Dependable Grid Services: DIGS Initial Implementation Notes

Stuart Anderson[†*] Glen Dobson[‡*] Stephen Hall[‡*] Conrad Hughes[†*]
Ian Sommerville[‡*]

September 28, 2004

Abstract

We describe the preliminary implementation of a "proxy" which can combine groups of identical grid services in various configurations (such as fallbacks and parallel execution), giving the appearance of a single, "better" service; we aim to build on this to provide (according to user specification) improved dependability for arbitrary applications composed of individual grid services.

1 Introduction

The goals of this project centre around *dependability* of *composite grid services*: given an application (itself possibly a service) built on top of a number of grid services, we wish to manage this application's ability to satisfy various *performance requirements*, such as availability, time to complete, etc. — in fact, once a user describes their preferences, acceptable tradeoffs, limits, etc. among measurable service characteristics, we intend to achieve *automatic* reconfiguration of the application to satisfy these expressed requirements.

[†]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*

1.1 Plurality, diversity, choice

In a grid scenario, it is expected that many commonly used services will be offered by multiple providers, so an individual service requirement would be satisfiable by any of a plurality of (ideally diverse and independent) service implementations, each with differing characteristics — be they availability, cost, accuracy, etc. Some of these implementations will satisfy performance requirements better than others, and using several of them together (in series as backups, in parallel with expectation and verification of identical results, or in other configurations) may match requirements even better if the choice of services and manner of their combination are appropriate.

1.2 Metrics, metadata, and requirements

We limit our definition of dependability to that which can be improved using the above-outlined structural and choice-based techniques; these will only influence certain characteristics of a system, and may do so in unexpected or negative ways: having a long line of if-it-fails-then-try-this-alternative backups may improve your chances of success but will almost certainly not improve performance; using backup services which all run off the same local electrical grid (breaching our "independence" desideratum) will not significantly improve availability if electrical outages are the major source of downtime. A language (or languages) to describe these characteristics, how they are affected by our structural modifications, possibly to record performance *distributions*, user tradeoff options (for example the user may be prepared to accept an increase in frequency of failures if overall downtime is reduced) and shared service dependencies is needed. The project will explore such metadata issues as it progresses.

2 Scenario

2.1 Motivation

In order to demonstrate the concepts behind our system we decided to implement an example grid. What we sought for this purpose was a scenario closely related to some real world application that involved a small number of services taking part in a well-defined and easily demonstrable conversation. The advantages of using a real world scenario were seen to be that it would highlight some of the practical advantages and disadvantages of our system, suggest directions for future development and improvement and act as a useful aid in exposition.

The field of medical imaging was chosen as particularly suitable because it involved potentially very large data sets (multiple high resolution images) and nontrivial computations (e.g. image analysis, comparison and rendering). All of these aspects were desirable in order to test the effects of our system on dependability and because they are key application areas for grid technology.

2.2 Scenario details

The specific scenario chosen was inspired by a number of grid-based Computer Aided Diagnosis (CAD) initiatives. CAD uses computer analysis of medical images in order to aid in their interpretation and inform the decision-making process. One such grid-oriented project which uses this technique is the National Digital Mammography Archive (NDMA) [1]. This project aims to demonstrate the feasibility of a nationwide system to collect, manage, store, retrieve and index mammographic images. Of particular interest to us was its emphasis on the suitability of the service model for CAD provision and the suggestion that such a service might be available from multiple vendors. This fits well with the kind of diversely available service for which we envision our system to be most suitable.

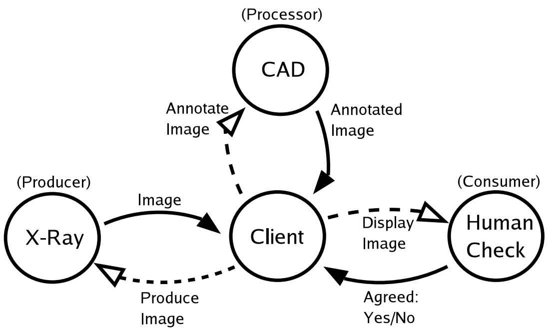

Figure 1: Overview of Computer Aided Diagnosis scenario

Our greatly simplified demonstration (Figure 1) should not be taken as an attempt to implement anything of the complexity of the NDMA, but references this real world problem as a means of clarifying the application of our system. In fact our scenario uses just three types of service, following the producer-processor-consumer pattern. In this pattern the services form a chain: the consumer depends upon the processor for processed data and the processor in turn depends upon the producer as its data source.

The producer in our scenario takes the form of an X-Ray service. This nominally represents any medical image source. In our implementation images are not generated, but are simply picked randomly from a set of source images. The service which processes these images is the Computer Aided Diagnosis (CAD) service itself, which annotates the image as a 'pass' or a 'fail'. In reality the diagnosis would involve more complex markup of the image in order to highlight suspect areas — but since we had no access to real data and no diagnostic expertise available, our implementation simply performs an arbitrary deterministic computation to decide whether to 'pass' or 'fail' the image. The final service in the chain (the consumer) allows the diagnosis to be checked by a human. In our implementation this service displays the annotated image and asks the human expert whether they agree with the computer diagnosis or not.

The client in figure 1 initiates and coordinates the conversation required to produce, annotate and display (for human confirmation) the image and the associated diagnosis. In a final system this 'client' could be viewed simply as the composition of the three constituent services. The make-up of this composition will be decided by our system engine in order to best meet the dependability requirements placed upon it.

3 Proxy

The engine serves two distinct purposes: to form compositions of services that offer calculable levels of dependability, and to intercept client/service communications transparently imposing dependability primarily in the form of fault tolerance over a set of services. The engine must not affect the operation of either the client or service. Additionally, the engine must be as generic as is practical in order to work with any grid service.

Logically, we can impose the dependability composition at one of three locations:

1. Inside the client.
2. Inside the service hosting container.
3. At the messaging level.

We review the statement "the engine must not affect the operation of either the client or server". Clearly this would be difficult with options 1 and 2, where the middleware upon which they run would need to be changed to support our infrastructure. The problem with option 3 is the difficulty of keeping everything generic. However, we can make the simplifying assumption of the ubiquity of the SOAP XML messaging protocol. The term "service" in no way implies that only SOAP should be used, but vendors of web service-based middleware have built their systems upon SOAP. This is particularly true for version three of the Globus Toolkit, which is based on the open source Apache Axis service engine. We therefore adopt option 3: intercepting SOAP messages, enabling dependability over a set of services transparently.

To intercept messages at the network level we use web-based proxy servers as a template. Proxy servers are typically used for load balancing and the reduction of network traffic by caching messages. We extend the notion of a proxy such that it can execute a model that exhibits fault tolerant behaviour. In this model the client sends a request to the proxy as if it was a real service. The message is passed through the fault tolerance model and routed to one or more real services. Responses are routed back through the proxy where further processing such as correlation or voting can take place before the result is sent back to the client.

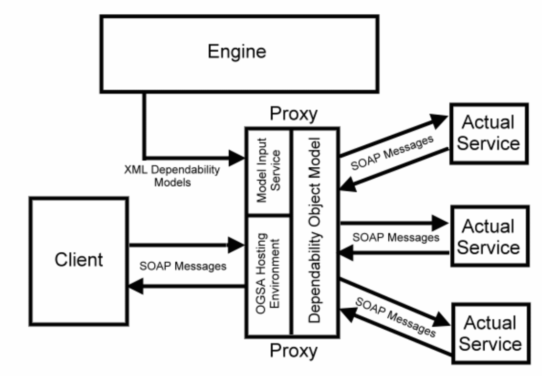

Figure 2: Engine and proxy

Though conceptually the engine is one unit, in reality the system is made up of two parts, the engine and the proxy. The engine generates models that are passed to the proxy. Messages passed to the proxy are processed by an object model generated from the information received from the engine. Models passed from the engine to the proxy are in XML form. The object model generated inside the proxy is passed an internal representation of a SOAP message and processes it according to the model rules. The object model has the ability to inspect the SOAP message for certain information, such as a grid service handle when dealing with "create service" requests. Figure 2 shows the new conceptual layout of the engine.

4 Dependability model

The dependability model is the heart of the proxy; it dictates what to do upon receipt of a given message. The proxy can contain any number of models at any given instant. The original intention was to have a one-to-one mapping between models and services. However, by mapping each model to a set of services, a set of interactions can be coordinated through the proxy. The proxy can still work on a model per service basis if required. Figure 3 shows the parts of the model.

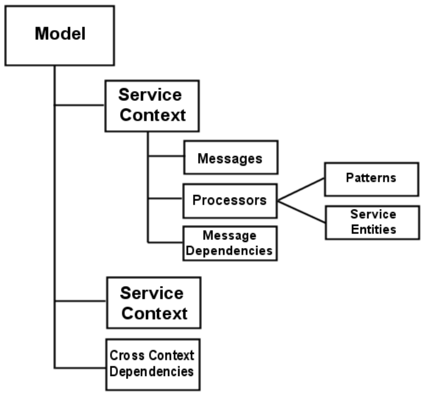

Figure 3: Structure of dependability model

4.1 Message targeting

The proxy receives a message represented as a Java messaging SOAP object, and a message context that contains metainformation. The message context has two critical keys. Firstly the target service: this is essentially the end part of the grid service handle (GSH) once all the container, and proxy references have been taken out. For example the GSH

`http://localhost:8080/ogsa/services/digs/proxy/test/TestService`

renders the target service

test/TestService.

This key indicates the model to be used. The second key is the SOAP Action URI: this is passed in the header of the transport protocol such as HTTP. It is a unique name that identifies the message. A SOAP action URI can be calculated from a WSDL description of a particular operation. It is made up of the operation name-space, a hash symbol '`#`' and the operation name; for example

`http://www.gridforum.org/namespaces/2003/03/OGSI#createService`

is the SOAP action URI for a factory "create service" message. Although the SOAP action URI is technically optional in OGSA, its use is sufficiently widespread in generated OGSI code to warrant its inclusion in the proxy. The SOAP action URI is used as a key to identify the message object to be used within the dependability model.

A message when passed to the proxy from the underlying OGSA container is firstly wrapped in a proxy message object. This object contains the message context and the message. The proxy message also contains the mechanisms required to query the actual SOAP message. A target service key is obtained from the message context and used as a lookup for the correct model object for this message. The model object exposes a process method to which the proxy message is passed. When the model object has finished its work the process method returns with the proxy message populated with the SOAP message response (if there is one). The process method may also throw an exception if the model was unable to process the message for some reason. The model is responsible for checking for model-level dependencies (a model-level dependency is a list of messages that can apply to any service context, with the implication that all messages should be called in top to bottom order. If an earlier message in the dependency list has not been called before the message being processed, then a failure should be raised — this feature does not cache messages: it is only an order check for the messages being processed by the model). Figure 4 shows the operation of the proxy using the target service key to identify the model and context.

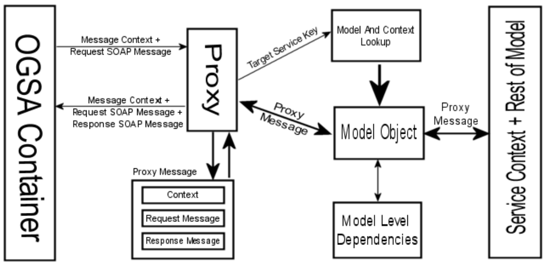

Figure 4: Proxy operation from OGSA container to service context

4.2 Proxy message processing

Once the model dependency checks have been completed, the proxy message object is passed to the process method of the appropriate service context. The service context object is identified by the target service key. Essentially, the service context can be viewed as a representation of a service within the proxy, although it may be mapped to any number of actual services. All the actual services must provide the same WSDL interface. The main action of the service context object process method is to find the appropriate message object. The message object is identified by the SOAP action URI key which is obtained from the message context via the proxy message. Each message object exposes a process method to which the proxy message is passed. Figure 5 demonstrates the operation of the message object and its relationship to the message-level dependencies. The references between a message and dependencies form a many-to-many relationship.

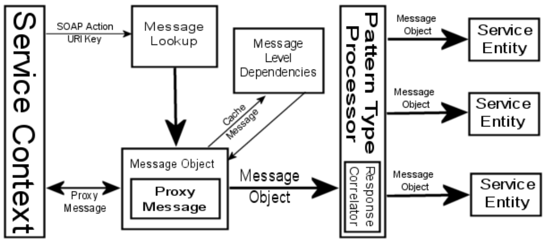

Figure 5: Proxy operation from service context to service entity

The message object finally wraps the proxy message object allowing message dependency objects to be passed to objects further down the processing chain. A processor object is passed a message object containing a proxy message. As shown in figure 3 the processor object can be of one of two basic types: pattern, or service entity. A service entity is an object that represents an actual service that SOAP messages are to be redirected to. A pattern is a means of grouping processors together in some way to increase the dependability. Because patterns and service entities are types of processors it is possible to nest patterns within each other forming a tree-like call structure. However, the call structure cannot be cyclic and must terminate in service entities, thus preventing infinite loops within the proxy.

4.3 Dependability patterns

Two simple examples of dependable patterns are "order" and "concurrent". The "order" pattern comprises two or more processors in an ordered list. Its operation is to pass a message to the first processor in the list; if that succeeds then the pattern terminates and returns any response. If a failure is detected in the first processor, then the message is passed to a second processor; if that fails then the message is passed to a third processor. The pattern will keep trying processors in order until one succeeds or it runs out of processors at which point it will raise an exception. The "order" pattern demonstrates recovery block behaviour. The "concurrent" pattern comprises an unordered list of processors: its operation is to pass the message to all the processors concurrently. Threads are used internally to achieve the concurrency required. A "concurrent" pattern demonstrates redundancy behaviour. A problem with the "concurrent" pattern is what to do with the many possible responses. We use a correlation object that accepts several responses and reduces them down to one valid response. A good example of correlation is voting, where for example if three processors return identical answers and a fourth processor returns a different answer, the anomalous response is ignored. Perhaps the simplest correlation is "first valid response", where once a valid response is received from any processor the pattern returns and all subsequent responses from other processors are ignored.

The model is designed to be extended to allow more complex and service specific patterns to be used. An obvious example of a useful pattern would be checkpoints, where state can be transferred from one processor to another in the event of a failure. Examples of service-specific patterns would include hot, warm and cold standby services in grid based multimedia applications.

4.4 Service entities

The endpoint of any pattern is the service entity. Eventually the message object will be delivered to the service entity object. Every service entity contains the GSH of a real service. Its basic operation is to take the proxy message from the message object, give it the GSH and then request the proxy message to redirect its SOAP message via a service call. However, the service entity must check for message level dependencies. Message-level dependencies, like the model-level dependencies, are lists of messages, but these are related to a particular message. By having these we are saying each service must complete these messages in this order. This allows lazy invocation of services. A prime example of this is in the order pattern for recovery block fault tolerance, where service one is created but fails on the invocation so we want to invoke service two but it has to be created first. By having the dependencies and caching of messages we can check that service two has not yet been created and send it a create service message before sending the invocation. The message-level dependencies have the ability to cache each message sent through the proxy in order to support such lazy invocation. Figure 6 shows the relationships between the service entity and the message-level dependencies.

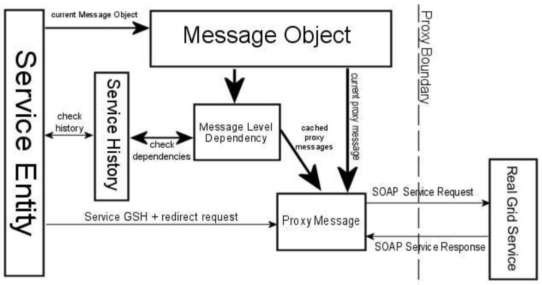

Figure 6: Proxy operation from service entity to actual service

Each service entity object holds its own history of invocations indicating whether they were successful or not. It is this history that is mapped against the message-level dependencies to decide whether a set of cached messages need to be sent first. This history can also be used to mark certain services as bad because of their earlier failures on messages that are in the dependency lists for the current message. Currently, once a service has been marked as bad it cannot be recovered for the lifetime of the model instance. Proposed extensions to the proxy include the ability to recover a service.

4.5 OGSA special cases

How does the proxy relate to the OGSA-specific services? The proxy has the ability to wrap any type of SOAP web service and therefore any OGSA service. However, the "create service" operation is treated as a special case. Service entities representing grid service instances have the problem of having a generated GSH or endpoint. In OGSA the GSH of the instance is an extension of the GSH of the factory that created the instance (usually by adding slash and the instance name). There is little point in hard coding the instance name into the service entity. Instead, they are randomly generated as unique identifiers. A "create service" invocation can contain a suggested instance name in the header of the SOAP message; this is used to create a target service identifier for the service context object within the proxy. If no instance name is in the SOAP message then again a unique one is generated. The response message for the "create service" is rewritten to include the target service identifier so that the client has a GSH that corresponds to a notional service instance in the proxy. The notional service instance within the proxy always corresponds to one and only one service context in the model.

The "create service" operation in OGSA is provided by a factory service. Within the proxy, the "create service" operation is associated directly with the instance. This removes the requirement for an extra association between the factory and instance service contexts and entities. The following occurs when a "create service" message is received:

1. The message is matched against the instance service context using the factory GSH.
2. The instance name is retrieved from the SOAP message header. If there is no name in the header then one is generated and held in the service context. We will call this the context instance name.
3. A new context GSH is recorded against the service context and parent model made up of the factory GSH and the context instance name.
4. The message is passed to a service entity.
5. The service entity modifies the instance name in the header of the SOAP message to a generated unique name.
6. The message is passed to a real factory service.
7. The response from the real service is queried for the returning locator GSH which is then held in the service entity; this locator GSH is replaced by the context GSH and the response sent back to the client.

To enable the querying of SOAP messages we have built a simple XPATH query engine that operates in conjunction with the proxy message object. This engine is used, for example, in the retrieval and replacement of GSHs and instances in the "create service" request and response messages. The limitation of this engine is that it cannot handle namespaces so it just looks for element names and attributes within the SOAP message.

5 Future work

We have successfully demonstrated the feasibility of improving dependability in composite grid applications by transparently proxying service usage through structural logic which may use more than one actual service to improve critical characteristics of the required task. Future work includes the following:

We will choose, extend, or develop a service metadata language capable of expressing such ideas as whether and how a service supports checkpointing, whether its outputs have restricted scope (only valid for a short time or a particular recipient for example) and whether they can be directly compared with each other (for voting or averaging). Exposure of hidden dependencies might also be handled with this language.

Similarly, we will choose, extend, or develop a language which describes performance metrics suitably for this application — accurate quantification of "edge" characteristics such as failure likelihoods is not trivial — and which facilitates both representation of this information to human users (for decision making) *and* computation over these metrics: when several services with known characteristics are combined in some way, we will need to know what the characteristics of the aggregate will be. Possession of incomplete information when performing these computations is a likely scenario.

Successful management of streaming workflows will be essential — our example in this paper deliberately has an easy-to-handle "star" topology; real systems will usually be much more complex, involving dataflows outwith any central application's control. Such topologies will complicate our job since many dependability strategies are predicated on the ability to lift processing entities out of the workflow (in the event of failure) and replace them without disrupting an otherwise live system. Data transport systems such as NaradaBrokering [2] may offer the solution, but are not useful without widespread support by service providers.

Greater understanding of the structure of any workflow being executed will enable more accurate planning. A language to describe certain aspects of workflows is needed for this. Mechanisms for "rewinding" workflow execution in the client application would give us greater flexibility when reacting to failure, so we are particularly interested in an "interactive" workflow language capable of both representation and control.

Our preliminary research proves that our core idea works; as the above developments unfold we shall develop a much more powerful dependable planning and execution engine which will be useful in a broad range of applications.

References

[1] M. Schnall. National digital mammography archive website. `http://nscp01.physics.upenn.edu/ndma/`.

[2] Shrideep Pallickara et al. The NaradaBrokering project. `http://grids.ucs.indiana.edu/ptliupages/projects/narada/`, 2001-2004.