2005

An agent-based peer-to-peer grid computing architecture

Jia Tang
University of Wollongong

Recommended Citation

Tang, Jia, An agent-based peer-to-peer grid computing architecture, M.Info. Thesis, School of Information Technology and Computer Science, University of Wollongong, 2005. http://ro.uow.edu.au/theses/410

An Agent-based Peer-to-Peer Grid Computing Architecture

A thesis submitted in fulfillment of the
requirements for the award of the degree

Master by Research

from

UNIVERSITY OF WOLLONGONG

by

Jia Tang

School of Information Technology and Computer Science
October 2005

Dedicated to
Yingsheng Tang, Yao Jin, and Lili Ge

Declaration

This is to certify that the work reported in this thesis was done by the author, unless specified otherwise, and that no part of it has been submitted in a thesis to any other university or similar institution.

Jia Tang
8th December 2005

Abstract

Grid computing as an emerging technology has made great achievements in scientific computation. Leveraged by other technologies, such as cluster computing and web services, Grid computing for the first time seamlessly integrated large numbers of interconnected computers and provided Internet-scale computing resource sharing, selection, and aggregation.

The sheer numbers of desktop systems today make the potential advantages of interoperability between desktops and servers into a single Grid system quite compelling. However, these commodity systems exhibit significantly different properties than conventional server-based Grid systems. They are usually highly autonomous and heterogeneous systems, and their availability varies from time to time. We call such an environment an open environment.

This thesis aims at bridging the gap between conventional Grid computing and its potential application in open environments by proposing an agent-based peer-to-peer Grid computing architecture, whilst also providing reasonable compatibility and interoperability with conventional Grid systems and clients.

We introduce developments in Grid computing and highlight the targeted research questions concerning Grid computing in open environments. Using these questions as a basis, we review the architecture of the conventional computing Grid and related standards. We indicate that the conventional Grid has five problems, which are barriers to the deployment and application of the Grid in an open environment.

Aiming at solving these problems, we propose a hybrid solution, which is a combined solution that employs both client/server computing architecture and peer-to-peer computing architecture. This solution abandons conventional super-local Grid architecture, and is more efficient, flexible and robust in open environments. We also introduce a multi-purpose task model to handle state persistence and provide help for task decomposition. Furthermore, we employ multi-agent technology to construct the underlying components of our Grid architecture, which brings flexibility and robustness.

Based on the hybrid solution, we develop the architecture to a pure peer-to-peer architecture. In the new solution, we make improvements to the task model so that it provides additional support to task decomposition and inter-task communication in a transparent manner. We develop two frameworks for message passing and routing, and for resource management respectively. These frameworks, together with various intelligent and evolving mechanisms, promote the new adaptability and performance to a higher level.

We widely adopt Web Services and other Grid standards in both solutions to maintain compatibility and interoperability with existing Grid systems and clients. Finally, we discuss the remaining problems with Grid computing in open environments, and outline potential research directions.

In summary, we show that Grid computing architecture, integrating peer-to-peer computing and multi-agent technologies, presents good scalability, efficiency, flexibility, and robustness for Grid computing in open environments in comparison with conventional Grid computing architecture.

Acknowledgements

Studying abroad has been a long and at times arduous journey, and one that could not have been completed without the support of many people.

I am indebted to my supervisor, Professor Minjie Zhang. Her constant commitment and guidance was instrumental in the completion of this thesis, and in making it a fulfilling experience. I am also grateful to her for regularly providing me with insightful comments, and for her kind help, encouragement and understanding in my everyday life. I would also like to thank the School of Computer Science and Information Technology and the University of Wollongong for their financial support towards attending research conferences and their ongoing efforts to create a better workplace.

My thanks are extended to Juliet Richardson, who helped to find and correct English errors in this thesis, and Diana Nguyen, for her help in correcting the English errors in my research papers.

On the personal front, few words can describe the sacrifices that my parents and wife have made on my behalf. This thesis would not have been possible without their financial support, encouragement, understanding and love. I would like to express my deepest gratitude to my mother, who has always made her presence and support felt, especially when most needed. Thanks too, to my wife, Lili, for her constant understanding. During our one and a half year separation, she has always been supportive and understanding. She has constantly tolerated my absence, when instead she should have received my undivided attention. I hope she will forgive me for all that lost time. I have dedicated this thesis to my parents, Yingsheng Tang, Yao Jin, and my wife, Lili Ge, for their patience, understanding, and unconditional love and generosity throughout the years.

Thanks also go to Xixin Zou, who introduced me to the area of computer science at the age of nine. With his enlightenment, I developed a deep interest in computer science in my boyhood. It was at that point that I started my journey of study and research in computer science. Since then, Mr Xixin Zou's original and insightful views on problem solving still have an indelible influence on me. His sound knowledge and ongoing guidance in my early years nourished my mind. Without his support and encouragement, I would never have been able to achieve my research in computer science.

Last but not least, thanks to all the anonymous viewers of my research papers, and all my other dear friends and relatives who have supported me.

Publications

Some of the material in this thesis has previously appeared in, or has been adapted from, the following publications:

- J. Tang and M. Zhang. A Peer-to-Peer Grid Computing Architecture: Convergence of Grid and Peer-to-Peer Computing. In *Proceedings of the 4th Australasian Symposium on Grid Computing and e-Research (AusGrid 2006)*, Hobart, Australia: January 2006 (will appear).
- J. Tang and M. Zhang. An Agent-based Peer-to-peer Grid Computing Architecture. In *Proceedings of the 1st International Conference on Semantics, Knowledge and Grid (SKG 2005)*, Beijing, China: December 2005 (will appear).
- J. Tang and M. Zhang. An Agent-based Grid Computing Infrastructure. In *the 3rd International Symposium on Parallel and Distributed Processing and Applications (ISPA 2005)*, Lecture Notes in Computer Science (LNCS), vol. 3758, Nanjing, China: Springer-Verlag, November 2005, pp. 630-644.
- Q. Bai, J. Tang, and M. Zhang. A Coloured Petri Net Based Strategy for Agent-based Grid Computing. In *Proceedings of the 8th Pacific Rim International Workshop on Multi-Agents (PRIMA 2005)*, Kuala Lumpur, Malaysia: September 2005, pp. 175-186.
- J. Tang, M. Zhang, and H. Zhang. Scheduling and Resource Management in Metacomputing System. In *Proceedings of the 3rd International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications (CSITeA'04)*, Cairo, Egypt: December 2004, pp. 417-422.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the past decade, many studies have been undertaken to increase the performance of parallel systems and host-centric enterprise computing centres. However, these centralised computing technologies have not been able to fulfil the demand for computational power and distributed collaborations by both the scientific area and the industrial area.

By exploiting existing centralised and distributed computing technologies to harness distributed heterogeneous computing resources, Grid computing for the first time fulfils the need for super computational capability and resource sharing in computational science, distributed collaborations and data storage and analysis.

The notion of computing Grid (*Grid* for short) was inspired by the power Grid [8, 22]. A computing 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, capability, performance, cost, and users' quality-of-service requirements" [5]. It is "distinguished from conventional distributed computing by its focus on large-scale resource sharing, innovative applications, and in some cases, high performance orientation" [23].

A three-point checklist [19] can be used to determine whether a computing system is a Grid, according to which a Grid is a system that:

- “Coordinates resources that are not subject to centralised control”- A Grid integrates and coordinates resources from different control domains. There is no global centralised structure and the system is totally distributed.
- “Using standard, open, general-purpose protocols and interfaces” - A Grid must use multi-purpose protocols and interfaces for communications and operations. These protocols and interfaces must be standard and also open, so that resource-sharing arrangements can be established dynamically with any interested party. Standards are also important in providing a general-purpose interface between the Grid and the clients.
- “To deliver nontrivial qualities of service” - A Grid coordinates its constituent resources to deliver various qualities of service. The utility of the combined system is significantly greater than that of the simple addition of all its parts.

This chapter aims at providing an overview of Grid computing in open environments. Section 1.1 describes developments in Grid computing, with some discussion on computing architectures and metacomputing. Section 1.2 explains the significance of the convergence of Grid and peer-to-peer (P2P) computing, and the application domain of the new architecture. Section 1.3 presents the research questions that remain open for Grid computing in open environments. Section 1.4 outlines this thesis, and describes the outcomes of this research and how these outcomes are embodied in this thesis.

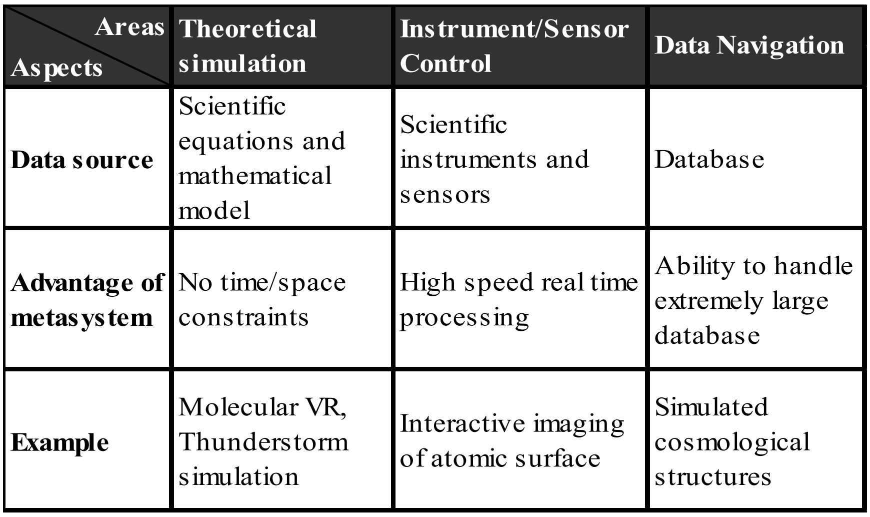
1.1 Developments in Grid Computing

The evolution of computing architecture follows the major technological advances in PCs and networking. In the mainframe era, almost everything was done by mainframe computers. Processing in the mainframe quickly became a bottleneck in any information system. Continuous investment in mainframe upgrades cannot maintain efficiency under increased processing demands and are thus not cost effective. With the miniaturisation of computers and the emergence of computer networks, the *client/server* (C/S) architecture [31] was first proposed as an alternative to conventional mainframe systems. This shifts the processing burden to the client computer, and therefore improves overall efficiency [2]. Later, we saw the rise of LAN-based *cluster computing* [40, 44] in the 1980s, and WAN-based *metacomputing* [32, 48] in the 1990s, both of which derive from the client/server architecture, and aim at further sharing the workload through computer networks. Inspired by the power grid, Grid computing further exploits cluster computing and metacomputing to Internet-scale computing resource sharing, selection, and aggregation.

1.1.1 Application-oriented Metacomputing

Metacomputing is the predecessor of Grid computing. The rise of metacomputing derived from the popularity of parallel processing, which was facilitated by two major developments: *Massively Parallel Processors* (MPPs) and the widespread use of distributed computing. The similarity between distributed computing and MPP is the notion of *message passing*, around which two systems were developed - the *Parallel Virtual Machine* (PVM) [27] and the Message Passing Interface (MPI) [28].

PVM aimed at exploiting a collection of networked computers and the heterogeneity of architecture, data format, computational speed, machine load, and network load. From the beginning, it was designed to make programming for a heterogeneous collection of machines straightforward, whereas the MPI standard was not intended to be a complete and self-contained software infrastructure that could be used for distributed computing. The main purpose of MPI was to establish a message-passing standard for portability. And indeed, it provided MPP vendors with a clearly defined set of routines that they could implement efficiently or provide hardware support for.



Areas Aspects	Theoretical simulation	Instrument/Sensor Control	Data Navigation
Data source	Scientific equations and mathematical model	Scientific instruments and sensors	Database
Advantage of metasystem	No time/space constraints	High speed real time processing	Ability to handle extremely large database
Example	Molecular VR, Thunderstorm simulation	Interactive imaging of atomic surface	Simulated cosmological structures

Figure 1.1: Applications of Metacomputing on Computational Science

The LAN metacomputer at NCSA [48] based on PVM was the earliest example of nation-wide metacomputing system. The purpose of building metacomputing systems was to solve computational science problems. Figure 1.1 shows the applications of metacomputing on computational science. These applications cut across three fundamental areas of computation science [48]:

- Theoretical simulation, which can be described as using high-performance computing to solve scientific problems numerically by using scientific equations and mathematical models.
- Instrument/Sensor control, in which a metacomputing system is used to manipulate real time and interactive visualisation from raw data supplied by scientific instruments and sensors.
- Data Navigation, the method through which most computational science is carried out. This involves exploring large databases, and translating numerical data into human sensory input.

Condor [15] and Legion [33, 34] are the early successes of general-purpose metacomputing systems. A general-purpose metacomputing system must be responsible for [33] 1) transparently scheduling application components on processors; 2) managing data migration, caching, transfer and coercion; 3) detecting and managing faults; and 4) providing adequate protection to users' data and physical resources. Other general-purpose metacomputing systems include Charlotte [4], Javelin [10, 41], WebFlow [1], Gateway [25], CX [7], and the early version of Globus [21]. Bake et al. have given a comprehensive description of most existing metacomputing systems [3].

1.1.2 Service-oriented Grid Computing

The idea of computing Grids come from power grids. It evolved from cluster computing, but with a remarkable distinction - the way of resource management [5]. In the case of Grid computing, there is no global centralised structure, and the system is totally distributed. In a cluster environment, however, the resource allocation is performed in a centralised fashion, and a master/slave relationship always exists, where the master node acts as a load balance proxy or task scheduler.

The first definition of a computing Grid was given by Ian Foster and Carl Kesselman in their book "*The Grid: Blueprint for a New Computing Infrastructure*" [22]. In a subsequent article, they stated that "Grid computing is concerned with coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organisations (VOs)" [24]. The key concept is to exploit synergies that result from cooperation - the ability to share and aggregate resources among these VOs, and then to use the resulting resource pool for a certain purpose. This notion was further developed in Open Grid Services Architecture (OGSA) [23], where a Grid was viewed as an extensible set of Grid services that may be aggregated in various ways to meet the needs of VOs.

The state of the art in Grid computing is represented by Globus Toolkit [20, 30] and its related standards. The key standards and outcomes of Globus are listed below:

1. Open Grid Service Architecture (OGSA) [23], defines the core services of the Grid. Typical services include a security infrastructure (Grid Security Infrastructure (GSI) [55] or WS Authentication & Authorization [20]), an execution management framework (Grid Resource Allocation and Management (GRAM) [14] or WS GRAM [20]), an information service framework (Monitoring and Directory Service (MDS) [16] or WS MDS [20]), and the core runtime libraries (Java/C Core, or Java/C WS Core).

2. The complete adoption of Service Oriented Architecture (SOA) [53] and Web Services [50], brings the Grid better interoperability with existing industrial standards and Web Services infrastructure. Employing Web Services standards

to build its core services and infrastructure improves the versatility of the implementation, and reduces conflict between the Grid community and the Web Services community.

3. Web Services Resource Framework (WSRF) [17] / Open Grid Service Infrastructure (OGSI) [29], defines the services infrastructure of the Grid. WSRF exploits the Web Services infrastructure for resource management. WS-Resource, as part of WSRF standards, proposes a standard means of describing resources.

1.2 Convergence of Grid and Peer-to-Peer Computing

Today, the sheer numbers of desktop systems make the potential advantages of interoperability between desktops and servers into a single Grid system quite compelling. Peer-to-Peer (P2P) computing [2], as another emerging computing architecture, is tackling an overlapping set of problems with Grid computing [38]. Although a parallel situation exists, the differences between Grid computing and P2P computing originate from their usages. Computing Grids were first used for scientific computation, while P2P computing gained prominence in the context of multimedia file exchange. Globus [21], the defacto Grid standards, was initially an umbrella project. It was designed to federate underlying workload management systems to work for collaborations. This objective destined its role as a middleware and its super-local architecture. On the other hand, P2P computing aims at the collaboration of massive commodity computing devices. There are no such constraints on its architecture as Globus has, which makes P2P computing more flexible. In fact, it uses the computing power at the edge of a connection rather than within the network. The client/server architecture does not exist in a P2P system. Instead, peer nodes act as both clients and servers - their roles are determined by the characteristics of the tasks and the status of the system. This architecture minimises the workload per node, and maximises the utilisation of the overall computing resources among the network.

In contrast to the application domain of Grid computing in the scientific research area, P2P computing primarily offers file sharing (e.g. Napster and Bittorrent [11]), distributed computation (e.g. SETI@home [36]), and anonymity (e.g. Publius [54]). Although the two types of computing architectures have both conceptual and concrete distinctions, their convergence is significant - “the vision that motivates both Grid and P2P computing - that of a worldwide computer within which access to resources and services can be negotiated as and when needed - will come to pass only if we are successful in developing a technology that combines elements of what we today call both P2P and Grid computing” [18].

Despite the wide acceptance of Grid computing in the scientific research area, its server-based architecture in the local context and middleware nature in the global context limits its application in *open environments*, where the computing nodes are highly autonomous and heterogeneous, and their availability varies from time to time. An example of an open environment is the Internet, where enormous idle resources exist, which are normally not organised in terms of providing computing power for a certain purpose.

This thesis aims at bridging the gap between Grid computing and P2P computing by proposing and implementing an agent-based peer-to-peer Grid computing architecture which can be deployed in open environments, while providing reasonable compatibility and interoperability with conventional Grid systems and clients.

1.3 Research Questions

Research questions concerning Grid computing in open environments include:

- What is the best way to support task decomposition, inter-task communication, and state persistence?

  We consider these three features to be essential to the Grid. With task decomposition support, a computational task, which consists of parallel subtasks, can be decomposed automatically to achieve parallelism, and therefore leads to better performance and efficiency than a serial computing model. This support of inter-task communication and state persistence will save a great deal of time for the application developers, as they will not need to write their own frameworks to support the two features.

- What is the best strategy for resource management and scheduling? Open environments are remarkably different from the application domain of the conventional Grid. In order to find the best way to manage resources and schedule tasks, the properties of open environments must be carefully considered. The autonomy, heterogeneity, intermittent participation and highly variable behaviour of the constituents of open environments are the major concerns regarding resource management and scheduling.

- How to provide compatibility and interoperability? To what level?

  Today, there are hundreds of production Grids all over the world, and thousands of Grid applications running at this moment. Any new Grid system must take compatibility and interoperability with existing Grids, Grid clients and applications into consideration. With the standardisation of Grid computing and the embrace of Web Services standards, it is easier to achieve compatibility and interoperability in any new Grid, as long as it follows these standards.

- How to support organisational hierarchy in a distributed system?

  We consider organisational hierarchy as one of the research questions, because logically centralised information, such as identity, access control information, software repository, and files, will be stored in environments where no global centralised structure exists, and this information must be accessible whenever needed in the context of the highly dynamic nature of the resource providers.

- How to secure the Grid?

  This entails how to authenticate users, authorise their operations, and secure data and communications.

- How to handle the different network connectivities of the participants?

  This question relates to the network bandwidth, latency, and accessibility. All these attributes are crucial for the quality of service (QoS) of the Grid.

In this thesis, we address the first three questions by:

1. Proposing multi-purpose programming models to support task decomposition, inter-task communication, and state persistence.
2. Proposing resource management and scheduling strategies to solve resource discovery, selection, allocation, and release, load balance, task execution, task monitoring, fault-tolerance, and how to store data. We divide these issues into two areas: computing architecture and resource management framework, where the former probes into the dispatch of tasks and service requests, while

the latter works on how to record resources and match them with service requests.

3. Keeping compatibility and interoperability in mind. The solutions presented later (see Chapter 3 and Chapter 4) provide good compatibility and interoperability with existing Grids, Grid clients and applications.

The rest of the questions are discussed in Chapter 5 as the future work of this research.

1.4 Thesis Structure and Outcomes

This thesis begins with a presentation and review of developments in Grid computing and its conventional application domains. The emerging application of Grid computing in open environments leads to the convergence of Grid and P2P computing, which is the future direction of Grid computing. Research questions regarding this area are considered to be sixfold, three of which are tackled in this thesis as part of the full approach to Grid computing in open environments.

The major contributions of this thesis include: 1) a multi-purpose task model proposed in Chapter 3, with further improvements in Chapter 4, which makes task decomposition, inter-task communication, and state persistence straightforward; 2) a hybrid computing architecture, described in Chapter 3 and then its descendant in Chapter 4, which handles the autonomy, heterogeneity, intermittent participation and highly variable nature of computing resources; and 3) a resource management framework, demonstrated in Chapter 4, for resource discovery, matching, and selection. The remaining chapters of this thesis are organised as follows:

Chapter 2 reviews the conventional Grid in regard to our targeted research questions. Discussion focuses on the service-oriented architecture, the state persistence mechanism, and the resource management and scheduling strategy of the conventional Grid. Five problems with the conventional Grid are presented as a result of discussion on the application of computing Grids in open environments.

Chapter 3 introduces a hybrid solution to Grid computing in open environments, whose architecture is a combination of client/server computing architecture and P2P computing architecture. A novel task model is also proposed in this chapter for multiple purposes.

Chapter 4 gives a P2P solution, which derives from the hybrid solution. With an improved task model, a P2P computing architecture, and a resource management framework, the new Grid solves the problems outlined in Chapter 2, as well as presenting an elegant solution to our targeted research questions.

Chapter 5 compares the two solutions and summarises this research, with discussion on the rest of the research questions.

Chapter 2

Review of Related Research and Literature

The conventional computing Grid has developed a service-oriented computing architecture, with a super-local resource management and scheduling strategy. In Globus Toolkit 4 [30] (GT4, the official implementation of the defacto Grid standards), nine high-level Grid services, defined by Open Grid Service Architecture (OGSA) [23], are implemented, using Web Services mechanisms to provide functionalities such as resource management, scheduling, etc. As these services are required to be stateful, and because Web Services are usually stateless, Web Services Resource Framework (WSRF) [17] was introduced so that stateful information can be preserved as WS-Resources between different service invocations. In this chapter, we review the research and literature that relate to our targeted research questions (recall Section 1.3).

We first briefly introduce the Web Services architecture, on which GT4 is based, in Section 2.1. We also discuss the relevant specifications that are adopted in the Grid community.

In Section 2.2 and Section 2.3, we review the Web Services Resource Framework and the Open Grid Services Architecture. We present the architecture of GT4, which is the result of the convergence of OGSA and WSRF.

We look into the resource management and scheduling of OGSA in Section 2.4. We introduce the components used for resource management and scheduling respectively, and then demonstrate how they are integrated to provide services.

At the end of the review, we consider the problems associated with the conventional Grid in terms of open environments. These problems can also be viewed as the objectives of this research with regard to the targeted questions.

2.1 Web Services Architecture

W3C defines the Service Oriented Architecture (SOA), which is a form of distributed systems architecture that is typically characterised by the following properties [53]:

- “Logical view: The service is an abstracted, logical view of actual programs, databases, business processes, etc., defined in terms of what it does, typically carrying out a business-level operation.”
- “Message orientation: The service is formally defined in terms of the messages exchanged between provider agents and requester agents, and not the properties of the agents themselves. The internal structure of an agent, including features such as its implementation language, process structure and even database structure, are deliberately abstracted away in the SOA: using the SOA discipline one does not and should not need to know how an agent implementing a service is constructed. A key benefit of this concerns so-called legacy systems. By avoiding any knowledge of the internal structure of an agent, one can incorporate any software component or application that can be ‘wrapped’ in message handling code that allows it to adhere to the formal service definition.”

- “Description orientation: A service is described by machine-processable metadata. The description supports the public nature of the SOA: only those details that are exposed to the public and important for the use of the service should be included in the description. The semantics of a service should be documented, either directly or indirectly, by its description.”
- “Granularity: Services tend to use a small number of operations with relatively large and complex messages.”
- “Network orientation: Services tend to be oriented toward use over a network, though this is not an absolute requirement.”
- “Platform neutral: Messages are sent in a platform-neutral, standardised format delivered through the interfaces. XML is the most obvious format that meets this constraint.”

A Web Service is [53] “a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialisation in conjunction with other Web-related standards”.

To put it quite simply, Web Services are a distributed computing technology, such as CORBA, RMI, EJB, etc., which helps programmers to create client/server applications. An important characteristic that distinguishes Web Services from other technologies, such as CORBA, is that Web Services are more adequate for loosely coupled systems, where the client might have no prior knowledge of the Web Service until it actually invokes it. This advantage of Web Services, together with other advantages, such as the fact that they are platform-independent, language-neutral, and transport-agnostic over other technologies, also provides evidence of the wide acceptance and application of Web Services, and the embrace of Web Services in Grid computing.

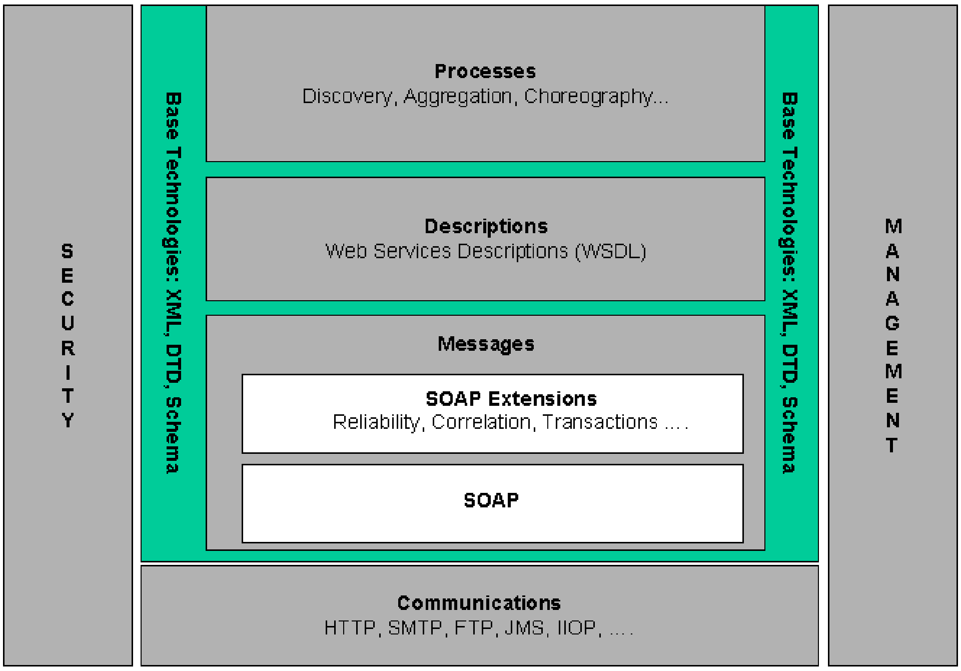

SECURITY
Base Technologies: XML, DTD, Schema
Processes
Discovery, Aggregation, Choreography...
Descriptions
Web Services Descriptions (WSDL)
Messages
SOAP Extensions
Reliability, Correlation, Transactions ....
SOAP
Base Technologies: XML, DTD, Schema
MANAGEMENT
Communications
HTTP, SMTP, FTP, JMS, IIOP, ...

Figure 2.1: Web Services Architecture

Figure 2.1 [53] provides an illustration of the Web Services architecture. The essential parts of this architecture are:

- Service Processes. This part of the architecture includes service registration, discovery, etc.
- Service Description. Web Services are self-describing through using Web Services Description Language (WSDL) [9]. This means that once a Web Service is located, it can describe what operations it supports and how to invoke it.
- Service Invocation. Invoking a Web Service involves passing messages between

the client and the server. SOAP (Simple Object Access Protocol) [52] specifies how the client should format requests to the server, and how the server should format its responses. The Web Services architecture allows the use of service invocation protocols. However, SOAP is by far the most popular choice for Web Services.

- Transport. All the messages must be transmitted between the client and the server. The most popular choice for this part of the architecture is HyperText Transfer Protocol (HTTP) [51]. In theory it could be other protocols.

From the client's perspective, a Web Service is simply a network accessible service. On the server side, the situation is more complex. A common Web services implementation (see Figure 2.2) includes:

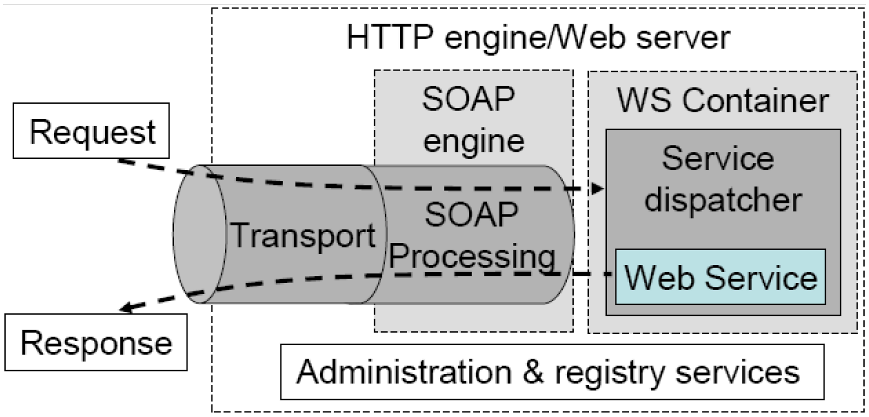


Figure 2.2: A Typical Web Services Implementation

- Web Services. Basically, these are pieces of software that expose sets of operations. They know nothing about how to interpret SOAP requests or how to create SOAP responses.

- SOAP engine. This is the software component that knows how to handle

SOAP requests and responses. Normally, a generic SOAP engine is used to manipulate SOAP messages for all Web Services.

- HTTP engine/Web Server. This is the software component that transports the SOAP messages (typically via HTTP). This part is sometimes called the application server.

Figure 2.3 displays a typical Web Service invocation.

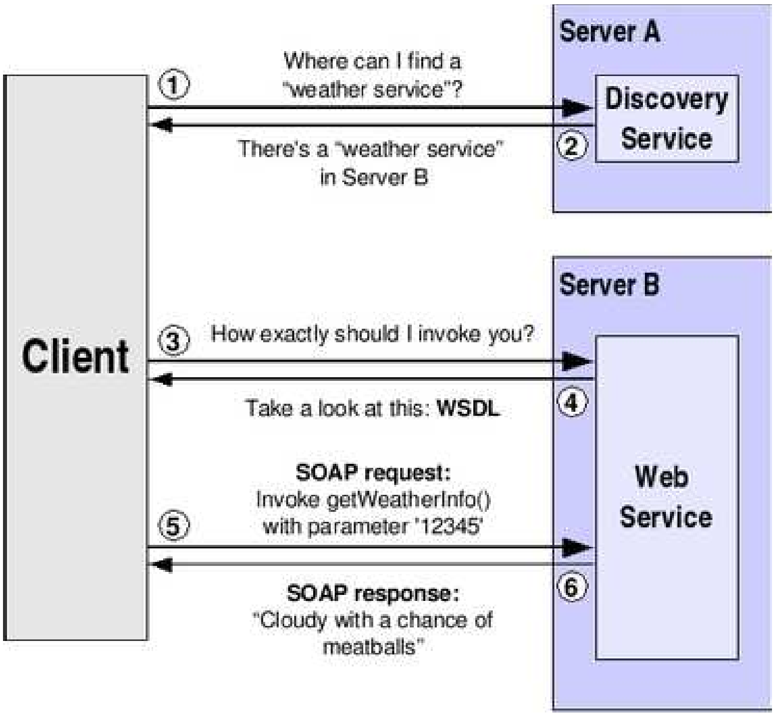

Server A
Where can I find a "weather service"?
①
Discovery Service
②
There's a "weather service" in Server B
Server B
Client
③
How exactly should I invoke you?
④
Take a look at this: WSDL
SOAP request: Invoke getWeatherInfo() with parameter '12345'
⑤
Web Service
⑥
SOAP response: "Cloudy with a chance of meatballs"

Figure 2.3: A Typical Web Service Invocation

1. A client may have no knowledge of what Web Service it is going to invoke. The first step is to discover a Web Service that meets the requirements through a discovery service (which is itself a Web Service).

2. The discovery service replies with information about what servers can provide the required service.

3. The client contacts the service provider and asks the Web Service to describe itself.
4. The Web Service replies in WSDL.
5. The client acquires the description of the service and sends a SOAP request for a certain operation according to the service contract (i.e. the description).
6. The Web Service replies with a SOAP response that includes the required information, or an error message if the SOAP request was incorrect.

Web Services specifications that GT4 adopts [20] in regard to the research questions include:

- XML, which is used extensively within Web services as a standard, flexible, and extensible data format.
- SOAP [52], which provides a standard, extensible, composable framework for packaging and exchanging XML messages between a service provider and a service requester. SOAP is independent of the underlying transport protocol, but is most commonly carried on HTTP.
- WSDL [9], which is an XML document for describing Web Services. More specifically, a set of standardised binding conventions define how to use WSDL in conjunction with SOAP and other messaging substrates.
- Web Services Resource Framework (WSRF) [17], which defines a generic and open framework for modelling and accessing stateful resources using Web Services. This framework comprises mechanisms to describe views on the state to support management of the state through properties associated with the Web Services.

2.2 Web Services Resource Framework

As discussed in the previous section, the Web Services architecture was chosen as the underlying distributed technology on which the Grid bases its architecture. Although the Web Services architecture was certainly the best option, it still did not meet one of the Grid's most important requirements: the underlying distributed infrastructure had to be stateful. Although Web services can be either stateless or stateful in theory, they are usually stateless and there is no standard way of making them stateful, hence the introduction of the Web Services Resource Framework (WSRF) [17].

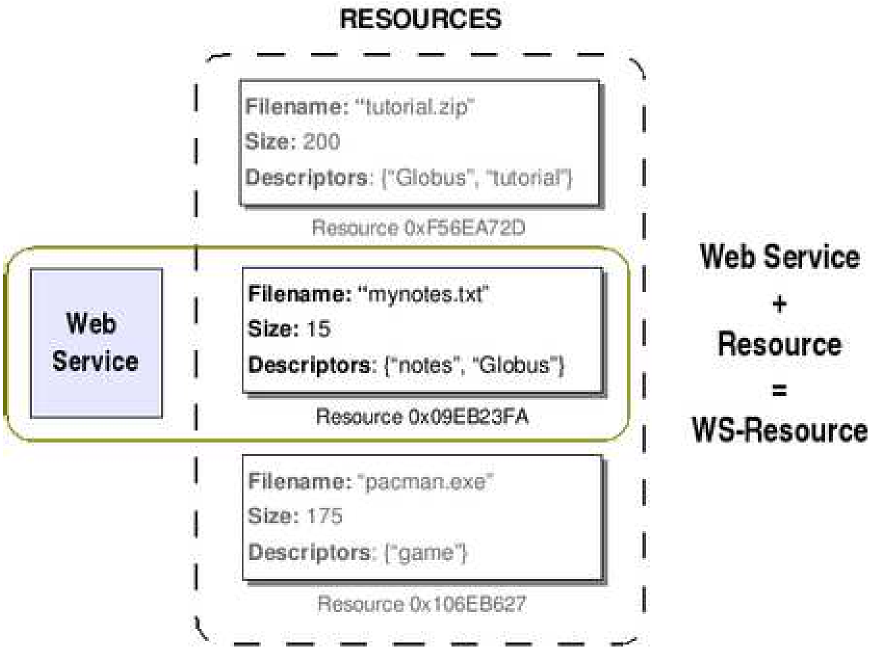


Figure 2.4: A Web Service with File Resources

WSRF is a joint effort by the Grid and Web Services communities. It specifies how to make Web Services stateful, which is required by the Grid. The approach is to keep the Web Service and the state information (called a resource) separately. Each resource has a unique key. Whenever a stateful interaction with a Web Service is needed, an instruction is given to the Web Service to use a particular resource. A pairing of a Web service with a resource is called a WS-Resource. Figure 2.4 exhibits a Web Service using file resources.

2.3 Open Grid Services Architecture and the Grid

The Open Grid Services Architecture (OGSA) [23] is a collection of standards that define a common, standard and open architecture for Grid-based applications. The goal of OGSA is to standardise practically all the services commonly found in a Grid application, by specifying a set of standard interfaces for these services. The typical Grid services include:

- VO Management Service. To manage what nodes and users are part of each Virtual Organisation.
- Resource Discovery and Management Service. So that applications on the Grid can discover resources that suit their needs, and then manage them.
- Job Management Service. So that users can submit tasks to the Grid.
- Other services. These include security, data management, etc.

GT4 is a reference implementation of both OGSA and WSRF. More specifically, it is a set of software components that implement Web Services mechanisms for building computing Grids [20]. As shown in Figure 2.5 [30], these components are divided into five categories:

- Common Runtime. Common Runtime components provide a set of fundamental libraries and tools which are needed to build both WS and non-WS services.

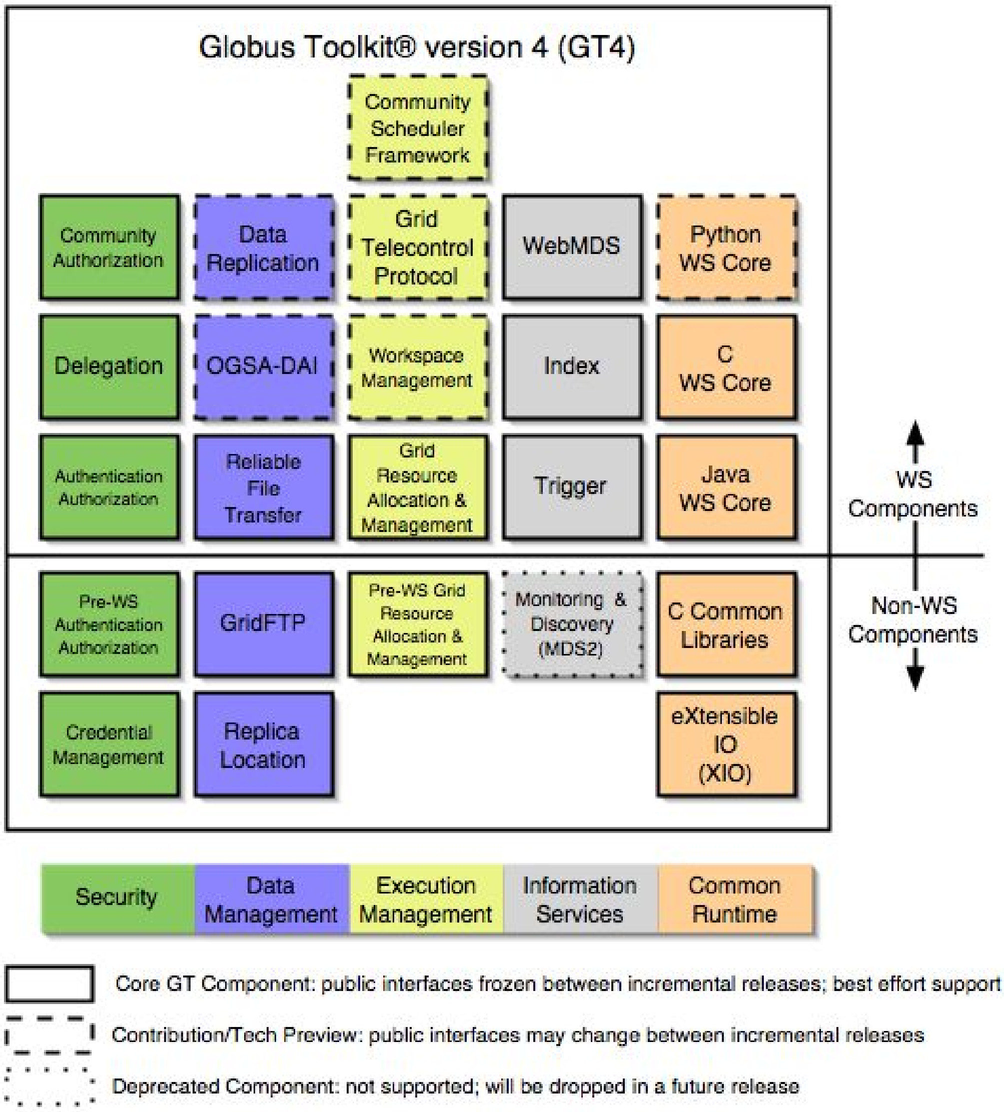

Globus Toolkit® version 4 (GT4)
Community Scheduler Framework
Community Authorization
Data Replication
Grid Telecontrol Protocol
WebMDS
Python WS Core
Delegation
OGSA-DAI
Workspace Management
Index
C WS Core
Authentication Authorization
Reliable File Transfer
Grid Resource Allocation & Management
Trigger
Java WS Core
WS Components
Non-WS Components
Pre-WS Authentication Authorization
GridFTP
Pre-WS Grid Resource Allocation & Management
Monitoring & Discovery (MDS2)
C Common Libraries
Credential Management
Replica Location
eXtensible IO (XIO)
Security
Data Management
Execution Management
Information Services
Common Runtime
Core GT Component: public interfaces frozen between incremental releases; best effort support
Contribution/Tech Preview: public interfaces may change between incremental releases
Deprecated Component: not supported; will be dropped in a future release

Figure 2.5: Schematic View of Globus Toolkit 4 components

- Execution Management. Execution Management components deal with the initiation, monitoring, management, scheduling and coordination of executable programs (i.e. the tasks) in a Grid.

- Information Services. Information Services commonly refer to the monitoring and discovery services, which can streamline the tasks of monitoring and discovering services and resources in a virtual organisation.

- Data Management. These components enable the management of large sets of data in a virtual organisation.

- Security. Security components, based on the Grid Security Infrastructure (GSI) [55], secure the communications in a Grid.

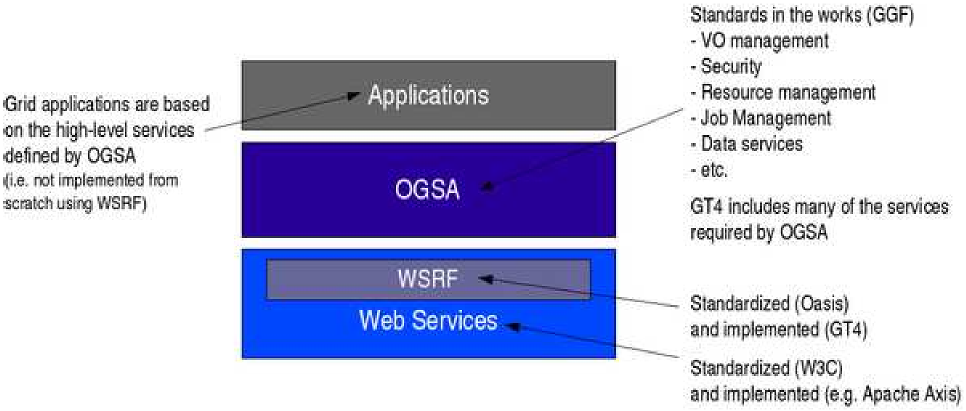

Grid applications are based on the high-level services defined by OGSA (i.e. not implemented from scratch using WSRF)
Applications
OGSA
WSRF
Web Services
Standards in the works (GGF)
- VO management
- Security
- Resource management
- Job Management
- Data services
- etc.
GT4 includes many of the services required by OGSA
Standardized (Oasis) and implemented (GT4)
Standardized (W3C) and implemented (e.g. Apache Axis)

Figure 2.6: Architecture of Globus Toolkit 4

Most of the above services and components in GT4 are implemented on top of WSRF. GT4 also includes a complete implementation of the WSRF specification. The convergence of the Grid and the stateful Web Services is the architecture displayed in Figure 2.6.

2.4 Resource Management and Scheduling

The term *meta-scheduler* is used to describe the scheduling and resource management components in the Grid [46]. A meta-scheduler consists of two core components: a super scheduler, which is in charge of resource discovery, selection and submission for a certain job [46, 47]; and a local scheduler, which serves as a broker that "performs resource quoting or resource discovery and selection based on various strategies, assigns application tasks to those resources, and distributes data or co-locates data and computations" [46]. Although the local schedulers, such as Condor-G [26], Nimrod-G [6], and InfoGram [37], are different from system to system, many of the Grid systems use Monitoring and Discovery System (MDS) [16] and Grid Resource Allocation and Management (GRAM) [14] as their high-level resource management and scheduling services.

In GT4, MDS manages the monitoring and discovery of resources. It obtains information from several information providers and publishes it to other services. Three of the information providers are related to job execution: two for gathering data related to cluster resources, and one for providing information about the local schedulers.

GRAM manages the submission and execution of jobs. It uses a super-local scheduling strategy: the super scheduler schedules a job to a suitable local scheduler, based on the job's requirements and the local schedulers' statuses provided by MDS; the local scheduler then schedules the job to a specific computing node. Figure 2.7 [20] depicts this strategy. The dashed area indicates the service host (i.e. the super scheduler). The compute element consists of a local scheduler and computing nodes.

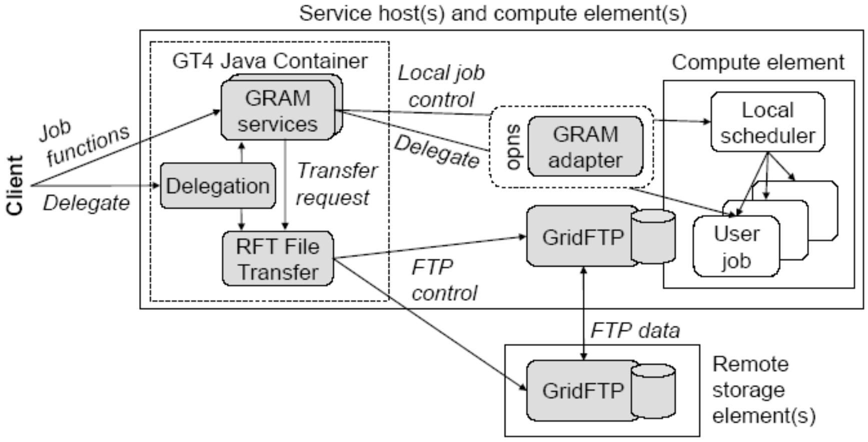


Figure 2.7: The Super-local Scheduling Strategy in Globus Toolkit 4

2.5 An Study of the Grid in terms of Open Environments

As Web Services provide a standard means for communication and object invocation between clients and service providers, the embrace of Web Services increases the interoperability of the Grid. The super-local scheduling strategy is also a success in high-end computational environments, because of its flexibility in the face of various widely accepted local schedulers such as Condor [5]. But in order to implement and deploy a Grid in an open environment, the autonomous, heterogeneous, and highly dynamic nature of such an environment must be carefully considered. These properties further lead to the following problems with the conventional Grid:

1. WSRF was developed as a complement to Web Services in order to make stateless Web Services stateful. However, it can result in significant overheads on network traffic and object invocations due to transmissions of WS-Resources between the client and the service host, and conversions between internal states of a service and their WS-Resource equivalents.

2. The current service-oriented architecture has poor adaptability in terms of performance, availability and scalability, as no facility has been provided by current Grid systems to allow automatic deployment of services according to the clients' requests and the load in the Grid.

3. The dependence on local schedulers increases the complexity of application programming in the Grid environment, as it is difficult to provide various local schedulers with a uniform programming interface that supports task decomposition, state persistence and inter-task communications.

4. The super-local resource management and scheduling strategy intensively relies on the underlying local schedulers. This two-level process leads to more complex handling of resource discovery, selection and allocation compared with a one-level process. The lack of direct management of the computing nodes can cause unsuitable selection of resources and unbalanced loads, and therefore limits the overall performance. In addition, as new computing nodes can only join local schedulers, instead of joining the Grid directly, the scalability of local schedulers greatly affects the overall scalability of the Grid.

5. It is not feasible to introduce local schedulers into our targeted environment, as local schedulers require a relatively static and non-autonomous environment.

2.6 Summary

This chapter reviews the state of the art of Grid computing in regard to our targeted research questions proposed in Chapter 1. We briefly introduced the Web Services architecture and the Web Services Resource Framework, on which the defacto standard Grid system, GT4, is based. We then discussed the Open Grid Services

Architecture, followed by a review of the architecture and software components of its reference implementation - GT4. In particular, we looked into the resource management and scheduling strategy of GT4, which is one of the key subjects of this thesis. Finally, we studied the current Grid architecture and relevant standards in terms of the application of Grid computing in open environments.

In the next chapter, we present our first solution to the problems outlined in Section 2.5.

Chapter 3

A Hybrid Solution to Grid Computing in Open Environments

Five problems have been outlined in Chapter 2 in relation to Grid computing in open environments. Aiming at solving these problem, we propose a hybrid solution using multiple intelligent agents [39] combined with server-based computing architecture and P2P computing architecture. We call this solution smartGRID (service-oriented, microkernel, agent-based, rational, and transparent Grid).

We first present in Section 3.1 a description of the overall architecture of smartGRID. We describe the essential components and adaptive mechanisms of smartGRID that make it flexible and robust.

We then introduce the novel task model of smartGRID in Section 3.2. We demonstrate how the new task model can support state persistence, as well as how the task description can assist the scheduling process.

Section 3.3 focuses on the scheduling process and the evolving mechanisms of smartGRID. Coloured Petri Nets (CPNs) [35] are extensively used in this section to describe the agent interactions and communication protocols. All evolving mechanisms are described in detail, followed by an explanation of how these mechanisms can make smartGRID self-contained.

Finally, we discuss compatibility and interoperability issues in Section 3.4. We explain why smartGRID is compatible with existing Grid clients. Two methods are described in regard to how to preserve states, as well as how to use one of the methods to achieve task decomposition. Lastly, we discuss promising approaches to the interoperability between smartGRID and existing local schedulers.

3.1 Overall Architecture and Core Components

There are three tiers in smartGRID: the clients, the trackers, and the computing nodes, which are defined in the table below.

Definition 3.1 *A client is a generic computing device that seeks services from the Grid using Web Services Standards.*

Definition 3.2 *A tracker is a computer which performs task scheduling operations in its managed LAN.*

Definition 3.3 *A computing node is the place where tasks are executed and computing occurs. A client or a tracker can act as a computing node at the same time.*

Table 3.1: Definitions of Tiers in smartGRID

Figure 3.1 shows the tiers in smartGRID. We assume that the tiers discussed in the following sections of this chapter are in the same LAN.

Table 3.2 defines the trackers and the basic communication rules in smartGRID. A tracker maintains the following information: 1) the computing resources available (called *profile*) on each of the nodes in the tracker's LAN; 2) all tasks submitted to the tracker (including the running tasks, and the tasks in its waiting queue); 3) the overall load of the tracker's LAN; and 4) the contacts of a limited number of other trackers. Multiple trackers can exist in the same LAN for performance and/or fault-tolerance consideration.

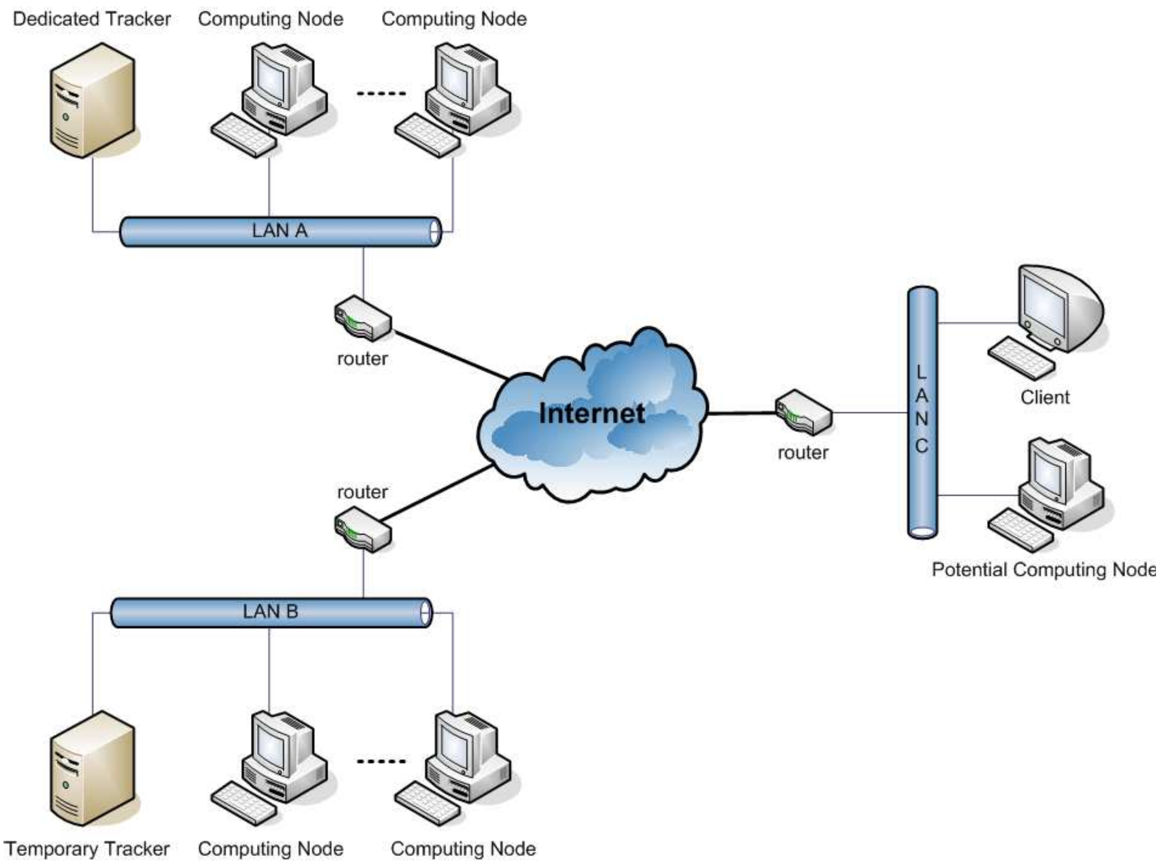


Figure 3.1: Tiers in smartGRID

The detailed self-organising process is explained in Figure 3.2. This process allows new computing nodes to join smartGRID and enables smartGRID to expand dynamically, which is essential to the scalability. It also works as one of the evolving mechanisms that dynamically optimises the configuration of smartGRID by selecting the most suitable temporary tracker to handle the LAN-based operations, so that the computing nodes can contribute their computing power to their fullest extent.

Definition 3.4 *A number of computers which have high availability, good connectivity, and good performance are selected as the top-level trackers when the Grid is constructed.*

Definition 3.5 *Any other computer becomes a tracker by registering itself to an existing tracker. The existing tracker is called the parent of the new tracker. Any tracker therefore has at least one parent, except the top level trackers.*

Definition 3.6 *Trackers such as the top-level trackers that can guarantee their availability and serviceability are called dedicated trackers. To become a dedicated tracker, a computer must register itself to an existing dedicated tracker, except the top-level trackers.*

Definition 3.7 *A tracker can communicate with other trackers for scheduling purposes.*

Definition 3.8 *The clients or the computing nodes only communicate with the tracker in the same LAN, as long as such a tracker exists. In case there is no existing tracker, a process called self-organising is triggered, so that a most suitable tracker can be returned to the client or the computing nodes.*

Definition 3.9 *When a new computing node joins smartGRID and no tracker exists in its LAN, the node will be upgraded to a temporary tracker as a result of the self-organising process. A new computing node can also become a temporary tracker attributed to the self-organising process, if the process selects it as the replacement of an existing temporary tracker in its LAN.*

Table 3.2: Definitions of Trackers and Basic Communication Rules in smartGRID

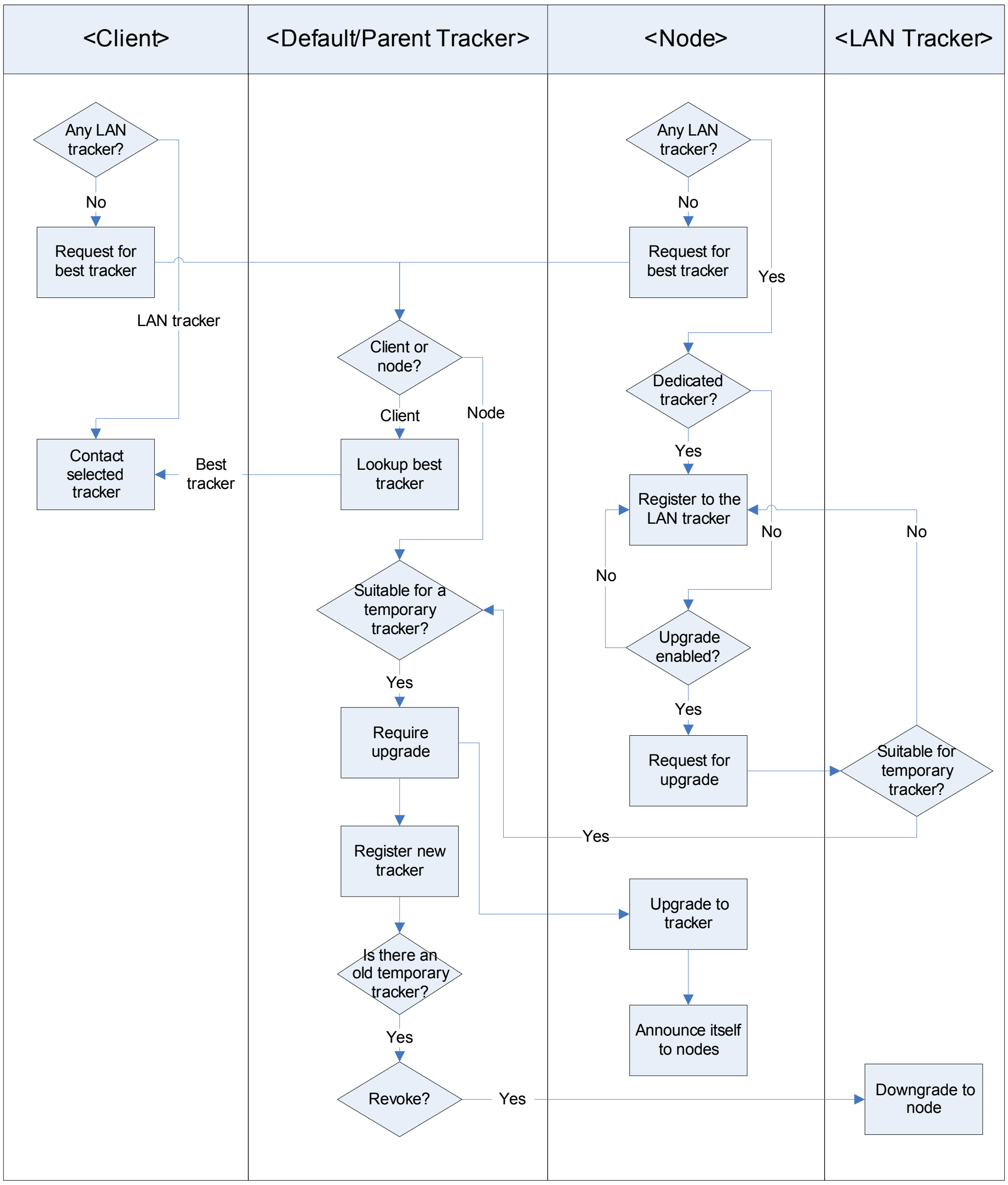


Figure 3.2: Self-organising Process in smartGRID

A microkernel Grid container runs on every computing node and tracker. The container serves as the runtime and managerial environment for the tasks. The smartGRID container consists of four components: the Runtime Environment (RT), the Management Agent (MA), the Profiling Agent (PA), and the Scheduling Agent (SA). Figure 3.3 displays its architecture.

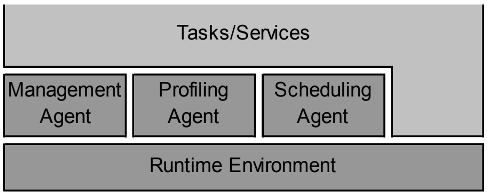


Figure 3.3: Schematic View of smartGRID Container

The Runtime Environment provides the runtime libraries and software components for both the agents and the tasks. For example, the XML parsing libraries, and the implementations of the Web Services standards [50], such as SOAP, are included in the Runtime Environment. The Management Agent provides the service and managerial interface within the Grid and for the client. The policies and configurations are managed by the MA as well. The Profiling Agent gathers the status of the network, the trackers, the computing nodes and the running tasks, and provides dynamic and optimised configurations for the Scheduling Agent. The Scheduling Agent is responsible for the scheduling and management of the tasks. It manages the lifecycle of the tasks, and provides scheduling, fault-tolerance, and load balance services for the Grid. Figure 3.4 depicts the agent interactions within a smartGRID container.

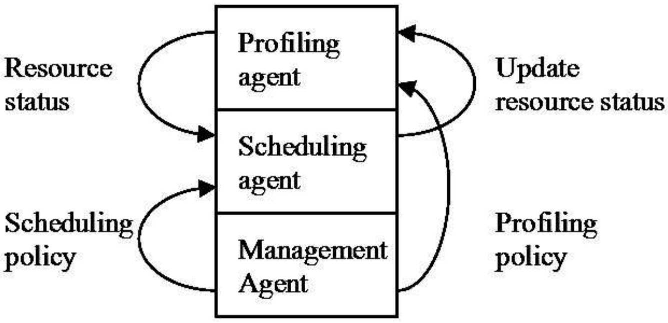


Figure 3.4: Agent Interactions in smartGRID

3.2 Task/Service - A Novel Task Model

smartGRID has a service-oriented architecture regarding its clients, and conforms to the Web Services (WS) standards [50]. The adoption of Web Services gives smartGRID good interoperability with WS-compatible clients and other WS-compatible Grids. However, in order to support state persistence and task decomposition, smartGRID introduces a novel task model, called *Task/Service (TS)*, which is a hybrid model of the conventional task model and the service model.

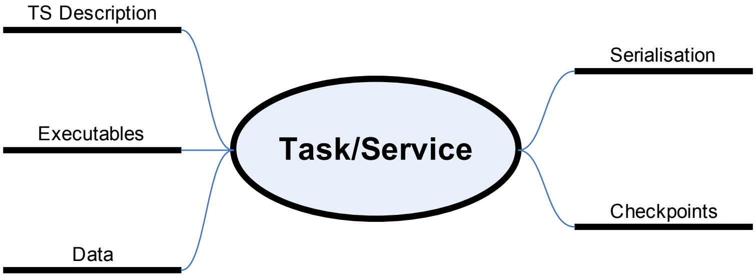


Figure 3.5: Task/Service of smartGRID

A TS comprises five components: *TS description (TSD)*, *executables*, *the data*, *serialisation*, and *checkpoints*. The serialisation and checkpoints are automatically generated and managed by smartGRID when the TS is rescheduled (i.e. when a running task is suspended). A TS without the serialisation and checkpoints is called *Raw TS (RTS)*. Figure 3.5 shows the composition of Task/Service of smartGRID.

The TSD has two sections: the task section and the service section.

{Dependencies
{Bundle dependencies
Service dependencies}
Scheduling policies
{Instance policies
Minimal hardware requirements
Estimated computation amount
Expected completion time
Priority level
Chaining policies}
Information
{Executables information
Data information
Checkpoints information}}

Figure 3.6: Task Section of the Task/Service Description

Figure 3.6 displays the task section of the TSD. The task section of TSD includes three subsections, which are described as follows:

- The dependencies subsection defines the runtime components and the services that the TS depends on.
- The scheduling policies subsection defines (a) the instance policies (the minimum number of active instances, the maximum number of active instances, the minimum number of standby instances, the maximum number of standby instances) (discussed in Section 3.3); (b) the minimum hardware requirements for machine type, processor type, the amount of cycles contributed, the amount of memory contributed, and the amount of storage contributed; (c) the estimated amount of computation; (d) the expected completion time; (e) the priority level; and (f) the chaining policies (discussed in Section 3.3).

- The information subsection defines information about the executables, the data, and the checkpoints.

The service section of the TSD uses the Web Service Description Language (WSDL) [9] and WS-Resource [17] specifications to define the service interfaces and the related stateful information.

The executables are Java bytecode files or .NET executables. The data is optional, and may come from multiple sources that are defined in the data information section of the TSD. The serialisation is equivalent to the object serialisation [49] of Java. It stores the runtime dynamics of any suspended TS. smartGRID also supports checkpoints. As not all runtime states can be preserved through the serialisation process, the checkpoint mechanism is provided to give the TS a chance to save its additional runtime states as checkpoints when the TS is suspended. When rescheduled, the TS is deserialised, and then resumed, so that the TS is able to restore its states from previous checkpoints. Checkpoints are also useful if a TS wants to roll back to its previous states.

3.3 Scheduling Process and Evolving Mechanisms

The scheduling process in smartGRID mainly involves coordinating the agents' actions within and between the Grid containers, and constructing a self-organised evolving computing network. More specifically, there are two separate processes - to schedule the TSs to suitable computing nodes, and to balance the requests and schedule the corresponding TSs to the computing nodes to serve these requests.

It is agreed that CPNs [35] are one of the best ways to model agent interaction protocols [12, 13, 42, 45]. In the CPN model of an agent interaction protocol, the protocol structure and the interaction policies are a net of components. The states of an agent interaction are represented by CPN places. Each place has an associated type determining what kind of data the place may contain. Data exchanged between agents are represented by tokens, whose colours indicate the value of the representing data. The interaction policies of a protocol are carried by CPN transitions and their associated arcs. A transition is enabled if all of its input places have tokens, and the colours of these tokens can satisfy the constraints that are specified on the arcs. A transition can be fired, which means the actions of this transition can occur when this transition is enabled. When a transition occurs, it consumes the input tokens as the parameters, conducts the conversation policy and adds the new tokens into all of its output places. After a transition occurs, the state of a protocol is changed. A protocol is in its terminated state when there is no enabled or fired transition. The detailed principles of CPNs will be discussed, together with their use, in Subsection 3.3.2.

In the rest of this section, we first discuss the lifecycle of the TS, and then explain respectively the Task-related and request-related scheduling processes mentioned at the beginning of this section. We use CPNs to describe the agent interaction protocols. We also describe the detailed algorithms used in these processes.

3.3.1 Lifecycle of the Task/Service

Figure 3.7 shows the states of a TS in its lifecycle. When a Raw TS is submitted by a client via a tracker's MA, the MA checks the TS's validity. If the TS is valid, it enters the *SUBMITTED* state. A set of pre-schedule operations are then applied to the TS by the MA and SA of the tracker. These operations include making a backup of the submitted TS, and allocating and initialising the internal resources for the purpose of scheduling that TS, etc. If all operations succeed, the TS enters the *READY* state.

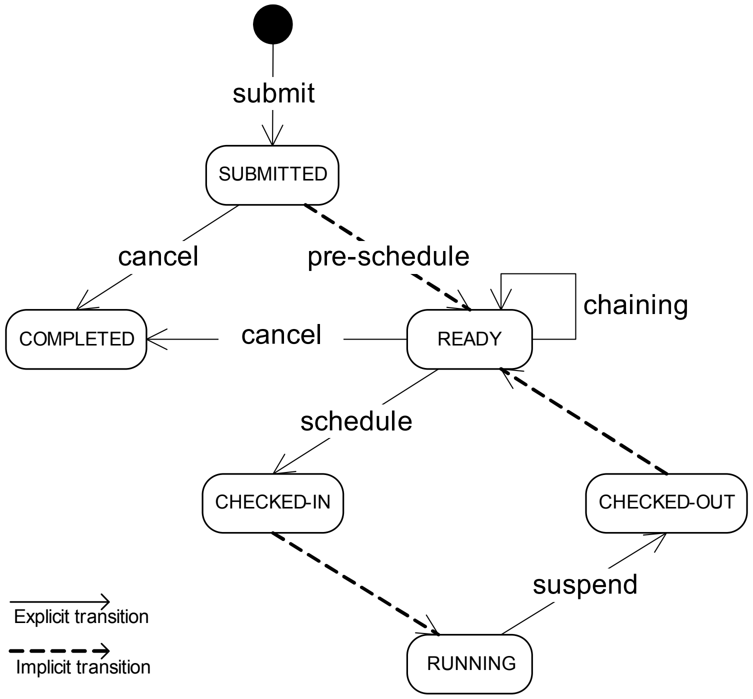


Figure 3.7: States of a Task/Service in smartGRID

The *READY* state means that the TS is ready to be scheduled. In this state, the SA of the tracker uses a "best-match" algorithm to determine whether the managed computing nodes of the tracker are suitable for the TS. If a suitable computing node is found, a schedule operation is applied. Otherwise, the SA (called *chaining source*) extracts the TSD from the TS, and passes it to the SAs of other known trackers. Every time the TSD passes by a tracker, the TTL (Time-to-Live) specified in the chaining policies of the TSD decreases by 1. If one of the trackers happens to be able to consume the TS according to the best-match algorithm, it contacts the source SA to transfer the TS to it. If the tracker is not able to consume the TS, it keeps passing on the TSD until the TTL equals 0. The above process is called *chaining*. After chaining, the TS remains in the *READY* state. Chaining is the core mechanism in smartGRID to balance the loads and requests globally. The detailed chaining and related protocols are discussed later.

The TS enters the *CHECKED-IN* state after the schedule operation, which means that the TS is scheduled to a computing node, the executables are resolved by the runtime environment of the computing node, and the runtime dynamics and checkpoint have been restored for a suspended TS. The TS then automatically enters the *RUNNING* state until the suspend operation is applied, where the TS is serialised and suspended, and enters the *CHECKED-OUT* state. Following this, the TS is automatically transferred to the tracker, where the computing node registers for rescheduling. A special situation is that if the TS exits, it fires the suspend operation itself and stores the computing result whilst being suspended.

3.3.2 Task-related Scheduling

In smartGRID's scheduling strategy, the TSs, requests, and profiles of the trackers and computing nodes are represented as three kinds of tokens. The transition rules of these tokens are different when the tokens are placed in different places. The agents in smartGRID are responsible for allocating the tokens and modifying them after the transitions are fired.

The task-related scheduling process can be described as three sub-processes: scheduling within a tracker, scheduling between the tracker and the computing nodes, and scheduling among the trackers.

Scheduling within a tracker

Figure 3.8 demonstrates the scheduling process with a tracker modelled by a CPN. There are four types of places defined in the CPN: Task-related places, operation places, the profile/load place, and the simulated synapse place. They are described as follows:

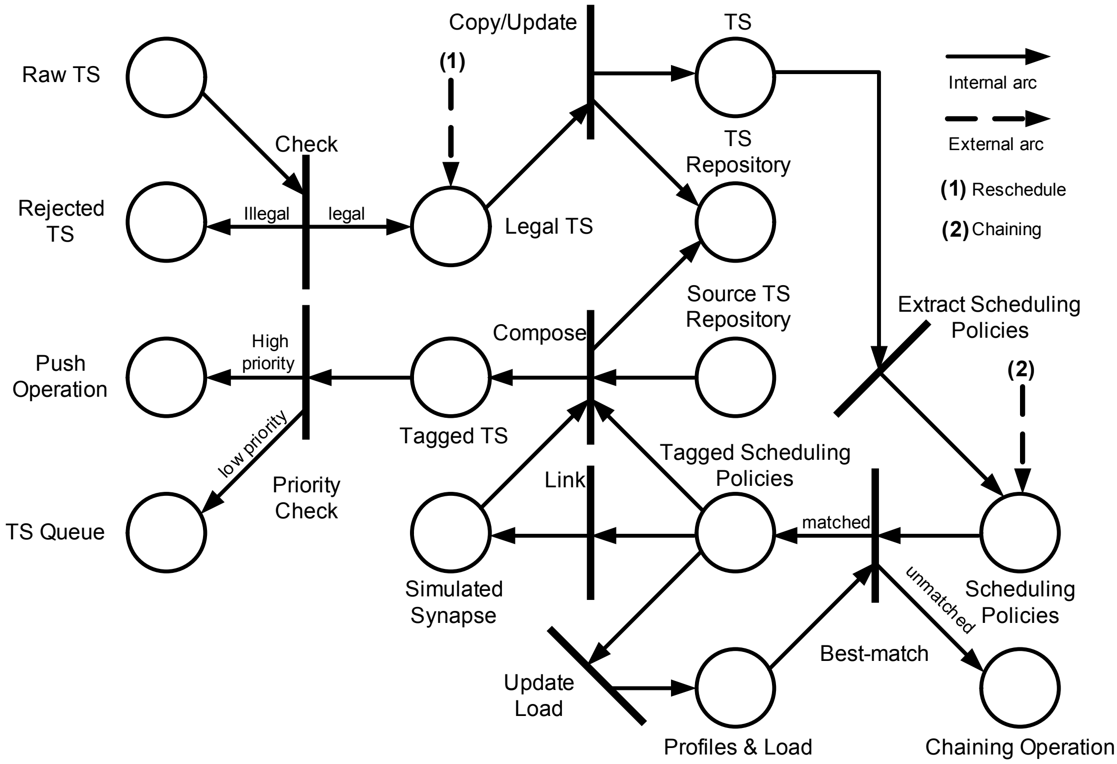


Figure 3.8: Scheduling Process within a Tracker

1. The Raw TS place holds the Raw TS token, which is received from the client.
2. The Rejected TS place holds the Raw TS tokens, which are rejected by the Check transition.
3. The Legal TS place holds the Raw TS token, which is asserted as legal by the Check transition. The legal Raw TS token may also come from the tracker itself due to a reschedule operation.
4. The TS Repository place holds the backup TS tokens. A backup TS token is removed when the corresponding TS exits or moves to another tracker through

the chaining process. A backup TS token is updated when the corresponding TS is rescheduled.

5. The TS place holds the TS token, which is produced by the Copy/Update transition.

6. The Scheduling Policies place holds the scheduling policies token, which is extracted from its corresponding TS token. The scheduling policies token may also come from another tracker through the chaining process.

7. The Profiles and Load place holds the profile tokens and load token. Each profile token contains the information and status (called *profile*) of its corresponding computing node. The load token contains the status of the overall load of its corresponding computing nodes. Figure 3.9 depicts the scheme represented by the profile token and the load token.

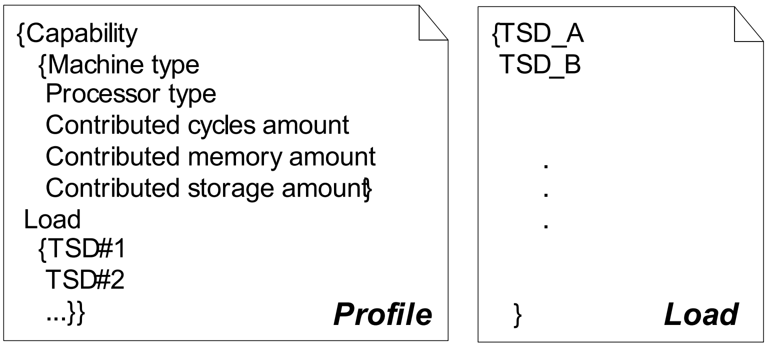


Figure 3.9: Profile and Load

8. The Chaining Operation place holds the unmatched scheduling policies token, which is consumed by the chaining process.

9. The Tagged Scheduling Policies place holds the Tagged Scheduling Policies token, which is produced by the best-match transition. The tagged token has

"winner" tags, which contain the identifiers of the best suitable nodes (the *winners*).

10. The Synapse place holds the synapse token, which represents the link between the destination and the source of a chaining process.
11. The Source TS Repository place holds the corresponding TS token of the scheduling policies token, which is passed through the chaining process.
12. The Tagged TS place holds the Tagged TS token, which is composed of the TS token and the Tagged Scheduling Policies token.
13. The Push Operation place holds the Tagged TS token, which will be "pushed" to its corresponding computing node.
14. The TS Queue place holds the Tagged TS tokens, which will be "pulled" by any of the winner nodes.

There are eight transitions, which represent eight operations. They are described as follows:

1. The Check transition checks the syntax of the TSD of the Raw TS token. It also checks whether the dependent bundles and services exist, and whether the services defined by the Raw TS conflict with the existing services (e.g. conflict due to the same service name). In addition, the Check transition converts the Raw TS token into the TS token.
2. The Copy/Update transition either duplicates the TS token, or updates the TS token in the TS repository place.

3. The Extract Scheduling Policies transition extracts the scheduling policies from the TSD.

4. The Best-match transition performs the best-match algorithm. Figure 3.10 explains the algorithm. PAES stands for Profile-Aware Eager Scheduling, which will be discussed later.

5. The Update Load transition converts the scheduling policies into the computing load, and adds the load to the overall load of the tracker.

6. The Link transition connects the two endpoints of a chaining process. Scheduling from one node to another node within the same LAN is a special case, as a tracker is always linked with itself.

7. The Compose transition transfers the TS token from the source TS repository, updates the local TS repository, and composes the Tagged TS token from the TS token and the tagged scheduling policies token.

8. The Priority Check transition compares the priority of the tagged TS token with the current loads of the winners, to determine whether the token is "pushed" to its corresponding computing node, or stored in a queue for the "pull" operation.

Scheduling between a tracker and its nodes

smartGRID uses a scheduling algorithm called *Profile-Aware Eager Scheduling (PAES)*, which is derived from eager scheduling, to schedule the TSs from the trackers to their managed computing nodes.

The eager scheduling algorithm was first introduced in Charlotte [4]. Its basic idea is that faster computing nodes will be allocated tasks more often, and if any

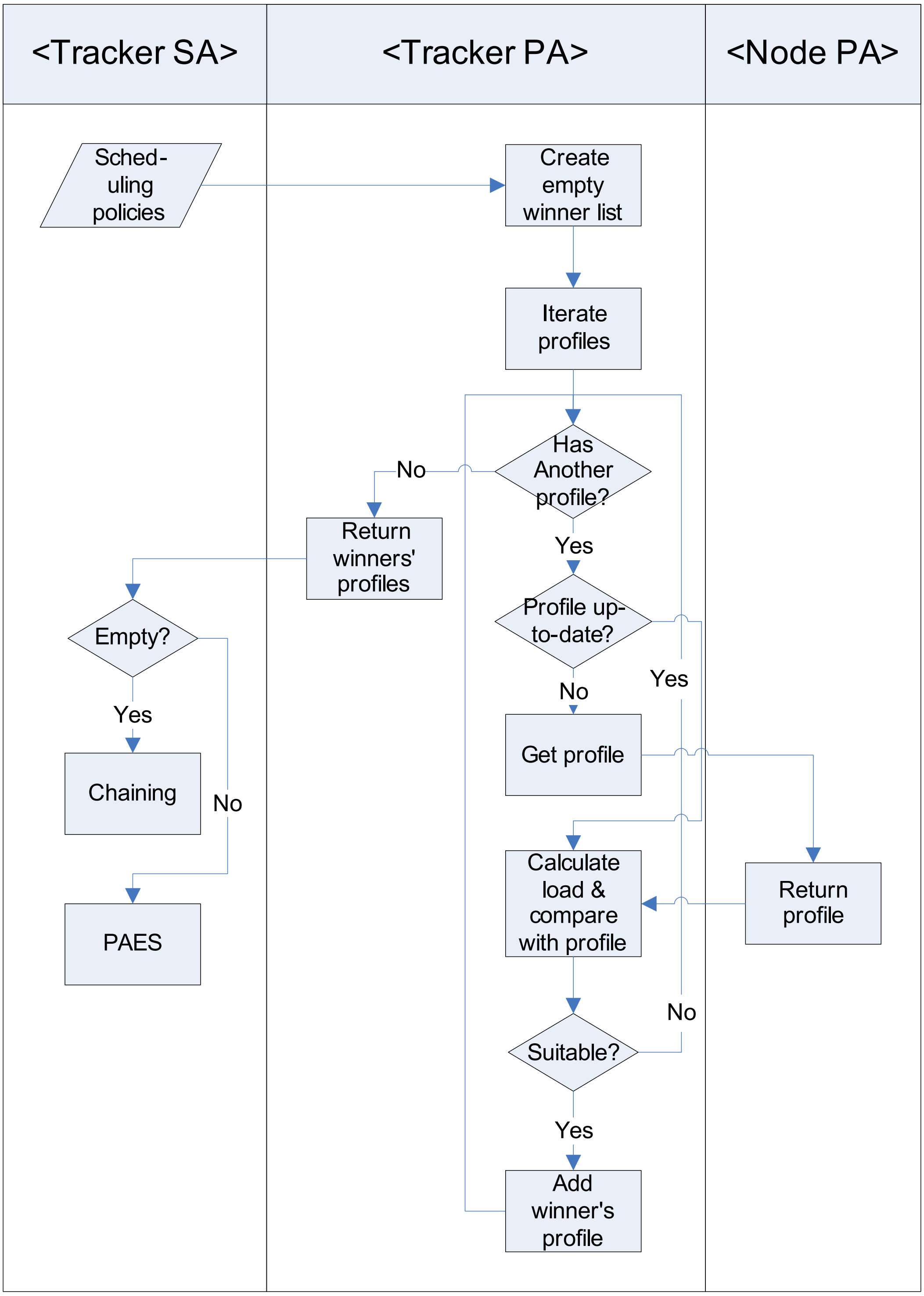


Figure 3.10: Best-match algorithm

task is left uncompleted by a slow node (failed node is infinitely slow), that task will be reassigned to a fast node. In other words, it uses a "keep the faster nodes busy" strategy. It also guarantees fault-tolerance by using a redundant task cache with a time-out mechanism. The PAES algorithm takes the profiles of the computing nodes provided by the profile agent and the scheduling policies provided by the TSs into consideration when performing scheduling. In contrast to eager scheduling, it allows bidirectional scheduling operations, i.e. pull and push. Figure 3.11 demonstrates the two operations.

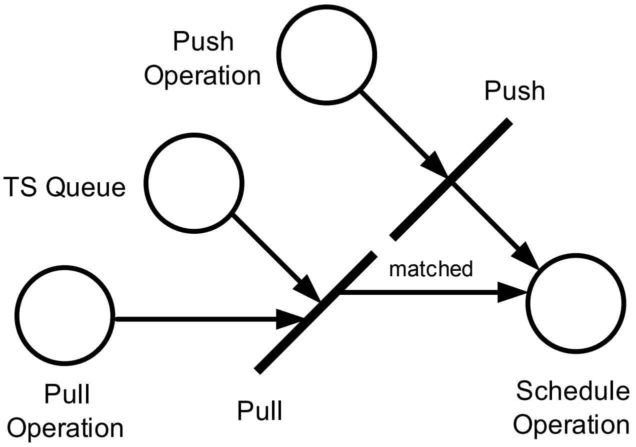


Figure 3.11: Push and Pull Operations

The Schedule Operation place holds the TS token, which is scheduled to the corresponding computing node. The push operation is straightforward. The Push transition represents the push operation, i.e. to assign the TS to one of the winners. The Pull Operation place holds the requests from the computing nodes. Whenever the scheduling agent of a node determines that it is able to run a new task, it sends a request to the tracker. The Pull transition represents the pull operation, i.e. the scheduling agent matches the computing node requesting the TSs with the tagged TS tokens. If the node is the winner of the TS, the TS is assigned to the node.

Scheduling among the trackers

Trackers are linked by the chaining process, which is the core of the scheduling process among the trackers.

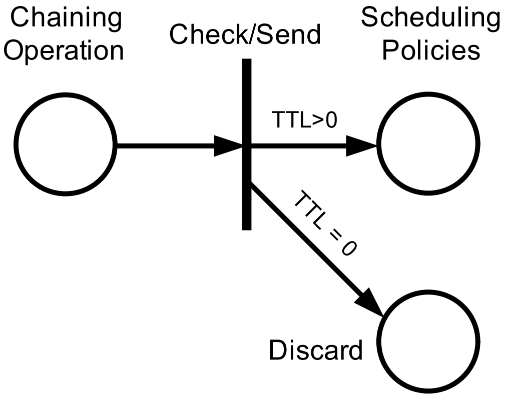


Figure 3.12: Basic Chaining Mechanism

Figure 3.12 shows the basic chaining mechanism. The two places are defined exactly the same as those in Figure 3.8. However, in this case, they represent places in different trackers. The Check/Send transition checks the TTL in the scheduling policies token first. If it is greater than 0, the TTL decreases by 1, and the scheduling policies with the new TTL is sent to all known trackers. If the TTL equals 0, the scheduling policies token is discarded.

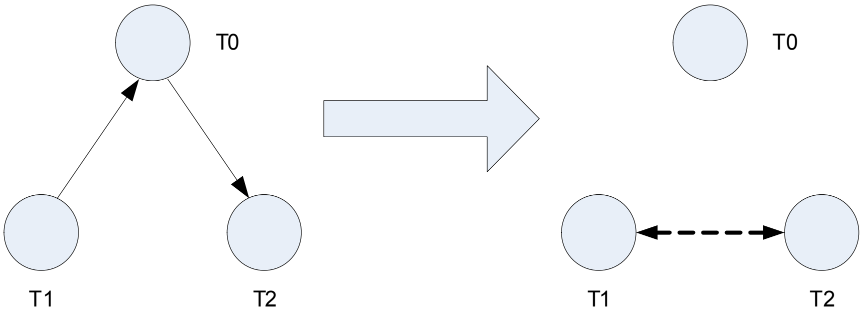


Figure 3.13: Formation of the Simulated Synapse

Recalling Figure 3.8, there is a link transition, which makes two chained trackers (i.e. if tracker A successfully schedules the chained TS of tracker B, A and B are chained) learn, and preserve each other's information for future chaining processes. However, if the links exist permanently, the performance of the chaining process will gradually decrease as time goes by because of the explosive numbers of links. A link must therefore be able to be strengthened and weakened. Such a link is called a *simulated synapse*. Figure 3.13 shows the formation of the simulated synapse.

The underlying algorithm used to strengthen and weaken the link can be defined in the chaining policies. One of the simplest algorithms is the *aging* algorithm. In this algorithm, every simulated synapse has an associated weight. A weight is a numerical value between 0 and 1, which is used to evaluate the strength of its associated chain (1 representing the strongest link, and 0 representing no link). Weight is calculated based on the frequency of communication occurring on its associated chain. When a simulated synapse is created, an initial weight is specified. Then for each interval I, the weight squares. If the resulting weight is less than the threshold θ, the simulated synapse is removed. On the other hand, each time the Link transition is fired, the square root of the weight is calculated. Table 3.3 shows a pseudo implementation of the aging algorithm using monitor.

To take the advantage of the simulated synapse, the chaining process must take the strength of the simulated synapse into consideration. Figure 3.14 demonstrates an example of the advanced chaining mechanism.

3.3.3 Request-related Scheduling

As the TSs are allowed to register services in smartGRID, one of the functions of scheduling is to balance the requests and schedule the corresponding TSs to the

```
/* Global Area */
DEFINE MONITOR M          /* monitor */
DEFINE CONSTANT $\theta$        /* threshold */
DEFINE CONSTANT I         /* interval */
DEFINE OBJECT synapse     /* simulated synapse */

/* Link transition thread */
synchronised(M) {
    if synapse.weight = 0 THEN
        INITIALISE synpase.weight
    ELSE
        synpase.weight = SQRT(synpase.weight)
    END IF
    NOTIFY();
}

/* Background daemon thread */
synchronised(M) {
    WHILE synpase.weight > $\theta$
        WAIT(I)
        synpase.weight = synpase.weight * synpase.weight
    END WHILE
    synpase.weight = 0
}
```

Table 3.3: A Pseudo Implementation of the Aging Algorithm using Monitor

computing nodes to serve these requests. In fact, the only difference between Task-related scheduling and request-related scheduling is the objects that are actually scheduled. In the former case, the object is the TS or the scheduling policies extracted from the TS. In the latter case, the object is the service request. As the requests have no common characteristic in terms of the potential load that they may bring in, it is hard for the scheduling components to make rational decisions. However, smartGRID still provides two ways to help services achieve high throughputs.

Recalling the TSD, there is a subsection called instance policies, which defines

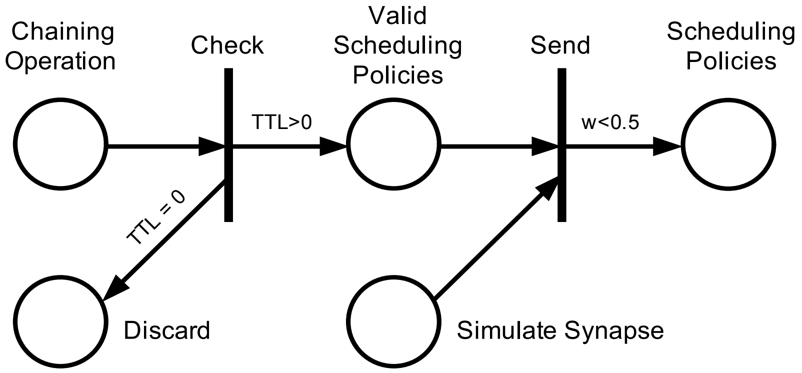


Figure 3.14: An Example of the Advanced Chaining Mechanism

the Minimum number of Active Instances (MINAI), the Maximum number of Active Instances (MAXAI), the Minimum number of Standby Instances (MINSI), and the Maximum number of Standby Instances (MAXSI). When a service TS (a TS that defines services) is scheduled, the instance policies are used to guide the scheduling components to keep a proper number of service instances. Then, when a client attempts to invoke these services, it uses the Web Services standards to discover the service instances. It is at that time that the clients' requests are distributed to the pre-allocated service instances, so that these requests are balanced.

Another way to balance service requests is to let the service providers themselves manage the requests, as only they know about the internals of the requests and the best way to handle them. The multi-agent architecture of smartGRID allows the service TSs to use the underlying APIs to provide their own scheduling strategies, and schedule the requests themselves.

3.4 Compatibility and Interoperability

In this section, we discuss compatibility and interoperability issues with existing Grid systems and clients, and how the new architecture can operate with existing local schedulers.

Recalling the Task/Service model (see Section 3.2), it is easy to see that the TS model enables the modelling of both conventional stateless services and stateful tasks. As the Web Services standards do not define whether a service is stateless or not, both stateful TSs and stateless TSs can use WSDL to register their own interfaces with clients. Therefore, any WS-compatible client is capable of accessing these interfaces through smartGRID.

There are two means by which stateful information for a conventional service in smartGRID can be maintained. The client and the service can use agreed methods, e.g. WS-Resource, to exchange stateful information. smartGRID supports WSRF standards, hence a WS-Resource based client needs no modification to work with smartGRID, as long as the service interface is not changed. Another way to preserve the states throughout different service transactions is to dynamically create transaction-specific service tasks. In smartGRID, a TS can be transaction-specific (which is specified by the instance policies in the TSD). Whenever a request for such a TS is received, a TS instance will be created to serve that request. One variation of this method is that there is a main TS serving as a proxy. Whenever a request is received by that TS, it delegates the request to a service task, which is created by the main TS. The use of a proxy task can also be extended to support task decomposition, by spawning the sub tasks from the proxy task.

As smartGRID conforms to the Web Services and WSRF standards, any TS in smartGRID is able to operate on the services in other WS-compatible Grids using these standards. However, being different in its architecture and programming model, smartGRID has neither the binary compatibility nor the source code compatibility for programs running in existing Grids.

With its multi-agent architecture, smartGRID has promising interoperability with existing local schedulers. There are two approaches. In the first approach, a local scheduler specific agent can be deployed to the local scheduler. It keeps the same interface with smartGRID and adapts itself to the scheduling and job management interface provided by the local scheduler. In the scheduling and job management process, it works as an intermedium or an adapter to interpret the scheduling and job management operations and data between smartGRID and the local scheduler. This approach is straightforward, but different local schedulers need different adapter agents. In the second approach, a more generic design of smartGRID's agents is required. Instead of hard coding a full version of the scheduling and management operations and protocols into smartGRID's agents, a set of predefined preliminary operations and protocols, which allow the construction of more complex and complete operations and protocols using a uniform scheme, are carefully selected and implemented into these agents. Hence, the scheduling and management operations and protocols of smartGRID itself and the local schedulers can be represented by these schemes. These schemes are understandable and checkable for smartGRID's agents. Once the agents are deployed, they read the schemes in, check them before any scheduling and management operation occurs, and then use them in the operations. A promising way to represent the scheme is to use CPN and the Matrix Equation Method [43], which allows the agents to check whether a scheme is understandable.

3.5 Summary

This chapter presented an agent-based hybrid Grid computing architecture, called smartGRID. This hybrid architecture, which is a combination of server-based computing architecture and P2P computing architecture, is scalable and robust in open environments.

We introduced the notion of task/service aimed at the programming issue discussed in Section 2.5. This novel task model can successfully solve the state persistence issue and task decomposition issues (using a proxy task to spawn the sub tasks).

We abandoned the conventional super-local scheduling strategy, and proposed a multi-agent based scheduling strategy. The intelligent agents in the system are able to make rational decisions and exhibit flexibility in the face of uncertain and changing factors. These advantages make the new architecture more efficient and flexible when dealing with open systems.

We extended the eager scheduling algorithm to the profile-aware eager scheduling algorithm, and introduced the best-match algorithm and chaining mechanism, which achieve local optima and global optima respectively in terms of load balance for both the tasks and the requests. The policy free best-match algorithm abstracts itself from decision-making by extracting the scheduling policies from the user configurations (i.e. the TSD). This enables sophisticated scheduling and resource utilisation.

We clarified how smartGRID preserves compatibility with WS-compatible clients, and discussed its promising interoperability with existing Grids and local schedulers.

In the next chapter, we develop smartGRID further to a pure P2P Grid computing architecture, which offers better performance and broader applications with the simplification of the structure.

Chapter 4

A Peer-to-Peer Solution to Grid Computing in Open Environments

The use of P2P computing architecture with a chaining mechanism and simulated synapse for Grid computing in open environments has been proposed and discussed in Chapter 3. This hybrid solution which has a P2P architecture in the global context and a client/server architecture in the local context has presented its flexibility and robustness in the face of uncertainty. In this chapter, we develop the hybrid solution further by applying the P2P computing architecture only. We call this pure P2P solution, smartGRID2.

In terms of the problems outlined in Chapter 2, a brief review of the hybrid solution is given in Section 4.1 to identify the remaining open problems that smartGRID has not solved. The methodologies used in smartGRID are also discussed.

Section 4.2 presents an overview of smartGRID2's architecture and core components. The three major components of smartGRID2 are briefly discussed in this section. The Grid container and its multi-agent [39] architecture are described as well.

Section 4.3 defines the improved task model, based on the notion of module. We demonstrate how the modules can be used to compose tasks, and give details of the interface between a module and the Grid container.

Section 4.4 depicts a P2P evolving computing network. We focus on the relay process, which derives from the chaining process, and explain how the relay process can bring adaptive and evolving mechanisms to smartGRID2.

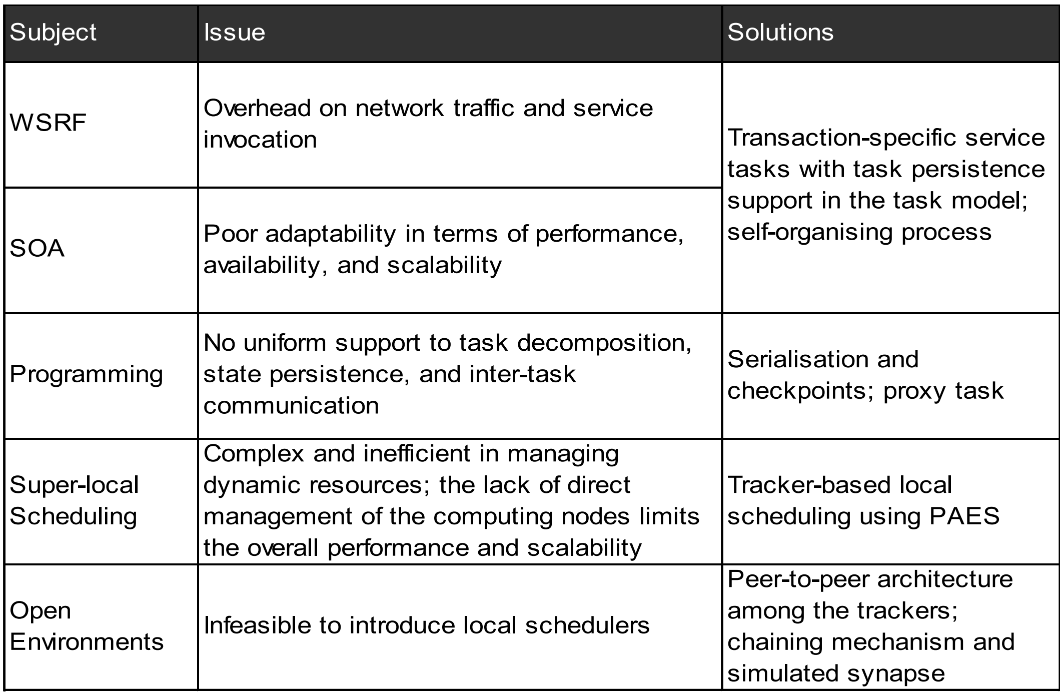
The application of the improved task model and the P2P computing architecture to resource management and scheduling is discussed in Section 4.5, with a resource management framework. The task execution process and service invocation process are also discussed, as two examples of the complete applications of the three components of smartGRID2.

As in Chapter 3, we examine compatibility and interoperability issues at the end of this chapter.

4.1 A Brief Review of the Hybrid Solution

Table 4.1 describes the problems (recall Section 2.5) with Grid computing in open environments and outlines related solutions in smartGRID. Although the hybrid solution has solved most of the problems, it has the following limits:

- It lacks internal support to transaction-specific service tasks, which are used in solving the first two problems.
- Task decomposition is supported by using proxy tasks, which is indirect and not supported in the bottom layer of smartGRID.
- Inter-task communication is not supported in the task model.
- The intention of the use of checkpoints is to preserve the heavyweight states. However, an extra layer needs to be added between the Grid container and the tasks, which increases the complexity of the architecture.



Subject	Issue	Solutions
WSRF	Overhead on network traffic and service invocation	Transaction-specific service tasks with task persistence support in the task model; self-organising process
SOA	Poor adaptability in terms of performance, availability, and scalability	
Programming	No uniform support to task decomposition, state persistence, and inter-task communication	Serialisation and checkpoints; proxy task
Super-local Scheduling	Complex and inefficient in managing dynamic resources; the lack of direct management of the computing nodes limits the overall performance and scalability	Tracker-based local scheduling using PAES
Open Environments	Infeasible to introduce local schedulers	Peer-to-peer architecture among the trackers; chaining mechanism and simulated synapse

Table 4.1: Problems and Related Solutions in smartGRID

- It lacks a uniform message passing and routing framework. Both the self-organising process and the chaining process have their own mechanisms of passing and handling messages, which is redundant and unnecessary.

- Although the hybrid architecture does have direct management of the computing nodes, the server-based computing architecture, which is used between the tracker and its managed nodes, can cause unsuitable selections of resources and unbalanced loads due to the tracker's limited view over the whole system.

4.2 Overall Architecture and Core Components

smartGRID2 consists of three major components: an improved task model, which derives from the task model of smartGRID; a P2P computing architecture, which develops the chaining mechanism and simulated synapse into a message passing and routing framework; and a resource management framework, which uses profiles to match computing resources with requests, and provides up-to-date information about matched resources. In this section, we present an overview of these components, and discuss each component in turn.

There are two tiers in smartGRID2: the clients and the computing nodes (or *peers*). Table 4.2 defines these tiers.

Definition 4.1 *A client is a generic computing device that seeks services from the Grid using Web Services Standards.*

Definition 4.2 *A computing node is the place where tasks are executed and computing occurs. A client or a tracker can act as a computing node at the same time.*

Definition 4.3 *A computing device can serve as a client and a computing node at the same time.*

Table 4.2: Definitions of Tiers in smartGRID2

A microkernel Grid container runs on every computing node. These containers serve as the runtime and managerial environment for the tasks. A task (i.e. a job or a service) is described as a group of linked modules in smartGRID2. A module is a fundamental unit that can be scheduled among the peers. All modules run on peers, or more specifically, within the smartGRID2 containers. Figure 4.1 shows the relationship between the modules and the container.

The smartGRID2 container allows modules to register to the service portal as Web Services. The service portal conforms to Web Services standards [50], and allows clients to interact with the Grid using SOAP messages. Figure 4.2 demonstrates the overall architecture of the smartGRID2 container.

Inside the container, there are four components: the Runtime Environment (RT),

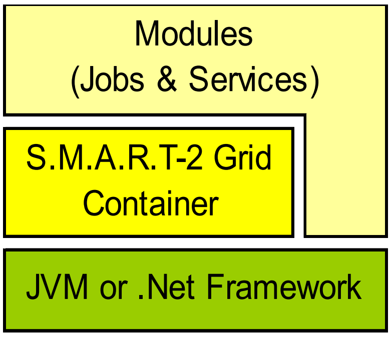


Figure 4.1: Components within a smartGRID2 Computing Node

the Management Agent (MA), the Profiling Agent (PA), and the Computing Agent (CA). The Runtime Environment provides fundamental routines and runtime libraries for both the agents and the modules. For example, XML parsing libraries, and implementations of Web Services standards [50], such as SOAP and WSDL [9], are included in the Runtime Environment; the service portal is also part of the Runtime Environment. The Management Agent provides the managerial interface between the container and the Grid Management Service. It manages the container, the policies and the configurations as well. The Profiling Agent gathers the status of the network, the peers and the running modules, and provides optimised dynamic configurations for the Computing Agent. The Computing Agent is responsible for managing the lifecycle of modules, locating resources and modules, discovering services, and scheduling modules and service invocations among the peers, while providing fault-tolerance and load balance. Figure 4.3 shows the agent interactions within the Grid container.

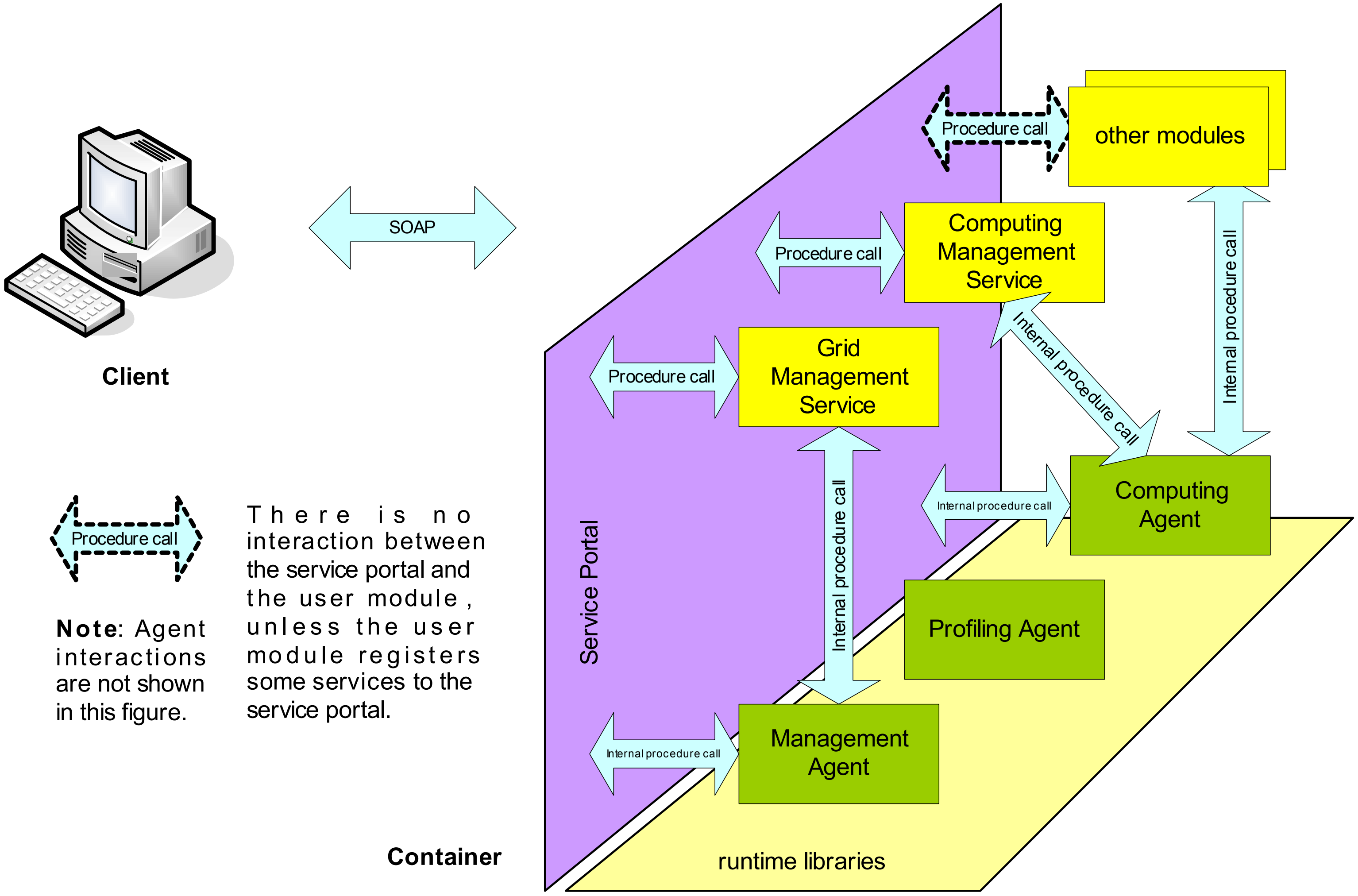


Figure 4.2: Schematic View of smartGRID2 Container

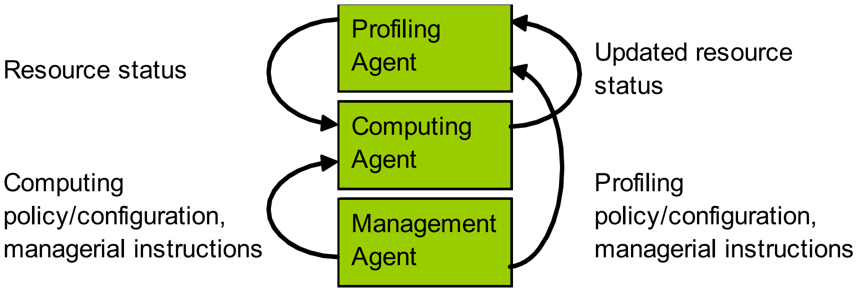

Profiling Agent
Computing Agent
Management Agent
Resource status
Updated resource status
Computing policy/configuration, managerial instructions
Profiling policy/configuration, managerial instructions

Figure 4.3: Agent Interactions in smartGRID2 Container

Besides these components, there are two predefined modules, which register as Grid Management Service (GMS) and Computing Management Service (CMS) respectively. GMS allows users who have certain privileges to manage the Grid, e.g. specifying the computing policy/configuration, and monitoring the status of the Grid. CMS provides interfaces for clients to manage the computing resources. In smartGRID2, all objects involved in the computing process are regarded as resources. These resources include the executables of the modules, the service descriptions that the modules register, data files, storage, computational cycles, etc.

4.3 Module - An Improved Task Model

As mentioned in the previous section, smartGRID2 uses modules to describe tasks. A module consists of the module description, the executables, the serialisation and the module-owned files. Figure 4.4 displays the composition of a module.

The Module Description (MD) has two sections: the task section and the service section.

Figure 4.5 shows the task section of the MD. This section defines the task-related information and consists of two subsections, which are listed as follows:

- The deployment description subsection defines information about a module's

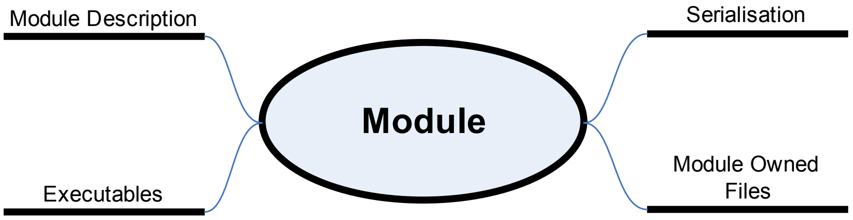


Figure 4.4: smartGRID2 Module

executables (e.g. what is the entry point of the module if it is a startup module), and the dependencies of that module. A module's dependency is another module or a service that the module depends on.

- The computing policy subsection defines a module's (a) minimum hardware requirements on a peer's machine type, processor type, and contributed cycle/memory/storage amount; (b) estimated amount of computation; (c) expected completion time; (d) priority level; and (e) relay policies (see Section 4.4).

```
{Deployment description
  {Executable description
   Module dependencies
   Service dependencies}
 Computing policies
  {Minimal hardware requirements
   Estimated computation amount
   Expected completion time
   Priority level
   Relay policies}}
```

Figure 4.5: Task Section of the Module Description

The service section of the MD is optional and is only needed if the module registers one or more services to the Grid. It uses WSDL to define the service interfaces.

The executables are Java bytecode files or .NET executables. When a running module is suspended by a user, or if it is relocated (see Section 4.5), it will be serialised. This process is equivalent to the object serialisation [49] of Java. It allows the Grid container to store the runtime dynamics of the module, and restore them when the execution of the module is resumed. The module-owned files (MOFs) are files that tightly bind to the module. These files are regarded as part of the module, and migrate, together with the module's description, executables and serialisation.

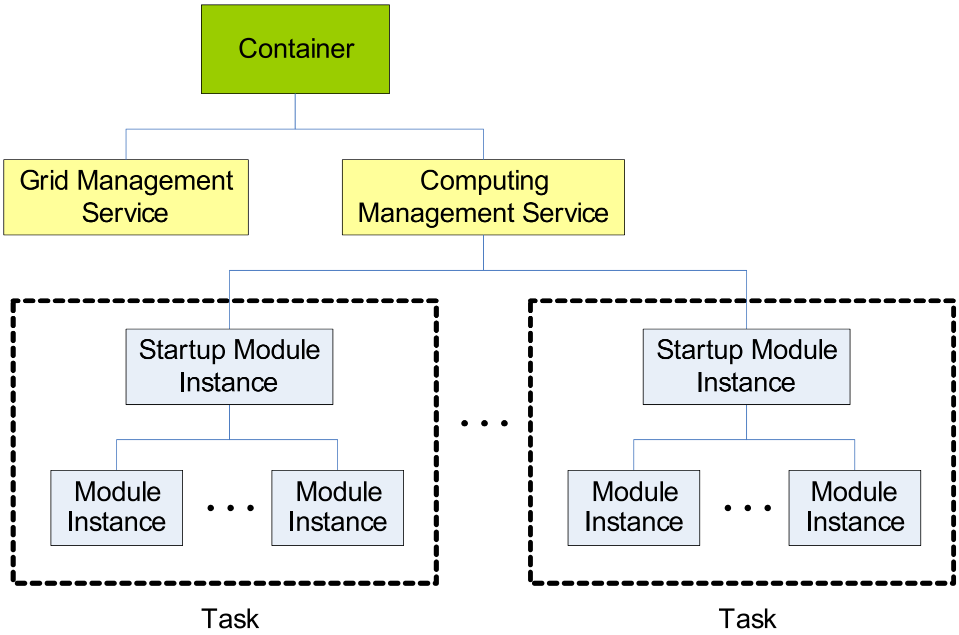


Figure 4.6: Hierarchy of the Module Instances in smartGRID2

A group of linked modules consists of a complete task. Each module implements a fraction of the overall task. As these modules can be executed at the same time on different peers, load balance and parallelism are achieved. Each task has a startup module. After all the modules of a task have been deployed to the Grid, the client can start the task through CMS. CMS then uses the `create` method of the `IModuleContext` interface to create an instance of the startup module. Once the startup module is instantiated and runs, it can start instances of other modules by using the same interface. Figure 4.6 depicts the hierarchy of the modules' instances in smartGRID2.

```
public interface IModuleContext {

    public ModuleInstance create(String moduleName,
                                 Object... args)
                          throws ModuleException;

    public Object invoke(ModuleInstance moduleInstance,
                         String method,
                         Object... args)
                  throws ModuleException;

    public void delete(ModuleInstance moduleInstance)
                throws ModuleException;

    /**
     * For static method only
     */
    public Object invoke(String moduleName,
                         String method,
                         Object... args)
                  throws ModuleException;
}
```

Table 4.3: IModuleContext Interface

When a module is instantiated, it gains access to the `IModuleContext` interface, which is provided by the Computing Agent. This interface defines three kinds of methods, which respectively allow a module's instance (a) to create instances of other modules, (b) to perform procedure calls (i.e. invoke methods of other modules), and (c) to delete the instances which are not in use in order to release their occupied resources. Table 4.3 lists the `IModuleContext` interface. The internals of the creation process, the subsequent procedure calls, and the deletion process are discussed in Section 4.5.

4.4 Peer-to-Peer Computing Architecture

A number of interconnected peers comprise smartGRID2. Table 4.4 defines the notion of connection in smartGRID2.

Definition 4.4 *A connection represents a message passing route from one peer to another, and is not equivalent to a network connection. A connection from peer A to peer B means peer A has the information to send messages to peer B successfully, where A is the source of the connection, and B is the destination of the connection.*

Definition 4.5 *A connection is directional, i.e. "peer A connects to peer B" does not presume "peer B connects to peer A". "Peer A connects to peer B" is represented as* $A \mapsto B$*. If peer B also connects to peer A, then A and B have a two-way connection, which is represented as* $A \leftrightarrow B$*.*

Definition 4.6 *A peer's connections are the connections whose source is the peer. When recording these connections, only the destination peers (destinations for short) are recorded.*

Definition 4.7 *Each connection has an associated strength. Depending on the strength, a connection can be permanent or temporary.*

Table 4.4: Definitions of smartGRID2's Connection

The peers which have a relatively large number of connections are called *hubs*. When the Grid is constructed, a number of computing nodes which have high availability, good connectivity and good performance are selected as the *backbone* of the Grid. Each of them permanently has at least two two-way connections with the others. As new nodes appear, they register to at least one of the backbone nodes, so that a two-way connection can be established between them.

```
public class Synapse {
    public double strength;
    public double deathThreshold;
    public double activateThreshold;
    public double permThreshold;

    public static Synapse createPermSynapse() {
        Synapse synapse = new Synapse();
        synapse.strength = 1;
        return synapse;
    }
    public static Synapse createTempSynapse() {
        Synapse synapse = new Synapse();
        synapse.deathThreshold =
            SynapseManager.deathThreshold +
            deathRange * random.nextDouble();
        synapse.permThreshold =
            SynapseManager.permThreshold +
            permRange * random.nextDouble();
        synapse.activateThreshold =
            synapse.permThreshold -
            (synapse.permThreshold -
             synapse.deathThreshold) * GOLDEN_SECTION;
        synapse.strength =
            Math.pow(synapse.activateThreshold, 2);
        return synapse;
    }
}
```

Table 4.5: A Sample Implementation of Simulated Synapse

In smartGRID2, the connections of a peer are recorded in a hash table, where the destinations of the connections are the keys, and the objects representing the strength of the connections (called *simulated synapses*) are the values. Table 4.5 shows a sample implementation of simulated synapse (synapse for short).

Definitions of the above fields are described in the table below.

Definition 4.8 *`strength`, whose range is (0, 1], represents the current strength of the connection. A value "1" means that the connection is a permanent connection. A random initial value which is less than `activeThreshold` is given to `strength`, when a connection is created.*

Definition 4.9 *`deathThreshold`, whose value is randomly selected from a user configured range, when a connection is created. When `strength` is less than `deathThreshold`, the connection is removed from the hash table, which means the connection breaks up.*

Definition 4.10 *`activateThreshold`. When a connection is created, a random value is selected from a user configured range as `activateThreshold`. At that stage, the connection is inactive. Afterwards, if `strength` grows to a value greater than `activateThreshold`, the connection becomes active, and the `activateThreshold` is set to 0.*

Definition 4.11 *`permThreshold`. If an active connection's `strength` continues growing to a value greater than `permThreshold`, then `strength` is set to 1, and the connection becomes a permanent connection.*

Table 4.6: Definitions of Attributes of Simulated Synapse and their Associated Rules

Two operations can be applied to a synapse: the `grow` operation, which increases the strength of the connection; and the decay operation, which decreases the strength of the connection. Table 4.7 shows the internals of these operations.

```
private static Hashtable<Peer, Synapse> synapses;

public static void grow(Peer peer) {
    Synapse synapse = synapses.get(peer);
    if(synapse == null)
        synapses.put(peer, Synapse.createTempSynapse());
    else {
        if(synapse.strength == 1)
            return;
        if(synapse.activateThreshold == 0) {
            synapse.strength =
                Math.pow(synapse.strength, 0.5);
            if(synapse.strength > synapse.permThreshold)
                synapse.strength = 1;
        }
        else {
            synapse.strength +=
                Math.pow(synapse.activateThreshold, 2);
            if(synapse.strength >
               synapse.activateThreshold) {
                synapse.strength =
                    synapse.deathThreshold +
                    (synapse.permThreshold -
                     synapse.deathThreshold) *
                    GOLDEN_SECTION;
                synapse.activateThreshold = 0;
            }
        }
    }
}

public static void decay(Peer peer) {
    Synapse synapse = synapses.get(peer);
    if(synapse.strength == 1)
        return;
    if(synapse.activateThreshold == 0)
        synapse.strength = Math.pow(synapse.strength, 2);
    else
        synapse.strength -=
            Math.pow(synapse.activateThreshold, 2);
    if(synapse.strength < synapse.deathThreshold)
        synapses.remove(peer);
}
```

Table 4.7: Operations on Simulated Synapse

There are three kinds of computing operations in smartGRID2, i.e. deploying resources, locating the resources, and utilising the resources. In order to achieve load balance, and allocate the most suitable peer to perform a computing operation or a series of computing operations, or to locate certain resources, various messages are generated by the peer which receives the client's instruction, and then delivered to other peers before performing the operation(s). These messages and the replied messages are encapsulated into impulses, and transmitted among the peers. This process is called relay. Table 4.8 shows the definition of Impulse.

```
public class Impulse {

    public int type_ttl;

    public long serial;

    public Peer from;

    public Message message;
}
```

Table 4.8: Impulse

Assume that O represents the peer which generates the message, and R represents any of the peers which reply to O. An impulse transmitted from O to R is called an *outbound impulse*. An impulse transmitted from R to O is called an *inbound impulse*. For any outbound message, the value of `type_ttl` indicates the Time-To-Live (TTL) of the impulse, and is set by O when O creates the impulse; the `serial` field contains a unique number generated by O; the `from` field is set to O; and the `message` field contains the actual message carried by the impulse. When R replies to O, it resets `type_ttl` to -1 to indicate that the impulse carries a replied message; `serial` is not changed; `from` is reset to R; and message is set to the replied message.

When a peer starts, a fixed-size queue, which is used to cache the impulses relayed by the peer, is created. `Hashtable<Long, List<Impulse>> impulses` is also created to store the inbound impulses, where the key (whose type is `Long`) denotes the serial of the impulse, and the value (which is a list of `Impulse`) denotes the inbound impulses. When a relay process starts, an outbound impulse is created by O with its fields being set, and an empty list is created and put into the hash table. Then O transmits the impulse to all of its active connections. When any of the peers receives the impulse, it checks whether the impulse is already in its queue. If it is, it discards the impulse; otherwise it decreases the TTL by one, and then checks whether TTL equals 0. If it does, the impulse is discarded; otherwise the peer appends the impulse to the end of the queue, and relays the impulse to all of its active connections. Finally, it checks whether it is able to respond to the message carried by the impulse. If it can, an outbound impulse will be generated and transmitted directly to O. Table 4.9 demonstrates the relay process.

After O transmits the impulse, it suspends the calling thread for a period of time specified before the transmission or until the number of replies reaches a threshold. Whenever a reply comes back from R to O, and there exists a corresponding list in the hash table, it is added to the list, and the grow operation is performed on the connection to R. When the thread is resumed, the replies are retrieved from the corresponding list in the hash table. Then O goes through all its connections, and performs the decay operation on the connections without a reply. Afterwards, all replies are returned to the thread for selection. Table 4.10 shows the `getReplies` method, which is implemented in CA.

```
    impulse = CA.receiveImpulse();

    if(impulse.isReply()) {
        List<Impulse> list =
            impulses.get(impulse.getSerial());
        if(list != null) {
            list.add(impulse);
            grow(impulse.getFrom());
        }
    }
    else if(queue.indexOf(impulse) == -1) {
        Handler handler =
            Container.getHandler(impulse.getMessage());
        if(--type_ttl > 0) {
            appends impulse to the queue
            CA.relay(impulse);
        }
        if(handler != null) {
            Message reply =
                handler.handle(impulse.getMessage());
            impulse.type_ttl = -1;
            Peer dest = impulse.from;
            impulse.from = Container.getLocal();
            impulse.message = reply;
            CA.send(impulse, dest);
        }
    }
/**
  * else
  *     Discard impulse
  */

    impulse.destroy();
```

Table 4.9: Relay Process

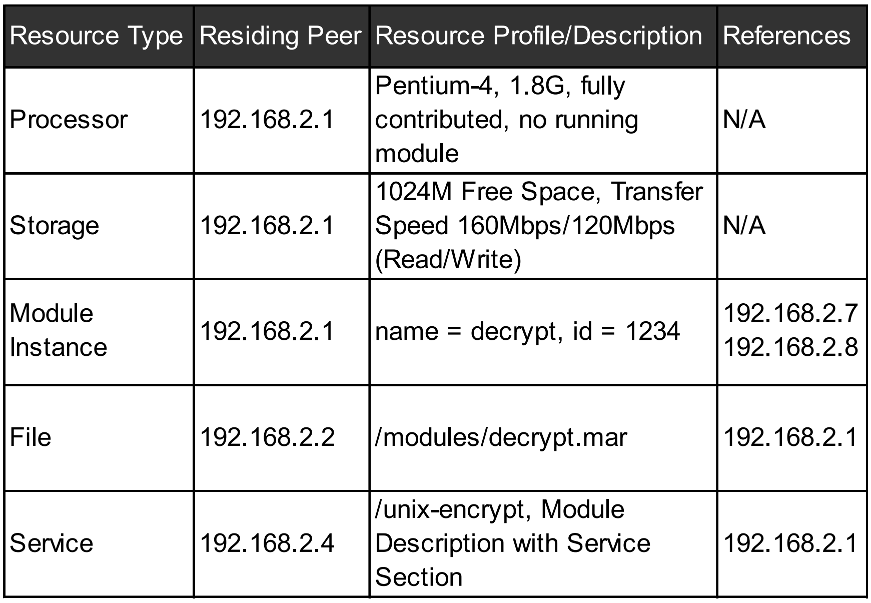
```
public static List<Impulse> getReplies(Impulse impulse) {
    List<Impulse> list =
        impulses.remove(impulse.getSerial());
    ArrayList<Peer> peers =
        new ArrayList<Peer>(synapses.size());
    peers.addAll(synapses.keySet());
    for(Impulse i : list)
        peers.remove(i.getFrom());
    for(Peer peer : peers)
        decay(peer);
    return list;
}
```

Table 4.10: getReplies Method

With the selection process (see Section 4.5), the relay process enables load balance and the election of the most suitable peer for a certain payload (i.e. the message). In the long run, the connections between the peers are optimised according to the characteristics of the payload. News hubs are also developed, so that the Grid will gain better connectivity and a higher ratio of resource utilisation, and work more efficiently.

4.5 Resource Management and Scheduling Mechanisms

smartGRID2 uses resource matrices to track the status of computing resources. Table 4.11 displays a sample of the matrix. It defines the type of resource, where the resource resides, and the resource's status (called profile) or the description of the resource. A peer's local resources are registered by the profiling agent when the peer starts. The profiling agent also updates the profiles of the local resources when they change. Figure 4.7 shows a sample definition of the processor profile.



Resource Type	Residing Peer	Resource Profile/Description	References
Processor	192.168.2.1	Pentium-4, 1.8G, fully contributed, no running module	N/A
Storage	192.168.2.1	1024M Free Space, Transfer Speed 160Mbps/120Mbps (Read/Write)	N/A
Module Instance	192.168.2.1	name = decrypt, id = 1234	192.168.2.7 192.168.2.8
File	192.168.2.2	/modules/decrypt.mar	192.168.2.1
Service	192.168.2.4	/unix-encrypt, Module Description with Service Section	192.168.2.1

Table 4.11: A Sample Resource Matrix of Peer 192.168.2.1

When a module requires a resource, its container C may match the required resource with those in the resource matrix first. If none of them matches the requirement, the container starts a relay process. Alternatively, the container may start the replay process immediately upon receiving the module's request. How the container behaves is determined by the type of resource. For example, local service resources have precedence over remote service resources, but there is no such difference in terms of processor resources.

During the relay process, the participating peers look up the required resource in their resource matrices. If matching resources exist, the references to these resources are returned to C. If multiple replies exist, C starts a resource selection process to determine which resource is the most suitable one. The outcome is then returned to the module for its subsequent operations. And if the type of resource located has local precedence, it will be cached in the resource matrix of C. A resource matrix only caches a limited number of references. Each time a cached reference is retrieved, it is regarded as "updated". The least updated entry will be removed if the cache is full and a new reference comes in.

```
{Capability
  {Machine type
   Processor type
   Contributed cycles amount
   Contributed memory amount
   Contributed storage amount}
 Load
  {Module#1
   Module#2
   ...}}
```

Figure 4.7: A Sample Processor Profile Definition

Once the reference to a resource is obtained, the resource is accessible to the module through smartGRID2. Each time a resource is accessed, its reference is quoted and passed to the resource's residing peer R. Then R will perform the actual operations and send the results back to the module. When the module finishes using the resource, it notifies R so that the resource can be released on R.

A peer also keeps records of other peers which have cached references to its local file, service, and module instance resources, so that the references can be updated when the actual resources migrate to other peers.

In smartGRID2, files can be uploaded to the backbone nodes through CMS. Unlike other resources, local files never appear in the resource matrix. When a file is located and used, it can be cached by the peer that uses the file, if there is sufficient storage on that peer.

The executables of any modules are regarded as files, and need to be uploaded to the Grid before the execution of the module. smartGRID2 has a two-stage scheduling mechanism. Once a module is uploaded, its residing peer O will trigger a relay process, informing other peers of the potential workload. Other peers will reply to O if they can execute the module. O then determines the suitability of these peers (including O). The module will be moved to the winner if the winner is a backbone node; otherwise it is transferred to the winner and cached there. At the second stage, when the module is about to be created, a relay process will be started to locate the module. Once it is located, it will be scheduled and executed by its residing peer. Table 4.12 shows the reference to the instance of the module used in the procedure calls.

```
public class Resource {

    private Peer peer;
}

public final class ModuleInstance extends Resource
                               implements Serializable {

    public String name;

    public String id;
}
```

Table 4.12: Reference of Module Instance

The only difference in a service's execution process is that the service has to be discovered before its module execution process. Figure 4.8 demonstrates that process.

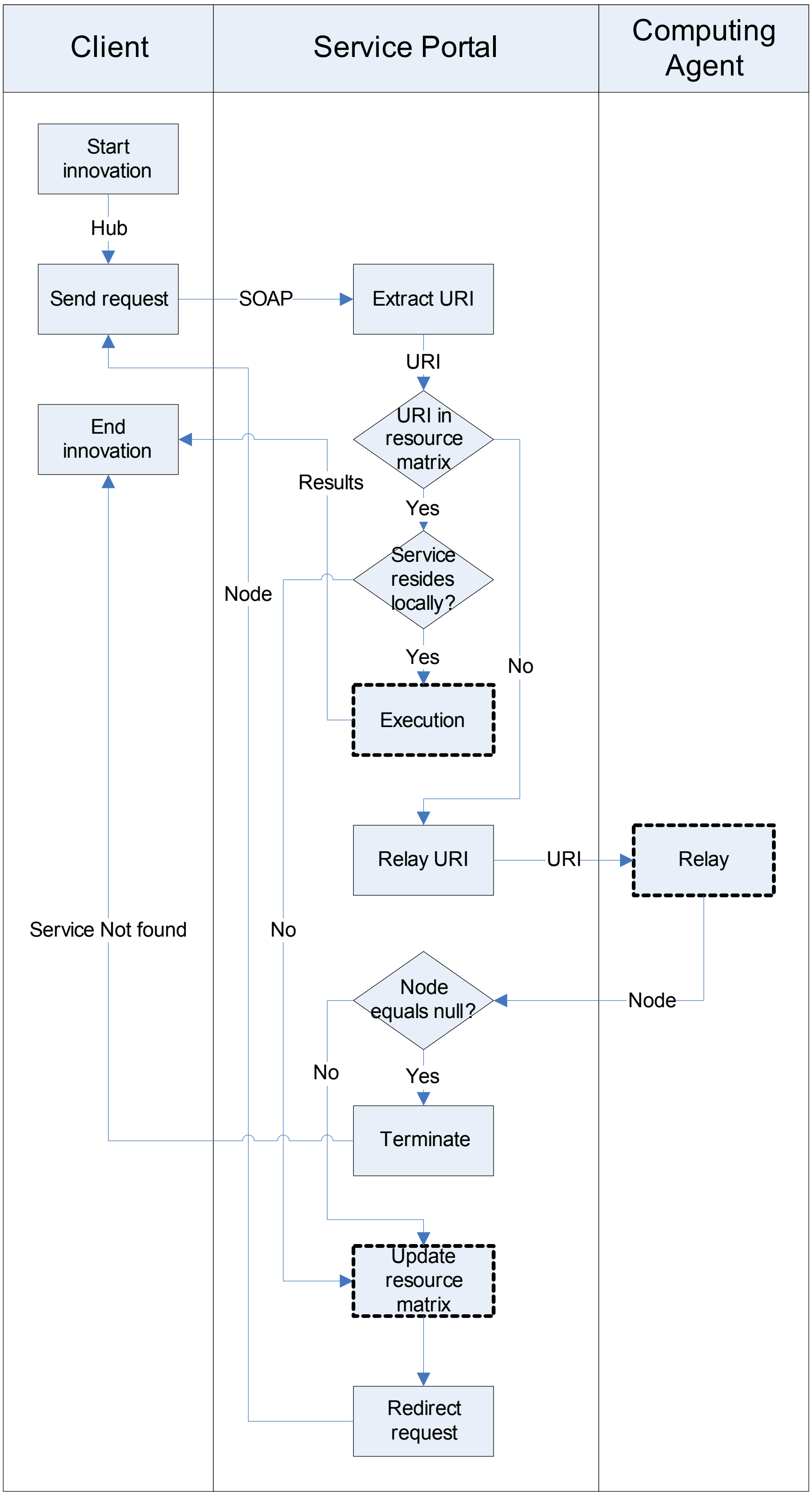


Figure 4.8: Service Invocation Process

4.6 Compatibility and Interoperability

In this section, we discuss compatibility and interoperability issues with existing Grid systems and clients.

Recalling the task model (see Section 4.3), it is easy to see that the new model enables the modelling of both the conventional stateless services and stateful tasks. A module is allowed to register its own services to the service portal using Web Services standards. Hence, any WS-compatible client is capable of accessing these services through smartGRID2.

There are two means by which maintain stateful information for a service in smartGRID2 can be maintained. The client and the service can use agreed methods, e.g. WS-Resource, to exchange stateful information. smartGRID2 supports WS-Resource standards, hence a WS-Resource based client needs no modification to work with smartGRID2, as long as the service interface is not changed. Another way to preserve the states throughout different service invocations is to create a transaction-specific service module. In this case, a token representing a certain transaction is passed in the service invocations. When a new transaction starts, the startup module of the service creates a new service module to serve the transaction. The stateful information is maintained by the service modules. The tokens act as the identifiers for the startup module to dispatch the service invocations to an appropriate service module. Once the transaction is done, the client implicitly notifies the service's startup module, so that the startup module can delete the corresponding service module and release the resources.

As smartGRID2 conforms to Web Services and WS-Resource standards, any module in smartGRID2 is able to operate on the services provided by other WS-compatible Grids using these standards. However, being different in its architecture and programming model, smartGRID2 has neither the binary compatibility nor the source code compatibility for programs running in existing Grids.

4.7 Summary

Based on the hybrid solution proposed in Chapter 3, we developed smartGRID2 to tackle the limits and remaining problems presented by smartGRID. Aiming at solving these problems, we proposed our novel task model. With the serialisation and module-owned files, the internal states of a task are easy to maintain, and the process is totally transparent to the users. As for the adaptability of the services, the freedom of how to approach it is left to the programmers. The simplest solution is to spawn more service modules to accommodate new service requests. With the help of the relay process and resource matrices, the new task model provides a common programming interface that supports task decomposition, state persistence, and inter-task communication. All the support is through the `IModuleContext` interface. Instead of using the super-local strategy, which can cause several problems in open environments, we applied the P2P computing architecture to the Grid, and innovatively proposed a multi-purpose message passing and routing mechanism and a generic resource selection mechanism to achieve load balance, a high ratio of resource utilisation, and fault-tolerance. These mechanisms also allow the Grid to intelligently reconstruct and utilise computing resources. Finally, we clarified how smartGRID2 preserves compatibility with WS-compatible clients, and discussed its promising interoperability with existing Grids.

Chapter 5

Conclusions

Hundreds of thousands of computers in the Internet form a computing resource pool with tremendous computational power and storage, as well as a great variety of services and contents. For years, computer scientists have been chasing after the vision of a worldwide computer that can utilise all these resources and services. Computing Grids, as one of the emerging technologies that aim at making the above vision reality, have "generated" enormous computing power for scientific research and have "incrementally scaled the deployment of relatively sophisticated services and application, connecting small numbers of sites into collaborations engaged in complex scientific applications" [18]. As the scale of systems increases, Grid computing is now facing and addressing problems relating to high autonomy and heterogeneity, intermittent availability, and dynamic and variable factors, which we call open environments.

The primary objective of this thesis was to solve the fundamental issues relating to the architecture of Grid computing in open environments. More specifically, we investigated conventional Grid computing architecture and proposed new architectures with a number of self-configuring, adaptive and evolving mechanisms, by answering the following targeted research questions.

- What is the best way to support task decomposition, inter-task communication, and state persistence?
- What is the best strategy for resource management and scheduling?
- How can compatibility and interoperability be provided? And to what level?

Using these questions as a basis, we reviewed the state of the art of Grid computing, and pointed out that five problems obstruct the application of the conventional Grid in open environments:

- WSRF can result in significant overheads on network traffic and object invocation.
- Current service-oriented architecture has poor adaptability in terms of performance, availability, and scalability.
- The dependence on local schedulers increases the complexity of application programming.
- The super-local resource management and scheduling strategy limits overall performance and scalability.
- It is not feasible to introduce local schedulers into open environments.

In the rest of this chapter, we present the major contributions of this thesis with a comparison of the two solutions. We then discuss the remaining questions of Grid computing in open environments, and outline potential research directions.

5.1 Discussion and Major Contributions of the Thesis

In Chapter 3 and Chapter 4, we proposed smartGRID and smartGRID2 as two solutions to the problems mentioned in the previous section. Although smartGRID2, as the development of smartGRID, solves the remaining problems presented by smartGRID (see Section 4.1), they have different concerns from the practical point of view. We explain these differences and outline our contributions in the following subsections.

5.1.1 Task Model

Both the task model of smartGRID and its client/server computing architecture in the local context is based on the notion of "job". Hence, there is almost no difference conceptually between the conventional Grid and smartGRID in how computations are carried out - the user submits jobs to the Grid, the Grid runs the jobs and finally gives the results back to the user. The major difference between smartGRID and the conventional Grid is in their architectures, where the former allows deployment and application in open environments.

The task model of smartGRID2 was designed from the beginning based on the view of a "virtual machine" that consists of computing nodes in an open environment, and therefore requires tasks to run across distributed and heterogeneous computing nodes. The notion of "module" is used as the atomic "job" that can be scheduled to any single node, yet the real job or task that a user executes in smartGRID2 is represented by a group of modules. The advantages of the new task model are:

- It solves the issues of task decomposition, inter-task communication, and state persistence. Furthermore, it provides a transparent programming interface that frees the developers from having to take care of the above issues.

- Each module can represent a software component that provides a certain functionality, hence allowing the construction of new tasks based on existing modules.

- By developing general-purpose fundamental programming utilities based on the novel task model, larger and more complex programs and applications can be built. For example, imagine a program that requires a hash table containing hundreds of thousands of entries. An immediate solution is to implement a distributed hash table. But without underlying support, such a solution have to consider several issues, such as the organisation of the distributed nodes, message passing, fault-tolerance, etc. Normally, the practical solution to such problems is some sort of "workaround", where the task is decomposed into small, "handleable" tasks, each of which only requires ordinary hash table implementation. However, if the task is simple enough, and cannot be decomposed, the only solution is the distributed hash table. With the current module-based task model, such a distributed hash table is easy to implement.

- The module-based task model allows more interactions between the users and the applications. In contrast to the conventional task model, where the only interaction is the submission of jobs and the retrieval of results, the users of smartGRID2 can run their applications interactively without being aware of the distributed environment. This is because the modules that comprise an application can immediately be scheduled and executed according to their priority levels (recall Section 4.3), and can therefore provide intermediate results that are essential for interactions in most cases. Besides, the `IModuleContext` interface allows client programs to invoke other modules directly, hence the client can provide more diverse user interfaces than those used in the conventional Grid.

5.1.2 Computing Architecture

The different computing architectures of smartGRID and smartGRID2 are partially attributed to their different task models. Another factor that determines their computing architectures is how they schedule the tasks.

smartGRID schedules a task in the LAN first, and if the task cannot be scheduled, it is transferred to another suitable LAN. The major consideration of this design is based on the fact that conditions of a LAN are always better than those of a WAN, hence scheduling in a LAN gains better performance and efficiency in most cases. Since LAN-based scheduling is the first choice, there must be a place to store the tasks, and forward them to other LANs if they cannot be scheduled. The simplest and most effective way is to use client/server computing architecture, part of which are the trackers. As the trackers are physically connected through WANs, the use of client/server architecture among the trackers will lose its advantage, because there is no guarantee of good network connections to the server, and the limited computational power of the server will become a bottleneck sooner or later, when the scale of the system increases. That is why P2P architecture is selected.

The major issues that P2P architecture introduces are the method of decision-making as well as the dynamic and changing factors. The chaining mechanism and the simulated synapse are introduced to tackle these issues. In a peer group, voting is often used as the decision-making strategy, where each participant can vote according to a set of pre-agreed criteria, and the result is collected and processed by an authority. The strategy used in smartGRID is a variant of the voting strategy, where the conceptually-connected peers "vote" and make decisions directly, without the participation of an "authority". The tracker which starts the "voting" follows the "first come, first served" rule, hence the first respondent is chosen to schedule the task. Obviously, without the votes of all the peers, a decision might not be the best. However, in smartGRID's scenario, this solution is the most effective one, because no tracker has full knowledge of the rest of the trackers, and therefore there is no way to gather all the "votes". The main purpose of the simulated synapse is to maintain the efficiency of the P2P computing network. As stated in Section 3.3, it guarantees that each tracker will not have too many links with others, and with the help of the advanced chaining mechanism, it guarantees the balance of load and the response speed, since the messages are always sent to the trackers with relatively lower strength, which means they have less load in all probability.

The scheduling architecture is changed in smartGRID2. The design of smartGRID2 avoids using any type of client/server computing architecture. The concern that is different from smartGRID is that WAN can still be robust enough to accomplish scheduling. Consequently, the chaining mechanism and the simulated synapse are extended to the whole structure, and the computing architecture evolves into a message multicast and routing framework. In the meantime, the decision-making strategy is changed. The system does not use the "first come, first served" rule any more. Instead, choices are given to the message sender - the decision is not made until there are enough "votes", or until one of the "votes" is good enough, or until the prescribed response time is up. The reason for doing this is based on the belief that the message sender is most suitable entity to determine the decision-making strategy according to its requirements.

Both the hybrid architecture and the P2P architecture have various advantages over the conventional Grid architecture:

- Both solutions have direct management of the computing nodes, and load balance is guaranteed by various mechanisms.
- The adaptability of the services can easily be achieved by spawning subtasks to serve increased requests.
- Both solutions are resilient to faults by keeping redundant copies of tasks and their intermediate results.
- The P2P solution transparently supports inter-task communication to the developers through the `IModuleContext` interface.

5.1.3 Resource Management Framework

There is no generic resource management framework in smartGRID. It does, however, have a resource selection algorithm called the best-match algorithm (recall Section 3.3). The best-match algorithm is based on the profiles of computing resources. By comparing the description of a message and the profiles of the required resources, the algorithm helps the scheduling process to select the most suitable computing nodes.

In smartGRID2, this idea is developed further. smartGRID2 takes everything as resources, e.g. files, storage, services, even the instances of modules. Every resource has a profile or description, and every type of resource has a resource handler, which is used in the resource selection process to match certain types of resources with messages. The benefits are that 1) the framework takes care of how to store the profiles and descriptions of resources, and maintains valid references to resources; 2) the framework allows developers to define their own types of resources with user-defined resource handlers, so that the applications can take advantage of the framework; and 3) the process of handling a message can be represented by a very simple logic with the resource operations and the relay process(recall Figure 4.8). Compared with the centralised resource management framework of the conventional Grid, the distributed resource management framework used in smartGRID2 is therefore more efficient and robust.

5.2 Future Work

A full solution to Grid computing in open environments relies on answering the six questions outlined in Chapter 1. Figure 5.1 shows the research areas that these questions belong to.

In this thesis, we have solved problems in the area of the programming model, and resource management and scheduling. We have also addressed related compatibility and interoperability issues. The remaining three areas are considered to be the future work of this research:

- Organisational Hierarchy. The major concerns of this area include how to organise and manage the physically distributed computing resources to model the hierarchy of an organisation, and how to store the logically centralised information/resources, such as identity, access control information, software repository and files.

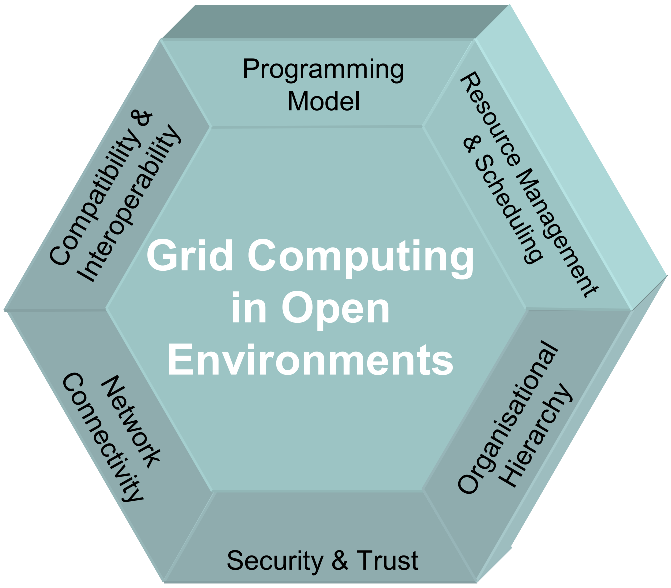

Programming Model
Resource Management & Scheduling
Organisational Hierarchy
Security & Trust
Network Connectivity
Compatibility & Interoperability
Grid Computing in Open Environments

Figure 5.1: Research Areas concerning Grid Computing in Open Environments

- Security and Trust Model. This area investigates issues of authentication and authorisation, as well as encryption of data and communications.
- Network Connectivity. This area solves connectivity problems between participants in the Grid. For example, two computing nodes behind different firewalls should be able to communicate with each other.

Appendix A

Glossary of Terms

C/S *Client/Server*

CA *Computing Agent*

CMS *Computing Management Service*

CPNs *Coloured Petri Nets*

GMS *Grid Management Service*

GRAM *Grid Resource Allocation and Management*

GSI *Grid Security Infrastructure*

GT4 *Globus Toolkit 4*

MA *Management Agent*

MAXAI *Maximum Number of Active Instances*

MAXSI *Maximum Number of Standby Instances*

MD *Module Description*

MDS *Monitoring and Directory Service*

MINAI *Minimum Number of Active Instances*

MINSI *Minimum Number of Standby Instances*

MOFs *Module-Owned Files*

MPI *Message Passing Interface*

MPPs *Massively Parallel Processors*

OGSA *Open Grid Services Architecture*

OGSI *Open Grid Service Infrastructure*

P2P *Peer-to-Peer computing*

PA *Profiling Agent*

PAES *Profile-Aware Eager Scheduling*

PVM *Parallel Virtual Machine*

QoS *Quality of Service*

RT *Runtime Environment*

RTS *Raw Task/Service*

SA *Scheduling Agent*

SOA *Service Oriented Architecture*

SOAP *Simple Object Access Protocol*

TS *Task/Service*

TSD *Task/Service Description*

TTL *Time-to-Live*

VOs *Virtual Organisations*

WS *Web Services*

WSDL *Web Serivces Description Language*

WSRF *Web Services Resource Framework*

Bibliography

[1] E. Akarsu, G. C. Fox, W. Furmanski, and T. Haupt. WebFlow: High-level Programming Environment and Visual Authoring Toolkit for High Performance Distributed Computing. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, San Jose, California: IEEE Computer Society Press, November 1998, pp. 1–7.

[2] W. L. Alfred. The Future of Peer-to-Peer Computing. *Communications of the ACM*, 46(9): 56–61, September 2003.

[3] M. Baker, R. Buyya, and D. Laforenza. Grids and Grid Technologies for Wide-area Distributed Computing. *Software: Practice and Experience*, 32(15): 1437–1466, November 2002.

[4] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *Proceedings of the 9th Conference of Parallel and Distributed Computing Systems (PDCS-96)*, Dijon, France: September 1996, pp. 181–188.

[5] R. Buyya. *Grid Computing Info Centre: Frequently Asked Questions (FAQ)*. 2002. http://www.gridcomputing.com/gridfaq.html.

[6] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid. In *Proceedings of the 4th International Conference on High Performance Computing in Asia-Pacific Region (HPC ASIA'2000)*, vol. 1, Beijing, China: IEEE Computer Society Press, May 2000, pp. 283–289.

[7] P. Cappello and D. Mourloukos. A Scalable, Robust Network for Parallel Computing. In *Proceedings of the 2001 joint ACM-ISCOPE Conference on Java Grande*, Palo Alto, California: ACM Press, June 2001, pp. 78–86.

[8] M. Chetty and R. Buyya. Weaving Computational Grids: How Analogous Are They With Electrical Grids. *Computing in Science and Engineering*, 4(4): 61–71, July-August 2002.

[9] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *Web Services Description Language (WSDL) 1.1.* 2001. http://www.w3.org/TR/wsdl.

[10] B. O. Christiansen, P. Cappello, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-based Parallel Computing using Java. *Concurrency: Practice and Experience*, 9(11): 1139–1160, November 1997.

[11] B. Cohen. Incentives Build Robustness in BitTorrent. May 2003. http://www.bittorrent.com/bittorrentecon.pdf.

[12] R. S. Cost. Modeling Agent Conversations with Coloured Petri Nets. In *Proceedings of the Workshop on Specifying and Implementing Conversation Policies*, Seattle, Washington: May 1999, pp. 59–66.

[13] S. Cranefield, M. Purvis, M. Nowostawski, and P. Hwang. Ontology for Interactison Protocols. In *Proceedings of the 2nd International Workshop on Ontologies in Agent Systems (AAMAS'02)*, Bologna, Italy: July 2002, pp. 15–19.

[14] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. In *Proceedings of IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, Orlando, Florida: March-April 1998, pp. 62–82.

[15] D. H. J. Epema, M. Livny, R. vanDantzig, X. Evers, and J. Pruyne. A Worldwide Flock of Condors: Load Sharing among Workstation Clusters. *Future Generation Computer Systems*, 12(1): 53–65, May 1996.

[16] S. Fitzgerald, I. Foster, C. Kesselman, G. v. Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *Proceedings of the 6th IEEE Symposium on High Performance Distributed Computing*, Portland, Oregon: IEEE Press, August 1997, pp. 365–375.

[17] I. Foster, K. Czajkowski, D. Ferguson, J. Frey, S. Graham, T. Maguire, D. Snelling, and S. Tuecke. Modeling and Managing State in Distributed Systems: the Role of OGSI and WSRF. *Proceedings of the IEEE*, 93(3): 604–612, March 2005.

[18] I. Foster and A. Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS 2003)*, Lecture Notes in Computer Science (LNCS), vol. 2735, Berkeley, California: Lecture Notes in Computer Science (LNCS), Springer-Verlag, Feburary 2003, pp. 118–128.

[19] I. Foster. What is the Grid? A Three Point Checklist. *Grid Today*, 1(6), July 2002. http://www.gridtoday.com/02/0722/100136.html.

[20] I. Foster. *A Globus Toolkit Primer.* 2005. http://www-unix.globus.org/toolkit/docs/4.0/key/GT4_Primer_0.6.pdf.

[21] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications and High Performance Computing*, 11(2): 115–128, Summer 1997.

[22] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kauffman, San Francisco, California, 1st ed., 1999.

[23] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration.* June 2002. http://www.globus.org/research/papers/ogsa.pdf.

[24] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3): 200–222, Fall 2001.

[25] G. Fox, T. Haupt, E. Akarsu, A. Kalinichenko, K.-S. Kim, P. Sheethalnath, and C.-H. Youn. The Gateway System: Uniform Web Based Access to Remote Resources. In *Proceedings of the ACM 1999 Conference on Java Grande*, San Francisco, California: ACM Press, June 1999, pp. 1–7.

[26] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proceedings of the 10th IEEE Symposium on High Performance Distributed Computing (HPDC10)*, San Francisco, California: IEEE Press, August 2001, pp. 55–63.

[27] A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, Massachusetts, 1994.

[28] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. PVM and MPI: a Comparison of Features. *Calculateurs Paralleles*, 8(2): 137–150, June 1996.

[29] Global Grid Forum. Open Grid Services Infrastructure (OGSI) Version 1.0, June 2003. http://www-unix.globus.org/toolkit/draft-ggf-ogsi-gridservice-33_2003-06-27.pdf.

[30] Globus Alliance. *Globus Toolkit 4.0 (GT4).* 2005. http://www-unix.globus.org/toolkit/docs/4.0/GT4Facts/.

[31] J. Goldman, P. Rawles, and J. Mariga. *Client/Server Information Systems.* Wiley, Hoboken, New Jersey, 1999.

[32] A. Grimshaw, A. Ferrari, G. Lindahl, and K. Holcomb. Metasystems. *Communications of the ACM*, 41(11): 46–55, November 1998.

[33] A. S. Grimshaw and W. A. Wulf. Legion: Flexible Support for Wide-area Computing. In *Proceedings of the 7th workshop on ACM SIGOPS European Workshop*, Connemara, Ireland: ACM Press, September 1996, pp. 205–212.

[34] A. S. Grimshaw, W. A. Wulf, and Corporate. The Legion Vision of A Worldwide Virtual Computer. *Communications of the ACM*, 40(1): 39–45, January 1997.

[35] K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, vol. 1, Basic Concepts. EATCS Monographs in Theoretical Computer Science, Springer-Verlag, Berlin, Germany, 1992.

[36] E. Koepela. SETI@home: Massively Distributed Computing for SETI. *Computing in Science and Engineering*, 3(1): 78–83, January 2001.

[37] G. v. Laszewski, J. Gawor, C. J. Pena, and I. Foster. InfoGram: A Grid Service that Supports Both Information Queries and Job Execution. In *Proceedings of 11th IEEE International Symposium on High Performance Distributed Computing (HPDC'02)*, Edinburgh, Scotland: IEEE Computer Society Press, July 2002, pp. 333–342.

[38] J. Ledlie, J. Shneidman, M. Seltzer, and J. Huth. Scooped, Again. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS 2003)*, Lecture Notes in Computer Science (LNCS), vol. 2735, Berkeley, California: Springer-Verlag, Feburary 2003, pp. 129–138.

[39] V. Lesser. Cooperative Multiagent Systems: A Personal View of the State of the Art. *IEEE Transactions on Knowledge and Data Engineering*, 11(1): 133–142, January 1999.

[40] E. Marcus and H. Stern. *Blueprints for High Availability: Designing Resilient Distributed Systems*. John Wiley & Sons, New York, 1st ed., 2000.

[41] M. O. Neary, B. O. Christiansen, and P. Cappello. Javelin: Parallel Computing on the Internet. *Future Generation Computer Systems*, 15(5-6): 659–674, October 1999.

[42] M. Nowostawski, M. Purvis, and S. Cranefield. A Layered Approach for Modeling Agent Conversations. In *Proceedings of the 2nd International Workshop on Infrastructure for Agents, MAS, and Scalable MAS*, Montreal, Canada: May-June 2001, pp. 163–170.

[43] J. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[44] G. Pfister. *In Search of Clusters.* Prentice Hall, 2nd ed., 1997.

[45] D. Poutakidis, L. Padgham, and M. Winikoff. Debugging Multi-agent System Using Design Artefacts: The Case of Interaction Protocols. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multi Agent Systems*, Bologna, Italy: July 2002, pp. 960–967.

[46] M. Roehrig, W. Ziegler, and P. Wieder. *Grid Scheduling Dictionary of Terms and Keywords.* Global Grid Forum, Nov 2002. http://www.ggf.org/documents/GWD-I-E/GFD-I.011.pdf.

[47] J. Schopf. The Actions When SuperScheduling, Jul 2001. http://www.ggf.org/documents/GFD/GFD-I.4.pdf.

[48] L. Smarr and C. E. Catlett. Metacomputing. *Communications of the ACM*, 35(6): 44–52, June 1992.

[49] Sun Microsystems Inc. *Java Object Serialization Specification.* 2004. http://java.sun.com/j2se/1.5/pdf/serial-1.5.0.pdf.

[50] W3C. *Web Services.* 2002. http://www.w3.org/2002/ws/.

[51] W3C. *HTTP - Hypertext Transfer Protocol.* 2003. http://www.w3.org/Protocols/.

[52] W3C. *Simple Object Access Protocol.* 2003. http://www.w3.org/TR/soap/.

[53] W3C. *Web Services Architecture.* 2003. http://www.w3.org/TR/ws-arch/.

[54] M. Waldman, A. D. Rubin, and L. F. Cranor. Publius: A Robust, Tamper-Evident, Censorship-Resistant, Web Publishing System. In *Proceedings of the 9th USENIX Security Symposium*, Denver, Colorado: August 2000, pp. 59–72.

[55] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for Grid Services. In *Proceedings of the 12th International Symposium on High Performance Distributed Computing (HPDC-12)*, Seattle, Washington: IEEE Press, June 2003, pp. 48–57.