Failure Semantics in a SOA Environment

Chris Hobbs, Hanane Becha
Nortel
3500 Carling Avenue
Ottawa, ON, K2H 8E9 (Canada)
{cwlh,hananebe}@nortel.com

Daniel Amyot
SITE, University of Ottawa
800 King Edward
Ottawa, ON, K1N 6N5 (Canada)
damyot@site.uottawa.ca

Abstract

In a Service-Oriented Architecture (SOA), services publish descriptions to permit their composition into larger services. There are however serious gaps in the semantics of such service descriptions *and these hinder the adoption of SOAs in mission-critical applications. This paper identifies some of these lacunae and proposes a foundation for resolving one of them—service failure. The technique of crash-only failure is proposed as a useful first step and we illustrate how it is particularly applicable to web services in a SOA.*

Keywords: Failure Semantics, Crash-Only, Orchestration, Runtime Governance, Semantics, SOA, Web Service.

1. Introduction

This paper addresses a problem associated with the technique, common in Service-Oriented Architectures (SOAs), of creating new services by orchestrating existing ones. The problem of interest here is one of failure semantics and is presented in section 1.2. The technique of *crash-only* failure is proposed as a useful first step to solving this problem. Section 2 contains a very short précis of the crash-only paradigm and section 3 illustrates how it is particularly applicable to web services in a SOA. Section 4 describes the failure semantics of such software and section 5 addresses briefly the related question of service availability.

1.1. A Note on Terminology

The terms for components of a SOA used in this paper are taken from the OASIS SOA Reference Model [9]. This means in particular that, rather than a *client* requesting service from a *server*, this paper speaks of a *consumer* requesting service from a *provider*.

The major components of a SOA are illustrated in simplified form in Figure 1: a provider offers some form of service and advertises its interface and behavioural and contractual properties by storing a *service description* in a registry. For the purposes of this paper, the registry may be as formal as a computer system running a protocol such as Universal Description, Discovery and Integration (UDDI) or as informal as an email sent from service provider to consumer. A potential consumer retrieves the information from the registry and invokes the service. Runtime governance, which forms an important part of the proposal in this paper, is installed between the consumer and the provider to perform common functions—see section 3.1.

1.2. The Problem

Within a Service-Oriented Architecture (SOA), new services are typically created by orchestrating existing ones. Figure 2 illustrates a particularly simple case wherein two underlying services, X and Y, are orchestrated in some way to produce a new one, Z. In the general case, X and Y will not be owned by the developer of Z, each being a service exposed by other service providers.

To determine Z's characteristics, so that service-level guarantees can be offered to customers, it is necessary to combine the characteristics of X and Y with those of the additional logic provided by Z. Although the SOA specifications make provision for X and Y to advertise their interface syntax, their behaviour and their contracts, no method has been proposed for defining many of the other necessary characteristics. Sanders *et al.* have proposed semantic interfaces that capture behavioral contracts in more details than interfaces based on signatures [10], but non-functional aspects are still not addressed. For instance, the performance, scaling, management, security, privacy, resource (see section 3.2), availability, reliability[1] and many other models of X and Y need to be published so that they can be used by the developer of Z to determine the corresponding characteristics of Z.

[1] We distinguish between availability (is an answer received?) and reliability (is the received answer correct?)

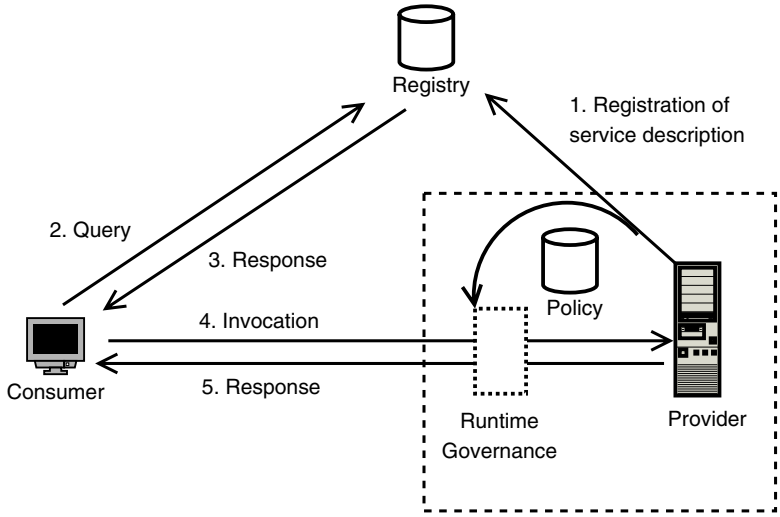

Figure 1. SOA Components

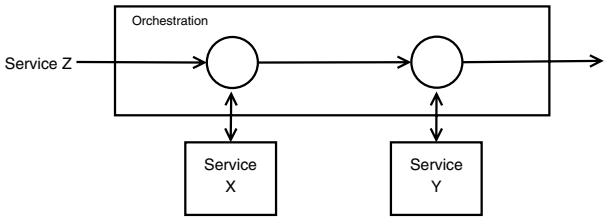

Figure 2. Service Orchestration

As a simple example, consider privacy: X, Y and Z may be implemented in countries with different laws regarding privacy and security of data. For the developer of Z to ensure that that service complies with the local regulations and to be able to offer reassurances to customers about the privacy of their data, the privacy policies of X and Y need to be available.

Each of the models listed above, and others, are needed but this paper addresses one in particular: the failure model. If the failure model of Z is to be calculated, the failure modes of X and Y have to be known. It may be, for example, that X detects faults before they become failures and rolls back to a state stored before the interaction, guaranteeing that it returns to a sane state but putting the responsibility for re-submission of inputs onto the consumer (Z). Y, on the other hand, may buffer information and the precise state of an interaction may be difficult to determine when a failure occurs.

To permit Z to determine necessary actions following the failure of X or Y and to allow it to make claims about its own failure modes, a failure ontology is required which can capture X's and Y's (and Z's) failure semantics. This paper argues that the technique of "crash-only software", introduced by Candea and Fox [5], is particularly suited to the loosely-coupled environment of SOAs, providing particularly simple behaviour that can be described and advertised in a formal manner. It is unrealistic to expect all services to comply with this failure paradigm but it is proposed that it form the basis of the failure semantics for web services.

2. Crash-Only Software

2.1. Fault Tolerance

Studies (some dating back to 1986—see the work of Gray [7]) and anecdotal evidence support the view that failures in deployed software are mainly caused by Heisenbugs, i.e. bugs caused by subtle timing interactions between threads and tasks. These prove impervious to conventional debugging, being non-reproducible and sensitive to tracing and other observation. Reproducible bugs, the so-called Bohrbugs, are easier to detect and fix during development and beta-deployment and can largely be removed before shipment of a final product. It must be accepted that, in any software-based system, Heisenbugs exist and failures will occur.

Telecommunications and other high-availability systems are built using the techniques of fault-tolerant computing (e.g., see [12]). From its early days with *Recovery Blocks* [1], fault-tolerant computing has been concerned with intercepting faults before they become errors and errors before they become failures. At each level the principle is to save as much state as possible, gracefully shut down the offending task and other affected components (defined by a failure tree), take whatever recovery action is required and then restart the affected components. This process re-

quire complex logic, some of which has to run in the failing module. Errors, for example in the *programming by contract* paradigm, are detected by checking invariants and pre- and post-conditions, typically by the code itself. Recovery for some high-availability techniques such as virtual synchrony (see [2]) can be very sophisticated, involving the execution of protocols to expel the failing server from the process group and resynchronise it on recovery.

2.2. Crash-Only

The technique of *crash-only*, which builds on the foundations laid by fail-stop techniques (see, for example, [11]), argues that the sophistication described above is not only unnecessary but, in many cases, counter-productive. Consumers of a service, it is argued, must anticipate that their provider will, from time to time, crash cleanly without the opportunity for sophisticated failure handling (perhaps because of a loss of power to the computer running the provider). Consumers must therefore already have the capability of handling such a crash. If this is the case, then rely on it and always crash the component whenever any fault is detected or failure occurs.

This *crash-only* semantics has several advantages:

- it defines simpler macroscopic behaviour with fewer externally-visible states.
- it reduces the outage time of the provider by removing all shutting-down time.
- it simplifies the failure model significantly by reducing the size of the recovery state table. In particular, crashing is stimulated from outside the software of the provider—it therefore assumes nothing about the continued correct behaviour of the failed component. The *crash-only* paradigm coerces the system into a known state without attempting to shut down cleanly: reducing substantially the complexity of the provider code.
- it simplifies testing by reducing the failure combinations that need verification.

If software is to crash cleanly more often, then it should also be written in such a way as to restart quickly—for example, see reference [6].

A crash manager would typically control the external trigger for the crash (the "on/off switch") and indicate to the consumer that it should retry after a specified time, generating the `RetryAfter(t)` response. This paper argues that SOAs already provide the necessary components to act as crash managers.

3. Crash-Only and Web Services

3.1. The Rôle of Runtime Governance

The description of *crash-only* software in [5] assumes, when recast using SOA terminology, that the providers (X and Y in Figure 2) will exhibit *crash-only* failure behaviour and that consumers, having failed to obtain timely or correct service, can initiate the crash. This may be acceptable when the consumer and provider, although loosely-coupled, are within one trust domain. This is clearly not generally the situation with web services.

Runtime governance (sometimes misleadingly called "management" in the literature) is a layer inserted by the service provider in front of services to perform common, policy-driven, functions such as load-balancing, encryption/decryption, consumer authentication and the monitoring of service-level agreements—see, for example, [3] and Figure 1. This layer is made possible in a SOA environment by having access to the service description of the invoked service, being in a position to intercept and decode all incoming requests and responses and by the common underlying encodings (XML, HTTP).

One common function of this software layer is to monitor response times from the service to ensure that the consumer is getting the level of service guaranteed in the service level agreement (e.g., over any 1 hour period, 90% of all requests will be responded to within 30 ms). This type of monitoring is typically specific to a particular service and consumer and provides the perfect location for invocation of the "power-off" switch provided by the *crash-only* software. That switch is external to the service, relying in no way on continued correct operation of the service code itself, and its operation is idempotent meaning that the decision to kill the server does not require the knowledge of its internal state.

Deliberately induced crashes are also a useful technique for software rejuvenation (see [13]) and this requires detection or prediction of periods of low usage of the service and, if necessary, buffering of requests. Again, runtime governance is an obvious candidate for recognising such periods, performing the buffering and causing the restarts.

3.2. Mapping Crash-Only to Web Services

The previous section argues that, if services are built in accordance with a crash-only paradigm, SOA components are available to handle the on/off switch. This section argues further that the crash-only paradigm is also particularly suited to web services. Fox and Patterson [5] list the properties required of a *crash-only* system and these can be abstracted remarkably well to match those of web services as described in [9]:

- **Components have externally enforced boundaries.** This is an implementation recommendation supported by the virtual machine concept used on many web service systems.
- **All interactions between components have a timeout.** While the web service standards provide support for asynchronous invocations protected by a timeout, they are not universally used. Where an interaction is not asynchronous, it can be made so by the runtime governance layer offering a synchronous interface to the consumer and invoking an asynchronous interface to the provider.
- **All resources are leased to the service rather than being permanently allocated.** This technique is particularly useful in avoiding another of the problems associated with service orchestration: resource contention. Understanding resource usage in underlying services is essential for avoiding deadlocks—Z needs to be aware if X and Y make use of a common resource and, when invoked with some particular timing, may deadlock. For resources completely internal to X and Y this is not likely but, in practice, X and Y are themselves likely to be composed of subservices, some of which may be common (Figure 3). This problem is impossible to avoid but leasing resources for time-limited periods provides a possibility of escape from the resulting deadlock.
- **Requests are entirely self-describing.** For *crash-only* services, requests must carry information about idempotency and time-to-live. In a SOA environment, it is unreasonable (and dangerous) to expect the consumer to provide this information. Again runtime governance, equipped with the understanding of the request from the service description (see Figure 1) and the associated service- and consumer-related policies, can step into the breach and provide this information. Note that Candea and Fox [5] map the generation of idempotency and time-to-live to a REST-like[2] environment but the comments are equally applicable to a true SOAP-defined[3] web service.
- **All important non-volatile state is managed by dedicated state stores.** Many interactions require that some state be held and the crash-only semantics requires that this be held externally to the provider and defines the characteristic required of a state store. It is very common in web service applications to have a "back-end" database which can provide this functionality.

[2]REST: Representational State Transfer
[3]SOAP: originally "Simple Object Access Protocol"

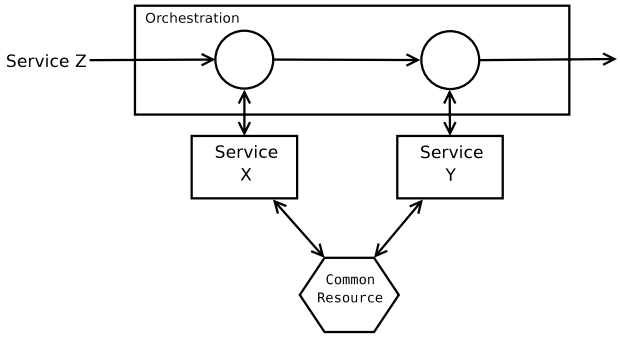

Figure 3. Common Resources

The major observation in this section is the strategic positioning of runtime governance and its rôle of intermediary between the consumer and provider of services. In this position it has the necessary information to:

- add idempotency and subscriber-dependent time-to-live information to requests to the provider.
- monitor the provider for anomalous behaviour.
- be the trusted source of crash commands for the provider, both as a result of delayed or insane response or as a result of a need for rejuvenation.
- protect the provider, Z, while its crash recovery is in progress, holding off or rejecting incoming requests until recovery is complete.
- generate `RetryAfter()` instructions to consumers.

4. Crash Semantics

Software failure models are normally large and difficult to combine, typically consisting of a Markov or semi-Markov chain (see, for example, [8]) expressed in a format proprietary to the tool used by the analyst.

As a foundation for the taxonomy of failure models, this paper proposes the crash-only semantic. As more complex models are required, they can be added but, where applicable, developers should be encouraged to build idempotent services with the crash-only failure semantic.

The published service description will then contain three properties:

1. the failure and recovery type—in this case *crash-only*
2. whether the service is idempotent or not
3. the anticipated (modelled or measured) failure distribution

Note that, if the service is *not* idempotent then all responsibility for determining the state of a recovered server lies with the consumer.

5. Availability and Crash-Only

One possible criticism of a crash-only architecture is a potential reduction in availability: the crash-only paradigm effectively removes layers of sophistication built using fault-tolerant techniques and trades Mean Time to Repair (MTTR) for Mean Time to Failure (MTTF). As Bev Littlewood famously said in [8]:

> Availability is maximised by maximising the mean time between failures and minimising the mean time to repair or recovery...

The actual availability required of a service depends on the system of which it is a part but the fabled "five nines" (99.999%) can be taken as an example. This availability allows a maximum downtime of about 5 minutes 16 seconds per year but does not specify how that downtime is distributed: if the MTTR is low, then the MTTF may also be correspondingly low. The technique of *crash-only* software is based on the assumption that it is often simpler and more effective to reduce MTTR than increase MTTF. Crash-only techniques provide a low MTTR and it is interesting to calculate the corresponding MTTF for a "five nines" service.

Candea *et al.* [6] cite experiments using an eBay-like auction system known as eBid. This is a complex Java application programmed using the *crash-only* and *micro-reboot* paradigms. Running on 3GHz Pentium machines with Linux operating systems and J2EE 1.3.1, this application was subject to a number of different fault types, and outage times (MTTRs) were between 411 and 601 msecs.

To achieve 99.999% availability, this permits between 526 and 769 outages per year: an outage every 11 to 17 hours (MTTF).

In conjunction with the simpler recovery actions when the underlying software is known to support a simple crash-only paradigm, and the possibilities of averting failure through micro-rejuvenation during periods of low usage, the availability targets do not seem difficult to meet. Fox and Patterson [4] analyse this tradeoff between MTTR and MTTF more fully.

Whether 769 failures per year, each of 411 msecs, are better than a single failure of just over 5 minutes depends on the application—it is easy to think of examples where each would be inappropriate.

6. Conclusions

For a developer to build service Z (Figure 2) orchestrating sub-services and offer service-level agreements on Z, the underlying services, X and Y, must publish information defining their behaviour in various areas including performance, failure, scaling, management, security, privacy, resource, availability and reliability.

Proposals are being created for some of these areas (e.g., management models within the Distributed Management Task Force's (DMTF's) Telecommunications Work Group[4]) but it is not clear what semantics could be applied to many of the other areas.

In the area of failure, this paper proposes a baseline and first category—crash-only. While not adequate for all web services, it appears to be a failure mode particularly suitable for a SOA environment and its semantic expression is relatively simple.

Future work will involve (1) the inclusion of failure information in service interfaces and (2) the study of other categories of failures and their validation and formal comparaison in different application domains, including SOA and web services.

References

[1] T. Anderson and R. Kerr. Recovery blocks in action: A system supporting high reliability. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 447–457, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[2] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138, New York, NY, USA, 1987. ACM Press.

[3] G. Cuomo. IBM SOA "on the edge". In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 840–843, New York, NY, USA, 2005. ACM Press.

[4] A. Fox and D. Patterson. When does fast recovery trump high reliability? In *2nd Workshop on Evaluating and Architecting Systems for Dependability (EASY)*, San Jose, USA, 2002.

[5] G. Candea and A. Fox. Crash-only software. In *9th Workshop on Hot Topics in Operating Systems*, San Jose, USA, 2003.

[6] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman and A. Fox. Microreboot—a technique for cheap recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[7] J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[8] B. Littlewood. A reliability model for markov structured software. In *Proceedings of the international conference on Reliable software*, pages 204–207, New York, NY, USA, 1975. ACM Press.

[9] OASIS SOA Reference Model TC. Reference model for service-oriented architecture 1.0. Technical report, OASIS, 2006.

[10] R.T. Sanders, R. Braek, G. Bochmann, and D. Amyot. Service discovery and component reuse with semantic interfaces. In *12th SDL Forum (SDL 2005), LNCS 3530*, pages 85–102. Springer, 2005.

[4] DTMF: http://www.dmtf.org/

[11] R. D. Schlichting and F. B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, 1983.

[12] M. Treaster. A survey of fault-tolerance and fault-recovery techniques in parallel systems. *ACM Computing Research Repository (CoRR)*, 2005.

[13] K. Vaidyanathan, R. E. Harper, S. W. Hunter, and K. S. Trivedi. Analysis and implementation of software rejuvenation in cluster systems. In *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 62–71, New York, NY, USA, 2001. ACM Press.