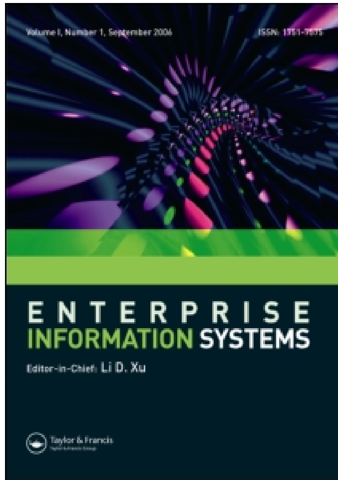
This article was downloaded by: *[Canadian Research Knowledge Network]*
On: *3 March 2010*
Access details: *Access Details: [subscription number 918586895]*
Publisher *Taylor & Francis*
Informa Ltd Registered in England and Wales Registered Number: 1072954 Registered office: Mortimer House, 37-41 Mortimer Street, London W1T 3JH, UK



Enterprise Information Systems
Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t748254467

Architectural design for resilience

Dong Liu [a]; Ralph Deters [a]; W. J. Zhang [b]
[a] Department of Computer Science, 176 Thorvaldson Building, University of Saskatchewan, Saskatoon, SK, Canada [b] Department of Mechanical Engineering, University of Saskatchewan, Saskatoon, SK, Canada

First published on: 17 August 2009

To cite this Article Liu, Dong, Deters, Ralph and Zhang, W. J.(2009) 'Architectural design for resilience', Enterprise Information Systems,, First published on: 17 August 2009 (iFirst)
To link to this Article: DOI: 10.1080/17517570903067751
URL: http://dx.doi.org/10.1080/17517570903067751

PLEASE SCROLL DOWN FOR ARTICLE

Full terms and conditions of use: http://www.informaworld.com/terms-and-conditions-of-access.pdf

This article may be used for research, teaching and private study purposes. Any substantial or systematic reproduction, re-distribution, re-selling, loan or sub-licensing, systematic supply or distribution in any form to anyone is expressly forbidden.

The publisher does not give any warranty express or implied or make any representation that the contents will be complete or accurate or up to date. The accuracy of any instructions, formulae and drug doses should be independently verified with primary sources. The publisher shall not be liable for any loss, actions, claims, proceedings, demand or costs or damages whatsoever or howsoever caused arising directly or indirectly in connection with or arising out of the use of this material.

Enterprise Information Systems
2009, 1–16, i*First* article

Taylor & Francis
Taylor & Francis Group

Architectural design for resilience

Dong Liu[a]*, Ralph Deters[a] and W.J. Zhang[b]

[a]*Department of Computer Science, 176 Thorvaldson Building, University of Saskatchewan, 110 Science Place, Saskatoon, SK, S7N 5C9, Canada;* [b]*Department of Mechanical Engineering, University of Saskatchewan, 57 Campus Drive, Saskatoon, SK, S7N 5A9, Canada*

(*Received 9 March 2009; final version received 26 May 2009*)

Resilience has become a new nonfunctional requirement for information systems. Many design decisions have to be made at the architectural level in order to deliver an information system with the resilience property. This paper discusses the relationships between resilience and other architectural properties such as scalability, reliability, and consistency. A corollary is derived from the CAP theorem, and states that it is impossible for a system to have all three properties of consistency, resilience and partition-tolerance. We present seven architectural constraints for resilience. The constraints are elicited from good architectural practices for developing reliable and fault-tolerant systems and the state-of-the-art technologies in distributed computing. These constraints provide a comprehensive reference for architectural design towards resilience.

Keywords: software architecture; resilience; reliability; fault tolerance; virtualisation; service-oriented architecture; redundancy; self-management

Downloaded By: [Canadian Research Knowledge Network] At: 21:25 3 March 2010

1. Introduction

The term 'resilience' was originally used to describe the property of a material that can absorb external energy when it is forced to deform elastically, and then be able to recover to its original form and release the energy. When 'resilience' is used for complex systems in the context of ecology, it refers to 'the capacity of a system to absorb disturbance and reorganise while undergoing change so as to still retain essentially the same function, structure, identity, and feedbacks' (Walker *et al*. 2004). For information systems, resilience refers to a system's capability to 'provide and maintain an acceptable level of service in the face of various faults and challenges to normal operation'.[1] Besides that, it also implies a system's capability to recover to normal operational state with all available resources (Ahmed and Hussain 2007). In this paper, resilience refers to the capability of an information system that sustains the same function with little degraded service level under unexpected workload or disturbance that can cause partial failures in the system, and then is able to recover to its normal service level easily when such unexpected situations disappear.

The resilience requirement for information systems can be classified into nonfunctional requirements categories that includes other properties like scalability, reliability, reusability, and maintainability. Many design decisions have to be made

*Corresponding author. Email: dong.liu@usask.ca

ISSN 1751-7575 print/ISSN 1751-7583 online
© 2009 Taylor & Francis
DOI: 10.1080/17517570903067751
http://www.informaworld.com

at the architectural level in order to fulfil those nonfunctional requirements (Gross and Yu 2001, Kruchten 1995). A software architecture is a set of architectural elements that are selected and organised according to particular constraints (Perry and Wolf 1992). Although every system's architecture is unique, there are still some common architectural elements and constraints shared by many systems. An architectural style is a collection of such elements and constraints that is abstracted from specific architectures (Perry and Wolf 1992). Architectural styles are very important for system architects to start quickly and avoid common pitfalls at the stage of requirements analysis and architecture design.

Resilience is relatively new as a requirement compared with other nonfunctional properties, and so far there are few references on how to design an information system's architecture so that it will have resilience property. On the other side, how to design a system with properties like scalability, reliability, and availability have been well studied. Some of those properties may potentially contribute to resilience, and others may conflict with it. This paper discusses the relationships between resilience and other architectural properties, and elicits architectural elements and constraints from the available architectures and architectural styles in order to derive a reference for resilience.

The rest of the paper is structured as follows. Section 2 discusses the relationships between resilience and other common nonfunctional requirements. Specially, we claim a conjecture about the pursuit of consistency, resilience, and partition-tolerance. In Section 3, we present the architecture style for resilience, and discuss the details of seven constraints. Related architectures and architectural styles are discussed in Section 4. Section 5 concludes the paper and discusses future work.

2. Resilience and other architectural properties

ISO 9126 specified five key quality attributes for software systems besides functionality – reliability, usability, efficiency, maintainability, and portability (Pressman 2004). There are also other quality factors such as reusability, scalability, security, accuracy, and fault-tolerance (Robertson and Robertson 2006, Cavano and McCall 1978). Many of these nonfunctional properties are strongly related to each other. In this section, we discuss the interdependency between resilience and scalability, reliability, reusability, manageability, and consistency.

2.1. Scalability

Scalability is a system's ability to sustain acceptable mean service time for increasing concurrent jobs by adding correspondingly proportional computing or operational resources into the system without affecting its reliability and other features (Luke 1993, Hill 1990, Bondi 2000). There are two ways to add resources to a system, namely, to scale vertically (scale up) and to scale horizontally (scale out). By scaling vertically, resources are added to a single node in the system, for example, replacing an old CPU in a machine with a new one of higher speed, or spawning more processes running in a web server. By scaling horizontally, more nodes are added to the system, for example, adding two more web servers to a system that used to have one web server. Vertical scaling is easier to achieve than horizontal scaling. However, vertical scalability is always bounded by the maximum capacity that a single node can reach. The idea of 'one size fits all' is overwhelmed by the pursuit of horizontal

Downloaded By: [Canadian Research Knowledge Network] At: 21:25 3 March 2010

scalability in every area from computing chips[2] to web servers and databases (Stonebraker and Cetintemel 2005).

The similarity between scalability and resilience is that both are a system's capability to maintain functionality in abnormal working situations. The difference is that scalability is about the capability of dealing with increasing workload by adding more resources into the system, while resilience is about tolerance with partial failures caused by unexpected workload. Resilience does not necessarily depend on new resources added into the system. If extra heavy workload will cause partial failures in a system, then the approaches contributing to scalability will also benefit resilience. In other words, resilience can be a system's capability to scale up or out automatically when partial failures caused by extra heavy load emerge, and to adjust the scale when its load returns to normal level.

Downloaded By: [Canadian Research Knowledge Network] At: 21:25 3 March 2010

2.2. *Reliability*

Reliability of systems is often described and measured as the probability of no failure within a given operating period (Shooman 2002). The reliability can also be derived via failure rate or hazard, which describes the probability that a failure happens instantly after a failure-free period t. If the failure rate is *constant*, then $R(t) = e^{-\lambda t}$, where $R(t)$ is the reliability of operating period t, λ is the failure rate. Sometimes, availability is used to quantify how reliable a system is. It is the probability that a system is available at any point of time. Simply, availability can be measured by $A = \frac{\text{availableTime}}{\text{availableTime}+\text{unavailableTime}}$. A system's reliability is determined by its critical components. There will be a reliability limitation for any component to reach due to physical or economical reasons. Meanwhile, a system's reliability can always be improved by removing single point of failure by introducing redundancy to the system. Note that too much redundancy will still increase the cost and management complexity of a system.

Reliability and availability both focus on how stable a system can be with the possibility of disturbance during a given period, and resilience emphasises more the abilities that a system can continue working in the face of partial failures, and that the system can be recovered from failures. A reflection of this difference is that resilience is about fault tolerance and fault recovery, but not fault avoidance. Generally, most approaches to improving system's reliability can be applied for resilience as well (Alsberg and Day 1976).

2.3. *Reusability*

In software engineering, the concept of reusability was originally about the reuse of code. Later, it was extended to all kinds of software artefacts including components and systems. The coupling between a component and its environment is often the obstacle from reusing it in a new environment. Therefore, decreasing the coupling between a component and its environment will increase its reusability. Coupling refers to 'the measure of the interdependency between two modules' in the context of structured program design (Page-Jones 1988). Two modules are coupled if one module calls the other (normal coupling), they share the same common data (common coupling), or one module refers to the inside of the other (content coupling). Common coupling and content coupling are considered bad designs because one module depends on the other so much that even a minor change of the

depended module may result in the dependent module not working properly. It is difficult to quantify the coupling between two modules in a multi-module program. The terms of 'loose' and 'tight' are used to qualify the degree of coupling.

In object-oriented programming, the callers of an object's operations are loosely coupled with the object's implementations by using interface (Booch *et al*. 2007). In distributed computing, two applications are loosely coupled when they communicate by message passing rather than memory sharing (Hariri and Parashar 2004). In web services, a service and its consumer are loosely coupled by explicit message passing than implicit method invocation (Kaye 2003). There are also examples of loose-coupling in everyday life. Lego blocks are loosely coupled, and any block can be easily connected to other block in the same set. Electrical devices are loosely coupled with power supply, and they can be plugged into standard outlets to get power. An effective design principle to achieve loose-coupling is to emphasise and make use of the simplest and standardised interfaces.

Reusability of components makes it possible that a system with a failed component can continue its functionality by substituting that component with a different one. By specially engineering components of a system, serendipitous reuse is possible (Vinoski 2008b), and the emergent nature of such 'serendipity' is exactly what we expected for resilience.

Downloaded By: [Canadian Research Knowledge Network] At: 21:25 3 March 2010

2.4. *Manageability*

Manageability refers to the capability of a system to be managed. A system's manageability is determined by how much a system's state can be reported and adjusted, and how easy the interaction with the system can be conducted. Manageability of a system depends on the manageability of its subsystems and components. It is impossible to manage a large-scale system if its subsystems offer no manageability. Built-in manageability of components is the basis for developing self-managing systems.

As discussed in Section 2.2, a difference between resilience and reliability is that a system needs to be able to cope with possible failures and recover by itself. The detection of failures and actions towards recovery requires that a system has the ability to manage itself. A system's manageability is a precondition for the self-managing ability required by resilience.

2.5. *Consistency, Brewer's conjecture, and resilience*

In the context of databases, consistency together with atomicity, isolation, and durability (Haerder and Reuter 1983) are properties of a transaction. It refers that a database remains consistent after a transaction (Gray 1981). More generally, a strong sense of consistency means that all processes get the same representation of a system state after the state is updated by one of the processes no matter where a process resides in the network (Vogels 2009). In other words, if an information system is consistent, then its clients can always get accurate information at the same time.

Brewer (2000) made the following conjecture known as the CAP theorem.

Theorem 2.1: *It is impossible to have a network-based system that has all three properties of consistency, availability, and tolerance to network partitions.*

This conjecture was later formally proved by Gilbert and Lynch (2002). An implication of CAP theorem is that a system *cannot* be consistent if it has to be highly available and tolerant to network partitions. We apply this theorem to the case of resilience and derive the following corollary.

Corollary 2.2: *It is impossible for a system to have all three properties of consistency, resilience and partition-tolerance.*

This corollary implies that a tradeoff between consistency and resilience has to be made when a system is designed. For some situations, the availability of incomplete information is still better than total unavailability. Therefore, resilience should be preferred to consistency. Vogels (2009) reported that availability was more desired than consistency in the design of Amazon web services.[3] The rationale behind this design decision is that AWS customers prefer the combination of high availability and weak consistency (eventual consistency) to strong consistency and low availability.

Downloaded By: [Canadian Research Knowledge Network] At: 21:25 3 March 2010

3. Architectural constraints for resilience

As discussed in Section 1, an architecture is composed of elements and constraints. The elements in a system can be classified into three categories – processing elements, data elements, and connecting elements. A constraint describes how elements should be selected and organised, and also the rationale behind such decisions. In this section, we present seven constraints for resilience, respectively, redundancy, partition, virtualisation, decentralised control, explicit messaging, uniform interface, and self-management.

3.1. Redundancy

Redundancy is one of the most fundamental approaches to achieving reliability (Siemwiorek 1991), and also to achieving resilience (Alsberg and Day 1976). In most cases, redundancy refers to deploying into a system more components than are required for the functionality. Functional redundancy refers to functional overlap of different components. Functional redundancy makes it possible to substitute a failed component with a different one in order to recover the whole or part of lost functionality. We call this serendipitous redundancy by reuse, and discuss it later in this subsection.

The most common scenarios of redundancy in information systems are data redundancy. All forms of data can be redundant – files, databases, and objects in memory. Data and its replications can reside in the same machine or in different locations across machines, networks, and geographical areas. In a server–client system, data and its replications can all be on server side, or some on server side and some on client side, for example, client-side cache. Redundancy of data increases data's availability in case of failures due to unexpected heavy load, data node failures or lost connection between client and server.

Similar to data elements, processing element can also be redundant in a system. This form of redundancy is also very common in enterprise information systems, for example, clustering of web servers and application servers. However, redundancy of processing elements is more difficult than data elements because of

the problem of state. If a processing element needs to maintain state for an undergoing computation task, then it is difficult to make the processing element truly redundant. For if the processing element fails, there is no way to recover the task state that only exists in that element. There are two approaches to addressing this problem – to make the task stateless or to restart the whole task from its initial state in case of failure.

A common way to achieve redundancy is to create replicas of an element. The structure and functionality of a replica are exactly the same as that element so that they can substitute each other without problem. Besides simple replication, there are two other approaches to redundancy. The second approach is known as n-version programming (Chen and Avizienis 1995, Knight and Lev 1986). At least two programs or components are *independently* developed for the same specification in n-version programming. Different versions are sensitive to different system failures because of their inter-independence, and therefore n-version programming can make a system tolerant to more unexpected failures. We name the third approach 'serendipitous redundancy by reuse', since we cannot find a proper name for it in literatures. Serendipitous redundancy happens when the function of a failed component can be *partially* substituted with a different component that was designed to achieve a totally different goal from the failed one. Serendipitous redundancy is possible only if (1) those two components share some common functionality, and (2) their interfaces to provide the common functionality are the same. The constraint of uniform interface discussed later in this section contributes to the second condition.

A redundant element in a system can work in two different modes – active or passive. In active mode, a redundant element shares workload with other elements. On the contrary, a redundant element will not take any workload until it is required to switch to working state because of detected failures. A redundant element needs to work in a passive mode when it is updated in order to avoid the failure happening to a previous active element. The system will recover from a failure and furthermore be immune to the same failure when the updated element is switched to on line and the old failed element is substituted. Figure 1 shows the transition between active and passive for two redundant elements, A and B, in a system that can detect failures and recover from failures.

Downloaded By: [Canadian Research Knowledge Network] At: 21:25 3 March 2010

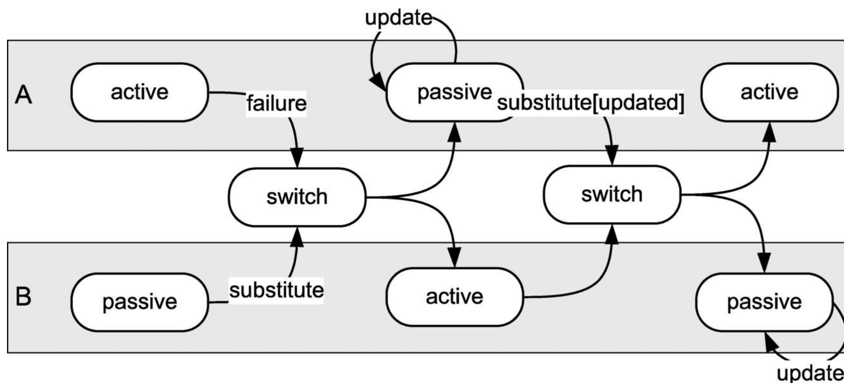


Figure 1. The state transition of two redundant elements, A and B, when a failure in A is detected.

When redundancy is applied to achieve resilience, the designer has to decide a proper redundancy degree for critical elements. The costs of both initialisation and reconfiguration will be high for highly redundant systems (Anderson and Bartholdi 2000). The management of such a system will be difficult and costly especially when the redundant elements need to be synchronised from time to time. For example, the replication of a large database takes a long time and also uses a large network bandwidth. The possibility of inconsistency between elements will be increased with the degree of redundancy.

3.2. Partition

Partition is a common way to increase the reliability of a database by splitting the data in a database into small pieces and storing them in distributed databases. Although the reliability of each database remains the same, the reliability and availability of data as a whole can be improved. The failure of one database will not affect the availability of the data in other databases. Partition also yields performance gains since queries of different tables can be paralleled easily. Partition provides an approach for partial failure isolation that is essential to resilience. In some cases, the unavailable data due to failure can even be derived from the available data based on predefined relationship between them.

The difference between redundancy and partition is shown in Figure 2 when the nodes are used to increase the resilience of a database. Redundancy and partition can be applied to a database at the same time.[4]

3.3. Virtualisation

The functionalities of either a processing element or a data element can be abstracted as a service. A service refers to a concrete autonomous technical functionality of retrieving and processing information. Most services are not easy to access and integrate across platform, language, or ownership borders. Service-orientation is a paradigm of virtualising the functionalities into services that enable easy access and integration by enforcing explicit boundary and schema-based interfaces (Box 2004).

Software can be virtualised into a service, which is also known as 'software as a service' or SaaS (Greschler and Mangan 2002). The functionality that used to be

Downloaded By: [Canadian Research Knowledge Network] At: 21:25 3 March 2010

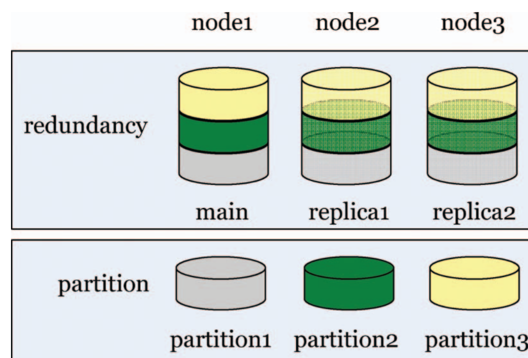


Figure 2. Redundancy and partition of a database in three nodes.

provided by a component in a system can be replaced by a service provided by third parties via the SaaS paradigm. Furthermore, it is possible for a system to choose the best from a set of services providing the same functionality and even switch to other service at runtime if the current service is satisfied, for example, a failure happens. The loose-coupling between a system and its consumed services makes the system more resilient because it is immune to failures of any single service.

The application of virtualisation into infrastructures yields cloud computing or 'infrastructure as a service' (Weiss 2007). The cloud provides services for almost every aspect of enterprise information systems including data service (Amazon S3,[5] Amazon SimpleDB,[6] Google BigTable (Chang *et al.* 2006)), messaging service (Amazon SQS[7]), application hosting service (Google App,[8] Microsoft Azure[9]) and operating system hosting service (Amazon EC2[10]). Via cloud computing, data elements, processing elements, and further any form of *computation* can be virtualised. New resources can be allocated easily and quickly to replicate elements or substitute failed elements in the cloud. It is costly to maintain an in-house infrastructure with the ability to allocate resources on demand. The cloud can dramatically lower such cost in the way of utility computing. The elastic characteristics of cloud economically and technically make it feasible to develop computation infrastructure with resilience property (Armbrust *et al.* 2009).

3.4. Decentralised control

The conceptualisations of (de)centralisation are related to the concept of distance. For computing systems, the distance does not refer to physical distance, but hierarchical difference within a system or position difference within a network topology. Control refers to the management activities of an element like configuring, healing, optimising, and protecting. It is difficult to implement a centralised control mechanism for a system of heavy redundancy because of the high communication and processing load on the central control component. Such mechanism could lead to catastrophic failure for the whole system when a partial failure of the central control component happens. Decentralised control is a widely used strategy for such systems, for example peer-to-peer systems, to achieve reliability and fault-tolerance. Amazon web services platform relies on a decentralised control mechanism to detect failure nodes and maintain node membership (DeCandia *et al.* 2007).

A system applying decentralised control paradigms can easily reach several local optimal solutions, and it is hard for such a system to check which solution is the global optimal solution. The systems are sometimes trapped in locally optimised situations, and cannot get out without outside interferences. The 'circular mill' of army ants is a typical example for the local-optimisation issue. For army ants that are blind and move by following the ant ahead, an isolated group of ants may form a circle which will get larger and larger until the ants die of starvation (Schneirla 1971) (cited by Anderson and Bartholdi (2000)). Figure 3 shows the 'circular mill'.

The 'circular mill'-like situation also happens in real computing systems with heavy redundancy and decentralised control. On 20 July 2008, Amazon S3 system was unavailable from 8:40am PDT, and did not return to normal operation until 4:58pm PDT.[11] The unavailable period of more than 8 hours meant S3's service level promise of 99.99% availability[12] failed for more than 20 billion objects[13] stored in S3 in a month cycle. The error rates reported by the nodes in datacentres kept increasing and reached a point that implied no storage resource could be located for new

Downloaded By: [Canadian Research Knowledge Network] At: 21:25 3 March 2010

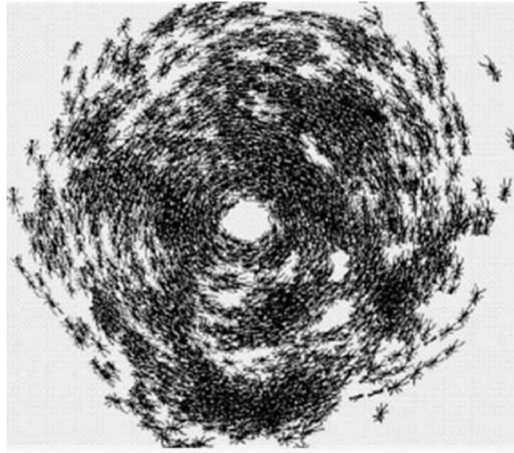


Figure 3. A circular mill of army ants (Schneirla 1971) (cited by Anderson and Bartholdi (2000)).

Downloaded By: [Canadian Research Knowledge Network] At: 21:25 3 March 2010

requests. Later investigation showed that it was not because the nodes really failed, but because the working status report of nodes was wrong. S3 depended on a gossip-based protocol for distributed failure detection, and a node was considered temporarily unavailable if it failed to forward requests. A major problem happened when the forwarded messages were corrupted. Each node trying to forward such a message was considered temporarily unavailable. Things got worse when many nodes forwarded such messages. It was like a 'rumour' keeping going around the system and never stopping. Obviously, correcting the state of only part of the system could not prevent the 'rumour' from coming back. Amazon S3 team had to shut down the communication between all S3 nodes, and then reactivate it. That process took about 6 hours because of S3's system scale. This lesson is significant for design of all decentralised systems.

3.5. *Explicit messaging*

Enterprise information systems with resilience are normally distributed systems. Distributed systems can be roughly classified into two categories by communication methods: distributed shared memory, and message passing (Hariri and Parashar 2004). Distributed shared memory makes synchronisation between processing easier than message passing (Coulouris *et al*. 2005), and therefore yields better consistency. However, a failed process will affect all other processes that share memory with it. Message passing paradigm, on the contrary, does not suffer for this because two processes sending and receiving messages do not rely on the same memory space.

Distributed object technologies like CORBA and RMI are implementations of RPC (Remote Procedure call) in objects. Distributed objects communicate with each other like local object method invocation via *implicit* messages. Waldo *et al*. (1994) argued that distributed object technologies have problem to deal with situations like network latency and partial failure. The implicitness of RPC messaging leads to system flaws in a distributed environment (Vinoski 2008a). The answer to this problem is explicit messaging, that is, messages are self-descriptive. Explicit

messaging makes message sender, message receiver, message intermediaries loosely coupled with each other. A computation task can be recovered based on the available messages if a processing element fails in the route of messaging.

3.6. *Uniform interface*

The term 'uniform interface' came from Fielding (2000) in his research work on network-based software architectures. Uniform interface brings better scalability, reusability and reliability to a network-based system like the Web. The Web is fully resilient in the sense that partial failures of any individual server, user agent, DNS server, proxy or gateway will not affect the availability of the Web or at least the rest of the Web.

In this paper, uniform interface refers to the following requirements that are a little different from those in REST (Fielding and Taylor 2002, Richardson and Ruby 2007).

- Data, process, and other forms of computations in general are identified by one mechanism, and they can be accessed via their identifiers.
- The semantics of operations in the messages for accessing data, process, or other forms of computations are unified.

Note that uniform interface does not imply that a system can only rely on some public standards like HTTP. A system-specific protocol can also satisfy the uniform interface constraint.

3.7. *Self-management*

As discussed in previous subsections, a system with resilience tends to be complex because of redundancy and decentralised control. The complexity of such systems brings challenges for management. On the other side, what makes resilience different from reliability is the capability of restraining the effects of failures and recovery from failures. An architectural strategy is self-management, which is the core concept of autonomic computing. Autonomic computing is to develop self-managing systems that can self-configure, self-heal, self-optimise, and self-protect (Kephart and Chess 2003). A self-managing system is composed of self-managing components. The conceptual structure of a self-managing component is shown in Figure 4. A self-managing component includes two parts: a managed element, and a manager. The manager is able to access the state of the managed element through the sensors bounded on the element. The manager monitors the state of the managed element, analyses the obtained information, plans the actions that the managed element should take, and then executes them by the effectors on the managed element. The monitor-analyse-plan-execute control loop is based on the knowledge about the managed element.

The principle of self-management is applied in many systems management approaches (Dan *et al*. 2004, Sahai *et al*. 2001, Tosic *et al*. 2006, Wang *et al*. 2004), though the term 'autonomic computing' may not be used. The congestion control mechanism of TCP (Transmission Control Protocol) (Padhye *et al*. 1998) is an application of self-management in maximising TCP performance on unpredictable network.

Downloaded By: [Canadian Research Knowledge Network] At: 21:25 3 March 2010

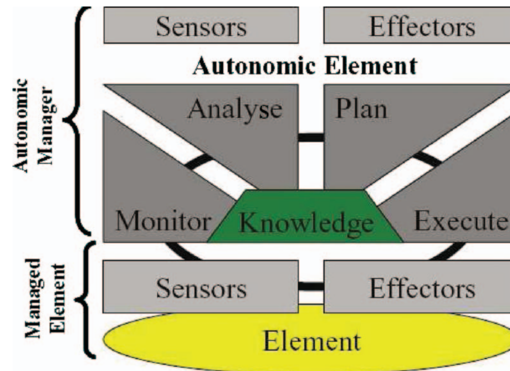


Figure 4. The conceptual structure of a self-managing component (IBM 2003).

Downloaded By: [Canadian Research Knowledge Network] At: 21:25 3 March 2010

4. Related architectures and architectural styles

The architectural constraints for resilience discussed in previous section are either directly derived from or inspired by the architectures and architectural styles reviewed in this section. In practice, many systems based on these architectures or styles show more or less resilience.

4.1. SOA and the cloud

There are many definitions of the service-oriented architecture. The OASIS SOA reference model technical committee defines it as 'a paradigm for organising and utilising distributed capabilities that may be under the control of different ownership domains' (Estefan *et al.* 2008). Box (2004) summarised the essential principles of SOA into four tenets:

- *Boundaries are explicit*. Explicit service boundaries need communication by explicit message passing rather than implicit method invocation. Explicit service boundaries reduce the cost of communication across service and organisation boundaries.
- *Services are autonomous*. A service is self-contained. A consumer does not need any assumption about a service's dependency with other parties.
- *Services share schema and contract, not class*. Services interact by message passing. Message structures are specified by schemas, and message-exchange behaviours are specified by contracts.
- *Service compatibility is determined based on policy*. The semantic compatibility of a service, i.e. its capabilities and requirements, is based on explicit policies.

The loose-coupling between a service and its consumer brings to a service-oriented system the flexibility to achieve functionality by binding to different services. Therefore, a system is able to switch to different services in runtime, which reduces the unavailability risk due to failures of any specific service. The constraints of virtualisation and explicit messaging are important principles of SOA. Di Marzo Serugendo *et al.* (2007) proposed a model for dynamically resilient system based on SOA.

The emergence of public clouds like Amazon web services, Google App, and Microsoft Azure has refreshed the popular perception of service-oriented computing and utility computing (Armbrust *et al.* 2009). The dream of *computation as a utility* comes true in the form of infrastructure as a service. The success of Amazon EC2 shows the possibility to develop large-scale elastic and resilient systems via virtualisation and service-orientation. These clouds also provide technical and economical basis for the systems that can allocate redundant computation resource on demand.

4.2. REST

Representational State Transfer (REST) is an architectural style derived by Fielding (2000) that is used to guide the development of Hypertext Transfer Protocol (HTTP) (Berners-Lee *et al.* 1996, Fielding *et al.* 1999) and Uniform Resource Identifier (URI) (Berners-Lee 1994, Berners-Lee *et al.* 2005). The modern Web can be considered as an instance of REST. The Web is the most popular network-based system of significant scalability and reliability in the history. The core concept in REST is resource, and the base styles that REST is derived from include replicated repository, cache, client–server, layered system, stateless, virtual machine, code on demand and uniform interface. Uniform interface makes REST different from RPC-based or SOAP-based approaches, and brings visibility, simplicity, cacheability, and scalability to a distributed system. The constraint of uniform interface is detailed by the following list:

- Resources are identified by one resource identifier mechanism.
- Access methods are the same for all resources.
- Resources are manipulated by exchanging representations.
- Representations are in self-descriptive messages.
- Hypermedia acts as the engine of application state.

Erenkrantz *et al.* (2007) extended REST to Computational REST (CREST) by adding a new constraint as the following:

- The representation of a resource is a computation that can be a program, a closure, a continuation, or a computation environment and a document describing the computation.

Via CREST, the virtualisation of data and processing can be unified.

4.3. Actor and Erlang style

The actor (Hewitt *et al.* 1973, Agha 1985) provided a generic model for reliable and scalable distributed computing. The constraints for each distributed computation entity, or an actor, include:

- Actors communicate by asynchronous message passing.
- Each actor has a mailbox for incoming messages.

The first constraint ensures that no shared variables are needed for actors to collaborate. The second constraint enables an actor to do asynchronous processing.

Downloaded By: [Canadian Research Knowledge Network] At: 21:25 3 March 2010

The actor model provided a formal reference for distributed computing with high performance, scalability, and fault-tolerance.

These styles of the actor model were carried forward by Erlang/OTP,[14] a concurrency-oriented programming language and libraries original developed by Ericsson and now open-sourced. An Erlang-based system, the AXD301 switch by Ericsson, achieved 99.9999999% (nine 9s) reliability (Armstrong 2007), which is equivalent to about only 3 seconds downtime in a year. It is not by failure avoidance that the system achieved such high reliability but by isolation of partial failures and recovery from partial failures. Armstrong (2003), one of the major contributors of Erlang/OTP, summarised the design principles of Erlang towards reliable distributed system in the presence of software errors as follows:

- Computation is virtualised as processes.
- Processes running on the same machine are isolated.
- Each process is identified by a unique unforgeable identifier.
- Processes share no state (memory), and asynchronous message passing is the only way for processes to communication.
- Unreliable message passing is never exceptional. There is no guarantee of delivery.
- The failure of a process can be detected by another process, and the failure reason is described in a failure message.

A significant difference between the Erlang-style concurrency model and the actor model is that the guarantee of message delivery is assumed by the actor model but not the case in the other. This difference makes the Erlang model more attractive for developing systems of high reliability and resilience. Resilience can be achieved only if no extra reliability assumptions are made in the design.

5. Conclusions

The discussions about architectural constraints for resilience and the relationships between resilience and other architectural properties lead to the following two points of view:

Design for resilience is hard. The difficulties of designing systems of resilience are raised from both theoretical and practical perspectives. In theory, to design a system that is able to response to unexpected failures in a 'predictable' way will lead to *nondeterminism*. In practice, the facts that partial failures can happen in a number of components, the failure rate can be extremely small, and the distribution of time to failure is unknown lead to *uncertainty*. For example, there is no way for Amazon AWS team to figure out that corruption of gossip messages could result in a system-wide failure as the 20 July 2008 event in the design or verification phase of the S3 system.

Tradeoffs are needed. As discussed in Section 4, to develop a system with three properties of consistency, resilience and partition-tolerance is technically impossible. The tradeoff between resilience and other properties like cost, performance, scalability, maintainability, and security has to be made in design. Another tradeoff

Downloaded By: [Canadian Research Knowledge Network] At: 21:25 3 March 2010

is needed for the selection of available architectural constraints as discussed in Section 3, since each of them has both pros and cons.

Although resilience is relatively new and at the same time hard to achieve, practices and lessons learned from architectural design for other properties can still help to tackle this problem. The architectural constraints summarised in this paper form a comprehensive reference for architectural design towards resilience.

Acknowledgements

This work is partially supported by NSERC Strategic Project Grant on Resilience Engineering (STPGP-351342-20). We would like to thank anonymous reviewers for their valuable comments.

Notes

1. See https://wiki.ittc.ku.edu/resilinets_wiki/index.php/Definitions.
2. See ftp://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf.
3. See http://aws.amazon.com/.
4. See http://blog.maxindelicato.com/2008/12/scalability-strategies-primer-database-sharding.html.
5. See http://aws.amazon.com/s3/.
6. See http://aws.amazon.com/simpledb/.
7. See http://aws.amazon.com/sqs/.
8. See http://code.google.com/appengine/.
9. See http://www.microsoft.com/azure/default.mspx.
10. See http://aws.amazon.com/ec2/.
11. See http://status.aws.amazon.com/s3-20080720.html.
12. See http://aws.amazon.com/s3-sla/.
13. See http://aws.typepad.com/aws/2008/10/amazon-s3---now.html.
14. See http://erlang.org/.

References

Agha, G.A., 1985. *ACTORS: A model of concurrent computation in distributed systems*. Thesis (PhD). MIT.

Ahmed, A. and Hussain, S.S., 2007. *Meta-model of resilient information system*. Master's thesis. Blekinge Institute of Technology.

Alsberg, P.A. and Day, J.D., 1976. A principle for resilient sharing of distributed resources. *In*: *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, San Francisco, California, United States. Los Alamitos, CA, USA: IEEE Computer Society Press, 562–570.

Anderson, C. and Bartholdi, J.J., 2000. Centralized versus decentralized control in manufacturing: Lessons from social insects. *In*: I.P. McCarthy and T. Rakotobe-Joel, eds. *Complexity and Complex Systems in Industry*, September. UK: The University of Warwick, 92–105.

Armbrust, M., *et al*., 2009. Above the clouds: A Berkeley view of cloud computing. Technical report UCB/EECS-2009-28, EECS Department, University of California, Berkeley.

Armstrong, J., 2003. *Making reliable distributed systems in the presence of software errors*. Thesis (PhD). The Royal Institute of Technology, Stockholm, Sweden.

Armstrong, J., 2007. *Programming Erlang: Software for a Concurrent World*. Raleigh, North Carolina: Pragmatic Bookshelf.

Berners-Lee, T., 1994. Universal Resource Identifiers in WWW. [online] http://tools.ietf.org/rfc/rfc1630.txt

Berners-Lee, T., Fielding, R., and Frystyk, H., 1996. Hypertext Transfer Protocol – HTTP/1.0. [online] http://tools.ietf.org/rfc/rfc1945.txt

Berners-Lee, T., Fielding, R., and Masinter, L., 2005. Uniform Resource Identifier (URI): Generic Syntax. [online] http://tools.ietf.org/rfc/rfc3986.txt

Downloaded By: [Canadian Research Knowledge Network] At: 21:25 3 March 2010

Downloaded By: [Canadian Research Knowledge Network] At: 21:25 3 March 2010

Bondi, A.B., 2000. Characteristics of scalability and their impact on performance. *In*: *WOSP '00: Proceedings of the 2nd international workshop on Software and performance*, Ottawa, Ontario, Canada. New York, NY, USA: ACM Press, 195–203.

Booch, G., *et al.*, 2007. *Object-Oriented Analysis and Design with Applications*, 3rd ed. Redwood City, CA, USA: Addison-Wesley Professional.

Box, D., 2004. Code Name Indigo: A Guide to Developing and Running Connected Systems with Indigo. *MSDN Magazine*. http://msdn.microsoft.com/msdnmag/issues/04/01/indigo/default.aspx

Brewer, E.A., 2000. Towards robust distributed systems (abstract). *In*: *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, Portland, Oregon, United States. New York, NY, USA: ACM, p. 7.

Cavano, J.P. and McCall, J.A., 1978. A framework for the measurement of software quality. *In*: *Proceedings of the software quality assurance workshop on Functional and performance issues*, ACM Press, 133–139.

Chang, F., *et al.*, 2006. Bigtable: a distributed storage system for structured data. *In*: *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, Seattle, Washington Berkeley, CA, USA: USENIX Association, 205–218.

Chen, L. and Avizienis, A., 1995. N-version programming: A fault-tolerance approach to reliability of software operatlon. *In*: *Twenty-Fifth International Symposium on Fault-Tolerant Computing, June, 1995, 'Highlights from Twenty-Five Years'*, June, 113–119.

Coulouris, G., Dollimore, J., and Kindberg, T., 2005. *Distributed Systems: Concepts and Design*, 4th ed. fourth Redwood City, CA, USA: Addison Wesley.

Dan, A., *et al.*, 2004. Web services on demand: WSLA-driven automated management. *IBM Systems Journal, Special Issue on Utility Computing*, 43 (1), 136–158.

DeCandia, G., *et al.*, 2007. Dynamo: amazon's highly available key-value store. *In*: *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, Stevenson, Washington, USA. New York, NY, USA: ACM, 205–220.

Di Marzo Serugendo, G., *et al.*, 2007. A metadata-based architectural model for dynamically resilient systems. *In*: *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, Seoul, Korea. New York, NY, USA: ACM, 566–572.

Erenkrantz, J.R., *et al.*, 2007. From representations to computations: the evolution of web architectures. *In*: *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, Dubrovnik, Croatia. New York, NY, USA: ACM, 255–264.

Estefan, J.A., *et al.*, 2008. Reference Model for Service Oriented Architecture V 1.0. [online] http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf

Fielding, R., *et al.*, 1999. Hypertext Transfer Protocol – HTTP/1.1. [online] http://www.ietf.org/rfc/rfc2616.txt

Fielding, R.T., 2000. *Architectural styles and the design of network-based software architectures*. Thesis (PhD). University of California, Irvine.

Fielding, R.T. and Taylor, R.N., 2002. Principled design of the modern Web architecture. *ACM Transactions on Internet Technology*, 2 (2), 115–150.

Gilbert, S. and Lynch, N., 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33 (2), 51–59.

Gray, J., 1981. The transaction concept: virtues and limitations (invited paper). *In*: *VLDB '1981: Proceedings of the seventh international conference on Very Large Data Bases*, Cannes, France VLDB Endowment, 144–154.

Greschler, D. and Mangan, T., 2002. Networking lessons in delivering 'Software as a Service'—part I. *International Journal of Network Management*, 12 (5), 317–321.

Gross, D. and Yu, E., 2001. From non-functional requirements to design through patterns. *Requirements Engineering*, 6 (1), 18–36.

Haerder, T. and Reuter, A., 1983. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15 (4), 287–317.

Hariri, S. and Parashar, M., 2004. Tools and Environments for Parallel and Distributed Computing. Somerset, NJ: Wiley-Interscience.

Hewitt, C., Bishop, P., and Steiger, R., 1973. A universal modular ACTOR formalism for artificial intelligence. *In*: *IJCAI*, 235–245.

Hill, M.D., 1990. What is scalability? *ACM SIGARCH Computer Architecture News*, 18 (4), 18–21.

IBM, An Architectural Blueprint for Autonomic Computing, 2003. Technical report http://www.ibm.com/developerworks/autonomic/library/ac-summary/ac-blue.html.

Kaye, D., 2003. *Loosely Coupled: The Missing Pieces of Web Services*. Marin County, Calif: RDS Press.

Kephart, J.O. and Chess, D.M., 2003. The vision of autonomic computing. *IEEE Computer*, 36 (1), 41–50.

Knight, J.C. and Lev, N.G., 1986. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, 12, 96–109.

Kruchten, P., 1995. Architectural blueprints—The '4+1' view model of software architecture. *IEEE Software*, 12 (6), 42–50.

Luke, E., 1993. Defining and measuring scalability. *In*: *Proceedings of the Scalable Parallel Libraries Conference, 1993*. IEEE, 183–186.

Padhye, J., *et al.*, 1998. Modeling TCP throughput: a simple model and its empirical validation. *In*: *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, Vancouver, British Columbia, Canada. New York, NY, USA: ACM, 303–314.

Page-Jones, M., 1988. *The practical guide to structured systems design*, 2nd ed. Englewood Cliffs, NJ: Yourdon Press.

Perry, D.E. and Wolf, A.L., 1992. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17 (4), 40–52.

Pressman, R., 2004. *Software Engineering: A Practitioner's Approach*, 6th ed. Ontario, Canada: McGraw-Hill Higher Education.

Richardson, L. and Ruby, S., 2007. *RESTful Web Services*. Sebestopal, CA: O'Reilly.

Robertson, S. and Robertson, J., 2006. *Mastering the Requirements Process*, 2nd ed. Redwood City, CA, USA: Addison Wesley Professional.

Sahai, A., Durante, A., and Machiraju, V., 2001. Towards Automated SLA Management for Web Services. Technical report, HP Laboratories Palo Alto http://www.hpl.hp.com/techreports/2001/HPL-2001-310R1.pdf

Schneirla, T.C., 1971. *Army Ants. A Study in Social Organization*. San Francisco: W.H. Freeman & Co.

Shooman, M.L., 2002. *Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design*. Malden, MA: Wiley-Interscience.

Siemwiorek, D., 1991. Architecture of fault-tolerant computers: an historical perspective. *Proceedings of the IEEE*, 79 (12), 1710–1734.

Stonebraker, M. and Cetintemel, U., 2005. "One size fits all": An idea whose time has come and gone. *In*: *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2–11.

Tosic, V., Lutfiyya, H., and Tang, Y., 2006. Extending Web Service Offerings Infrastructure (WSOI) for management of mobile/embedded XML web services. *In*: *The 8th IEEE International Conference on and Enterprise Computing, E-Commerce, and E-Services, The 3rd IEEE International Conference on E-Commerce Technology*, 87–94.

Vinoski, S., 2008a. Convenience over correctness. *IEEE Internet Computing*, 12 (4), 89–92.

Vinoski, S., 2008b. Serendipitous reuse. *IEEE Internet Computing*, 12 (1), 84–87.

Vogels, W., 2009. Eventually consistent. *Commun. ACM*, 52 (1), 40–44.

Waldo, J., *et al.*, 1994. A Note on Distributed Computing. Technical report, Sun Microsystems Laboratories http://research.sun.com/techrep/1994/smli_tr-94-29.pdf

Walker, B., *et al.*, 2004. Resilience, adaptability and transformability in social–ecological systems. *Ecology and Society*, 9, 5.

Wang, G., *et al.*, 2004. Integrated quality of service (QoS) management in service-oriented enterprise architectures. *In*: *EDOC 2004: Proceedings of Eighth IEEE International Enterprise Distributed Object Computing Conference*, 21–32.

Weiss, A., 2007. Computing in the clouds. *netWorker*, 11 (4), 16–25.

Downloaded By: [Canadian Research Knowledge Network] At: 21:25 3 March 2010