are correct but differ because of numerical variation, and results that are erroneous. BOINC provides a feature called *homogeneous redundancy* for such applications.

Failure and backoff

Public-resource computing projects may have millions of participants and a relatively modest server complex. If all the participants simultaneously try to connect to the server, a disastrous overload condition will generally develop. BOINC has a number of mechanisms to prevent this. All client/server communication uses exponential backoff in the case of failure. Thus, if a BOINC server comes up after an extended outage, its connection rate will be the longterm average. The exponential backoff scheme is extended to computational errors as well. If, for some reason, an application fails immediately on a given host, the BOINC client will not repeatedly contact the server; instead, it will delay based on the number of failures.

Local scheduling

The BOINC core client, in its decisions of when to get work and from what project, and what tasks to execute at a given point, implements a local scheduling policy with the goal to maximize resource usage, to satisfy result deadlines, to respect the participants resource share allocation among projects, to maintain a minimal variety among projects. This goal stems from user perception in the presence of long workunits. Participants will become bored or confused if they have registered for several projects and see only one project running for several months. The core client implements a scheduling policy, based on a dynamic resource debt to each project, that is guided by these goals.

Attracting people

Computer owners generally participate in distributed computing projects only if they incur no significant inconvenience, cost, or risk by doing so. BOINC lets participants control how and when their resources are used, during what hours can BOINC do work, how much disk space can BOINC use, how much network bandwidth can BOINC use; and so on. These preferences are edited via a web interface, and are propagated to all hosts attached to the account. Some BOINC-based applications perform computations that are so floating-point intensive that they cause CPU chips to overheat. BOINC allows users to specify a duty cycle for such applications on a given CPU.

The way to attract more participants and keep them involved is by the *accounting-credit* mechanism. The idea is to stimulate a competition between users or team of them, the best one in the competition is the one who gain more credits, where a single unit of credit" is a weighted combination of computation, storage and network transfer.
Credits information is typically displayed on web-based leaderboards showing the ranking of participants, BOINC provides a mechanism that exports credit-related data in XML files that can be downloaded and processed by credit statistics sites operated by third parties. As part of the accounting system, BOINC provides a cross-project identification mechanism that allows accounts on different projects with the same email address to identified, in a way that doesnt allow email addresses to be inferred. This mechanism allows leaderboard sites to display credit statistics summed over multiple BOINC-based projects.

3.6 PRC social aspects

To explain the relation among people and *PRC* is enough to explain how SETI@home have attracted 4.6 million participants, and why 600,000 of them remain active. First PRC, because its definition, is effective only if many people participate so the project team always try to attract clients in fact, in the case of SETI@home People could have learnt about it through several mechanisms.
The mass media have covered SETI@home, as have Internet news forums like Slashdot [60].
The same SETI@home's screensaver graphics are a powerful promotionalmechanism: in offices and school, where computers are seen by many people, a computer running SETI@home is a highly visible advertisement.
Research showes that 93% of SETI@home users are male, and that most of them are motivated primarily by their interest in the underlying science: they want to know if intelligent life exists outside earth.

Another major motivational factor is public acknowledgement. SETI@home keeps track of the contribution of each user (i.e. the amount of computation performed) and provides numerous web-site ”leader boards” where users are listed in order of their contribution. Users can also form ”teams”, which have their own leader boards. The team mechanism turned out to be very effective for recruiting new participants. In addition, users are recognized on the web site and thanked by email when they pass work unit milestones [56]. Kind of virtual communities were born around seti@home because people wanted become popular easily meeting others people inside the net; to achieve this purpose volunteers have translated the SETI@home web site into 30 languages, and have developed many kinds of add-on software and ancillary web sites.

Part II

A Super-peer model for Public Resource Computing

4

Super peer model and super peer model simulator

To execute a large number of jobs in a computer network is required an efficient communication protocol.
A typical task is the analysis of a set of data files to support scientific research, the case of PRC scenario. Otherwise it can be the execution of a workflow over the net to process data and extract relevant information, as in recommendation systems.
This kind of applications are usually managed through a centralized framework, i.e. BOINC, in which one server assigns jobs to workers, sends them input data, and then collects results, according to the *client-server model*; however this approach limits scalability. To cover the issue of scalability the application requires a decentralized protocol.
Main distributed application frameworks work well in the case of a small amount of data exchange, where the server congestion level rises not as much to became the bottleneck of the network. Anyway, even if the client-server model in some cases works well, a server node can be hard and expensive to manage.
If the number of servers increases as the dimension of the network, there will be a bigger set of servers and then the inter servers communication managment problem grows.
A scalable solution comes from the analisys of emerging paradigms for distribute computing as the $P2P$ and the SP model. True decentralized systems like the original Gnutella do not work well because they cannot scale - and arguably this is the defacto example of a $P2P$ topology i.e. completely decentralized.
SP seems to balance the features of others existing paradigms: inherent efficiency of centralized/hierarchical Virtual Organizations as well as autonomy, load balancing and fault-tolerance of $P2P$ networks.

Super-peer nodes act as centralized resources and connect to each other forming a $P2P$ system at a higher layer in the case of a limited number of nodes.

In *PRC* one node owns all the data, the *data source* node, and spreads these across the network, and this why the *PRC* is intrinsecally slow. It becames clear that at growing of network's nodes the number of initial nodes owning data files, the *data cachers*, must proportionally rise, specific loading algorithm is named *dynamic caching*. At the time data cachers are acquiring data files, they can immediatly forward them at any requiring node. This is possible thanks a decomposition of data files in smaller pieces, this idea reminds to Bittorrent. The resulting model, proposed in this work, includes a super-peer model to manage jobs submission in which node roles and their intercommunication flow over the net are formally defined in next sections.

4.1 Job Submission protocol

The super-peer model relies on the definitions of different *roles* that can be assumed by super-peers or by simple nodes:

- *Data source (DS)* is the node that owns all data files at the beginning of computation. Such node could have received data from an external sensor, for example a gravitational wave detector, for the GridOneD [66] scenario, and provide this data to nodes for the execution of jobs. Each data file is associated to a *data advert*, such as a metadata document which describes the characteristics of this file.
- *Job manager (JM)* produces *job adverts*, files that describe the characteristics of the jobs that must be executed, and is also responsible for the collection of output results.
- *Job assigner (JA)* is a super-peer node that receives *job advert* messages and stores them to be reassigned to workers asking for.
- *Worker (W)* is a node available for job execution. It first issues a *job query* to obtain a job to execute and then a data query to retrieve the input data file. A worker can disconnect at any time; if this occurs during the execution of a job, the latter will not be completed.
- *Super-peer (SP)* is a very important node to allow the cluster generation, in each peer's cluster there must be at last a super peer node that connect workers through a centralized topology. At the same time this SP is linked to others SP through a high level *P*2*P* network constituting the backbone of the super-peer overlay.
  In the proposed protocol, each super-peers play the role of *rendezvous nodes*, since they compare job and data description documents (*job* and *data adverts*) with queries issued to discover these documents, thereby acting as a meeting place for job or data providers and consumers.
- *Data cacher (DCa)* is a special super-peer which has additional abilities to cache data (and associated data adverts) retrieved from a data source, and can directly provide such data to workers. Super peers and data cachers can be distributed on separate nodes if desired but in test cases, we just hosted the data cachers on super peers for simplicity.

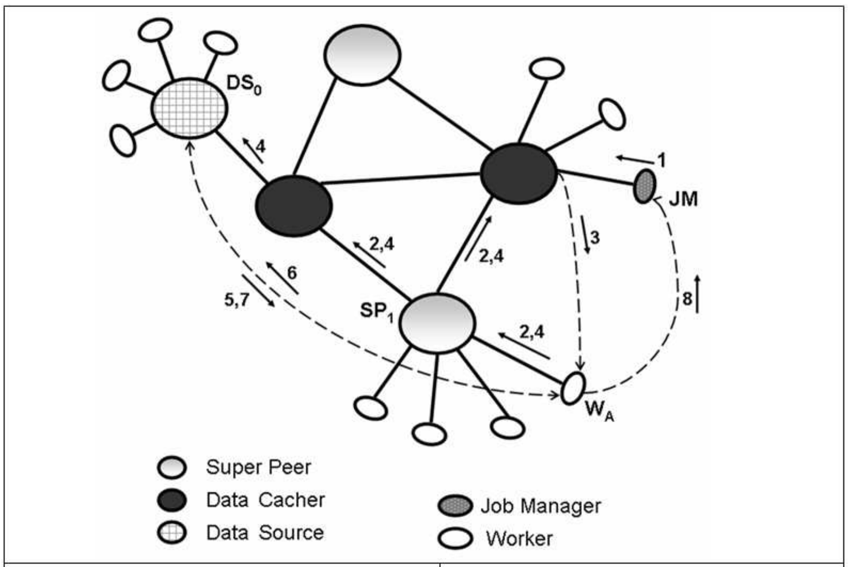
In the following, a *data source* and a *data cacher* are both referred as *data center (DC)*, since collectively they are able to provide data to workers, although at different phases of the process: data sources from the very beginning, data cachers after retrieving data from data sources or other data cachers.
For simplicity, it is assumed that only super-peer can assume the role of data center, but the protocol can be easily extended to the case in which even simple peer can cache and provide data.

It's not compromising supposing that the same user-driven process is used to configure a peer; that is, each user decides if he wants to be a super peer and/or data center, as well as a worker. In the BOINC scenario, the existing dedicated machines would form the obvious data-center backbone and other peers (with high storage and network capacity) would also make themselves available in this model.

4.1.1 Job Assignment and Data Download

Figure 4.1 depicts the sequence of messages exchanged among Ws, SP and DCs for the execution of the job submission protocol, in a sample topology with 5 SP, of which one is a DS and two others are $DCas$. This example describes the behavior of the protocol when a *job query* is issued by the worker W_A; in this case dynamic caching is not exploited because: (i) input data is only available on the data source DS_0, i.e., no data cachers have yet downloaded data; (ii) data cannot be stored by the super-peer connected to W_A, since this is not a data cacher. The behavior of the protocol with dynamic caching is explained later.

The protocol requires that job execution is preceded by two matching phases: the *job-assignment* phase and the *data-download* phase. In the *job-assignment* phase the *job manager*(the node JM in the figure) generates a number of *job adverts*, which are XML documents describing the properties of the jobs to be executed (job parameters, characteristics of the platforms on which they must be executed, information about required input data files, etc.), and sends them to the local rendezvous super-peer, which stores the adverts. This corresponds to step 1 in the figure. Each worker, when ready to offer a fraction of its CPU time (in this case, worker W_A), sends a *job query* that travels the network through the super-peer interconnections (step 2), until the message time-to-live parameter is decremented to 0 or the job query finds a matching job advert. A job query is expressed by an XML document and typically contains hardware and software features of the requesting node as well as CPU time and memory amount that the node offers. A job query matches a job advert when the job query parameters are compatible with the information contained in the job advert. Whenever the job query gets to a rendezvous super-peer that maintains a matching job advert, such a rendezvous assigns the related job to the requesting worker by directly sending it a *job assignment* message (step 3).



1. job adverts	*5. data advert*
2. job query	*6. data download request*
3. job assignment	*7. data download*
4. data query	*8. job results*

Fig. 4.1. Super-peer job submission protocol: sample network topology and sequence of exchanged messages to execute one job at the worker W_A. Dynamic caching is not used because it is assumed that data cachers have no yet stored data.

In the *data-download* phase, the worker that has been assigned a job inspects the job advert, which contains information about the job and the required input data file, i.e., size and type of data. In a similar fashion to the job assignment phase, the worker sends a *data query* message (step 4), which travels the super-peer network searching for a matching input data file stored by a data center. Since the same file can be maintained by different data centers, a data center that successfully matches a data query does not send data directly to the worker, in order to avoid multiple transmissions of the same file. Conversely, the data center (in this example the data source DS_0) sends only a small *data advert* to the worker (step 5). The worker chooses a data center, and initiates the download operation (steps 6 and 7). After receiving the input data, the worker executes the job, reports the results to the job manager (step 8) and possibly issues another *job query.*

It can be noticed that in the job assignment phase the protocol works in a way similar to the BOINC software, except that job queries are not sent directly to the job manager, as in BOINC, but travel the super-peer network hop by hop. Conversely, the data download phase differs from BOINC in that it exploits the presence of multiple data centers in order to replicate input data files across the network.

The case of multiple assigners

In the case of multiple assigners the *job-assignment* phase the change is in the *adverts* propagation way, see at figure 4.2. The *JM* sends an *assigner query* message to search for JA inside the network (step 1) and it waits a predefined amount of time. Once a JA receives the message it replies with an *assigner advert* (step 2). The JM can ignore or cache the messages depending the arrival time, specially if it's not outdate it generates a number of *job adverts* and sends them to available *JA* if any (step 3), otherwise to the local rendezvous super-peer as before, which stores the adverts.

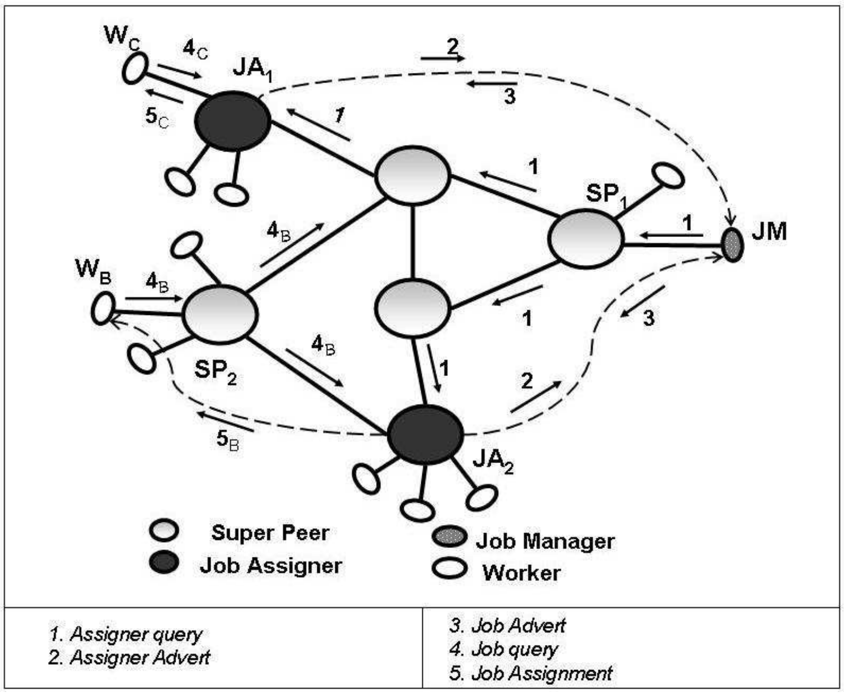


Fig. 4.2. Sample network topology and sequence of exchanged messages to propagate adverts to a JA_i and following their assignment to workers W_i.

Each worker sends a *job query*, with fixed TTL, that travels the net through the super-peer interconnections (step 4), until the job query finds a matching

job advert from any of available JA after accepting job from it with. Finally the job query assigns the related job to the requesting worker by directly sending it a *job assignment* message (step 5).
About download phase almost nothing changes except that once a JM receives a *job result* by a worker, in the case the job has been executed at last the required number of times, it must inform all JA so they won't ever assign this job again.

4.1.2 Dynamic Caching

One of the main features of our super-peer protocol is the dynamic caching funtionality which allows for the replication of input data files on multiple data cachers. This leads to well known advantages such as increased degree of *data avalaibility* and improved *fault tolerance*.
Dynamic caching allows for the replication of input data files on multiple data cachers. This leads to well known advantages such as increased degree of *data availability* and improved *fault tolerance*. Moreover, dynamic caching allows for a significant saving of time in the data download phase, because *data queries* have a greater chance to find an available data center, and most Ws can download data from a neighbor data cacher instead of a remote data source. The remaining part of this section illustrates the dynamic caching mechanism, while the performance evaluation is discussed in Section 4.4.

Figure 4.3 shows how the protocol handles dynamic caching, both in the *replication* phase (which occurs when data is downloaded from a data source and stored by a data cacher) and in the *retrieval* phase (which occurs when data is retrieved from a data cacher by a worker). These two mechanisms are described by displaying the messages exchanged when two workers W_B and W_C, connected to the same data cacher DC_2, issue two job queries at different times, first W_B then W_C. For simplicity, only messages related to the *download phase* are shown, and they are distinguished by subscripts A and B, corresponding to the two workers. The data query issued by W_B finds a matching in the data source DS_0. As opposed to the case described in Figure 4.1, this time the super-peer connected to W_B is a data cacher, DC_2. To let this data cacher store the data file, the data advert is not sent directly to the worker W_B, but first to DC_2 and then from DC_2 to the worker. Analogously, the data file is downloaded by DC_2, which replicates and caches it, and then passed to the worker. Subsequently, DC_2 will act like a data source for the period of time in which it maintains the data file in its cache. In this example, the data query issued by W_C will be served directly by the cacher DC_2 instead of the data source DS_0.

To increase performance, a file splitting approach is adopted: data files are not downloaded as a whole but split in ordered fragments. This is useful to avoid the generation of unnecessary messages, which would cause a larger network load that is wasteful of resources. For example, if the data cacher DC_2 , when receiving a data query, does not hold the entire data file but has

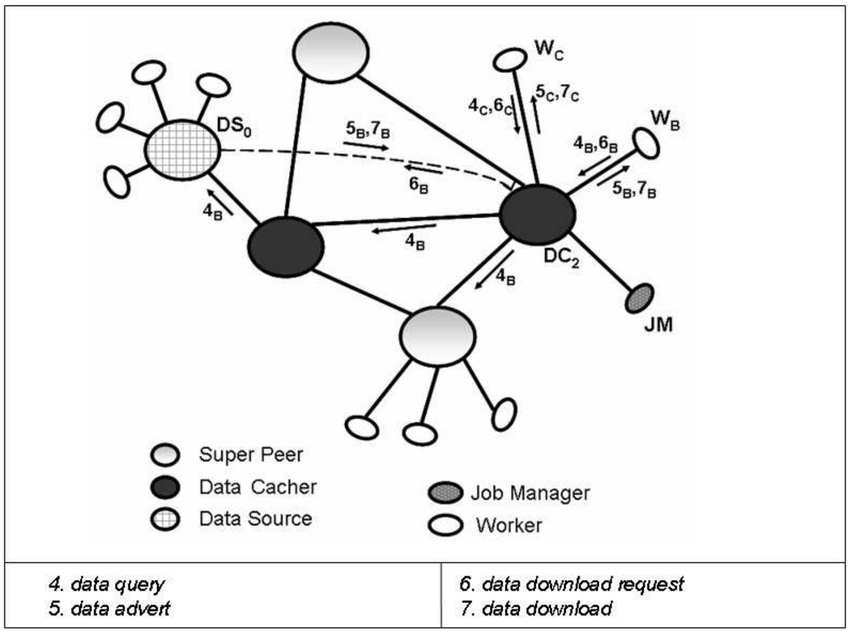


Fig. 4.3. Download phase of the super-peer job submission protocol, with dynamic caching. After the request of worker W_B, the data cacher DC_2 retrieves the data file from the data source DS_0, replicates and caches the file, and delivers it to W_B. Subsequently, the request of worker W_C is directly server by the data cacher DC_2.

already received a part of it from DS_0, it will not forward the data query, because it will soon receive the remaining fragments from DS_0. As soon as it receives these fragments, DC_2 will pass them to the requesting worker W_C.

A further improvement could be obtained by enabling the parallel download of data segmentes; it reminds to Bittorrent approach.
In the ideal scenario depicted so far, we never considerate that peers can disconnect. This anyway in real situations happen most of the time so in our model we assumed that only super-peer node doesn't disconnect. Since peer then can disconnect it must be considerated that a job started will be completely waste and this can't be avoid. One possible approach is to resume this job at the point it was interrupted. In our dynamic scenario time is a constraint, to solve the job later can make no sense, infact it could have been in that time complited by one other peer in the case of multiple job assignment or just to be out of date, so we just choose to reassign new job belonging to the current execution once a peer reconnects.

The model analyzes so far, has been proposed in [83, 84].

4.2 Discovery mechanism

In most unstructured $P2P$ system, discovery techniques consist in *flooding* algorithms. *Simple flooding* and its variations fit our case. To modelize a *limited flooding* is quite easy. In fact it's enought to include in each *query message* the information about the *time to live TTL* the upper limit of nodes it can go through. Generally in flooding a message is spreaded over the net in any destination by neighbour nodes. Possible optimizations of limited flooding algorithm can be generated if we considerate that the messages do noy reach the senders nodes, so they will not be forwarded if the query has been already examined. This approach is used in Gnutella. The last feature can be implemented assigning a random number to each query, then super-peer nodes must know (store) which query (number) have already examined, so will not forwarded again.
To implement *iterative deepening* algorithm again we must set the TTL; obviously the main task is to set the appropiate value of it.
Above consideration avoid the cycle generation problem and avoid the network messages overflow.
In our case the flooding is used only for *query messages*, because normally other messages go directly to destination by address matching. The flooding of *query messages* involves only the super-peers backbone, see at *job query* and *data query* messages flow in figure 4.1. This means that only a limited subset of nodes, the super-peer nodes, are throught inside the network.

4.3 Simulator description

One ad hoc event-based simulator, has been realized in C++ language to implement the super-peer protocol as illustrated below. The protocol has been simulated on different networks. To generate a network a membership management network generator protocol based on *contact nodes* has been used, such protocol has been realized with a Java language program.

In the design of the simulator, it's been preliminary assumed that all the jobs have similar characteristics and that can be executed by any worker, each data file is unique and a job execution fails upon the disconnection of the corresponding worker.

At the beginning of one simulation, you can act on different settable *Simulation Parameters* concerning the underlying network configuration as well as jobs and data features in computation execution. To evaluate how performance changes respect to simulation parameters setting, you can refers to a fixed set of *Performance Indices.*

4.3.1 Simulator messages

The event based simulator uses real messages travelling the network to simulate the underlying protocol as i.e. the request for an advert, a request or a query. Anyway others messages are used to drive the simulation, expecially management messages such as start and end of simulation, start of the workers node, timers to notify the end of an operation or the beginning of a new operation. Here we list all the messages, the second type is featured by the (∗) symbol:

- End(*)
- StartSim(*)
- StartWorker(*)
- JobDescriptionQuery
- JobAdvert
- JobAssignment
- JobQuery
- DataDescriptionQuery
- DataDescriptionAdvert
- DataQuery
- DataMessage
- JobCompletion(*)
- JobResults
- FragmentDownLoad
- Interruption(*)
- JobCompletionVerification(*)
- JobCompletionAdvert
- DataFragmentAdvert(*)
- AssignerDescriptionQuery
- AssignerDescriptionAdvert
- TimerAssignerVerification(*)
- StopAssigner
- EndFragmentDownload(*)

A generic simulation starts after it has been activated by the *Sim manager (SM)*. Actions in the system happen after has been received a message and this brings to the generation of new messages and eventually to the system state changing.
The whole messages change can be synthesized in the following schemas; later in this section the state automatas will be analyzed.
The case of *SPs* nodes:

Message Received: a JobAdvert that contains a job code, the characteristics of the platforms on which must be executed, how many times it must be executed (MTL), the code of JM asking for the job, and more info about input data for job execution as filename and filesize.

Precondition: The SP receiving such message is a JA.
Action: The JA stores in the cache these informations and then it's ready to assign the job MTL-times to workers asking for.

Message Received: a JobDescriptionQuery contains the characteristics of the platform that will execute the job.
Action: If the query has been already processed, nothing happens. In general the SP, that is a JA, searches inside its cache which job can assign by matching characteristics of the platform sending the request. Once the job has been found there are two possible situations: the case the JA it's unique and then it assignes the job to the requiring W by sending a $JobAssignment$ message and decreases the job MTL value; the other case, the JA sends a *JobDescriptionAdvert* to inform the W that a job is available. If the JA hasn't a matched job or the SP is not even a JA, if the request is valid ($TTL > 0$), it forwards the *JobDescriptionQuery* to neighbour SPs except the one from which it received the query.
Message Send: JobAssignment or JobDescriptionAdvert or JobDescriptionQuery.

Message Received: JobQuery.
Precondition: The SP receiving such message is a JA with a job to assign, please notice we are in the case of not unique JA.
Actions: the choosed JA sends a *job assignment* message and it decreases the job MTL value. In case of peer disconnection and job re-assignment, the JA must use a timer *(JobCompletionVerification)* to check if the task has been completed and eventually to re-assign it.
Message Send: JobAssignment and JobCompletionVerification.

Message Received: DataDescriptionQuery.
Actions: If the query has been already processed, nothing happens. If the SP is a DC and it owns data, it sends a *DataDescriptionAdvert* to the requiring W, otherwise it forwards the data request query to all of its neighbours except the one from which received the query. If the SP is a DC without data or just a part of them, and the request comes from a W of its cluster, it sends a *DataDescriptionAdvert* to the requiring W, and later it could eventually search for data.
Message Send: DataDescriptionAdvert, DataDescriptionQuery.

Message Received: DataQuery.
Precondition: The SP is a DC.
Actions: If the DC hasn't the data it starts searching for, sending to each SP neighbour a *DataDescriptionQuery*. Once the DC has all the data, the case of *Total Caching* or it has a part of them, the case of *Partial caching*, it sends the first fragment to the requiring job and it informs about that with a *DataFragmentAdvert*, and then it generates a *EndFragmentDownload* mes-

sage.
Message Send: DataFragmentAdvert, DataDescriptionQuery, EndFragment-Download.

Message Received: FragmentDownLoad is the request of a fragment.
Precondition: The SP is a DC that has the data.
Actions: The DC sends the fragment to the requiring W or DC and then it sends a *DataFragmentAdvert* to inform about the end of the download, in the case of the last fragment of a message it uses a *DataMessage*, anyway then it generates a *EndFragmentDownload* message.
Message Send: EndFragmentDownload, DataFragmentAdvert, DataMessage.

Message Received: JobCompletionAdvert contains the just completed job code.
Precondition: The SP is a JA.
Actions: The JA that receives such messages from the Job-Manager can now refresh its database updating the info about new completed jobs by matching the job code.

Message Received: JobCompletionVerification contains the job code.
Actions: When SP receives this timer, it verifies if the job has been completed by matching the job code, if not it must be re-assigned updating the database. Please notice we are in the case of peer disconnection and job re-assignment.

Message Received: DataFragmentAdvert.
Precondition: The SP is a DC that has just received a data fragment, but not the last.
Actions: The DC asks for the next fragment by sending a *FragmentDownLoad* message. In the case of Partial Caching, the DC uploads the just received fragment to Ws waiting for and it sends a *DataFragmentAdvert*, then it generates a *EndFragmentDownload*.
Message Send: FragmentDownLoad, EndFragmentDownload, DataFragmentAdvert.

Message Received: AssignerDescriptionQuery.
Actions: If the query has been already processed, nothing happens. If the SP is a JA, it sends to the JM a *AssignerDescriptionAdvert* otherwise the SP forwards the query to all SP neighbours except the one who has sent the request.
Message Send: AssignerDescriptionAdvert, AssignerDescriptionQuery.

Message Received: StopAssigner contains the already completed job code.
Precondition: The SP is a JA.
Actions: The JM sends this message to the JA to inform that the required

number of execution for certain job has been reached, then the job mustn't be assigned anymore.

Message Received: EndFragmentDownload: a DC sends such message to inform about the end of a fragment download.
Actions: it's useful to update the number of P and SP connected to the DC.

Message Received: DataDescriptionAdvert contains the code of the DC that has made a request for data.
Precondition: The SP is a DC.
Actions: At the first istance of this message the DC reply by sending a *Data-Query.*
Message Send: DataQuery.

In the case the receiver is a JM there are all these possible messages:

Message Received: StartSim is the message sent by the SM to the JM for starting the simulation.
Actions: If there is just one default JA, the Super-Peer of the custer, the JM sends every *JobDescriptionAdvert* to it; otherwise the JM sends an *Assign-erDescriptionQuery* to search for JA in the network and it sends to itself a *TimerAssignerVerification* to prevent infinite time waiting.
Message Send: JobDescriptionAdvert, AssignerDescriptionQuery, TimerAs-signerVerification.

Message Received: TimerAssignerVerification is a message arriving when the waiting time to search for JA is gone.
Actions: if the JA required number have been already reached, the message is ignored. If no JA has been found, JM sends all the *JobDescriptionAdvert* to default JA, its cluster SP, otherwise it splits the descriptions among available JA.
Message Send: JobDescriptionAdvert.

Message Received: AssignerDescriptionAdvert contains the code of the JA available
Actions: If this message is out of date, the JA availability will be ignored. If not yet the JA required number in the simulation has been reached, this message is cached, otherwise it splits the process descriptions among available JA by sending a *JobDescriptionAdvert.*
Message Send: JobDescriptionAdvert

Message Received: JobResults contains the job code and its execution re-sult and the address of the JA assigned the job.
Actions: The JM stores code and result in its cache. In the case of peer dis-connection, the JM adverts the JA about the job was assigned and ended by

a *JobCompletionAdvert*. Then, in the case there are many JA and the job has been executed at last Nexec times, the JM sends *StopAssigner* to all JA so they won't ever assign this job again.
Message Send: JobCompletionAdvert, StopAssigner.

Last group of messages refers to the case of the receiver is a Worker:

Message Received: StartWorker
Precondition: the W is in Interrupted or Idle state.
Actions: The peer's state becomes ready. The W sends a *JobDescriptionQuery* to local SP to find a job. In the scenario of Ps disconnection, the W computes in how much time it'll disconnect and it sends itself an *Interruption* message.
Message Send: JobDescriptionQuery, Interruption.

Message Received: JobAssignment contains information about required input data to execute the job.
Actions: if the job execution depends from data, the W, now in DataRequest state, will send a *DataDescriptionQuery* to the local SP (the code of this W is put into the query as a return pointer); otherwise the peer's state becames InProgress, the W executes the job, this means W calculates a (random) job time T_j and it sends itself a *JobCompletion* message, that will be received after currentSimTime+T_j.
Message Send: DataDescriptionQuery, JobCompletion

Message Received: JobDescriptionAdvert contains the code of the JA having the described job.
Actions: This message is sent by a JA to inform a W that a new job is available. If the W is available as well, it sends a *JobQuery.*
Message Send: JobQuery.

Message Received: DataDescriptionAdvert contains the id of a DC owning required data.
Actions: At the first istance of this message the W replies by sending a *DataQuery.* The peer's becames DataFound.
Message Send: DataQuery.

Message Received: DataMessage.
Actions: the W executes the job; the way to simulate the job execution is that W calculates the job time T_j and sends itself a *JobCompletion* message, that will be received after currentSimTime+T_j. The peer's state becames InProgress.
Message Send: JobCompletion.

Message Received: JobCompletion alerts the W that a job has been ended.

Actions: The W sends a *JobResults* message directly to the Job Initiator (its address was included in the last JobAssignment message received); The peer's state becames Pause. Then it can wait for a random time and it sends a new *JobDescriptionQuery* to the local SP. The peer's state becomes Ready again.
Message Send: JobResults, JobDescriptionQuery.

Message Received: Interruption alerts the W that it must disconnect
Actions: The W disconnects, this means that it computes a time T_j and sends itself a *StartWorker* message that will be received after currentSimTime+T_j. The peer's state becomes Interrupted.
Message Send: StartWorker.

Message Received: DataFragmentAdvert.
Precondition: The W has just received a data fragment from a DC.
Actions: The W asks for the next fragment by sending a *FragmentDownLoad*.
Message Send: FragmentDownLoad.

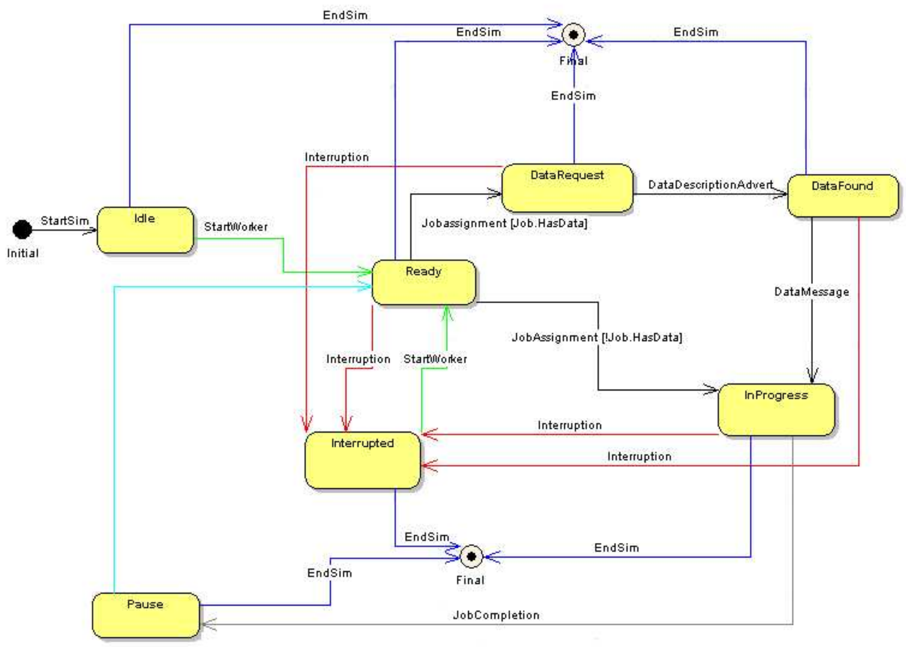


Fig. 4.4. Worker finite state automata (case 1)

To modelize the workers behaviour inside the system, two possible scenarios have been represented by a state automata, the first shows how a worker acts when there is just a JA in the network in figure 4.4, the second in case of many JAs, see at figure 4.5.

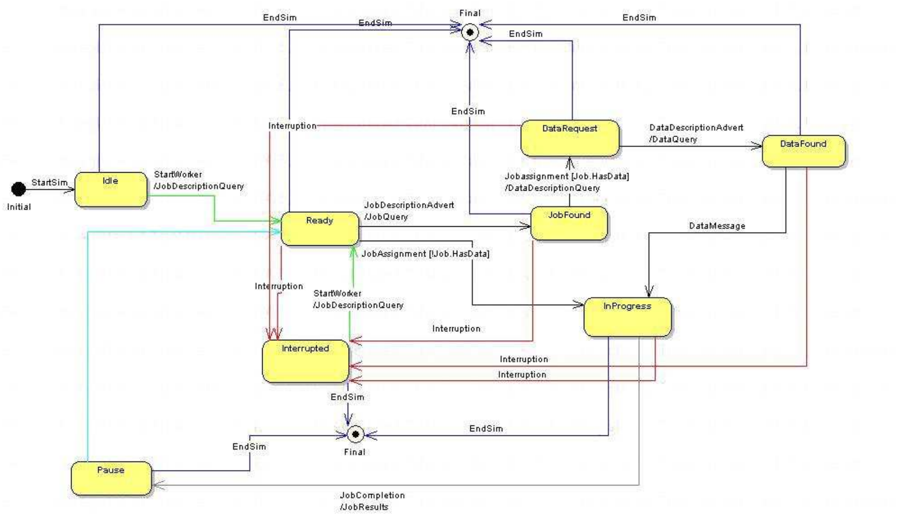


Fig. 4.5. Worker finite state automata (case 2)

4.3.2 Membership protocol

The membership protocol exploits the characteristics of *contact nodes*. A contact node is a node that plays the role of an intermediary node during the building process of the network. One or more contact nodes are available and once a super-peer node wants to connect to the net, contacts a subset of contact nodes and registers at those nodes.
In turn, the selected contact nodes randomly choose a number of previously registered super-peers and communicate their addresses to the requesting super-peer, after that the super-peers add the new super-peer address to their neighbour set.
The figure 4.6 shows a schema of the membership management protocol. A number of contact nodes are depicted, and for each of them the corresponding set of registered super-peers is reported. In Fig. 4.6(b), the super-peer S wants to connect to the net and selects two contact nodes. In Fig. 4.6(b), the selected contact nodes add S to the list of registered super-peers and respond to it by communicating the addresses of a number of super-peers, which will constitute the neighbour set of node S. The membership management protocol requires a proper setting of the contact parameter K, i.e. the number of contact nodes at which a new super-peers registers (K=2 in Fig.4.6).

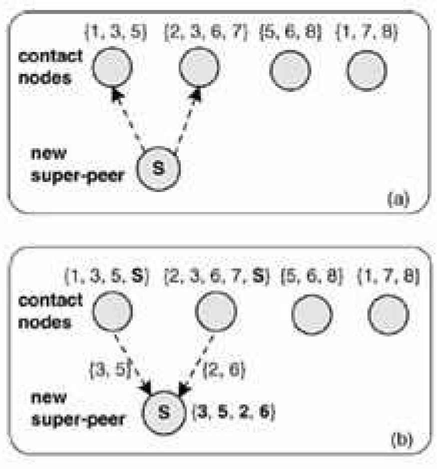


Fig. 4.6. The membership management protocol: (a) a new super-peer registers at a set of contact nodes and (b) receives the identities of its neighbour super-peers.

In general a super-peer communicates with contact nodes either periodically or whenever it detects the disconnection of a neighbour super-peer, in order to ask for its substitution. In our case this never happens, in fact only peer nodes can disconnect but not super-peer nodes. For our purpose a P is connected at last to one SP so it belongs exactly to one cluster. This means that one the network has been generated it doesn't change, so we used the same network for each simulation.
It's pointless if ordinary nodes, i.e. simple peers, are already connected to the super-peer before it initiates the joining procedure or can connect to the super-peer after it has joined the cluster, it's enough that peer knows the cluster super-peer address.
Membership management input parameters are:

- *graphSize*: the total number of peer and super peer nodes inside the network;
- *clusterSize*: the medium number of nodes inside a cluster, that corresponds to the number of peers connected to a super-peer;
- *nContactPeer*: number of contact peer available;
- *k*: number of contact peer assigned to a super peer;
- *Neighborhood*: number of neighbour of a super-peer;
- *DCpercentage*: percentage of super-peer data centers respect to super-peers.

To set the network simulator you must fill in a configuration file setting the previous parameters value. Particularly cluster size, nContact and k are required to generate the pure super-peers network, graph size is used to compute the super peers number according the equation:

$$n = \frac{graphSize}{clusterSize} \tag{4.1}$$

To allow the full network generation, you need to design peers node and clusters they belong to, required parameters are Neighborhood and DCpercentage.
The algorithm described before generates the decentralized $P2P$ network and all the clusters with Ps. The network layout output is given in textual format, specially a file containing the SPs nodes connections and number of Ps for each SP inside a cluster. In these files a simple formalism is used:

- $X \rightarrow Y$ represents the direct monodirectional connection from the node X to Y;
- $X == Y[> 1]$ means that the SP X owns Y P and if the optional part > 1 appears in the expression it means that such SP is a datacenter.

Because structural constraint, for each $X \rightarrow Y$ connection, will be automatically generated a $Y \rightarrow X$ connection.

4.3.3 Simulation Parameters

The network is characterized by the *number of super peer* (N_{speer}) and by the *average number of workers* each *connected to the super peer*. Each super-peer with its peers composes a grid organization. The super-peer overlay network is organized so that each super-peer is connected to at most at *Maximum number of neighbor for a super-peer.*
You can also define which super-peer assumes the role of DS or DCa or JA and which peer the role of JM.

In the model, *Bandwidth* and network *Latency* is either a settable parameters, this to deal with realistic scenarios where local connections (i.e. between a super-peer and a local simple peer) can have a larger bandwidth and a shorter latency than remote connections. To limitate the network load it's possible to opportunaty set the TTL value, then to limitate the numbers of hops that a messages does inside the net.
It must be setted for each file its *size* and, in the case of dynamic caching, the size of each segments the file can be splitted in.
For a generic peer you must describe its runtime behaviour by setting its *mean job execution time* and *average connection/disconnection time of workers* but not the *download time*, since it's calculated assuming that the downstream bandwidth available at a data center is equally shared among all the download connections that are simultaneous active from the data center to different workers.
Generally you must set the *Number of jobs* to be executed but in redundant computing each job will be executed more then once. To achieve multiple execution of every single job (which can be useful to enhance statistical accuracy

or perform parameter sweep analysis) two parameters have been added: *Number of execution requested for each jobs*, N_{exec} and *Matches to live, (MTL)*, which must be not lower than N_{exec}. A proper choice of MTL can compensate for possible disconnections of workers and consequent job failures.

4.3.4 Performance Indices

Performance indeces, valutated in our simulations are:

- *Overall execution time* T_{exec} represents the time needed to execute all the jobs, in general expressed in seconds. This index is crucial to determine the rate at which data files can be retrieved from any extern source while guaranteeing that the workers are able to keep the pace with data.
- *Throughput* T_{hr} is the average number of jobs completed per time unit (job/s). By T_{hr} it's possible to evaluate the efficiency of the job submission system.
- *Percentage of DC activity time* P_{act} is the average percentage of time in which a data center is active, i.e. has at least one download connection in progress.
- *Mean download time* T_{dl} corresponds to the average time that it takes for a worker to download a data file from a data center.
- *Network load* N_{load} is the average number of messages travelling inside the network per time unit (messages/s).
- *Max number of executed jobs* J_{max} represents the maximum number of executed jobs from a single worker.
- *Mean number of executed jobs* J_{avg} represents the average number of executed jobs from a single worker.

The last performance indices help determine the load that is experienced by data centers and by workers in different scenarios and the network load in general.

4.4 Performance Evaluation

Performance evaluations have been based on realistic scenarios of distributed applications, for example a data-intensive applications on Grids as the GridOneD project [66].
This project shows how one might conduct a massively distributed search for gravitational waveforms produced by orbiting neutron stars. In this scenario, a data file of about 7.2 MB of data is produced every 15 minutes and it must be compared with a large number of templates (between 5,000 and 10,000) by performing fast correlation. It is estimated that such computations take approximately 500 seconds to compute. Data can be analyzed in parallel by a number of Grid nodes to speed up computation and keep the pace with data

production. A single job consists on the comparison of the input data file with a number of templates, and in general must be executed multiple times in order to assure a given statistical accuracy. Evaluation indices and specially T_{exec} allows to determine the rate at which data files can be retrieved from the astronomic telescope.

The simulation scenario is described in Table 4.1. The parameters of the representative astronomy scenario mentioned in Section 4.1 are used for the test case (file size, job execution time, etc.). It is assumed that all the jobs have similar characteristics and can be executed by any worker.

Table 4.1. Simulation scenario.

$Scenario feature$	*Value*
Number of super-peers, N_{speer}	25 to 100
Maximum number of neighbors for a super-peer	4
Average number of workers connected to a super-peer	10
Average connection time of workers	4 h
Average disconnection time of workers	1 h
Number of data centers (data sources + data cachers)	about 50% of N_{speer}
Latency between two adjacent super-peers (*or between two remote peers in a direct connection*)	100 ms
Latency between a super-peer and a local worker	10 ms
Bandwidth between two adjacent super-peers (*or between two remote peers in a direct connection*)	1 Mbps
Bandwidth between a super-peer and a local worker	10 Mbps
Size of input data files	7.2 Mbytes
Mean job execution time	500 s
Number of jobs, N_{job}	50 to 500
Number of executions requested for each job, N_{exec}	10
Matches to live, MTL	10 to 30

In next chapter many simulations will be analyzed, starting from this base scenario. In these simulations the performance indices will be computated to analyze our model behaviour.

5

Model analysis by simulation techniques

This chapter describes the result of our simulator respect to a decentralized architecture for data-intensive scientific computing according to the *PRC* model.
In the various scenarios presented here, a small group of nodes maintains and advertises job description files and a large number of dispersed worker nodes execute the required tasks. Job assignment is performed by a group of rendezvous peers, which form a super-peer overlay network and match job descriptions with job queries when they are issued by available worker nodes. To provide support for this scheme, has been evaluated the impact of application (the number of jobs and the number of times that each of them is assigned to workers for statistical analysis) and network parameters (the number of workers and data centers) on performance indices such as the overall time to execute a given set of jobs, the utilization of data centers, the network load and the computational load of a single worker in different cases:

- base scenario
- redundant submission of job scenario
- dynamic caching scenario
- peers disconnection scenario
- multiple jobs assigner scenario
- comparition of variable size network scenarios

From the cross analysis comes up some meaningful combined scenarios such *base scenario with disconnection*, *multiple job submission with or without peers disconnection.*

Results show that the use of several DSs can bring benefits to network applications in terms of lower total execution times, higher throughput and load balancing among worker nodes. However, since a large number of DSs also causes a smaller utilization of a single DS, the study can also be used to determine the number of DSs that can maximize the return of investment related to the deployment of new DSs.

We present the analysis of (1) redundant computing for applications that require multiple executions of each job; (2) caching of data file fragments on the $P2P$ network, instead of storing entire files, to improve data download performance. Further development can take on account the study of performance of the super-peer protocol in the case that input data is progressively fed as a data stream by an external source.

5.1 Base scenario

It's now analyzed a simple scenario [83] based on a net containing 25 super-peers and 250 ordinary peers.
Simulation parameters, and corresponding values, are reported in table 5.1. The network is composed of 25 cluster organizations, each containing one super-peer node and 10 regular nodes on average. The super-peer overlay network is organized so that each super-peer is connected to at most 4 neighbor super-peers. It is assumed that local connections (i.e. between a super-peer and a local simple peer) have a larger bandwidth and a shorter latency than remote connections. To compute download times with a proper accuracy, a data file is split in 1 MB segments, and for each segment the download time is calculated assuming that the downstream bandwidth available at a DS is equally shared among all the download connections that are simultaneous active from the DS to different workers.
In this preliminary study, it is assumed that the DSs download input data files before the workers join the system and issue their job queries. In the rest of the chapter, analysis will focus on a more complex scenario in which data files are replicated.
The last two rows of Table 1 are related to parameters that were given varying values in the simulation runs, specifically the number of jobs Njob and the number of data sources Nds i. e. the number of super-peer nodes that own data files at the time job execution starts.

Figure 5.1 shows that the overall execution time decreases as more DSs are made available in the network, for two main reasons: first DSs are less heavily loaded and therefore data download time decreases, second workers can exploit a higher parallelism both in the downloading phase and during the execution of jobs. However, depending on the number of jobs to be executed, it is possible to determine a suitable number of DSs, beyond which the insertion of a further DS produces a performance increase which does not justify the related cost. For example, if 10,000 jobs are to be executed, a significant reduction of T_{exec} is perceived as the number of DSs is increased up to a value of 9, whereas if the number of jobs is not greater than 1,000 two or three DSs are sufficient to achieve a good performance level. Analogous comments can be made about the throughput index, reported in figure 5.2. A further consideration is that the throughput increases with the number of jobs because

Table 5.1. Simulation parameters

$Scenariofeature$	*Value*
Number of super-peers, N_{speer}	25
Maximum number of neighbors for a super-peer	4
Average number of workers connected to a super-peer	10
Latency between two adjacent super-peers (*or between two remote peers in a direct connection*)	100 ms
Latency between a super-peer and a local worker	10 ms
Bandwidth between two adjacent super-peers (*or between two remote peers in a direct connection*)	1 Mbps
Bandwidth between a super-peer and a local worker	10 Mbps
Size of input data files	7.2 Mbytes
Mean job execution time	500 s
TTL parameter for job and data queries	4
Number of jobs, N_{job}	from 250 to 10000
Number of Data sources, N_{ds}	1, 2, 3, 5, 7, 9, 11, 13

download and execution periods are alternated more efficiently if workers execute a larger number of jobs. But this increase tends to be negligible as the number of jobs is so large that the job submission system begins to approach a stable working condition.
Figure 5.3 reports the average percentage of time in which a generic DS supports at least one download connection. Results confirm that the presence of an excessive number of DSs can be inappropriate, especially if the number of jobs is not very large. Indeed when the percentage of activity time decreases below 60%, machine utilization is very low resulting in a poor return of investment (ROI).
Figures 5.4 and 5.5 show performance results related to workers. Figure 5.4 proves that the download time decreases as the number of DSs is increased, resulting in smaller overall execution time. On the other hand, the download time hardly depends on the number of jobs because the simultaneous number of connections that a DS must serve is only related to the number of workers (250), not to the number of jobs. Finally, figure 5.5 compares number of jobs executed by a worker on average (obtained as $N_{job}/250$) to the maximum number of jobs executed by a single worker, dotted line. It is interesting to note that the two indices approach one another as the number of DSs is increased, leading to a fairer load balancing among workers.
Finally figure 5.6 show network load, this by number of messages traveling over the net per second, respect to number of DSs. Network load grows, this because the fast download in fact the single W mean download time decreases as seen in figure 5.4.

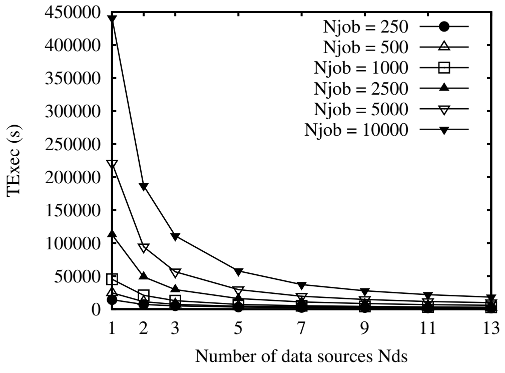


Fig. 5.1. Overall execution time vs. the number of data sources for different numbers of jobs.

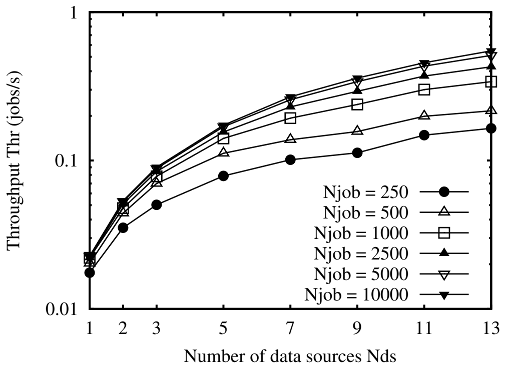


Fig. 5.2. Throughput vs. the number of data sources for different numbers of jobs.

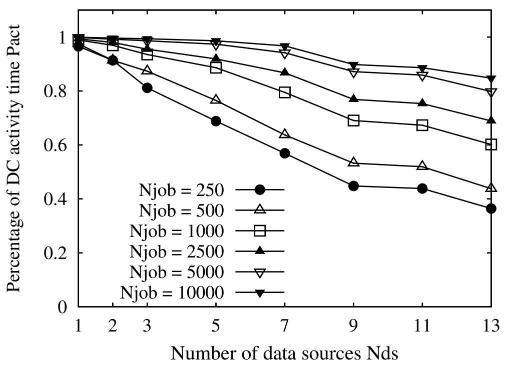


Fig. 5.3. Percentage of activity vs. the number of data sources for different numbers of jobs.

5.2 Redundant submission of job scenario

Respect to the base scenario, redundant executions of each job can be submitted to more peers, obviously not less peers then how many executions are

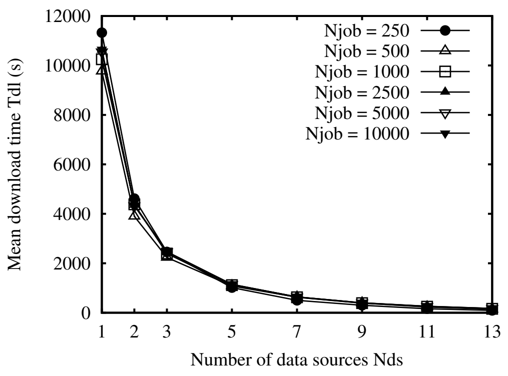


Fig. 5.4. Mean time to download an input file vs. the number of data sources for different numbers of jobs.

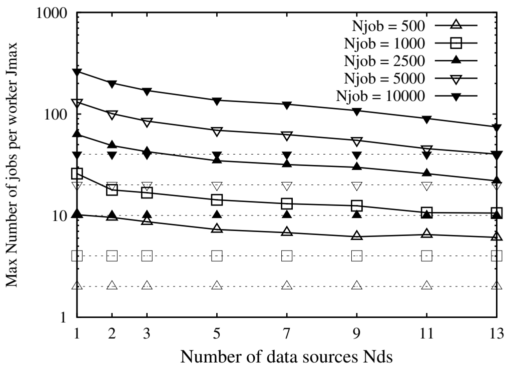


Fig. 5.5. Maximum number of jobs executed by a single worker vs. the number of data sources, for different numbers of jobs. This index is compared to average number of jobs executed by single worker (dotted lines).

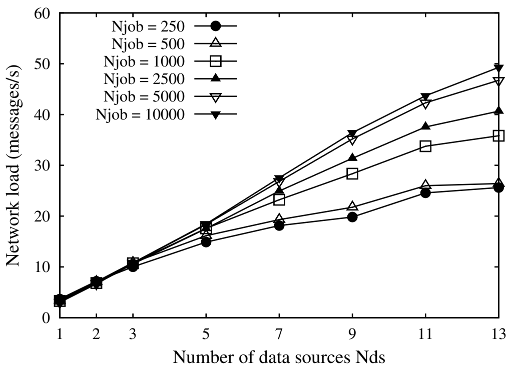


Fig. 5.6. Network load vs. the number of data sources for different numbers of jobs.

required N_{exec}.
In this scenario simulations the DSs number is setted with value 13, and then has been analysed how performance indices changes when multiple executions of each job is submitted to more then 10 peers with $N_{exec} = 10$. The number of jobs varies from 100 to 1000.
Scientific analysis performed inside *PRC* context, can require multiple execution of every single job, either to enhance statistical accuracy or minimize the effect of malicious executions. The parameter N_{exec} is set to 10 in this analysis. To achieve this objective, redundant job assignment is exploited: each job advert can be matched and assigned to workers up to a number of times equal to the parameter *Matches To Live (MTL)*, whose value must be not lower than N_{exec}. The job manager assigns a job until either the MTL parameter is decremented to 0 or the job manager receives the results for at least N_{exec} executions of this job. A proper choice of MTL can compensate for possible disconnections of workers and consequent job failures.
The performance parameters, starting from the T_{Exec}, are examined respect to NJob and multiple submissions time (MTL) variation in certain conditions.

Figure 5.7 shows the overall execution time vs. the MTL value, with the number of jobs N_{job} ranging from 50 to 500. The execution time tends to decrease as the value of MTL increases, then it gets stabilized.
The reason of this is that a larger MTL allows to better compensate for the possible hetereogeneity of job execution time duration, it's reasonable to imagine that al last 10 jobs end on time having assigned 50 jobs to hetereogenus workers. This represents a further test that our simulator can capture W with different behaviour and it captures the scenario that a job can take longer for a W and shorter for one other, exactly as in real life it happens.
All the jobs ended later are not taken under account, then it means network resources, see at figure 5.9 and 5.10, and peer working time waste, but this last point is not a big deal in *PRC* scenario.
This effect disappears once the MTL exceeds a threshold: in fact very large values of MTL are not exploited because the job manager stops the assignments of job adverts when output data related to N_{exec} executions have been received.

Figure 5.8 shows that the average utilization of data centers, and hence the efficiency of the protocol, increases with the amount of computation assigned to workers, i.e., with the number of jobs and, more slightly, with the MTL value. To understand this, it must be considered that data cachers are not heavily utilized in the first phase of the process, because they have not yet retrieved data from the data source, whereas they are fully exploited only after they have retrieved such data. Therefore, the utilization of data centers is high only when the number of required job executions is large enough to make the caching of data convenient. On the other hand, when the amount of computation is low, the time interval required by data cachers to retrieve

data files is relevant with respect to the overall execution time, therefore data cachers are not exploited for a large fraction of time, which explains the low values of the utilization index.
After this preliminary set of experiments, it was decided to set MTL to a constant value, in order to better evaluate the effect of other parameters and configuration options. The value of MTL was set to the lowest value for which the overall execution time is at most 10% higher that the "steady" execution time, for every tested value of N_{job}. This steady value was set as the execution time obtained with MTL equal to 50, beyond which no further variations of T_{exec} can be perceived. According to this strategy, the set of MTL was then set to 20. Furthermore, this value allows for the successful execution of jobs in most of the considered scenarios, as will also be discussed in the next section.

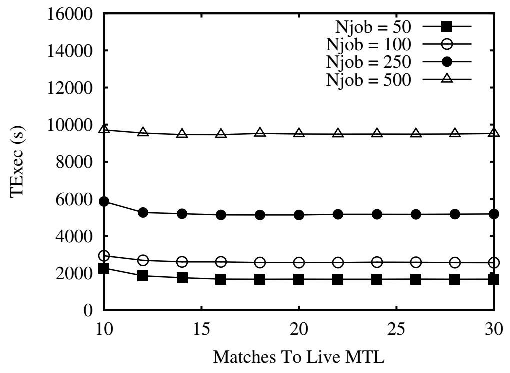


Fig. 5.7. Overall execution time vs. the value of MTL for different numbers of jobs.

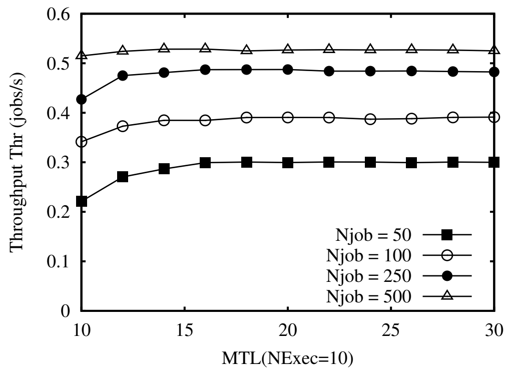


Fig. 5.8. Throughput vs. the value of MTL for different numbers of jobs.

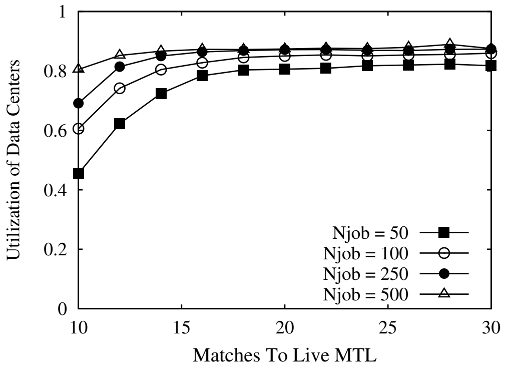


Fig. 5.9. Percentage of activity vs. the value of MTL for different numbers of jobs.

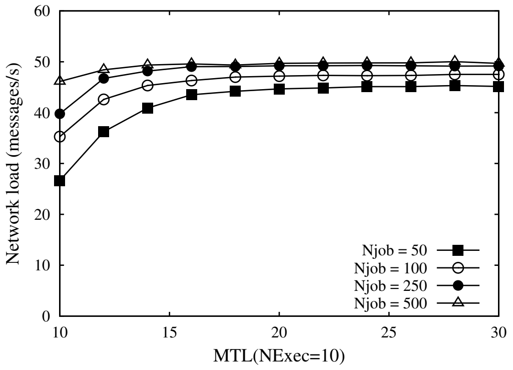


Fig. 5.10. Network load vs. the value of MTL, for different numbers of jobs.

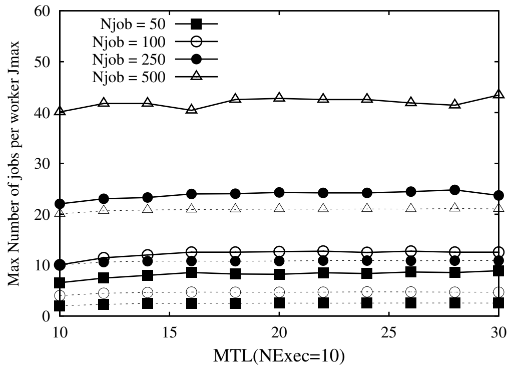


Fig. 5.11. Maximum number of jobs executed by a single worker vs. the value of MTL for different numbers of jobs. This index is compared to average number of jobs executed by single worker (dotted lines).

5.3 With data caching scenario

The base scenario can be extended removing the assumption that data centers already acquired all data they could need in start up. Once a peer asks for data, a DC acquires data dynamically and start to deliver them (totally caching) even before they are not yet all available (partially caching). The effectiveness of the dynamic caching mechanism has been evaluated in different experiments. Simulations were performed for a network analogous to that examined before.

This analysis essentially compares how our approach may affect a BOINC-like network if the administrator provides more data cachers into the network. BOINC-like applications infact are able to replicate their current static data server functionality through a dynamic and decentralized data distribution system that enables projects to automatically scale their data needs without additional administrative overhead as their user-base or problem size increases.

Instead of 13 DSs at the beginning of simulation, here we have one DS with required data and 12 DCs ready to dynamically download data. Making a comparison between without and with caching scenario performances indices behaviour there is not a big difference in quality. For example analyzing figures 5.7 respect to 5.12 we can see that T_{exec} grows faster in the second scenario. Comparing 5.8 and 5.13 the throughtput decreases in the second because Ws must wait for DSs loading all the data. Comparing 5.9 and 5.14 you can avvert a diminution of DCs utilization in fact they are not sending data until they don't finish caching them. Comparing 5.10 and 5.15 message interchange in the network increases in the second case because *data query messages.*

The differences among with and without caching scenario are more evident specially at job number decreasing. In fact in the case of a shorter execution time, the preliminary caching phase affects proportionaly a longer time, in the opposite, in the case of a longer execution, the caching phase can be ignored. Introduce caching mechanism then allows to simulate a more realistic scenario without lose too much in performances.

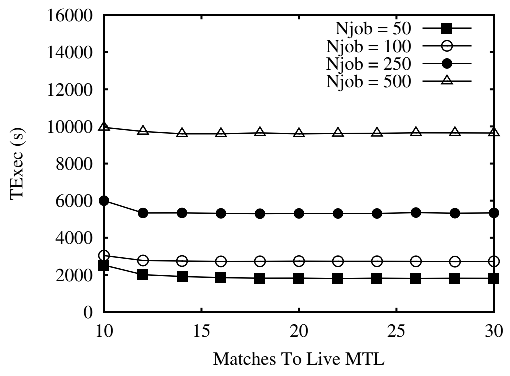


Fig. 5.12. Overall execution time vs. the value of MTL for different numbers of jobs.

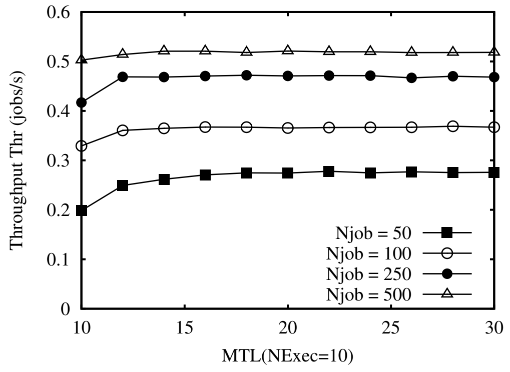


Fig. 5.13. Throughput vs. the value of MTL for different numbers of jobs.

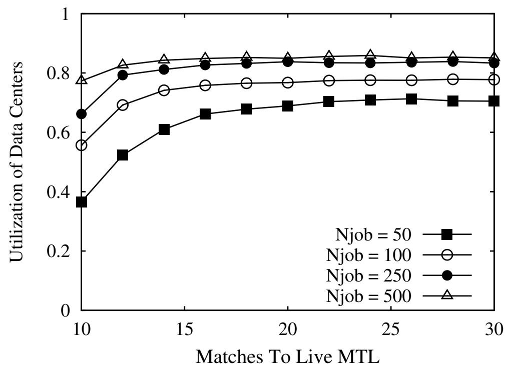


Fig. 5.14. Percentage of activity vs. the value of MTL for different numbers of jobs.

To proof that really the introduction of data caching mechanism doesn't affect the overall performances, further analysis have been done. This time in total DCs are 13, the number of available DSs is varied from 1 to 13 by step 2, the number of DCas is the complement.

Number of data sources variation affects slightly performance indexes so it seems reasonable to project the network in a way that at the beginning there are not more then one DS, in fact it seems that it makes no difference if there are many DSs or not.
In the following only DCas number changes and the number of DSs is one. Anyway the results are almost the same obtained in first scenario.

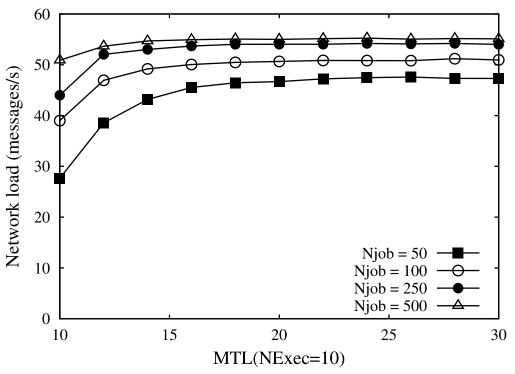


Fig. 5.15. Network load vs. the value of MTL for different numbers of jobs.

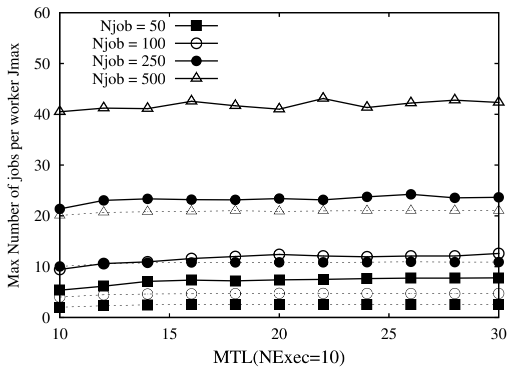


Fig. 5.16. Maximum number of jobs executed by a single worker vs. the value of MTL for different numbers of jobs. This index is compared to average number of jobs executed by single worker (dotted lines).

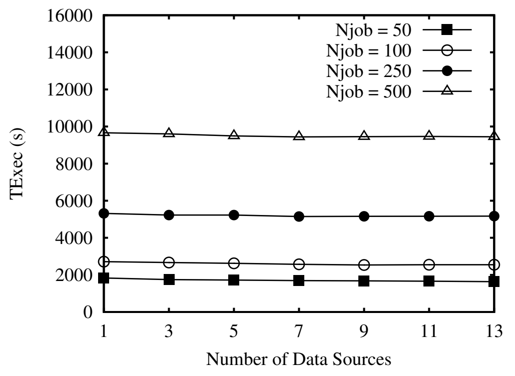


Fig. 5.17. Overall execution time vs. the number of data sources for different numbers of jobs.

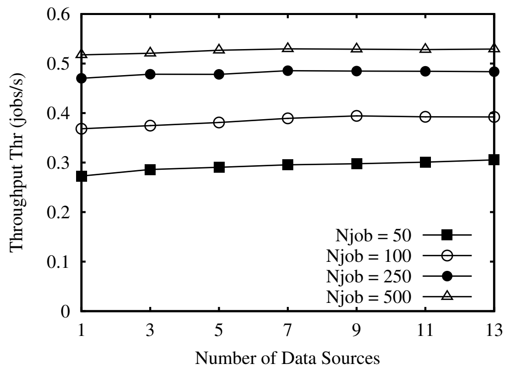


Fig. 5.18. Throughput vs. the number of data sources for different numbers of jobs.

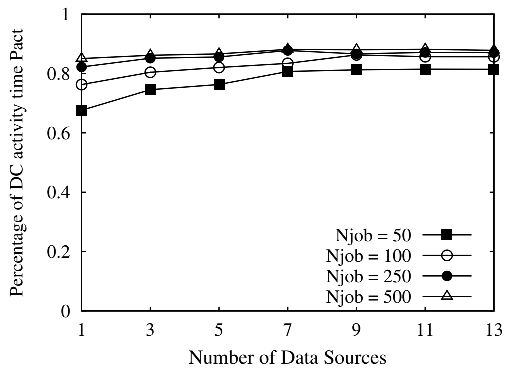


Fig. 5.19. Percentage of activity vs. the number of data sources for different numbers of jobs.

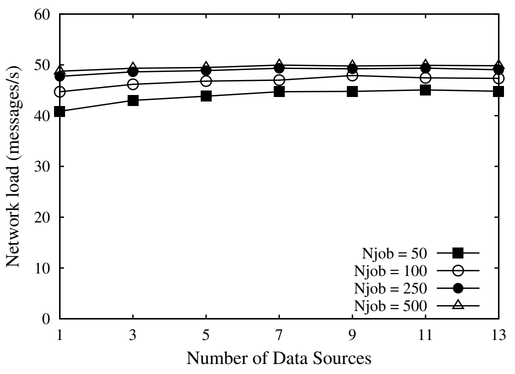


Fig. 5.20. Network load vs. the number of data sources for different numbers of jobs.

5.4 Peers disconnection scenario

In a more realistic scenario Ws could end a job out of date or not end it. It happens respectively according the peer structure and because peers could

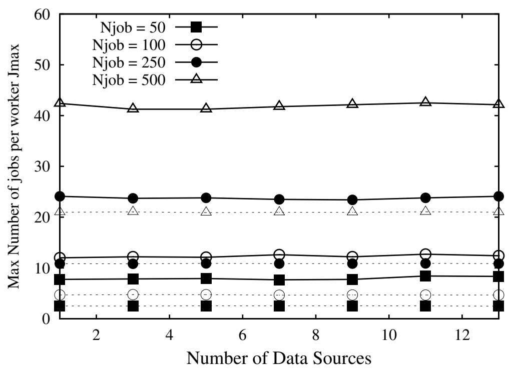


Fig. 5.21. Maximum number of jobs executed by a single worker vs. the number of data sources for different numbers of jobs. This index is compared to average number of jobs executed by single worker (dotted lines).

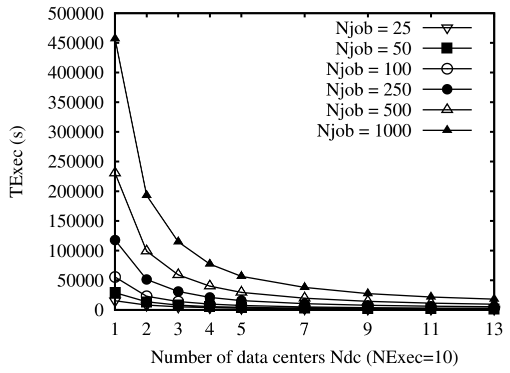


Fig. 5.22. Overall execution time vs. the number of data centers, for different numbers of jobs.

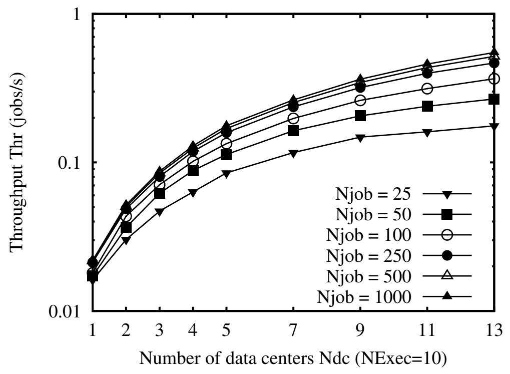


Fig. 5.23. Throughput vs. the number of data centers, for different numbers of jobs.

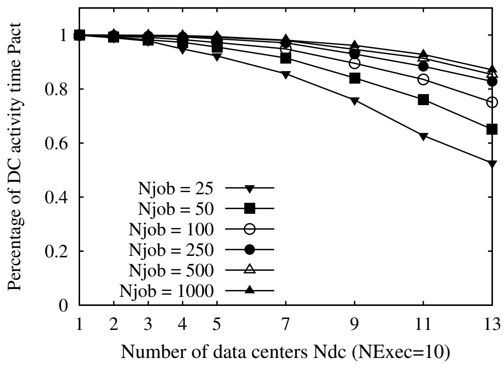


Fig. 5.24. Percentage of activity vs. the number of data centers for different numbers of jobs.

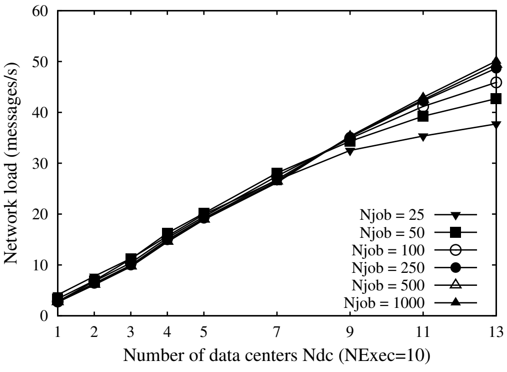


Fig. 5.25. Network load vs. the number of data centers for different numbers of jobs.

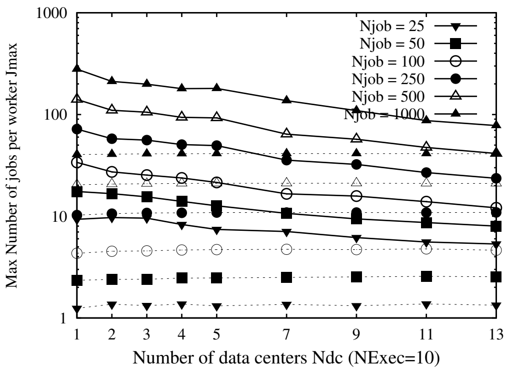


Fig. 5.26. Maximum number of jobs executed by a single worker vs. the number of data centers for different numbers of jobs. This index is compared to average number of jobs executed by single worker (dotted lines).

disconnect. In general all nodes can disconnect except SP node and JM. The mechanism to handle with disconnection follows two behaviours:

1. a priori assignment of jobs with fixed MTL value: the idea is to assign all required jobs to peers and to activate a timer for the result, all the results that haven't been calculated on time aren't taken into account for the final result.
2. dynamically re-assignment jobs with fixed minimum MTL value: the idea is to assign all required jobs to peers and to activate a timer for the result, while there is a job which results haven't been calculated on time, the job has assigned to a peer again with a timer. Eventual results that will be calculated on late are accepted, even if they have been re-assigned.

The JM starts the job assignment phase. Please notice that anytime a JA assigns a job, it automatically increases the available assignment value without even know if such job will ended on time. In the first situation when the worker doesn't deliver the results before timeout, the partially completed job execution has been totally waste, so it makes sense to calculate well the timer value. A timer value is good if the job totally completed percentage is high. The JA starts the timer only once the it has assigned all the jobs to the workers. The timer value is chosen according an intuitive algorithm which uses input and intermediate calculated parameters:

- NJobTot: effectively total job required number.
- currentT: current time in millisecond.
- NCompletedJob: represents the number of job which result has been already calculated until currentT.
- NIncompletedJob: represents the number of job which result hasn't been yet calculated at currentT.
- TPerJob: job estimated execution average time.

The Timer is calculated as $Timer = C \cdot (NIncompletedJob \cdot TPerJob)$, where the constant value C (10 in our case) has been chosen according different simulation running.
In the base scenario only the Job-Manager could end the simulation once it receives the results for any job, now it can be ended also by the JA once the timer expires. For this scenario T_{exec} parameter has been analysed. In the case MTL value is exactly equals or near to NExec value, TExec strictly depends from the timer imposed by JA. When $MTL >> N_{exec}$, obviously T_{exec} is lower and the simulation is ended by the JM.

In the second case the timer rule assumes a different meaning, in fact a timer is set per each job assigned and not to the whole simulation. For this reason the Job Assigner node has to memorize the status (assigned or not, completed or not) for each job. Anytime one job timer expires, the Job Assigner checks the job status, if it's assigned but not yet completed then it put the status to not assigned and it allows the job to be re-assigned and executed.

The idea is that if the job timer expires because the worker node hasn't yet finished the computation, at the end this job result will be delivered anyway, even if on late.
For each re-assigned job a new timer value has been calculated dynamically according the well known Jacobson algorithm. This re-assignment approach guarantees the wished minimum N_{exec} has been reached.
Every time a worker ends a job execution it sends a job results message to the JM that sends a *jobCompletionAdvert* message to the JM so it can update the current job status. As well as in the base scenario only the JM can end the simulation once it receives the expected results for any job.

This second case requires a complex protocol, it requires additional data structure and computation resources, and all that it's far from PRC approach and out of our modelization intension. For these reasons this second case haven't been further analyzed but only the first.
In the following new simulation parameters have been introduced because disconnection and they are the connection and the disconnection time, respectively 3 hours and 1 hours in our case. Firstly a network with 1 DS and 12 DCas have been used. The performance index for this scenario have been evaluated and can be compared with the analogous DS and DCas situation in the section 5.3.

Figure 5.27 shows the overall execution time vs. the MTL value, with the number of jobs N_{job} ranging from 50 to 500. The execution time tends to decrease as the value of MTL increases, then it gets stabilized, this behaviour reminds at without disconnection scenario.
Note that the execution time could not be computed for values of MTL next to T_{exec} (i.e. lower then 13 if N_{job} is 500). In such cases, worker disconnections do not allow to perform at least N_{exec} executions for each job. Therefore, the redundant approach is not only useful to decrease the execution time, but it is even necessary to complete the required job executions. Not depending from NJob and MTL the T_{exec} is bigger then in without disconnection scenario, see at figure 5.12.
The throughput value is lower respect the base scenario, obviously because the working time of a peer that can disconnect is discontinuos, this seems clear at first look of figure 5.13 and 5.28, same consideration work for the Network load value, this consideration comes from the comparison of figure 5.15 and 5.30.

Figure 5.29 shows that the average utilization of data centers, and hence the efficiency of the protocol, increases with the amount of computation assigned to workers, i.e., with the number of jobs and, more slightly, with the MTL value. To understand this, it must be considered that data cachers are not heavily utilized in the first phase of the process, because they have not yet retrieved data from the data source, whereas they are fully exploited only after they have retrieved such data. Therefore, the utilization of data centers is high only when the number of required job executions is large enough to make the caching of data convenient. On the other hand, when the amount of computation is low, the time interval required by data cachers to retrieve data files is relevant with respect to the overall execution time, therefore data cachers are not exploited for a large fraction of time, which explains the low values of the utilization index, see at figure 5.14.

In figure 5.16 respect to figure 5.31 the maximum number of jobs executed by a single worker increases. This happens because peers of DC cluster, since they are the faster nodes in downloading phase, take few time to complete a job and even if we are in the case of disconnection, generally they have a lower number of interrupted jobs.

These analysis brings to the consideration that opportunly choose MTL value is mandatory to optimize the execution but also this time to allow the end of N_{exec} executions for each job.

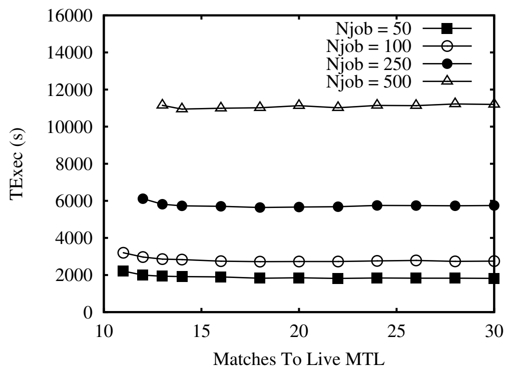


Fig. 5.27. Overall execution time vs. the value of MTL for different numbers of jobs.

Now the case of DCas variable has been analyzed with MTL=20, this case aims at evaluating the effectiveness of the dynamic caching mechanism.

Simulations were performed for the same network as before, except that the number of available data centers is varied from 1 to 13: one of these data centers is the data source, the others are data cachers.

Figure 5.32 shows the values of the overall execution time calculated for this scenario. The time decreases as more data centers are made available in the network, for two main reasons:

- data centers are less heavily loaded and therefore data download time decreases,

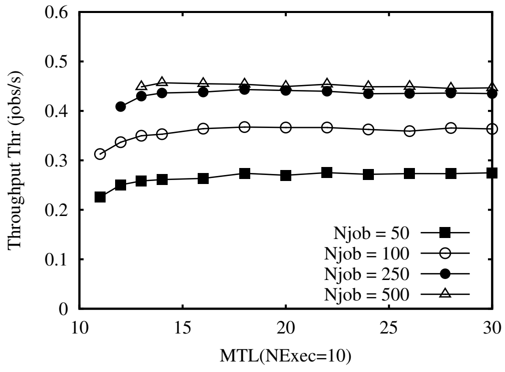


Fig. 5.28. Throughput vs. the value of MTL for different numbers of jobs.

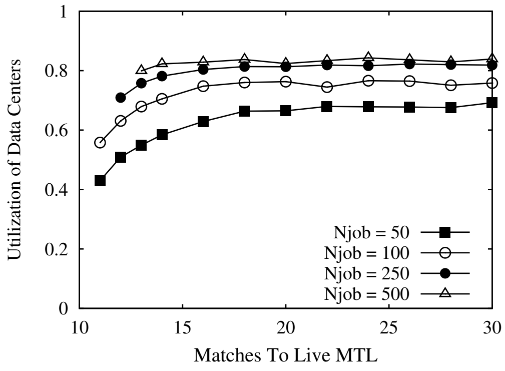


Fig. 5.29. Percentage of activity vs. the value of MTL for different numbers of jobs.

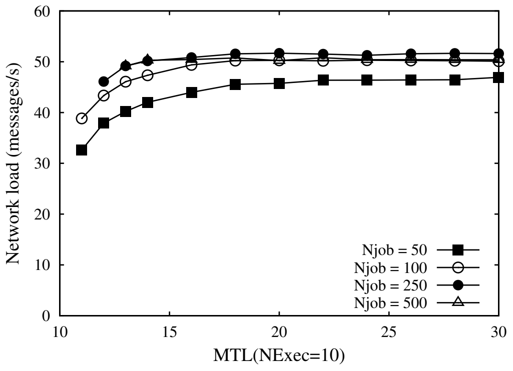


Fig. 5.30. Network load vs. the value of MTL,s for different numbers of jobs.

- workers can exploit a higher parallelism both in the downloading phase and during the execution of jobs.

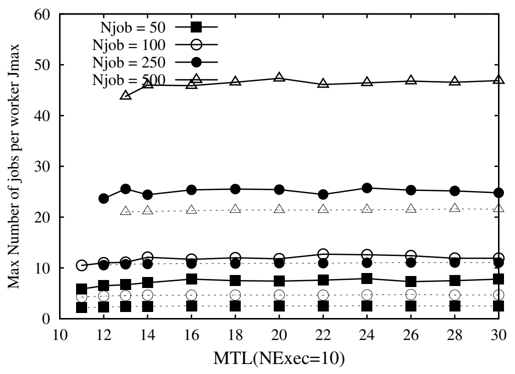


Fig. 5.31. Maximum number of jobs executed by a single worker vs. the value of MTL, for different numbers of jobs. This index is compared to average number of jobs executed by single worker (dotted lines).

Depending on the number of jobs to be executed, it is possible to determine a suitable number of data centers, beyond which the insertion of a further data center produces a very low decrease of execution time, or even a small increase of it. For example, if the number of jobs is 250, a significant reduction of T_{exec} is perceived as the number of data centers is increased up to a value of 7, whereas if the number of jobs is 25 or 100, 5 data centers are sufficient to achieve a good performance level, and adding more data centers is not effective.

Figure 5.32 does not report results for some combinations of the number of DCs and the number of jobs, because the disconnections of workers do not allow for the completion of all the required job executions.

In fact, if only a few data centers are available, each of these is likely to be overloaded by a large number of workers' requests; as a consequence, the download time increases and the disconnection of a worker during the download phase becomes a more probable event. Analyzing it we can choose the right number of DCs to be sure that the jobs execution will end.

It is very hard to deduce the overall execution time in an analytical way, due to the large number of network parameters (e.g., the number of workers and super-peers, the bandwidth and latency between nodes and so on) and the complexity of the super-peer protocol. However, we made several tests with different application scenarios and used Matlab tools to obtain a mathematical expression that is able to approximate simulation results as much as possible and at the same time is coherent with the dynamics of the protocol. We derived the expression shown in formula (5.1), that relates the overall execution time to the number of jobs to execute, N_{job}, and to the number of available data centers, N_{dc}.

$$T_{exec} = C_1 log(N_{dc}) + C_2 \frac{N_{job}}{N_{dc}} + C_3 \frac{N_{job}}{(N_{dc})^2} \tag{5.1}$$

Very interestingly, we found that this expression is valid for all the performed tests, regardless of the values of the other network parameters. Of course, the impact of these parameters is encompassed by the values of the coefficients that appear in formula (5.1).

The expression in the formula is composed of three terms, each of which can be associated to a basic characteristic of the protocol. In particular, the first term relates to the dissemination of input data to the network data centers. This term is logarithmic with respect to N_{dc}, because each data cacher, after retrieving data from a data center, is able to provide this data to a number of other data cachers. Due to the log-type relation, this term increases slightly with the number of data centers. The second term takes into account the time needed by workers to download data files from a single data center and execute the corresponding jobs (indeed we can consider one data center, since operations are made in parallel on different data centers). Specifically, this term is proportional to the average number of jobs which require a download operation from a single data center, N_{job}/N_{dc}. Finally, the third term gives an estimation of the additional amount of time that is required by worker disconnections. This "extra" time corresponds for the most part to the time taken by download operations that have to be re-executed because they failed during their first try. In fact, the third term comes out as the product of the time that would be taken if all the download operations failed (which is proportional to N_{job}/N_{dc}) and the probability that a single download operation actually fails. It was found that this probability is inversely proportional to the number of data centers, since download operations are longer and more at risk of failure as the number of data centers decreases. The third term takes into account only the possibility of repeating a download operation just once. The impact of multiple repetitions of download operations is actually negligible if the failure probability is much lower than 1. In conclusion, the overall execution time is the sum of three terms, of which the first increases with the number of data centers while the other two decrease. However, the effect of the first term is relatively low, except for the cases in which the number of data centers is large and the number of jobs is small, as can be seen in Figure 5.32. In fact, in such cases the overall execution time slightly increases with the number of data centers.

Figure 5.34 shows the average utilization of data centers for the same scenario. This index decreases as the number of data centers increases and, in contrast with the execution time, curves do not get to a relatively stable value. This is another useful indication for setting a proper number of data centers. For example, let's consider the submission of 500 jobs. While the overall execution time can be decreased until the number of data centers is increased to about 11, the utilization index continue to decrease as more data centers are made available. With 13 data centers there would be a worse exploitation of data centers and no significative reduction in the execution time, from which it can be concluded that an appropriate number of data centers is indeed 11.

Figure 5.35 shows the number of messages per second that circulate in the network. It can be noticed that the network load increases as DCs number increases, specially non depending by jobs number, in the case of a small number of DCs.

Finally, Figure 5.36 helps to examine the load balancing features of the protocol. Figure 5.36 reports the maximum number of jobs executed by a single worker. It is interesting to note that the a wider availability of data centers improves load balancing among workers, as the maximum number of executed jobs decreases and approaches the average number. In fact, with few DCs, the workers which are closer to them tend to execute more jobs, because the download phase is faster. However, if more data centers are installed, the differences among workers are attenuated, in particular when the overall computation load is high.

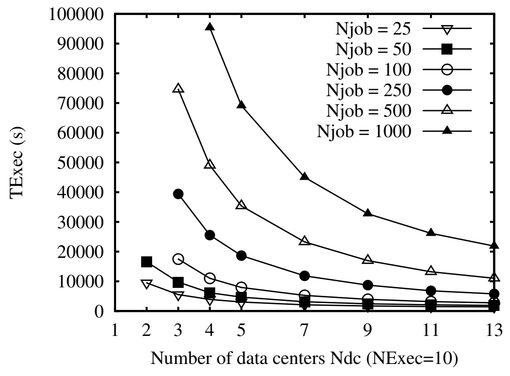


Fig. 5.32. Overall execution time vs. the number of data centers, for different numbers of jobs.

5.5 Scalability analysis

An additional set of simulations were performed to evaluate the behavior of the protocol in variable-sized networks, to specifically examine its scalability. We analyzed networks having 250, 500 and 1000 workers, that is with 25, 50 and 100 super-peers, respectively. As in the previous simulations, one data source is available, while the overall number of data centers, including this data source and the data cachers, is varied from 1 to half the number of

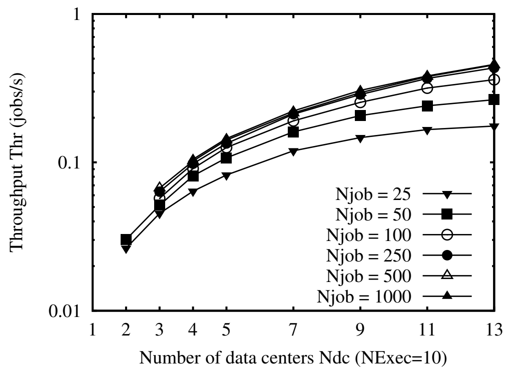


Fig. 5.33. Throughput vs. the number of data centers, for different numbers of jobs.

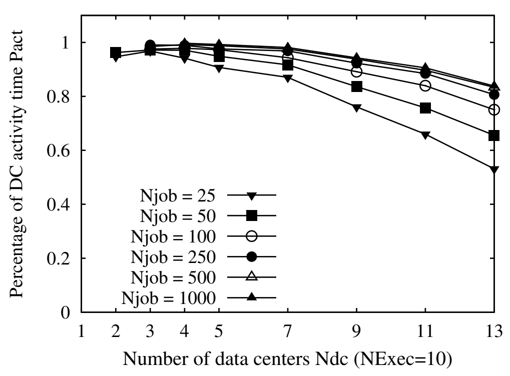


Fig. 5.34. Percentage of activity vs. the number of data centers, for different numbers of jobs.

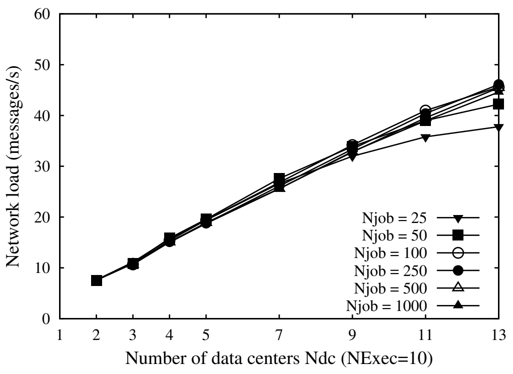


Fig. 5.35. Network load vs. the number of data centers for different numbers of jobs.

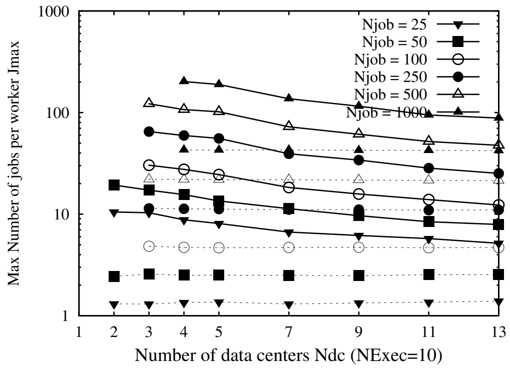


Fig. 5.36. Maximum number of jobs executed by a single worker vs. the number of data centers, for different numbers of jobs. This index is compared to average number of jobs executed by single worker (dotted lines).

super-peers (the maximum value is approximated to 13 in the case of 25 super-peers). The required number of executions of each job, N_{exec}, is set to 10, while the maximum number of assignments per job, MTL, is set to 20. All such features are summarized in table 5.2.

Table 5.2. Simulation scenario

$Scenario feature$	*Value*
Number of workers, N_{peer}	250 to 1000
Average number of workers connected to the one super-peer	10
Maximum number of neighbors of a super-peer	4
Average connection time of workers	4 h
Average disconnection time of workers	1 h
Number of data centers (1 data source + data cachers)	1 to 50% of super-peers
Size of input data files	7.2 Mbytes
Latency between two adjacent super-peers (*or between two remote peers in a direct connection*)	100 ms
Latency between a super-peer and a local worker	10 ms
Bandwidth between two adjacent super-peers (*or between two remote peers in a direct connection*)	1 Mbps
Bandwidth between a super-peer and a local worker	10 Mbps
Number of jobs, N_{job}	100 to 1000
Number of executions requested for each job, N_{exec}	10
Matches to live, MTL	10 to 25
Mean job execution time	500 s

Two different scenarios are taken into consideration [86]. In the first, the number of jobs is constant, and set to 500, while the number of workers is increased: this is useful to verify if the availability of more workers can actually improve performance. In the second scenario, the number of jobs and the number of workers are increased with the same pace: this analysis is particularly useful to verify if the protocol is able to sustain an increase in the problem size in the case that the average computational load of a worker is maintained constant.

Figure 5.37 shows results related to the first scenario. Because of the high range of the values obtained, a logarithmic scale is adopted for the y axis. This figure shows that, with a constant problem size, that is, a fixed number of jobs, the availability of a larger number of workers actually improves performance, on condition that a proportional number of data centers are installed. For example, if half the number of super-peers are set up as data centers, the execution time is reduced by about 50% (from 11023 seconds to 5421 seconds) if the number of workers is increased from 250 to 1000. The performance gain is therefore relevant.

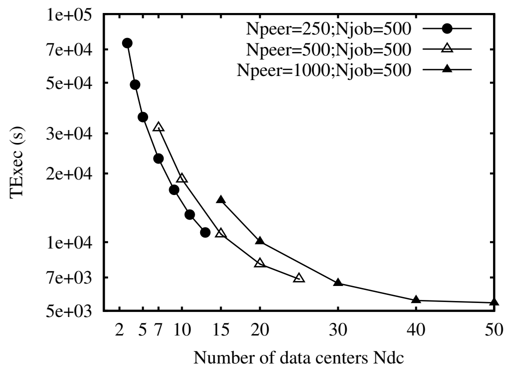


Fig. 5.37. Overall execution time vs. the number of data centers. The number of jobs is set to 500, and different network sizes are tested.

Figure 5.38 reports the overall execution time for the second scalability test, namely the test with variable problem size, which aims to verify if it is effective to increase the number of workers proportionally to the amount of computation load. Here the number of workers is maintained equal to the number of jobs, each of which - it is useful to recall - must be executed at least 10 times. Again, the results obtained when using a fixed percentage of data centers are to be compared. The dashed line in figure 5.38 connects points obtained when the mentioned percentage is set to 50% and shows that the proposed approach is satisfactory scalable. In fact, the execution time is equal to 5770 seconds when the number of jobs (and the number of workers) is 250, while it increases to 6886 seconds with 500 jobs, and to 8301 with 1000 jobs. Therefore, if the amount of computation is doubled, the execution time increases by only 19%, while if it is quadruplicated, the execution time

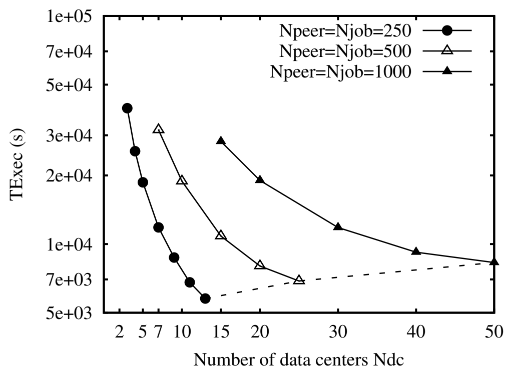


Fig. 5.38. Overall execution time vs. the number of data centers for different numbers of jobs. The network size is proportional to the amount of computation.

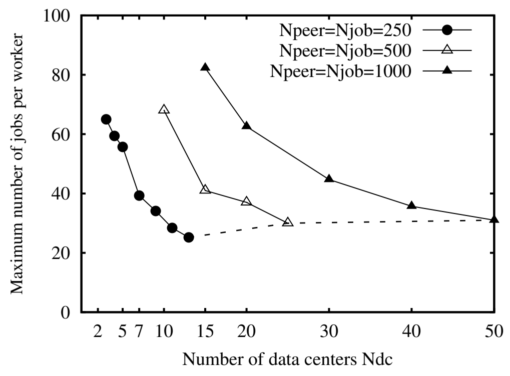


Fig. 5.39. Maximum number of jobs performed by a worker vs. the number of data centers, for different numbers of jobs. The network size is proportional to the amount of computation.

increases by only 43% per cent. Analogous results are obtained for different percentages of data centers in the network. This proves the good scalability of the approach presented in this paper, as the pace at which the execution time increases is much lower than the corresponding increase in the problem size.

Scalability actually derives from the ability of the protocol to fairly distribute the computational load to workers, even when the problem and the network size increase. This can be observed in Figure 5.39: the dashed line highlights the values of the maximum number of jobs executed by a worker, obtained when the number of data centers is equal to 50% of super-peers. The maximum number of jobs is 25.2 in a network with 250 workers, and increases very slightly with the computational load: specifically, it increases to 30.0 and to 31.1 as the number of jobs increases to 500 and 1000, respectively.

5.6 Multiple jobs assigner scenario

Instead of manipulating the TTL value by increasing it respect to the growing of the network, it's possible to change the JA number in a way to allow all the Ws obtaining a job assignment [87]. This is possible inside our model, in fact in the beginning of simulation it's possible to consider more then one JA and spread to them *job advert* messages. Increasing the JAs number Ws can, with higher probability, find a JA in their neighborhood. Then it's reasonable to use a low TTL value. An indirect consequence is that a smaller number of queries cross the network, see at figure 5.43 and 5.44, and in a shorter time a JA is found. This is a sign of communication protocol scalability.
The drawback is that if the TTL value is too small, some Ws will never find their JA; then they can't work as you can see examining the load balancing features of the protocol in figure 5.45 and from T_{exec} value in figure 5.40 that increases at decreases TTL.

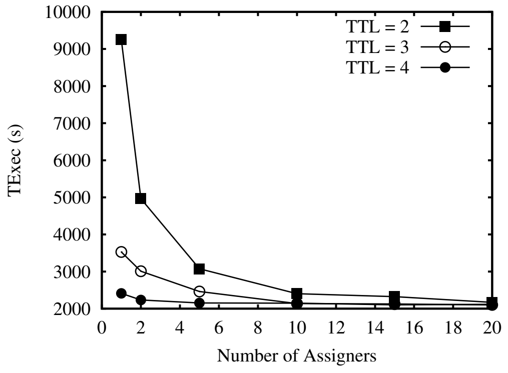


Fig. 5.40. Overall execution time vs. the number of Assigner for different TTL.

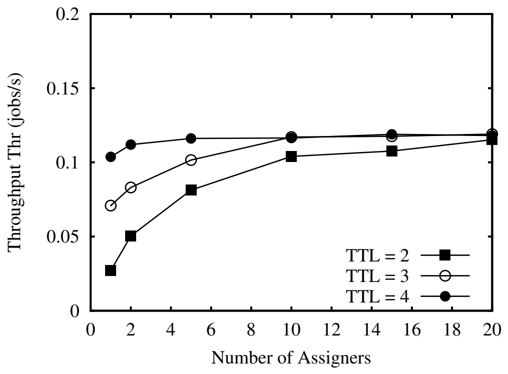


Fig. 5.41. Throughput vs. the number of Assigner for different TTL.

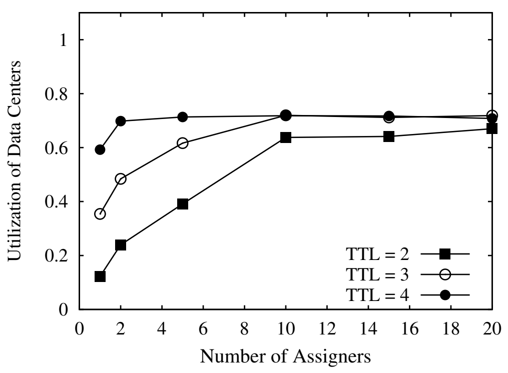


Fig. 5.42. Percentage of activity vs. the number of Assigner for different TTL.

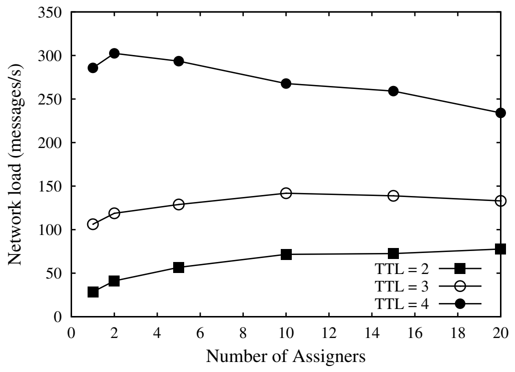


Fig. 5.43. Network load vs. the number of Assigner for different TTL.

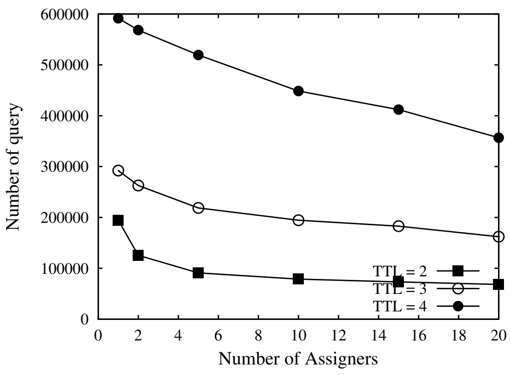


Fig. 5.44. Number of Query vs. the number of data centers for different TTL.

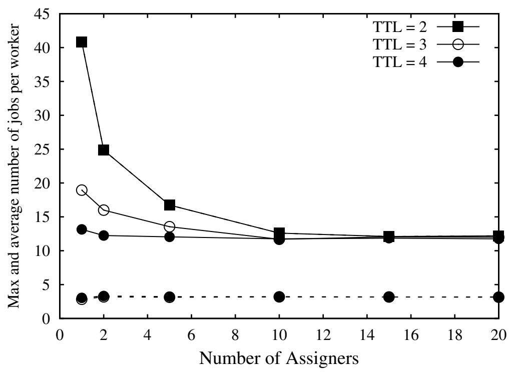


Fig. 5.45. Maximum number of jobs executed by a single worker vs. the number of Assigner for different TTL. This index is compared to average number of jobs executed by single worker (dotted lines).

6

P2P networks for music information retrieval

In this chapter it is analyzed the simulations results of the distribution of bundled workflows across ubiquitous peer-to-peer networks for music information retrieval in the context of the DART project [89, 88]. The DART project aims to develop a novel music recommendation system (MRS), by gathering statistical data using collaborative filtering techniques and the analysis of the audio itself in order to create a reliable and comprehensive database of the music that people own and which they listen to.
Mrs DART [90] employs the use of the underlying Distributed Audio Retrieval using Triana (DART) peer-to-peer subsystem in order to provide a fan-out mechanism for distributing workflows across the network and for retrieval and aggregation of results.
In the DART scenario, the home users, the Ws, perform analysis of their own music collection by executing Triana workflows that encompas the analysis to be performed at that time, according to PRC paradigm.
Triana [89] is a graphical Problem Solving Environment (PSE) for composing data-driven applications by executing workflows [94, 92]. Given its modularity, its support for high quality audio, and its ability to distribute processes across a Grid of computers, Triana has the potential to be an extremely useful piece of software that allows users to implement custom audio processing algorithms from their constituent elements, no matter their computational complexity.
DART uses a similar approach to BOINC [71] but differs in that the workers receive input data in the form of a bundled Triana workflow, which is executed in order to process any MP3 files that they own on their machine. Once analysed, the results are returned to DART's distributed database that collects and aggregates the resulting information. DART employs the use of package repositories, the DC, to decentralise the distribution of such workflow bundles. This approach is validated through simulations that show that suitable scalability is maintained through the system as the number of participants increases.

6.1 DART scenario

Dart project utilises a combination of distributed systems technologies, its goal is to leverage this technology such that the some kind of digital signal processing can be achieved with audio rate signals for the purposes of signal analysis, feature extraction, synthesis, and music information retrieval (Mrs DART).
DART uses a peer-to-peer approach, it is based on the super peer architecture but extends this idea to employ the use of secure Data Servers (called package repositories) that cache the workflow bundles for DART, to be able to replicate and decentralise the distribution of the workflows. Other techniques were considered, such as Bittorrent but this is unacceptable within the BOINC framework because of security constraints.
In DART, we are interested in forming unstructured $P2P$ networks and therefore need to employ technologies that can adapt and scale within such an environment. For distribution across dynamic networks, we use the P2PS binding for Triana. Peer-to-Peer Simplified (P2PS) [91] was a response to the complexity and overhead associated with JXTA [73]. As its name suggests, it is a simple yet generic API for developing P2P systems. P2PS encompasses intelligent discovery mechanisms, pipe based communication and makes it possible to easily create desirable network topologies for searching, such as decentralised ad-hoc networks with super peers or rendezvous nodes. P2PS is designed to cope with rapidly changing environments, where peers may come and go at frequent intervals.
At the core of P2PS is the notion of a pipe: a virtual communication channel that is only bound to specific endpoints at connection time. When a peer publishes a pipe advertisement it only identifies the pipe by its name, ID, and the ID of its host peer. A remote peer wishing to connect to a pipe must query an endpoint resolver for the host peer in order to determine an actual endpoint address that it can contact. In P2PS a peer can have multiple endpoint resolvers (i.e. TCP, UDP etc), with each resolving endpoints in different transport protocols or returning relay endpoints that bridge between protocols (i.e. to traverse a firewall). Also, the P2PS infrastructure employs XML in its discovery and communication protocols, which allows it to be independent of any implementation language and computing hardware. Assuming that suitable P2PS implementations exist, it should be possible to form a P2PS network that includes everything from super-computer peers to PDA peers.

To simulate this application a new protocol has been modelized, in this scenario, see figure 6.1, we represent the decentralised DART network in which nodes are organized in a super peer topology using the P2PS middleware. In P2PS, producers (i.e. providers of packages containing workflows or results) create *adverts* to advertise that they have something that is of interest to other participants in the network. Consumers (i.e. the peers that wish to use

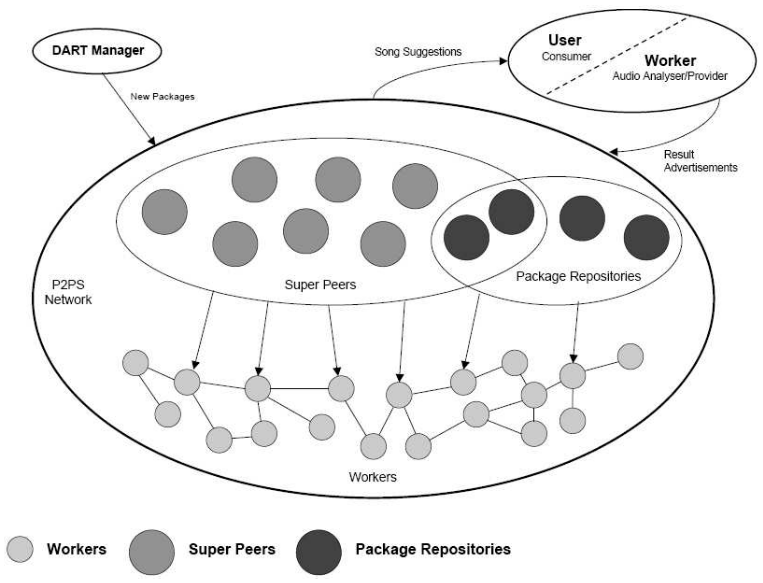


Fig. 6.1. High-level overview of the DART system, showing the various peers and their connectivity.

available packages, result sets and so on) issue queries in order to search for relevant adverts. P2PS rendezvous nodes are then responsible for matching queries with adverts within their local cache, in order to search for matches and respond appropriately. Consumers receive adverts when their query matches, and these adverts can be used to retrieve the relevant information they require, i.e. download the new workflow package to perform the analysis. The DART Manager node produces and advertises the workflow package representing the new DART bundle (called DART Package Adverts) containing algorithms that the worker nodes need to run (new Triana units and workflows). With the DART system, the data files that undergo analysis are on the users local systems hard drive, and therefore all data processing is local so network bandwidth is not consumed transferring large data files over the DART network. Although local, the processing is massively parallel, as participants analyse their own audio files in parallel.

Workers are available to execute the algorithms and workflows, and therefore issue a package query to download a package in order to start the analysis of their music collections. The entire workflow is executed by each worker which downloads the package; the workflow is not then broken down and farmed out to separate nodes to complete different tasks.

Super peer interconnections are used to make package queries travel across the network rapidly; super peers play the role of rendezvous nodes, since they can store package adverts and compare these files with queries issued to discover them; thereby acting as a meeting place for both package providers and consumers. Since packages could require a reasonable amount of storage space, it is assumed that only some of the peers in the network will cache these files. These peers are called Package Repositories (PR) and can also be super peers or worker peers. Each node in the system decides if they want to be a super peer and/or package repository, as well as a worker.

Figure 6.2 shows a sample topology with 5 super peers (2 of which are also package repositories), and the sequence of messages exchanged among different nodes in order to perform the package submission protocol. These messages are related to the execution of a workflow by a single worker, labelled as W_0. Note that in this figure, normal peers are not considered as package repositories.

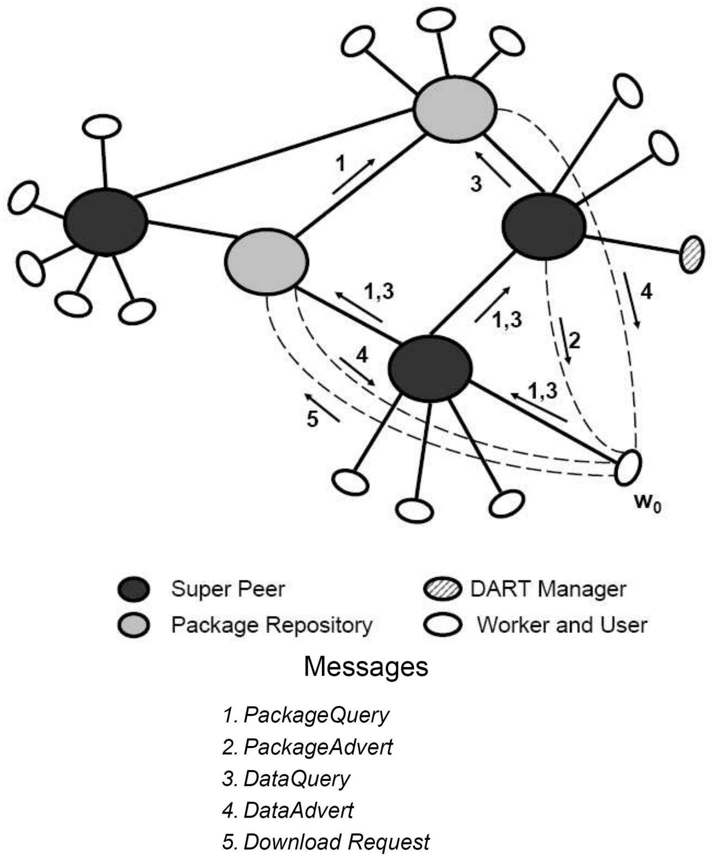


Fig. 6.2. Super peer protocol for the dissemination of workflow packages: sample network topology and sequence of exchanged messages to execute one package cycle.

When a new DART workflow package is available, the DART manager puts this package on one or more package repositories and propagates an associated metadata file, or *package advert*, on the super peer network. This advert is an XML file describing the properties of the algorithms to be executed (i.e. workflow parameters containing the units/tools, platform requirements if any, information about required input audio data files, etc.).

When available to offer some of its CPU time, a worker searches the network to verify if a new version of the package is available. More specifically, the worker sends a *package query* that travels the network through the super peer interconnections (message 1 in Figure 6.2). A package query is expressed by an XML document that contains hardware and software features of the worker node, if this is necessary i.e. available RAM, disk space or JDK version. The query succeeds whenever it matches an advert of a package that can be actually executed by the requesting worker. This *package advert* is then sent directly to the worker.

Thereafter, the worker must search for a package repository that stores the updated workflow package, and sends a *data query* to the network. As more than one package repository can match the query, a matching repository does not send the package directly to the worker, in order to avoid multiple transmissions of the same file. Conversely, the repository returns only a *data advert* to the worker.

A worker can choose a repository according to policies that can rely on the distance of repositories, their available bandwidth etc. Then the worker initiates the *download* operation from the selected repository.

The DART protocol allows for the progressive dissemination of workflow packages on different repositories. Initially these packages are stored on one or few repositories. However, when a worker downloads a package, if the local super peer plays also the role of a repository, the package is first downloaded and cached by this super peer, then forwarded to the worker. In the future another package query can be matched directly by this package repository. Replication of the workflow package on multiple package repositories allows for a significant saving of time in the querying phase and enables the simultaneous retrieval of packages from different repositories.

Once the worker has received the updated package, the workflow is executed by the workers, and they begin to analyse the audio files on the workers system during the systems idle time. Once a package cycle is complete and the worker has results to present, the worker then creates an XML advert containing the results and metadata generated by the algorithm specified in the package (a results advertisement). As the actual results generated would be extremely small in size in this DART system, the functionality of the super peer has been extended in order to also cache and make them available.

Each worker on the network can also be thought of a results provider on the DART system, as well as act as a user, as it can query for results (in this case a suitable music/song suggestion as generated on the super peer). There is no central results collector, but rather DART utilises a fully decentralised model and allows the results to propagate through the network hop-by-hop, to be stored on the super peers. The super peers can process the metadata and issue an XML results advertisement on receipt of a results query from the user. Once the query is received, the results may be sent to the user.

6.2 Distributed Simulations

Here it's analyzed the case of a workflow package of around 10.4MB in size, that correspond to the current size of the Triana audio toolkit, that is to be distributed to the worker nodes on the network. It is here assumed that the workflow can be executed by any worker and that only super peers can be package repositories. The simulation scenario is described in Table 6.1.

Table 6.1. Simulation scenario.

Scenario feature	*Value*
Number of workers, or simple peers, N_{peer}	1,000 to 20,000
Number of super peers, N_{speer}	100 to 2,000
Average number of workers connected to a super peer	10
Maximum number of neighbors for a super peer	4
Average connection time of workers	4 hours
Average disconnection time of workers	1 hour
Number of package repositories	1 to 50% of N_{speer}
Size of input data files	10.4 Mbytes
Latency between two adjacent super peers	100 ms
Latency between a super peer and a local worker	10 ms
Bandwidth between two adjacent super peers	2 Mbps
Bandwidth between a super peer and a local worker	1 Mbps
Mean workflow execution time	10 hours

This scenario simulates the performance and behaviour of a distributed P2P network with 1,000 to 20,000 workers, with a maximum value of 2,000 super peers as the number of super peers is assumed to be 10% of the number of workers. Workers can disconnect and reconnect to the network at any time. This implies that the download or execution of a workflow fails upon the disconnection of the corresponding worker. It is assumed that connections between two adjacent super peers have a larger bandwidth and a longer latency than local connections (i.e. between a super peer and a local simple node).

In the simulation scenario, each worker has to download and execute a workflow package; to this aim it issues a package query and follows the protocol described in previous section. If the download operation fails due to a worker disconnection, a new package query is forwarded and the procedure is repeated.

The experiment is aimed at evaluating the effectiveness of the dynamic caching mechanism. Therefore, the number of available package repositories was varied from 1 to half the number of super peers: one of these repositories provides the workflow package from the beginning, while the others act as *cachers*, as they can download, store and provide data on the fly.

Simulations have been performed to analyze the overall *dissemination time*, T_{diss}, defined as the time needed to propagate the workflow package to at least 95% of the workers. This time is crucial to determine the rate at which workflow packages can be retrieved from the package repositories in order to guarantee that the workers are able to keep the pace with the availability of new versions of the package. The average time needed to perform a single download operation, T_{dl}, is also calculated.

The average utilization index of package repositories, P_{act}, is defined as the fraction of time that a package repository is actually utilized, i.e., the fraction of time in which at least one download connection, from a worker (or another repository), is active with this repository. The value of P_{act} is averaged on all the repositories and can be seen as an efficiency index.

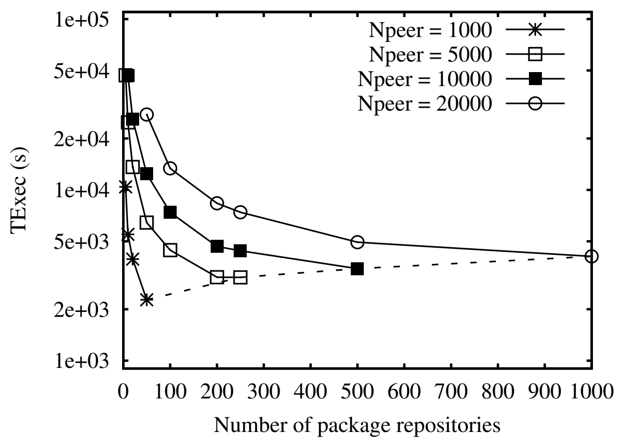


Fig. 6.3. Time at which 95% of workers have downloaded a new version of the workflow package from a package repository.

Figure 6.3 shows that the dissemination time decreases as the number of package repositories increases, as worker nodes can exploit higher parallelism and download workflow packages from multiple repositories (possibly closer) and also because the repositories themselves are under less stress and therefore data download time decreases. Conversely, if the number of repositories is constant, the dissemination time increases with the number of workers as more download operations must be performed and therefore a single repository has to serve more workers on average.

In these simulations, the protocol is shown as scalable when one observes the results obtained with a fixed percentage of repositories. As an example, observe results obtained when the number of repositories is set to 5% of peers (i.e., 50% of super peers, see dashed line in Figure 6.3). It is interesting to note that as the number of peers increases, the dissemination time increases very slightly, much less than the number of peers. For example, with 1,000 peers and 50 repositories, dissemination time is approximately 2,200 seconds; however with 20,000 peers and 1,000 repositories, dissemination time is approximately 4,100 seconds.

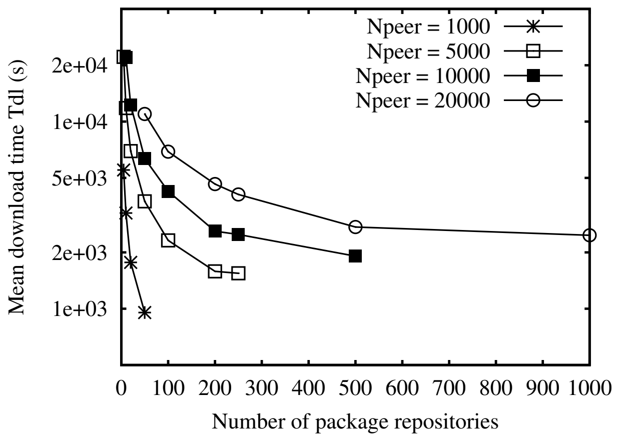


Fig. 6.4. Average download time of single worker from package repository when worker disconnection does not interrupt the download.

Figure 6.4 shows the average download time of a single worker from a package repository when a worker disconnection does not interrupt the download operation. The results here are analogous to the behaviour previously observed when considering Figure 6.3; the qualitative behaviour is the same, but the values are lower. The download time decreases as the number of repositories increases, and the download time will increase as the number of workers increases.

Figure 6.5 shows the percentage of time in which a repository is actually exploited (i.e. there is at least one download in progress). We can observe that as more repositories become available, the percentage of time decreases; therefore, one should be hesitant in setting up a high number of repositories, because while this can slightly decrease dissemination time, they can also be under-exploited. Another interesting result is that the utilisation of

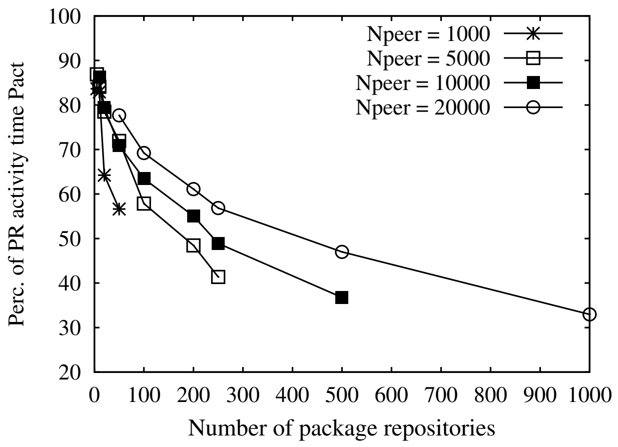


Fig. 6.5. Percentage of time in which a package repository is actually exploited (at least one download in progress).

a given number of repositories increases as the network becomes bigger and more workers need to download the workflow package. This is another verification of the scalability behaviour discussed earlier. It should be noticed that this percentage never reaches 100% because a data cacher (a package repository that has no data at the beginning) must download data from another repository before it can serve a worker.

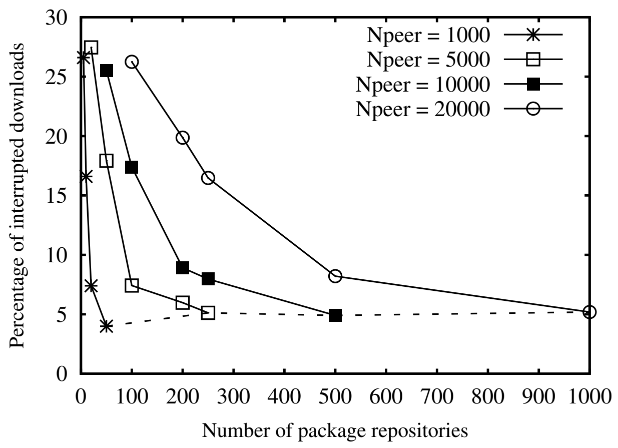


Fig. 6.6. Percentage of interrupted downloads.

Figure 6.6 displays the percentage of download operations that are interrupted due to the disconnections of corresponding downloading workers. In this simulation, only results for which this percentage is lower than 30% are displayed. It was observed that the percentage of interrupted downloads de-

creases as the number of repositories increases. In fact, the download time decreases if more repositories are available (see Figure 6.4), then a worker has more chances to conclude its download operation. Finally, if the percentage of repositories with respect to the number of peers is set to a given value (for example 5%), the percentage of interrupted downloads is almost constant (see dashed line), which is a further confirmation about the scalability of the dissemination protocol.
Summarizing we have performed extensive simulations using our ad-hoc event simulator to explore how the transmitted workflows will propagate throughout the network as the number of peers and super peers increases. The results show that the network scales as the number of members increases as long as the number of super peers that act as data providers increases by the same ratio. This shows that the real application should be capable of operating at an Internet scale with acceptable performance.

References

1. G. Couloris, J. Dollimore, and T. Kinberg, Distributed Systems - Concepts and Design, 4th Edition, Addison-Wesley, Pearson Education, UK, 2001.
2. A. Tanenbaum and M. Van Steen. Distributed Systems: Principles and Paradigms, Prentice Hall, Pearson Education, USA, 2002.
3. R. Buyya (editor). High Performance Cluster Computing, Prentice Hall, USA, 1999.
4. Beowulf Project website. http://www.beowulf.org
5. I. Foster and C. Kesselman (editors), The Grid: Blueprint for a Future Computing Infrastructure, Morgan Kaufmann Publishers, USA, 1999.
6. I. Foster. What is the Grid? A Three Point Checklist. GRIDToday, July 20, 2002.
7. D. Shotton. What is the Grid? Oxford e-Science Centre, August 2001
8. P Plaszczak, R Wellner, Grid computing, 2005, Elsevier/Morgan Kaufmann, San Francisco.
9. IBM Solutions Grid for Business Partners: Helping IBM Business Partners to Grid-enable applications for the next phase of e-business on demand.
10. A Gentle Introduction to Grid Computing and Technologies Retrieved on 2005-05-06.
11. The Grid Caf - What is Grid?. CERN. Retrieved on 2005-02-04.
12. C. Klaasmeyer. Java grid computing JavaWorld.com, 03/28/07
13. NASA Advanced Supercomputing (NAS) website. http://www.nas.nasa.gov/
14. M. Harvey, G. Giupponi, J. Villa-Freixa and G. De Fabritiis. PS3GRID.NET: Building a distributed supercomputer using the PlayStation 3, Distributed & Grid Computing - Science Made Transparent for Everyone. Principles, Applications and Supporting Communities (2007).
15. The Sloan Digital Sky Survey website. http://www.sdss.org/
16. The Globus Alliance website. http://www.globus.org/
17. Foster, I., Kesselman, C. and Tuecke, S. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. In *International Journal of High Performance Computing Applications*, 15 (3). 200-222. 2001.
18. D. Talia and P. Trunfio. Toward a synergy between p2p and grids. *IEEE Internet Computing*, 7(4):96–95, 2003.
19. I. Foster, A. Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. *2nd International Workshop on Peer-to-Peer Systems* (IPTPS '03), Berkeley, California, February 2003.

20. A. Luther, R. Buyya, R. Ranjan, and S. Venugopal- Peer-to-Peer Grid Computing and a .NET-based Alchemi Framework. In *High Performance Computing: Paradigm and Infrastructure*, L Yang and M. Guo (eds), Wiley Press, New Jersey, USA, June 2005.
21. A. Veytsel. There is no P-to-P Market... But There is a Market for P-to-P. Aberdeen Group Presentation at the P2PWG, May 2001.
22. Kindegerg, T. 2002. Personal Communication.
23. Luis Garces-Erice, Ernst W. Biersack, Keith W. Ross, Pascal A. Felber, and Guillaume Urvoy-Keller. Hierarchical p2p systems. In Proceedings of ACM/IFIP International Conference on Parallel and Distributed Computing (Euro-Par), Klagenfurt, Austria, 2003.
24. A. VEYTSEL. There is no P-to-P Market... But There is a Market for P-to-P. Aberdeen Group Presentation at the P2PWG, May 2001.
25. Peer-To-Peer Working Group website. http://www.p2pwg.org/.
26. C. SHIRKY. What is P2P... and what Isnt. An article published on OReilly Network, 2001
27. D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-peer computing. Technical Report HPL-2002-57, HP Lab, 2002.
28. Jim McKeeth A Guide to Peer-2-Peer BorCon 2003
29. R.L. GRAHAM. Traditional and Non-Traditional Applications; Peer-to-Peer Networks. Lecture. 2001. www.ida.liu.se/ TDTS43/tdts43-10-peer-topeer.pdf.
30. S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to peer file sharing technologies, 2002.
31. B. F. Cooper and H. Garcia-Molina. Ad hoc, self-supervising peer-to-peer search networks. Technical report, 2003.
32. U. Lechner. Peer-to-peer beyond file sharing. In IICS, pages 229249, 2002.
33. S. Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In Proceedings of Multimedia Computing and Networking 2002 (MMCN 02), San Jose, CA, USA, January 2002.
34. In suk Kim, Yong hyeog Kang, and Young Ik Eom. An efficient contents discovery mechanism in pure p2p environments. In GCC (1), pages 420427, 2003.
35. D. Tsoumakos and N. Roussopoulos. A comparison of peer-to-peer search methods. In WebDB, pages 6166, 2003.
36. B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. In ICDCS 02: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS02), page 5. IEEE Computer Society, 2002.
37. N. S. Good and A. Krekelberg. Usability and privacy: a study of kazaa p2p file-sharing. In CHI 03: Proceedings of the conference on Human factors in computing systems, pages 137144. ACM Press, 2003.
38. Gnutella website. http://www.gnutella.com/
39. M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. IEEE Internet Computing Journal, 6(1), 2002.
40. Freenet website. http://freenetproject.org/
41. I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. Lecture Notes in Computer Science, 2009:46+, 2001.

42. I. Clarke, O. Sandberg, B.Wiley, T.W. Hong Freenet: A Distributed Anonymous Information Storage and Retrieval System. Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability, LNCS 2009, ed. by H. Federrath. Springer: New York.2001
43. American National Standards Institute, American National Standard X9.30.2-1997: Public Key Cryptography for the Financial Services Industry - Part 2: The Secure Hash Algorithm (SHA-1).
44. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. ACM SIGCOMM 2001, pages 149160, 2001.
45. A. Rowstron and P. Druschel, Pastry: Scalable, distributed object location and routing for largescale peer-to-peer systems. Accepted for Middleware, November 2001.
46. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable. ACM SIGCOMM, 2001.
47. Jabber website. http://www.jabber.org/
48. Groove website. http://www.groove.net/
49. KaZaA website. http://www.kazaa.com/
50. Waste website. http://sourceforge.net/projects/waste
51. Napster website. http://www.napster.com/
52. Morpheus website. http://www.morpheus.com/
53. B. Cohen. Incentives build robustness in BitTorrent. In *Proceedings of the First Workshop on the Economics of Peer-to-Peer Systems*, Berkeley, CA, June 2003.
54. D. Qiu and R. Srikant Modeling and Performance Analysis of BitTorrent-Like Peer-to-Peer Networks In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, ACM, pages 367–378,2004
55. B. Yang and H. Garcia-Molina. Designing a super-peer network. In *19th International Conference on Data Engineering ICDE*, 2003.
56. D. P. Anderson. Public computing: Reconnecting people to science. In *Proceedings of Conference on Shared Knowledge and the Web*, pages 17–19, Madrid,Spain, November 2003.
57. D. P. Anderson, E. Korpela, R. Walton. High-Performance Task Distribution for Volunteer Computingin In *e-Science and Grid Computing, 2005* page(s): 8 pp.- Dec. 2005
58. M. Shirts, V.S. Pande. Screen Saver of World unite! Volume 290,Number 5498, Issue of 8 Dec 2000, pp.1903-1904
59. T. E. Gort. Declaration of Principles Concerning Activities Following the Detection of Extraterrestrial Intelligence. Acta Astronautica 21(2) pp. 153-154, 1990.
60. Slashdot website. http://www.slashdot.org
61. GIMPS website. http://www.mersenne.org/prime.htm
62. Distributed.net website. http://distributed.net
63. Folding@home website. http://folding.standorf.edu
64. Einstein@home website. http://einstein.phys.uwm.edu/
65. Climate@home website. http://climateprediction.net/
66. GridOneD website. http://www.gridoned.org/
67. SETI@home website. http://setiathome.berkeley.edu

68. D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11), 2002.
69. G. Cocconi, and P. Morrison, Searching for Interstellar Communications. Nature, Vol. 184, n.4690, p. 844. Sept. 1959.
70. D. Werthimer, S. Bowyer, D. Ng, C. Donnelly, J. Cobb, M. Lampton and S. Airieau (1997). The Berkeley SETI Program: SERENDIP IV Instrumentation; in the book *Astronomical and Biochemical Origins and the Search for Life in the Universe*. Cosmovici, Bowyer and Werthimer, editors.
71. Berkeley Open Infrastructure for Network Computing, (BOINC) website. http://boinc.berkeley.edu/
72. D. P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.
73. JXTA project website. http://www.jxta.org/.
74. Gilles Fedak, Cecile Germain, Vincent Neri, Franck Cappello. XtremWeb: A Generic Global Computing Platform. IEEE/ACM - CCGRID'2001 Special Session Global Computing on Personal Devices, May 2001, IEEE press
75. A. Chien, B. Calder, S. Elbert, K. Bhatia. Entropia: architecture and performance of an enterprise desktop grid system. Journal of Parallel and Distributed Computing 63, 5. P. 597-610, May 2003
76. V. S. Sunderam, PVM: A Framework for Parallel Distributed Programming. Concurrency: Practice and Experience, 2(4) pp. 315-339, Dec. 1990.
77. H. Burton Bloom Space/time trade-offs in hash coding with allowable errors *Communications of the ACM*, 13 (7), pages 422-426,1970
78. A.M. Law, W.D. Kelton. Simulation Modeling and Analysis, Mac Graw-Hill, 1982.
79. K.S. Trivedi. Probability and Statistics with Reliability, Queueing and Computer Applications. Prentice Hall, Englewood Cliffs, NJ, USA, 1982.
80. J.Banks, J.S. Carson, B. L. Nelson. Discrete-Event System Simulation. Prentice Hall, 1996.
81. R. Fujimoto. Parallel and Distributed Simulation. Wiley Interscience, 2000.
82. R. Motwani and P. Raghavan. Randomized Algorithms. Cambridge University Press, 1995.
83. P. Cozza, C. Mastroianni, D. Talia, and I. Taylor. A super-peer model for multiple job submission on a grid. In *Euro-Par 2006 Workshops*, volume 4375 of *LNCS*, pages 116–125, Dresden, Germany, 2007. Springer-Verlag.
84. P. Cozza, I. Kelley, C. Mastroianni, D. Talia, and I. Taylor. Cache-Enabled Super-Peer Overlays for Multiple Job Submission on Grids. *ISC 2007 CoreGrid Workshop*, Dresden, June 07. To be published.
85. C. Mastroianni, D. Talia, and O. Verta. A super-peer model for resource discovery services in large-scale grids. In *Future Generation Computer Systems*, 21(8):1235–1248, 2005.
86. P. Cozza, I. Kelley, C. Mastroianni, D. Talia, and I. Taylor. A Scalable super-peer approch for public scientific computation. In *Future Generation Computer Systems*, to be published.
87. D. Barbalace, P. Cozza, D. Talia and C. Mastroianni P2P-Based Job Assignment for Public Resource Computing *Integration Workshop Coregrid* 2008, Malta. Submitted

88. E. Al-Shakarchi, P. Cozza, A. Harrison, C. Mastroianni, M. Shields, D. Talia, and I. Taylor Distributing Workflows over a Ubiquitous P2P Network In *Scientific Programming della Ios Press*, to be published.
89. TRIANA project website. http://www.trianacode.org/
90. MRS DART website. http://www.mrsdart.com/
91. Peer-to-Peer Simplified website. http://www.trianacode.org/p2ps/
92. A. Harrison, I. Wang, I. Taylor, and M. Shields. WS-RF Workflow in Triana. In *International Journal of High Performance Computing Applications (IJHPCA) Special Issue on Workflow Systems in Grid Environments*, to be published 2007.
93. M. Shields and I. Taylor. Alchemist: user driven searching in ubiquitous networks. In *Proceedings of the 1st international workshop on Advanced data processing in ubiquitous computing (ADPUC 2006)*, 2006.
94. I. Taylor, M. Shields, I. Wang, and A. Harrison. Visual Grid Workflow in Triana. In *In Journal of Grid Computing*, 3(3-4):153-169, September 2005.