Transparent Fault Tolerance for Web Services based Architectures

Vijay Dialani, Simon Miles, Luc Moreau, David De Roure, and Michael Luck
{vkd00r,sm,L.Moreau,dder,mml}@ecs.soton.ac.uk

Department of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ UK

Abstract. Service-based architectures enable the development of new classes of Grid and distributed applications. One of the main capabilities provided by such systems is the dynamic and flexible integration of services, according to which services are allowed to be a part of more than one distributed system and simultaneously serve different applications. This increased flexibility in system composition makes it difficult to address classical distributed system issues such as fault-tolerance. While it is relatively easy to make an individual service fault-tolerant, improving fault-tolerance of services collaborating in multiple application scenarios is a challenging task. In this paper, we look at the issue of developing fault-tolerant service-based distributed systems, and propose an infrastructure to implement fault tolerance capabilities transparent to services.

1 Introduction

The *Grid problem* is defined as flexible, secure, coordinated resource sharing, among dynamic collections of individuals, institutions and resources [10]. Grid Computing and eBusiness share a large number of requirements, such as interoperability, platform independence, dynamic discovery, etc. In the eBusiness community, Web Services have emerged as a set of open standards, defined by the World Wide Web consortium, and ubiquitously supported by IT suppliers and users. They rely on the syntactic framework XML, the transport layer SOAP [3], the XML-based language WSDL [2] to describe services, and the service directory UDDI [1].

The benefit of open standards has recently been acknowledged by the Grid Community, as illustrated by three projects embracing Web Services in various ways. Geodise (`www.geodise.org`) is a Grid project for engineering optimisation, which makes Grid services such as Condor available as Web Services [7]. myGrid (`www.mygrid.org.uk`) is a Grid middleware project in a biological setting, which addresses the integration of Web Services with agent technologies [16]. More recently, the Open Grid Service Architecture (OGSA) [9] extends Web Services with support for the dynamic creation of transient Grid Services.

Grid computing is characterised by applications that may be long-lived and involve a very large number of computing resources. Hence, applications need

to be designed with fault tolerance in order to be robust. As a result, the Grid community, and more generally, the distributed computing community have devised multiple algorithms for fault tolerance. However, the Web community has not focused on this aspect, and therefore, there is no standard way to develop fault-tolerant Web Services.

It is this specific problem that we address in this paper. Our approach may be summarised as follows: implementors of a Web Service have to implement an interface (e.g. checkpoint and rollback); the architecture dynamically extends the service interface (published as a WSDL document) with methods for fault tolerance; applications making use of different Web Services have to declare their inter-dependencies, which are used by a fault-manager to control fault recovery; an extension of the SOAP communication layer is able to log and replay messages.

The specific contributions of this paper are: *(i)* The design of an architecture for fault-tolerance of Web Services which supports multiple algorithms for fault-tolerance. *(ii)* The specification of the interfaces between the different architecture components. *(iii)* An overview of our implementation. The paper is organised as follows. In Section 2, we summarise some of the techniques for fault tolerance which we support in our architecture, while in Section 3, we present the Web Services stack. We describe our architecture in Section 4, and its implementation in Section 5 and we conclude the paper in Section 6.

2 Fault Tolerance Background

Before expanding on our design, we present a brief introduction to fault tolerance for distributed systems. This is intended as an aid to understanding terms used later in the paper, and not as an extensive survey.

Fault tolerance is the ability of an application to continue valid operation after the application, or part of it, fails in some way. Such failure may be due to, for example, a processor crashing. In order for an application suffering a failure to continue, the state of processes, and the data they use, must be returned to a *previous consistent state*. For example, object data may return (*rollback*) to previous values if the current values are lost, and processes may return to a state in which a message is re-sent, if the previous attempt apparently failed.

In order to return to a previous consistent state, an application must record a replica of its previous state. The entire state of a process can be copied using a *checkpointing* mechanism, or only the incremental changes to the process state using a *logging* mechanism. Both methods can be used to rollback to a previous valid state [8].

Fault tolerance becomes considerably more difficult in distributed applications, made up of several processes that communicate by passing messages between themselves. One process may fail without the other processes being aware of the failure. This can lead to the state of the application as a whole (the global state) being inconsistent. An application is in a *globally consistent state* if whenever the receipt operation of a message has been recorded in the state of some

process, then the send operation of that message must have been recorded also [15]. It is the aim of a fault tolerance mechanism for distributed applications to keep an application to a consistent global state, or return to the last known consistent state (also known as *maximal state*) in case of failure.

Fault tolerance mechanisms should have transparency, low overhead, portability and scalability [19]. Transparency implies that there exists a mechanism such that applications implemented using it can largely ignore processes failing or recovering, as this will all be dealt with by the mechanism. It is important that transparency exists so that both the developers' burden is eased and the fault tolerance mechanism can be replaced by another without the rest of the application requiring modification. The requirement for low overhead, portability and scalability can lead to a choice in the fault tolerance mechanisms to apply. Fault tolerance can also be achieved by using fault tolerance Object Replication techniques, e.g. [13].

Checkpointing can also be performed in a variety of ways in distributed applications. *Consistent* or *synchronous* checkpointing involves all processes being forced to *globally synchronise* before the state of all the processes is recorded [12]. Global synchronisation means that all processes are in a state in which they have processed all received messages and are blocked from sending any messages [12]. As blocking processes may reduce the speed of the application, consistent checkpointing may not always be the preferred means of fault tolerance. On failure, all processes using consistent checkpointing rollback to the last global checkpoint. A *quasi-synchronous* approach is suggested by Manivannan and Singh [15], where, rather than requiring global synchronisation, processes force each other to checkpoint at almost the same time through sending messages.

An alternative to consistent checkpointing is *independent* or *asynchronous* checkpointing. In this case, each process records its own state without attempting to coordinate with other processes, so potentially avoiding the overhead of global synchronisation. However, communicating processes may depend on each other, i.e. require that they are in certain states. The rollback of one process, on failure, may require that other processes also rollback to previous checkpoints. It is then possible that these rollbacks will require the original process to rollback even futher to attempt to reach a consistent global state. This repetition of rollbacks can lead to a *domino effect* where each process must rollback many times to reach a consistent global state, losing a lot of processing that has occurred without failure [20].

If the state of one process becomes invalid by the rollback of another, we consider there to be a *dependency* of the former process on the latter. When a rollback (or checkpoint) should take place on multiple processes, a fault tolerance mechanism should ensure that it does not create extra dependencies. In order to achieve this, the mechanism can employ a *two-phase commit*, in which processes are first put into a blocking state to prevent messages being sent and new dependencies forming, and then later rolled back (or requested to checkpoint) at an appropriate moment.

Independent checkpointing mechanisms deal with process dependency in two ways. *Pessimistic* independent checkpointing [19], requires that each process logs the changes since the last checkpoint after sending or receiving any message. As dependencies between processes are only due to messages passed between them, this ensures that rollback of one process to the previous checkpoint will not affect dependent processes. *Optimistic* independent checkpointing requires that dependencies are explicitly recorded somewhere in the system, so that on rollback of a process, dependent processes will be informed appropriately and possibly also rolled back [6, 11, 20]. The pessimistic approach places more restrictions on a process' autonomy in checkpointing and may require more checkpointing than optimistic approaches. Optimistic mechanisms will have more overhead in rollback, on the other hand. However, it should be noted that no single mechanism is universally applicable. The suitability of algorithms differ for each *application type*, namely that there exists a different set of algorithms for batch processing, shared memory and MPI based applications. In this paper we restrict ourselves to discussions on fault tolerance requirement of Web Services architecture.

3 Web Services

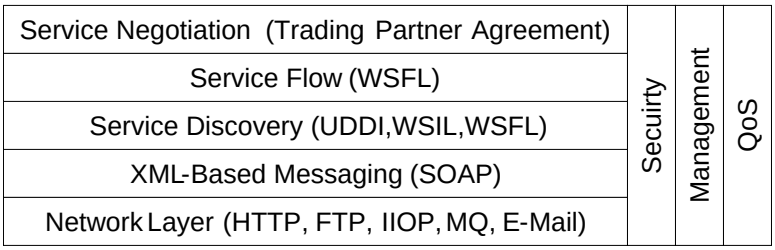

Fig. 1. A generalized Conceptual Web Services Stack

The World Wide Web is more and more used for application to application communication. The programmatic interfaces made available are referred to as Web Services.[http://www.w3c.org/2002/ws/]. To ensure interoperability between different architectures, the Web Services architecture describes standards for definition, discovery, binding and communication between services. A service provides a set of application functionality through a bound and advertised interface. This architecture provides an abstraction over the implementation of services.

Service discovery mechanisms such as *Universal Description, Discovery and Integration* (UDDI), aid in discovering services, statically or dynamically bound. To facilitate binding, services describe their behaviour by using a description language, such as WSDL[17]. However, there exists no explicit information about

its lifetime and instance creation, management policy differs across implementations.

The Web Services Stack : A number of Web Services implementations exist, each with a proprietary Web Services stack. Such stacks vary in the way that they gel or interact with legacy systems and proprietary technologies. A generalized conceptual Web Service stack is represented in figure 1.

The network layer, messaging layer and the service description layer have been standardized to ensure interoperability. SOAP is supported as the de-facto XML-messaging protocol for most of the Web Service implementations. Detailed discussions on SOAP protocol and WSDL are described in [4], [2] respectively. The "vertical layers" describe attributes of the framework and must be addressed at each level. At present, security, management and QoS are the widely accepted system attributes.

Error Handling in Web Services : Different layers in the conceptual stack employ different types of error handling. At the description layer, WSDL provides a mechanism by the way of `<wsdl:fault>` for applications to specify the error characteristics. This is similar to the way we define exceptions raised by the methods in a Java interface. Similarly the underlying SOAP messaging layer, provides a `<soap:fault>` for applications to communicate the error information. The error mechanisms of SOAP and WSDL help support errors raised by an application, but no mechanism exists for handling framework failures and system errors.

Service Lifetime Management : Web Services differ from usual message-based distributed systems. SOAP omits features often found in messaging systems and distributed object systems [4], such as: *(i)* distributed garbage collection; *(ii)* boxcarring or batching of messages; *(iii)* objects-by-reference (which requires distributed garbage collection); *(iv)* activation (which requires objects-by-reference). As Web Services do not support explicit activation or deactivation of services, it becomes difficult to have any lifetime management control. Most common implementations [5] use time-based expiry mechanism for controlling the underlying resources.

Fault Tolerance for Web Services : In service-based infrastructures, a single process may be part of multiple applications. Therefore, rollback cannot be initiated using the standard fault tolerance mechanisms mentioned earlier. We propose fault-tolerance as one of the vertical layers of the Web Services stack. In our earlier discussion, we described the need of fault-tolerance for service-based infrastructure. In the rest of the paper, we discuss the special requirements of each layer.

4 Architecture

In figure 2, we present an overview of the fault tolerant system, as applied to the Web Services architecture. The top of the figure shows various components of

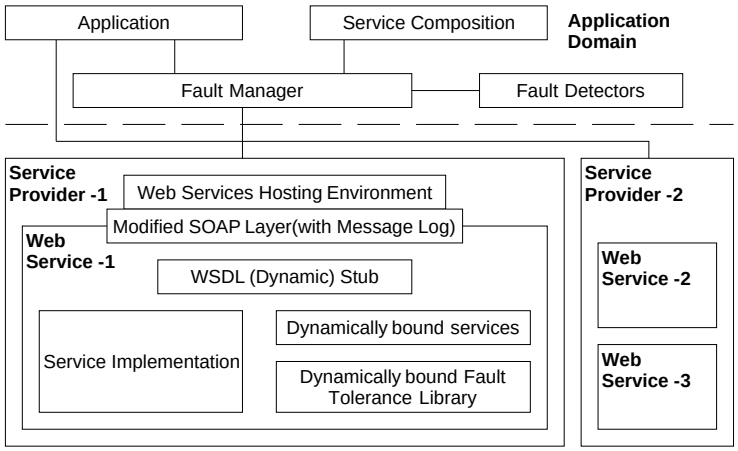

Fig. 2. Architecture for Fault Tolerant Web Services

the fault tolerance infrastructure that are specific to an application instance and are location independent. The lower half represents modifications to the existing hosting environment for services. The modifications in the latter case can be classified into a set of changes to the messaging layer (refer to next section for a detailed description) and a set of interfaces supported by individual services. Henceforth, we refer to the upper half as the *application layer* and to the lower half as the *service layer*. In general, the overall framework provides the capability to:

1. Detect a fault or failure,
2. Estimate the damage caused and decide on the strategy for recovery,
3. Repair a fault, and
4. Restore the application state.

The framework differs from traditional frameworks, such as CORBA [18] as it employs a two-pronged strategy to recover from a fault, namely the *local recovery mechanism* and the *global recovery mechanism*. The context of a local recovery is restricted to recovery of an individual service instance, while global recovery applies to the entire application. The local recovery mechanism tries to revive the service instance with minimal or no intervention by the global recovery mechanism. A local recovery mechanism escalates the failure notification to the global recovery mechanism in case of its failure to recover the fault locally. The architecture imitates an hour glass model to restrict the dependency between the two layers to a minimal set of interfaces, for co-ordination between the two layers.

The application layer assumes that an application instance aggregates a set of service instances to provide the overall capability for the application. The

concept of service aggregation, also known as service composition, is central to the definition of a Virtual Organisation (VO)[10].

However, our definition of service composition is not restricted to VOs and can be extended to service composition expressed by the way of workflow specification, e.g. WSFL [14] or X-LANG [21] . The composition of a service may be created statically at design time or can be created dynamically by using negotiation techniques, enactment description of WSFL, or any other technique. A detailed discussion on negotiations and composition of services is outside the scope of this paper. The application layer assumes that there exists a description of service composition that it can refer to for obtaining a list of collaborating services. The application layer can be initialized by the application instance or by enactment of a composition. The instantiation data can be held within the composition definition or it can be provided explicitly during creation. The application layer implements a set of key components, namely:

1. *Application*: It uses the services in a composition to provide the overall capability. An application can directly interact with the global fault manager(refer to the definition below) or allow the application framework to interact on its behalf.
2. *Global Fault Manager*: A coordinator that interacts with the applications or framework and the underlying services to implement a fault tolerant system. A fault manager is responsible for monitoring, fault diagnosis and checkpoint and rollback co-ordination; it may be central or distributed.
3. *Service*: An entity that is bound by its interface definition, usually a WSDL description, and executes as an independent process or within the process of Web Services Hosting Environment [5].
4. *Fault detector*: A fault detector detects a change in the perceived ideal environment and uses software interrupts to the fault manager to notify of any failure. In addition to providing the context for the fault, it may also provide behavioral override, allowing applications to extend the fault notification mechanism.

A global fault manager interacts with a set of services specified in the service composition. Each of the underlying services needs to support a set of interfaces to enable communication between the local and global fault managers. A local fault manager coordinates independent checkpointing and rollback of an individual service; it monitors the service and supports the fault detector interface for creating fault notifications. The local fault manager interacts with the messaging layer to initiate a blocking or non-blocking recovery, with or without the replay of messages. The global fault manager relies on a set of fault detectors to send a fault notifications. An application can register a custom list of detectors in addition to those supported by the individual services.

Our modified SOAP layer provides the message logging, message replay and a capability to acknowledge either the receipt or the processing of a message. It provides interfaces for interaction with the global fault manager and local fault managers. Modifications to the layer allow the application framework to maintain a log of messages and also to selectively suspend the communication

between the services. They also enable the framework to isolate a service instance from the rest of the system during a local recovery. In addition, the ability to suspend communication helps rollback by providing the ability to isolate the affected set of services. Modifications to the SOAP messaging layer enable us to support both fault tolerance by message-based checkpointing and rollback, and fault tolerance by object replication. In the following section, we describe interactions between the various components in the framework for implementing message based checkpointing and rollback. Later in our discussion, we describe how the framework could support fault tolerance by object replication.

5 Implementation

IBM WSTK-3.0, Apache SOAP, IBM Web-Sphere Application Server, IBM Web Hosting Environment were used to implement our proposed framework. Our implementation provides a modified SOAP layer, different libraries to initialise the application framework, and a set of plug-ins for various application types. The Application framework allows an application to specify a service composition; we support both design-time and run-time compositions of services. In its current implementation, the framework assumes service compositions to be static and immutable; however, the framework can be modified to allow dynamic compositions, to complement UDDI and WSIL support for dynamic discovery and binding of services. The Application layer can be implemented to be a part of the application instances execution space or be a Web Service by itself. In either case, the application framework creates and initializes a global fault manager. The global fault manager accepts fault tolerance mechanism specific parameters and the application type as immutable parameters; it uses service composition to discover and establish contact with the local fault managers; it performs a two-phase commit checkpoint operation to coordinate the checkpointing activity across the service instances.

The local fault manager is implemented as a set of libraries that can be bound dynamically to the service code. The local fault manager interacts with the modified SOAP layer to control the flow of messages during the recovery as the mechanism used by it may or may not support non-blocking checkpoint and/or rollback. In case of a failure, the local fault manager categorizes the fault, and then tries to recover the fault. In certain cases, it may be possible to recover the service locally and rollback to the current state by replaying messages. In case a full recovery is not possible the local fault manager tries recovering to a maximal state and escalates the fault notification to the global fault manager.

On notification, the global fault manager initiates a roll back by notifying the affected services. The dependency set for recovery can be provided by the application. Additionally, the fault detectors can provide a dependency set for the current fault: the provision is specifically useful in case of compositions that use different protocols or support different end-points. For example, service composition may consist of a set of Intranet and Internet services; services within an Intranet may use IIOP for inter-service communication and connect to the Inter-

net using a SOAP layer. The rollback is also implemented as a two-phase commit operation. The framework ensures loose coupling, by supporting different fault mechanisms for local and global fault managers. In addition, to checkpointing and rollback mechanism for fault tolerance, Object Replication can also be used to improve fault tolerance of applications. One of the possible ways is to enable the Web Services hosting environment to create a set of redundant services and define a mechanism for active or passive replication of services, client redirection. However, a detailed discussion on replication based fault-tolerance is beyond the scope of present discussion.

6 Conclusion and Future work

We have successfully conceptualised and implemented a fault tolerant architecture for Web Services, without affecting interoperability of existing services. The framework demonstrates a method of effectively decoupling the local and global fault recovery mechanisms. It provides a capability for monitoring the individual service instances as well the service hosts. The algorithm independence and support for different application types allow us to provide fault tolerant capabilities to Web Services that internally employ different programming models. Dynamic varying composition of services is an issue that needs to be addressed. However, much depends upon the composition schemes that will evolve from research in Web Services.

7 Acknowledgement

This research is funded in part by EPSRC myGrid project (reference GR/R67743/01) and EPSRC combichem project (reference GR/R67729/01)

References

[1] Uddi standards. http://www.uddi.org.
[2] W3c wsdl spec. http://www.w3c.org/TR/wsdl.
[3] Xml protocol working group. http://www.w3c.org/2000/xp/Group/.
[4] W3c soap standards, 2001.
[5] Web services hosting technology. http://www.alphaworks.ibm.com/tech/wsht, December 2001.
[6] B. Bhargava and S. Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems–an optimistic approach. In *Proceedings of the 7th IEEE Symposium on Reliable Distributed Systems*, pages 3–12, 1988.
[7] S. J. Cox, M. J. Fairman, G. Xue, J. L. Wason, and A. J. Keane. The Grid: Computational and Data Resource Sharing in Engineering Optimisation and Design Search. In *IEEE Proceedings of the 2001 ICPP Workshops*, pages 207–212, Valencia, Spain, September 2001.
[8] E. N. Elnohazy, D. B. Johnson, and Y.M. Wang. A survery of rollback-recovery protocols in message-passing systems.

[9] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The Physiology of the Grid — An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Argonne National Laboratory, 2002.

[10] Ian Foster, Carl Kesselman, and Steve Tuecke. The anatomy of the grid. enabling scalable virtual organizations. *International Jounral of Supercomputer Applications*, 2001.

[11] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proc. 7th Annual ACM Symp. on Principles of Distributed Computing*, pages 171–181, Toronto (Canada), 1988.

[12] M. Frans Kaashoek, Raymond Michiels, Henri E. Bal, and Andrew S. Tanenbaum. Transparent fault-tolerance in parallel orca programs. In *Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems III*, pages 297–312, 1992.

[13] Sean Landis and Silvano Maffeis. Building reliable distributed systems with CORBA. *Theory and Practice of Object Systems*, 3(1):31–43, 1997.

[14] Prof. Dr. Frank Leymann. Web services flow language. http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf, May 2001. Member IBM Academy of Technology, IBM Software Group.

[15] Manivannan and Singhal. Comprehensive low-overhead process recovery based on quasi-synchronous checkpointing.

[16] Luc Moreau. Agents for the Grid: A Comparison for Web Services (Part 1: the transport layer). In *IEEE International Symposium on Cluster Computing and the Grid*, Berlin, Germany, May 2002.

[17] Judith M. Myerson. Web services architectures. http://www.webservicesarchitect.com/content/articles/myerson01.asp, January 2002.

[18] OMG, http://www.omg.org/docs/formal/01-12-63.pdf. *Fault Tolerant CORBA*, December 2001. Version 2.6.

[19] D. J. Scales and M. S. Lam. Transparent fault tolerance for parallel applications on networks of workstations. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 329–341, San Diego, CA, USA, 1996.

[20] Robert E. Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[21] Satish Thatte. 'xlang'- web services for business process design. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm, 2001.