# A Container-Based Approach to Fault Tolerance in Service-Oriented Architectures

Glen Dobson
*Computing Department*
*Lancaster University*
*Lancaster, LA1 4YR, UK*
*g.dobson@lancs.ac.uk*

Stephen Hall
*Computing Department*
*Lancaster University*
*Lancaster, LA1 4YR, UK*
*s.hall@comp.lancs.ac.uk*

Ian Sommerville
*Computing Department*
*Lancaster University*
*Lancaster, LA1 4YR, UK*
*is@comp.lancs.ac.uk*

## Abstract

*This paper introduces an innovative approach to improving service availability and reliability. Central to the approach taken are what we call fault tolerant service containers. These 'contain' externally provided services and in doing so add fault tolerance to them. This is achieved by allowing the container to be configured with a policy which specifies what kind of fault tolerance mechanisms may be applied to the services it contains. The container proxies calls to its services, passing them on to replicas in a pattern determined by the specified policy. A tool and SDK simplify the creation and deployment of the container and its policy. We conclude that, as it stands, this architecture provides a consistent mechanism for achieving certain kinds of fault tolerance. As demonstrated by the astronomical ephemeris application discussed herein, this also requires minimal effort from the developer. Furthermore, the extensible nature of this architecture means that its users can complement and extend the set of fault tolerance mechanisms already implemented.*

## 1. Introduction

Service-Oriented Architectures (SOA) are in the early stages of adoption in both e-business and e-science. The service oriented paradigm differs from that used in earlier technologies by offering interoperability and loose coupling driven by open, ubiquitous standards such as XML. Broad based adoption of services depends on the availability and ease of use of supporting technologies.

A key area for such supporting technologies is in handling the inherent unreliability of services. Services may fail for many reasons including resource starvation, faults in implementation and network instability. Service-based applications must therefore employ fault handling techniques such as fault tolerance to cope with errors propagated by their constituent services, and thereby ensure an end-to-end Quality of Service (QoS). Supporting technologies should aid developers in doing this, and provide a consistent mechanism to do so. SOA has no standards or technologies to deal with this at the application level. Specifications such as WS-Reliability and WS-ReliableMessaging deal with faults at the transport level.

Fault tolerant computing accepts that faults are unavoidable and may lead to component failure. Well understood mechanisms have therefore evolved that anticipate component failures but prevent these from leading to system failure. These fault tolerance mechanisms rely heavily upon the replication of components. By suppling components from different sources, diversity is introduced. 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 [9]. This is because similar components will fail in similar circumstances. Diversity of components is therefore desirable to avoid simultaneous failures. Until now, systems deploying significant diversity have largely been limited to safety critical applications such as avionics where ensuring system safety outweighs the increased financial costs.

One well known example of redundancy and diversity in avionics is the Airbus 340 fly-by-wire system (see [11], [12]). This employs quintuple redundancy, utilising three primary computers and two secondary computers in parallel. The processors used are all from different manufacturers and the type of processor differs between primary and secondary computers. The primary and secondary computers are supplied by different companies. Each computer also has two separate hardware channels which are compared for consistency. If inconsistency is detected then control switches to another computer. The software running on each channel is developed by a different team using a different language. The flight control software on the primary and secondary computers is also developed by separate teams. As well as all of this there is also redundancy and diversity at the mechanical level. All of this ensures a small chance of catastrophic failure because diverse replicas are less probable to fail simultaneously.

The service vision introduces the notion of plurality and diversity, with service providers competing in an open market. A fault tolerant system can take advantage of this competition by engaging several services for one task. Moreover, with multiple providers offering different implementations of the same service, diversity will not come at a high premium. For the first time it could be practical for non-critical software to leverage multi-version programming.

We have developed an extensible architecture allowing the application of fault tolerance mechanisms to a set of services. This is achieved through a fault tolerant service container (Figure 1). This container concept is analogous to that of component containers as, for instance, used with Enterprise Java Beans. Like these containers, the service containers described here add non-functional properties to the software component (or in our case the service). The container is configured with an XML fault tolerance policy model which leverages existing work by allowing fault tolerance mechanisms (e.g. retry, recovery blocks or redundancy) to be applied to at the application level. Key to the architecture are the notions of service plurality and diversity.

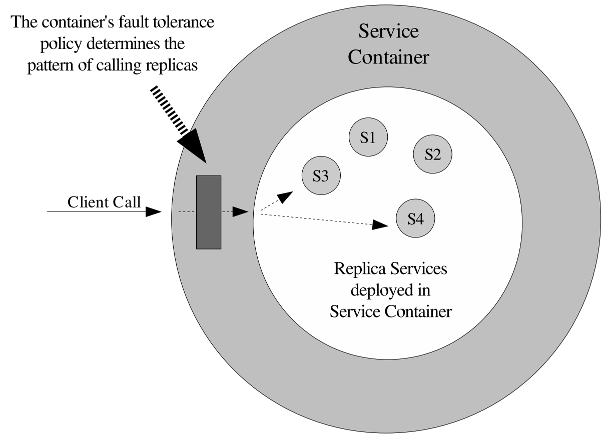

The container's fault tolerance policy determines the pattern of calling replicas
Service Container
S1
S2
S3
S4
Client Call
Replica Services deployed in Service Container

**Figure 1. Overview of Service Container**

Earlier we made the statement:

> *broad based adoption of services depends on the availability and ease of use of supporting technologies.*

We have therefore given due consideration to how a service container can be developed and deployed. Our approach is three pronged; firstly host the system in a middleware environment; have a software development kit (SDK) to enable the development of models and procedures within the context of the architecture. Finally, we present a tool that allows the aggregation and deployment of these models and procedures.

The following section sets the scene for our architecture by considering the details of the problem it addresses and the approach taken. Section 2 also introduces a scenario to be used as a point of reference throughout the paper. In section 3 we give a full review of the architecture. Section 4 discusses the practical aspects of utilising the architecture and making it concrete. We evaluate the architecture in section 5, and finally, in section 6, draw conclusions on this work.

## 2. Overview of Approach

There have been a number of efforts to achieve fault tolerance in service-centric systems. Many of these apply techniques used in other distributed systems. A common approach taken is hardware redundancy. This addresses the problem of service providers who are seeking to increase the availability of their own services. One of the advantages of this technique is that it is likely to be already applied by providers to increase the availability of their websites. In general the approach taken does not involve diversity however, but relies upon hosting replica services on multiple servers from the same manufacturer (often using the same server software and operating system). This lack of diversity means that failures are more likely to occur simultaneously in all replicas. A further problem with hardware redundancy is that there are likely to be some service calls which are never recovered because they occur in the time between failure detection and rerouting to a backup server. Software redundancy can avoid this problem.

In our architecture we achieve redundancy at the service level and therefore at the software and/or hardware level. That is, the host machines of our replicas may be different and the service implementations themselves may also be different. Secondly, because we are exploiting the nature of service-centric systems, our approach can easily be applied to third party services. It is therefore applicable beyond organisational boundaries. Discovering replica services at runtime through late binding is also possible. A final advantage is that in a service marketplace redundancy may be achieved at relatively small financial cost (compared to server replication within a single organisation for instance).

Some basic considerations of fault and failure handling have already been made in the web service architecture. Faults can be generated during service invocation and returned in SOAP messages as SOAP Fault elements. These are much the same as Java Exceptions. However such a construct does little to aid in dealing with byzantine faults.

Fault tolerance therefore has a role to play not only in responding to SOAP Faults - but in plugging the gap and allowing tolerance of byzantine faults. For instance a majority voting mechanism should increase availability in the presence of byzantine faults. This means that a degree of security can also be achieved since tampering can be detected by voting provided it only affects a minority of replicas (thus hosting replica services on diverse network routes is important where possible).

Our architecture decouples fault detection from fault tolerance and therefore the classes of fault it can handle are unbounded. In fact, this purely depends on whether appropriate fault detectors have been written. Currently, detection of unresponsiveness, network outages and explicitly returned SOAP faults are the main mechanisms that we have employed.

### 2.1. Ephemeris Voting Scenario

As an illustrative example of applying our architecture we introduce the example of an Ephemeris voting system that shall be used throughout this paper. An ephemeris is a table giving the coordinates of a celestial body at a number of specific times over a given period. Various online systems exist that will calculate Ephemerides (e.g. [2], [1]). We wrapped some of these as web services. Not only may these systems become unresponsive due to failure, they may return corrupt or incorrect data. This is both due due to the nature of the web and specific difficulties of calculating accurate ephemerides. To improve the reliability of the results returned by these services we simply perform a majority vote on the results returned by a number of them.

The essential inputs required to produce an ephemeris may include:

1. The name or designation of the body in question.

2. The start date, duration and interval to calculate the ephemeris for.

3. Some indication of whether the ephemeris is to be geocentric, heliocentric or topocentric (and longitude, latitude and altitude in the latter case)

4. Various inputs to affect the output quantities and format.

As an example, we concentrate on requesting a topocentric ephemeris for a Uranian Satellite: Ariel (UI). We request the ephemeris to be over a seven day period starting on January 1st 2005, 00:00, lasting 7 days, with a one day interval. We use the observatory at London's Regent's Park (observatory code 969) as the origin of the topocentric co-ordinate system ($0^o$09'16.6"W, $51^o$31'17.4"N, -4.02603 m). We request Ariel's Right Ascension (RA) and Declination (DEC) and ignore any other output returned. For the purpose of this paper it is not necessary to understand these values. Asa brief explanation RA/DEC coordinates are to the celestial sphere what longitude/latitude coordinates are to the earth's surface. RA is measured in units of hours, minutes and seconds whilst DEC is measured in units of degrees, minutes and seconds.

Different services return results formatted differently and with different accuracy. Therefore voting must be done only on the desired data and with some degree of tolerance for variation.

## 3. Architecture

Our architecture is an application of fault tolerance policies to a set of services via a service container. The container acts as a proxy to the actual services. Therefore, a message en route to a deployed service will be intercepted by the container, which adds a set of domain-independent peripheral services such as fault tolerance. The actions of the container are transparent to both the client and service provider. Our service container is exclusively programmed in Java. Containers achieve their relative transparency by relying on something programming pioneer David Wheeler noted in the 1950s (and quoted in relation to containers in [5]):

> *Any problem we are likely to encounter can be solved by introducing an extra level of indirection.*

### 3.1. Anatomy of a Service Container

We apply a policy, in this case a fault tolerance policy, to each message passing through the service container walls. To action the policy we intercept the service call and pass that message through a model representing that policy. The policy model itself consists of an acyclic graph of procedures. Essentially, we are providing nothing more than a mechanism by which actions can be applied to service messages. A procedure implements the actions of the policy model, for example in the case of a redundancy procedure, it clones a service message and concurrently redirects the clones down several connections. There are no limitations on the functionality of the policy model or constituent procedures, this is entirely extensible. Procedures are linked together with connections, these constitute the edges of the graph. Each connection is accessed programmatically from within a procedure but is deployed externally. Figure 2 expands upon Figure 1 by depicting a real instantiation of a policy model that relates to the Ephemeris voting scenario.

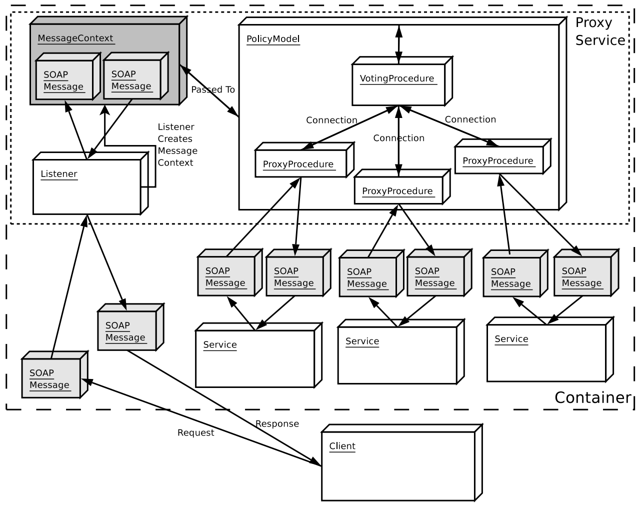


**Figure 2. Anatomy of Service Container**

## 3.2. Motivation for a Service Container

Experience with an EJB application server provides the motivation to create service-based containers. EJB components are only accessed indirectly via proxy objects generated by the EJB container. These proxy objects provide a range of non-functional peripheral services such as transaction management, security and load balancing. Non-functional properties are part of the bean's runtime deployment.

## 3.3. Service Container

Our service container (Figure 1), much like an EJB application server, deploys proxy objects, in the form of transparent services. These proxy services intercept messages *en route* to real services. The interception step is achieved by endpoint displacement; by which the endpoint of an actual service is replaced by the endpoint of a proxy service. Of course this introduces the requirement for the container to have access to the endpoints of real services. Endpoints are the addresses of services that are usually represented in the form of uniform resource identifiers (URIs). Because services are accessed by endpoints there are no requirements for a naming/directory service such as LDAP. However, a client must know the endpoint of the container based proxy service.

Unlike EJB components, the 'contained' services do not actually reside inside the container. This is also what distinguishes our service container from service containers as used in the more generic web server context. In reality, any of these services could be called directly without traversing the container. Another divergence between an EJB and service containers, is that restrictions are placed upon the components themselves. Inside an EJB container the bean is restricted, for example from using threads or being re-entrant; though this is only an advisory measure (a bean containing threads could be deployed). Our service container places no specific requirements on services given that their nature is outside our control.

## 3.4. Representing Policy Models and Procedures

In practice, the information essential to a policy model is represented in eXtensible Markup Language (XML). This XML model consists of three compulsory elements: a policy element plus one or more procedures and connections. The 'policy' element is the outermost element and represents the whole policy model for this proxy service. A policy element consists of 'procedure' sub-elements that map to procedure objects. Each procedure element indicates the class and identity of a procedure object. A procedure object is an instantiation of a Java class that implements a procedure interface. There is a one-to-one mapping between each procedure element in the policy model and a procedure sub-class. Each procedure sub-class must expose an invokable interface. In addition to the invokable interface, each procedure must have a constructor that takes the corresponding XML element from the policy model as an argument. The final compulsory element in the policy model is the 'connection' element. This must be nested inside a procedure element to represent a link to another procedure element. This link indicates that, after being invoked, the 'parent' procedure will pass the message context to the procedure indicated in its nested connection. Procedure elements represent the nodes of a graph whereas the connection elements represent the edges.

In Figure 3 we return to the example introduced in Section 2.1, and illustrate XML representing the policy model for the Ephemeris voting system. The XML shows a voting model requiring that at least 4 replicas respond. Not all elements are shown. Where etc. is shown a list of further elements akin to those above would be present. The ¡xpath¿ elements show how we extract the data to be voted upon from the SOAP message. Each value voted upon can also have the required majority and a tolerance specified. As can be seen the model is relatively simple, consisting of a voting procedure connected to N proxy procedures. It is these proxy procedures that are responsible for passing the message to external services.

## 3.5. Listeners

Figure 2 shows the container receiving a message from an arbitrary client, passing that message through a policy model containing message handling procedures. In order for a message to be passed through it must be wrapped in

```
<policyModel>
<procedure
      name="VotingProcedure"
      class="net.sourceforge.digs.endpoints.procedures.voting.VotingProcedure"
      start="true">
   <voting requirement="4">
      <election>
         <xpath>//RADECEntry/item[@date='2005-Jan-01']/DEC/deg</xpath>
         <electionClass>
            net.sourceforge.digs.endpoints.procedures.voting.election.IntegerElection
         </electionClass>
         <majority>3</majority>
         <tolerance>1</tolerance>
         <hungAction>0</hungAction>
      </election>
      etc...
   </voting>

   <connections>
     <connection id="c1" procedure="Proxy1"/>
     etc...
   </connections>

</procedure>

   <procedure name="Proxy1"
         class="net.sourceforge.digs.endpoints.procedures.ProxyProcedure">
      <endpoint
url="http://somehost.lancs.ac.uk:8080/ogsa/services/ephemeris/JPLEphemerisService"/>
   </procedure>
   etc...

</policyModel>
```

**Figure 3. Ephemeris Voting Policy Model**

a context. The job of wrapping a message is done by a Listener object, an instantiation of a EndpointListener subclass. Effectively, an EndpointListener is a Java servlet [4] that is mapped to a given endpoint. The listener class represents the first point of contact with our architecture, from which the policy model is invoked.

## 3.6. Contexts

A message context is a wrapper for any type of synchronous or asynchronous message. The MessageContext class itself is defined as abstract but is sub-classed for different types of messages. An example of MessageContext is the SOAPMessageContext used to wrap SOAP envelopes, other examples include XML and Stream contexts. The context is passed throughout a policy model from procedure to procedure through connections. All message contexts have an interface for creating, cloning, and storing properties. Sub-classes of MessageContext have specific methods for dealing with specific problems associated with the type of message. The architecture is designed to support a synchronous messaging by allowing not only a request but also a response envelope to be included in the message context. As with the Apache Axis architecture, asynchronous messaging is supported by simply ignoring the response envelope [3]. Problems associated with asynchronous messaging such as message correlation are not dealt with directly by the architecture, but we endeavour to provide the building blocks upon which this class of problem can be solved.

## 3.7. Proxy Service Architecture

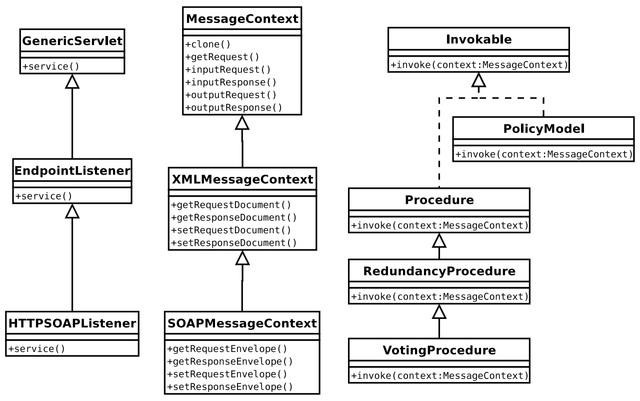


**Figure 4. Class Diagram for a Proxy Service**

## 3.8. Traversing a Proxy Service

Once a message context has been created (by the listener) the policy model is invoked. The only information the policy model receives is the message context although this may contain property objects. A model is instantiated as soon as it is deployed and so the message context gets passed directly to the root procedure. The root procedure is the procedure that has an attribute in the policy model XML that identifies to be the root. If no procedure is marked as the root, the first procedure in the list is deemed to be the root. From the root procedure the context is propagated to other procedures via connections. Within a procedure, connections can be accessed either through an iterative or direct interface. The direct interface requires forward knowledge of what the connection will be identified as in the policy model XML. It is always preferable to use the iterative interface of a connection set to prevent a lock-in between a procedure and its model.

## 3.9. Building Policy Models

A procedure is not limited in what it can do. The policy model could be implemented in just one procedure if required. But the architecture supports the division of procedures into the following broad classes:

1. Flow Control; these procedures implement the actual fault tolerance pattern, a redundancy procedure for example would split the flow into ’n’ concurrent nested procedures.

2. Redirection; these procedures map to services housed by the container. A message context passed to a redirection procedure will have it's contents serialized and sent directly to a service. A redirection service may block whilst waiting for a response.

3. Caching; these procedures will store messages, onto a context or possibly a database.

4. Query. The final class of procedure is querying where information can be retrieved from or placed into the message. A query procedure is useful for manipulating the headers of messages that might control the actions of actors later down the processing chain. A voting policy will use a query procedure to extrapolate the final results of a returned message.

Certain procedures will transcend classes for example the voting procedure is both of a flow control and query class.However, limiting the procedures to certain tasks serves three purposes:

1. Modularity. To ensure no part of the system is too big or unmaintainable.

2. Allow Procedure Re-use. Figure 4 shows that VotingProcedure inherits from RedundancyProcedure allowing code re-use and complex procedures to be built upon simpler ones. Incorporating all the tasks into one procedure would prevent inheritance from being possible.

3. Allow Profile Re-use. In this scope we define a profile as a generic policy model that we have defined for third parties to copy and extend. It much easier to extend an XML model than a large encompassing procedure.

The building of policy models can be tackled in one of two ways. Firstly by instantiating the generic profiles included with the architecture. Effectively, we have written several procedures encompassing several fault tolerance models including first past the post redundancy, voting, recovery, and filtering. Because these procedures are not designated to any specific service, the policy model itself contains most of the configuration information. For example a voting procedure for SOAP messages will require XPath references to the values being voted on. A second approach is to engineer your own procedures that are either generic or service specific. Understanding of the architecture and inventiveness are key requirements for any potential programmer. We supply an Application Programming Interface (API) and tools to support developers.

After introducing our architecture we present a set of tools, middleware and Software Development Kits (SDKs) that support the implementation of service containers.

## 4. The Architecture in Practice

### 4.1. Usage Scenarios

We envisage that the actor involved in adding fault tolerance to services or service-based applications may be one of several people. A knowledgeable user may have discovered that there are a number of replicated external services providing the functionality they desire. They may then make use this fact by configuring a fault tolerant container for these services and directing their application to that rather than directly to the services themselves. For groups of users making common use of services this knowledgeable user may be an administrator, who may set up the fault tolerant container on behalf of all of their users.

The application developer may also be the one to do this, perhaps building calls to a remotely hosted fault tolerant container into their application. On the other hand they may want to build the fault tolerant container directly into their client application to avoid extra messaging overheads and to keep a single point of maintenance.

It may also be a third party who wishes to add fault tolerance to external services (perhaps to charge the client for the added value or the service providers for the extra custom generated).

Finally, at the opposite end of the chain to the user, the service developer or service provider may wrap their own services to make them more reliable or available. They may also choose to set up the container to fall back to services provided by somebody else if it is important to avoid downtime.

### 4.2. Middleware

Rather than allowing the container to run standalone, middleware, in the form a Java servlet engine, is required to host the container. The justification for hosting the service container is to separate the concerns of fault tolerance from those of performance, security and stability in just the same way our architecture is separated from the service functionality it supports. Further justification is the ease with which the service container can be deployed (potentially dynamically) to a Java servlet engine. Java servlet engines range from the commercial such as IBM Websphere to the open source Apache Tomcat. Since Apache Tomcat is now recognised as a mature product, we have utilised it throughout our development and testing.

We could potentially use a more specific middleware for SOAP Services for example the Apache Axis SOAP engine. However, we wanted the architecture to handle non SOAP requests such as data streams. The architecture supports SOAP messaging (through the SOAPMessageContext), but could easily support other protocols such as Representational State Transfer (REST) [8] in this case by extending the XMLMessageContext. Also, we did not want to tie each proxy service to a given service interface; SOAP service engines generally provide interface checking before delegating the service request to a provider. Finally using a SOAP engine would dictate the structure of architecture affecting our ease of use philosophy.

Web based applications are deployed to a Java servlet engine in the form of a Web ARchives (WARs). A WAR can be copied to a launching directory where it will be dynamically deployed on the middleware, ready for client access. According to [4]:

> *A Web Application is a collection of servlets, html pages, classes, and other resources that can be bundled and run on multiple containers from multiple vendors.*

A WAR typically consists of resources that would be visible if the middleware was hit with a web browser or other human based client. Servlets are deployed as Java classes that are referenced by a ”web.xml” configuration file. All classes, jar files, and ”web.xml” are kept in a sub-folder called WEB-INF.

One of the advantages gained from using this middleware is that a web application can be dynamically updated without restarting the container. This feature is useful in our architecture because it allows for changing policies. The other advantage of using Java servlet technology is that it provides a degree of agnosticism with regards to the messaging protocol. Depending upon the servlet container employed, support for different underlying request/response protocols may also be gained. For instance Tomcat supports HTTPS.

### 4.3. Software Development Kit (SDK)

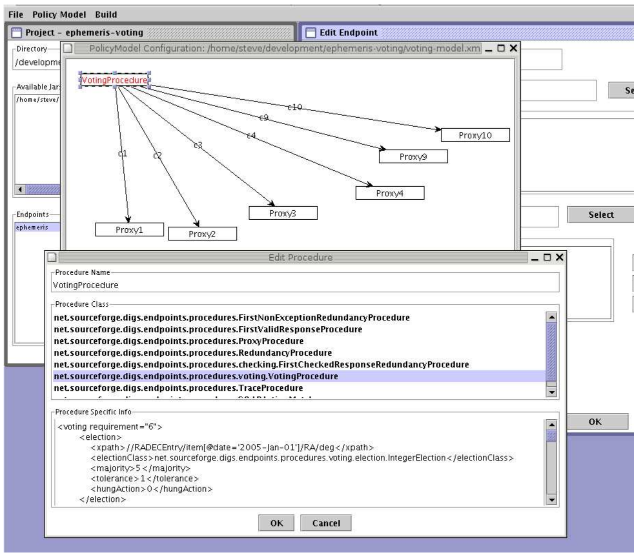
To allow our architecture to be truly extensible developers can include their own code. For this purpose the base code is abstract and can be extended using Java inheritance. A developer can extend the Listener, MessageContext and Procedure classes. The SDK consists of these base classes, a set of Application Programmers Interface (API) documents created by JavaDoc, and an architecture guide. The result of the development should be a set of jar files containing the extended classes. These must be deployed with a set of configuration files into a WAR file for deployment onto the middleware. The configuration files contain the policy model written in XML.

### 4.4. Deployment Tool

So far we have discussed the implementation of user-specific code and the requirement for these resources to be deployed to the middleware. The step of creating a WAR file containing enough information to function as a service container is sufficiently complex to require a tool. We have developed such a graphical tool to aid in building and configuring a fault tolerant service container. The tool represents a bridge between the code development and container deployment.

A user of the tool begins by creating a project. A project consists of a collection of jars developed by the user, and series of endpoint representations. The jars must contain the listener and procedure classes developed in accordance with the SDK. Each endpoint represents a proxy service to be deployed into the service container. The name of the endpoint, in addition to that of the project, is used to create its eventual url reference. For example a ephemeris endpoint in an ephemeris project could render the url http://somehost.com:8080/ephemeris/ephemeris.

Each endpoint maps to precisely one Listener and one policy model. The listener is selected from a list of candidates within the imported jar files. A policy model is chosen using a file selection dialogue. Once chosen the policy model can be edited rendering the display in Figure 5. Policy models can also be created and edited directly from the main menu or externally using any XML or text editor.



VotingProcedure

```
net.sourceforge.digs.endpoints.procedures.FirstNonExceptionRedundancyProcedure
net.sourceforge.digs.endpoints.procedures.FirstValidResponseProcedure
net.sourceforge.digs.endpoints.procedures.ProxyProcedure
net.sourceforge.digs.endpoints.procedures.RedundancyProcedure
net.sourceforge.digs.endpoints.procedures.checking.FirstCheckedResponseRedundancyProcedure
net.sourceforge.digs.endpoints.procedures.voting.VotingProcedure
net.sourceforge.digs.endpoints.procedures.TraceProcedure
```

```
<voting requirement="6">
      <election>
         <xpath>//RADECEntry/item[@date='2005-Jan-01']/RA/deg</xpath>
         <electionClass>net.sourceforge.digs.endpoints.procedures.voting.election.IntegerElection</electionClass>
         <majority>5</majority>
         <tolerance>1</tolerance>
         <hungAction>0</hungAction>
      </election>
```

**Figure 5. Ephemeris Model in Deployment Tool**

Every policy model is displayed as a directed graph and can be edited graphically by clicking on nodes or edges. Each node represents a map to a procedure and each edge a connection between procedures. Upon selection of a node, it can be edited by right-clicking, requiring a name, a implementing Java class, and a nest of embedded XML. A list of available procedure classes are displayed, again these come from the imported jar files within the project. The name will be reflected in the graph representation. One and only one procedure is chosen as the root. It is to this procedure that the message will be delivered. It is then the responsibility of the root procedure to distribute the message. In our example the root procedure is of the Voting Procedure type.

Every single procedure will require different XML to be nested. There are no limits placed on what that XML is, provided it is well-formed. No requirements have been lodged for XML schemas or document type descriptors (DTDs) by the tool or architecture. When developing a procedure, the XML required must be known as each procedure must parse its own nested set of XML. The abstract procedure class knows only how to deal with procedure and connection XML tags. An example of embedded XML is the voting rules in our ephemeris voting example.

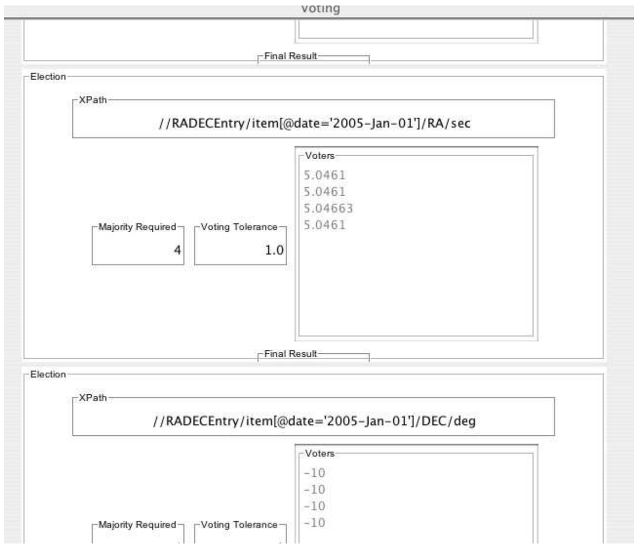
The policy model editor, consisting of the two front windows in Figure 5, saves the policy model in XML format, embedding custom XML in the appropriate procedure tag. Models will vary in complexity, but Figure 5 shows the model can remain simple whilst providing complex functionality. There is an obvious tradeoff between the complexity of the procedures and the model within which they reside.

Final deployment is invoked by the main menu. This creates a war file described earlier and places it wherever the user dictates. Creation of the war file requires the copying of jars and XML model files, the generation of a web application deployment file, and finally an archiving process to create the war. If the war is located correctly, in this case the webapps directory of tomcat server, the service container will be deployed dynamically and immediately ready for use. To use, the client must point at one of the endpoints mapping to each proxy service.

## 5. Assessment and Evaluation

We have successfully demonstrated the use of this architecture using the ephemeris voting example including a visualisation of the vote shown in Figure 6.

What has been shown is a successful application of a fault tolerance policy. However, this example has limitations. Firstly, this approach depends upon the fact that multiple replicas of service can be found or created. This has the obvious downside that a competitive service marketplace, which would make finding replicas simpler, does not exist at the moment. However, in the future is very likely to emerge in certain domains. Our architecture anticipates such a situation. In the mean time it applies to those services which do have multiple available implementations. Since these often have no consistent interface one of our future directions will be to simplify mapping service calls onto *similar* interfaces (as in [10]).



XPath: //RADECEntry/item[@date='2005-Jan-01']/RA/sec
Voters: 5.0461 5.0461 5.04663 5.0461
Majority Required: 4
Voting Tolerance: 1.0

XPath: //RADECEntry/item[@date='2005-Jan-01']/DEC/deg
Voters: -10 -10 -10 -10
Majority Required
Voting Tolerance

**Figure 6. Ephemeris Voting Example Visualistaion**

A further apparent shortcoming of our architecture is that many of the mechanisms we employ do not apply to stateful services. The policy model is stateless. All actions inside the proxy service occur within the scope of the message context. In reality the service container is required to store state beyond the scope of a single message and indeed allow for different proxy services to interact. However, the mechanisms we employ are in no way limited to those that are already implemented. Also, the fact that web services are inherently stateless, as well as the fact that those which hold state will do so using a database system with its own transaction management system, means that this is not an immediate problem. Our existing mechanisms can still be applied to many services.

In section 3 we made a reference to the fact that our architecture introduces an extra level of indirection. Whilst this is a common solution to software engineering problems it carries with it problems of its own. For instance, if the intermediate layer fails then it is often hard to track down the problem. In our architecture, if the redirected endpoint is hard-coded - but the Service Container itself no longer exists, or has failed for some reason, then the service call fails. To address this potential problem we propose a 'fault tolerance SOAP header'. An Endpoint Service can then be viewed as what is referred to as a SOAP intermediary in the SOAP specification [6]. By stipulating that the SOAP header must have the attribute mustUnderstand set to false, SOAP calls may pass to the original intended endpoint in the case of failure. Otherwise they may be routed via the Endpoint Service. Thus, provided it is made simple to add this header to calls when developing a client application (via an API) then transparent fall back to a straightforward service call is possible. Also, the client would no longer need to know the endpoint address for the container. Instead, a layer-7 routing component, would deal with mapping service calls to an appropriate fault tolerant container. Although this would be transparent to the client, somebody would still need to have set these mappings up in some form, even if this was just a case of publishing the redirected endpoints in some registry.

The added disadvantage of this approach would be that legacy client applications could not easily make use of the necessary SOAP header. A separate wrapper would have to be provided to support such applications without reimplementation. For new applications the headers could be added by using an API when implementing the client. The primary reason to introduce the header would simply be to mark the message for routing to a fault tolerant container as discussed above. However, it could potentially also contain a policy to be used to dynamically configure a container.

The one area in which fault tolerant service containers should be used with caution is in high performance applications. Some overhead is obviously introduced by the container itself. This is negligible in most situations, but may become significant if many contained services are composed together.

## 6. Conclusion

The architecture we have described in this paper is still in its youth. However it addresses an important problem for service-based systems: that of ensuring availability and reliability of services. As shown by the Ephemeris Service scenario, this can be achieved at low cost, both financially and in terms of development effort. In summary the additional effort involved creating a fault tolerance policy, which involved drawing the graph and writing a small amount of XML in our tool, and then building and deploying this in a container as a war file.

In the previous section we discussed several potential limitations of the architecture in terms of real world usage. We propose to build upon this architecture in the future to address these issues. One of these improvements is the routed access mode to our container using SOAP headers as discussed in the previous section. Another is extending the range of built-in policies. We currently support redundancy (in various forms) as our central fault tolerance mechanism. In the future we intend to implement check-pointing, and potentially group communication for maintaining stateful replicas (as shown to be feasible at the SOAP level in [7]).

We also intend to introduce an extended set of contexts. Currently limited to a message context, we propose having more contexts for trans-message and endpoint communications and synchronizations. A prime example of a requirement for trans-message communication is the asynchronous transaction. The response will be generated by multiple sources and arrive at the same endpoint. The result is several disparate message contexts that must be coordinated to form one response message using what ever policy is implemented. A trans-message context object that is available to each message context will allow information to be synchronised. The implementation of a trans-message context will rely upon each incoming message having a correlation ID. Given each response is a reply to the same request an ID is guaranteed we simply need to pull the required information from the SOAP header then establish the context. We intend to support context related standards such as WS-Context and WS-Transactions.

In the current implementation the policy models are deployed statically. Future plans include the dynamic deployment and adapting of models. Firstly, since policy models are represented as XML they or subsets of them can be pushed to a proxy service in a SOAP header. The end client can dictate what policy is used for fault tolerance and also which services should be referenced by the service container. Another vision for the architecture is to have dynamically adaptable models that will change in response to the service environment. A simple example of adapting is proxy procedures that will locate their own end services. More complex scenarios involve changing the fault tolerance policy in response to user requirements. Quality of service information about end services may also dictate what services are mapped to or overlooked.

Finally, we intend to utilise our architecture in roles beyond fault tolerance. An example is using the service container for injecting faults into a system. By placing a proxy service inline with a real service we can use the proxy to simulate failures such as fail-stop or even byzantine. These failures could be stochastic, probabilistic or even coordinated by a cross service model. From the client perspective the service would appear to have unstable behaviour, allowing fault tolerant techniques, in possibly another service container, to be tested. We propose that the service container can be used for third party service monitoring. Metrics about contained services such as response time, availability, and reliability can be easily obtained and stored into a database. This information can be passed to clients for them to choose the most dependable services available. Dynamically adaptable proxy services can also utilise dependability metrics.

## 7. Acknowledgements

Thanks are due to Stuart Anderson and Conrad Hughes our partners in Edinburgh University. This work was partially funded by the UK's Engineering and Physical Research Council, in conjunction with the DIRC project.

## References

[1] IMCCE ephemerides server. `http://www.imcce.fr/ephemeride_eng.html`.

[2] JPL HORIZONS ephemeris computation service. `http://ssd.jpl.nasa.gov/cgi-bin/eph`.

[3] AXIS architecture guide. `http://ws.apache.org/axis/java/architecture-guide.html`, 2003.

[4] D. Coward and Y. Yoshida. Java servlet specification version 2.4. `http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html`, 2003.

[5] J. O. H. et al. A container-based approach to object-oriented product lines. *Journal of Object Technologies*, 3(4):161–175, 2004.

[6] M. G. et al. SOAP specification version 1.2. `http://www.w3.org/TR/soap/`, 2003.

[7] X. Z. et al. Fault-tolerant grid services using primary-backup: Feasibility and performance. `http://www.cs.ucsd.edu/˜dzagorod/research/pubs/zhang_et_al-ft_grid_serv%ices_pb-cluster04.pdf`.

[8] T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

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

[10] M. Ouzzani and A. Bouguettaya. Efficient access to web services. *IEEE Internet Computing*, 8(2):34–44, 2004.

[11] I. Sommerville. A340 case study. `http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/Airbus340/inde%x.htm`.

[12] N. Storey. *Safety-Critical Computer Systems*. Addison Wesley, 1996.