UNIVERSITY OF WATERLOO
Faculty of Engineering
Nanotechnology Engineering

Self-Configuring and Self-Healing Web Services in Complex Software Systems

Special Projects Group
University of Waterloo
Waterloo, ON

Prepared By:

Rajesh Kumar Swaminathan
2B Nanotechnology
ID #20194189
rajesh@meetrajesh.com

May 12, 2008

Rajesh Kumar Swaminathan
#27-8289 121A St.
Surrey, BC V3W 1G6

May 12, 2008

Dr. Marios Ioannidis, Director
Nanotechnology Engineering
University of Waterloo
Waterloo, Ontario N2L 3B9

Dear Dr. Ioannidis,

This report, entitled, "Self-Configuring and Self-Healing Web Services in Complex Software Systems" is my third work term report for the two terms immediately following 2B spanning the months of September 2007 to April 2008. This report was completed during my work term at the University of Waterloo as part of the Special Projects Group (SPG) responsible for delivering the better and improved version of the CECS IT project, i.e. Jobmine. The purpose of this report is to investigate what is involved in building autonomous to semi-autonomous web services that can self configure themselves (a process known as service *bootstrapping*), self-manage, and of course self-heal in the event of a malfunction. There may be many methods to achieve this goal, so the goal of the report is to analyze each method's pros and cons in the light of relevant constraints.

The idea for this topic was hinted to me by my supervisor Trevor Grove. Trevor was very interested in knowing the kinds of difficulties people have encountered in constructing autonomous software, and how they have traditionally approached them. He wanted to know the current state of the universe and what the world thought of hard problems such as service bootstrapping. Trevor was also interested in a detailed analysis of advanced topics such as software *robustness*, which subsumes self-healing of web services, and how concepts like roll-back and roll-forward can help make a software system more stable and consequently more reliable.

The topic of this report proved itself to not only be insightful and exciting, but a topic that, I hope, will be of immense usefulness to the SPG group and the software product we will be shipping in about a year or so. There are a lot of issues, benefits and outcomes to be discussed while writing software that runs, manages and heals itself autonomously; however, due to the size limitations of this report, I have restricted myself to a select few targeted topics that are of direct relevance to our applications design.

I vouch to the fact that I have received no further help other than what is mentioned above and in the references section in writing this report. I also confirm that this report has not been previously submitted for academic credit at this or any other academic institution.

Sincerely,

Rajesh Swaminathan
20194189

Contributions

As software developer for the Special Projects Group (SPG) here at the University of Waterloo, I worked on the new and improved version of the software underpinning co-op, i.e. Jobmine. I was primarily responsible for writing code for a crucial component of the software, called the "Configurator", that acted as the manager of all web services within the system.

Our major milestone goal for the eight-month term was to get out a demo portal up and running for students, employers, and CECS staff to look at and critique. We wanted to release our software incrementally so we could get feedback on an ongoing basis.

The Configurator, arguably was the "brain" of the entire software system. It was aware of all the web services up and running, the stacks and virtual machines they were running on. It would resolve dependencies between services, and it would take appropriate action if one of the services died or lost connectivity. The configurator would also make intelligent decisions to balance user load across multiple servers for maximum performance. This was critical since we wanted the new Jobmine system to be available 24/7.

I was also responsible for writing code that would help all services log their actions to a database. This helper code had to be very generic since it was to be used by all web services in the system, and would write all log requests to an internal queue, which would then get processed later.

Overall, I had completed most (∼90%) of the specified functionality of the Configurator over the period of 8 months.

Summary

The purpose of this report is to investigate what is involved in building autonomous to semi-autonomous web services that can self configure themselves (a process known as service bootstrapping), self-manage, and of course self-heal in the event of a malfunction. There may be many methods to achieve this goal, so the goal of the report is to analyze each method's pros and cons in the light of relevant constraints.

This report attempts to look at currently available technologies and to suggest improvements that help towards the overall goal of software robustness. Software robustness refers to the quality of the software that makes it available at all times, flexible, and most importantly, resilient. The report then goes through a detailed survey of existing techniques for dynamically composing a set of web services each offering specialized functionality: detection, monitoring, and resolution of unforeseen situations. The report then concludes with recommendations for implementing *self-healing* services. The report identifies and classifies major pitfalls in service-oriented architectures and advocates best practices to deal with them autonomously, thereby making them self-healing.

The report is split into four sections. Section 1 contains an introduction to the problem, why it is important, and the various issues involved in tackling it. Section 2 dives into the nuts and bolts of three major components of self-management: bootstrapping, monitoring and recovery. Section 3 contains an analysis of the core of this report: the concept of self-healing web services. The analysis contains a detailed specification of the various architectural requirements needed to implement a self-healing system and the common issues that come up during implementation. Section 4 wraps up the topic with a concluding summary of the various topics discussed throughout the report and is intended to complement this summary.

Conclusions

In conclusion, a useful approach to on-the-fly automatic error detection and recovery has been described in this report. The report has also described various important issues of designing a self-healing system based on available literatures. In order to develop an effective self-healing system model we must be conversant to all these points described in this report. The proposed approach is an important one toward self-healing software design, which is very much a research topic for developing reliable web services and web applications of the future.

Recommendations

Based on the analysis and conclusions in this report, the following recommendations are proposed.

1. Possibly change from a heartbeat system to a service broadcast system to identify dead services.
2. Implement a tightly-coupled .NET web service in front of each SQL Server cluster so that we can treat SQL Server as just another regular web service.

Table of Contents

Contributions iii

Summary iv

Conclusions v

Recommendations vi

List of Figures viii

List of Tables ix

1 Introduction 1

2 What is Self-Configuration? 2

2.1 Service Bootstrapping 3

2.2 Monitoring 4

2.3 Software Robustness and Recovery 5

3 Self-Healing Web Services 6

3.1 What is Self-Healing? 6

3.2 Implementation Issues 8

3.3 Architectural Requirements 9

3.4 Implementation Details 11

4 Concluding Summary 12

References 13

List of Figures

1 Nervous System Analogy Acting on a Stimulus 7

List of Tables

1 Common Software Faults and Fixes 11

1 Introduction

Almost all of today's modern software systems, due to the inherent complexity involved in them, need to cope with two major influences:

1. Dynamism
2. Flexibility

An analysis of the current situation with respect to web services can be conducted by looking at current technologies such as the Web Services Description Language (WSDL). WSDL and its associated counterparts have proven to be insufficient to address current requirements relating to dynamism and flexibility.

One way to make software systems much more robust is to split individual functionality into "web services" and to integrate them tightly into a highly heterogeneous system. This service-oriented approach is commonly known as service-oriented architecture (SOA). SOAs are highly flexible in the way they integrate components in an environment of constantly changing and evolving contexts. Therefore the biggest advantage to using SOAs is that they allow businesses to integrate their various components (services) across business boundaries in a flexible and coordinated fashion. SOAs can tackle a high level of dynamicity, and can therefore operate in very unstable, unpredictable, and evolving environments.

A web service, to put it in formal terms, is therefore a stand-alone piece of software that is highly specialized and is designed to perform one task and one task only. This web service may use the help of other services to perform its task or may delegate its own task to another service depending on the circumstances. The entire software system is then a pool of web services all interacting with one another to get the job done.

Requirements for a high degree of flexibility and dynamism mean that all available functionality need to be discovered dynamically at run-time. The parameters required for a service's proper functioning are negotiated on-the-fly with the other service(s) directly or through a middle-man, known as the "service broker", which holds all information about all currently running services.

There are many examples in the area of ambient computing and automotive applications that need to cope with constantly changing requirements and configurations. It would therefore be unwise to hard-code these configurations into the software itself, but instead, have the configuration parameters negotiated dynamically by querying the current state of the universe.

This report attempts to look at currently available technologies and to suggest improvements that help towards the overall goal of software robustness. Software robustness refers to the quality of the software that makes it available at all time, flexible, and most importantly, resilient. The report then goes through a detailed survey of existing techniques for dynamically composing a collection of web services each offering specialized functionality: detection, monitoring, and resolution of unforeseen situations. The report then concludes with recommendations for implementing *self-healing* services. The report identifies and classifies major pitfalls in service-oriented architectures and advocates best practices to deal with them autonomously, thereby making them self-healing.

2 What is Self-Configuration?

Self-Configuration is the ability for software to configure itself with respect to the policies in order to adapt to dynamically changing environments. The language in which the policies are specified must be versatile enough to be able to specify the goals of the configuration. [1]

Due to the highly dynamic nature of the aforementioned complex software systems, it is difficult, if even possible, to identify before deployment all the components that define a given system. This adds uncertainty to the system. For example, a given service may need to respond to a given request by spanning the request across multiple executions; however, this may not be always possible since the context is always changing therefore altering the set of available services. This means we have a new problem using service oriented architectures (SOAs): discovery of services also need to be considered and handled at run-time.

The high degree of dynamicity means that it is often difficult to analyze if a system is trustworthy before deployment. Instead, a set of methods and tools are pre-programmed that help enforce

trust during run-time. The entire service-oriented architecture is based on trust and black-box abstraction: each service is highly specialized and programmed to do its job best.

There are three major phases involved in the implementation of a highly robust system:

1. **The selection or composition phase**: Also known as *service bootstrapping*, the phase involves discovering available services, negotiating working parameters, and implementing desired behaviour. This step is necessary to compose services together to meet the desired goal of the software.
2. **The monitoring phase**: This phase involves understanding if a given service is behaving properly, both functionally and non-functionally. This phase is crucial to immediately knowing when a service is down so that appropriate action can be taken.
3. **The recovery phase**: This phase involves reacting to disastrous situations, unexpected failures, shutdowns, etc. in a suitable manner for recovery. This phase is the key component of self-healing services.

A service-oriented architecture implies that the composition of all available services implements all required functionality as specified by predefined goals. A key part of the service environment is to analyze, monitor, and enforce non-functional requirements as well.

2.1 Service Bootstrapping

Bootstrapping refers to the selection or composition phase where each service, upon coming up, announces itself to the world of its presence. It then proceeds to detect all other services that are required for its own functioning. These other services are known as service dependencies. The service then negotiates important working parameters with other services, configures itself with those parameters, at which point the service is in a "ready" state, available to process incoming requests.

If the selection phase aborts unexpectedly, simply shutting down the system is not, in general, going to be a solution. The environment should be such that new services should make the best

of the available services in this “service ecosystem”. Essentially, a solution needs to be found that uses what is available in the event a perfect match does not exist. This service may then permit calls only on a subset of its methods. Once all dependencies have been resolved at a later time, the service should then revert to offering its entire functionality.

2.2 Monitoring

The monitoring phase comprises of a set of specialized probes that allow a monitoring service to detect unexpected situations in the working of a service. This happens when a) a service does not answer to a call within a given time-frame, b) when a service does not fully implement a predefined contract, be it either functional or non-functional, or c) when a service responds with an exception or an error message. Each service must therefore implement a heartbeat which can be called by the monitoring service to ensure the service is responding to calls and is not dead.

Monitoring is crucial to *early* detection of problems. The earlier a problem is detected, the easier it is to be fixed before the errors spiral inward causing multiple services to malfunction. Upon detection of a problem, the service cannot be asked to simply shutdown. Since SOAs are based on trust, causing one service to shutdown would cause virtually all of the services to shutdown due their tight mutual inter-dependencies.

One of the easiest monitoring mechanism uses the idea of packet broadcasting. Each component of the system periodically broadcasts an “aliveness message”, either to its neighbours, or to a central monitoring service. The central monitoring service keeps track of when it last heard a reply from each of the components it thinks should be alive. If it doesn’t hear from a service for a set period of time, the monitoring service triggers some appropriate response. This is a fairly primitive mechanism, but works quite effectively. However there is catch: the frequency of these “aliveness messages” from each service determines the speed with which faults can be detected. This therefore puts a bound on the shortest time before which a response can be sent out.

2.3 Software Robustness and Recovery

Why is Robustness Important?

Complex systems are faced with the need to react appropriately to unexpected system crashes, power failure, network disconnect, software malfunction, etc. by suitable recovery actions that either compensate the anomalies or at the very least mitigate the effects of the unavailable service. The recovery actions must handle the deviation and allow execution to proceed normally. The difficult part is to allow any or all of this happen *autonomously*. This autonomous detection, prevention, and mitigation of service failure is known as *self-healing*.

Software does not have the problems that hardware does, such as rusting or oxidation. But software does fail. Software can be hacked and modified by other software. Software can be sensitive to bad programming and issues might not appear till a specific sequence of code is executed. Unforeseen situations can result in unexpected bottlenecks, buffer overflows, etc. Software that is out-of-spec can cause many types of problems ranging from annoying slowdowns to failures, crashes, and malicious actions such as destroying data or using the system to attack other systems. Unfortunately, software often does not obey a specific MTBF (mean time between failure), MTTR (mean time to repair) or the related mathematical probability distributions. Any change in the status quo can result in a possibly risky situation when it comes to software — doing anything different, or doing something that has not been done for an extended period of time has its potential risks. And this concept extends to the tools used to create the software too.

How should software robustness be approached?

In a mission critical system, each major subsystem or component of hardware and software should be reviewed for robustness. A list of the possible things that may go wrong in the system, along with the severity of these problems must be documented. These problems must be structured according to the system's major sections (e.g. data in databases, sequence of commands, etc). The unwanted effects of these mishaps should then be detailed, along with methods for their detection, procedures for recovery (should they occur), and procedures that should be followed to mitigate

any inevitable non-robustness must be clearly documented *before* deployment into a production environment.

Software robustness must be approached in a manner that manages to be specific with regards to the various sections of the system and mishaps that may occur, but the analysis must be strictly agnostic to the kind of technologies used in the final implementation. This ensures that the analysis is useful even if the technology stack was to change in the future.

3 Self-Healing Web Services

In a nutshell, "self-healing" means to recover from failures. A system should be able to notice an injury and then act on it. In the field of software, injury is identified usually as a failure of a participating machine. A failure can happen in its hardware or the software running on it or a malfunctioning communication connection such as a network defect. The software would then proceed to repair itself in the most logical way given the circumstances. The remaining parts should try to find a way to continue working without the faulty part until such time. [2]

This section deals with the various issues that spring up while designing a self-healing software application that relies on the on-the-fly error detection and repair of web services. Highly dynamic systems need a fair amount of fault tolerance toward buggy code and/or hardware, and autonomous healing is the first step towards this goal.

This section aims to illustrate the critical points of self-healing software system and what is involved in constructing one.

3.1 What is Self-Healing?

Self-healing simply refers to the process of dealing with bugs, uncertainties, crashes, failures, and unexpected shutdowns in a suitable manner *during run-time*. This can be achieved by automatically restarting the service that went down or by awakening another instance of the same service elsewhere in the stack.

At its very basics, self-healing deals with imprecise specification, environment uncertainties, and continuous system reconfiguration and evolution. Software capable of detecting and reacting to software malfunctioning is known as self-healing software. Such software has the ability to examine failures in the system and take appropriate measures. The best measure would be someway for the system to repair itself in a logical way. This is a huge jump from simply notifying the system administrator as system administrators are typically unavailable at 3 in the morning, and most modern software is expected to function 24 hours a day. To use the biological analogy, the desired response is not very much unlike that delivered by the nervous system to the muscles as seen in Figure 1.

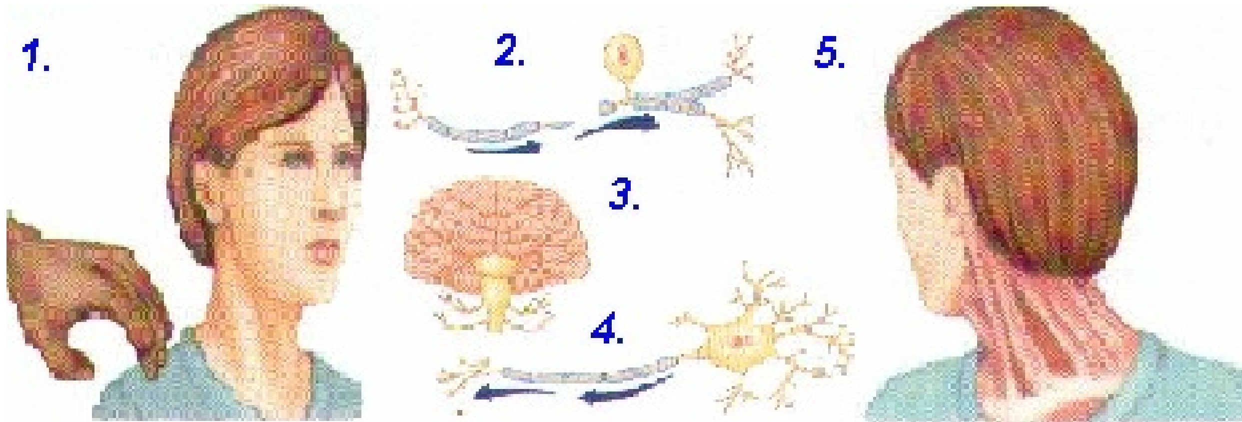

Figure 1: The nervous system acts in an appropriate and timely fashion upon receiving a stimulus. [3]

The software being constructed by the Special Projects Group is being designed as a mission-critical system that permits at most only one hour of downtime per year. Such a system needs to be extremely robust, capable of handling all but the most catastrophic of scenarios. To accomplish this, a thorough robustness analysis needs to be conducted to ensure that the system is capable of accomplishing this goal. While the system is not being designed for the proverbial 100 year storm, it is being designed for the 10 year storm scenario.

3.2 Implementation Issues

By its definition, a self-healing system will need to have full knowledge about its expected behaviour in order to examine whether its actual behaviour deviates from its expected behaviour with respect to the given environmental characteristics. A model needs to be constructed to address what specific faults or injuries will be dealt with based on the fault source, and what will be delegated to some other piece of software. For example, faults such as operational errors, uncaught exceptions, defective system requirements, or implementation errors, are most likely to be the categories of faults to be dealt with by self-healing software.

A self-healing system needs to be able to discover, diagnose, and try to react to bugs and failures immediately. Self-healing components need to have the ability to detect system malfunctions, upon which appropriate corrective actions are initiated without disrupting the environment. Such corrective action may involve a component changing its own state, or causing changes in other components of the system to make them match the state expected by a malfunctioning service. At this point, the rest of the system must continue to work in a "handicapped" state in the absence of the faulty component, either offering a reduced subset of functionality, or queuing its own requests to be processed at a later time.

There are many possible crashes in different systems, but the three most commonly occurring are defective hardware, crashed software, and broken network connectivity. [7] As noted previously, the amount of time before which a response can be sent out is limited by how frequently heartbeat requests are sent out. Faster response time comes with a higher cost of communication.

When a system crashes, all data on the system is lost. Therefore, all data needs to be copied in advance, so that a redundant copy can be fetched and used to continue work. One solution that quickly comes to mind is to duplicate all data, so that in case of a server crash, the lost data can be recovered from a backup server. However, to accomplish this, a copy of all data must be sent immediately over the network and this incurs much communication overhead and is therefore not a very feasible solution. An alternate approach is to store all applied data in memory and to redo the missing executions upon any data loss. However, this too would require a sophisticated mechanism

to decide which data can be deleted from memory and which data needs to remain.

In general, third-party components get in the way of self-healing services. It can be quite a challenge to develop a self-healing system that deals with the failure of a third-party component whose internal workings are hidden. The same can be said for quick patches that were applied after deployment.

In summary, the best self-healing approach is to validate at run-time a series of assertions. These assertions will be derived from a knowledge of application semantics, internal workings, and application domain specific knowledge of the system's expected behaviour. Self-healing is then achieved by examining whether the system's actual behaviour deviates from its expected behaviour in relation to the system's environment. The general idea is to detect errors on-the-fly in code and to autonomously recover those errors, thereby allowing a system to continue processing requests after it has sustained an error serious enough to require a restart.

3.3 Architectural Requirements

A self-healing system heals the system during times of distress by modifying its own behaviour at *run-time* in response to changes in its environment. The most important changes in the environment may be categorized into the following:

1. Resource variability
2. Changing user needs
3. Mobility
4. System errors

The most important tasks of a self-healing system may be described as follows [4]:

1. Monitoring the system at runtime
2. Planning the changes
3. Deploying the change descriptions
4. Enacting the changes

Given the above activities and characteristics of architectural issues (in Section 3.2), self-healing systems need to implement the following architectural requirements [5]:

1. **Awareness**: Self-healing software (SHS) must support monitoring of the individual components' health, heartbeat and performance. Various measurements relating to state, behaviour, correctness, reliability, etc. need to be measured and data collected. Comparisons of these measures with known "correct" measurements must be performed and any anomalies reported.
2. **Adaptability**: SHS must have the ability to change the system's structural, topological, behavioural, interactive, and run-time aspects.
3. **Dynamicity**: Must be able to adapt to situations during *run-time*.
4. **Autonomy**: Ability to plan, deploy, and enact necessary changes automatically without human intervention.
5. **Observability**: SHS must be able to monitor its environment for missing services and must also notice missing functionality reappearing. It must therefore be able to perform in degraded mode until the problem is addressed.
6. **Robustness**: SHS must effectively respond to a dead or disconnected service, or any other unforeseen operating circumstances. Various threats may be imposed by the system's external environment, namely malicious attacks, unpredictable behaviour, and unintended system usage. In addition, SHS must be able to respond to internal threats suchs as errors, faults, and failures within the system itself.
7. **Distributability**: SHS must be able to react nicely to varying traffic, and must be able to accommodate sudden spikes in usage by effectively load balancing requests across services. SHS must be able to start and stop redundant services automatically as desired.
8. **Mobility**: Self-healing systems must provide the ability to dynamically change, during *run-time*, the logical locations of a system's individual components.

3.4 Implementation Details

Reliability and ubiquity are absolutely critical in a typical mutli-tier computing infrastructure that uses a pool of web applications and web services in tandem. However, a number of failures occur nonetheless because of bugs in the source code, uncaught exceptions, thread deadlocking, etc. The most common faults and their corresponding remedies are detailed in Table 1.

Table 1: A list of common faults and fixes in traditional multi-tier architecture. [6]

Faults	**Fixes**
1. Source code bugs	Reboot Tier and/or service, send notification to administrator
2. Uncaught exceptions	Micro-boot service or component
3. Deadlocked threads	Micro-boot service, abort request
4. Buffer contention	Reparation the memory across various buffers
5. Aging	Reboot to reclaim leaked resources
6. Read/Write contention on table block	Repartition table to balance accesses ground partitions

Self-healing web applications need to deal with these kinds of failure, which can show up in both the system level as well as the logical level. Once the failure has been detected, the service needs to be able to make an optimal choice and solve the problem by reconfiguration. This "healing behaviour" must adhere to specified system-level policies. We can either cancel the request and try again at a another time hoping the problem fixes itself, or we can abandon the service and try to find another service that would process the request.

Self-healing software also needs to be equipped with a good test harness and test data to ensure the system responds to stated faults while keeping system-level policies in mind. We could produce a set of assertions based on the application's semantics. These assertions can then be validated during run-time against a known set of data.

4 Concluding Summary

A useful approach to on-the-fly automatic error detection and recovery has been described in this section. The report has also described various important issues of designing a self-healing system based on available literatures. In order to develop an effective self-healing system model we must be conversant to all these points described here. The proposed approach is an important one toward self-healing software design, which is very much a research topic for developing reliable web services and web applications.

References

[1] Brent Miller. "The autonomic computing edge: Can you CHOP up autonomic computing?" *IBM Corporation*.

[2] Luciano Baresi and Sam Guinea. "An Introduction to Self-Healing Web Services." Dipartimento di Elettronica e Informazione-Politecnico di Milano, 2005.

[3] Kunal Verma and Amit P. Sheth. "Autonomic Web Processes." LSDIS Lab, University of Geogia Athens.

[4] Oreizy, P. and Gorlick, M.M. and Taylor, R.N. and Heimbigner, D. and Johnson, G. and Medvidovic, N. and Quilici, A. Rosenblum, D.S. and Wolf A.L. "An Architecture-Based Approach to Self-Adaptive Software." *IEEE Intelligent Systems*, Vol. 14, No. 3, p.54-62, 1999, USA.

[5] Goutam Kumar Saha. "Software-Implemented Self-healing System". Center for Development of Advanced Computing, Kolkata, India.

[6] Cook, B. and Babu, S. and Candea, G. and Duan, S. "Toward Self-Healing Multitier Services." *Technical Report* of the Duke University, 2005.

[7] Liu, J. "Self-X Property and Application in Web Servers". *Fachbreich Informatik*. TU Darmstadt.