Patterns and Skeletons for Parallel and Distributed Computing

Fethi A. Rabhi
School of Information Systems
University of New South Wales, Sydney 2052 (Australia)
Email : f.rabhi@unsw.edu.au
and
Sergei Gorlatch
Technical University of Berlin
Sekr, FR 5-6, Franklinstr. 28/29
D-10587 Berlin (Germany)
Email: gorlatch@cs.tu-berlin.de

July 7, 2003

Chapter 1

Service Design Patterns for Computational Grids

Omer F. Rana and David W. Walker
Department of Computer Science
Cardiff University
Cardiff CF24 3XF, UK
{o.f.rana, david.w.walker}@cs.cf.ac.uk

1.1 Motivation and Introduction

Component integration has become an important area of research within the last decade – the intention being to combine a diverse set of custom and off-the-shelf components, from different vendors, to create applications. Components can implement simple functions, such as perform a matrix manipulation or perform a simple database query, or they can be more complex and wrap complete applications. Hence, the granularity of a component can vary from being coarse grained (when wrapping applications), to fine grained (when performing a simple database query). To facilitate the development of applications by combining components, an infrastructure is needed to enable components developed by different vendors to interoperate. The infrastructure must provide specialised services for managing, naming, locating and executing components, with standardisation efforts playing a crucial role in the process. Existing infrastructures such as DCOM from Microsoft, and CORBA/OMA from the Object Management Group (OMG), generally define this infrastructure in terms of (1) a component model, (2) containers which enable components to interact with services provided by the infrastructure, and (3) connectors, which enable components to interact with each other. The CORBA model for instance provides a platform independent language for defining component interfaces using an Interface Definition Language (IDL), and connectivity using an Object Request

Broker (ORB). Infrastructure services within the CORBA model include persistence support, lifecycle support, a component naming service, a trading service for identifying services undertaken by components, and others.

Within the context of component based software, another concept has become increasingly important, and referred to as the "Grid". The "Grid" or the "Computational Grid" (CG) signifies a component infrastructure that is based on the provision of infrastructure services which enable multi-vendor and multi-organisational services to be integrated (where a component may provide a single or a collection of services). A global Grid may therefore be a federation of many local Grids, and composed of components of varying granularity. Applications which utilise these services can range from large scale problems modelling some physical phenomenon, such as an oil spill, to the universal availability of customised and personalised services, accessible from wireless and mobile devices. Design patterns play an important role in constructing software services for such a Grid based environment, as many applications are likely to utilise common themes in their construction, and their subsequent interaction. We use the "service" abstraction as the common theme to identify how applications may be composed, shared, and managed over a CG – and define design patterns that become important in the context of such a service model. A central assumption of this paper is that applications on a CG will necessarily involve common services. However, it is important to note that a design pattern can only provide general guidelines to a possible design, and the system designer is responsible for exercising suitable judgement when applying them.

Based on a service perspective, a CG, foremost, requires an integration of services representing: (1) computation, (2) data, (3) hardware resources, and (4) people, in an effective manner to solve a particular problem of interest. The concept of a Grid as an infrastructure is important because it is concerned, above all, with large scale pooling of resources, whether computer cycles, data, sensors, or people, undertaken in a transparent and pervasive manner. The Grid is also being associated with the concept of creating a distributed computation environment out of a collection of files, databases, computers and scientific instruments. There are five major application classes that could benefit from CGs [13]:

- Distributed supercomputing, in which the Grid is used to aggregate computational resources to address very large problems that cannot be handled on a single system – such as distributed interactive simulations, simulation of physical processes (e.g. stellar dynamics, climate modelling etc)
- High throughput computing, in which the Grid is used to schedule large numbers of loosely-coupled or independent tasks, with the goal of putting unused processor cycles to work. Examples include the use of multiple distributed workstations to solve hard cryptographic or complex engineering design problems.
- On-demand computing, in which the Grid is used to meet short-term requirements for resources that cannot be cost-effectively or conveniently

located locally, facilitating resource sharing. The resources may be computational engines, but may also include software libraries, data repositories, specialised sensors and other devices. Unlike distributed supercomputing applications, these applications are often driven by cost-performance concerns rather than absolute application performance. The underlying infrastructure in these instances must deal with dynamic resource requests, and a potentially large population of users and resources. The challenges include resource location, scheduling, code management, application configuration, fault-tolerance, security and on-line charging for services.

- Data intensive computing, in which the Grid is used to synthesise new information from data maintained in geographically distributed repositories, digital libraries and databases. The synthesis process is intensive in terms of computational queries and communication between data sources. Challenges include the ability to handle the scheduling and configuration of complex, high-volume data flows through the network, at multiple levels of processing.

- Collaborative computing, in which the Grid acts as an enabler for enhancing human to human interactions, such as collaborative engineering design or "virtual worlds". In many cases, these applications involve providing the participants with shared access to data and computational resources, and share characteristics with the other application classes mentioned above.

We describe common themes that could be identified in applications requiring, and making use of, Grid technology. We look at existing Grid "enablers" such as Globus [2] and Legion [9], and identify common themes that emerge based on the tools that these systems provide. By identifying specialised services, or "roles", that components could play in the Grid, we aim to identify an infrastructure that is purely based on the aggregation, decomposition, discovery and execution of suitable services. We extend the standard component model with specialised components which have time varying behaviour, and which can interact with each other using a standard communication language. Each such component we call an "agent" – where an agent mediates access to one or more services. Although there is considerable debate in literature as to what constitutes an 'agent' – in this paper we consider an agent to be a program that can facilitate a user application service, and a computational service provider, interact more effectively.

Once identified, these common themes are encoded as "Service Patterns" that enable a designer to construct applications by combining patterns. A pattern can be specific to an application, or it may be more general, such as the "Adapter" and "Broker" service patterns discussed in detail in section 1.3. Each pattern can be coded in a particular programming language, or it may provide a more abstract way of designing applications making use of Grid services. We suggest that identifying services in this way provides a better way to design and build applications for the Grid, and complements the approach adopted in

Globus, where the Grid architecture aims to provide basic services, but does not prescribe particular programming models or higher level abstractions. The ideas presented in this chapter are based on concepts discussed within the DARPA CoABS project [19], but are aimed specifically at computational Grids for high performance distributed computing.

1.2 Resource and Service Management in Grids

We assume that a CG environment is composed of a number of heterogeneous resources, which may be owned and managed by different administrators. Each of these resources may offer one or more services. A service can be:

- A single application with a well defined API (which also includes wrapped applications from legacy codes). In this case, the software application is offering some service (which may vary in granularity from a simulation kernel to a molecular dynamics application, for instance). In order to run such an application, it may be necessary to configure the environment, and make available third party numeric libraries on the host where execution is requested. Furthermore, execution of the application may also involve the need to have dynamic link libraries available on the host platform, and for these libraries to be compatible with the compiler used to create the application executable. Seen as a service, these dependencies are hidden from the user needing access to the service – and the underlying environment which supports the service is required to handle these dependencies.
- A single application used to access services on other resources – controlled by a different administrator
- A collection of coupled applications, with well defined dependencies
- A software library, with a number of sub-services, which are all related in some functional sense. For instance, a graphics or a numerics library etc
- An interface to a third party software library
- A software library for managing access to a resource (this may include access rights and security, scheduling priorities, license checking software etc)

A distinguishing feature of a service is that it does not involve persistent software applications running on a particular server. A service, primarily, is executed only when a request for the service is received by a "service provider". The service provider publishes (advertises) the capability it can offer, but does not need to have a permanently running server to support the service. There is also a need for the environment which supports the service abstraction to satisfy all pre-conditions needed to execute a service. These pre-conditions can range

in complexity from the availability of particular link libraries and compilers, to the availability of specialised hardware platforms (such as a minimum number of free processors).

Various service description schemes are available today – from commercial and academic sources, such as the Service Oriented Architecture (combined with UDDI [21] and WSDL) [11] from IBM, Jini [17] from Sun Microsystems, various approaches based on XML (such as SOAP [3], ebXML etc), and DAML/DAML-S [7]. Each of these approaches provides a mechanism for encoding a data model to describe service properties, and identifying the location of the associated implementation of the service. These description schemes are also differentiated, in some cases, based on the domain they are used to encode – such as e-commerce or scientific computing. Some of these schemes, such as DAML, can be integrated with ontology definition languages such as OIL [10], to identify how services relate to each other. These approaches do not, however, specify how the data model (encoded in different schemes) is to be implemented or shared across services. It is unlikely that a single service representation scheme will be adopted, and many domain specific extensions are likely within existing schemes. Our approach therefore does not necessitate the use of a single representation scheme, allowing different brokers to co-exist, each of which could employ a different scheme for representing services.

1.2.1 Service Properties and Types

Generally, a given resource can offer one or more services, and also access one or more services on another resource. Similarly, a given service can access one or more services either on the local or a remote resource, and may be accessed by one or more services itself. Viewed in this way, a CG becomes, primarily, a collection of services – and application design then becomes a task of creating, managing and executing such services on the available resources. Each service within this model can support two kinds of interfaces:

- A *functional* interface which defines how the service is to be accessed and executed. Such interfaces have been defined previously in the context of the CORBA Trading service, for instance, and generally focus on data types (of calling and return parameters) and error handling [16]. A specification of an Object Service (in CORBA) usually consists of a set of interfaces (defined in the Interface Definition Language) and a description of the behaviour of the service. The semantics that specify a services' behaviour are, in general, expressed in terms of the OMG Object Model – based on objects, operations, types, and sub-typing. Hence, using this approach a service has an interface in IDL, which defines the types of called parameters, and return types for results, and support for exception handling for a particular called or return type or range.
- A *management* interface which defines parameters associated with service execution, licensing, cost etc. The management interface is used to differentiate between multiple resources offering a similar type of service,

and generally corresponds to the non-functional attributes of a service. A management interface may take into account the predicted performance of executing a service on a particular machine (and could also account for other performance attributes of a service – such as time to access a data store), and may be used as a selection criteria by a service broker (see below). Hence, the same service operating on different computational platforms will have the same functional interface, but different management interfaces.

Examples of a service can include: (1) a scientific or business application, such as a molecular dynamics application which takes as input the number of molecules, and uses the Lennard-Jones algorithm (for instance) to return forces and velocities of molecules after a particular time. In this case the application may be written in Fortran, and may require a multi-processor machine to execute. (2) A visualisation application which involves data migration to the platform where visualisation is to be performed, followed by subsequent rendering of the results. This service requires two sub-services to be invoked in succession – one to transfer the data, and another to render the results – and must ensure that the dependencies between these sub-services are maintained. A similar service would be the visualisation of data in real-time, where reservation of resources may be necessary. Examples include executing the results of a numeric solver on an Immersive platform. (3) A data access library which may be used to extract/record data from/into a structured database, a file system, or a repository collection. The service in this case would automatically determine the data access method based on the data source, and it is possible for multiple data sources to be used. Examples include extracting rainfall data for Wales over a given year, or stock trends for a particular market.

Based on the *functional* and *management* interfaces to a service, we can define a number of 'roles' that may be performed as part of a given service:

1. A *Service user* can request a service available at a local or remote host. The service user is responsible for initiating and terminating the service, and dealing with exceptions locally that are generated from the service.

2. A *Service provider* can generate service offers, and is responsible for establishing a service contract with a user. The service provider offers an interface for invoking a service, along with specification of parameters associated with managing the service.

3. A *Service broker* may be used to discover services based on one or more criteria. The broker acts as an intermediary between a service user and a service provider, and primarily supports service registration. The broker may also provide a 'matchmaking' service to help a user locate a service of interest. We use the term "broker" as a common intermediate service provider offering services of varying complexity – ranging from security, service decomposition and service scheduling. An important part of this

role is the provision of a discovery service interface – which enables a service provider to make itself known to a service user, and for a service user to identify its requirements. A broker may utilise the following properties to support discovery:

- Security: service security can vary from access rights (levels), to trust models that enable only a particular category of users to run the service.
- Cost: service cost can be associated with the management interface of a service, and correspond to computational time or access time, or access cost. Existing software tools such as Nimrod provide mechanisms (a 'virtual economy') to utilise such a parameter in selecting a service [6], [18]. Many resource providers at national centers also operate in this way, providing time on a computational resource or a percentage of a resource for a particular cost.
- Fairness: every service should be accessible from other services over a particular period of time. Service fairness issues arise when a particular service is prevented from being accessed (due to factors other than cost, security or performance issues). The issue of fairness is more complex when a service agent acts on the behalf of a service provider.
- Performance: the management interface of a service can be used to support service performance, and support a broker in discovering a service of interest. A service may also support additional levels of performance information, derived from analytical models of the service itself. For instance, if a service is to be run on a particular host, information about the host can be used to determine possible run times for a service with a given quantity of data. This information is also made available to a broker – and certain brokers may only select services which publish such information. It is up to the broker to decide how to use this information.

As can be seen from some of the criteria discussed above, a broker undertakes many complex but related roles. A general system is likely to have many brokers, each undertaking roles determined by application user demands, and the differences in representation schemes employed within the system. Results generated by all the brokers are however co-ordinated through a central authority – generally the user application service that initiated a request on the broker. Hence, there may be a broker to support service registration, a broker to support service discovery based on performance, service discovery based on cost etc.

4. A *Service adapter* is used to enable a service provider or user 'wrap' a given software library or application, and make this available as a single service. Many existing applications in Fortran/C, for instance, would need to be wrapped to enable them to be made available as a service. A service

adapter is also used to ensure that the activation policy associated with a service is respected by the service provider, and the service requester. A service adapter must also ensure that functional dependencies between the called software libraries which constitute the service are followed. Hence, a call to a given service may include the execution of an initialisation code, followed by the execution of a numeric solver, followed by the execution of code to record the results. The service adapter hides this level of detail from the service user. Service wrapping may be performed at the source code level (where this is available) or may involve adding an execution shell around an existing pre-compiled binary. Wrapping from source code is generally much more complex, as it involves a number of language specific issues, such as ensuring that type conversion does not modify the accuracy of the produced results. Wrapping from source code also requires the wrapping tool to have some knowledge of the structure of the application – and one that is likely to be a lengthy and error-prone process.

5. *Service aggregators/decomposers* are specialised brokers which can decompose or combine a service request to sub-requests to find better matches for service providers. It is possible for a popular service provider to be overloaded with requests, or for there to be no single service provider which can complete a given request. Service decomposers enable a given service request to be divided based on the available service providers, and for the results of these requests to then be combined before being returned to the user. Service decomposers can utilise domain specific ontologies to determine alternative service providers of interest, and enable these requests to be forwarded. The approaches adopted depend on the representation scheme used to encode the ontology, and the service definitions. The aggregator/decomposer must use the same representation scheme as the service discovery agents – and confirmed through an initial message exchange.

6. *Service discovery* is the most important part of the process, and is responsible for finding a match between a service request and a service provider. This match making can be supported through a number of possible criteria – as discussed previously – and it is possible for multiple service discovery agents to co-exit. A user may launch the same query to multiple such agents concurrently. A service discovery agent may utilise (1) a syntax match, (2) a context match, or (3) a semantic match. A syntax match would involve an exact textual comparison of the request, with the advertisement from the service providers. The other two approaches are based on the availability of a domain or problem specific ontology, which may be used by the discovery agent to resolve a given request. A "context" match would involve finding some similarity between the request and service providers that the service discovery agent is aware of. A context match is based on analysing other requests made by the same user previously in order to find a domain context for the current request. A domain context could enable a discovery agent to forward requests to

particular service providers. A "semantic" match would involve navigating the domain ontology, or relaxing domain constraints, to find suitable candidates that could be queried to find the required service. Additional details about these approaches can be found in [20].

7. A *Service optimiser* enables a group of service providers to work collectively to improve their cost, security, or performance. A service optimiser may be used to improve the behaviour of a resource cluster by sharing of common requests. A service optimiser may also be used to reserve a single service in advance, or make a reservation of a group of services, over a particular time frame. The service reservation mechanism is used to ensure that if a service aggregation/decomposition is to be performed, then all sub-services will be available.

8. A *Service execution* agent works with the broker and the user application to launch one or more tasks on the identified computational resources. This agent utilises existing Grid middleware, such as Globus and Legion services where available, to initiate the execution process. Hence, once suitable resources have been identified, the service execution agent can query the GIS (if Globus is involved) to verify the properties of the available resources, and how these are to be contacted. Grid services (such as GridFTP and GRAM in Globus) are used to transfer data and execution code of the application to these remote resources. The service execution agent primarily acts as an interface to third party resource execution systems – and does not directly undertake any scheduling on the remote resources itself.

9. A *Reputation Service* may be employed by a broker to rate a computational service. The rating function can be determined by the broker service, or it may be suggested by the application service needing resources. The rating service is used to provide each broker with a historical view of the available computational services, and enable a broker to filter service offers made by these resources. A broker undertaking service discovery may also query other brokers to determine similarities in their ratings of a given resource. Similarly, a computational resource may also wish to advertise its own ratings to a broker – which a broker may wish to ignore, or aggregate with its own results. To support a Reputation Service, each broker must be able to monitor the use of a resource by an application service, and be able to record these results locally. A broker utilising this service must also be able to modify its database if a computational service migrates, or modifies its properties.

10. A *Mobile Service* is not tied to a particular host, and may be migrated on demand. Service migration shares many ideas with object migration – in object migration based on a call-by-value semantics, for instance, the state of an object is sent to a remote location to create a new instance of the object. The new instance now has a separate identity, and does

not maintain links with the parent. As it is necessary for the receiving side to instantiate an instance, it must necessarily know something about the object's state and implementation – such as whether data members are `private` or `public` for instance. Service migration is defined at a coarser level of granularity to object migration, and consequently, may only involve the partial migration of state to the remote host. A service may be an aggregate of a number of other services, and consequently would require the migration of the complete dependency graph to the new host. By utilising a combination of call-by-value and reference semantics, a mobile service is able to create a new instance at the remote site of the service. Figure 1.1 illustrates this principle using a sequence diagram, where a service invocation from the user leads to a service being migrated to an environment more suited for its execution. Both the source and destination computational resources are registered as services with the Broker. At the end of the execution, results are returned directly to the user application. The "Migration Pattern" in section 1.3, contains more details about other participants involved in this process.

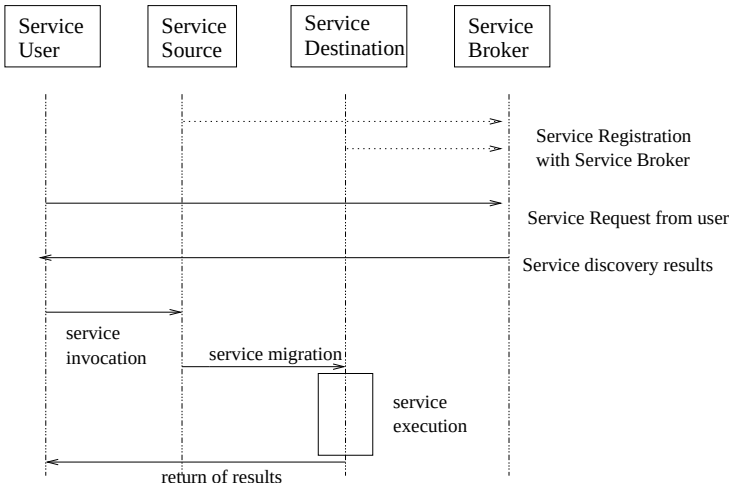

Figure 1.1: *Sequence diagram to demonstrate "Service Migration"*

1.2.2 Use cases

Based on these service roles, we identify use cases to demonstrate how these services can be deployed. Each use case highlights a particular role, and demonstrates how the 'service model' can play an effective role in utilising Grid based computational resources. It is implicit within the definition of a service that invoking it would incur an execution cost. This cost is difficult to quantify, as it depends on a number of implementation specific factors – such as caching

mechanisms, routing and referral delays, background workload etc – and we do not make an attempt to define it here.

Use of Broker Service

The first use case involves three types of participants: application users, service brokers, and computational resources. All three participants are represented as a service within our framework. The application is seen as a single service needing access to computational resources required to execute one or more tasks that constitute the application. To achieve this the application service makes a service request to one or more brokers, identifying its requirements in terms of performance, cost, and security needs. The broker service must now find computational services it is aware of, and which match the requirements identified by the application service. Each broker may have knowledge of different computational services (or groups of services), and may apply different heuristics to match the service requests with service advertisements it hold. A broker may make service requests to other intermediate service providers – such as a service discovery agent utilising more complex matching strategies, prior to returning results to the application service. Each broker may also be responding to service discovery requests from different application services concurrently. Each broker must therefore decide which application service should be given preference when suitable computational resources are discovered.

The broker plays an important role in identifying suitable computational services – but does not participate in subsequent scheduling and execution of the application. The matched services are returned to the user application which must now initiate execution on these remote resources. The user application may also decide to ignore resource offers made by a broker, and may rate the results of the returned matches to support subsequent requests.

Service decomposition

The second use case also involves the same participants as the first, but now the broker is unable to discover a suitable computational service. A broker can now invoke a service decomposer to divide the computational demands identified by the application user into sub-services. This division is supported by a domain ontology held by the broker and identified by the user application. A service decomposer (which must also understand this domain ontology) collaborates with the service broker to determine existing known services, and tries to find an aggregation of these services to fulfill the initial service request. The decomposition is therefore constrained by the service capabilities known by the broker, and their suitability in the context of the existing application domain. Depending on the capability requested by the user, a broker may utilise a more complex match making service to help support service decomposition.

Once a request has been decomposed, a service aggregator then synchronises and combines the results of each of the sub-services, before returning results to the broker. The decomposer can also cache service decomposition requests, and

support subsequent decomposition based on historical information. A decomposer may also utilise the Reputation service, performance or cost criteria if multiple decompositions are possible. Once suitable services have been identified the decomposer returns control to the broker, which in turn returns the list of matched resources to the user application. Initiation of the service on the available resources is subsequently undertaken by the user application, and the broker does not directly participates in the execution process.

Service execution

Once suitable computational services have been discovered, the application must now initiate service execution on these remote resources. The process is managed by the user application, but supported through an execution agent. The agent does not directly participate in the execution, but acts as an interface between a user application and the available computational services. This process may be mediated through Grid services (in systems such as Globus), or through vendor specific Web services where these are available. The execution agent is primarily responsible for monitoring the state of the execution process, and reports back errors to the user application. Scheduling of the application tasks is delegated to the third party resource management system, unless resource reservation has been undertaken previously.

A typical invocation would involve the execution agent checking the status of the discovered computational resources, transferring data to the resources (and code for execution where necessary), specifying the location of the results, and initiating the sequence to launch the application. The execution agent terminates after the application or task for which it is responsible for completes. Control is then returned back to the user application – which may invoke further execution agents.

1.3 Design Patterns to support Services

A distributed system in the context of this work is therefore composed of a number of roles, undertaken by participants which collaborate with each other based on the services they provide. Each role can support a number of different services, and the required service to fulfill a role can be determined at run time by intermediate service agents. The design process involves:

- Identifying the participants, and the role each participant is to undertake. Role, in this definition, specifies actions that are performed by a participant relative to other participants, and its own specialisation. An example role might be a "weather monitor", whereby a participant i is required to make another participant j aware of the prevailing weather conditions – see work by Kinny et al. [25].

- Identifying services to be supported within each role. In particular, identifying roles of intermediate agents – which do not directly represent a user

application or a computational service. The roles undertaken by intermediate agents are the most important parts of the design process, as they provide the essential service management functions within the system.

- Identify if a domain ontology is available, or if one should be created. As part of this process, the system designer should also identify what other Grid middleware is available.

- Identifying the interaction between participants, and conditions under which such interactions are to terminate. The condition for terminating interactions can be based on criteria ranging from the successful discovery of suitable computational services, the successful completion of a task or application, and the inability to find suitable computational services.

- Identify monitoring criteria for an executing application, and how this monitoring is to support the termination criteria.

We concentrate on "interaction" patterns between participants offering a particular role. The classification in [14] is used to describe our design patterns as primarily within the "Compound Jurisdiction" and as belonging to the characterization criteria. Hence, the specified design patterns describe how services relate to each other, interact with each other, and can be used to delegate and distributed responsibility. The results specified here build on existing work such as [5], [23]. Some familiarity with UML notation is assumed, and two particular approaches from UML are used to define patterns (a good introductory reference for UML is [1]). Sequence diagrams to show a particular instance of interaction between participants in a pattern, and a class diagram showing relationships between participants.

1.3.1 Broker Service Pattern

- **Intent:** Provides a service to support a user application discover suitable computational services. A Broker may utilise a number of other services to achieve this objective.

- **Motivation:** In the case of applications which do not need to be run in 'production mode' (i.e. industrial applications which are tuned for particular computational resources), a user may not be aware of suitable resources available in the CG. Additionally, a user may wish to explore possible alternative computational services which are available. It is also likely for resources to register and de-register dynamically, and do so without the direct knowledge of any given service user or provider within the CG. In this context, an application user should be able to determine computational services which are available at the time execution is required – and to reflect the state of the CG at this particular instance. It is also likely for the available computational services to differ based on their cost, performance characteristics and access rights.

The solution is to provide a generic Broker service which can be accessed by the application users, and the service providers to advertise their respective demands and capabilities. Associated with these are constraints to support match making. The Broker service may be specialised with additional capabilities based on service demands and system management policies.

- **Applicability:** The Broker Service pattern must be used when:
  - The user application service is not aware of suitable computational services
  - The environment within which the user application operates is very dynamic, and resources are likely to register and de-register often
  - The characteristics of a user application service can change often – dictated by changes in data sources or user profiles
  - There is a large number of widely differing resources – and each resource is likely to offer multiple services
  - Service access policies are likely to change over time or are dependent on resource workload
- **Participants and Structure:**

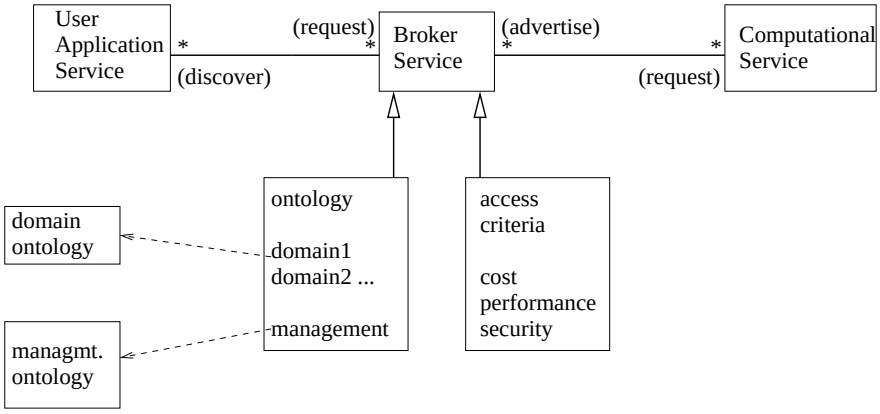

Figure 1.2: *Participants in the "Broker Service Pattern"*

Figure 1.2 illustrates the relationships between participants, and include:

- User Application Service
- Computational Service
- Broker Service
- Ontology Service (we consider this service to exist separate from a Broker. The ontology service may provide one or more "domain" ontologies, and a management ontology)

- **Collaboration:**

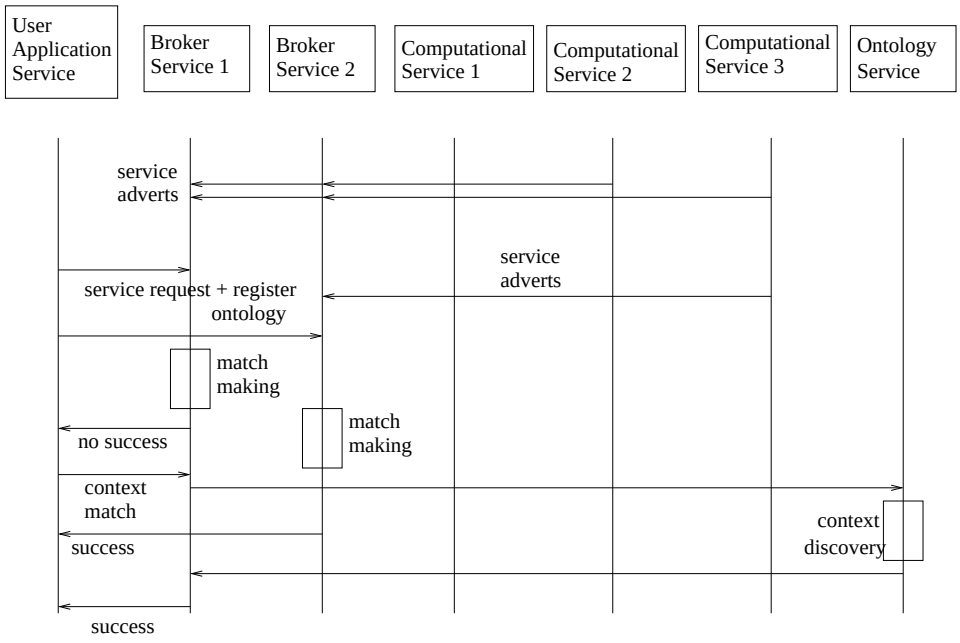

Figure 1.3: *A Sequence diagram showing one interaction between the Broker and other participants*

From a user perspective, an Application Service may utilise a single or multiple Broker Services to initially discover suitable computational resources. Subsequently, a Broker Service may interact with one or more Computational Services – based on the match criteria identified by the user. From a service provider perspective, one or more Broker services may be contacted to advertise capabilities. A sequence diagram indicating a possible service request is illustrated in figure 1.3, where various Computational Services first advertise their capability with a Broker Service, followed by the User Application Service making a request for a service. In this case, the Computational Services may also indicate one or more domain ontologies that they can understand. The User Application request also carries an ontology reference. Once one or more matches have been found, the Broker Service alerts both the Computational and User Application services of the result. As illustrated Computational Service 3 only advertises its capabilities to Broker 2 – and after Broker 1 has returned a "no success" (i.e. no match found) result to the User. Prior to getting a result back from Broker 2, a User Application Service may request a context match – and may ignore the results of this subsequently if Broker 2 returns before this context match has been achieved. A Broker service may be specialised in a number of ways, depending on the

complexity of discovery required by the application user or computational service. Therefore, a Broker Service may have a hierarchy, each layer providing more specialist service to the one above. Any particular Broker may inherit one or more layers of this specialisation, as indicated through the class hierarchy in figure 1.2.

- **Consequences:** The Broker pattern enables both computational and user services to discover each other on demand. Once a match has been found, the Broker must remove the advertisement for the computational service, and the request of the user service. The availability of an intermediary, which may be specialised in a number of ways, enables a user service to make more effective use of the available computational services. Similarly, computational services can improve their utilisation by monitoring requests by a broker and demands for a particular types of service requests (based on domain ontologies). A Broker service can also monitor its success in fulfilling a particular service request, or recording the number of matches made over a particular period. Based on this, a Service may specialise by adding additional complexity, or may wish to generalise by offering less. The choice is dictated by the Broker Service manager, and multiple instances of such specialisations may co-exist at any time in a CG.

1.3.2 Service Aggregator/Decomposer Pattern

- **Intent:** Provides a means to split a service request into sub-services, and to subsequently compose the results.

- **Motivation:** There are instances where a service request cannot be met based on the available computational services. This can arise because the suitable computational services are either executing other applications and would exceed their workload limits by accepting any new requests, or because there are no known computational services which can meet the required requests. When this happens, an Application User Service may decide to suspend itself and wait for a random time and re-issue the request. This is based on the premise that currently utilised computational services will be released after the wait time interval, or that new resources may have entered the CG which may be suitable. Alternatively, an Application User Service may issue an event subscription to the available computational service, which would notify it when it was released by another application. This assumes that suitable computational services are present, and they have access to an event service. If suitable computational services exist but are currently occupied, an Application User Service may also try to preempt the running application (task) on these resources – however, this is dependent on the system management policy of the computational service, and success for the general case cannot be guaranteed.

Most of the previous approaches rely on the existence of management expertise and policy with the service user or provider – to preempt a service, to suspend a service, or to register interest with a service provider. An alternative approach, which is less intrusive, is to utilise an Aggregator/Decomposer pattern to first determine which computational resources are available, and to determine mechanisms for dividing the service requests into sub-requests. This solution assumes that there is a shared ontology between the Application Service User and the Aggregator/Decomposer, and that the available request can be suitably split into what is known about the CG at the time.

- **Applicability:** The Aggregator/Decomposer pattern is applicable when:
  - A large number of computational services exist, but there are no requests which match these services exactly
  - No suitable computational services can be found, or if the computational services most suited to running an application are busy
  - Computational services which match a particular request criteria (such as cost, performance or security/access rights) cannot be met
  - It is possible to reserve computational resources
  - User application may wish to trade-off result criteria – such as precision vs. speed, precision vs. cost, or performance vs. cost.

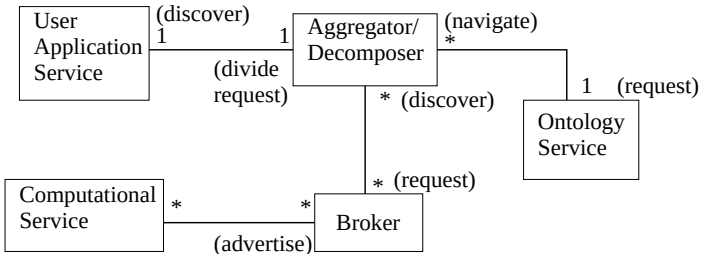

Figure 1.4: *Participants in the "Aggregator/Decomposer Pattern"*

- **Participants:** Figure 1.4 illustrates the relationships between the participants, and include:
  - Application User Service
  - Decomposition Service
  - Aggregation Service
  - Computational Service
  - Ontology Service

- **Collaboration:** Two kinds of collaborations between the Application User service and the Aggregator service are possible: (1) where the Application User service knows that it must interact with an Aggregation/Decomposition Service and can negotiate with it. In this instance, the Application User service actively participates in the decomposition process. (2) where the Application User service interacts passively with the Aggregator, and does not participate in choosing or supporting different decomposition strategies. In case (2), an Application User service may employ multiple Aggregators. Figure 1.5 illustrates a sequence diagram showing interactions between the participants for case (2).

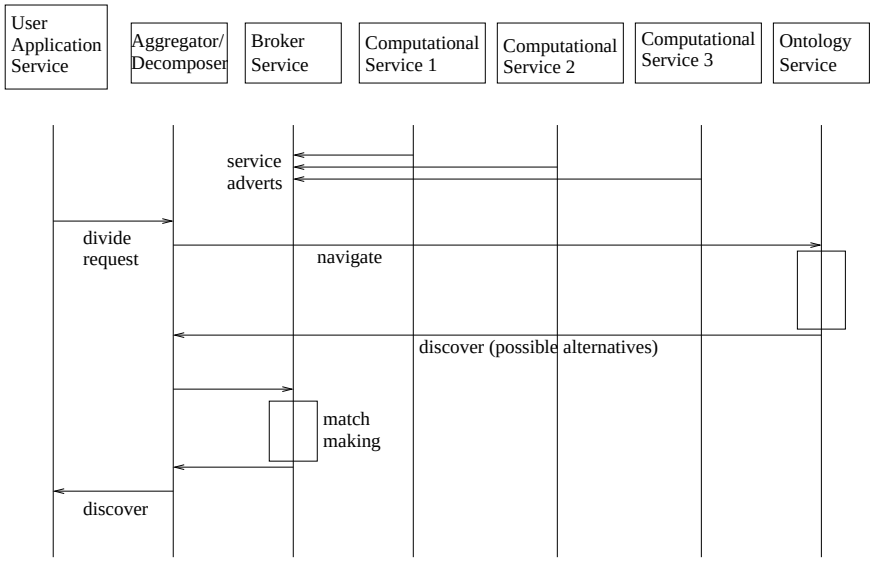

Figure 1.5: *A Sequence diagram showing interactions in the "Aggregator/Decomposer Pattern"*

- **Consequences:** The Aggregator/Decomposer pattern enables more effective utilisation of existing computational services. It also enables application services to consider alternative formulations of their requests, and to evaluate trade-offs between precision, access rights, speed of access, cost of access, execution performance etc. To be effective, this pattern assumes the availability of a shared ontology, and assumes that aggregation of results can be supported through a single service.

1.3.3 Service Adapter Patterns

- **Intent:** Attaches additional properties or behaviours to an existing application to enable it to be invoked as a service.

- **Motivation:** The CG is likely to be composed of a number of legacy applications and software libraries – which have not been written as a service. In our definition of a Computation Service, we consider both the software and the platform environment as a single entity (see section 1.2). Consequently, to wrap and modify existing applications to be offered as a "service", we provide the Service Adapter pattern. The particular adapter employed depends on whether an executable program is to modified (i.e. an executable binary), or whether the source code is also available.

  A service adapter pattern enables an application or software library to (1) identify how it may be invoked remotely – i.e. identify the types of requests, data types (both input and return) associated with these requests and types of exceptions generated, (2) identify the cost of accessing the service – or sub-services it supports, (3) identify if a performance model is available and how this model is to utilised and accessed, (4) provide a reference to a domain ontology for the data types and requests this application handles, (5) provide management information associated with the application: such as an owner ID, licensing issues associated with the application, and versioning references. In providing these additional properties, the execution and results generated from the original code should not change in any way.

- **Applicability:** This pattern is useful when additional behaviours need to be added to an existing application, to mediate access to this application. The application itself should not be modified, rather additional properties should be added so it can be discovered and invoked by another service. Another important criteria is to prevent re-writing of the original application – an error prone process – and to facilitate re-use of existing libraries.

- **Participants:** Figure 1.6 illustrates the participants involved, and include:

  – Service Adapter
  – Application or Software library
  – Ontology Service

- **Collaboration:** The Adapter pattern provides interaction between a legacy application and an ontology service. Requests to a service are mediated by the Adapter, and it also provides an interface to invoke the application. The application interacts with other resources and users on the CG via the Adapter, and therefore has to delegate execution authority to it.

- **Consequences:** An Adapter pattern enables applications to be made available as services, particular for domains where a well defined ontology exists. The approach is flexible and enables the adapter to monitor the

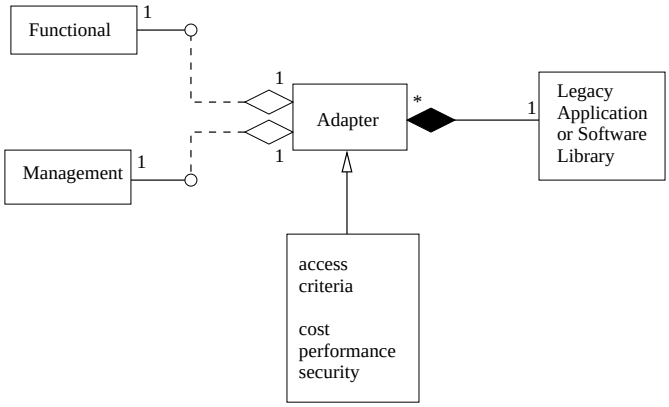

Figure 1.6: *Participants in the "Service Adapter Pattern"*

execution of the application, and offers advice on performance properties to a service discovery agent, based on an analytic model maintained by the Adapter.

1.3.4 Service Execution Pattern

- **Intent:** Supports the launching of an application (or tasks) on the available computational resources, and to monitor their subsequent execution.
- **Motivation:** Once suitable computational services have been discovered, it is now necessary for the User Application to initiate and co-ordinate execution of these services. Generally, the complexity and diversity of the underlying resources is such that it is unlikely for a single User Application service to manage such execution. Consequently, it delegates responsibility for achieving this to an execution agent, which acts as an intermediary to third party resource management systems where these are available. Where a tight coupling between the User Application and the underlying resource is required, execution may be directly managed by the User Application.

  The execution agent utilises a Service Execution Pattern to check if the required computational services are on-line, whether data or code needs to be transferred to these services, whether access rights are available, whether licenses are available, and whether execution can start. This execution agent must also decide where results of the computation are to be stored, and how exceptions generated from the computational service are to be dealt with. Generally, the execution agent forwards any exceptions to the User Application service. The user execution agent terminates once the execution has completed (successfully or not) and results have been recorded. Scheduling dependencies between tasks of an application are also handled by the execution agent – based on information provided to it

by the User Application service. Depending on the discovered computational services, an execution agent may forward all or part of an application to a given resource – reserving resources where all computation cannot be achieved due to task dependencies. It is possible for an execution agent to determine that the execution of a complete application is not possible – as computational services cannot be reserved, or because computational services are likely to change during the duration of the execution. Consequently, it alerts the User Application service of this, and may request a re-discovery to complete the execution, or determine that execution is not possible.

- **Applicability:** This pattern is used when:
  - The User Application service wishes to initiate the execution on the discovered computational services
  - The complexity of the discovered computational services renders direct execution control via the User Application service unlikely
  - Service management systems (such as schedulers and dispatchers) are available, and these can be invoked from other services
  - Data or code management is required prior to service execution
  - The state of a computational service needs to be confirmed prior to execution
- **Participants:** The participants include:
  - User Application Service
  - Execution Agent
  - Service Management System
  - Computational Service

Figure 1.7 illustrates the relationship between participants.

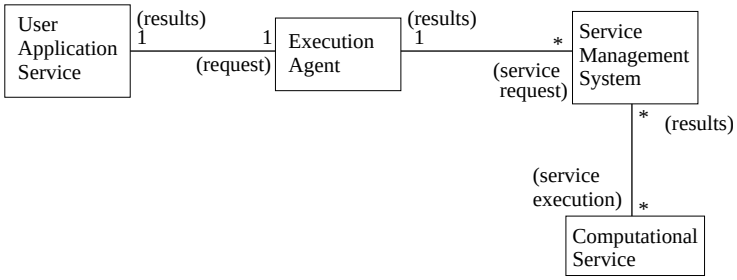

Figure 1.7: *Participants in the "Service Execution Pattern"*

- **Collaboration:**

  Interaction between participants is illustrated in figure 1.8. After the execution request has been sent by the User Application Service, the Execution agent must attempt to reserve the indicated services obtained through the discovery process. In this case, the Service Management System provides the interface through which the Execution agent attempts to achieve this. If services can be reserved, execution can commence – achieved by transferring data (and code) to the resource hosting the service, and invoking the service remotely. Results or errors are returned to the agent.

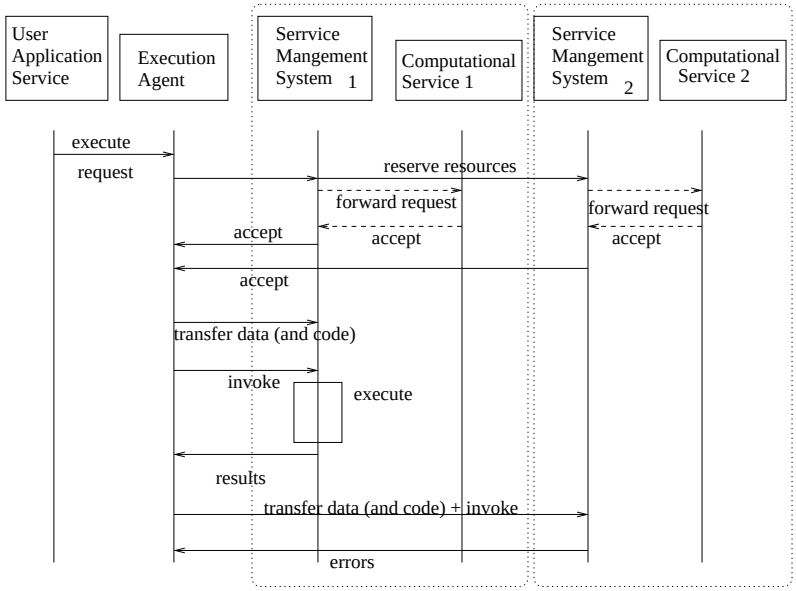

Figure 1.8: *A Sequence diagram for the "Service Execution Pattern"*

- **Consequences:** This is perhaps one of the most complex patterns in this catalogue, as it necessitates an interaction with low level service management systems. The advantage of the pattern is to enable an intermediate agent which co-ordinates execution, and monitor its progress. Using such an agent, the user application service does not need to be aware of the configuration of computational services, or how these services are to be accessed. The execution agent must ensure that task dependencies are met, and that resources will be available to execute subsequent tasks in the future. The execution agent therefore acts also acts as a task mapping service, although the services to which tasks should be mapped will have been pre-determined. All activities necessary for initiating execution must be performed by the agent (such as transferring data), and appropriate checks on resource availability (such as disk space) need to be made by

the agent through the appropriate service management interface. As the execution agent cannot directly negotiate with computational services, it also acts as a passive error propagation channel to the user application service.

1.3.5 Service Migration Pattern

- **Intent:** Enables a service to be moved to another computational platform more suitable for its execution.
- **Motivation:** There are several reasons why service migration may be necessary in the CG. A service may need to be migrated to a platform which contains more resources (such as memory, processors, or a third party library). A service may be migrated to support load balancing – especially if a large number of service requests are being made. Another use would be to merge the results of two services, which could be migrated to a single host, executed on the host, followed by the removal of the mobile service. Various approaches to supporting code mobility are provide in Aridor and Lange [4] – along with reasons why code mobility is a useful and important paradigm. Service migration can apply to both User Application and Computational services – and the same mechanisms can be employed to support both. A migrating user may decide to recreate the same operation environment s(he) had before – one way to achieve this is by migrating the execution environment to the remote computational resource. Similarly, a computational service may migrate to a new host to offer a better cost or performance to user application services. By allowing service migration, many new application behaviours can be supported.
- **Applicability:** The Service Migration Pattern is useful when:
  - Services are not tied to fixed platforms, and can be migrated on-demand
  - A host can accept external services, and will provide computational resources for these services
  - A service can checkpoint its state prior to transfer, and recreate the state after transfer
  - A service can alert a location service about its current status and availability
- **Participants:** The participants involved are:
  - Computational Service
  - Source and Destination platforms
  - Location Service
  - Ontology Service

- **Collaboration:** The computational service requesting the migration must contact the destination to ensure that it will accept executable code. The requesting service also publishes its domain and management ontology to the destination service. It is now up to the destination service to verify that it can host the service, and whether it is willing to accept the credentials and representation scheme employed by the source service. The destination service can utilise the Ontology Service to support this process. The Ontology service can confirm or deny similarities between the ontology supported by the source and the destination, and the results of this analysis can be verified by the destination service. If successful, the destination service accepts the transfer request – and the source can send the executable code, state of the execution achieved on the source platform, and any data necessary to support execution. Once the transfer has completed, the source service also updates the position of the migrated service within a Location Service – which must maintain a reference to the current location of the service. This process may be repeated a number of times – as illustrated in figure 1.9, until results are returned back to the "home" service (the starting point of migration). It is left to the Service Management System at the final destination to either dispose of the executable code, or return it back to the home service.

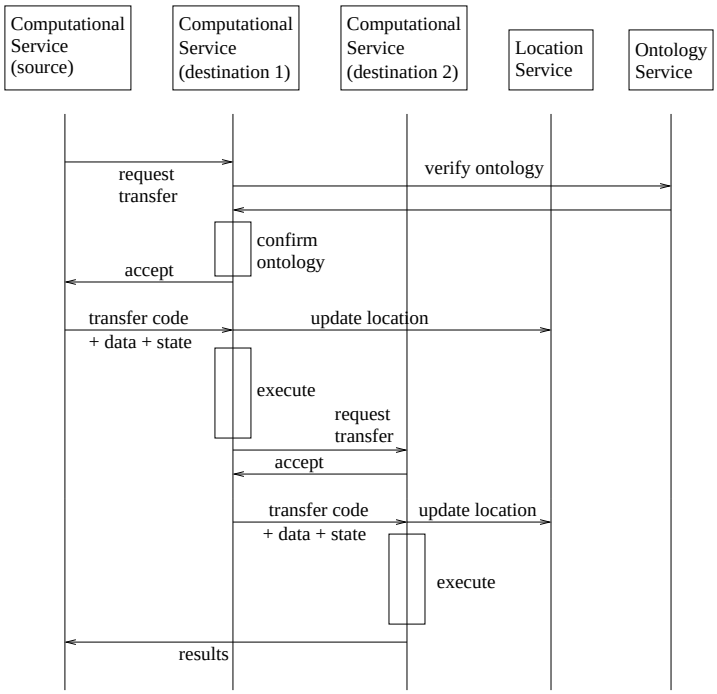

Figure 1.9: *Participants in the "Service Execution Pattern"*

- **Consequences:** The migration pattern is useful to support service behaviours where services can discover better operating environments, and migrate to support better cost/performance trade-offs. It is also useful to support user mobility, especially where the operating environment for a user needs to be maintained. Not all services can be migrated, as not all services will be capable of recording their state and restoring it after the transfer. For instance, the choice of a programming language and environment are important in determining whether a service can be migrated – as some programming languages will not allow execution state to be captured. More importantly, the service needs to be implemented in a manner which enables the service to execute efficiently and correctly on multiple types of platforms – or for pre-compiled versions of the service to co-exist for different platforms.

1.3.6 Implementation

The Jini-Grid [22] project and work undertaken within the Jini working group of the Global Grid Forum [15] is currently implementing these patterns and services, using Globus as the Service Management System. The Globus system comprises of the Grid Resource Information Service (GRIS) and the Grid Index Information Service (GIIS). GRIS acts as a 'white pages directory' to support service discovery, and also supports a uniform mechanism for querying resource properties. Such resources generally include computational resources (although the approach is general and can also apply to data resources) and must have a distinguished name. Information about the properties of each resource is maintained as an object, such as CPU speed, memory, cache etc. The GRIS server must exist on each host machine, and keeps information corresponding to only that host. Hence, issuing the command `grid-info-host-search -L -b ‘‘o=Grid’’ -h inca.cf.ac.uk ‘‘(objectclass=*)’’` to search for information on `inca` host machine with the starting point at "o=Grid" of its Directory Information Tree (as defined by the LDAP [24] protocol used in GIIS) will produce a list of objects, with each object corresponding to one property of host `inca`. Generally, the GRIS uses a well-known port number 2135 to listen for user requests. Subsequently, the GIIS or 'yellow pages directory' offers a means of integrating arbitrary GRIS services, to support resource queries which include parameter ranges or constraints on resource properties. The GIIS may be searched by Grid applications, and may maintain information gathered from multiple GRIS servers. There is only one GIIS server running at each site and it can be set up on any available port number. In addition, a GRIS server is automatically created during Globus installation, whereas a GIIS server must be specified during 'globus-setup'. A 'Referral Service' links multiple GRIS and GIIS servers together into a single LDAP namespace, with no content being present on Referral servers. We extend the resource centric approach in GRIS to enable description of software services available on these resources, in addition to properties of these resources – using Jini as a means to store, retrieve, and manage access to these services. A client query would consist of

inquiring system properties about a resource, and software libraries managed on it. The query is first serviced by GRIS, and subsequently by our Jini Service Manager. We are currently implementing service definitions, based on a system management ontology developed as part of the FIPA platform [8], and encoded in XML. For instance, a client request for a matrix solver could consist of `numeric.solvers.matrix`, or a call for all solvers as `numeric.solvers.*`, in which case a list of suitable solvers is returned from which a client may select. Such a `lookup` service aids the broker in first discovering services of interest, and then properties of resources on which these services exist – using GRIS and GIIS. Our current focus has been on implementing an effective service discovery mechanism, as it plays an important role in all design pattern. The discovery service will also record data obtained from execution runs of a service – and which will subsequently aid the discovery process.

Recent interest in integrating Grid computing with Web services – as part of the "Open Grid Services Architecture" (OGSA) [12] – is an imporant step in utilising the service paradigm for creating, naming, and indexGlobus!OGSA discovering transient Grid service instances. Implementation in this case makes use of the Web Service Description Language (WSDL) for defininig service interfaces, with a Grid service being defined as a Web service that conforms to a set of conventions (interfaces and behaviours – semantics). Interaction between services is supported by the SOAP protocol. In OGSA a WSDL based Grid services comprises of: Types (based on XML Schemas for data types that will be used within the service), Message (a collection to be communicated between one or more parties), Part (a names portion of a message), an Operation (a names end-point that consumes a message as input, and optionally returns a message as output), a PortType (a named collection of operations), Bindings (a description of a concrete network protocol and message format), and Service (binding specific information about the specific destination of operations). The generic Web services model is also extended to provide attributes such as `compatibilityAssertions`, `serviceType`, for instance, to enable service comparison, service substitution etc. These approaches do not yet support the composition of services to construct applications, based on service roles and properties. Although some work has been undertaken in the Web Services Flow Language, for combining services based on a data flow approach – this is still at a very preliminary stage.

1.4 Conclusion

A service oriented approach is adopted, to describe design patterns in the context of CG. The patterns primarily cover interactions between participants within a Grid environment where participants are restricted to undertake predefined roles. A CG is likely to be composed of a number of services – some of which are derived from existing legacy applications. A distinguishing feature of a CG is the existence of site specific management and activation policies controlled by each service owner, and the lack of a central co-ordinator. Users

of such services may also desire different trade-offs with respect to cost, performance and access rights. Service patterns which address these requirements are outlined. Commercial efforts towards Web services are likely to play an important role in how such computational services are defined, and it is likely for there to exist a computational economy influencing access costs to such services. Our work can utilise such efforts, and can also also allow multiple representation schemes to co-exist.

The motivation behind these design patterns is to enable effective management of services – and to enable service users and providers to share intermediate services.

Bibliography

[1] Sinan Si Alhir. *UML in a Nutshell.* O'Reilly and Associates, 1998.

[2] ANL. The Globus Project, http://www.globus.org/, 2001.

[3] Apache.org. Apache Simple Object Access Protocol (SOAP). http://xml.apache.org/soap/, 2001.

[4] Y. Aridor and D. B. Lange. Agent Design Patterns: Elements of Agent Application Design. *Proceedings of* 2^{nd} *Int. Conf. on Autonomous Agents, Minneapolis/St. Paul*, 1998.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture -- A System of Patterns*. John Wiley and Sons, 1996.

[6] R. Buyya, J. Giddy, and D. Abramson. An Evaluation of Economy-based Resource Trading and Scheduling on Computational Power Grids for Parameter Sweep Applications. *Proceedings of the* 2^{nd} *workshop on Active Middleware Services (AMS 2001), in conjunction with HPDC, Pittsburgh, US*, 2001.

[7] G. Denker, J. R. Hobbs, D. Martin, S. Narayanan, , and R. Waldinger. Accessing Information and Services on the DAML-Enabled Web. *Whitepaper, SRI International, Menlo Park, California,* http://www.daml.org/, 2001.

[8] Emorphia. FIPA -- Open Source. http://fipa-os.sourceforge.net/, 2001.

[9] Andrew Grimshaw et al. Legion: A World Wide Virtual Computer, http://legion.virginia.edu/, 2001.

[10] D. Fensel, I. Horrocks, F. Van Harmelen, S. Decker, M. Erdmann, and M. Klein. OIL in a nutshell. *Proceedings of the European Knowledge Acquisition Conference (EKAW-2000), R. Dieng et al. (eds.), LNAI, Springer-Verlag*, 2000.

[11] D. Ferguson. IBM Web Services: Technical and Product Architecture and Roadmap, 2001.

[12] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. *Technical Report, Argonne National Laboratory, US. Available at:* http://www.globus.org/research/papers/ogsa.pdf, 2002.

[13] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, Inc, 1999.

[14] Erich Gamma, Richard Helm, Ralph Johnsonand, and John Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. *Proceedings of ECOOP*, 1993.

[15] Global Grid Forum (GGF). The Jini Working Group. http://www.mcs.anl.gov/gridforum/jini/, 2001.

[16] Object Management Group. CORBA 3.0 Trading Object Service Specification (document 00-06-27), http://www.omg.com, 2000.

[17] Sun Microsystems. Jini, http://www.jini.org/, 2001.

[18] S. Newhouse, A. Mayer, N. Furmento, S. McGough, J. Stanton, , and J. Darlington. Laying the Foundations for the Semantic Grid. *Proceedings of AISB Workshop on AI and Grid Computing*, 2002.

[19] DARPA Project. CoABS: Control of Agent Based Systems, http://coabs.globalinfotek.com/, 1999.

[20] O.F. Rana, D. Bunford-Jones, D. W. Walker, M. Addis, M.Surridge, and K. Hawick. Resource Discovery for Dynamic Clusters in Computational Grids. *Proceedings of Heterogeneous Computing Workshop, at IPPS/SPDP, San Francisco, California*, 2001.

[21] IBM Research. Universal Description, Discovery and Integration of Business for the Web. http://www.uddi.org/, 2001.

[22] D. Saelee and O. F. Rana. Implementing Services in a Computational Grid with Jini and Globus. *First EuroGlobus Workshop, Leece, Italy. See Web site at:* http://www.euroglobus.unile.it/, 2001.

[23] D. Schmidt. A Family of Design Patterns for Applications-Level Gateways. *Theory and Practice of Object Systems*, 1(2):15--30, 1996.

[24] M. Wahl, T. Howes, , and S. Kille. Lightweight Directory Access Protocol (LDAP) v3.0 -- RFC 2251, 1997.

[25] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Proceedings of the third International conference on Autonomous Agents and Multi-Agent Systems*, pages 285--312, 2000.