High Availability Testing – Stories from Enterprise and Consumer Services

Srinivas Samprathi
snsrinivas@rocketmail.com

Abstract

Quality of a cloud service is more than having a defect-free service which meets the requirements. If the product or service has downtime or is slow in recovering from faults, it will directly impact the product or service adoption and customer satisfaction.

High availability is often an implied need; but customers expect it to be there and its absence will impact the business. With global users and mobile devices using cloud services, testing for high availability, also known as fault tolerance, becomes much more important.

One useful measurement to gauge service availability is customer feedback and the customer perceived downtime. If the user feels the cloud service is erroneous or has frequent downtime, it does not matter what process or tools are used for high availability, the customer is always right.

This document explains high availability testing with fault model techniques that can be used to improve the availability of a cloud service (enterprise or consumer) and to plan for the right thing, and get the right things for the right user at the right time – every time.

Biography

Srinivas Samprathi is a Senior Test Manager at Microsoft, working in the Windows Application Experience team.

He is passionate about inventing engineering processes to help manage and deliver high quality software and services. His passion is engineering excellence and he enjoys coaching engineers, managers and leaders to build high potential teams.

For the last 15 years, Srinivas has managed and mentored engineers in the software industry. He frequently drives projects on software engineering, project management, talent development, and quality assurance testing.

At Microsoft he was actively involved in developing innovative engineering methodologies and has led the implementation of fault modeling and high availability testing for the Microsoft Office 365 Platform. He has been awarded the Leadership Award, Microsoft Gold Star Awards, Quality of Service award, etc. He has managed the quality for products like Windows, Microsoft Sharedview, Microsoft BPOS, Microsoft Office 365, Microsoft Windows 8 Metro apps, and Amazon Kindle Content Services.

1 Introduction

Conventionally, software testing has aimed at verifying functionality but the testing paradigm has changed for software services. Developing a full-featured and functioning software service is necessary; however service real time availability is equally important. Service downtime will increase customers' negative perception towards service quality and reliability. Services can be consumer services such as Kindle Services or corporate services such as Office365 messaging and sharepoint. A service engineering team's goal is to understand the common faults associated with user scenarios. A service is used by novices and experts. Therefore, the service needs to provide high availability for all the customers.

High availability testing is essential to ensure service reliability and increased fault tolerance. Fault tolerance is defined as an ability of a system to respond gracefully to an unexpected hardware or software failure of some of its components. High availability testing can be used to prevent outright failures of the online cloud service and to ensure continuous operation, therefore increasing fault tolerance and reliability. High availability testing does not necessarily focus on preventing faults, but ensures designing for tolerance, recoverability and reduced occurrence of high severity faults. This testing measures business impact from faults and helps in planning for reducing impact.

A theoretical understanding of how the service or system in test behaves under possible faults is a prerequisite to high availability testing. This paper starts with a discussion of stories from both the enterprise and consumer services. With such a reference model, the paper delves into providing a systematic method to model faults. The fault modeling section introduces key concepts of faults, categories of faults, and the component diagram along with the fault modeling itself. Once the system has a fault model, the next step is to test for fault tolerance. Finally, the paper summarizes the results seen with the Microsoft Office365 platform by following fault modeling and testing methodologies outlined in this paper. The results section highlights the savings achieved from reduced operational effort costs.

2 Discussion – Stories from Enterprise and Consumer Services

This section calls out some patterns of availability issues experienced in service teams. These patterns imply the need for a paradigm shift in designing and testing software services for high availability. Below mentioned stories call out the need for a systematic design and testing approach for achieving high availability. There is no silver bullet to the challenge of high availability; one can build a cloud service which meets the availability expectations of users by planning it for fault tolerance, managing it with the right set of monitoring, recoverability metrics and leveraging fault model techniques.

2.1 Fault Tolerance with High Volume and Deep Dependency Levels

Amazon Kindle Service endpoints receive millions of calls in a day which in turn can lead to several million outgoing calls (average ratio of 1:12) to underlying services (for a peak service call rate of over 1000 requests per second). Faults are guaranteed with these many variables in the system. Even if every dependency service has an uptime of say 99.9%, if a service is dependent on 36 other services, each with 99.9% uptime, then the service will have less than 99.9% uptime due to the chain of dependent services' downtime. When a single dependent service fails at high volume, it can rapidly lead to abnormally increased request queues to or from dependent services (Christensen Ben, Netflix TechBlog). Similar high availability issues can be addressed via systematic analysis of faults. The next section mentions, an instance of how much monitoring is good.

2.2 Is More Monitoring Good for High Availability?

A service received an alert indicating a failure in one instance. Within the next 12 minutes, there were 12 alerts which looked similar but affecting all instances of that service in all datacenters. Since such monitors run separately against each datacenter, 4 alerts were triggered for those failures, for a total of

16 failure alerts so far. The first step was to root cause the issue and understand the customer impact. It was sufficient to look at one of the failures, as all alerts were for the same failure. Redundant alerts wasted valuable investigation time that could have been spent on failure mitigation. The cause of this issue was erroneous entry of service configuration values in the production environment.

There are two advantages to reducing alert duplication. Firstly, it increases the signal to the noise ratio of alerts. Secondly, it prevents ignoring alerts with the assumption that alerts are duplicates. The only drawback of reducing alert duplication is missing genuine alerts during the reduction process. The next section highlights with an example, the pros and cons of production environment developer access.

2.3 How Much Developer Access to Production Environment is Fail-Safe?

On 2012 Christmas Eve, Amazon Web services experienced an outage at its Northern Virginia data center. Amazon spokesmen blamed the outage "on a developer who accidentally deleted some key data. Amazon said the disruption affected its Elastic Load Balancing Service, which distributes incoming data from applications to be handled by different computing hardware." Normally a developer has one-time access to run a process (Babcock, 2012).

The developer in question had a more persistent level of access, which lately Amazon revoked to make each case subject to administrative approval, controlled by a test team signoff and change management process. The advantage of this approach is the reduced business impact from a controlled review of changes made to production environments. The drawback of the change management process is the additional delay in implementing changes in production environments.

2.4 How to Recognize and Avoid Single Point of Failures?

A web service, *WS1,* for enterprise customers sends data from enterprise systems to cloud subscription. This web service *WS1* depends on a 3rd party Identity service such as Windows Live for authenticating enterprise users. There is one call in the Windows Live web service that the developer knows about. That call depends on the Windows Live auditing database which is prone to single point of failure. Another WS2 call also provides same functionality without the single point of failure. WS1 call was used as the developer knew about it. When the third party Windows Live Service auditing database was brought down for maintenance, the WS1 call resulted in high impact failures. Therefore, there is a need to understand the impact to availability due to each dependent service, even though the dependent service is a third party service. In this case, change the design to invoke the service call WS2 that is resilient.

Another example of single point of failure: The servers have 2 Top of Rack (*ToR*) switches. If one switch fails, then the other switch takes over. This is a good N+1 redundancy plan but is it sufficient?

If a rack was supported by only one switch (which had a hardware fault) and the redundant switch was in maintenance, then this can lead to single point of failure. Therefore, all the hosts serving an externally facing web service endpoint were arranged on that single rack. It is a high impact single point of failure to have all the hosts that serve an externally facing web service endpoint to be placed on a single rack.

Questions to ask: Are one of these switches regularly taken down for maintenance? If yes, how long on average? Do we have to know where all hosts serving the web service *WS1* implementation are arranged in datacenter? – Yes, absolutely to the extent of knowing there is no single point of failure.

2.5 How Important is Special Case Testing?

Testing the behavior of service on February 29 (leap year day) or on December 31st (the last day of the year) is one example of date-time special case test. On February 29, an Azure protection process inadvertently spread the Leap day bug to more servers and eventually, 1,000-unit clusters. The impact was downtime of 8-10 hours for Azure data center users in Dublin, Ireland, Chicago, and San Antonio. As Feb. 28 rolled over to Leap Day at midnight in Dublin (and 4 p.m. Pacific time February 28), a security certificate date-setting mechanism (in an agent assigned to a customer workload by Azure) tried to set an expiration date 365 days later on February 29, 2013, a date that does not exist. The lack of a valid

expiration date caused the security certificate to fail to be recognized by a server's Host Agent, and that in turn caused the virtual machine to stall in its initialization phase. This story indicates the need for system-wide fault safe special case testing. In this case, it was important to test how the entire system behaves on special date-time values (Babcock, 2013).

3 Fault Modeling

Fault Modeling is a service engineering model to capture the behavior of the system against faults. In order to understand fault modeling, it is necessary to agree on fault definition and the goals of fault modeling. Faults can be a single non-desired state of a component, entire system, service or group of components (Hardware or Software). Faults can be represented by transitions or states in addition to being represented as a single state. Such a representation is possible irrespective of the type of faults or nature of faults (permanent, transient or intermittent).

The goals of fault modeling are to:

- Identify fault domains (machine boundary, process boundary, datacenter boundary) of each component that is part of service implementation.
- Identify the single and double point of failures in the entire system.
- Derive a comprehensive list of possible faults, business priority and recovery from faults.
- Prepare a recoverability and redundancy model for deployment and maintenance activities.

3.1 Team member roles in Fault Modeling and Testing

Table 1 describes the typical roles and responsibilities played by team members in high availability testing and fault modeling:

Table 1: Roles and Responsibilities

Typical roles	Responsibilities
Component testers	Work with component developer to create a fault model for component(s). Define and document high availability test cases. Test component's fault tolerance and follow through to fix issues. Test faults with dependent components. Participate in the system level test pass. Automate fault tolerance test cases to enable ongoing faster execution.
Component feature teams	Contribute to component or role fault modeling. Review the fault tolerance component test plan(s).
Fault tolerance Subject Matter Experts (SMEs)	Defines and documents technical templates for fault modeling and test plans. Generally, members from each discipline (Testing, Developer, and Program Management) are preferred. Train rest of the team about designing and testing for fault tolerance. Handles Scheduling and Status updates. Works with other fault tolerance subject matter experts (SMEs) in the company, and outside the company to learn principles and best practices. Aggressively follows through downtime caused by faults in the service. Completes Root Cause Analysis (RCA) on an ongoing basis.
Operations/hosting team(s); All datacenter infrastructure teams	Collaborates with Component and System level fault tolerance testers to Understand what kind of faults (high customer SLA impact) can occur in the service or system. Participates in executing high availability test cases. Execute recovery steps (where recovery is manual); File bug reports if recovery steps are unclear or recovery fails. During test passes, works with the engineering team component tester (s), developer(s) to execute, and clarify recovery steps when recovery fails. During the execution of high availability test cases, works with component

	tester (s) and developer(s) to recover from the component or system level faults in a time-bound manner.
Monitoring SMEs	Investigate and propose solutions needed for fixing holes where faults were not monitored and alerted. Ensure all non-automated recovery faults have documented Trouble Shooting Guides (TSGs) (These guides will be verified by fault tolerance SMEs; executed and signed off by Operations team SMEs during pre-production or live production testing).
Higher management	Approves and sponsors fault tolerance activities. Provides guidance to improve and implement design changes.

3.2 How is The Faults Modeled for Each Component (Software Component, Infrastructure or Hardware) of a Service?

For each fault category, identify and model:

- Fault definition: Define the fault (described in the above section).
- Customer Attributes:
  - Severity of customer impact can be high or low financial impact.
  - Scope of impact can be all users or one user.
- Historical data or measured attributes:
  - Probability of fault occurrence: Is the probability High or Low?
  - Time to Recover.
  - Time to Repair.
- Monitoring:
  - Does monitoring exist for a specific fault?
  - If monitor exists, how soon is impact notified by the monitor?
- Recovery:
  - Document Manual or Automated recovery expected.
  - Document Standard Operating Procedures (SOP) or Trouble Shooting Guides (TSG) for manual recovery.

To achieve a fault model for a service component, begin with the service component diagram as shown in Figure 1. This component diagram depicts a single active service instance. This service has high write load and is designed to be in the same datacenter as the active storage units. This service interacts with four layers which are the storage service, business logic workflows, simple mail transfer service (SMTP), and remote power shell. Windows live provisioning for user data happens through business logic workflows. The service authenticates clients through windows live tokens. The reads, writes, and searches are done directly through storage service API. The service sends emails for a batch of data using the SMTP server located in the datacenter. All datacenters have the exact same configuration. GSLB handles Global Server Load Balancing.

A component diagram is a diagrammatic representation of the component and it's dependencies as it will operate inside the service offering's datacenters. The important thing to capture with component diagrams is dependencies, especially out of process dependencies, so that when something is not working as expected, the escalation on-call engineer could look at a component diagram and say, "well, the service needs to talk to Windows Live in order to work." Enumerate dependencies to one level. Include datacenter boundaries and load balancing components in the component diagram. The diagrams identify the easy and most common interfaces that go faulty and the dependency faults. This makes a good first set of faults – what happens when dependencies fail, or perform poorly?

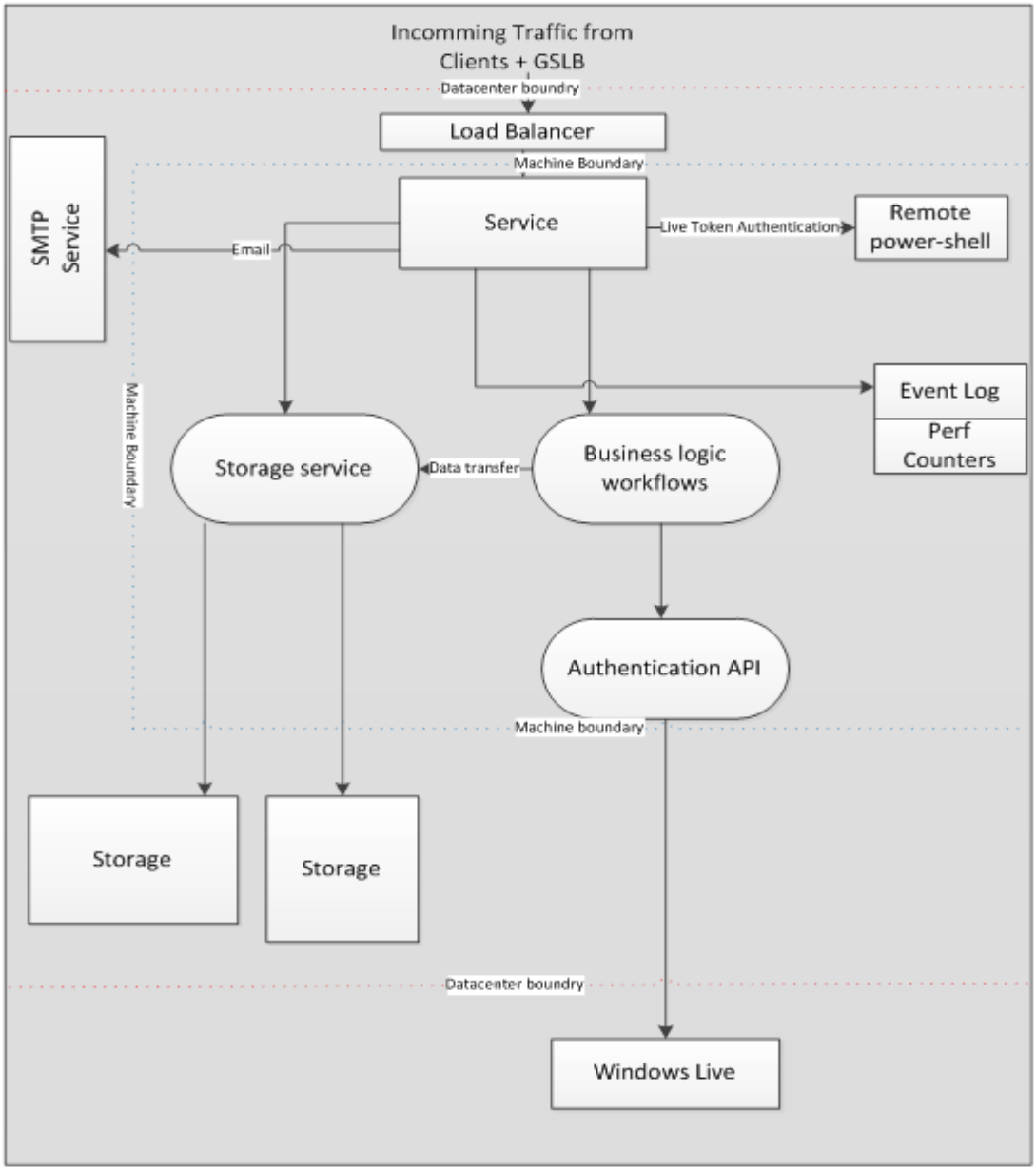

Figure 1: Example component diagram

3.3 Fault Categories with Examples

3.3.1 Within Machine Boundary

This category includes faults that can occur within a service component's physical or virtual machine boundary. This is not an exhaustive list but an illustration of fault types in this category.

Web service application pool crashes are common for service endpoints hosted in IIS (Internet Information Services) application pools. It is important to detect application-pool crashes configure application-pools to be self-recoverable.

Resource starvation such as CPU hog, memory starvation, and I/O hog are a common set of faults that can occur within a machine boundary. Key aspects of these faults are to keep them machine boundary specific and not cascade to all instances of a service. Also, to design for monitoring and recovery of service instances from resource starvation faults.

Certificate faults are another set of faults that can take reliable service operations by surprise. These faults can vary from certificate servers being single failure points or forgetting to renew key authentication certificates, e.g. Virtual IP certificates. It is crucial to have an infrastructure in place that monitors certificate expiration and acts upon them.

Excessive non-circular service logging (log files not cleared automatically) is a common fault seen in bigger service systems spanning hundreds of services. Ensure service logs are circular and regularly archived to a query able fault redundant database.

A single physical or virtual machine representing a single failure point for the service is an obvious fault that can easily cascade into dependent service failures as well. Such faults can be avoided by building in redundancy of virtual or physical machines.

3.3.2 Across Machine Boundary

This category includes faults that occur across the machine boundary of the same component or different components. This is by no means an exhaustive list but an illustration of fault types in this category. Dependent service high latency or unavailability is a high frequency fault in today's service offerings such as Kindle Services or Enterprise Office365. Only one instance of a service behind a Load balancer or Virtual endpoint is expected from services. But there are fault situations where this can happen. One such situation is when there are 'n' instances of a service representing a virtual endpoint. However, the certificates for all but one host of the service expired on the same day causing a single point of failure. Such issues can be mitigated by detecting dependent service failures and mitigating.

3.3.3 Infrastructure and Site-Level Faults

This category includes faults that happen in a datacenter network or in network infrastructure across datacenters. This is not an exhaustive list but an illustration of fault types in this category. High network latency in datacenter is a high impact and a low frequency fault. With Global Server Load Balancing (GSLB), internet traffic can be distributed among different data-centers located at different locations of the planet. This technology is highly efficient in avoiding the impact from local datacenter downtimes. GSLB failures or mis-configurations can cause downtime for end users and can be a high frequency fault if configurations are modified often. Cross-datacenter replication unavailability or high replication latency is often a high business impact fault and needs to be properly monitored.

A network infrastructure single point of failure such as a single primary Domain Name Service (primary DNS) with no updated warm secondary Domain Name Service (secondary DNS) can be an unrecoverable high business impact fault. These failures can be mitigated by monitoring updates to secondary instances, such that the secondary instances can replace the primary instances at any time.

3.3.4 Operational Faults

This category includes faults that happen in operational and hardware infrastructure in datacenters. This is not an exhaustive list but an illustration of fault types in this category. (Barroso Luiz André and Hölzle Urs). Bad rack allocation such as physically allocating all service instance machines on the same rack will make that service behavior prone to rack failures. Network rack switch failures with no redundant switch can cause entire racks to be down for days while repair and maintenance on switches is scheduled. Load balancer failures are expected to be low frequency but load balancer traffic management scripts (iRules) can change often and cause unhandled faults. Ensure testing of hardware configuration changes in a staging environment prior to production deployments.
Lack of datacenter redundancy indicates a cross datacenter operational fault and needs to be fixed to avoid physical datacenter or geographical disaster failures. Datacenter expansions or expansions of new services within a datacenter can also be faulty. These faults can be reduced by adding monitoring and

testing performance pressure from expansion on the rest of the existing system. Faults in monitoring infrastructure itself can be critical. Service monitoring must catch downtime issues before external customers report issues. It is critical to ensure monitoring infrastructure itself is stable.

3.4 Real-Time Monitoring of Customer Impact

In addition to fault modeling, it is necessary to model customer scenarios and automate execution of these scenarios as real-time monitors in production environments. This helps to identify immediately what customer scenarios are affected by fault(s) occurring in the system rather than waiting for customers to report issues. These automated monitoring modules can be referred to as Customer Scenario Synthetic Transaction Monitoring.

4 How to Test for High Availability?

Testing for high availability starts with the system fault model as a reference. From the fault model, it is essential to design and execute high availability test cases.

Steps in designing high availability test cases include, but are not restricted to:

- Converting faults derived from fault modeling to actionable test cases.
- Categorizing test cases into:
    - Component level fault test cases which can be within machine boundary or across machine boundary.
    - System level impacting fault test cases.
    - Third party service or cross datacenter fault test cases.
    - Infrastructure and operational level fault test cases.
- Prioritize fault test cases according to customer perceived impact.

Some of the execution steps for a typical high availability test case are:

- Inject fault. Fault injection is the test technique to introduce faults to execute and test error handling and other uncommon code paths in a component.
- Verify the monitoring system raised expected alert(s) to indicate the fault has occurred.
- Verify that no noisy (unexpected or too many) alerts are raised.
- Verify ‘Customer Scenario Synthetic Transaction Monitoring’ behavior shows expected failure. See section 3.4.
- Verify expected component or system behavior during or after fault has been induced
    - Test cases to execute during or after the fault injection.
    - Component or Role specific impact.
    - System level impact.
- Note time to recover (manual or automated recovery).
- If manual recovery: Execute documented manual recovery (Standard Recovery Procedure) steps.
- If Standard Recovery Procedure does not exist yet, then track a task item to create one.
- Execute test cases after recovery (automated or manual) to verify normal expected functionality
- Verify ‘Customer Scenario Synthetic Transaction Monitoring’ behavior shows expected success.

Documenting and executing a high availability test case in the above manner also facilitates modular test automation of each test case and easy manual execution.

4.1 Testing Environments and Testing Frequency

Table 2 explains which testing environment should be used for testing each fault category. This table also suggests the typical testing frequency for each fault category. For example, infrastructure fault cases must be tested in production as pre-production can be different or low scaled compared to actual production environment. However, the simplest case of within machine boundary faults can be simulated and verified in single test machines.

Table 2: Fault testing environments

Fault type	Testing environment	Testing frequency
Within machine boundary faults	Test in a single test machine environment (commonly referred to as one-box testing).	Automate tests and execute high impact cases when a service component is modified.
Across machine boundary fault cases	Test in pre-production and production environments. Focus on testing high impact scenarios.	Schedule monthly or quarterly, based on the business need and change frequency.
Infrastructure fault test cases	Test in Production environments. Focus on testing high impact scenarios.	Schedule monthly or quarterly, based on the business need and change frequency.
Operational fault test cases	Test in production environments. Focus testing the high impact scenarios.	Combine testing with planned operational maintenance or deployment or expansions.
Third party service or cross datacenter fault test cases	Test in pre-production and production environments. Focus testing the high impact scenarios.	Schedule after major updates are deployed to individual data centers. Schedule during and after expansion of data centers.

5 Why Test for High Availability in Production?

Because pre-production environments, which are scaled down versions of production environments, often lag behind on infrastructure upgrades and maintenance compared to production. This causes pre-production environments to be different from production. Since faults (software, firmware and hardware) can be environment specific, testing in pre-production environments alone is not sufficient. It is also important to test regularly in production as production environments have regular deployments, expansions for scaling, new datacenters added, and more database hardware added.

Some examples of issues found by testing in production environments are:

- Secondary Active Directory Integrated DNS (INS) server not in sync with Primary Active Directory Integrated DNS (INS) server; therefore Primary INS is a single point of failure.
- Some key role hosts in production may not have monitoring infrastructure client installed therefore these hosts or their service failures were not being monitored.
- Replication servers (Active Directory bridgehead servers) are misconfigured in production and there was no monitoring in production to verify replication status.

6 Results and Conclusions

Adopting fault modeling and testing method in Microsoft Office365 platform resulted in an estimated savings of 600 work days per year. This savings was achieved in alert escalation, the manual operational cost to resolve issues, and cost of communicating impact to enterprise customers. The savings comparison is attained by comparing the first version of Office365 called ‘Microsoft Online Services’ (MOS) (which did not use a fault modeling and testing approach) with the Office365 platform (O365).

Estimated savings was determined using all of the below data:

- The number of fault recovery and fault identification issues found pre-release.
- Historical data from versions of Office365:
    a) frequency of severity 1 faults as illustrated in Figure 2, Chart 1
    b) time to resolve customer issues as illustrated in Figure 2, Chart3.
- The cost of fault modeling, high availability testing, and redesign for handling recovery.
- Team’s increased understanding of the single and double point of failures and designing for resiliency as illustrated by Figure 2, Chart 2.

- Realization that software or hardware faults are not necessarily prevented, but proper design, fault modeling and testing can ensure resiliency.
- Paradigm shift and the progress with:
    - Testing availability of cloud services.
    - Integrating high availability testing with monitoring testing.
    - Including different teams in availability and monitoring testing.
    - Testing regularly for availability in different environments, including production.

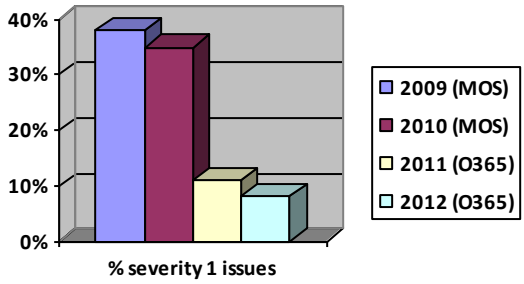

Chart 1: % severity 1 issues

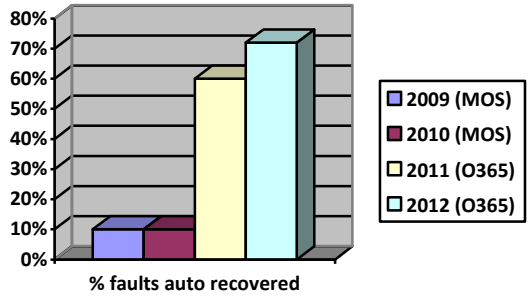

Chart 2: % faults auto recovered

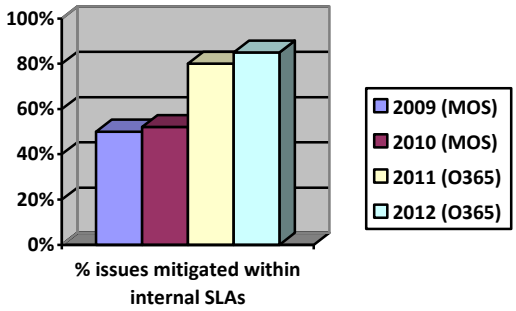

Chart 3: % issues mitigated within SLAs

Figure 2. Historical data from versions of Office365

From above mentioned results and charts, it can be derived that adopting fault model and testing methodology contributed to reducing high severity alerts by at least 65%, which is a statistically significant result. It can also be seen from the charts that the focus on auto recovery and testing manual recovery steps helped reduce mitigation times in 50% of issues compared to when fault model and testing methodology was not utilized.

References

- Christensen Ben, Netflix TechBlog, available http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html
- Babcock Charles, “Amazon's Dec. 24th Outage: A Closer Look”, Information Week, January 2013, available http://www.informationweek.com/cloud-computing/infrastructure/amazons-dec-24th-outage-a-closer-look/240145533
- Babcock Charles, “Azure Outage Caused By False Failure Alarms: Microsoft”, Information Week, March 2012, available http://www.informationweek.com/cloud-computing/infrastructure/azure-outage-caused-by-false-failure-ala/232602382
- Barroso, Luiz André and Hölzle, Urs, “The Datacenter as a Computer”. Examples of operational faults, frequency are available from: http://blog.s135.com/book/google/dc.pdf Chapter 4.