An Adaptable Fault-Tolerance for SOA using a Peer-to-Peer Framework

Stephen Hall, Gerald Kotonya
Computing Department
University of Lancaster
United Kingdom
{s.hall, gerald}@comp.lancs.ac.uk

Abstract

The onset of Service Oriented Architectures (SOA) in domains like trading and banking has considerably heightened the need for dependable system operation. Late binding to services in business-to-business operations pose a serious problem for dependable system operation as it delegates the decision to trust a service to an external agent. However, it is impossible to guarantee that any service is 100% fault-free due to failings in hardware, software and human error. This means that fault tolerance remains the most practical way to address the problem. Unfortunately, there is currently no standard way to achieve this in SOA. This paper describes a novel adaptable fault tolerance framework that overlays a peer network formed by JXTA technology protocols to address this problem. We have adopted a layered approach by incrementally adding protocols and supporting the code infrastructure. The framework is implemented exclusively in Java and XML to ensure cross platform compatibility.

1. Introduction

Service-oriented architectures (SOA) consist of geographically and topographically distributed services that can be combined using open messaging standards to form mission critical systems [15]. New generation of service-oriented architectures, such as web services, pose a serious problem for dependable system operation [16, 19] because they promise late binding (i.e. discovery and invocation during runtime). Late binding delegates the decision to trust a service to an external software agent (i.e. the consumer must assume the service is dependable). However, it is impossible to guarantee that any service is 100% fault-free due to failings in hardware, software and human error [19].

One way to address the problem is to incorporate fault tolerance (FT) within the service orchestration and provision environment. Fault tolerance within distributed computing is attained by the application of interaction patterns over a series of replicated components with the goal of maintaining the operation of the system in the presence of faults. Improvements in dependability are qualified by two properties (1) availability is the uptime of a service, often expressed as a percentage or “nines” (four nines = 99.99%) metric; (2) reliability is a probabilistic measure that a service will not deviate from its specified operation. Unfortunately, no similar standards exist for service-oriented architectures [7].

FT is the continual satisfaction of three requirements, safety, liveness and performance [1, 2, 3, 5]. We use the term safety, in common with [5], rather than reliability (indicating continuity of service) because to ensure no consequences for the end user or environment [2]. However, by ensuring safety we recognise that we desire continual reliability and availability. We express safety in terms of \( f \) and \( n \), the maximum number of faulty replicas that can be tolerated, and the total number or replicas respectively. For example, for Byzantine FT \( f = (n-1)/3 \) [5, 9]. Each function is calculated depending on factors such as failure mode and synchrony. Liveness is the guarantee that clients will eventually receive replies given that safety is assured [5]. Performance is defined as an acceptable degradation in QoS from a non-FT equivalent system, the primary example being response latency. We observe performance rather than make assertions about it because of the general complexity of FT patterns.

Our solution has been to develop tailorable and scalable FT framework to process SOA transactions. Our framework is built on a P2P overlay network using JXTA to leverage its properties of distribution and scalability [18]. Unlike existing frameworks we focus on adaptability requirements with the goal of producing a FT framework that tailors itself to the problem based on monitored metrics.

In this paper we demonstrate how our FT framework caters for crash, Byzantine and common-mode failure modes in a SOA [1, 6, 13]. Crash failures are the most common and benign of failures since they
are detectable. We are able to divide crash failures into
two further groups, signalled and fail-silent [2, 3]. The
difference between them is that signalled failures notify
all interest parties, and then halt, thus requiring
no detection services. Fail-silent, however, requires
failure detectors. Byzantine failures are characterised
by the Byzantine Generals Problem [9]. These are the
least common and most severe of failures. Byzantine
failures occur when services act arbitrarily and
inconsistently. They commonly manifest themselves as
commission (computational) errors but could result in
omissions or timing failures. Byzantine failures cannot
be ascribed to solely malicious origins [3] but this is
the most likely source [5]. Solutions to Byzantine FT
(BFT) are derived from [9]. Common-mode failures
occur when the same fault is activated in multiple
services. This can occur in mobile-agent solutions
where the same software is distributed to many places.
The natural solution is diversity, having different
software operating on replicas performing the same
task. In this case replicas become variants [19].
Diversity is achieved by N-Version Programming
(NVP) techniques [1].

The rest of this paper is structured as follows.
Section 2 reviews prior SOA based FT frameworks.
We introduce our framework in section 3 describing
our layered approach to achieving our goals. In section
4 we introduce a web service based stock trading test
case with which we are to evaluate our framework.
Section 5 describes how we provisionally evaluated or
framework and the results obtained. Finally, we
conclude and describe further work in section 6.

2. Background

There is currently no FT standardization in SOA
with bodies, such as OASIS concentrating on reliable
messaging namely WS-Reliability [7]. We have
identified several FT frameworks that operate with web
services, the model SOA. We can classify these
frameworks with regards to the failure mode. Some
frameworks such as WS-BPEL FT [7] incorporate FT
into the service composition, however all the
frameworks reviewed in this paper treat FT as a
separate concern.

FT-SOAP [10] and Fault Tolerance for web services
(FAWS) [8] are examples of crash FT frameworks.
They both consist of a group of replica SOAP services
that are managed by external components. These
services can be replicas or variants thus providing
common-mode FT. A dedicated routing proxy service
is used by FAWS to map messages to actual services.
FT-SOAP has hooks in the SOAP engines themselves
to redirect messages to correct services. Failure
detection is by an external notifier that informs
replication management decisions. Both frameworks
consist of dedicated components that represent single
points of failure within the system.

Resilient Web Services Infrastructure (RWSI) [14]
is another crash FT that is built on the Chord P2P ring
overlay network. Here services are deployed to the
network rather than to individual hosts so that host
failure results in the service being moved to another
host. This framework specifically addresses single
points of failure by treating services as mobile agents.
RWSI is also the first framework to achieve scalability
since service discovery is by P2P protocols.

Unfortunately, since a RWSI service is replicated
across multiple hosts the framework is susceptible to
common-mode failures.

Web Service-Fault Tolerance Mechanism (WS-
FTM) is a framework that is designed to capture the
NVP pattern to achieve Byzantine FT [11]. It is an
extension to the Java based Apache Axis SOAP
Engine. We contend that WS-FTM is not truly
Byzantine FT because it relies on a single quorum that
cannot cater for faulty clients or compromised, out of
order, messages. In addition WS-FTM makes
underlying synchrony assumptions about safety
delivered by the base, Apache Axis, uses an RPC style
messaging system. The framework is built on the
assumption of variant services, with the results
combined by a voting quorum. Thus WS-FTM is
actually a common-mode FT framework. Looker et al
on the number of replicas.

THEMA is a dedicated BFT framework [13]. It is a
demonstration of the application of CLBFT algorithms
[5, 6] in a web service SOA. Unlike earlier frameworks
it makes no assumptions of synchrony to achieve
safety. THEMA uses the BASE/CLBFT libraries [6]
on clients, services and any supplementary services to
convert normal service requests into BFT requests.
There are many complex interactions involved in
CLBFT thus performance is a concern. Meredith et al
[13] demonstrate that the latency of a BFT case is
always approximately twice that of a baseline case
irrespective of message size. The CLBFT algorithm
provides safety despite crash failures since the liveness
property is always maintained [5]. CLBFT does not
cater for common-mode failures since it requires
service replicas.

As far as we are aware there is currently no SOA
based FT framework that addresses safety in all failure
modes. Liveness is only explicitly maintained by the
THEMA framework with the others relying on the
underlying synchrony assumptions of the SOA
environment. Performance is not addressed by the any
frameworks except WS-FTM [11] and THEMA [13],
these papers give latency figures against baseline non-
FT equivalents. RWSI provides scalability through its JChord P2P protocols. Finally, no prior framework addresses FT adaptability.

3. Web Service Peer Based Fault Tolerance

We have created a framework, WS-PBFT, to make web service SOA applications fault tolerant. Like the other frameworks WS-PBFT sits between the SOA client and service. This is to create transparent FT. The WS-PBFT architecture is shown in Figure 1. WS-PBFT exclusively supports SOAP messaging and service represented in WSDL, the two key Web service SOA standards.

To make the framework decentralized and scalable, it uses an underlying P2P network as in RWSI [14] to distribute services. However, unlike RWSI, WS-PBFT FT models are also distributed amongst peers to ensure that no service or model acts as a single point of failure to the framework.

Web services can be executed inside the framework execution space or accessed via a proxy. To ensure WS-PBFT is cross-platform it is developed in Java, all protocols and storage is in XML to allow close integration with web services. Our framework is built using a layered approach on a base of a P2P network to leverage the properties of decentralization and scalability.

3.1. P2P Layer (JXTA)

To take advantage of the emergent properties of P2P networks we have built our framework on the JXTA programming environment. JXTA is an industry leading set of XML protocols and associated Java bindings to build P2P applications [4]. We have adopted JXTA instead of alternative network paradigms, such as Chord, because of the useful abstractions it affords.

A primary goal of JXTA is pervasive messaging allowing peers to be accessed on otherwise hidden networks. Every entity in a JXTA environment, real or abstract, is represented by an advertisement. Advertisements are stored and distributed by the Rendezvous protocol [4]. Any peer within the network can be nominated as a rendezvous; peers can connect to a rendezvous to retrieve advertisements. In addition rendezvous can act as relays to peers that are hidden behind firewalls or network translation. WS-PBFT distributes a list of rendezvous peers with every installation allowing the network to dynamically form.

3.2. Platform Layer

Though it exposes an API to create and manipulate P2P networks, JXTA is passive and does not model the overlay network effectively. It is, therefore, impossible to integrate FT models directly with JXTA. We have addressed this problem using a Java based platform layer. This breaks the API into a set of API objects for managing different aspects of the WS-PBFT platform. These APIs are described in Table 1.
<table>
<thead>
<tr>
<th>Group</th>
<th>Peer Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peer Group</td>
<td>Is responsible for collecting and maintaining information about all peers in the group. JXTA is accessed through the peer API.</td>
<td></td>
</tr>
<tr>
<td>Engine Group</td>
<td>Supports the model layer described in section 3.3. It creates an environment in which FT models can be executed by routing events.</td>
<td></td>
</tr>
<tr>
<td>Model Peer/Group</td>
<td>Validates and stores SCXML FT models.</td>
<td></td>
</tr>
<tr>
<td>Service Peer/Group</td>
<td>Validates and stores WSDL documents.</td>
<td></td>
</tr>
<tr>
<td>Service Bridge Group</td>
<td>Discovers services within the framework, routing messages to them. Uses a registry to discover endpoints.</td>
<td></td>
</tr>
<tr>
<td>Model Bridge Group</td>
<td>Discovers models within the framework, routing messages to them. Uses registry to discover endpoints.</td>
<td></td>
</tr>
<tr>
<td>Metric Group</td>
<td>Gathers information about messages in the framework.</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Platform APIs

Both the Model Bridge and Service Bridge APIs rely on a XML-based registry to discover and access models/services. The registry is periodically updated to maintain a consistent group-wide view. XPATH is used to query each registry to provide a list of models/service endpoints given an event or message identifier. Figure 2 shows a screen shot of the discovery registry. This approach ensures models and services are transparently accessible from anywhere in the framework.

Generally, messages alternate between the model and service bridge. The framework does not require models to function because messages are routed directly between services if no matching model endpoints can be discovered. This means we only need to intercept certain messages, therefore increasing efficiency.

The metric API gathers ongoing information about the transactions taking place in the frameworks. A monitoring aspect gathers selected messages, time stamping them. Every time interval $t$ the metric aspect takes a set of messages logged by the monitor and computes the latency, throughput and failure rates.

This facility can gather safety $(f)$ and the maximum number of faulty replicas that can be tolerated $(n)$ recorded by FT models.

3.3. Model Layer

To support complex FT operations such as those performed by THEMA [13] we have developed an asynchronous event-driven distributed state machine. The state machine is purely asynchronous to prevent unintentional synchrony assumptions. Its operation is represented by FT models. Our state model is based on State Chart XML (SCXML). SCXML has the following advantages: Simplicity; it is XML; W3C will soon support it as a standard; there are no predefined semantics for its actions; the Apache commons project provides a SCXML engine. Figure 3 shows a graphical representation of simple SCXML FT models. In this case we can see active and passive FT patterns.

Events are messages passed between models and services that wrap SOAP messages and context identifiers. The names of events, for example `message.traderRequest`, are used by a model to trigger transitions. These transitions trigger SCXML actions and facades to Java operations. Actions typically forward messages to the model or service bridges APIs, these in turn locate models or services somewhere in the framework, routing events onwards.

The engine API supports multiple instances of SCXML FT models in multiple Apache commons engines. To handle the many models concurrently.
executing they are divided into contexts. Each event has a set of contextual identifiers that the engine API uses to locate model instances that can process that event. The engine API contains two contexts. A session context correlates a set of interactions with one client.

The FT models executed by the framework are organized by pattern, as shown in Figure 4. The four basic models are passive for crash FT, inspired by Recovery Block [17], active with voting for common-mode failures, and finally CLBFT for Byzantine failures. However, there are many other factors, motivated by safety and performance requirements.

3.5. Service Layer

The service layer maintains a set of services described by WSDL. A description may include a reference to an implementing Java class included in the framework otherwise the service is treated as though accessible through the proxy. Deployed services are loaded and held by the service bridge API. The framework is able to operate without any models because events are routed back to the service bridge if no matching model is found in the registry.

3.6. Adaptability Layer

To address the adaptability requirement the framework includes a specific layer. In response to results from the metric API, this layer changes the state of the model discovery registry by deploying specific models for a model class, where each class is a collection of events names that will be handled by a model. Metrics include latency, failure rates and, \( n \) and \( f \), in addition to the number of services available.

3.7. Value Added FT

A number of FT facilities are available to the model layer:

- **Log-based Recovery.** Check pointing is not supported in our framework since it is SOA application specific. However, all events are logged by the engine API, in contexts, and can be replayed. In addition all FT models instances can be run sand-boxed to restore their state.

- **Authenticated Messaging.** To prevent forgery messages can have digital signatures [5]. This method uses PK cryptography to encrypt all messages. A better performing method involves two nodes authenticating each other once and generating message authentication codes (MAC). Our framework supports authentication with MACs because peers exchange keys with all other peers when they join the network, they generate MACs at the same time. MACs are used with the CLBFT algorithm [5, 6].

- **Fail-silent Detection.** The discovery registry for models/services always reflects the current state of peers and their deployments. If a peer fails then that information is reflected with an update period \( t \). To accomplish this each peer broadcasts its portion of the registry every \( t \). Each peer then rejects any portion not updated. A peer cannot, therefore, fail silently.

- **Partial Synchrony.** We have implemented a liveness API that periodically generates timeouts when invoked by the Start Timeout action (In Figure 3). This partial synchrony ensures that messages are not lost.

4. Test Case – SOA Stock Trading Example

To evaluate the WS-PBFT framework we have created a non-trivial SOA stock trading example. The example has several advantages. Firstly, it is a real-world example with stakeholders that require safety and performance, justifying a FT framework. This case offers genuine diversity; it uses competing stock brokers providing services on-line. Lastly, it provides a good basis for comparing our WS-PBFT with other FT frameworks that have published similar SOA examples, including FAWS [8], FT-Grid [20], and WS-FTM [11].

Our test-case comprises four services:

- **Banking Service.** This service provides functionality to transfer funds between a trader and a broker, it is modeled on the UK based BACS service but with instant transfers. It is assumed to be safe (of
an acceptable standard), and cannot be replicated like a real world service. To ensure it can be used with BFT algorithms it acts deterministically. For security all interactions with this service are synchronous and secured by SSL.

Broker Service. The broker service is provided by different sources allowing for diversity. The interface is asynchronous. It provides stock quote and trade functionality by interaction with a trading service.

Client. The client acts as an asynchronous service for responses from the trading service. It is responsible for starting a session with the trading service and performing financial transactions with the bank.

Trading Service. This asynchronous service intervenes between the client and broker service(s). It takes a trade request from the client and finds an appropriate deal from a set of brokers and starts the trade. It does not interact directly with the banking service for security reasons. Instead it requests the client to transfer funds via the banking service.

Figure 5. SOA Stock Trading Example

A SOA based stock trading scenario is integrated with the WS-PBFT framework as follows: The bank service is hosted separately from the framework and is accessed via a framework proxy. The trading and broker framework services are deployed in the application space of WS-PBFT. External Web services are accessed by the broker service providing genuine stock quotes, for example Yahoo stock ticker. Finally, the client service is a framework service that provides a facade to the WS-PBFT client API. The client API is a set of XML models containing test scenarios to be processed by the system. These repeatedly run at a selected rate to drive the framework evaluations.

5. Evaluation

The model for evaluation consists of six aspects:

Test case – Defined in the previous section.

Test bed – In this paper our test bed consists of 4 test machines that host the scenario services and framework models. An extra two machines are used as the test driver and metric monitor. The test driver emulates a client service. In this paper the test scenarios are limited to the normal case operation shown in Figure 6. We intend to expand the test bed to 10 test machines for comprehensive BFT testing.

Configurations – How the web services and FT models are deployed. These are defined in Table 2. Configurations can have variations such as message distribution and liveness settings.

Metrics – Consisting of input/output rate, latency and failure rate. The Metric API is shown in Figure 6.

Injection Campaigns – Describes the faults that are injected into the framework, defined in Table 3.

<table>
<thead>
<tr>
<th>Layout</th>
<th>Trader FT</th>
<th>Broker FT</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Baseline</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>B</td>
<td>Framework</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>C</td>
<td>Framework</td>
<td>Passive</td>
<td>None</td>
</tr>
<tr>
<td>D</td>
<td>Framework</td>
<td>Passive</td>
<td>Passive</td>
</tr>
<tr>
<td>E</td>
<td>Framework</td>
<td>Active (First Passed Post)</td>
<td>None</td>
</tr>
<tr>
<td>F</td>
<td>Framework</td>
<td>Passive</td>
<td>Passive</td>
</tr>
<tr>
<td>G</td>
<td>Framework</td>
<td>Passive</td>
<td>Active (Quorum)</td>
</tr>
</tbody>
</table>

Table 2. Configurations

In this paper the set of configurations is limited due to lack of space and the fact that our evaluation is ongoing.

Table 3 shows a limited set of fault injection routines that constitute the injection campaign.
5. Results

We present two sets of results, represented by Figure 7 and 8 respectively: Performance under a constant load, failure rate under injection campaigns (at load of 60 trans/m) including lost messages.

All configurations show acceptable performance until the higher loads. This result is due to excessive load on 4 machines. We expect to improve these results (especially for 600 trans/m) by having a larger test bed and spreading the load.

6. Conclusions and Further Work

We have demonstrated that WS-PBFT can integrate Web services and FT models, encoded as SCXML state machines, to provide effective FT. The framework is also able to drive and monitor results making it an effective evaluation tool. We have combined JXTA and services to make service discovery transparent, demonstrating late binding in keeping with the SOA ethos.

This paper presents only a subset of the evaluation model due to space and time considerations. We intend to improve WS-PBFT by making having a larger set of models and configurations. We expect performance gains with a larger test bed. Nonetheless we believe the framework is a significant research contribution to FT in SOA. Our current research focus is on BFT scenarios for the evaluation framework. Once we have...
sufficient models as discussed in section 3.4 we will evaluate their performance and compile an adaptability layer meta-model to satisfy those requirements.

7. References


