Patterns for Fault Tolerant Cloud Software

ROBERT S. HANMER, Alcatel-Lucent

Patterns for Fault Tolerant Software was published in 2007 (Wiley). It contains sixty-three patterns related to designing software to tolerate faults with the goal of high availability. The patterns all presuppose a traditional software on hardware operating model that was prevalent in the computing industry up to the mid-2000s. With the widespread adoption of software virtualization and cloud computing some of the underlying assumptions present in the book have changed. In this PLoP paper several patterns from the book will be reexamined to discuss the modifications needed to make them useful or relevant to virtualized or cloud environments. Three new patterns for the collection are also in this paper.

Categories and Subject Descriptors: **C.4: [Performance of Systems]** *Fault tolerance, Reliability, availability, and serviceability*, **D.2.11 [Software Engineering]:** Software Architecture –*Patterns*, **K.6.3 [Management of Computing and Information Systems]:** Software Management – *Software Development*

General Terms: Cloud, Patterns, Availability, Virtual, Redundancy

Additional Key Words and Phrases: virtualized

INTRODUCTION

Patterns for Fault Tolerant Software (PFFTS) was published in 2007 containing sixty-three patterns to design software to tolerate faults with the goal of high availability. The patterns were written presupposing the traditional software on hardware operating model that was prevalent in the computing industry up to the mid-2000s. [PFFTS] Widespread adoption of software virtualization and cloud computing has changed some of the underlying assumptions which presents a new context for the application of these patterns.

The patterns are still useful, but how they will be used has changed slightly due to differences with the cloud. In some cases the underlying assumptions and needs have changed and the cloud-based solution is slightly different than the solution for traditional systems.

There are also new patterns that are specific to virtualized or cloud based implementations that are previously undocumented, such as "NO HARDWARE TO PRESERVE".

This PLoP paper reexamines two patterns from this book to discuss the modifications needed to make them useful or relevant or to reflect the changed nature of virtualized or cloud environments. These two architectural patterns are fundamental to designing a system that tolerates faults. The two patterns discussed here are UNITS OF MITIGATION (1) and REDUNDANCY (3). But both change because in a virtual environment THERE'S NO HARDWARE TO PRESERVE (64) when the application or system and new virtual machines can be created easily and quickly. (*Numbers in parenthesis from one to sixty-three refer to pattern numbers in Patterns for Fault Tolerant Software. Numbers from sixty-four upwards are present in this paper.*)

This paper also contains three new patterns for this collection. Unlike traditional systems that require management of hardware and software to achieve availability, in virtualized systems there's NO HARDWARE TO PRESERVE (64). FAST FAILING VMS (65) increase the availability because the workload can be transitioned quickly to REDUNDANT (3) VMs . Systems well designed for virtualization and the cloud

Author's address: R. Hanmer, 2000 Lucent Lane, Naperville, IL 60563 USA, Robert.hanmer@alcatel-lucent.com

adapt easily to VMs creation and destruction because of the LOOSE AFFILIATIONS (66) between their functions.

Not all the patterns in PFFTS make the transition as well as UNITS OF MITIGATION and REDUNDANCY, consider ERROR CONTAINMENT BARRIER (13). In the context of traditional systems an ERROR CONTAINMENT BARRIER was set up inside the application to contain errors; in virtualized systems setting up such a barrier inside a VM might be wasted effort. A better strategy instead destroys the virtual machine with the error with a whole new virtual machine created to replace it, after all the system has FAST FAILING VMS (65).

The context for this study of the existing patterns and the new patterns is that of an application designer. A basic assumption for this paper is that the cloud itself is highly available. The availability of the hardware and virtual machine manager is not in this context. In the cloud the availability of these elements will be managed by the cloud provider. While this paper is not focused on the virtual machine manager, the patterns in PFFTS are relevant to its design as well and will contribute to the availability assumed to exist in that layer.

Terminology

Certain terms are used in this paper that could lead to confusion due to different interpretations by the reader. This section will describe these terms and what is meant within this paper's context.

Application	A set of software assets that provides a desired functionality (i.e. a software-only asset, no hardware included). Applications that make extensive use of the networking are ideal candidates for cloud applications.
Architectural Pattern	A pattern that has wide impact on the design of the system, touching or influencing lower level design decisions in many different parts of the system. See [POSA] for the canonical examples of Architectural Patterns as well as a discussion of the different layers of patterns (architecture, design, idiom).
Cloud	At the lowest level Cloud means applications, systems or tasks running in virtual machines on hardware that the application owner may or may not have physical control over. In addition, Cloud in the context of this paper refers to the offering of services that have the NIST [NIST] definition: on-demand self-service, broad network access, measured service, resource pooling and rapid elasticity. The cloud can be provided by a 3rd party such as Amazon [AWS] or privately (a could run by the application supplier) or a combination. The economics of doing it yourself or paying someone else isn't the defining aspect for these patterns.
Distributed system	This term could mean either an application that is virtualized and hence a number of Virtual Machines communicate to provide the functionality or a traditional system with multiple parts that support the application functionality
Error	The incorrect operation of the application. Errors might be tolerated by the application or they may cause in a failure.
Failure	A failure occurs when the application no longer complies with the specification or requirements on the application.
Fault	A defect in the application. When triggered faults cause errors.
Physical system	Sometimes used in this paper as synonymous with a traditional system, defined below.
System	A combination of hardware and software assets that provides a desired functionality.
Traditional	Traditional system, for the purpose of these patterns, is one where the application or

systems	system is dedicated to a purpose. It might be shared with other applications or systems, but that complicates the management of fault tolerance. Traditional systems might be implemented as virtual machines on top of a Virtual Machine Manager, but then they don't benefit from the Cloud properties defined by NIST.
Virtual Machine (VM)	A Virtual Machine is the logical encapsulation of a computer in such a way that an application or task executing in the VM operates in the same way that it does when on dedicated hardware. VMs are the basic unit of processing and are the things that are rapidly elastic in the NIST definition of a cloud.
Virtual Machine Manager	A Virtual Machine Manager or hypervisor administers hardware resources providing a platform for multiple VMs to share the hardware. It manages scheduling the VMs and any hardware contention issues. There are different flavors of hypervisors (e.g. an operating system built in or underneath) which not significant for the purpose of this paper.
Virtualized	An application or system is virtualized if it executes in Virtual Machines. The Virtualized application or system may or may not be executing in a cloud.

Example Background

As a running example to illustrate the impact of these patterns on real system consider a submarine warfare game application. In particular the game that is used to illustrate use of the BLACKBOARD [POSA] pattern in the book Pattern Oriented Software Architecture for Dummies [POSAFD]. The game consists of a number of different components: Target Identification and Selection (TI), Target Prediction (TP), Maneuvering Control (MC), Trajectory Calculation (TC) and Fire Control (FC). The game controls and shoots a torpedo at a target from a submarine. These components direct the actions of the submarine after at least one target has been identified. Figure 1 shows how these components interact, and how the results from one build upon the results of another.

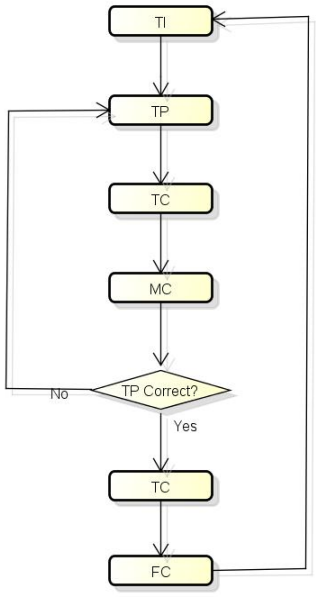

Figure 1: Game component interaction

Using a shared *Blackboard* these different components share information and possibly fire torpedoes at enemy ships. Each of these components places the latest information that it has computed onto the

blackboard. The components, which are all running asynchronously, can always access the latest information by checking the common, shared blackboard.

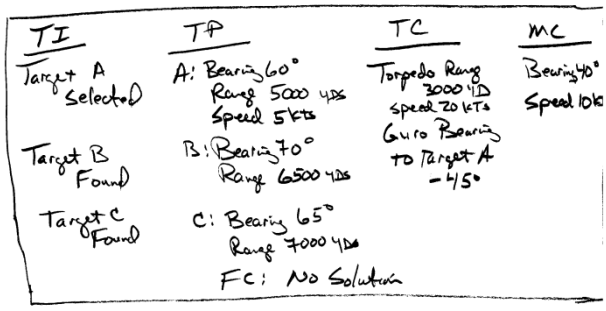

Figure 2: An intermediate state of the Blackboard

The example as presented in POSAFD is not necessarily fault tolerant, but assume for this paper that the application must be fault tolerant. This means that the game is always available for the game player. Even though, please assume, that the components are extremely complicated and hence prone to have errors the game player never notices failures because of the inherent REDUNDANCY built into the game. The game also has an infrastructure not seen in figure 1 to identify when components fail and can resume the game play using a duplicate of the component with minimal time interruption.

In the traditional system implementation, each of the components operates on a separate processor. Each one is isolated from the others and doesn't require direct interaction with the others except through the Blackboard. An example hardware configuration is a bladed system with five blades. In order to manage all these blades, i.e. start them initially and shut them down and to host the shared blackboard there is a sixth blade, which is called the "Management-blackboard" or "MB". Since the example game is highly available, each active blade has a mate that is operating in a standby manner. The coordination of which blade is active and which is standby is made by the MB blade.

The cloud environment strives to free these seven components from their confines of their blades and spread them through the cloud. The patterns presented in this paper use this as an example of converting a traditional system implementation into a cloud application.

Reviewing Selected Existing Patterns

Two of the most important concepts in the design of fault tolerant software are presented as the first architectural patterns in Patterns for Fault Tolerant Software. These are UNITS OF MITIGATION (1) and REDUNDANCY (3). Their importance is unchanged when the context switches from traditional to virtual systems. This section discusses these two patterns.

UNITS OF MITIGATION (1)

The pattern UNITS OF MITIGATION asks the question: "How can you keep the whole system from being unavailable when an error occurs?" The solution is to divide the system into **parts** that can both contain the errors and upon which error recovery can be done. The divisions should make sense for the system and the rest of the system should be designed around these units of mitigation.

In traditional systems these parts might be defined by the hardware design. The units might be the hardware processor on which the system is running, or might involve a particular sub-element such as Network Interface Controller chip. Frequently the units of mitigation have the same boundaries as FRUs (Field Replaceable Units) or other hardware artifacts.

The units of mitigation might alternatively be based on the software design, for example the part that manages the system's database, or the part that manages networking. Software boundaries exist at the process, task and application boundaries.

A determining factor in selecting the parts that will contain errors and be the units of error recovery is the convenience of setting up ERROR CONTAINMENT BARRIERS (13). In traditional systems, choosing hardware boundaries or fine-grained software boundaries (like the examples above) makes sense because these are the boundaries of things that can be controlled independently.

Virtualizing the system makes it entirely a software-based entity. The application doesn't have control of the hardware such as the networking hardware in the cloud, so the only remaining elements of mitigation are software based. The hardware boundaries are irrelevant because there's NO HARDWARE TO PRESERVE (64). The boundaries of software mitigation will be examined further in the discussion of REDUNDANCY (3) below.

An underlying force in this pattern is the speed at which recovery can be done. It takes much longer to restart a whole traditional system than it does to restart a process, which in turn takes longer to restart than a thread. Detection takes time but with FAST FAILING VMS (65) the failure is detected quickly and a redundant VM can be available that can take over operation as fast or faster than any recovery of hardware, process or threads. Using VMs as the UNIT OF MITIGATION requires conscious effort however because adequate REDUNDANCY (3) infrastructure to support that choice is needed.

In summary, the pattern applies to any system being designed that should be tolerant of faults. The nature of the parts into which the system is divided might change when transitioning from traditional systems to virtualized systems. They might transition from software and hardware elements like network controllers to entire virtual machines. However the need to divide the system into units that can be recovered quickly and hence the decisions about those unit boundaries must continue to be made early in the architectural process.

REDUNDANCY (3)

The third pattern in Patterns for Fault Tolerant Software is REDUNDANCY. It asks the question how can the time between error detection and resumption of normal operation be reduced. The resolution of this question is to add redundant elements into the system that can be activated quickly to take over the workload of the erroneous element.

In traditional systems the UNITS OF MITIGATION (1) would be hardware elements like processors, chips or discrete peripherals which are duplicated and operate in an active-standby or active-active manner. Sometimes the UNITS OF MITIGATION are software elements such as applications, processes or threads. These are monitored by a WATCHDOG (18) mechanism and restarted if necessary. Whether hardware or software, this duplication provides redundant elements to quickly assume the workload of failing parts of the system.

In many cases it will be easier to replicate a whole software element that previously required dedicated hardware by making it is own virtual machine. Virtualized systems, and the Cloud, makes creating new software elements easy and cheap, they're just virtual machines. See FAST FAILING VMS (65). Elasticity, the ability to easily grow and shrink the number of VMs is one of the five key characteristics of a cloud [NIST].

All the VMs can be available at the same time sharing the load. If one fails, who cares! An ELASTICITY MANAGER [FRLSA] manages the growth (and degrowth) of VMs with an eye towards providing the required level of redundancy.

Applications that are heavily reliant on specific hardware peripherals are less well suited to migration to the cloud. However, if those peripherals are smart enough to use the network then the applications can be sited anywhere, making a move to the cloud feasible.

Multiple VMs can be assigned to do the same work and the "best" result chosen. There are many ways to choose a result and consider it "best" ranging from picking the first response, the most popular or the average (see VOTING (21)) to select the most common response.

You have to be concerned that too many VMs will too fully use the resources provided by the infrastructure. This is a valid concern that can lead to adding measurements into the system to enable the VM-resident applications to know if they are really getting their fair share of processing resources. The cloud provider can solve this by using the patterns from PFFTS.

The REDUNDANCY pattern discusses three different types of redundancy: spatial, temporal and informational. Spatial redundancy refers to the system having multiple copies in different places that are redundant. This is a concern in virtualized systems which is addressed by anti-affinity rules and geographic redundancy.

Temporal redundancy refers to redundancy over time, such as repeating calculations and choosing the most popular one, or using RECOVERY BLOCKS (4). Virtual systems increase temporal redundancy by hosting multiple VMs that simultaneously repeat calculations bringing parallelism into play which can also increase overall system performance.

Another effective method of providing temporal redundancy and preventing software errors from occurring simultaneously in redundant elements is to provide diverse, heterogeneous implementations. The different versions can be different release levels of the same program, or they can be consciously different versions, commonly referred to as N-version Programming [CA78][KL86][KL90][Hanmer09]

Informational redundancy refers to having multiple copies of the information. Within a system or part of the system this is not enhanced by virtualization.

VMs can be created and given a state and then suspended. This is a very effective technique to speed up recovery, migration and elastic growth of the VM's application. This is described as STANDBY POOL in [FRLSA].

Redundancy supports the rapid detection and mitigation of errors and resumption of normal operation. Redundant elements allow resumption of normal operation to overlap with mitigation of errors in the erroneous element. State information must be shared between the redundant elements to allow one to pick up where another has become erroneous. This need doesn't change in virtualized systems.

In summary redundancy can be enhanced through virtualization because multiple VMs can be created easily and cheaply. In order for the overall system to achieve higher availability the redundancy must be designed in ways that take advantage of the virtualization and the ease with which VMs can be started and stopped.

New Patterns

Three new patterns that extend the non-virtualized system patterns in Patterns for Fault Tolerant Software are presented in this section.

64. NO HARDWARE TO PRESERVE

You are evolving an existing application from running on dedicated hardware in a traditional system to running in a virtualized environment. Either that or you're designing an application that will run in a virtualized environment. In either case the virtualization might be supplied by dedicated hardware or in a cloud environment.

Fault tolerant systems must react to any errors that occur and mitigate their effects. This means reacting to both hardware and software errors. Hardware wears out and breaks over time. Software also degrades with time, but in different ways. The software errors can occur in infrastructure or application.

❖ ❖ ❖

From previous systems you know that reacting to hardware errors will take significant portions of the system's software, you'd like to get away from this burden.

The new system you're designing will be virtual. The software will be self-contained in virtual machines. Working in the cloud separates the applications which are run in VMs from the underlying hardware and software infrastructure. The hardware infrastructure is supplied and managed by an infrastructure supplier. The software infrastructure required to manage the software is also the responsibility of the infrastructure supplier.

It can be faster to start a new VM than to try to recover from errors of the hardware. A copy of the VM can be created that saves a post-initialization state in such a way that when it is started it is ready to go already and doesn't need to be initialized.

In the submarine warfare game example the Maintenance-Blackboard (MB) component is responsible for the ongoing performance of the game; it acts as SOMEONE IN CHARGE (8). Each of the hardware processors or blades in the traditional system has a REDUNDANT (3) mate that can assume operation when the active component or its hardware has an error.

After a successful switch from active to mated standby then the MB must ensure that the failed element recovers from the error. This needs to happen quickly, because while it is happening there is a window in which there is no mated standby element. If another error occurs while the recovery of the component that failed earlier is underway the service offered by the component might be unavailable for a long time.

Therefore,

Concentrate your fault tolerant design on the mitigation of software errors in your application. Managing the hardware errors will be done outside your application. If you have an existing application, strip it of the hardware management functions since they won't be used any longer.

Managing the hardware availability is the responsibility of the infrastructure supplier. This is outside the application, or system, context. The failure of hardware elements will be more sudden and more silent than when the system software managed the hardware. The application controls that manage its own redundancy state may require modification because of these abrupt hardware failures.

The application in a virtual environment will not have responsibility to monitor the state of hardware. It also won't have responsibility to initiate failovers from failing hardware to redundant hardware.

The size of the overall system is reduced enhancing by availability. There is no hardware management software that might itself fail.

The recovery of the virtualized submarine warfare game is simplified. If the functions being performed by a VM fail then the primary responsibility can be assigned to a new VM -- this is similar to switching to a mated piece of hardware. But after that happens a new VM can be created to act as the standby in the future. The VM that failed initially can be destroyed (FAST FAILING VMS (65)), it does not need to "recover" in the same sense as in mated hardware. If a VM is destroyed all that is lost are some bits -- no hardware sits idle..

The MB serves as both SOMEONE IN CHARGE (8) and the ELASTICITY MANAGER [FRLSA] in the virtualized game.

65. FAST FAILING VMS
also known as VIRTUAL MACHINES ARE CHEAP

You're designing a system that will live inside a virtual machine.

Once an error has been detected it must be mitigated and the system recovered. Software recovery takes time during which the system is not processing, so this time needs to be reduced.

It's much easier to set up ERROR CONTAINMENT BARRIERS (13) around hardware errors than it is around software errors. Hardware can have buses deactivated and units powered off. Some software error containment techniques require that the element being contained is error-free to react properly, sometimes a contradiction.

Software recovery software can be a significant percentage of the software in a system. Starting a new VM and assigning it the functions of the erroneous element is the fastest way to resume normal operation. If REDUNDANCY (3) was achieved by having multiple VMs simultaneously performing the same function then this startup is even faster, this is a STANDBY POOL [FRLSA] of VMs.

Functions assigned to virtual machines don't tie up hardware resources in the same way that functions assigned to specific processors or blades do. You can't assume that the underlying processor has infinite resources and that you can create an infinite number of VMs.

One of the problems associated with abandoning a VM and starting another to take its place is the potential loss of state information. The failing VM might be the only repository of some state information. Using an external state store mitigates this, but adds to complexity and adds a additional redundancy related issues. Another way of reducing the scope of the problem of losing state is to share state between redundancy groups on a very frequent basis. Yet another option is to share state between all the VMs performing the functions of the STATEFUL COMPONENT [FRLSA].

Highly available, or fault tolerant, commercial systems built as combinations of VMs by necessity include multiple VMs. So while VMs are a common unit of cost from cloud providers, they are needed to support high availability.

In the submarine warfare game each VM stores some state, such as the current target location and intermediate computations. The Blackboard is used as the primary repository of centralized, shared state.

Therefore,

Don't recover the software, start a new VM instead and destroy the erroneous one.

VMs are virtual and don't tie up any physical resources if they disappear, so just destroy the ones with errors.

❖ ❖ ❖

This enhances availability by substituting the time to start a new VM performing a function for the time to isolate and mitigate an error.

Of course, report the errors to allow further analysis to be done to mitigate the fault -- correcting the software defect or the better handling the erroneous situation. VM failures or infrastructure actions that kill a VM occur suddenly. The application should keep logs, checkpoints and state information up to date so that a VM charged with continuing service can quickly begin and not too much state information is lost.

In the submarine warfare game example each of the functions (TI, TP, MC, TC, FC and MB) should be assigned to their own VM. The VMs should have only LOOSE AFFILIATIONS (66) rather than being tightly coupled. Since VMs don't tie up hardware in the same way that non-virtualized processes if a VM fails it can be destroyed without losing access to hardware resources.

State information stored in the submarine warfare game is primarily stored in the Blackboard. The intermediate results can be obtained easily by a replacement VM during its initialization phase.

66. LOOSE AFFILIATIONS

The application should be highly available. You are designing an application that will have many different virtual machines. Some are providing REDUNDANCY (3) and others are providing different functionality.

There are some key components, databases for example, whose function is closely associated with the function of other components. If you were building a monolithic traditional system you might build it as one big component. Now that you're building the application with VMs though you see the benefit of small, special purpose VMs. They simplify the redundancy because since FAST FAILING VMS (65) you can have many more of them.

You're now designing the overall structure and are wondering how the pairs of components should be designed. Should it be tightly coupled as you would have done in the monolith, or should they be loosely associated?

How closely should VMs work together?

VMs can be anywhere and aren't tied to specific hardware. This is one of the things that makes them so very flexible. FAST FAILING VMS (65) which so you can easily make as many as you like.

The whole design of your application assumes that the components you're designing in detail are closely coupled. For example, one holds the data that the other works on. You see the benefit of designing each to be the most focused -- the software in each VM will have higher cohesion and less inadvertent coupling with other components.

If you were designing a traditional, non-virtualized application you would design them to reside on the same processor and use intra-processor communication mechanisms. These might be intra-process communications, or if the components aren't in the same process space then some other optimized intra-processor (but not intra-process) communication method used instead. Since you're separating the functions out onto different VMs you can't use intra-process communication and have to rely on inter-processor communication (IPC), which is inherently slower.

The higher latency introduced by separating the functions into different VMs make it tempting to design the components to depend closely on each other. You can use a non-standard IPC mechanism or a private protocol on a standard IPC. But using non-standard IPC and making the components depend on each other makes design changes more difficult and also makes it more difficult to take one out of service and move to a redundant element.

LOOSE AFFILIATIONS makes it easier for VMs to come and go. If VMs are only loosely coupled then the handshake and configuration effort is simplified when the VMs are removed and new ones take their placed. The components must be prepared for VMs that they work with to disappear and be replaced by new VMs since that happens routinely in highly available systems.

Spreading the VMs that make up your application widely across a network decreases the risk that the failure of a single hardware element or zone of hardware (e.g. a computing center) will remove your whole application from service. This isn't quite geo-redundancy, but the loose affiliation can support and enhance geo-redundancy for disaster mitigation.

The LOOSE AFFILIATION should be able to add in new VMs for scaling or to replace missing VMs. VMs that go silently missing should be automatically replaced.

The mechanisms for VMs to join and to leave the community must be optimized for frequent changes. This optimization aids in making the application scale elastically.

Therefore,

Give the VMs that make up your system only a loose affiliation to make it easier for VMs to leave the system and for others to join.

Design your loosely affiliated VMs to exhibit these properties:

- They perform discrete functionalities.
- They work independently of other VMs; their actions are not closely interwoven with the actions of other VMs.
- They need little external assistance in their startup.
- They communicate with their peers using standard inter-processor communication techniques.
- They share information using common repositories that are not tied to the identity of creating or using VMs.

❖ ❖ ❖

This enhances availability by reducing the impact when VMs are destroyed and recreated. The time to incorporate new VMs into a running system is reduced and the impacts of VMs leaving the system is reduced because VMs coming or going is a routine occurrence, it isn't something that only happens during fault recovery.

In the example submarine warfare game the Target Identification and Selection (TI) and Target Prediction (TP) components must work closely together. TI examines sensor data that indicates that a possible target is in sight and will select between multiple targets the one to attack. TP predicts the future movement of the selected target. In a traditional system you created tight bonds between the two blades that host these components, tightly coupling them.

As you virtualize the system you realize the benefit of making this coupling looser. This will allow you to consider options like multiple TP components working in parallel (since FAST FAILING VMs (65) you'll use more VMs) -- this will allow you to increase the number of targets that you're keeping track of.

Loosely coupling the components allows a failing component to be removed from service and a completely new replacement integrated more easily. The tight protocols that couple them in a non-virtual environment are eliminated making the coupling more standard and easier to implement.

LOOSE COUPLING [FRLSA] sounds similar but is more concerned with decoupling communications in the ways that a BROKER [POSA] decouples communications.

Conclusion

In this paper two classic patterns from Patterns for Fault Tolerant Software were discussed and shown to be of continuing relevance in virtualized and cloud environments. These were UNITS OF MITIGATION (1) and REDUNDANCY (3).

Three new patterns, NO HARDWARE TO PRESERVE (64), FAST FAILING VMS (65) and LOOSE AFFILIATIONS (66) were described. These new patterns are especially relevant and useful when traditional systems are evolved to virtualized cloud applications.

Acknowledgements

Thank you to Peter Sommerlad for shepherding this paper and for being a constant supporter of PTFFS as I am of POSA.

Thank you to Ralph Johnson and the members of his Security Writers' Workshop group at PLoP 2014: Ali Alkazimi, Eduardo Fernandez, Russ Rubis, Jose Fran. Ruiz and Antonio Maña.

Thanks also to Christoph Fehling for his suggestions and pointers to other literature.

References

[AWS] Amazon Web Services -- *http://aws.amazon.com/*

[CA78] Chen, L., and A. Avizienis, 'N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation,' Digest of Papers FTCS- 8: Eighth Annual International Conference on Fault-Tolerant Computing, Toulouse, pp 3–9 (June 1978).

[FRLSA] Fehling, C., R., Retter, F. Leymann, W. Schupack, P. Arbitter. Cloud Computing Patterns. Springer, 2014.

[Hanmer09] Hanmer, R. S., N-Version Programming, Workshopped at PLoP 2009, Monticello, IL. http://www.hillside.net/plop/2009/papers/Process/N-Version%20Programming.pdf

[KL86] Knight, J. C. and N. G. Leveson, 'An Experimental Evaluation of the Assump- tion of Independence in Multi-version Programming,' IEEE Transactions on Software Engineering, vol. SE-12, no. 1 (January 1986), pp 96–109.

[KL90] Knight, J. C. and N. G. Leveson, 'A reply to the criticisms of the Knight & Leveson experiment,' SIGSOFT Softw. Eng. Notes 15, 1 (Jan. 1990), 24–35.

[NIST] Mell, Pl, Grance, T.. The NIST Definition of Cloud Computing. National Institute for Standards and Technology. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf (2011) Accessed July 14, 2014.

[PFFTS] Hanmer, R. S. Patterns for Fault Tolerant Software. Wiley, Chichester, UK, 2007.

[POSA] Buschmann, F., R., Meunier, H. Rohnert, P. Sommerlad and M. Stal. Pattern Oriented Software Architecture: A System of Patterns. Wiley, Chichester, UK, 1996.

[POSAFD] Hanmer, R. Pattern Oriented Software Architecture For Dummies. Wiley, Chichester, UK, 2013.