A Differentiation-Aware Fault-Tolerant Framework for Web Services

Gerald Kotonya[1] and Stephen Hall[2]

[1] Computing Department, Lancaster University, InfoLab21, Lancaster LA1 4WA UK
[2] ESRC Centre for Economic and Social Aspects of Genomics (Cesagen), Institute of Advanced Studies, Lancaster University, Lancaster LA1 4YD, UK
{gerald,s.hall}@comp.lancs.ac.uk

Abstract. Late binding to services in business-to-business operations pose a serious problem for dependable system operation and trust. If third party services are to be trusted they need to be dependable. One way to address the problem is by adding fault tolerance (FT) support to service-oriented systems. However, FT techniques are yet to be adopted in a systematic way within service oriented computing. Current FT frameworks for service-oriented computing are largely protocol-specific, have poor service quality differentiation and poor support for the FT process model. This paper describes a service differentiation-aware, FT framework based on the FT process model that can be used to support service-oriented computing.

Keywords: Service-oriented systems, fault-tolerance, differentiation-aware.

1 Introduction

Service-oriented architectures (SOA) such as web services, pose a serious problem for dependable system operation because they promise late binding. Late binding delegates the decision to trust a service to an external software agent. However, if third party services are to be trusted they need to be dependable. One way to address the problem is by adding fault tolerance support to service-oriented systems. Fault tolerance (FT) build reliable systems using mediated replication techniques. However, the adoption of FT techniques within service-oriented computing is still patchy and variable. Current FT frameworks used in service oriented computing suffer from a number of limitations:

- *Limited coverage of fault tolerance techniques*. There is a tendency for approaches to be problem or protocol specific [1]. Some frameworks provide extensibility mechanisms, but these are limited to simple active and passive replication techniques. A direct consequence of this limited coverage is a lack of evaluation of known FT protocols with regards to reliability and performance in service-oriented computing.
- *Poor support for the FT process model.* The FT process model [2] is based on a pure asynchronous messaging environment to remove all implicit timing assumptions about interactions. Current FT frameworks struggle in to work with

standard SOAs to provide asynchronism. Some are based on SOAs that only support synchronous request-response exchanges by being tied the underlying transport protocol such as HTTP [3], [4], [5]. Others bypass the SOA in favour of a hard-wired approach [6], [7], severely limiting extensibility.

- *Lack of discoverable FT services*. Existing service- oriented frameworks do not support discoverable fault tolerance services. FT protocols are embedded at a transport level using indirection. This makes them transparent to processes that may have specific FT requirements.
- *Poor support for service differentiation*. Existing FT frameworks do not provide a means for the system to select different protocols at runtime. In addition, current frameworks do not provide a means to differentiate between services that fulfil the same well-known role.

Our solution has been to provide FT support through an asynchronous messaging framework that provides a pluggable means to represent fault tolerance protocols as process models and to expose them as discoverable services. The framework provides runtime service differentiation mechanism based on quality of service and is supported by a decentralised platform. The rest of this paper is structured as follows. Section 2 reviews related work. Section 3 introduces our FT framework. Section 4 describes how a real *Trading Floor* application was used to evaluate the framework. Finally, we provide some closing thoughts in section 5.

2 Related Work

There are a number of initiatives for fault tolerance (FT) in service-oriented computing (SOC). Table 1 provides a summary of some notable FT frameworks in SOC. Because of space considerations, these are representative rather than exhaustive.

The Generic FT Container for SOA [8] is a mediated FT approach for synchronous SOAP invocations. Its limitations lie in its centralised approached to FT. Failure of the mediator would result in overall failure of the system. Its failure detection is also weak because it relies primarily on faults being raised by the services it mediates or the underlying connection time outs. Web Service-Fault Tolerance Mechanism (WS-FTM) [3] is a simple framework that is based on NVP [9]. WS-FTM is very limited in its scope and can only operate with stateless web services. It is preconfigured and cannot be reconfigured. It only allows active replication where most frameworks provide support for passive replication. Lastly, it can only use synchronous RPC type messaging.

FAult tolerance for Web Services (FAWS) [4] adopts a similar approach to the Generic FT Container and has similar limitations. It uses a mediator to route messages to variant implementations of the same service interface. It communicates with a management component, the FT-Admin, using Java based RMI calls. The FT-Admin component is responsible for informing the FT-Front of policy based on input from a failure detection component.

CORBA provides transference to service-oriented computing with the FT-CORBA standard. FT-CORBA has two implementations, FT-SOAP and FT-Web. FT-SOAP provides many useful properties including pseudo group membership, but suffers from a singleton topology where management component themselves can fail. FTWeb [5] provides a similar approach, but decentralises the management components. However, it is limited to synchronous RPC interactions.

Thema [6] is a FT framework for web services that provides an implementation of the Castro and Liskov Byzantine Fault Tolerance protocol (CLBFT) [10] by extending the BASE library [11]. Thema requires three libraries that extend the BASE libraries to provide integration to SOAP. A major problem with Thema is its requirement of UDP based IP multicast for its fault-tolerant framework for Web Services. CLBFT is very complex and using reliable TCP connections diminishes the efficiency of the process significantly. The Byzantine Fault-Tolerance framework for Web Services (BFT-WS) [7] addresses the topology problems of Thema by implementing the CLBFT algorithm directly on top of Web Services Reliable Messaging (WS-RM) [12]. However, there are major concerns over the performance of BFT-WS. Multicasting is replaced by a series of unicasts that degrades the performance. In addition, both Thema and BFT-WS are fixed implementations in two ways. Firstly, they are tied to the CLBFT protocol. Secondly, replicas are not free to join or leave at any time.

WS-Reliable Messaging (WS-RM) is a standard developed by the OASIS consortium [12]. The standard works by the sender indexing outgoing messages whilst requiring the receiver to send acknowledgements for all messages received. WS-RM suffers from several limitations. It does not support the broadcast of primitives required for state machine replication. While several reference implementations of WS-RM exist, the standard provides poor performance because of the overhead of starting/terminating sequences and acknowledgements.

Table 1. FT-SOC frameworks

Framework / Feature	Generic Container	WS-FTM	FAWS	FT-Web	THEMA	BFT-WS
Approach	Intermediary	Intermediary (Client)	Intermediary	CORBA	CLBFT	CLBFT
Passive replication	√	×	√	√	×	×
Active replication	√	√	×	√	×	×
State machine replication	×	×	×	×	√	√
Crash model	√	√	√	√	√	√
Byzantine model	≈	≈	×	≈	√	√
Requires synchrony	√	√	√	√	×	×
Decentralization	×	×	×	×	√	√
Messaging complexity	$\varphi=2(f+1)$	$\varphi=2n$	$\varphi=2(f+1)$	$\varphi=2(f+1)$	$\varphi=2n$	$\varphi=2n(n-1)^{\#}$
Scalable	√	√	√	√	×	×
Diversity	√	√	×	√	√	√
Late binding	×	×	×	√	×	×

Key

√ - Feature supported

≈ - Feature partially supported

× - Feature not supported

# - Includes $2n^2$ digital signature validations.

3 A Differentiation-Aware Fault-Tolerant Framework

This section describes our proposed FT framework. The framework comprises two main components; Late Asynchronous Message Brokering system (LAMB) and Sandbox (shown in Figure 1). LAMB is an asynchronous message brokering system that routes SOAP messages to services based on their header and content. LAMB brokers messages to services based on name matching. We assume all semantic decisions including deep interface matching are made at design-time [13]. Sandbox is a container for web services that provides logging, authentication, failure detection, synchronization and election. LAMB and Sandbox are supported by a JXTA-based platform for peer connectivity, organization and distribution of WSDL-based services.

Every host that forms part of our architecture possesses one instance of LAMB and Sandbox. Communications between hosts take place over protocols provided by the JXTA P2P protocols [14]. We adopt JXTA to provide a distributed information model based on adapted WSDL (with bindings to LAMB and QoS metrics). This is achieved by wrapping WSDL in a new type of JXTA advertisement, the WS-Advertisement forming the basis for service discovery. We also make use of JXTA's ability to share peer information and self-organize. Finally, we use the JXTA pipe that abstracts unreliable asynchronous unicast and broadcast primitives. JXTA supports implementation bindings, but these are passive. To link the LAMB and Sandbox infrastructure to JXTA we provide the FT platform.

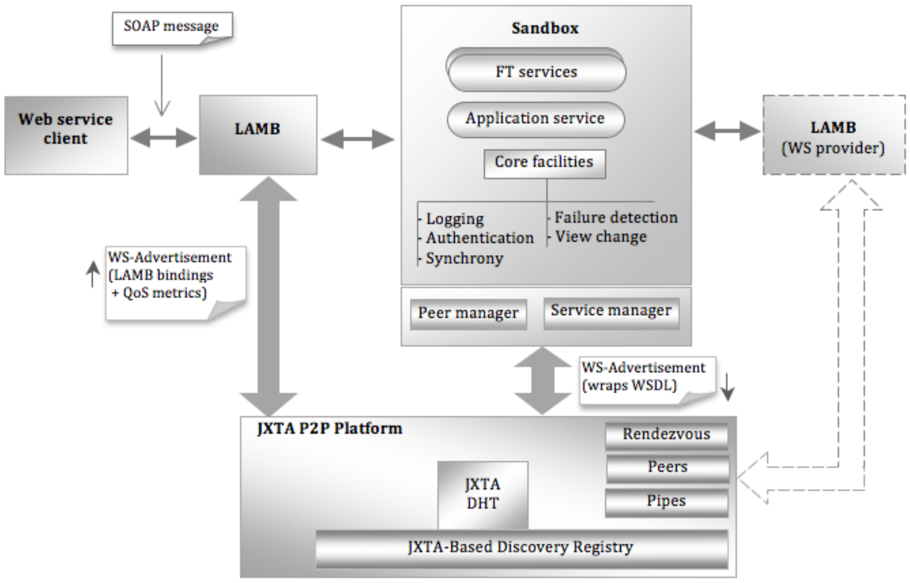


Fig. 1. Peer-based fault tolerance for web services

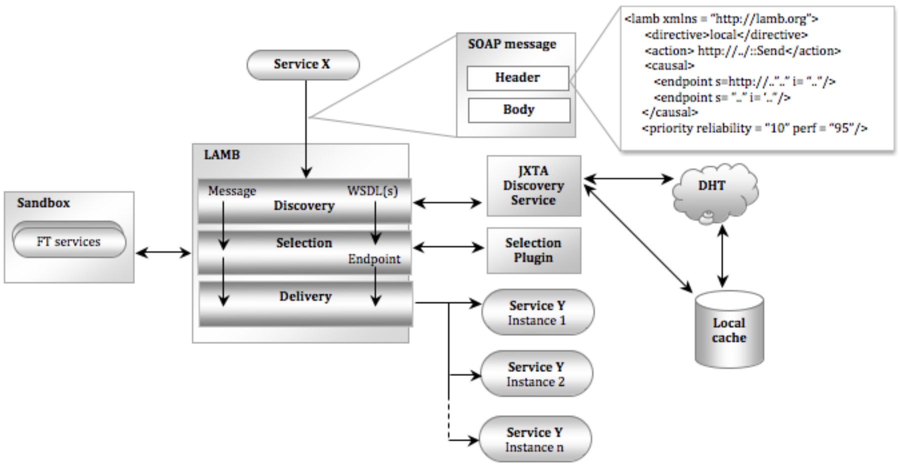
3.1 LAMB

LAMB is a message-oriented middleware (MOM) that asynchronously routes messages to web services, based on content. LAMB differs from other MOMs [18], [16], [17] by adhering to the following propositions:

1. *Any message* $m_x \longmapsto \{i_1 ... i_n\}$, where i_x is a service interface. In turn the interface $i_x \longmapsto \{s_1 ... s_n\}$, where s_x is a service definition. Finally, the service $s_x \longmapsto \{e_1 ... e_n\}$, where e_x is a service endpoint.

   A message has a name that identifies its content. LAMB uses the message name to identify candidate services that are able to consume the message. A WSDL defined service provides a set of endpoints to which the message can be routed. The assumptive part of this proposition is that a message will always uniquely identify services. Finally, if a message is consumed by two different interfaces then they intersect to form another interface $m_i \longmapsto i_x \wedge m_i \longmapsto i_y \Rightarrow m_i \longmapsto i_z : i_z \equiv i_x \cap i_y$

   All LAMB entities $\{m_x, i_x, s_x, e_x\}$ are uniquely identified by URIs.
2. *Everything is exposed as asynchronous web services.* LAMB treats the world as asynchronous web services that consume messages. This means that any orthogonal policies (in our case, FT) are also treated as services. Every service, FT or not, that is discovered by a LAMB broker must have a WSDL description.
3. *Policy Agnosticism.* Stemming from proposition point 2, a LAMB broker is not tied to any specific policy. Unlike other MOMs, such as WS-BUS [2], JMS [4] or Narada Brokering [19], it does not assume publish-subscribe interactions.
4. *Transport Agnosticism.* A LAMB broker should be able to use any well-known protocol to deliver messages to their endpoint; this includes HTTP, TCP/IP, SMTP or JXTA protocols.
5. *Optimistic brokering.* LAMB implements no specific infrastructure to ensure FT. It optimistically assumes all messages get delivered. If a failure occurs, LAMB neither notes it nor takes remedial action.
6. *Interoperability with SOAP and WSDL.* LAMB has bindings to SOAP and WSDL to ensure web service interoperability. In common with standards such as WS-Addressing [20], LAMB annotates SOAP message headers with its brokering information.
7. *Support for stateful services.* Web services can be stateful between different interactions. LAMB assumes that services are partitioned according to state and maintains a causal service history for each message.
8. *All brokers are equal.* LAMB brokers can reside on any host and see the same set of services within a domain. Service discovery does not depend on the topological or geographical location of a LAMB broker.
9. *LAMB Enabled Web-Services.* Web services must be altered to work with LAMB. Firstly, all outbound messages relative to a service must be passed directly to a LAMB broker service. Secondly, web services must correlate causally related messages by copying any LAMB headers between them.
10. *Zero recursion.* LAMB prevents a broker from selecting the service that a message has just come from to avoid *recursion*.



```
<lamb xmlns = "http://lamb.org">
    <directive>local</directive>
    <action> http://../::Send</action>
    <causal>
      <endpoint s=http://..".." i= ".."/>
      <endpoint s= ".." i= '..''/>
   </causal>
   <priority reliability = "10" perf = "95"/>
```

Fig. 2. Architecture of LAMB

As shown in Figure 2 LAMB consists of three services:

- *Discovery*. Take a message and search for matching WSDL descriptions based on qualified and URI names.
- *Selection*. Given a set of discovered services for m_x, the selection service chooses the most appropriate. We have implemented a simple priority scheme where a message can define its requirements in terms of QoS, performance and reliability. A service publishes similar QoS and the selection service chooses a service based on these criteria. The benefits of this priority scheme are discussed in section 4.
- *Delivery*. A service may consist of 1 or more endpoints. LAMB sends the message to all endpoints of a selected service using the most appropriate transport protocol. This may be over JXTA or using transport such as HTTP.

By setting a special field within the LAMB header of a SOAP message, discovery and selection modes can be changed. A *direct* mode means that a service endpoint is embedded in the header and the message gets routed without discovery or selection. *New* mode ensures that no service in the causal header is used. A *local* mode ensures discovery only looks for services on the same host. A *broadcast* mode bypasses the selection service. A *default* mode uses all three services with a bias towards services in the causal history. We have integrated LAMB with the JXTA Peer-to-Peer framework to provide decentralised discovery. It uses a *WSAdvertisment* an extended *JXTA Advertisement*, to index web service descriptions against message URIs to provide fast lookup times. Service publication and discovery is simply the distribution of *WSAdvertisements* over the JXTA SRDI a distribute hash table. This provides a consistent view across all LAMB brokers. Service descriptions are cached within all JXTA peers to further improve discovery times.

3.2 Sandbox

Sandbox is a container that houses FT protocol implementations and exposes them as services. It stems from the *Engine* API in [21] and container described in [8]. Sandbox uses introspection to expose FT protocols as process models enabled with WSDL service descriptions. Sandbox provides a core set of APIs or facilities that provide many common FT abstractions including failure detection and bounded synchronisation. These facilities include:

- *Message Routing*. Sandbox provides a conduit for incoming SOAP messages to reach FT services. Sandbox inspects an incoming message for a service URI and context identifier. The service URI is used to lookup a deployed service class. The context is used to lookup a service instance.
- *Service Lifecycle*. It manages the creation, destruction and state of embedded service instances.
- *Logging*. Many FT protocols require that messages and events get logged either in memory or to stable-storage. Our mechanism, called Domesday, stores arbitrary objects or messages, allowing querying in a variety of chronological ways.
- *Authentication*. Gatekeeper is a facility available to embedded services that provides message authentication and encryption using either RSA digital signatures or Message Authentication Codes (MACs). This allows FT services to check if a message has been tampered with and to verify that it is truly from the sender.
- *Crash Failure Detection*. We have developed a distributed crash failure detector, called *Eternity*, using heartbeats over a ring-topology to inform embedded services of the liveliness of other processes.
- *Synchronisation*. We have developed *Clockwork,* an API that records message-received times for incoming messages. Clockwork generates events when two correlated messages do not synchronise within an upper-bound.
- *Deterministic view changes*. We have developed *Viewpoint*, an API that supports the view-change protocol [22]. Viewpoint allows processes to be chosen deterministically as leaders across the network. It works by taking a view number and selecting the leader from a set of identifiers.

3.3 Fault Tolerance Protocols

To demonstrate the agnosticism and coverage of our framework we have implemented six different protocols that are exposed as FT services:

- *Patmos*. This is a distributed version of ubiquitous recovery block protocol for passive replication, as also found in [4], [5], [8]. However, in our version the intermediary is rotated between, and monitored by, multiple protocol instances so that there are not single points of failure.
- *Elegant*. This protocol provides N-Version Programming like [3], [5], [8] without voting. The protocol has no service implementation, hence it is elegant, and it relies on the distribution of endpoints for one functional service.

- *Atakos*. Enhances *Elegant* to provide voting on multiple service responses, this enables the leverage of diversity.
- *Ionian*. As far as we are aware this is the first web service implementation of the Paxos protocol for state-machine replication (SMR) [1]. Ionian is a clone of multi-Paxos instances that includes an extra messaging step to regulation of total-order proposals.
- *IonianNB*. Ionian non-blocking is an extension to Ionian that removes the proposal regulation step and instead internally consensus decisions.
- *Andros*. Andros is a clone of the CLBFT protocol [10] as supported by [6], [7]. It uses three-step consensus and authentication (through Gatekeeper) to provide Byzantine FT. Andros has both RSA digital-signatures and MAC based variants.

4 Case Study – Trading Floor System

Our case study is based on the Trading Floor system used by one of London's large financial institutions. The system provides real-time information on stocks, bonds, commodities, derivatives and currency to many traders simultaneously. The system consists of three core services as shown in Figure 3. A coordinator service initiates the cycle by sending a fetch indicator message to one or more source services, which fetch the data from online systems. Once the source has fetched an indicator it sends a show indicator message to all available screen services. Each screen service displays the indicator value on the chart shown in the actual trading floor screen. The screen service keeps a record of all indicators. To complete the cycle a screen service sends a log indicator message back to the coordinator. The coordinator can determine what screen services are currently operating or if a source service has failed.

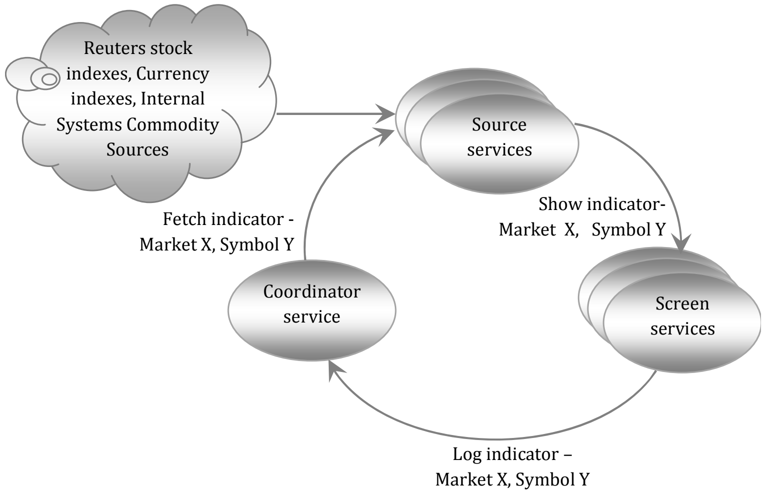

Reuters stock indexes, Currency indexes, Internal Systems Commodity Sources
Source services
Show indicator- Market X, Symbol Y
Fetch indicator - Market X, Symbol Y
Coordinator service
Screen services
Log indicator – Market X, Symbol Y

Fig. 3. Trading floor services

This type of application is suitable for evaluating our FT framework as it depends on the plurality of both the sources and the screens. The test case also allows us to contrast the effect of different FT protocols. To evaluate the effect of errors, we doped the source services to generate deterministic pseudo-random values based on the indicator name and a clock value set by the coordinator. Hence unless instructed otherwise, the sources service instances generated the same "random" value.

We scripted doping profiles to generate a wide range of failure conditions including conditional, stochastic and periodic. The doping mechanism was used to drive the injection of requests. The client dope used send methods in the trading floor coordinator service to perform each request. We used two forms of injection; soak injection to create client requests at a fixed rate for a given time and load injection to increase the rate of requests over time.

4.1 Performance Monitoring

The framework incorporates an interface to assess the performance and reliability of the FT protocols. The interface takes snapshots of messages at predefined points in time, during the system operation and sends the results to a monitoring peer that stores all snapshots. Relevant FT metrics are computed by sampling the snapshots over a given period of time. Snapshots are grouped using a correlation identifier to provide start and end points. The in-rate can be computed from the start points, and the out-rate from the end points. Based on this, the throughput, cumulative loss and average latency can be determined. It is also possible to determine when messages are received out of order.

4.2 FT Configuration

A configuration in this context is the deployment of test-case and FT services across a set of nodes. Table 2 shows all configurations used in our case study. A selection of the FT scenarios applied to the case study are describe next.

Table 2. Configuration used in case study

Configuration	Coordinator Nodes		Sources Nodes			Screen Nodes	
	Services	Instances	Services	*n*	*f*	Services	Instances
No-FT	TF Coordinator	1	TF Source	1	0	TF Screen	1
Elegant	TF Coordinator	1	TF Source	3,5,7	2,4,6	TF Screen	1
Atakos	TF Coordinator	1	TF Source	3,5,7	1,2,3	TF Screen Atakos	1
Patmos	TF Coordinator	1	TF Source Patmos	3,5,7	2,4,6	TF Screen	1
Ionian	TF Coordinator	1	TF Source Ionian	3,5,7	1,2,3	TF Screen Atakos	1
IonianNB	TF Coordinator	1	TF Source IonianNB	3,5,7	1,2,3	TF Screen Atakos	1
Andros MAC	TF Coordinator	1	TF Source Andros MAC	4,5,10	1,2,3	TF Screen Atakos	1
Andros RSA	TF Coordinator	1	TF Source Andros RSA	4,5,10	1,2,3	TF Screen Atakos	1

4.3 Normal Operation Scenario

This scenario evaluated the framework's general performance and scalability. It was divided into two parts. First, the soak variant injected requests at a fixed rate for a given time allowing the user to contrast all configurations directly. By testing different values of *n* for each protocol (for example *Andros* with n = 4,7,10) we observed the properties of *n*-scalability directly. Secondly, a load variant was used to linearly increase load to test the maximum throughput that each configuration could tolerate, therefore assessing the load-scalability.

For the soak case our expectation was that latency would increase with the complexity of the underlying protocol, results. The results, shown in Table 3, are in-line with the expectation. At a fixed, one client request per second, there is a near zero percent loss (given a 1.5% margin for error in the metric API). *No-FT* has the lowest latency and *Andros* the highest.

Table 3. Normal operation with soak results

Name	***n***	**Inject 1 Request/sec**			**Inject 3 Requests/sec**		
		Loss %	Latency (ms)	MD Msg/sec	Loss %	Latency (ms)	MD Msg/sec
No-FT	1	0	177	2.2	0	177	3.2
Elegant	3	0	220	3.5	0.25	150	11.5
Elegant	5	0	270	3.65	0	260	12.75
Elegant	7	0	340	4.25	0.25	310	14.1
Atakos	3	0	625	3.6	0.25	322	13.6
Atakos	5	0	375	3.1	0.5	380	14.5
Atakos	7	0	405	3.25	0.25	430	15.0
Patmos	3	0.5	375	2.8	0.3	350	4.3
Patmos	5	0.5	302	3.3	0	390	6.2
Patmos	7	0	375	4.15	0.31	432	7.5
Ionian	3	0	500	3.5	0	700	9.0
Ionian	5	0	600	5.9	0	750	15.0
Ionian	7	0	1250	8.8	0	15000	27.0
IonianNB	3	0	625	3.0	0	500	5.0
IonianNB	5	0	760	8.0	0	620	12.5
IonianNB	7	0	740	12.0	9.0	4300	37.0
Andros MAC	4	0	1130	5.0	0	1200	5.0
Andros MAC	7	1.2	1170	15.0	85.0	28000	35.0
Andros MAC	10	0	1220	53.0	98.0	20000	10.0
Andros RSA	4	0.8	775	5.0	0	1000	6.0
Andros RSA	7	0.8	1050	22.5	75.0	17000	34.0
Andros RSA	10	1.0	1030	22.0	90.0	30000	19

As Figure 4 shows, all FT protocols had a maximum throughput, however we did not manage to overload the *No-FT* configuration. *Elegant, Patmos* and *Atakos* reached a throughput of 4-5 trans/s whereas *Ionian* was limited to a throughput of 2 trans/s. Andros demonstrated the poorest maximum throughput. All the consensus-based protocols are erratic one their maximum throughput is exceeded.

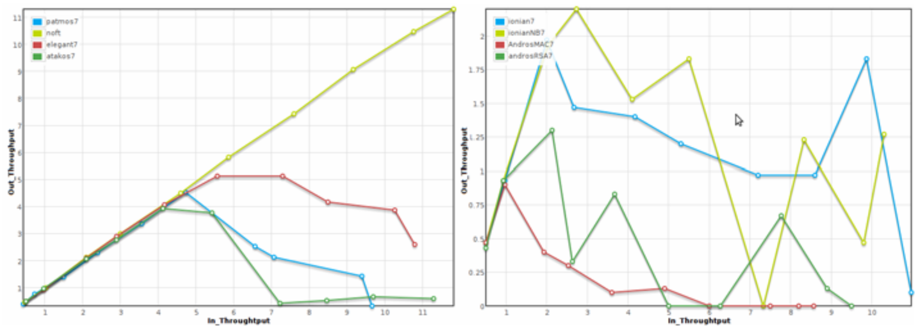


Fig. 4. Behaviour with increasing load (*n*=7)

4.4 Runtime Reconfiguration Scenario (Differentiation-Awareness)

The aim of this scenario was to test the property of dynamism that is enabled by the priority service selection scheme. We wanted to show that when a more resilient FT protocol (indicated by a higher reliability metric) is deployed, incoming requests are delegated to it rather than lesser protocols. So, for example, *Andros* would be chosen over *Ionian*, *Ionian* over *IonianNB* and so forth. To do this we constructed a series of configuration deployments over time whilst soak injecting requests. The result, shown in Figure 5, indicates that the transitions were smooth without the expected problems, clearly demonstrating the effectiveness of the FT priority selection scheme.

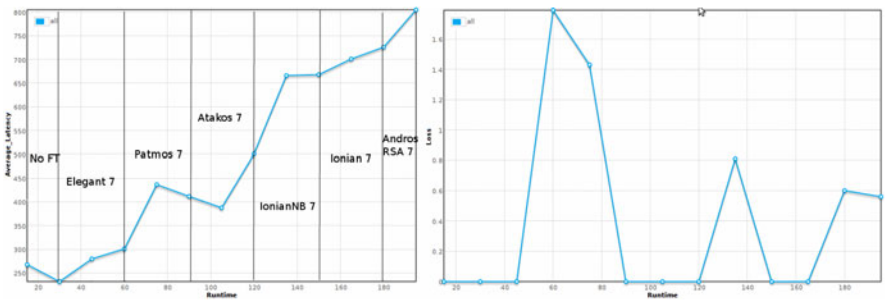


Fig. 5. Runtime reconfiguration results

4.5 Concurrent Fail-Stop Scenario

The aim of this scenario was to establish whether the protocols could support concurrent crash failures within a fail-stop distributed system model. We created two boundary cases when $n-1$ and n nodes crash concurrently. To tolerate concurrent fail-stop the configurations needed to survive $n-1$ crashes but not, trivially, n. For No-FT $n-1$ =0 so we did not run that case.

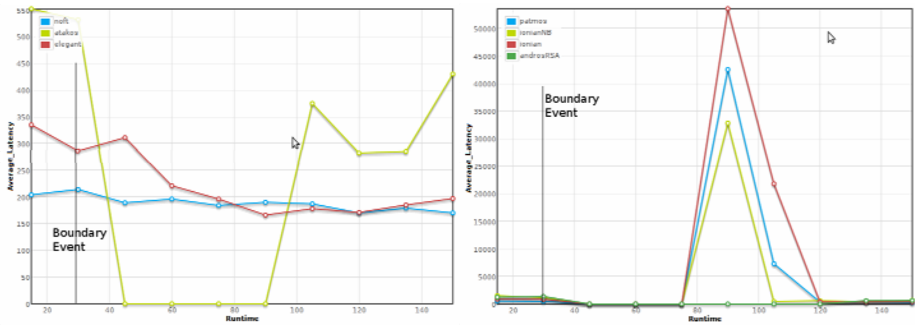


Fig. 6. Concurrent fail-stop (*n-1*)

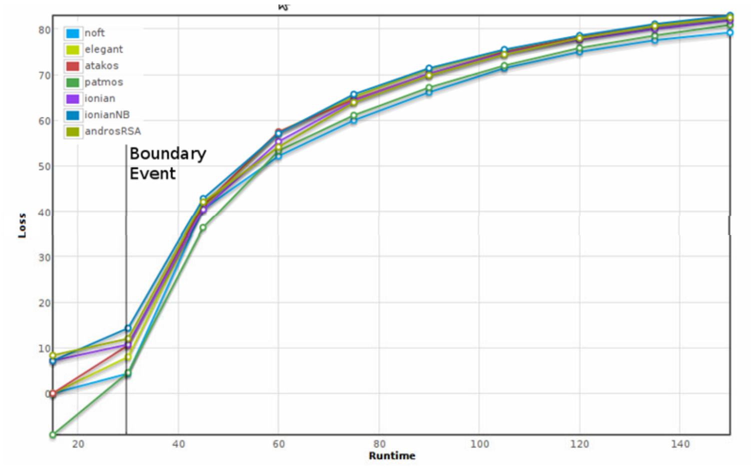


Fig. 7. Concurrent fail-stop (*n*)

Figure 6 shows that all the configurations tolerated *n−1* concurrent fail stops successfully (the latency is measured in milliseconds). Both Andros RSA and *Atakos* showed a much slower recovery than the other protocols. Though in both cases there was slightly recovery towards the end of the run. Figure 7 clearly shows that all the protocols failed totally completing the boundary case.

4.6 Byzantine Scenario

The aim of the Byzantine scenario was to demonstrate that our framework is able to tolerate arbitrary failures in common with [6], [7]. We divided the scenario into four cases: *f* and *f+1* concurrent failures; stochastic failures with an increasing probability; post recovery state. A Byzantine failure is an alternation between the following: fail-silent, omission, timing, denial-of-service or commission.

We expected that only *Andros* would be able to tolerate *f* Byzantine failures. This result is demonstrated in Figure 8 where *Andros* has less loss than either *Ionian* or

IonianNB. However, the result was not perfect. Through observational evidence we noted that a fail-silent, omission, timing or Byzantine attack caused some less powerful nodes to fail-stop. This triggered constant reconfigurations inside the SMR protocols leading to a temporary increase in loss, this result is reflected in Figure 8. In the Byzantine *f+1* scenario as expected all configurations showed an upward trend in loss and latency times.

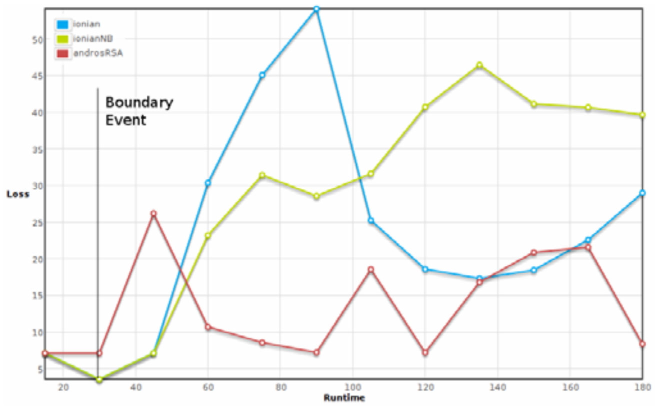


Fig. 8. Byzantine failures (f) for SMR protocols

Figure 9 shows part of the test-bed running the Trading Floor application

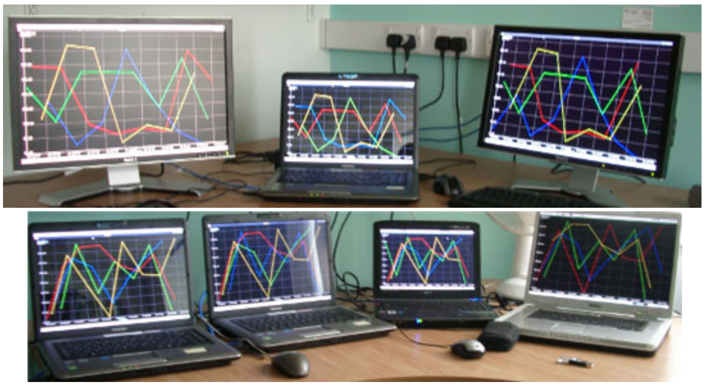


Fig. 9. Test-bed

5 Conclusions

In existing service-oriented frameworks FT is treated as an orthogonal issue to services. Service-oriented frameworks do not support discoverable fault tolerance services. Instead FT protocols are embedded at the transport level making them transparent to the application. Current FT frameworks provide poor support for the FT process model and protocol differentiation restricting the applications ability to select appropriate FT protocols at runtime. Our solution is a differentiation-aware FT framework that provides a pluggable means to represent fault tolerance protocols and to expose them as discoverable web services. We have demonstrated using a real case study how our framework addresses the problems outlined in section 1 (introduction). We have evaluated the framework over many different fault scenarios covering common failure models and showed it to be effective.

Our framework provides a messaging environment for fault tolerance protocols to operate using LAMB, an asynchronous message brokering system. LAMB improves on existing message-oriented middleware by providing a full service-oriented architecture. We have addressed the issue of representing FT protocols as process models and exposing them as services by using Sandbox. Sandbox is container for FT services that allows protocols to be represented as process models. Sandbox extends the approach described in [5] by adding structure and introspection. We have provided a decentralised platform for FT by implementing our framework over a P2P overlay. Ensuring the framework has no singleton components.

We have provided a means for discovering FT services by providing LAMB with the ability to publish and discover WSDL descriptions. To ensure that the descriptions are disseminated to all LAMB brokers we use the JXTA SRDI as the information model. Lastly, our framework supports runtime protocol differentiation by incorporating a simple QoS matching scheme within LAMB. All services have a priority based on reliability and performance metrics.

While our FT framework is a significant improvement on existing frameworks for service-oriented computing, further improvements are needed if it is to address complex service-oriented systems. We are currently exploring ways to improve its performance and to make it more scalable.

Acknowledgements. The support of the Economic and Social Research Council (ESRC) is gratefully acknowledged. This work is part of the Research Programme of the ESRC Genomics Network at Cesagen (ESRC Centre for Economic and Social Aspects of Genomics).

References

1. Lamport, L.: Paxos Made Simple. ACM SIGACT News (Distributed Computing Column) 32(4), 51–58 (2001)
2. Guerraoui, R., Rodrigues, L.: An Introduction to Reliable Distributed Programming. Springer, Heidelberg (2006)
3. Looker, N., Munro, M., Xu, J.: Increasing web service dependability through consensus voting. In: Proc. of the 29th Annual International Computer Software and Applications Conference, vol. 2, pp. 66–69. IEEE Computer Society, Los Alamitos (2005)
4. Jayasinghe, D.: FAWS for Soap-based Web Services. IBM Developerworks. (2005), `http://www.ibm.com/developerworks/webservices/library/ws-faws/`

5. Santos, G.T., Lung, L.C., Montez, C.: FTWeb: A Fault Tolerant Infrastructure for Web Services. In: Proc. of the 9th IEEE International EDOC Enterprise Computing Conference (EDOC 2005), pp. 95–105. IEEE Computer Society, Los Alamitos (2005)
6. Merideth, M.G., Iyengar, A., Mikalsen, T., Tai, S., Rouvellou, I., Narasimhan, P.: Thema: Byzantine-Fault-Tolerant Middleware for Web-Service Applications. In: Proc. of the 24th IEEE Symposium on Reliable Distributed Systems, pp. 131–142. IEEE Computer Society, Los Alamitos (2005)
7. Zhao, W.: BFT-WS: A Byzantine Fault Tolerance Framework for Web Services. In: Proc. of.Enterprise Computing Conference (EDOC 2007). Eleventh International IEEE, pp. 89–96 (2007)
8. Sommerville, I., Hall, S., Dobson, G.: A Generic Mechanism for Implementing Fault Tolerance in Service-Oriented Architectures. Tech. Report, Computing Dept., University of Lancaster (2005)
9. Avizienis, A.: The N-Version Approach to Fault-tolerant Software. IEEE Trans. Software Engineering 11(12), 1491–1501 (1985)
10. Castro, M., Liskov, B.: Practical Byzantine Fault Tolerance and Proactive Recovery. ACM Trans. Comput. Syst. 20(4), 398–461 (2002)
11. Rodrigues, R., Castro, M., Liskov, B.: Base: Using Abstraction to Improve Fault Tolerance. In: Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001), pp. 15–28. ACM Press, New York (2001)
12. Davis, D., Karmarkar, A., Pilz, G., Winkler, S., Yalinalp, U.: Web Services Reliable Messaging (WS-ReliableMessaging) Version 1.1. Web, (2007), `http://DOCS.OASIS-OPEN.ORG/WS-RX/WSRM/200702/WSRM-1.1-Spec-OS-01.html`
13. Alonso, G.: Myths of Web Services. IEEE Data Engineering Bulletin 23(4), 3–9 (2002)
14. Verstrynge, J.: Practical JXTA, Lulu.com (2008)
15. Erradi, A., Maheshwari, P.: WSBUS: A Framework for Reliable Web Services Interactions. In: Proc. of the 2005 ACM Symposium on Applied Computing (SAC 2005), pp. 1739–1740. ACM, New York (2005)
16. Hapner, M., Burridge, R., Sharma, R., Fialli, J., Stout, K.: Java Messaging Service. Web, (2002), `http://java.sun.com/products/jms/docs.html`
17. Pallickara, S., Fox, G.: NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. In: Proc. of Middleware 2003, pp. 41–61. Springer, Heidelberg (2003)
18. Chinnici, R., Haas, H., Lewis, A., Moreau, J.-J., Orchard, D., Weerawarana, S.: Web Services Description Language (WSDL) Version 2.0 part 2: Adjuncts. Web, (2005), `http://www.w3.org/TR/wsdl20-adjuncts`
19. Reiter, M.K.: The Rampart Toolkit for Building High-Integrity Services. In: Birman, K.P., Mattern, F., Schiper, A. (eds.) Dagstuhl Seminar 1994. LNCS, vol. 938, pp. 99–110. Springer, Heidelberg (1995)
20. Gudgin, M., Hadley, M., Rogers, T.: Web services addressing 1.0 - core. Web,(2006), `http://www.w3.org/TR/ws-addr-core`
21. Hall, S., Kotonya, G.: An adaptable fault-tolerance for SOA Using a Peer-to-Peer Framework. In: Proc. of the IEEE International Conference on E-Business Engineering (ICEBE 2007), pp. 520–527. IEEE Computer Society, Los Alamitos (2007)
22. Liskov, B.: From Viewstamped Replication to BFT, (2007), `http://www.inf.unisi.ch/30YearsOfReplication/pps/Liskov.pdf`, 30 Years of Replication Lecture Series