Dependable Service Engineering: A Fault-tolerance based Approach

Ian Sommerville, Stephen Hall and Glen Dobson
Computing Department, Lancaster University, Lancaster UK.
{is, s.hall, g.dobson}@comp.lancs.ac.uk

Abstract

This paper is concerned with the engineering of dependable web services. We have developed an approach based on deploying existing web services within a middleware framework so that they are fault tolerant. Our approach is independent of the services themselves and may be configured to support a range of different fault tolerance mechanisms. Central to the approach are what we call fault tolerant service containers. These 'contain' externally provided services and incorporate support for fault detection and recovery. Containers are configurable with an XML-based recovery policy model that specifies what kind of fault tolerance mechanisms may be applied to the services it contains. The container serves as a proxy so that the existence of fault tolerance can be transparent to a service client. A tool and SDK simplify the creation and deployment of the container and its policy. We discuss the strengths and weaknesses of our approach and include some measurements of the overheads involved. We conclude that our container-based mechanism provides a simple, low cost approach to enhancing web service dependability.

Categories and subject descriptors: D.2.2 Design Tools and Techniques, D.2.13 Reusable Software

General terms: Design, Reliability

Additional key words and phrases: web services, service engineering, fault-tolerance

1 Introduction

Service-oriented computing is emerging as a new approach to distributed computing based on interoperable and loosely coupled elements (services). Services are distributed, stand-alone processing elements that are made available by service providers to service clients. Because they are language-neutral and based on open standards, services open up opportunities for resource and data sharing across organisations. Hence, they have been embraced by the e-business community as a means of facilitating B2B e-commerce and supply chain management and by the scientific community as a means of sharing computational resources in large-scale grids [1].

Services have much in common with components (as in component-based software engineering [2]) in that they can be considered as functional black boxes which are not developed by their user. However,

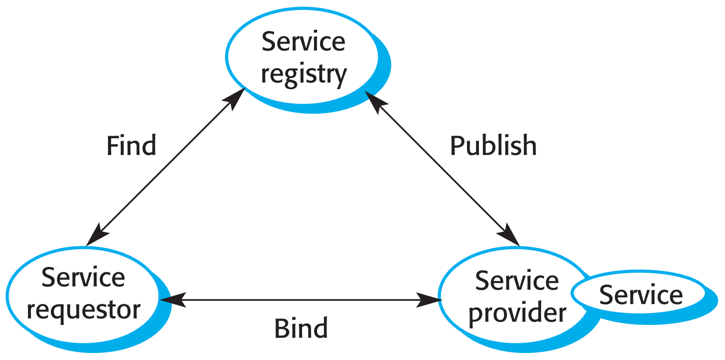


Figure 1: Service-oriented architectures

unlike program components which are integrated with other components in an application and which may be dependent on them, services are independent entities – they do not have a 'requires' interface. Consequently, unlike in component-based software engineering, programs using services do not have the responsibility of making required resources available.

The currently accepted model of service-oriented architectures (SOA) is shown in Figure 1, taken from [3]. In an SOA, services communicate by exchanging messages and open, ubiquitous standards underpin the notion of service-oriented architectures. Service communication protocols such as SOAP (a name, not an acronym) and service definition standards, such as WSDL (Web Services Description Language), and UDDI (Universal Description, Discovery and Integration) have been agreed, along with a wide range of supporting standards.

In the generally accepted vision of SOA, it is assumed that a range of services is available from multiple service providers. Service requestors (clients), depending on their specific requirements at a point in time, find a service in a service registry then bind to that service for execution. Late binding is possible so that an application binds to a service at run-time, immediately before that service is required. At some later time, with different requirements or resources available, the same client may chose a different service to perform the same operation.

Realising this vision of service-oriented architecture with late binding to services from different providers involves ensuring that service clients trust service providers and vice-versa. Trust is a multi-faceted notion but we maintain that if a service does not offer a high level of availability and reliability, it will not be trusted. We use the term *dependability* here to embrace both reliability and availability and we argue that dependability is a necessary requirement for trust. We consider a dependable service to be one that is available to deliver services when requested, that delivers results according to its specification and to an acceptable degree of accuracy and that delivers those results within an acceptable time.

Approaches to software dependability have generally assumed that failures are a consequence of program faults and that dependability is improved by avoiding the introduction of faults into a program, detecting and removing faults before the program is deployed and, for critical applications, including mechanisms that tolerate run-time faults and allow the program to recover from them [4]. Stringent development processes using techniques such as formal specification, program inspections, static analysis and systematic testing are used to increase the confidence of the developer that critical faults are avoided or are removed from a deployed system.

In situations where services are implemented in a conventional programming language, such as Java, these techniques of fault avoidance for dependability may, of course, be used. However, they have a high associated cost both in effort and time required. In practice, we believe that most service providers will be unwilling to incur the additional costs of rigorous development processes and, if available, will prefer easier and cheaper ways to enhance the dependability of their services. Furthermore, where third-party services are used, access to the service source code may not be available. Third-party service testing may be limited both by cost (assuming that each instantiation of the service incurs some cost) and by the service provider's disinclination to have an additional load placed on their service.

We are therefore convinced that fault tolerance is likely to be the most effective way to enhance service dependability. A widely accepted assumption in SOA is that there will be competing service providers offering comparable services. This means that there will be multiple, diverse implementations of the same functionality. Given that providing conventional fault-tolerant systems often involves developing several implementations of the same component, we conjectured that, if an appropriate support framework were available, then new dependable services, based on existing redundant, diverse services could be implemented relatively cheaply.

Our research has focused on providing such a service dependability framework, where we have separated the provision of service fault tolerance from the implementation of the services. Our aim was to support dependable service engineering by allowing service engineers to implement new services by embedding existing services within a general-purpose fault-tolerance mechanism. Given this aim, our research objectives were:

1. To provide service engineers with a simple way of specifying how fault tolerance should be provided for an existing service or group of services. This should not rely on a single model of fault tolerance built into the system but should allow the service developer to choose a model that is most appropriate for the type of service being developed.
2. To develop tool support for this mechanism so that fault tolerant services can be developed at a low-cost.
3. To develop middleware support for our approach to service fault tolerance.

The paper focuses on how we have developed support for dependable service engineering and describes our mechanism and supporting tools for developing fault tolerant services. We have not been concerned with the development of new approaches to fault tolerance nor with the effectiveness of particular fault tolerance techniques. We assume that the service engineer will choose the most appropriate fault tolerance approach for their application.

In the remainder of the paper, we review related work on fault tolerance and service dependability that is relevant to the work here. We then go on to describe how we can provide service-independent fault tolerance and discuss the current implementation of our system. We briefly evaluate the work that we have done, reflect on its strengths and weaknesses and, finally, discuss other possible uses for the generic approach to dependable service engineering that we have described.

2 Related work

There are a range of different approaches to fault tolerant computing which have been summarised by Pullum [5]. All of these rely on the existence of redundant compatible functionality, a means of fault detection and a means of selecting an acceptable result from multiple computations. An underlying requirement is that the redundant functionality should be diverse – that is, the design and implementation of different components should be different. This reduces the probability (as occurred in the Ariane 5 launcher explosion in 1996 [6]) of common failure of identical redundant components.

Approaches to fault tolerance centre on two basic models – a sequential execution model and a concurrent execution model. In the sequential model, characterised by Randell’s recovery block approach [7] some computing element is invoked and an acceptance test applied to the result. If the result satisfies the test, execution proceeds. Otherwise, an alternative implementation of the required functionality is invoked (or, sometimes, the same implementation is simply retried). In the concurrent model, several executions of redundant components proceed in parallel. The results are compared using a voting mechanism and it is assumed that the majority result is the correct result. Several variants of this model exist but the best known is, perhaps, N-version programming [8].

Web service standards do not currently have provision in the architectural infrastructure for software fault tolerance. It therefore has to be implemented in a proprietary manner by service developers. By contrast, CORBA has fault tolerance included in its specification. In Fault Tolerant CORBA [9], redundancy is achieved through replication of CORBA objects. The following fault tolerance strategies are supported in F/T CORBA:

1. Request retry
2. Redirection to an alternate server
3. Passive replication (one object performs operations, others are there as backups if faults occur)
4. Active replication (all objects perform operations concurrently)

In F/T CORBA, a group of replicas is called an object group. It is the object group with which a client inter-operates, thus making replication and the failure of replicas transparent. For the sake of scalability object groups may be managed in separate fault tolerance domains. Properties are associated with an object group or all object groups in a domain. Object groups are managed through a Replication Manager. Variations on basic replication schemes are achieved by setting the appropriate properties. Properties are also involved in configuring check-pointing, consistency and fault monitoring for an object group.

Much of the work on dependability in service-oriented architectures has been concerned with low-level issues that are complementary to our work on dependable service engineering. There have been several proposals for TCP-level server-side approaches to enhance service dependability [10-12]. Standards development has focused on the development of a reliable messaging protocol (WS-Reliability) [13] for information interchange between services. This protocol supports three aspects of reliability – guaranteed message delivery, duplicate elimination and message order maintenance. Our work does not overlap with this and there does not appear to be other standards-based work in the area of service fault tolerance.

Grid-WFS [14] is a failure handling framework for applications using grid services. The approach taken is to use a workflow as a recovery policy specification. This provides the flexibility to support a wide range of fault tolerant techniques and separates the recovery policy from the application code. The framework includes the XML Workflow Process Definition Language (XML WPDL) to express workflows. Fault tolerance strategies are separated into task-level and workflow-level techniques. At the task level, retrying, replication and checkpointing can sometimes mask failures. Checkpointing is supported through external libraries such as libckpt [15]. Successfully masked failures have no effect on the flow of the encompassing workflow.

Workflow-level strategies, which have to be explicitly programmed by the workflow developer, involve the alteration of the workflow. The workflow-level fault tolerance techniques supported by Grid-WFS are Alternative Task (essentially retry using a different task) and Workflow-level Redundancy (essentially redundancy using diverse tasks). User-defined exception handling is also possible at the workflow-level. Failure detection in Grid-WFS relies upon heartbeat monitoring and event notification. Thus, halting failures and exceptions thrown by services can be detected. Whilst Grid-WFS does provide a flexible framework for Grid fault tolerance, it is limited in its failure detection abilities and in the number and configurability of its fault tolerance techniques. It also relies on its own non-standard workflow language.

The ADAPT approach [16] is also a workflow-based approach to fault tolerance but it includes some support for service fault-tolerance. It relies on replication of the primary server and multicasting state to replicas. Similarly, work by Birman [17] adopts a server-based approach to support service fault-tolerance. Both of these proposals differ from our work in that the fault tolerance model is embedded in the system rather than under the control of the service developer and requires some server-side support.

A feasibility study into using primary backup for fault tolerant Grid services [18] was undertaken as part of the GriPhyN (Grid Physics Network) project. The study was also used to investigate the dependability/performance tradeoffs associated with implementing a fault tolerant Grid-based system. Three implementation approaches were investigated. One used OGSI notification, one standard Grid requests, and the other worked at a lower level, communicating over TCP sockets. In each implementation state was updated after every operation (i.e. a warm passive replication strategy was used). Because of this, the service interface had to be altered to include extract_state and inject_state operations. On the client side, a fault tolerant stub interceded between the client and the normal SOAP stub. This made the use of primary backup somewhat transparent, however did require alteration of the client code to reference the correct stub.

Whilst this work demonstrated the feasibility of one particular fault tolerance strategy in service-oriented architectures its scope did not go beyond this. The transparency of the approach is also questionable, given the need for a client stub, as well as alterations to the service interface.

Looker and Munro [19] discuss a specific mechanism for web service fault tolerance based on N-version programming with voting. The approach, like ours, relies on the creation of proxy services but appears to rely on specific technology in the Apache Axis server. It appears that the fault tolerance mechanism is hard-wired into the system and, so far, they have only implemented an N-version approach. This is ‘hard-wired’ into their system so supporting alternative fault tolerance mechanisms requires program change.

3 A container-based, fault tolerance framework for service engineering

Our goal was to develop a general-purpose mechanism to support fault tolerance that could operate with any suitable web service or set of equivalent web services. We assumed that services were externally provided and that we had no control over their implementation or deployment – we could not, for example, re-host a service on a server that we controlled. In addition, the fault tolerance mechanism should be transparent. Users of a service should not have to be aware of the fault tolerance support added to the service and service engineers should not have to take fault tolerance into account when developing their services.

The conceptual approach that we have used is similar to that adopted in component-based software engineering [2] where generic support facilities are made available to components by deploying these components in a container (e.g. in Enterprise Java Beans). For example, an EJB container provides support for transaction management and load sharing and components deployed in the container need not be concerned about implementing such support themselves. We conjectured that fault-tolerance is a characteristic that could also be incorporated into a container.

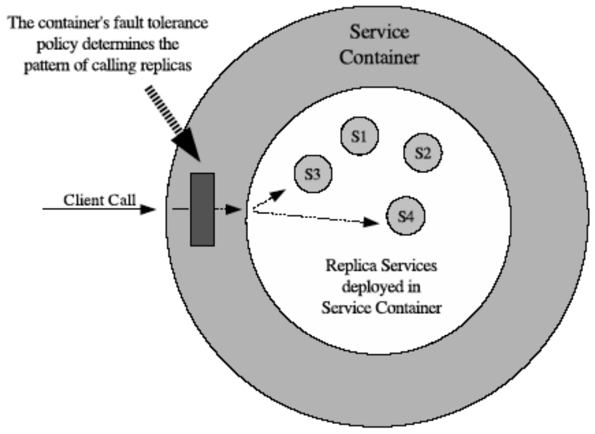

The container's fault tolerance policy determines the pattern of calling replicas
Service Container
Client Call
S1
S2
S3
S4
Replica Services deployed in Service Container

Figure 2: A service container

We have designed and implemented a general purpose container that incorporates fault tolerance mechanisms. When a service (or several redundant services) is referenced through our fault tolerance container, a fault tolerant proxy service is created that replicates the functionality of the original service. This can be used transparently by the service requestor who calls this proxy service in exactly the same way as any other service. Hence, services can be transformed to fault tolerant services without modification of either the service or the client using the service. Service clients need only be aware of the URI of the proxy service. Figure 2 illustrates this situation where several replica services are 'contained' with the container providing a single entry point to these services. Note that, unlike components, the services themselves are not actually deployed in the container – the container simply includes a reference to the services' URI.

This mechanism is a general one that does not rely on any particular fault tolerance model. Nor does it rely on the availability of diverse, redundant services. As we discuss later, the container is configured with the preferred approach to fault tolerance. For many web services, transient failures are the most common class of failure so fault tolerance can be achieved through be a simple retry mechanism that detects service failure and resubmits the service request.

We anticipate that this mechanism for providing fault tolerance could be used in different ways:

1. Service providers may replicate their services to ensure availability and use the container mechanism to request retries. They might simply embed their service in a container and use this to recover from transient failures. In addition, they could cope with server failure by hosting replica services on other servers and switch execution automatically to these if a failure is detected.

2. Third-party providers may wrap services from other providers and offer value-added fault tolerant services to customers with critical business applications. In essence, they are choosing to become service providers but are outsourcing the provision of the services themselves.

3. Developers of service-oriented systems who are concerned about the dependability of a service that they plan to use, may use the framework to develop their own service that incorporates functionally comparable services from different providers and thus handles failures of externally provided services. This might be used in situations where the service providers are untrusted and replication of different services provides re-assurance to the client.

Our approach can be used without change to support these different types of use.

3.1 An illustrative example

As we have discussed, a central goal of our work was to provide a mechanism for fault tolerance that is independent of the service implementation. Hence, to assess and evaluate our work, we require the existence of multiple instances of the same or comparable services. This is problematic as service provision is currently fairly limited so we have adopted an approach where we wrap existing services or web sites to create web services with a common interface. This wrapper approach addresses a more general problem of ensuring that services offering comparable functionality all present the same interface.

To demonstrate our architecture, we have developed an example of a fault-tolerant weather forecasting application that makes use of public weather forecasting services (Figure 3). This system can cope with service failure and with some inconsistencies in the results produced by different services. A user accesses the application through a web browser, which presents the appearance of a single weather forecasting service. The application actually accesses several services offered by different providers. Each weather service receives a postal code (zip code) and responds with weather information about that location, organised in a consistent way. In this trial application, we collect results from these different services and use a voting mechanism to decide which results to return to the application. The system currently consists of five diverse weather services:

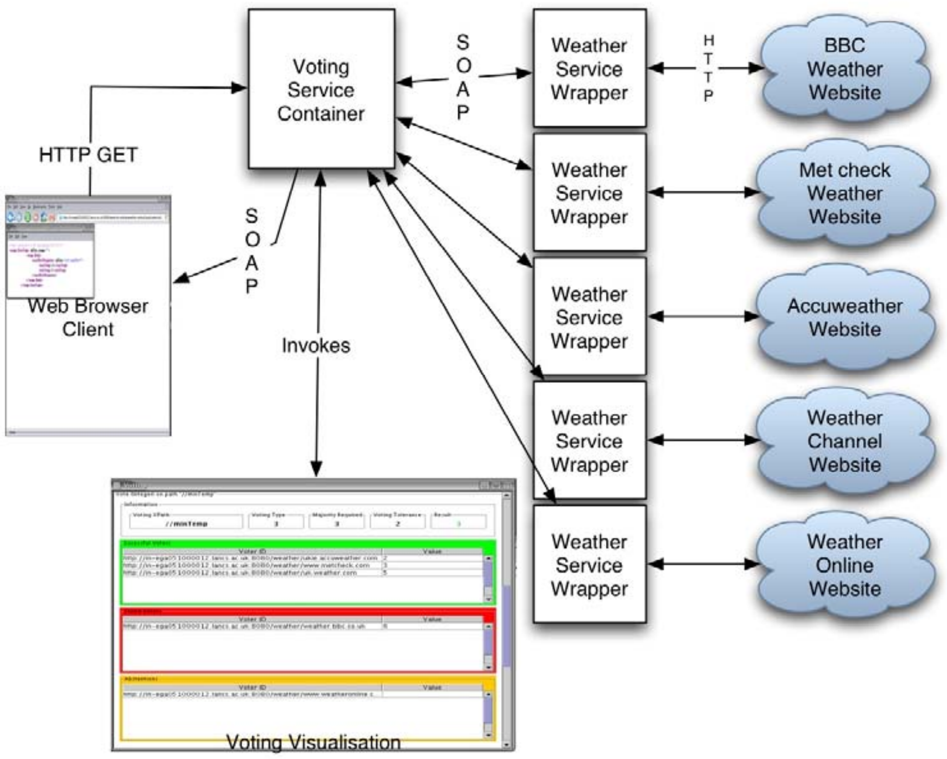


Figure 3: A fault tolerant weather forecasting application

1. BBC Weather (http://www.bbc.co.uk/weather)

2. Met Check (http://www.metcheck.com/)

3. Accuweather (http://ukie.accuwweather.com)

4. Weather Channel (http://uk.weather.com)

5. Weather Online (http://www.weatheronline.co.uk)

Obviously, there are many attributes of a weather forecast upon which we could vote. To demonstrate the voting, we limit ourselves to two fields – the maximum and minimum temperatures for a given day. The voting mechanism that we use consists of a simple majority algorithm that clusters responses based on their equivalence within a set tolerance. Once a cluster obtains enough responses such that the majority threshold is reached it declares itself the winner of the vote. The returned value is simply the mean of the responses within the winning cluster.

With the tolerance and majority set to appropriate values, the weather forecasting voting system should operate in the presence of crash and silent fails in the individual services. The voting mechanism should also mask malicious or Byzantine failures. Assuming that all services have a comparable probability of error, it also handles inaccuracies in the forecast. The system is designed to illustrate our generic fault tolerance mechanism and the voting mechanism used. As weather forecasts are inherently inexact, we do not claim that the final result is any more or less likely to be correct.

3.2 Fault-tolerance models

There are a number of different ways to implement service fault tolerance with the most appropriate approach dependent on the application domain of the system being developed and the hardware/software configuration that is used. These approaches include:

1. *Rewind and retry*. A request for service is submitted and fails. The state is re-initialised and the same request resubmitted to the same service. This approach is appropriate in situations where faults are likely to be transient and where the service is transaction-oriented i.e. failure of the service does not result in effects outside of the service itself. This approach is therefore appropriate for many web-based services where failure is a consequence of temporary load on the service platform.
2. *Resubmission to alternative implementation*. In this case, failure results in the resubmission of a request to a functionally-equivalent service provided by some different service provider. This approach is appropriate when either the results returned from a service do not meet some acceptance test (as in the recovery block scheme [7]) or when the service fails to return a result. In principle, dynamic discovery of alternative services could be invoked after a service failure has been detected.
3. *First past the post redundancy*. In this case, the service request is submitted simultaneously to a number of different services offered by different providers. The result of the call is taken to be the first result returned by any provider that meets some acceptance test. This is a variant of 2 and may be used when as rapid as possible a response is required and service response times are highly variable.
4. *Multi-version executing with voting*. The service request is submitted to different services offered by different providers. Once all results are available, a voting mechanism is used to select the majority result. This approach, copes with service unavailability, and is appropriate in critical situations where it is imperative that a result with a high likelihood of being correct is returned.

Our system provides support for service fault tolerance by providing a set of general Java components that implement different approaches to fault tolerance such as ‘rewind and retry’, ‘multi-version execution’, etc. These components are configured for use in an application using a *recovery policy model*. As discussed in the introduction, the specific recovery policy is implemented by a proxy service that ‘wraps’ the actual functional service that is requested by the service client.

The basic policy model element is a ‘procedure’, which is an abstraction that can be implemented either by an external service or by a Java component. Each procedure names an actual Java class or external service that is to be configured along with configuration information that is specific to that procedure. This may include the definition of connections that specify other procedures that are to be applied.

A policy model must be defined for each proxy service. The policy model, essentially, defines the fault tolerance model to be used, its configuration and the names and endpoints of the actual services that require to be fault tolerant. For example, Figure 4 is the policy model for the weather service that we have illustrated in Figure 3.

The first procedure definition specifies that the Java class VotingProcedure is to be used. This class implements multi-version execution with voting. It has to be configured with the number of voters that are required to return results (where a result can be a failure indicator) and the majority required for the computation to be successful. The declaration of VotingProcedure defines the location of the associated Java class (net.sourceforge.digs.endpoints.voting.VotingProcedure) and a requirement that 5 voters must be available. The <xpath> tag indicates the information from the SOAP message that is passed to each of the voters and the following tags indicate the location of an election procedure that counts votes, the required majority and the allowed tolerance of 2 degrees. This tolerance is required because the results from temperature forecasts may differ slightly.

Figure 4 also illustrates how we make links to external services in the definition of the procedures Proxy1, Proxy2, etc. These simply package the external service reference so that Proxy1 uses the BBC weather service (weather.bbc.co.uk), Proxy2 uses a service located at ukie.accuweather.com, etc.

Currently, we have implemented general fault tolerance mechanisms corresponding to all of the approaches discussed above. However, this set can easily be extended and we have implemented tool support that allows alternative approaches to be implemented and embedded in our system.

```
<policyModel>
  <procedure
    name="VotingProcedure"
    class="net.sourceforge.digs.endpoints.voting.VotingProcedure"
    start="true">
    <voting requirement="5">
      <vote>
        <xpath>//maxTemp</xpath>
        <voteClass>net.sourceforge.digs.endpoints.voting.vote.IntegerVote</voteClass>
        <majority>3</majority>
        <tolerance>2</tolerance>
      </vote>
      <vote>
        <xpath>//minTemp</xpath>
        <voteClass>net.sourceforge.digs.endpoints.voting.vote.IntegerVote</voteClass>
        <majority>3</majority>
        <tolerance>2</tolerance>
      </vote>
     </voting>

    <connections>
      <connection id="Proxy1" procedure="Proxy1"/>
      <connection id="Proxy2" procedure="Proxy2"/>
      <connection id="Proxy3" procedure="Proxy3"/>
      <connection id="Proxy4" procedure="Proxy4"/>
      <connection id="Proxy5" procedure="Proxy5"/>
    </connections>
  </procedure>

  <procedure name="Proxy1" class="net.sourceforge.digs.endpoints.proxy.ProxyProcedure">
    <endpoint
      url="http://in-ega051000012.lancs.ac.uk:8080/weather/weather.bbc.co.uk"
      proxyHost="wwwcache.lancs.ac.uk"
      proxyPort="8080"/>
  </procedure>
  <procedure name="Proxy2" class="net.sourceforge.digs.endpoints.proxy.ProxyProcedure">
    <endpoint
      url="http://in-ega051000012.lancs.ac.uk:8080/weather/ukie.accuweather.com"
      proxyHost="wwwcache.lancs.ac.uk"
      proxyPort="8080"/>
  </procedure>
  <procedure name="Proxy3" class="net.sourceforge.digs.endpoints.proxy.ProxyProcedure">
    <endpoint
      url="http://in-ega051000012.lancs.ac.uk:8080/weather/www.metcheck.com"
      proxyHost="wwwcache.lancs.ac.uk"
      proxyPort="8080"/>
  </procedure>
  <procedure name="Proxy4" class="net.sourceforge.digs.endpoints.proxy.ProxyProcedure">
    <endpoint
      url="http://in-ega051000012.lancs.ac.uk:8080/weather/www.weatheronline.co.uk"
      proxyHost="wwwcache.lancs.ac.uk"
      proxyPort="8080"/>
  </procedure>
  <procedure name="Proxy5" class="net.sourceforge.digs.endpoints.proxy.ProxyProcedure">
    <endpoint
      url="http://in-ega051000012.lancs.ac.uk:8080/weather/uk.weather.com"
      proxyHost="wwwcache.lancs.ac.uk"
      proxyPort="8080"/>
  </procedure>
</policyModel>
```

Figure 4 An XML policy model defining a fault tolerance policy using diverse redundant services

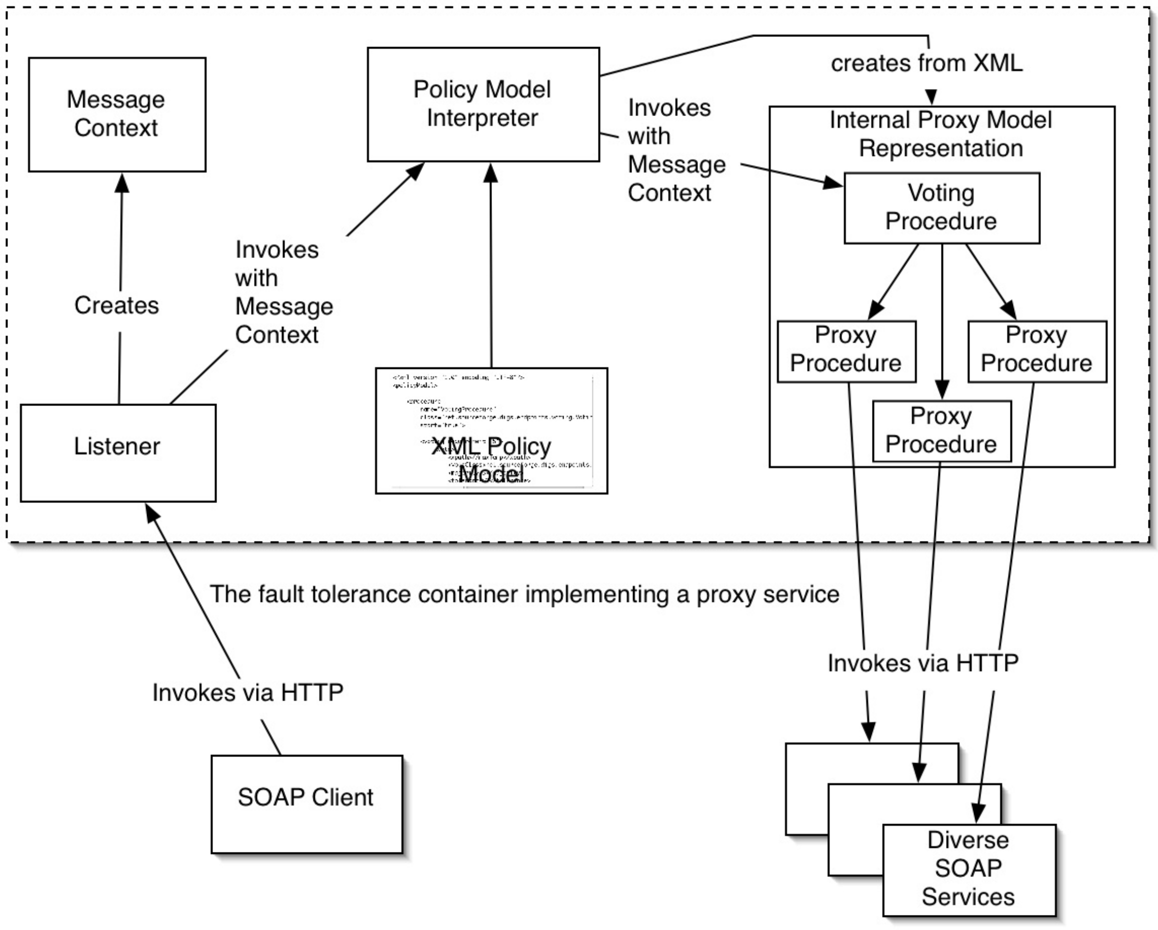


Figure 5: The anatomy of a fault tolerance container

3.3 Policy interpretation

The fault tolerance container is responsible for accepting incoming service requests from a client (as SOAP messages), passing these on to the called services then returning the results to the client as a SOAP message. The key components of a container are shown in Figure 5, which illustrates a simplified version of the policy model shown in Figure 4. To make the diagram easier to manage, we have simply shown 3 rather than 5 proxy procedures and corresponding external services.

The container components are:

1. The Listener, which is responsible for handling SOAP messages, sent to and from the container. When an incoming SOAP message is received, the Listener creates a Message Context and invokes the Policy Model Interpreter. After execution of the embedded services, the Listener then abstracts the response and returns this to the service client as a SOAP message.
2. A Message Context is used to allow a single service call to be replicated and sent to multiple services. It includes a unique identifier plus slots for the incoming SOAP message making a service request and the SOAP message which is a response from a service. We need this facility so that we can clone incoming SOAP messages to send them to different services and ensure

that the responses from each these different services can be identified, coordinated and related to the input message. The message context also includes methods to allow SOAP messages to be managed, and slots where meta-information (such as performance or failure information) about service execution can be recorded.

3. The Policy Model Interpreter is the run-time system that invokes the fault tolerance class to be used and which manages all communications with that class. Its primary role is to parse the policy model, creating an internal representation of this which is used by the specific fault tolerance components such as Voting Procedure. It also provides access to a set of components which can be used by invoked procedures, such as a redundancy component that clones message contexts where redundant service execution is required.
4. A generic fault tolerance component such as Voting Procedure in Figure 4. This component uses the internal representation of the policy model to configure itself and hence to implement the particular fault tolerance strategy that is being adopted.

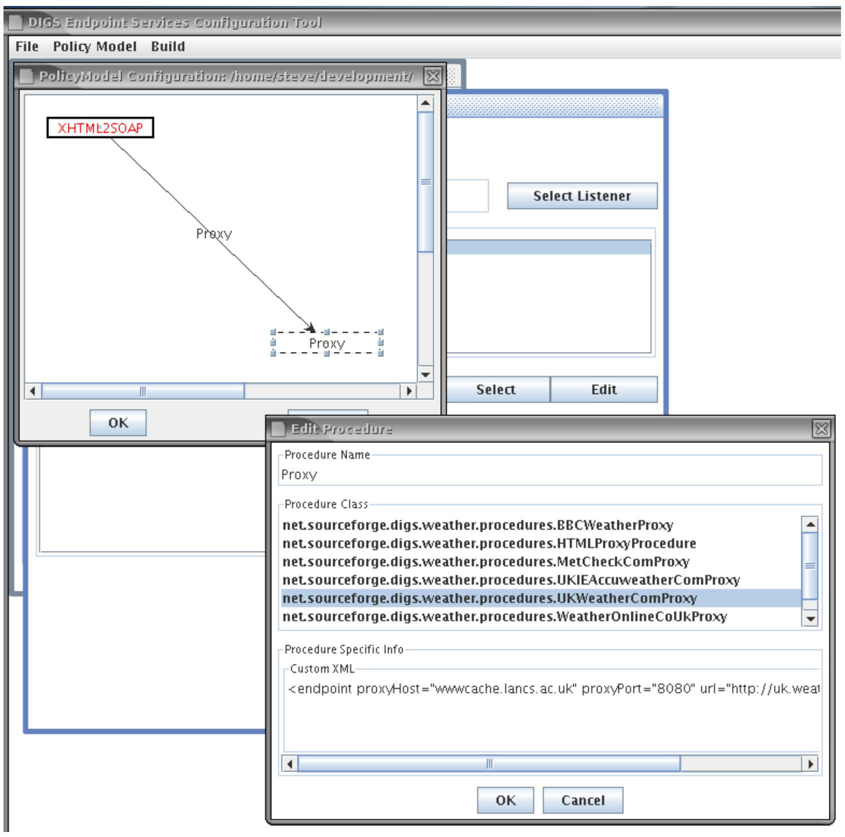
3.4 Failure detection

Failure detection is obviously central to any system that supports fault tolerance. However, when services are developed and made available by a range of different service providers, the only way in which a coherent approach to fault detection may be supported is through the use of standards for fault reporting. Unfortunately, such standards do not currently exist. In our case, the PolicyModelInterpreter is responsible for failure detection and for completing the appropriate components of the MessageContext if a failure occurs.

Currently, our approach to failure detection relies on the following mechanisms:

1. If a service involves short transactions and so can reasonably be expected to complete in a small number of seconds, we assume that the service has failed if there is no response after N seconds. Currently, we simply use TCP timeouts but this could easily be amended to any other timeout mechanism, implemented through procedures.
2. If a service involves long transactions (e.g. implements a computationally-intensive task), we assume that it either includes a heartbeat monitor or can respond to ‘pings’ during execution.. Service failure is assumed if there is no heartbeat detected or if there is no response to a ‘ping’.
3. If the service can report exceptions using the <fault> field in the SOAP message, we assume that a fault has occurred if any such exception is reported.

If a service fails, failure information can be encoded in the meta-information in the message context and returned to the Listener for logging, etc. Given that the majority of service failures are likely to be


DIGS Endpoint Services Configuration Tool
File
Policy Model
Build
PolicyModel Configuration: /home/steve/development/
XHTML2SOAP
Proxy
Proxy
OK
Select Listener
Select
Edit
Edit Procedure
OK
Cancel

Procedure Name
Proxy

Procedure Class
net.sourceforge.digs.weather.procedures.BBCWeatherProxy
net.sourceforge.digs.weather.procedures.HTMLProxyProcedure
net.sourceforge.digs.weather.procedures.MetCheckComProxy
net.sourceforge.digs.weather.procedures.UKIEAccuweatherComProxy
net.sourceforge.digs.weather.procedures.UKWeatherComProxy
net.sourceforge.digs.weather.procedures.WeatherOnlineCoUkProxy

Procedure Specific Info
Custom XML
<endpoint proxyHost="wwwcache.lancs.ac.uk" proxyPort="8080" url="http://uk.weat

Figure 6: The policy model editor

availability-related i.e. the service has not responded to a request, we believe that the failure detection approach that we have adopted is appropriate. Of course, reliability-related failures, where a computation is carried out incorrectly, can be detected by using concurrently-executed services and a voting mechanism to select the common results.

4 Tool support for fault tolerant service implementation

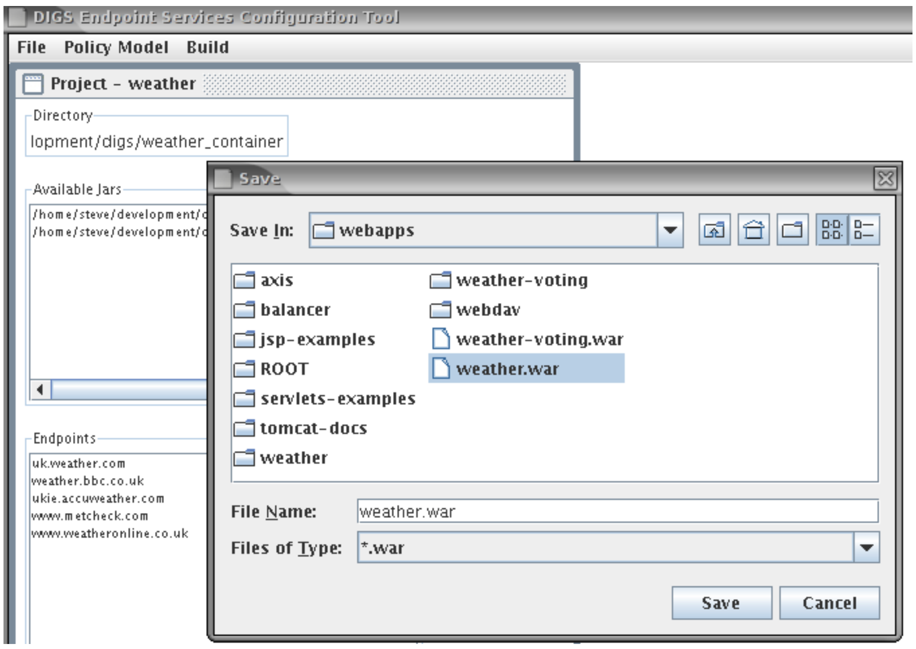
To simplify the process of implementing a fault tolerant service, we have developed tool support for container implementation and deployment. The tool includes a graphical policy model editor so that users do not have to write policy descriptions in raw XML, support for managing the defined procedures and the services that are to be used and support for deploying the container on a Java servlet engine. In addition, the mechanism we have used to represent the components implementing the generic fault tolerance mechanisms allows the Java code implementing these to be accessed and modified as required.

Writing raw XML is both tedious and error prone so we decided to implement a policy model editor that removes some of this burden from fault tolerant service implementers. This editor is a simple graphical system that allows policy procedures and their connections to be defined. Using this editor, the user can draw the tree of connections then fill in the specific XML associated with each procedure in the lower editor window. Figure 6 shows the editor being used to edit the XML model for the proxy procedure associated with uk.weather.com. The upper window shows the tree of connections for a procedure. As the weather forecast in this example is presented as a web page, a reusable procedure, XHTML2SOAP is used to turn this into a web service. The bottom pane in the lower window shows the XML that is specific to UKWeatherComProxy – in this case, the definition of the service endpoint. In addition to the graphical editing facilities, our tool includes facilities to manage lists of procedures that are being defined and service endpoints.

In addition to editing and file management, our tool also supports the deployment of fault tolerant services on a server. Containers are implemented as servlets running on a Java servlet engine. We have used Apache Tomcat as a servlet engine throughout our development and testing but the approach is portable to any other comparable middleware.

Containers are deployed as web archives (WARs) on the servlet engine. A key benefit of this approach to implementation is that the 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, different underlying request/response protocols may be supported (e.g. HTTPS as well as HTTP).

The WAR files are made up of a number of Java archive files (JARs), the policy model and other generic deployment information. As the construction of WAR files is fairly complex requiring the marshalling of information from different sources, we have implemented a tool to simplify this. Figure 7 shows the interface for the deployment tool that we have developed. Deployment of the container is straightforward – the WAR file is created using our tool and copied to a launching directory (webapps) on the servlet engine. It is then automatically deployed on the middleware, ready for client access.



Directory: lopment/digs/weather_container

Available Jars:
/home/steve/development/
/home/steve/development/

Endpoints:
uk.weather.com
weather.bbc.co.uk
ukie.accuweather.com
www.metcheck.com
www.weatheronline.co.uk

Save In: webapps

axis
balancer
jsp-examples
ROOT
servlets-examples
tomcat-docs
weather
weather-voting
webdav
weather-voting.war
weather.war

File Name: weather.war
Files of Type: *.war

Figure 7: WAR file deployment

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. These base classes can be made available through our tool consists of these base classes along with a set of Application Programmers Interface (API) documents created by JavaDoc, and an architecture guide.

5 Assessment and evaluation

The assessment of any recovery mechanism is a difficult task. Service faults appear to clients to be random and are relatively rare so measuring the actual improvement in dependability is an expensive and long-term process. This can only be justified for mission and safety-critical systems. Furthermore, because we can support different recovery models, any dependability measurement will not really measure the contribution of our particular approach but rather the inherent benefits of the recovery model that is used.

In terms of dependability improvement, we can therefore only realistically make a qualitative assessment of our approach and highlight what we see as its strengths and weaknesses as far as dependability is concerned. However, we have made some measurement of both the process of service creation and the performance overheads imposed by the container approach. In essence, in this evaluation of the system, we have tried to answer the following questions:

1. How much effort is required to create a fault tolerance version of a service?
2. What overheads are involved in the proxy mechanism?

Service	Mean access time	Mean access time of containerised service	Delta
BBC Weather	736 ms	897 ms	161 ms
Met Check	708 ms	851 ms	143 ms

Figure 8: Measurements made using a single service embedded in a container

Service	Mean access time	Delta from longest response
BBC Weather	710 ms	
Met Check	788 ms	
Weather Online	612 ms	
Container with 3 voters	1264 ms	476 ms

Figure 9: Measurements made with 3 services voting in a container

Service	Mean access time	Delta from longest response
BBC Weather	693 ms	
Met Check	788 ms	
Weather Online	612 ms	
UK Weather	989 ms	
Accuweather	974 ms	
Container with 5 voters	1264 ms	518 ms

Figure 10: Measurements made with 5 services voting in a container

The effort required to create a fault tolerance service by embedding existing services in a container varies depending on whether or not an existing fault tolerance mechanism is used. If the fault tolerance required consists of a pre-written policy, such as voting then building fault tolerant services simply involves configuring the policy model for the specific requirements. We conservatively estimate this task to take no more than one hour. This effort is reduced significantly by using the tool to create the model as demonstrated in figure 6, we estimate it took about half an hour for the weather voting model. About 10 minutes is required for system deployment and the creation of the WAR file. So, where an existing policy is available, then creating a fault tolerant web service should normally take about an hour.

If an approach to fault tolerance is to be used that is not supported by our reusable components, then significantly more time is required. Custom procedures are easily produced by subclassing the Procedure interface but the actions of a procedure are not limited or guided and hence require programming experience and skill. Nevertheless, the time required for an experienced Java programmer should not normally be more than a day. However, the mechanisms that we have implemented cover fault tolerance policies that are applicable to a wide range of application types and custom development should only be rarely required.

To assess the performance overhead imposed by the system, we have made three separate sets of measurements:

1. Measurements with only a single service deployed in a container.
2. Measurements where three services are deployed in a container and a voting mechanism is used.
3. Measurements where five services are deployed in a container and a voting mechanism is used.

The first set of measurements allows us to assess the container overhead without concern for the overheads imposed by the voting mechanism. The second and third set of measurements gives us an indication of the additional overheads imposed by voting. Measurements were made by accessing the services continuously over a 20-minute period, with the measurements repeated at different times of the day. This results in several hundred 'hits' on each web service. Figures 8, 9 and 10 show the results of these tests.

For the measurements of a single service, we wrote a program that simply accessed the service then measured the response time in milliseconds. We embedded this service in a container and repeated this thus accessing a proxy for the weather service. This measurement gave us information about the overhead resulting from the use of a container to implement the proxy service without any significant processing going on inside the container. As is clear from Figure 8, the overhead is approximately the same for different services and of the order of 150 ms.

We repeated the experiment using 3 and 5 voters as shown in Figures 9 and 10. This measurement gave us some information about the overhead that is added by including a voting algorithm. With 3 voters, this is of the order of 470 ms and, as expected, is higher with 5 voters (518 ms) because of the additional processing required.

With 3 and 5 voters, we measured the access times of services from calls within the container. Interestingly, the same services showed different access times in each case – for example, the BBC weather service had average response times of 693 ms, 710 ms and 736 ms. We consider these are probably simply random variations depending on server load – they are within 10% of each other although we were a little surprised that our averaging mechanism did not smooth out these variations.

Overall, the overhead imposed by introducing a fault tolerance mechanism based on voting seemed to involve adding about 0.5 seconds to the response time of the services used. Sequential fault tolerance approaches will obviously have a longer response time as the same or functionally-equivalent services are re-executed. We believe that these overheads are acceptable for the benefits of fault tolerance that result from our approach.

We have discussed the benefits of our container-based approach in some detail in the body of the paper. In short, it provides a mechanism whereby any service by any provider may be part of a fault tolerance mechanism and that there is no particular fault tolerance mechanism hard-wired into the system. Rather, by defining a recovery policy model, a range of alternative approaches are supported that, in principle at least, may be modified dynamically. Our discussion above reveals that the time required to create a service with embedded fault tolerance is short (assuming that the required service functionality is actually available) and that the overhead of using a container are relatively low.

We have identified the problems with our container-based approach to be:

1. While the system allows for failure of the services used, the proxy service offered by the container remains a potential single point of failure in a system.
2. Only services without ‘unrecoverable actions’ can be transparently embedded in a fault tolerance container.
3. We assume that diverse, functionally equivalent services will be available.

A general problem with any mechanism to support dependable software operation is that the system dependability can be compromised by the failure of that mechanism. Hence, normal practice is to attempt to ensure that the probability of failure of the fault tolerance support mechanism is much lower than the probability of failure of the components used by that mechanism.

In this case, because of the flexibility of the system and the fact that we do not know what services will be deployed in the container, we cannot know for sure whether or not our container is more or less dependable that the services contained. However, our assumption is that the container will be deployed on a computer owned by whoever is the ‘dependability stakeholder’. As we have discussed this could be a service provider, a service client or a trusted third party. In any case, they are in a position to monitor and control the container operation and hence detect and restart the container in the case of failure. This is not necessarily the case for external services.

Of course, it would be relatively straightforward to replicate the container on different servers but this would have the important disadvantage that the application would then have to be made aware of the existence of recovery mechanisms. While this would certainly be required if web services were used for mission or safety critical systems, we believe that a less stringent approach to fault tolerance is required and that the advantages of transparency outweigh any marginal benefits that may arise from container replication.

In principle, services are stateless, atomic entities without side effects. In practice, composed services are more complex and may have significant costs associated with undoing actions that have been completed. For example, consider a service that books a flight and then a hotel. Generally, lower-cost flights have high cancellation costs so if service failure occurs after the flight booking has been made but before the hotel booking has been completed then the costs of undoing the service actions are significant. Compensating actions have been suggested in WS-BPEL to cope with this but it is not clear if these can be used in all cases.

In general, transparent failure recovery from composite services without compensating actions is not possible. If such a service fails before completing its execution, there may be changes to the state of the world outside of that service and recovery by re-executing the same or a functionally equivalent service is impossible because the entry conditions cannot be replicated. Work elsewhere is looking at this issue of dependability of composite services [20] [21] and there may be future opportunities to integrate this with our container model. However, it will always be the case that a large number of services will be atomic services, so our approach will be widely applicable in spite of this limitation.

Finally, we assume that there will, in fact, be diverse services offered by different providers that offer comparable functionality. This extent to which this assumption is justified is currently unclear. Certainly, there may be many different organizations offering a service but these may all depend on some other common service – hence, the diversity is non-existent. For example, UK train operating companies all offer apparently different timetabling services. However, these all rely on the same underlying timetable database service and failure of this service is reflected in failure of all timetabling services. Furthermore, services offered by different providers could, in fact, be hosted on the same third-party server or could rely on the same implementation components e.g. Java libraries.

While this may negate the assumption that fault tolerance can be provided cheaply through redundant, diverse services, it does not represent a fundamental shortcoming of our approach. It requires service providers to provide more information about their services so that service users can decide whether or not these are likely to be diverse.

6 Conclusions and further work

The architecture we have described in this paper addresses an important problem for service engineering: that of ensuring the availability and reliability of a provided service. This can be achieved at low cost, both financially and in terms of development effort. In summary, the additional effort involves creating a recovery 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. As we have discussed, the overhead involved in deploying services in a container is fairly low and within acceptable bounds in most cases.

Current development of the system is focusing on three areas:

1. Extending the range of software fault tolerance mechanisms that are supported by developing new reusable Java components. 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 by Zhang et al. [18]).

2. Extending the tool support to further reduce the costs and effort of fault tolerant service development. We are developing a fault tolerance wizard where the user will be guided towards the selection of a particular fault tolerance policy depending on their application needs and will be able to use this policy and relate services to it without the need to write any XML.

3. Developing an extended set of contexts which will support asynchronous transactions. Currently limited to a message context, we propose having further 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. 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 synchronized. We intend to support context related standards such as WSContext and WS-Transactions.

In our 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. For example, proxy procedures could dynamically discover and bind their own end services if and when service failure occurs or depending on the quality of service offered.

Finally, we intend to utilise our architecture in roles beyond fault tolerance. One possibility is to use 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. Another possibility is to use 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.

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 (Dependability Inter-disciplinary Research Collaboration) project and by the European Commission's Framework 6 Initiative in the SeCSE Integrated project.

8 References

[1] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "Grid Services for Distributed System Integration," *IEEE Computer*, vol. 35, pp. 37-46, 2002.

[2] C. Szyperski, *Component Software: Beyond Object-oriented Programming, 2nd ed.* Harlow, UK: Addison-Wesley, 2002.

[3] H. Kreger, "Web Services Conceptual Architecture (WSCA 1.0)," IBM 2001.

[4] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11-33, 2004.

[5] L. Pullum, *Software Fault Tolerance Techniques and Implementation*. Norwood, MA.: Artech House, 2001.

[6] B. Nuseibeh, "Ariane 5: Who Dunnit?" *IEEE Software*, vol. 14, pp. 15-6, 1997.

[7] B. Randell and J. Xu, "The Evolution of the Recovery Block Concept," in *Software Fault Tolerance*, M. R. Lyu, Ed. Chichester: John Wiley & Sons, 1995, pp. 1-22.

[8] A. A. Avizienis, "A Methodology of N-Version Programming," in *Software Fault Tolerance*, M. R. Lyu, Ed. Chichester: John Wiley & Sons, 1995, pp. 23-46.

[9] L. Moser, M. Melliar-Smith, and P. Narasimhan, "Tutorial on Fault Tolerant CORBA," Carnegie-Mellon University, 2003.

[10] N. Aghdaie and Y. Tamir, "Client-Transparent Fault-Tolerant Web Service," presented at *20th IEEE International Performance, Computing, and Communications Conference*, Phoeniz, AZ., 2001.

[11] M. Marwah, S. Mishra, and C. Fetzer, "TCP Server Fault Tolerance Using Connection Migration to a Backup Server," presented at IEEE International Conference on Dependable Systems and Networks, San Francisco, 2003.

[12] D. Zagorodnov, K. Marzullo, L. Alvisi, and T. C. Bressoud, "Engineering Fault-Tolerant TCP/IP Servers Using FT-TCP," presented at *IEEE. Conf. on Dependable Systems and Networks*, San Francisco, 2003.

[13] C. Ferris and D. Langworthy, "Web Services Reliable Messaging Protocol (WS-Reliability)," 2005.

[14] S. Hwang and C. Kesselman, "Grid Workflow: A flexible failure handling framework for the Grid," presented at *12th IEEE International Symposium on High Performance Distributed Computing.*, Seattle, 2003.

[15] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix," presented at Proc. Usenix Winter 1995 Technical Conf., New Orleans, 1995.

[16] A. Bartoli, R. Jiménez-Peris, B. Kemme, C. Pautasso, S. Patarin, S. Wheater, and S. Woodman, "The ADAPT Framework for Adaptable and Composable Web Services," *IEEE Distributed Systems Online*, 2005.

[17] K. Birman, R. van Renesse, and W. Vogels, "Adding High Availability and Autonomic Behavior to Web Services," *presented at 26th Annual International Conference on Software Engineering* (ICSE 2004), Edinburgh, Scotland, 2004.

[18] X. Zhang, D. Zagorodnov, M. Hiltunen, K. Marzullo, and R. D. Schlichting, "Fault-tolerant Grid Services using Primary Backup: Feasibility and Performance," *presented at Proc. Cluster 2004*, San Diego, CA., 2004.

[19] N. Looker and M. Munro, "WS-FTM: A Fault Tolerance Mechanism for Web Services," University of Durham, Technical Report 02/05. 2005.

[20] V. Kharchenko, P. Popov, and A. Romanovsky, "On the Dependability of Composite Web Services with Components Upgraded Online," *presented at Int. Conf. on Dependable Systems and Networks (DSN '04 - Workshop supplement)*, Florence, Italy., 2004.

[21] F. Tartanoglu, V. Issarny, A. Romanovsky, and N. Levy, "Coordinated Forward Error Recovery for Composite Web Services," *presented at 22nd Symposium on Reliable Distributed Systems* (SRDS), Florence, Italy., 2003.