Dependable Grid Services

Stuart Anderson[†*] Yin Chen[†] Glen Dobson[‡*] Stephen Hall[‡*]
Conrad Hughes[†*] Yong Li[†] Sheng Qu[†] Ed Smith[†] Ian Sommerville[‡*]
Ma Tiejun[†]

August 18, 2003

Abstract

The provision of dependable computer systems by deploying diverse, redundant components in order to mask or provide recovery from component failures has mostly been restricted to systems with very high criticality. In this paper we present an architecture and prototype implementation of an approach to providing such redundancy at low cost in service-based infrastructures. In particular we consider services that are supplied by composing a number of component services and consider how service discovery, automatic monitoring and failure detection have the potential to create composed services that are more dependable than might be possible using a straightforward approach. The work is still in its early stages and so far no evaluation of the approach has been carried out.

1 Introduction

Dependability is a complex attribute which recognises that simpler attributes such as availability and reliability cannot be considered in isolation. Laprie [1] defines dependability to be *that property of a computer system such that reliance can justifiably be placed on the service it delivers.*

A common feature of dependable systems is that they use *diverse, redundant* resources to reduce the probability of common mode failures and to allow the system to continue in operation in the presence of component failure. Critical components of dependable systems may therefore be implemented as a composition of components, some of which are intended to deliver the same services but are implemented in diverse ways in an attempt to ensure their failures are not highly correlated.

Formal analysis of fault-tolerant software systems has demonstrated that diversity can contribute significantly to an improvement in the reliability and availability of the overall system [2]. However, a major inhibiting factor in deploying diversity in systems is cost. Systems deploying significant diversity, for example the avionics system of the A320 commercial transport aircraft, which uses diverse multi-channel hardware and software, are very expensive to develop. The use of diverse, redundant components has therefore been limited to systems where the consequences of system failure are severe, and consequently they have high availability requirements.

With the advent of the Web Services [3] definition there has been growing interest in service-based architectures where functionality is packaged as a standardised service with implementation details hidden from the users of these services. One particular example of this is the work on Grid Services as a means to deliver computational and data resources to the e-Science community. However, services provided remotely over computer networks are subject to frequent failure (tardy response, no answer, etc.) for diverse reasons, ranging from resource starvation and network instability through to implementation or specification error. This means that service users must currently incorporate code in their applications to detect and cope with service failures.

The automatic composition of services (now possible with various Web Service and Grid standards) may reasonably be expected to have a multiplicative effect on these failures since breakdown of a single operation in a composition could jeopardise the entire procedure. Furthermore, different error recovery strategies in-

[†]School of Informatics, University of Edinburgh
[‡]Department of Computing Science, Lancaster University
[*]These authors acknowledge the support of EPSRC award no. GR/S04642/01, *Dependable, Service-centric Grid Computing*

corporated in services may interact in unexpected ways with each other and with explicit error recovery mechanisms in the application. We therefore believe that there is a need for a standardised mechanism that allows applications to continue in operation (perhaps in a degraded way) in the presence of individual service failure.

In this paper we outline our approach to improving the dependability of Grid Services that are provided by composing other Grid Services. At first sight deploying diversity to improve the dependability of Grid Services may appear to be an excessively costly approach for all but the most critical services but two features of Grid Services make this approach much more attractive:

1. For some (commonly used) services it is envisaged that the pool of different implementations could be quite large. Although we cannot be sure that these implementations are diverse, it is reasonable to conjecture that it is likely that there will be significant differences between them and consequently they will exhibit diverse failure behaviour.

2. The Grid architecture provides a service discovery mechanism that facilitates identification of a range of different providers for a given service.

These features mean that we may be able to draw on large collections of different service implementations without incurring huge cost. This preliminary paper outlines our approach to exploiting this diversity and provides a description of an early implementation.

Our goal is to use our early implementation to investigate the feasibility of dependable Grid Service provision and to explore a range of approaches to some of the issues we identify in this paper. The main issues we address in this work are:

1. We envisage that each service will have dependability data associated with it. We are considering how best to represent this information. Many monitoring systems collect individual measures of reliability or availability – but we know these are not independent and, from the point of view of a service user, the tradeoffs between different parameters are important.

2. Composed services will rely on service discovery to find available candidate services to compose, along with their associated dependability data. This assumes a mechanism to gather dependability data on a particular service. We are considering the design of monitoring systems including the extent to which we can trust monitoring data and how dependability data is updated to reflect past experience.

3. It should be possible to calculate the dependability of a composed service range on the basis of dependability data for the component services in the composition. This will require us to develop appropriate means to estimate the dependability of the composed service on the basis of the dependability data for the component services.

4. The composed service will respond to failure data for the component services as it delivers the composed service. This will involve investigating real-time failure monitoring of the component Grid Services.

5. Users of services need a way to select services by specifying the dependability characteristics they require. There are a range of existing QoS specification languages but we believe they have some weaknesses. In particular, they are poor at expressing tradeoffs between different parameters.

The remainder of the paper considers our proposed architecture, its operation, a brief description of our prototype implementation and an outline of future directions.

2 Architecture

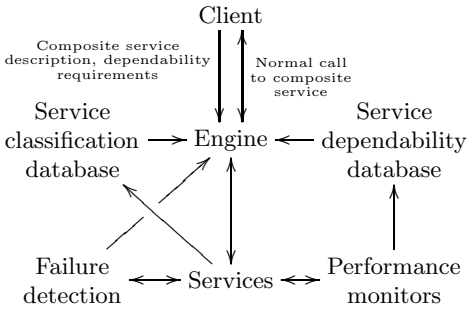


Figure 1: Architecture

Our proposed architecture is outlined in figure 1; its components break down as follows (more detailed information may be found in later sections):

2.1 Engine

The engine performs two tasks — planning and execution.

During the planning stage, a client requests execution of a service (expressed as a composition) to a specified level of dependability within constraints (usually cost or time). The engine identifies which specific services might be used to instantiate this composition by querying the service classification database, and for each identified service it retrieves relevant dependability metrics from the service dependability database. The net properties of possible composition instantiations are then evaluated, and if none satisfy requirements, modifications to the basic composition are considered (see section 3). When a satisfactory combination of service choice and composition modification has been found, the planning stage is complete. It is conceivable that some negotiation between client and engine may take place here — for example a choice of several candidate compositions with different, but satisfactory, dependability and constraint tradeoffs may be offered to the client.

The execution phase comprises the exhibition of the requested Grid Service interface (representing the composed service) to the client, and execution of the chosen augmented composition on receipt of service calls against the interface. This process may involve setting up sensors to identify failures of the component services, conditional (in event of failure) and/or parallel execution of services, lifetime management of elements of the composition, etc.

2.2 Information Services

Before it is possible to create an instantiated composition of services we require mechanisms to discover and filter candidate services. Within the overall architecture of the system these mechanisms are called the service classification and dependability databases. The service classification database provides a means of discovering services that are syntactically and semantically compatible with the current task. Although our architecture does not prescribe how service discovery takes place, it does require a service classification scheme.

In the context of a dependable architecture it is not acceptable to have services that provide no information with regards to their dependability or quality of service. Therefore the service dependability database must provide quality of service metrics for all services within the system. Our ultimate intention is to populate the service dependability database with metrics from monitors within the system, allowing changes within the Grid to be reflected in the makeup of the composition. The parameters associated with quality must be modelled within the database and evaluated by the engine.

2.3 Sensors

This architecture requires two varieties of sensor: failure detectors and performance monitors.

Failure detectors are intended to provide immediate feedback to the engine (in order that it might take corrective or compensatory action) in the event of failure of an operating service: in some cases, services may be trusted to quickly and accurately report their own problems (running out of disc space, for example), while in others (network outage, host subversion), an external observer is needed. Rapid response usually comes at a cost: heartbeats, keepalives, pings, all consume super-linearly increasing amounts of network bandwidth as the mesh of sensors grows and required response time decreases. Schemes which can manage this cost/benefit balance are needed.

Performance monitors are concerned with the long-term measurement of service dependability. They may do so in order to place records in the dependability database, or in order to verify compliance with a service level agreement [4]. They may form part of a service, or be operated by independent third parties. Performance monitors have less of a real-time requirement than failure detectors, but may log and transmit large quantities of statistical information.

3 Example

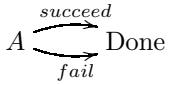
The client has a composed service description which describes how to amalgamate a hotel booking service and a flight booking service into a single travel booking service (this is a common example in composition language papers). The client also has a cost limit and a desire that the composed service succeed more often than their past interaction with airline websites has.

The travel booking service description might be a commonly available composition document taken from the web, one which will work with any booking services which conform to

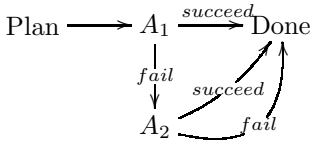
some commercial standard. The QoS requirement is vague, however; we need to address the issue of how human users may clearly express themselves to a system which deals in numbers and probability distributions. This work will come later in the project — for the time being, we interpret the user as desiring a 99% success rate.

Lack of space constrains us to limit a detailed description of the service but one of the authors[1] has carried out a detailed study of the provision of this kind of service. In order to compress the presentation we provide schematic illustrations of the kind of building blocks we use to build dependable composed services.

Given the trivial service "composition" of a single service of type A as follows...

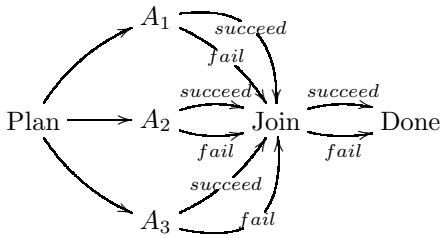


... where "Done" indicates communication of results — be they success, failure, or something more nuanced — to the client, the following might be a simple augmented instantiation:



Here, during the planning phase two *specific* services of type A (A_1 and A_2) are chosen with A_1 as primary and A_2 as backup. If A_1 fails, A_2 is called; if A_2 also fails, then failure is admitted. Obviously this process could be continued forever, but the planning phase would establish in advance how many fallbacks were necessary to satisfy the client's requirements within cost and time constraints.

Another option follows:



Here A_1, A_2, and A_3 are all run in parallel; "Join" could be chosen to mean waiting for the first successful reply and then finishing (favouring availability and (to some extent) speed), or it could mean a vote, wherein at least two identical results were required before a successful outcome was recorded — favouring reliability instead. Different clients would get different solutions according to their requirements, and the engine will plan these solutions using estimation based on dependability data for the available services.

[1] Sheng Qu, as part of his Masters dissertation in Edinburgh.

4 Implementation

Currently we have prototype implementations of many of the components of the proposed system. These include:

1. Real-time sensors utilising notification in the GT3 toolkit to transfer information to direct recovery from service failures.

2. Performance monitors based on the JXTA peer-to-peer infrastructure that gather long-term data on the dependability of individual service instances.

3. Testing frameworks that integrate with the GT3 toolkit to provide the ability to simulate a range of service failures in a predictable manner.

4. A rejuvenation service based on the GT3 toolkit aimed at providing a component that implements service rejuvenation

5. Extensions of BPEL4WS to consider fault tolerance, this provides explicit fault-tolerant construction in BPEL.

Information services play a key role in providing the information that guides particular decisions taken in providing a composed service. Access to information services plays a key role in coordinating the individual components. In this section we outline some of the considerations in deciding on a prototype infrastructure for the project.

4.1 Information Services

The architecture section of this paper describes the service classification and dependability databases used to achieve service discovery. Here we identify an infrastructure to meet the requirements laid out in the architecture. This infrastructure is being utilised to build initial prototypes of our system. Our choice of infrastructure will evolve as we learn from experience of the prototypes.

4.1.1 Service Data

The underlying architecture of Grid Services, namely the Open Grid Services Architecture (OGSA), provides mechanisms that can be utilised to achieve service discovery. Service Data is one such mechanism that can be used to encapsulate information about a service. Information that is critical in this context is service classification and quality of service. The rest of this section details the mechanisms and schema for describing service related information and the processes that utilise this information.

Service Data is a collection of information that is associated with a Grid Service. Service Data is encapsulated as Service Data Elements (SDEs) that represent XML based containers for structured information [5]. SDEs are linked to every Grid Service instance and are accessible via the Grid Service and Notification interfaces. Service discovery is facilitated by information held in service data. When selecting services we use matching upon the service data information against our own requirements. The latest implementation of the OGSA framework, Globus toolkit version 3.0, has service data at the heart of all service discovery, forming the Monitoring and Discovery Service version 3 (MDS-3) which supercedes the LDAP based MDS-2.

Service data is represented in XML, in keeping with all elements of OGSA, but there are no restrictions on the structure of the XML. The most elementary way of imposing a structure on the Service Data is to apply an XML Schema. The schema that is used depends upon the requirements of Grid; in our case we want to identify services that:

a) have the correct interface and functionality required by the service we aim to provide

   and

b) have dependability attributes that ensure the composed service achieves its promised level of dependability.

4.1.2 Service Discovery

Service discovery relies upon the knowledge of what sort of service you are invoking. A partial picture of this is painted by the WSDL description of the service interface. However, the service interface does not describe the functionality of the service. An example is two services providing the "add interface", an add operation with two integer inputs and an integer output; the add operation is invoked on both services by passing the integer value 1 to both the inputs; one service returns the integer value 2 and the other the value 11; both services have operated correctly upon the same interface but have different functionality. The example demonstrates the requirements for a classification of services based upon functionality and interface type.

Much research has taken place in the area of classification ontologies led by the notion of the Semantic Web. Current proposals for ontologies centre on frame based languages such as OWL [6]. These languages are very similar in structure to object oriented programming languages with the notion of class inheritance at their heart. DAML-S [7] is a leading example of a frame based ontology language that is designed principally for representing Web Services within the semantic web. It is not the intention of this project to perform research into service classification or to implement a complex ontology language. Instead in our prototype we have opted for a simple classification schema within the service data element container. Every service belongs to a particular named class. The named class is then used as the criterion for discovery. This classification mechanism in addition to the WSDL description constitutes the service functionality database.

4.1.3 Dependability Attributes

We require services be endowed with information that informs potential clients of the quality of service with regards to a series of important criteria, namely dependability. Like the service classification, these QoS parameters are expressed within the context of service data elements. The originator for this QoS information can be the services themselves, but we envisage that the QoS metrics are more likely to come from the clients or a trusted third party. Many QoS specification languages have been developed, primarily in the field of distributed multimedia applications. Jin and Nahrstedt [8] and Aurrecoechea et al [9] provide a good contrast of these languages including the resource specification language (RSL) created by Globus for toolkit versions 1.0 and 2.0. QoS is expressed at three layers: user, resource and application. User layer QoS represents a series of parameters that are set by the user, usually through a graphical user interface, implying that the parameters must be simple. Resource layer QoS represents parameters relating to the physical hardware environment such as CPU allocation.

Service oriented architectures represent a virtualization of resources where applications are composed of lower level services. Application level QoS is in keeping with the service oriented paradigm because application developers express their requirements for a a given level of QoS from a service thus controlling the qualities of that service [8].

Our criteria for selecting a specification language were primarily simplicity and expressiveness. In addition a script and control based languages such as SafeTcl and fuzzy-control were dismissed being too complex for our requirements. QoS Modeling Language (QML) developed by HP Laboratories [10] is an application layer, parameter based, QoS specification language [8]. QML is a generic language that provides specification refinement (similar to object oriented inheritance) and simple contract types such as reliability and performance. The simplicity of QML does not mask its ability to express complex QoS specifications, for example using percentiles, variance and frequency [10]. We have selected QML as a way of specifying QoS requirements within the service discovery phase of this project. Unfortunately, QML despite its name is not an XML based language, thus we have taken a subset of the QML specification [10] and implemented it in XML Schema.

QML defines categories named contract types for conformance in given dimensions. Each dimension corresponds to a domain of elements that are either defined manually or are built in numeric types. A contract is a set of QoS restrictions upon the dimension domains defined within the contract type. Dimension domains have ordering applied to their elements to allow certain elements to be stronger and therefore conform to weaker elements. For example if a latency of ten seconds is required, then a latency of nine or less is acceptable, in this case nine is said to be a stronger element in the dimension domain latency than ten. Contracts can be specialised using the refine relationship to form more stringent contracts, however, all sub contracts must still conform to the original contract type. Profiles are used to link QML contracts to interface (or portType) definitions.

4.1.4 Service Matching

QML is used both to specify QoS requirements and to express QoS information for specific service instances. QML specifications are included in the service data for our Grid Service instances. These definitions will occur within the SDE sections of each service description. An aggregation of the QoS service data elements constitutes the service dependability database.

Service matching is notionally a single process, however, the process is split into two phases. In the initial phase, the services are actually discovered using the service classification as criterion. Services are retrieved in the form of Grid Service handles with additional service data elements containing the QoS specifications. A second phase exists that uses the QoS information for each of the services to identify the service that is most appropriate for the present task. The job of identifying the service is done by the main engine because of the dynamic nature of the engine means that the QoS requirements for a given task are only identified at the last minute. Different services may be "plugged" into a workflow in combination to identify end-to-end QoS given the QoS information for each service. Early implementations of the engine use exhaustive algorithms to test the best combinations to give the best QoS. Future developments will concentrate on optimising this service matching process. The second phase of the matching process is implicitly tied to the engine. However, the service classification matching can be delegated to an index service.

5 Testing

In order to help verify correctness of operation of the engine, we have developed a framework which allows us to (largely) transparently "wrap" arbitrary Grid Services and control their apparent behaviour by modifying, suppressing, or injecting messages at the service interface level. These impositions are scripted, so we can isolate constituents of a composition, cause them to appear to fail in various ways, and then observe the system's reaction.

This facility is particularly important because our composed services include fault-tolerance capabilites and in developing our approach we need to test both the real-time monitoring of services we use to detect failure and communicate it to the composed service and to test the fault tolerant mechanisms included in the composed services.

Once we have established that our monitoring and fault tolerance is robust we will require this framework in our experiments with the system. The ability to script failure scenar-

ios across a distributed network of Grid Services will help us to measure our ability to deliver improved level of service in context of networks with "known" (in this case, manufactured) dependability metrics.

6 Flexible framework

Our work to date has concentrated on creating a working prototype of the architecture described earlier. At the time of writing that prototype is close to working. However, it is unlikely that the prototype will provide an ideal solution to the problem. In the end our approach is to provide a flexible framework for the provision of dependable Grid Services.

In the current prototype we are considering the following (non-exhaustive) list of issues that will help direct our next steps in the project:

1. The two service databases may in fact be a single entity (MDS perhaps).
2. Service monitoring may be undertaken by third parties, or services may be trusted to advertise their own metadata honestly.
3. All monitoring may in fact be supplanted by service level agreements.
4. The finer nuances of failure detection may be service-specific so the framework must admit specialised detection mechanisms.

Clear separation of design components will admit all of this flexibility. The greatest such issue arises in the core engine: what language describes the composition, and how is that then executed? Several candidates (none open source) exist, and we hope to achieve an implementation which can be "ported" to a new composition engine with a minimum of effort.

7 Conclusions

Our project is still in its early stages. We took an early decision that our work should be based on the GT3 (OGSA) framework since that is likely to be widely used by the e-Science community. Our experience over the past six months has been that the GT3 platform is still quite unstable and that has led to considerable problems in developing components that depend on detailed issues of the implementation of the toolkit. However, we believe that the service-orientation of GT3 provides a good basis for the provision of dependable services using the techniques outlined in this paper.

This paper describes our provisional architecture and implementation, unfortunately at the time of writing we have yet to evaluate our work. This will be reported in later papers.

References

[1] J. C. Laprie. *Dependability — Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-tolerant Systems*. Springer-Verlag, 1992. IFIP WG 10.4.

[2] B. Littlewood. The impact of diversity upon common mode failures. *Reliability Engineering and System Safety*, 51:101–113, 1996.

[3] K. Gottschalk. Introduction to web services architecture. `http://www.research.ibm.com/journal/sj/412/gottschalk.html`.

[4] A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreement s for web services. *IBM Research Report*, May 2002.

[5] I. Foster, C. Kesselman, J. Nick, and S. Tueke. The physiology of the grid: An open grid services architecture for dist ributed systems integration. *Open Grid Service Infrastructure WG, Global Grid Forum*, 2002.

[6] D. L. McGuiness and F. van Harmelen. OWL web ontology language overview. 2003.

[7] S. Mcllraith and D. Martin. Bringing semantics to web services. *IEEE Intelligent Systems*, 18(1):90–93, 2003.

[8] J. Jin and K. Nahrstedt. Classification and comparison of QoS specification languages for dist ributed multimedia applications. 2002.

[9] C. Aurrocoechea, A. Campbell, and L. Hauw. A survey of QoS architectures. *Multimedia Systems*, 6(3):138–157, 1998.

[10] S. Frolund and J. Koisten. QML: A language for quality of service specification. 1998.