



TECHNISCHE UNIVERSITÄT
CHEMNITZ

Reaching High Availability in Connected Car Backend Applications

Master Thesis

Dept. of Computer Science
Chair of Computer Engineering

in Cooperation with

NTT DATA
Global IT Innovator

Submitted by: Arpit Yadav
Matriculation No: 366669
Date: 14.05.2017
Supervisors: Prof. Dr. W. Hardt
Andreas Maier, M.Sc.
Dipl.-Inf. René Bergelt

Acknowledgement

I would like to extend my sincere gratitude to all the individuals who helped me throughout the thesis work.

I sincerely thank Prof. Dr. Wolfram Hardt for his kind supervision and support in pursuing the master thesis and providing the opportunity to write the thesis at NTT Data Deutschland GmbH.

Furthermore, I would like to thank my supervisor Mr. Andreas Maier for his continuous motivation and guidance. It is my great pleasure to acknowledge the entire team at NTT Data for giving me the opportunity to perform my work.

In the end, I would like to thank Mr. René Bergelt and the Computer Science Department at Technische Universität Chemnitz for helping me at every stage of the master thesis.

A handwritten signature in blue ink that reads "Arpit Yadav". The signature is written in a cursive style and is placed on a light blue rectangular background.

Place: Stuttgart
Date: 30.04.2017

Arpit Yadav

Abstract

The connected car segment has high demands on the exchange of data between the car on the road, and a variety of services in the backend. By the end of 2020, connected services will be mainstream automotive offerings, according to Telefónica - Connected Car Industry Report 2014 the overall number of vehicles with built-in internet connectivity will increase from 10% of the overall market today to 90% by the end of the decade [1]. Connected car solutions will soon become one of the major business drivers for the industry; they already have a significant impact on existing solutions development and aftersales market.

It has been more than three decades since the introduction of the first software component in cars, and since then a vast amount of different services has been introduced, creating an ecosystem of complex applications, architectures, and platforms. The complexity of the connected car ecosystem results into a range of new challenges. The backend applications must be scalable and flexible enough to accommodate loads created by the random user and device behavior. To deliver superior uptime, back-end systems must be highly integrated and automated to guarantee lowest possible failure rate, high availability, and fastest time-to-market.

Connected car services increasingly rely on cloud-based service delivery models for improving user experiences and enhancing features for millions of vehicles and their users on a daily basis. Nowadays, the software applications become more complex, and the number of components that are involved and interact with each other is extremely large. In such systems, if a fault occurs, it can easily propagate and can affect other components resulting in a complex problem which is difficult to detect and debug, therefore a robust and resilient architecture is needed which ensures the continuous availability of system in the wake of component failures, making the overall system highly available.

The goal of the thesis is to gain insight into the development of highly available applications and to explore the area of fault tolerance. This thesis outlines different design patterns and describes the capabilities of fault tolerance libraries for Java platform, and design the most appropriate solution for developing a highly available application and evaluate the behavior with stress and load testing using Chaos Monkey methodologies.

Keywords: Connected Car, High Availability, Cloud Computing, Chaos Monkey

Table of Content

Abstract.....	ii
Table of Content.....	iii
List of Figures.....	vi
List of Tables.....	vii
List of Abbreviations.....	viii
1 Introduction.....	1
1.1 Motivation and Goal.....	1
1.2 Problem Statement.....	2
1.3 Structure of the Report.....	4
2 State of the Art.....	5
2.1 Related Work.....	5
2.2 Monolithic to Microservices.....	6
2.3 Cloud-based Design Patterns.....	6
3 Background.....	8
3.1 Connected Car.....	8
3.2 High Availability.....	10
3.2.1 Definition of Availability.....	10
3.2.2 Availability Parameters.....	11
3.2.3 High Availability Mechanism.....	11
3.3 Distributed Systems.....	13
3.4 Failure Characteristics.....	15
3.4.1 Hardware Failures.....	15
3.4.2 Software Failures.....	16
3.4.3 Causes of a Downtime.....	16
3.5 Cloud Computing.....	17
3.5.1 Cloud Computing Reference Architecture.....	19
3.5.2 Cloud Service Model.....	20

3.5.3	Connected Car and Need for Cloud Computing.....	23
4	Analysis	25
4.1	Service Architecture	25
4.1.1	Monolithic Approach for Software Development	25
4.1.2	Microservices – Tackling the Complexity.....	28
4.1.3	Docker – The Container Standard	31
4.2	Design Patterns for Cloud-based Application	33
4.2.1	Availability Design Patterns	33
4.2.2	Resiliency Design Patterns	37
4.3	Fault Tolerance Libraries for Resilient Applications.....	41
4.3.1	Hystrix	41
4.3.2	JRugged	48
4.3.3	JavaSlang.....	49
4.4	Resilient Platforms for Microservices	50
4.4.1	Spring Boot.....	50
4.4.2	Wildfly Swarm	51
4.4.3	Vert.x	51
4.5	Testing	51
4.5.1	Testing Distributed Systems.....	52
4.5.2	Chaos Monkey	52
4.5.3	Simian Army by Netflix	54
5	Concept and Design.....	57
5.1	Requirements	58
5.2	Design of solution	59
5.3	Architecture	61
6	Implementation.....	63
6.1	Normal Execution	63
6.2	Timeout Execution.....	65
6.3	Hystrix Execution	67
6.4	Monitoring with Hystrix Dashboard.....	71

6.5	Testing	74
7	Conclusion	79
7.1	Observations	79
7.2	Summary	82
7.3	Outlook	82
	Bibliography	x

List of Figures

Figure 3-1: Evolution of Connected Car	8
Figure 3-2: Complex distributed system	14
Figure 3-3: Bathtub curve	15
Figure 3-4: Cloud Deployment Models.....	19
Figure 3-5: Cloud Computing Architecture	19
Figure 3-6: Cloud Service Models.....	20
Figure 3-7: Cloud Service: IaaS, PaaS & SaaS	22
Figure 3-8: Cloud and Connected Car.....	23
Figure 4-1: Monolithic Architecture	26
Figure 4-2: Monolithic vs. Microservice Architecture	29
Figure 4-3: Docker for Developers.....	31
Figure 4-4: Virtualization vs. Containers	32
Figure 4-5: Health Endpoint Monitoring Pattern	34
Figure 4-6: Queue-Based Load Leveling Pattern	35
Figure 4-7: Design Pattern Working	36
Figure 4-8: Throttling Pattern	37
Figure 4-9: Circuit Breaker Working	40
Figure 4-10: Working of Hystrix- Normal Workflow.....	42
Figure 4-11: Working of Hystrix- Failed Dependency	43
Figure 4-12: Working of Hystrix.....	43
Figure 4-13: Hystrix Circuit Breaker Working	45
Figure 4-14: Hystrix Dashboard.....	48
Figure 5-1: Concept - Normal Execution	57
Figure 5-2: Concept- (Problem Scenario)	58
Figure 5-3: Design- Use of Hystrix	60
Figure 5-4: Design- Problem solved with Hystrix	60
Figure 5-5: Architecture design at the application level.....	61
Figure 6-1: Eureka server when microservices are not running.....	72
Figure 6-2: Registration of Services on runtime	73
Figure 6-3: Screenshot from Hystrix Dashboard.....	73
Figure 6-4: Screenshot of Circuit Breaker in action.....	74
Figure 6-5: Test Execution Plan 1	75
Figure 6-6: Test Execution Plan 2	76
Figure 7-1: Availability vs. Complexity, Cost	80
Figure 7-2: Result- HA application	81

List of Tables

Table 1-1: Major IT Players Service Outages	2
Table 3-1: Example of Connected Car Services	9
Table 3-2: Availability measure in terms of number of nines	13
Table 3-3: Unplanned Downtime	16
Table 3-4: Planned Downtime	17
Table 4-1: Deployment Time Comparison	32
Table 4-2: Comparison of FT libraries	50
Table 4-3: Simian Army	55
Table 6-1: Implementation - Normal Execution.....	64
Table 6-2: Screenshot- Normal Execution	65
Table 6-3: Implementation - Use of Timeouts	66
Table 6-4: Screenshot- Timeout Implementation	67
Table 6-5: Implementation - Use of Hystrix	68
Table 6-6: Screenshot- Hystrix Implementation	69
Table 6-7: Screenshot to show availability (Timeout)	70
Table 6-8: Screenshot to show availability (Hystrix).....	71
Table 6-9: Components used for the implementation	71
Table 6-10: Test Results	77

List of Abbreviations

API	Application Programming Interface	RPC	Remote Procedure Call
AWS	Amazon Web Service	SaaS	Service as a Service
C2C	Car to Car	SLA	Service Level Agreement
C2I	Car to Infrastructure	UI	User Interface
CC	Cloud Computing	URL	Uniform Resource Locator
CRM	Customer Relationship Management	VM	Virtual Machine
DA	Distributed Application		
DNS	Domain Namespace		
DS	Distributed System		
FT	Fault Tolerance		
GCE	Google Compute Engine		
HA	High Availability		
HTML	Hypertext Markup Language		
HTTP	Hypertext Transfer Protocol		
IDE	Integrated Development Environment		
IaaS	Infrastructure as a Service		
ICMP	Internet Control Message Protocol		
IEEE	Institute of Electrical and Electronics Engineers		
IT	Information Technology		
ITIL	Information Technology Infrastructure Library		
JSON	JavaScript Object Notation		
MTBF	Mean Time between Failure		
MTTR	Mean Time to Repair		
NIST	National Institute of Standards and Technology		
NTT	Nippon Telegraph and Telephone		
OEM	Original Equipment Manufacturer		
OS	Operating System		
OSS	Open Source Software		
PaaS	Platform as a Service		
REST	Representational State Transfer		

1 Introduction

Service availability has always been an essential software requirement, mainly for safety critical systems for which any service interruption may have fatal outcomes. The advancement in the information technology is changing the way services are being designed and delivered. In some sectors, services are considered highly available if they are accessible 99.999% of the time. The five nine's requirement is becoming a reality which accounts for the service downtime of 5.25 minutes per year. End users and service consumers are increasingly demanding and expect the services to be available every time and everywhere. The ongoing transformation of cloud computing has led to the rising trend of migration to it. Nowadays, services are increasingly being hosted in the cloud, to avail the benefits of this technology. Although cloud computing offers high availability measures, there is an emerging need for a solution at the application level that could fulfill the demand for high availability and resiliency. The complexity of distributed systems combined with cloud delivery model opens new challenges in the software industry. The need for high availability does not only apply to automotive sector – but apparently connected cars accessing the backend services are one of the main topics in this field, due to its emerging need for hosting critical and real-time streaming applications.

In recent years, companies like Netflix and Amazon found new ways to face the challenge of availability by adopting new architectures and mechanisms for the application development. The shift from monolithic applications to microservices introduced a new way of application development and allowed the integration of fault tolerance mechanisms. The thesis outlines design patterns for applications development and proposes an appropriate solution for the development of highly available application using Hystrix fault tolerant library and adopting Chaos Monkey testing mechanisms.

1.1 Motivation and Goal

Businesses are adopting the cloud delivery model, and achieving a high level of availability has become a challenge for the providers. Cars of the future will access the services built on the distributed system model. The complex critical applications deployed in the cloud increase the responsibility of providing better availability.

The harsh reality in distributed system is that failures will happen all the time. Developers design the distributed system with the expectation of failure, and therefore it is necessary to make them resilient. So there is a need and scope of enhancing the availability of the services at the application level by making them resilient.

To strongly motivate the thesis work, some major service outages that happened in the last year 2016 were studied and discussed. The lesson learned from these cases, justify the need for a solution for high availability at the application level.

Service Outages		
(Approximate downtime in hours)		
Verizon (Jan.14)	Microsoft Office (Jan.18:5 days)	Salesforce (March3:10 hours)
Google Service (April11: 18 minutes)	Apple icloud (June2)	Amazon Web Services (June4: 10 hours)
Microsoft Azure (September15: 2 hours)	DYN (October21)	Symantec (April11:24 hours)

Table 1-1: Major IT Players Service Outages [2]

The lesson learned from most of the outages is that there is a need for finding a feasible solution for developing resilient applications and stop cascading failures by isolating them and to find the mechanism for testing the applications for failures.

The primary goal is to find a solution to achieve high availability by analyzing different design patterns, fault tolerant libraries, and technologies. Moreover, also to introduce the Hystrix library for fault tolerance and Chaos Monkey methodologies for testing the application. NTT Data, working on the project for Daimler Connected Car applications wants to introduce Hystrix Library features in the upcoming projects and therefore Hystrix should be explored and tested. The Hystrix library and Chaos Monkey are tested in a cloud environment where the combined benefits can be achieved which will result into High Availability. A concept is needed by the company on which they can work in the future to minimize downtime by introducing new mechanism at the application level and the concept can be used by the development teams at NTT for their upcoming projects.

1.2 Problem Statement

The challenge of achieving high availability is not new. Earlier the issue was handled by redundancy and other measures at the hardware level. Soon it was realized that efforts taken at the hardware level alone are not enough to gain high availability. Deploying applications in the cloud introduced uncertainties because now the applications are not under the direct control of the owner. With the introduction of complex software solutions, there is a need to configure new approaches at application level which can lead to high availability.

Keeping in mind the overall goal of reaching high availability, the thesis aims to analyze the service architecture, design patterns, and fault tolerant libraries best-suited and design a feasible solution by gathering most relevant requirements. Testing distributed systems is hard and therefore a concept should be proposed which tests the system for failures and can be used efficiently.

Cloud-based connectivity enables the connected car to offer new services to the end users. The only way to manage and transfer a vast amount of data is by using distributed services

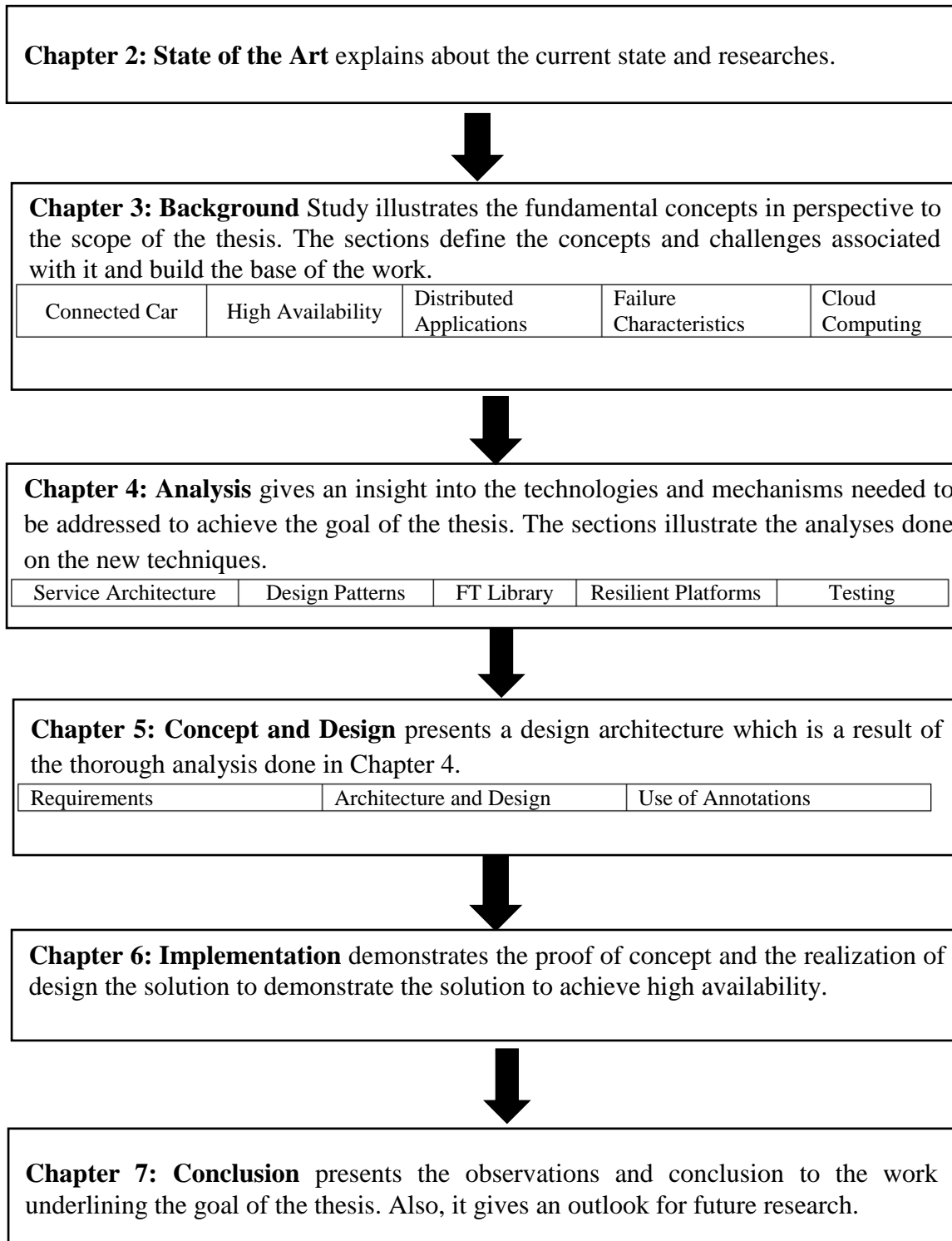
running in the cloud. Future cars will be connected to electronic devices ranging from smartphones to smart homes, the challenge that remains to be met is ensuring availability of the services. For a normal smartphone application, a little downtime can cause a minor inconvenience, but for connected and autonomous cars, the effect of a downtime of a critical application could be potentially dangerous.

Therefore there is a need arises to find a solution which can be adopted at the application level which focuses on high availability. With increasing numbers of vehicle services depending on the constant availability of IT backend systems, it is necessary to design resilient services with low latency.

Moreover, the traditional approach of testing like unit testing, load testing, and functional testing are not capable of catching all types of errors in the distributed system and even in non-distributed systems. Therefore, there is a need for implementing a non-deterministic approach, which uses random mechanism for detecting the unexpected faults.

1.3 Structure of the Report

The structure of this thesis is organized in the following chapters which give an outline of the work, indicating topics and individual sections.



2 State of the Art

The chapter describes the current state of the research done towards the goal of achieving high availability. The first section states the related work done so far which is relevant to the thesis topic. The second section will describe the mechanisms which are already present in the IT market availing HA. The conclusion of this chapter will explain the need of the thesis work and the current state of the solution which is used by the NTT Data to achieve high availability.

2.1 Related Work

In this section, studies of related work are presented in which different concepts were discussed for reaching high availability.

The author in [3] describes the need for reaching high availability and classifies the solution into two categories: middleware and virtualization-based approaches. The author investigates a set of availability solutions based on existing cloud computing, including Open Stack, VM, and OpenSAF. The work also gives a foundation to address application failure and host failures.

Ali Kanso and Wubin Li [4] in their work throw light on the problem of ensuring high availability for applications hosted in virtualized environments. The author provides comparison result of virtualized platform and investigates the limitations of features like failure detection and fault tolerance and recommends a use of container technologies for HA.

The authors [5] in their ongoing work on High Availability, presented a novel approach for cloud applications at 2015 IEEE International Conference on Cloud Engineering. The concept leverages Linux containers to achieve HA. They adopted the checkpoint/resume strategy and found that the overhead imposed on the application side is inevitable.

Lots of papers illustrate the mechanisms for achieving HA. However, there is not so much research done focussing on the solutions to achieve HA using fault-tolerance mechanisms at the application level. The FT library named Hystrix is new in the market and not many companies introduced it into their projects. The testing mechanism named Chaos Monkey is also new and has been successfully implemented only by Netflix Teams for their internal projects. The work done by researchers [6] at Netflix introduce the Chaos Automation Platform, a system running failure injection experiments on the production system at Netflix to verify that failures in some services do not result in system outages.

Hlover Tomasson [7] makes use of a framework called as Jata Test Framework for testing distributed system and compared it with the traditional ways of testing.

The thesis is inspired by the work by these authors as they focusses on achieving HA at the application level. During the research it was found that most of the IT companies relies on the

cloud technology for availability. There are many researches and practical works has been done till now on cloud IaaS level but very few considering at the application level. Cloud technology provides mechanism such as load balancer and recovery tools to improve availability but there is a huge scope of availing HA by designing a resilient application. When talking about gaining HA at application level, microservice architecture is adopted by companies like Netflix and Amazon which were hosted on cloud PaaS as containers. The improvement in availability inspires to go more deeper towards the application design level and opens the gate for introduction of fault tolerant libraries in the application and find out its effect.

The current state at NTT Data was studied and it was observed that, there is no solution at application level tested so-far to avail HA. The company is working for Daimler Connected Car solution which needs a solution that can be used to gain HA for the critical applications which will be developed by the NTT team in near future. Currently the company is using solutions at cloud IaaS level and started using OpenShift PaaS to host the applications. The current solution is build using microservice architecture to avail the benefit of scalability but there is no sign for improvement in availability. The next section will describe briefly the current solutions which would be taken into consideration in the thesis work.

2.2 Monolithic to Microservices

Many organizations, such as Amazon, eBay, and Netflix adopted the Microservice Architecture pattern. The idea of this approach is to split the application into a set of smaller components and interconnected services, instead of building a single monolithic application. A service having a set of features and functionality is known as a microservice. Each microservice is a mini application that has its own hexagonal architecture comparable to a monolithic architecture consisting of business logic and adapters. Some microservices can expose a REST, RPC or message-based API which can be consumed by others. There are many advantages of using a microservice approach which includes the advantage of scalability and reliability.

2.3 Cloud-based Design Patterns

The design patterns for availability and resiliency for the cloud based applications are present in the IT world and new patterns keep on adding to the list with an advancement of technology. Availability patterns such as Health Endpoint, Message Queues and Throttling are already in existence. The resiliency design patterns such as Retry and Timeouts are used commonly when designing an application but for cloud based applications which are based on Distributed Systems, these patterns lack features for High Availability. The new patterns such as Circuit Breaker and Bulkhead will be analyzed and used in the thesis work.

The current state shows that there is not much research done on the FT libraries on the application level to achieve HA. The FT library named Hystrix along with new mechanism for testing inspired by Chaos Testing should be tested so that it can be used by NTT Data for their upcoming projects on Connected Car. The current state for achieving HA at NTT Data only focusses on Cloud technology at IaaS level and therefore the thesis works to find the solution at the application design level. The next chapter describes the background study along with the definition of HA.

3 Background

The chapter explains the basic concept of the topic; the main pillars on which the thesis work is carried out are described in this section starting with connected car definition followed by the idea of availability, distributed applications, cloud computing and also gives an overview of different types of failures which cause service downtime.

3.1 Connected Car

“A connected car provides the possibility of internet based data transfer between the car and its surroundings” [1]. The automotive industry has undergone rapid change in the last 30-40 years. In the last two decades, we have seen many advanced electronic devices implemented in a modern car like camera, sensors and advanced infotainment devices. However, now the modern car is coming to a new age where it is accessing and utilizing the power of internet and communication. The connected car is a part of a digital revolution in the automotive industry where the car can connect among themselves and also with other technologies.

The figure 3-1 shows the evolution of the connected car, the features in the connected car will increase with time.

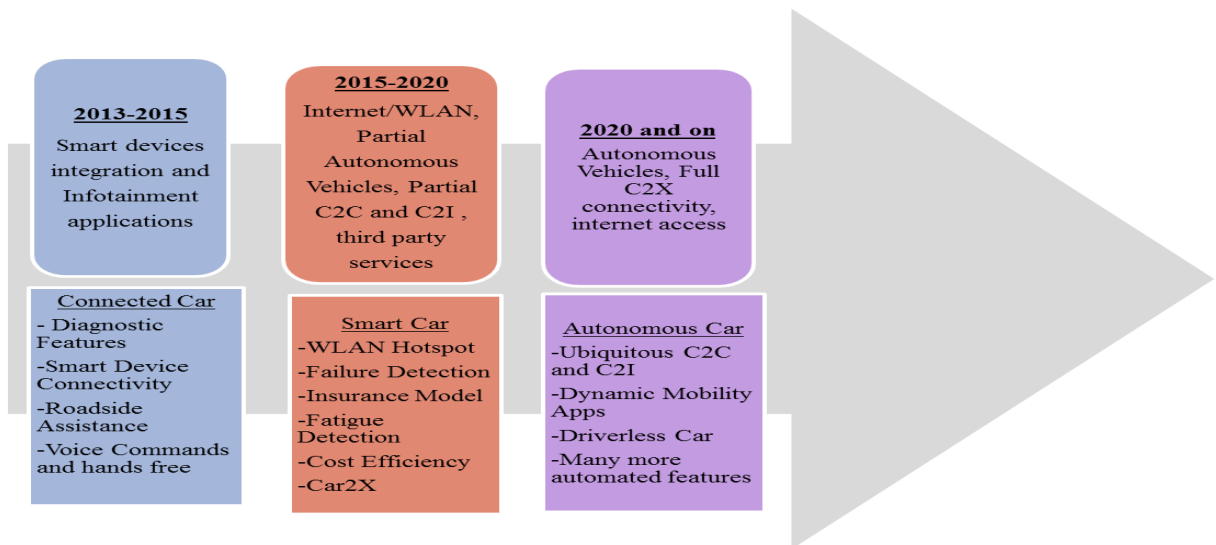


Figure 3-1: Evolution of Connected Car

Connected Car Services

From the user perspective, there are mainly five categories for the connected car services which are shown in the Table 3-1. Traffic Safety offers services which assist the drivers in the situation of accidents and also in the case of breakdowns. The infotainment services include all kinds of pleasure and entertainment applications, ranging from accessing the internet, live music or watching videos on Netflix. Traffic efficiency services provide real-time traffic situation and divert the vehicle to the most efficient route. Cost efficiency category which will

give rise to new business models like insurance, telematics, driver behavior monitoring, and others. Apart from these, there are some applications such as Car sharing, Electronic toll collection, and others, which are associated with convenience and interaction category [1].

To get an overview of the application fields of connected car services, below are the categories listed. The services shown below in the Table 3-1 are those on which NTT Data (Solution Provider for OEM’s clients) is focusing on:

Traffic Safety	Infotainment	Traffic Efficiency	Cost Efficiency	Convenience and Interactions
<ul style="list-style-type: none"> • Smart SOS • Roadside Assistance • Stolen Vehicle Assistance • Geofencing and Speed Monitoring • Remote Diagnostics and Maintenance 	<ul style="list-style-type: none"> • Multimedia Streaming • Live Music Stream • Social Media and Networking • In-car Wi-Fi networks and internet facility 	<ul style="list-style-type: none"> • Traffic Information on Street View • Route Planning • Map Update • Fuel Prices • Parking Information 	<ul style="list-style-type: none"> • Insurance Telematics • Driver Behavior Monitoring • Electric Vehicle Charging • Eco Tax 	<ul style="list-style-type: none"> • Remote Control • Car Sharing and Rentals • Electronic toll collection • Driver profiles

Table 3-1: Example of Connected Car Services [8]

The following section gives a short description of some of the main and upcoming connected car services and applications.

Smart SOS

Smart SOS function is used to transmit information of a car along with the location to the rescue forces in case of an accident or breakdown. If an accident occurred, the detailed information about the position, location, and a number of people affected could be transmitted to the rescue forces which are near to the accidental site. Moreover, a voice connection could also be established automatically under such circumstances.

Geofencing and Speed Monitoring

The service helps to monitor the location and speed of a car. This service is useful for parental controls and also for the companies to monitor their vehicles used by the employees.

Remote Diagnostics

The service transmits data to the repair shop informing in advance about the failure and also informing about the component which needs to be replaced. The service or repair shop after receiving the information can arrange for the part and can prepare themselves beforehand to

diagnose the failure. This service saves time and workforce and improves maintenance services.

Infotainment Services

The services include an in-car internet browser, live music, feeds, on demand videos, messaging, app store, accessing emails and social media.

Traffic Efficiency

The services make the journey comfortable by providing real-time traffic information. It also gives real-time recommendations for a new route plan which has fewer congestions and saves time and resources.

Cost Efficiency and Convenience

New insurance policies which are based on the usage are soon coming to the market. Other very useful services include eco tax, remote services, call center, car sharing, etc.

3.2 High Availability

High Availability is all about designing applications and systems that provide superior uptime, in the wake of component failures. It is the ability of a system to remain functional and operate properly at any given time. Almost every business is reliant on the availability of their systems, services, and applications. Downtime can degrade user experience; moreover, can result in revenue losses. For an e-commerce website, half an hour of downtime can result in millions lost in income and losing customer trust and focus.

3.2.1 Definition of Availability

According to the ITIL V3 [9], the term “Availability” is the ability of any service or application to operate and perform its agreed function when required. In other words, availability is the state of an application being accessible to the end user.

In numerical terms, availability is usually measured as:

$$Availability = \frac{(Service\ Time - Downtime) * 100}{Service\ Time}$$

[4]

Where Service time is the time when service should be available and working often described as agreed service time, Downtime is the time when the service is unavailable.

3.2.2 Availability Parameters

The factors that define and measure availability are discussed below:

Mean Time between Failures (MTBF)

Mean time between failures is a reliability term used throughout many industries which refers to the average amount of time that a service or any product functions before failing [10]. MTBF is a unit of measurement that includes only the operational time between failures and does not include repair time. It is commonly used to measure reliability, which is the average time interval between two consecutive failures. The time interval is calculated in terms of a number of hours.

Mean Time to Repair (MTTR)

Mean time to repair is the time taken to detect and fix a defect. The defect can be a hardware module failure, a system failure or some software failure [10]. MTTR is also measured in terms of hours. For example, the estimation of Hardware MTTR varies from 30 minutes to 24 hours for the on-site operation and from 1 day to 7 days for an off-site operation.

Availability Measure

Availability is often measured by the rate of service availability and calculated in terms of percentage or number of nines. The standard way of calculating availability is stated below:

$$Availability = \frac{MTBF}{(MTBF+MTTR)} \quad [10]$$

Where, MTBF and MTTR are taken in number of hours.

The traditional stability approach for gaining availability is to maximize MTBF, but this approach is not suitable for distributed systems. Failures in today's complex, distributed systems are not predictable, and situation is getting unpleasant with the introduction of cloud-based systems and microservices. The thesis focusses on the mechanism of minimizing MTTR, and this can be done by making applications resilient by preparing them to handle unexpected failures. In the thesis work, this definition is considered and worked on with a aim of minimizing the mean time to repair by making applications resilient.

3.2.3 High Availability Mechanism

Availability is expressed as a percentage of time when the system was up and working fine. In reliability engineering, availability calculated in percentage are sometimes referred in terms of a number of nines. This section gives a detailed description of the mechanism of defining availability.

The Table of Nines and High Availability

Availability is usually specified in nines notation. It is measured by the number of nines which is stated in the Service Level Agreement (SLA). SLA is a contract committed to the customers by the service provider to reach a defined level of operational performances and availability of services to meet the requirements. It allows a complete transparency between clients and service provider.

To get an idea, below are some of the commitments written in the SLA which is usually offered by a service provider to its clients:

- Server Hosting
- High Availability of services: Defining availability rate that must be fulfilled under all circumstances. Circumstances which causes downtime can be a power failure, network problem, server or software problem.
- Planned Downtime
- Penalties and other legal conditions

A Historical Perspective

In the late 1950s computers built offered around twelve-hour mean time to failure. At that time, dozens of engineers and maintenance staff could repair the system in about eight to ten hours. On calculation, we can estimate that this failure-repair cycle could provide the availability of only 60%. This percentage of availability means that a system can rarely operate for more than a day without any interruption or failure on average. The primary source of failures in those systems where the hardware components such as vacuum tube and relay devices; having a lifetime of a few months.

With the advancement in technology, many fault detection, and masking techniques were introduced and checkpoints were implemented which saved the state on a stable media. Once a failure occurred, the program read the most recent checkpoint and continued the operation and computation from that point. By 1980, well-run computer system offered an availability of 99% [11]. Comparing with the year of 1950's, 99% availability sounds good, but this much of availability means 100 minutes of downtime per week. Critical applications require much more availability delivering an uptime rate of 99.999%. The percentage states that the service has a downtime of at most five minutes per year.

Below Table 3-2 shows the degree of availability which is characterized by orders of magnitude.

Availability Measures			
System Types	Unavailability (min/Year)	Availability	Availability Class
Unmanaged	50,000	90%	1
Managed	5,000	99%	2
well-managed	500	99.9%	3
fault-tolerant	50	99.99%	4
high- availability	5	99.999%	5
very-high- availability	.5	99.9999%	6
ultra-availability	.05	99.99999%	7

Table 3-2: Availability measure in terms of number of nines

On an average, unmanaged systems on the internet fail every two weeks and take an average of ten hours to recover giving an availability of 90%. Managed systems also fail several times a year and take around two hours to repair [12]. These managed systems provide about 99% availability. Similarly, the fault-tolerant systems fail once every few years giving an availability of 99.99%. With five nines, highly available systems provide a downtime of around 5 minutes per year. As the number of nines increases, downtime of the system decreases. If the system's availability is A, the availability class is calculated as $e^{\log_{10}(\frac{1}{1-A})}$ [13].

Connected car applications are developed on the principle of distributed systems, and therefore the need for high availability arises because of the complex nature of these systems.

3.3 Distributed Systems

Distributed system is a network of computers that interacts with each other in order to achieve a common goal. Based on the principle of distributed systems, a distributed applications or software runs on multiple computers or hosts within a network at the same time and performs a single task or job by interacting with each other. The application can be stored on a server or with cloud computing and can communicate from any geographical location. One of the biggest examples of a distributed system is the World Wide Web. Connected car apps and services are built on the principle of distributed systems which have the capability to interact within or with third party services. Nowadays, systems are becoming complex where loosely coupled systems interact frequently to each other. Figure 3-2 given an idea about a complex distributed system where so many systems are working independently on different platform and interacting to each other for a common goal.

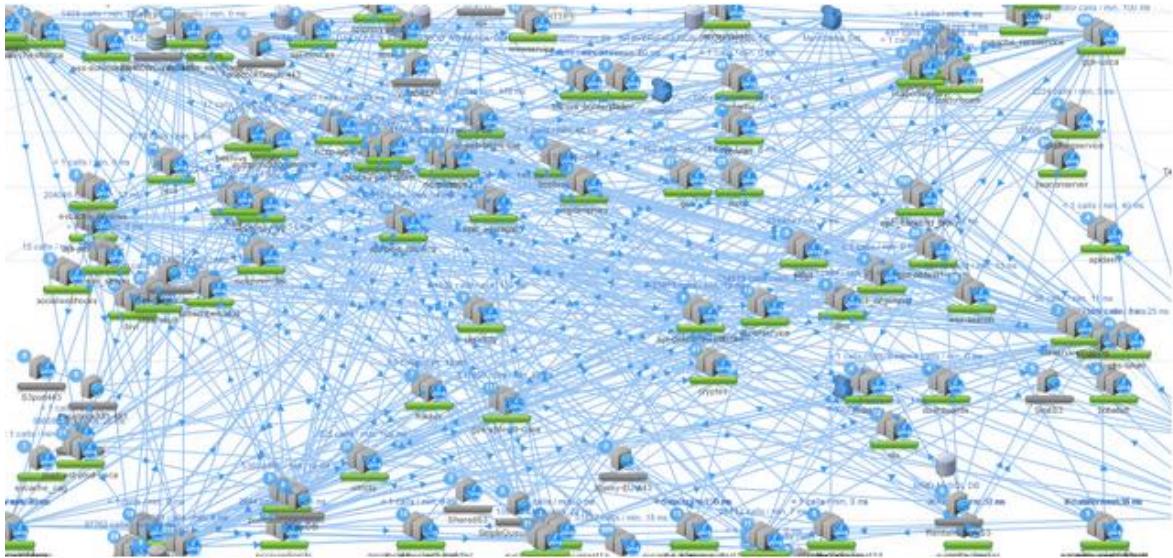


Figure 3-2: Complex distributed system [14]

According to [15], “*Distributed systems can be particularly challenging to program, for a variety of reasons. They can be difficult to design, difficult to manage, and, above all, difficult to test.*”

Distributed applications can provide more powerful features when compared with single stand-alone systems. However, it is not easy to handle distributed applications because of their nature of interacting with several simultaneously running components. To be truly reliable, a distributed application must have the following characteristics:

- **Fault-Tolerant:** The capability to recover from component failure without processing the incorrect response.
- **Highly Available:** Applications are always available and also resilient to face any failure.
- **Scalable:** In the situation when the system needs to be scaled up, the application should work correctly.

The thesis work tries to find the solution to one the most difficult challenge which is: Distributed systems must operate correctly even when a component fails. The challenge is to stop cascading failures, and in the upcoming chapter, various design patterns and fault-tolerant libraries are analyzed to make distributed applications resilient and improve their availability.

Many distributed applications must handle failures such as components failure, system crashes, application deadlocks, and application livelocks. A common way for a distributed application to tolerate crashes is to detect them and recover from them explicitly. Detection of failures in a distributed application is hard and usually takes much longer than recovery.

3.4 Failure Characteristics

According to Werner Vogels, “*Everything fails all the time*” [16]. Failures can happen anytime, and it is crucial to detect them as early as possible. In the field of computing failures can be mainly classified as a hardware failure or a software failure.

3.4.1 Hardware Failures

Hardware failures can cause downtime. According to Brian Kirsch, “*With the popularity of software features that help improve application availability; IT professionals often forget that they cannot prevent every hardware failure, which causes them to forget the plan to recovery.* [17]” There can be many causes of a failure during the life cycle of a product, some of which are discussed below:

- Design failures: The main cause of this type of failure is due to inherent design flaws in the system. If well designed, this class of failure should make a small contribution to the total number of failures.
- Infant mortality: The type of failure can cause newly manufactured hardware or components systems to fail. This failure may be due to some poor soldering or leaking capacitor etc.
- Random failures: Hardware failures can occur anytime during the life cycle of a hardware component. Sometimes these failures can cause system failures and most of the time the only option to avoid them can be a redundancy of these elements.
- Wear Out: Degradation of component characteristics can cause components to fail, so better to detect the degradation and take preventive measures.

Bathtub Curve:

According to N. Slack “*Failure, for most parts of an operation, is a function of time* [18]”. Plotting of failure rate against a continuous time scale results into a curve which is commonly called as a bathtub curve. The graph in figure 3-3 shows how different failure modes discussed above contribute to the overall failure rate. The time duration indicates that the first phase is the early life phase where the failure rate is more than the useful life period. After a useful life period comes a wear out period where the failure rate again goes up, and there is a rising chance of hardware failure.

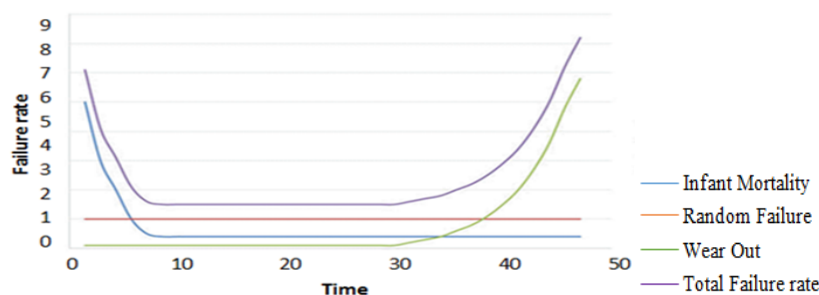


Figure 3-3: Bathtub curve [18]

3.4.2 Software Failures

Software failures can occur anytime, and it can be minimized by keeping track of software errors and defect density. Keeping track of defects and failures can avoid future failure. Defect density can be typically measured as a number of defects per thousand lines of codes ($\frac{defects}{KLOC}$). There are a number of factors on which the software failure density depends which are discussed below [19]:

- Software process incorporated during the design and code
- Complexity of the software product
- Size of the software
- Experience of team members
- Testing process
- Code reused from previous software

3.4.3 Causes of a Downtime

“Downtime means that a system or service is not working at a given time. [20]” There are many reasons for a downtime; it can be related to some hardware failure or some software failure or maybe due to some human error. Downtime is categorized into planned and unplanned downtime. Below Table 3-3 shows some of the causes of an unplanned downtime. In a broader fashion, they can be classified as system faults, data faults, media errors, or site outages [20].

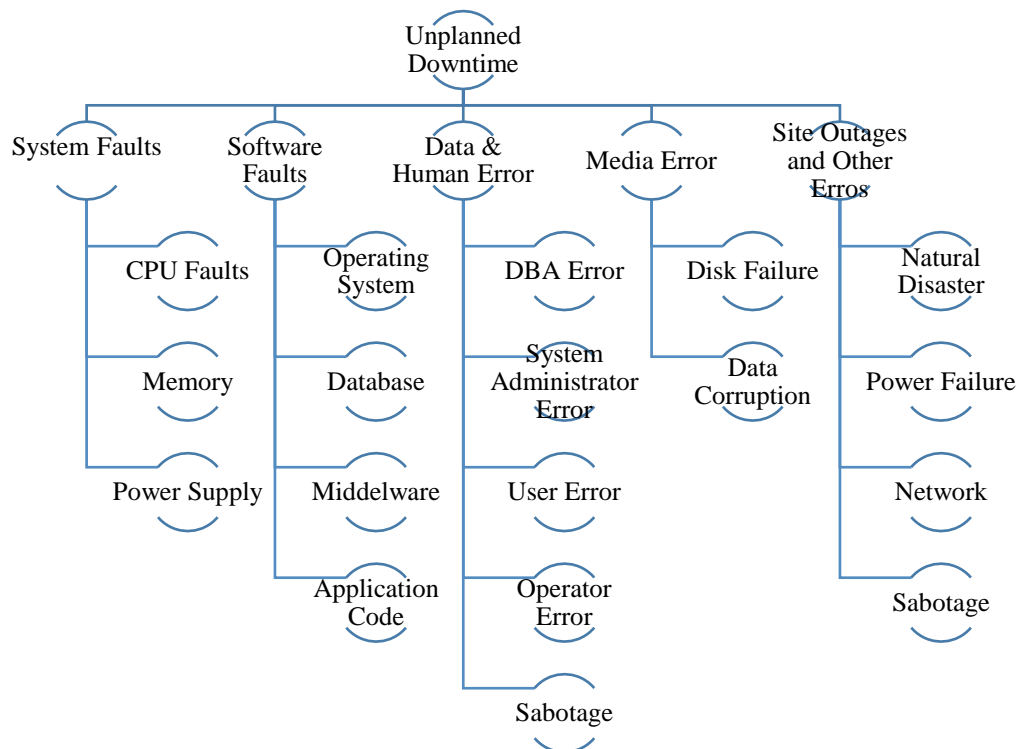


Table 3-3: Unplanned Downtime [20]

The unplanned error is disruptive and can be serious because it is tough to predict them and even harder to predict its timing. Planned downtime also causes almost the same impact on the service. Planned downtime can be classified as periodic maintenance, routine operations, and planned upgrades as shown in Table 3-4. Planned downtime can also be disruptive, especially in global enterprises where the users access the service in multiple time zones [20].

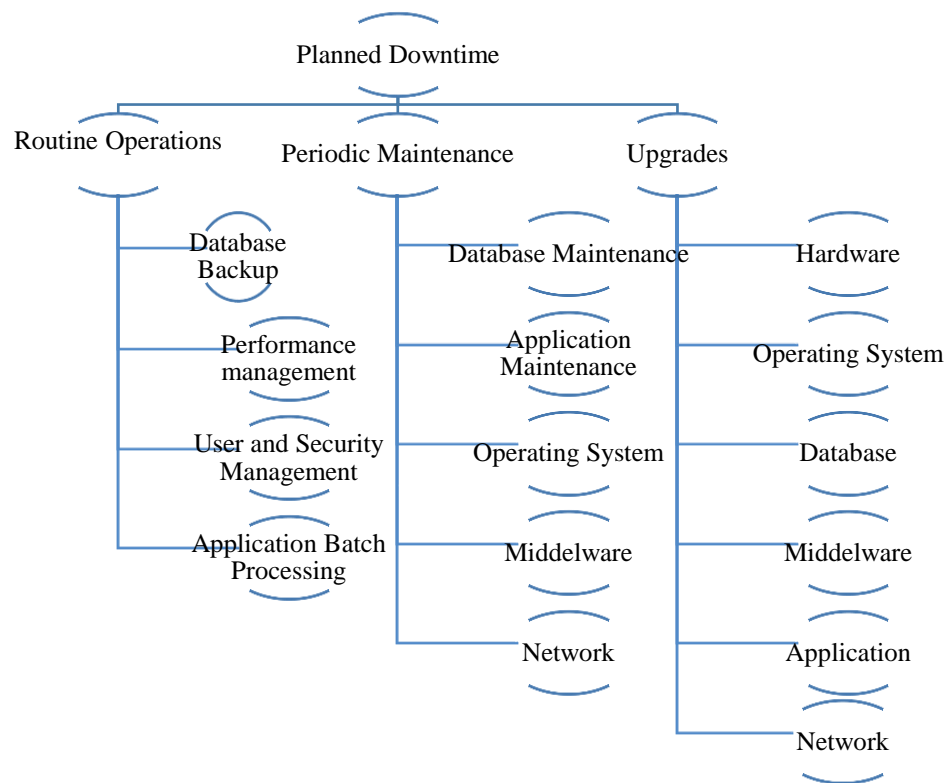


Table 3-4: Planned Downtime [20]

Distributed systems are developed keeping in mind that failures will occur and the section above dictates several types of failures which are impossible to ignore and therefore applications should be capable of handling these failures.

3.5 Cloud Computing

Cloud computing emerged as a novel technology which has come into existence since the year of 2000. In simple terms, cloud computing is a computing paradigm, where a large pool of systems are connected in private to provide shared processing resources and dynamically scalable infrastructure and services for all types of applications, data, and storage. According to David Linthicum, *“If you think you have seen this movie before, you are right. Cloud computing is based on the time-sharing model we leveraged years ago before we could afford our own computers. The idea is to share computing power among many companies and people, thereby reducing the cost of that computing power to those who leverage it. The value*

of timeshare and the core value of cloud computing are pretty much the same, only the resources these days are much better and more cost effective. Moreover, you can mix and match them to form solutions, which were not possible with the traditional time-sharing model. [21]”

Cloud computing makes it possible to access information from anywhere around at any time. According to Mell & Grance of NIST (National Institute of Standards and Technology), CC is “*a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction*” [22].

Some of the key features of cloud systems are as follows:

- **Utility Computing:** Customers are charged on the basis of specific usage rather than a flat rate, that means customers are charged when they are using the service, this type of policy makes cloud computing very attractive for most businesses and especially to the new growing businesses.
- **Elasticity:** When it comes to differentiating cloud technology to other domains, elasticity is the characteristics which give cloud computing more power and value. It gives an ability to increase the workload on its current and additional (dynamically added on demand) resources very easily and efficiently.
- **Availability:** Cloud computing relates closely to the characteristics and need of availability. If a service is down, it is of no use.
- **Ease of use:** Cloud computing reduce the overhead for managing resources and monitoring them in an easy way. The users are enabled to provision their own services and manage their service characteristics.

Cloud computing has mainly three types of cloud deployment models shown in the figure 3-4: private, public, and hybrid. Community cloud model is an additional type of model which is not commonly used.

- **Private Cloud:** Built and managed within a single organization where the team uses software that enables the functionality of cloud processing, some of the common examples would be VMWare, vCloud Director, or OpenStack.
- **Public Cloud:** The third-party providers provide computing resources. Some of the most common examples would be AWS (Amazon Web Services), Google AppEngine, and Microsoft Azure.
- **Hybrid cloud:** As the name suggest this type of model incorporates the functioning of both public and private cloud models where the computing resources are provided by both private and public clouds.
- **Community Cloud:** Computing resources are shared among several organizations, managed by third-party providers or the IT team of an organization.

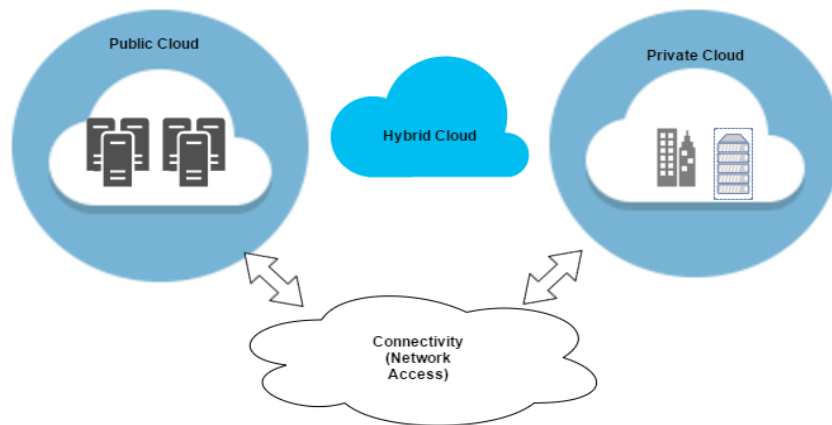


Figure 3-4: Cloud Deployment Models

3.5.1 Cloud Computing Reference Architecture

NIST (National Institute of Standards and Technology) defines a set of actors, activities, and functions which act as pillars for the process of developing cloud computing architecture. The architecture describes five principal actors where each actor is an entity that participates in a process and performs the task in cloud computing [23]. The architecture is shown in the figure 3-5 and discussed below:

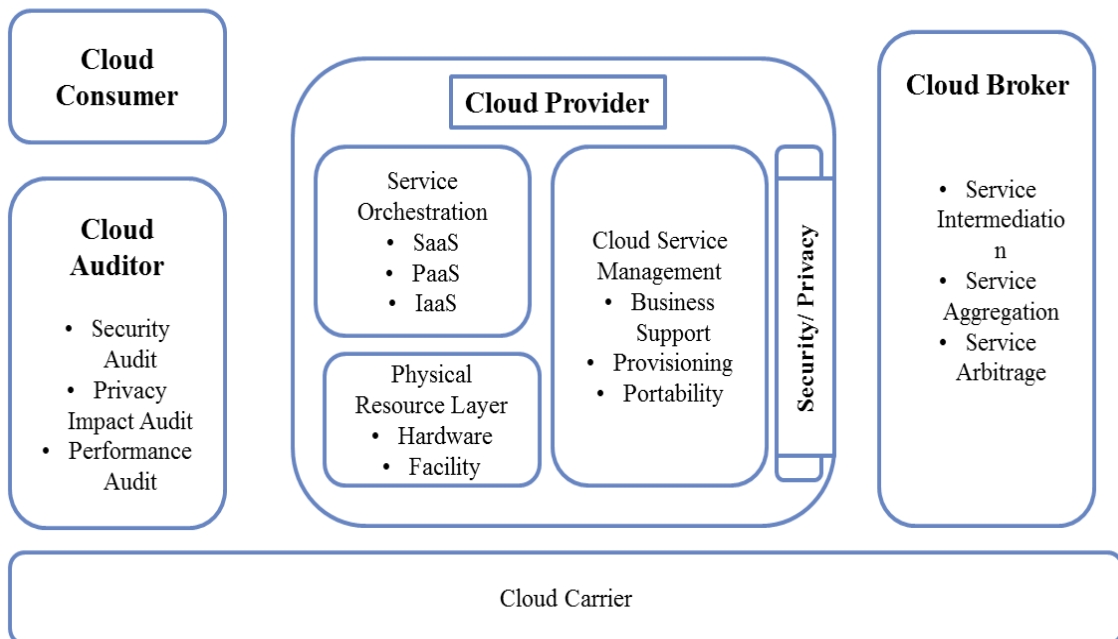


Figure 3-5: Cloud Computing Architecture [23]

Cloud Consumer: Person, or organization that maintains a business relationship with, and uses service from Cloud providers.

Cloud Providers: Entity or organization responsible for making a service available to Cloud Consumer.

Cloud Broker: An entity that manages and accountable for the delivery of cloud services. Also responsible for negotiations among Cloud Providers and Cloud Consumers.

Cloud Auditors: A party or organization that can conduct an assessment of cloud services independently. Assessment can be of performance, security or some information on system operations.

Cloud Carrier: Provides connectivity and transport of cloud services from Cloud Providers to Cloud Consumers.

Service orchestration is the system component that supports cloud provider to provide services to the consumers. The lower layer consists of physical resources such as computers, storage components, networks, etc. In cloud broker level, service intermediation enhances services provided by cloud broker and service aggregation combines multiple services to provide some new services. Service arbitrage works same as service aggregation and provides more flexibility.

3.5.2 Cloud Service Model

Cloud computing so far has three service models categorized on the basis that how services are provided to the clients. These three service models as shown in the figure 3-6 are known as IaaS (Infrastructure as a Service), PaaS (Platform as a Service), and SaaS (Software as a Service). These services are commonly used as a combination but have different work role and characteristics which will be discussed in the upcoming section:



Figure 3-6: Cloud Service Models

The **IaaS (Infrastructure as a Service)** model provides the computing resources associated with the infrastructure components. The infrastructure components may include the primary storage, servers, data center space, networks, firewall, load balancers, and so on. IaaS is a cloud infrastructure service, where instead of having your own infrastructure or instead of having to purchase and manage hardware, users can buy the service offered by the IaaS provider on a consumption basis. The service provider of IaaS manage servers, virtualization, storage, hard drives, and networking and allow users to install and run any required platform

and manage them without worrying about the infrastructure. Users can build a virtual data center in the cloud and scale them as per need without worrying about the physical maintenance and management of it.

Characteristics of IaaS

Some of the most common features of IaaS are as follows:

- Dynamic Scaling
- Resources are distributed.
- It has a variable cost, a model of utility pricing.
- A single piece of hardware can serve many users.

Cases where IaaS make sense

- Volatile Demand: The demand is elastic which means any time one can expect significant spikes in terms of demand on the infrastructure.
- New Organizations: Organizations which are new to the market with less capital for the infrastructure setup.
- Trial basis: where an organization wants to try something on a temporary basis.

IaaS Examples: Amazon Web Service (AWS), Cisco Metapod, Microsoft Azure, Google Compute Engine (GCE), Joyent.

The **PaaS (Platform as a Service)** model operates at the layer above raw computing hardware; it may be a physical hardware or a virtual one. PaaS is a platform for developers who can simply code their application and deploy them on the PaaS without worrying about the environment required for their application to run because PaaS manages everything on its own. It provides a method for programming languages to interact with servers like database, web server, or file storage, without dealing with the lower level of details and requirements. This service model manages everything for the application like database needs, making a copy on several servers, distributing the workload, etc. So basically PaaS service model provides a platform on which software can be developed, deployed and run. It abstracts all the complexity and manages itself and also manages the underlying hardware and infrastructure.

Characteristics of PaaS

Some of the key features are stated below:

- Integrated development environment: develop, test, deploy, host, and maintain applications in the same environment.
- User Interface tools: Web-based user interface tools to create, modify, test and deploy different scenarios very easily.
- Multi-tenant architecture: Multiple concurrent users utilize the same development application.
- Scalability: Load balancers and a failover mechanism.
- Collaboration: Project planning and communication tools.

- Easy integration: Very easy integration with web services and databases.

PaaS Examples: Openshift, Windows Azure, Heroku, Google AppEngine.

The **SaaS (Software as a Service)** model is the top layer of cloud computing model, which provides software solutions for end users like email solution, office, or a business CRM. It also charged as per user and provides flexibility according to the number of users. SaaS application usage reduces the cost of software purchases and ownership and also removes the need for technical teams and administrators to manage, install, and upgrade the software.

Characteristics of SaaS

Some of the key features are stated below:

- The software package is operated from a central location which makes it easy to administer them.
- Web access is provided to access the software.
- Integration between the two software components is easy and allowed with the help of Application Programming Interfaces (API).
- No initial setup cost and updates of software are automated.

SaaS Examples: Google Apps, Salesforce, Workday, Concur, Cisco WebEx.

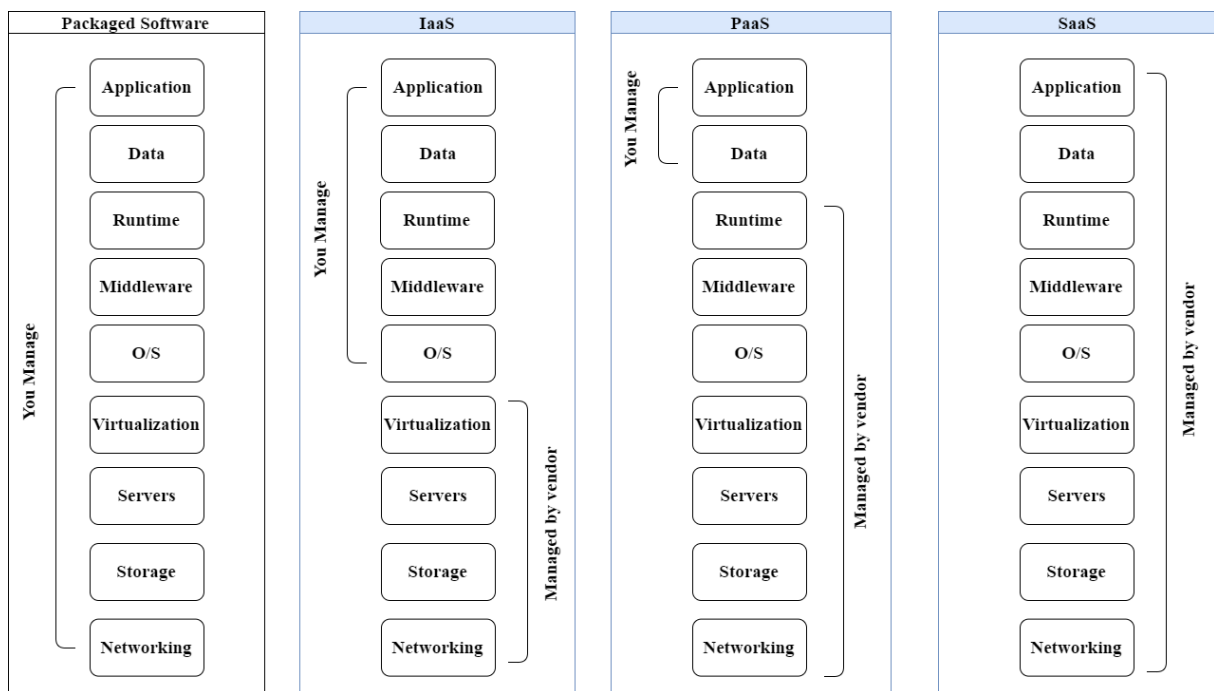


Figure 3-7: Cloud Service: IaaS, PaaS & SaaS

Cloud computing is very beneficial and used frequently nowadays for every business. The figure 3-7 shows how the cloud service is managed at different levels such as IaaS, PaaS and SaaS. There are many different ways and mechanisms for improving the availability of services at IaaS level but the thesis work focusses on how to reach the availability at application level, therefore, the implementation is done using a PaaS platform where

Openshift is used to host the the application and chaos monkey service which will be used for the testing.

The next section, answers that how connected car and cloud computing are interrelated and this technology is needed for the future car.

3.5.3 Connected Car and Need for Cloud Computing

Global industries are going through a period of transition, especially in the automotive sector. The connected car of future will come out with many features and new services. As the connected Car industry adds new capabilities, it also faces challenges in the form of ever-growing complexity and cost of managing IT assets. The cloud can provide much-needed elasticity both in terms of cost and agility needed for the future growth.

The overall picture of the IT system landscapes with context to connected car is described with the help of figure 3-8. The general architecture shows that the backend server uses cloud technology that means every IT based functionality relies on cloud computing. The frontend can be anything like a smartphone or any display in the car which access the information and services from the backend. Cloud computing is used in the connected car as it offers scalability which is very much needed in this model. Connected car will access third-party services and applications which can be seen in the figure named as content-provider. The car itself has an onboard unit which has the capability of communicating and accessing different services.

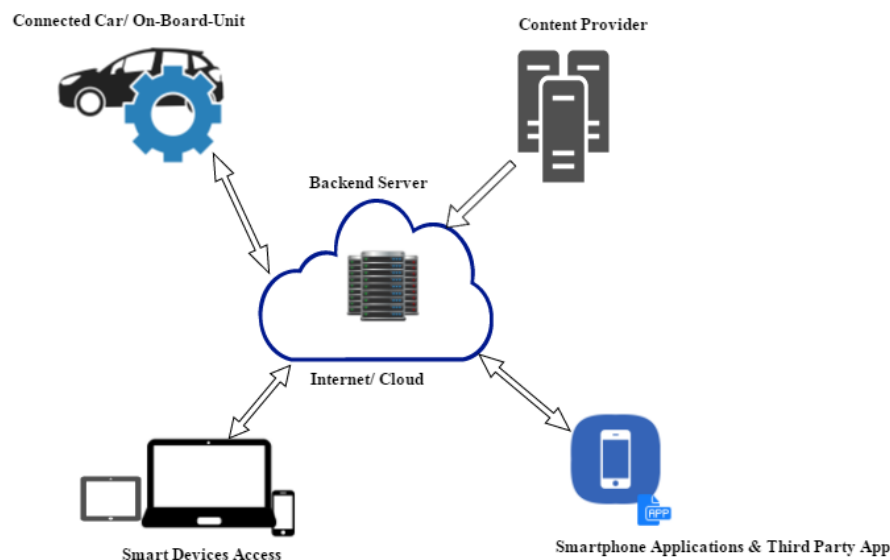


Figure 3-8: Cloud and Connected Car

Cloud computing is already used by several OEMs for the connected car. For example Daimler is using the technology for after sales services and soon in the Projects: Daimler coding service and Update Over the Air on which NTT Data is currently working on.

The previous sections described some fundamental concepts of the thesis. Connected car is the future of the automotive sector, and the prime goal is to provide interactive services to the users. This could be possible when services are developed on a distributed scale and use the cloud delivery model. This complex structure gives birth to many challenges which are discussed below. These issues are still open, and there is a need to tackle them. [24]:

- **Availability:** The proportion of time that the application is functional and working. Some of the causes of downtime are system errors, application level errors, infrastructure problems, malicious attacks, and load on the system.
- **Resiliency:** The ability of a system to handle failures gracefully and try to recover from them as quickly as possible. In distributed systems where applications use shared platform services, communicate over the network and interact with one another increases the likelihood of failure increases. Challenge is to detect the failures soon and recover efficiently.
- **Performance and Scalability:** The responsiveness of a system to execute a request within a given time interval is measured as performance, whereas scalability is the ability of the system to handle the increase and decrease in load by scaling up in the case of an increase in demand and scale-down when demand decreases. The applications are running on cloud encounter variable workloads. Therefore, it is important to maintain the performance and handle scalability efficiently.
- **Management and Monitoring:** Distributed cloud applications run in many different data centers, and with so many services running and interacting, it's hard to monitor the real-time action. In order to monitor and detect problems, applications must expose real-time information which can be monitored by administrators and operators.

The issue discussed above depicts the prime focus of this thesis work and gives the direction towards analyzing new concepts at the application level to find solutions.

4 Analysis

In the previous chapter, an overview of the problems and challenges were discussed, this chapter is a detailed description of the analysis of new mechanisms that are used to obtain high availability at an application level. This chapter also gives an overview of the findings and comparison between different techniques used for fault tolerance.

4.1 Service Architecture

Microservice architecture is a new trend for the development of an application. The focus is to develop a set of small independent services, which run as a stand-alone process. The principle of microservice opens the gate for a new mechanism which can make the services highly available. The below section describes the traditional method and compares it with the microservice architecture.

4.1.1 Monolithic Approach for Software Development

It is the traditional approach used for the design of a software application. In simple terms monolithic means “all in one piece,” i.e. a monolithic application is built as a single unit. Generally, a web application is based on three parts: a database consisting of many tables and data, a client-side user interface composed of HTML pages and JavaScript running in a browser, and a server-side application [25]. Traditional models for software development usually involve large teams working on a single, monolithic deployment artifact.

At the core the business logic is defined, which usually consists of the implementation of different services, domain objects, and various events. Outside the logic are different adapters which interface with the external world. For example, messaging components, database access components, and web components. This type of application has a logically modular architecture, but it is packaged and deployed like a monolithic one. With different language patterns come different ways of deployment, for example, many Java applications are packed as war files and then deployed on servers like Jetty, Tomcat, etc.

Benefits of Monolithic architecture:

- Simple to develop: Most of the IDEs and other development tools are focused on the development of a single application.
- Simple to test: Implementing end-to-end testing is easy by launching the application and testing the UI with some easy to use testing tools like Selenium.
- Simple to deploy: Copying the package to the application server and deploying them is an easy task.

The ongoing trend in the software market indicates that applications have a habit of growing over time and eventually becoming huge. In agile development, in each sprint, the

development team implements more and more stories, which means adding more functionality and lines of codes. Once the application becomes large, the monolithic approach starts facing problems [25]. Some of the drawbacks are explained below:

- Complexity increases with time, and hard for any developer to understand the application fully.
- The problem with reliability and availability i.e. a bug in any module can bring down the entire process. In monolithic architecture all instances of the application are identical, so a bug on one module component can impact the availability of the entire application.
- Application grows and therefore takes longer to start up and up time which impacts the deployment. Overloaded web containers take a long time to start which has an enormous impact on developer productivity.
- Continuous deployment is difficult. To update one component, the entire application has to be re-deployed.
- Fixing bugs and implementing new features becomes challenging and time-consuming.
- Overloaded IDE with larger code base makes it slow and decrease the overall efficiency of a developer. The overloaded IDE takes more time to function like debugging, changes, clean code, etc.
- The obstacle to scaling developments and a barrier to adopting new technologies.

Netflix Example:

Testing monolithic applications is very challenging and arduous, as the application grows, the complexity also increases and therefore testing any bug or failure is time-consuming. In the year of 2000, Netflix was using monolithic approach, and as they grew, the complexity of the system increased rapidly [26].

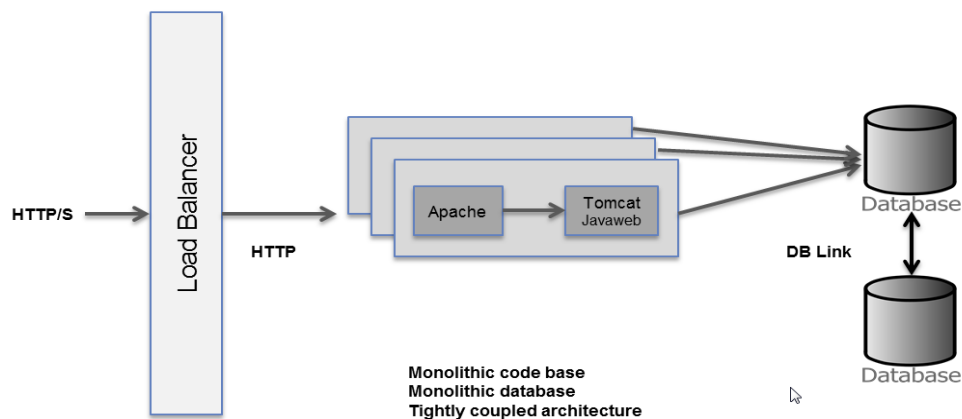


Figure 4-1: Monolithic Architecture [27]

The figure 4-1 shows the data center of Netflix in the year 2000 when they were using monolithic approach. The environment on which the application was hosted and running comprises of the following components:

- Hardware based load balancer
- Java web application hosted on Linux platform
- Monolithic code base where everyone was contributing to a single code base
- Monolithic database

Having a monolithic code base created problems at Netflix, in case of any error arises due to change in code, testing becomes tough. The problems in monolithic applications were analyzed which resulted in the conclusion that for modern businesses, this architecture is not suited and therefore microservices will be analyzed and discussed in the next section. Before discussing the microservices, it is important to understand the concept used before them such as web application design and rich client application design. A small introduction to the topics are stated in the section:

Web Applications: An application that can be accessed by a web browser is known as web applications. The users access applications using specific URLs which are used by the browsers to create HTTP requests to resources placed on a Web server. On receiving the request, the server returns the HTML pages to the client which are displayed on the browser. The core of any application is the server-side logic, and the architecture comprises of three layers named presentation, business and data layer [28]. Web applications were designed keeping in mind the complexity of the monolithic approach. The main characteristics of designing the web application such as:

- Logically partition the application using the layers presentation, business, and data access.
- Implementation of loose coupling between layers by defining interface components.
- Communication between the components and use of caching to minimize the load on the server.

Rich client application was designed in a way that can provide high performance, interactivity, and rich user experiences for applications. This application can operate as a stand-alone, connected or in disconnected scenarios. A traditional web application puts a heavy workload on the server side where the server needs to maintain the user session, process the requests and process results. On the contrary, rich client leverages the power of client machine. It installs a runtime (such as a plug-in) on the client and executes the functions locally.

4.1.2 Microservices – Tackling the Complexity

The microservice architectural style is an approach of developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.

-Martin Fowler [29]

Microservice architecture is an approach of building large enterprise applications and distributed systems by implementing multiple small services; each service can be developed, deployed and tested individually. Each service runs individually either on a single machine or on different machines and executes its own process separately. These services can communicate and interact to one another using some communication protocol like REST web services. Each service can have its own database, or they may share a common database or storage.

From the architecture point of view, each functional area of the application is implemented by a small service which means a complex application is now split into a set of simpler applications. Each backend service exposes a REST API which can be consumed by other services.

Example to show the need of microservices: In the figure 4-2, on the left side, the architecture shows a monolithic way of building an application. The application is for carpooling, and in a monolithic structure, the development is simple as seen on the left side of the figure 4-2, the business logic is kept at the central location and new features are added to the core making it complex. With the increase in the complexity, it is difficult for a single developer to understand the codebase and make changes. The application is difficult to scale because different modules have conflicting resource requirement. Testing takes time when the code base is very large and debugging is difficult therefore to tackle these complexities the same application is built using microservice architecture shown on the right side. The system is decomposed, and each functional area is implemented by its own microservice. The application is split into simpler services which can interact with each other. The microservices are now easily managed and configured.

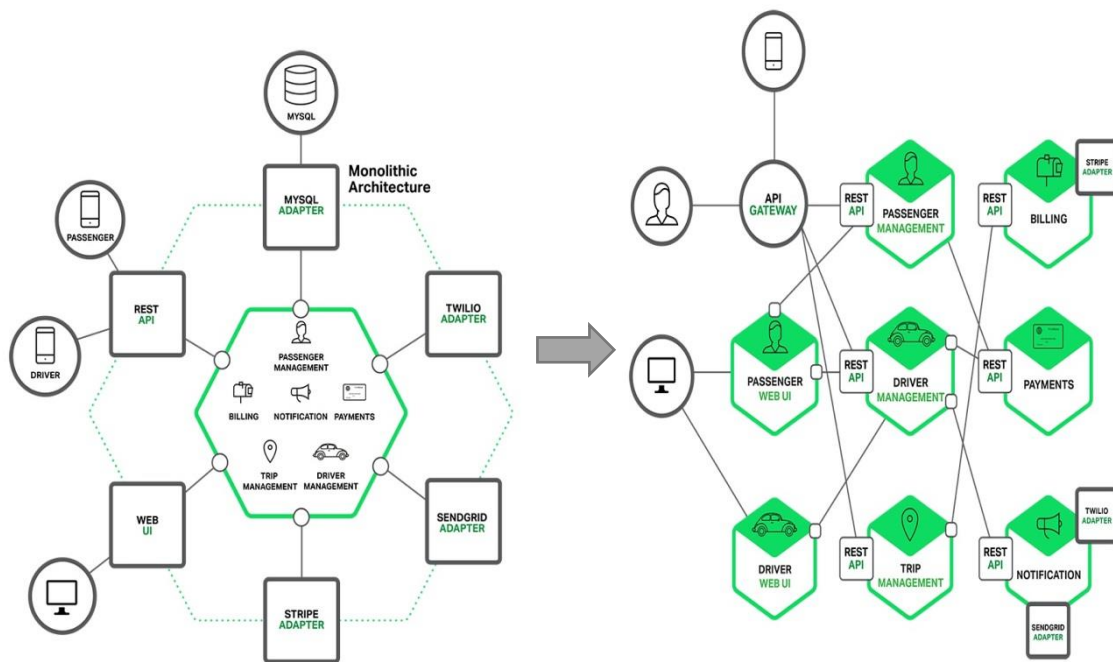


Figure 4-2: Monolithic vs. Microservice Architecture [30]

In order to understand more about the scalability and how does it related to microservices, below three scaling mechanisms are mentioned. The property of scalability is important when dealing with availability and therefore it is explained with the help of a scale cube model.

Microservice Scalability: The Art of Scalability

The book, *The Art of Scalability*, written by Martin L. Abbott & Michael T. Fisher, describes a three-dimensional Scalability model known as scale cube. The scale cube comprises of an X, Y and Z axes where each axis addresses a different approach to scale a service. The point where the value of coordinates is zero ($X=0, Y=0, Z=0$) is known as a worst case scenario where a service is developed with a monolithic approach, and all the functions of the service exist within a single code base on a single server.

X-Axis Scaling:

X-axis scaling holds the concept of scale by cloning. The concept here is to run multiple copies of an application behind a load balancer.

For example, if there are N copies then each handles $1/N$ of the load. This approach is simple but has a drawback as each copy accesses all the data of the application and therefore caches require more memory.

Y-Axis Scaling - Microservices:

The microservice architecture corresponds to the Y-axis where the application is split into multiple smaller services. It defines an architecture which structures the application as a set of loosely coupled and collaborating services. The service implements a set of related functions and communicates to other services using protocols such as HTTP/REST. Each service has its own database and thus is decoupled from other services. These independent services can be easily scaled according to the requirements. In practice, there are two well-known approaches

to split the application; first approach is to use verb-based decomposition, where the services are defined that implements a single use case. The second option is to split the application by noun, where the services are responsible for all operations related to a particular entity. An application can use both the approaches if needed.

Z-Axis Approach:

Z-axis is similar to x-axis approach as z-axis scaling mechanism also runs an identical copy of the code on each server, but the difference is that each server is responsible for only a subset of the data. In z-axis scaling approach, each server only deals with a subset of the data which improves the utilization of cache and also reduces memory usage & I/O traffic. Z-axis approach claims many benefits but on the other hand, increases application complexity [30].

Characteristics of Microservice Architecture [31]:

- **Componentization:** Breaking down of the application into small services. The software application can now be developed by plugging together different components, much in the way we see things being developed in the physical world.
- **Organized around business capabilities:** The services are split around the business capabilities. Each service can implement software for the specific business area, including user-interface, storage, and any other external collaborations.
- **Product model:** Microservice allows developers to handle a complete product over its full lifetime. Earlier development team work on the project model where after completion of a project, the software application is handed over to a maintenance team. Amazon believes in the notion of “*you build it, you run it*” where after the development of a service, the development team also takes the responsibility of the maintenance and support of the product and works in close cooperation in case of any update and modifications.
- **Decentralized Governance:** Splitting the monolith’s applications into services gives the chance to decide about the technology for each of them separately. Each service offers some functionality, and the technology best suited should be used, and microservice allows the developer to do so. Each team working on different services can use the technology of their own choice enhancing the functionality and efficiency.
- **Design for failure:** Application needs to be designed so that they can tolerate any failure to its services. If one fails, it should not result in cascading failures. There are several ways to which microservices are designed for failures.

Microservices clearly wins against the monolithic approach for designing a cloud based web application; now there is need to analyze the design patterns and fault tolerant library which can be used to make them resilient and design for failures. Before describing in detail about the design patterns and libraries, below section 4.1.3 gives an overview of Docker which manages the application containers.

4.1.3 Docker – The Container Standard

According to [32] “*Docker is an open platform for developers and system admins to build, ship, and run distributed applications.*” In simple terms, it is an open source tool for running isolated containers on operating system platforms which results in the fast deployment of applications inside containers. Docker has the capability to create portable, self-sufficient containers from any applications. The concept of containers enables application developers to pack up the entire contents of the application, for example, all the libraries, code and other dependencies associated with the application and ship it all as one package, figure 4-3. The advantage of containers is that after binding everything in one package, the package can be deployed on any machine running on Linux without any other installation taking place. Docker container wraps everything required to run the application, and therefore the application can run in any environment like production system or on cloud solutions.

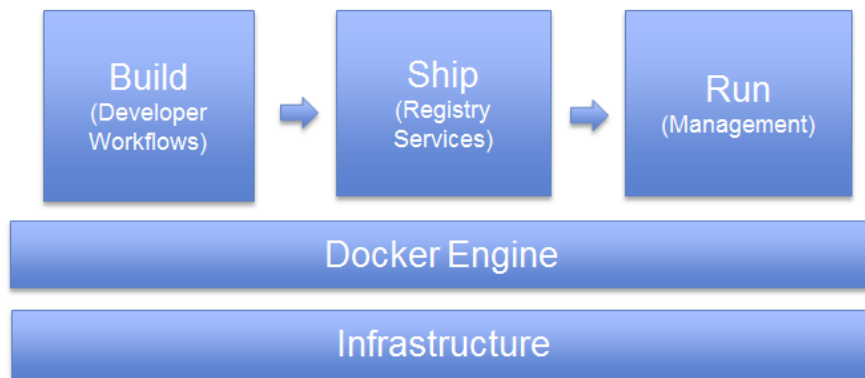


Figure 4-3: Docker for Developers

Docker Engine is a client-server application having components as:

- A server: Long-running program known as a daemon process.
- A REST API: API which specifies interfaces used by programs to talk to the daemon and provide with instructions.
- A command line interface client.

Clearing out the difference between virtual machines and containers is important. Below with the help of figure 4-4, the basic difference is stated.

Containers vs. Virtual Machines

Containers and VMs do not oppose each other; rather they complement each other for combinatorial benefits. Containers provide operating system level process isolation whereas virtual machines provide isolation at the hardware abstraction layer level. Containers are lightweight and may only be tens of megabytes in size in comparison to the virtual machine which comprises of entire operating system resulting into several gigabytes in size.

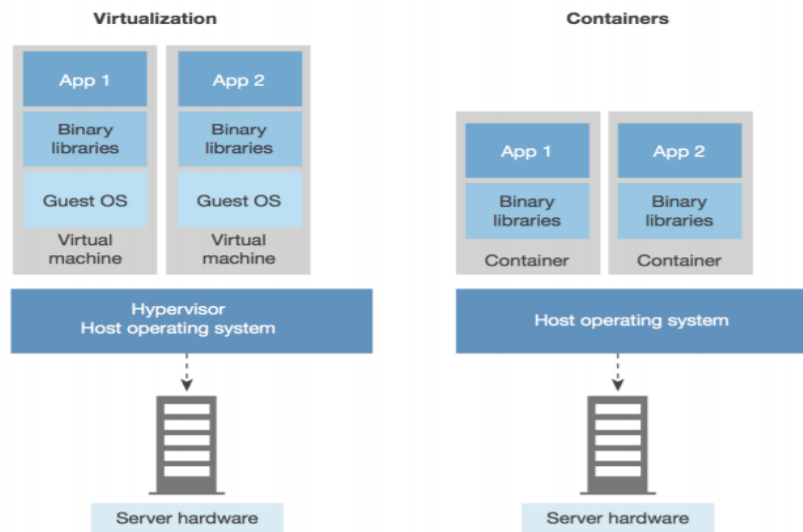


Figure 4-4: Virtualization vs. Containers [32]

Docker containers are a solution to run a piece of a software application in any environment reliably. The developer can code and run the application on a local machine and the same can be done in a test environment, production, virtual machine or in a private or public cloud. There is no need of alteration if the application is deployed as Docker container [32]. Docker containers are smaller, more agile and are faster. They can save on both labor and infrastructure overheads [33].

The Docker concept is very useful to developer and system administrator. Developers can concentrate on the coding and don't have to worry about the environment within which the code will be deployed.

Docker for Rapid Deployment

Technology is changing to make work easy and straightforward. The journey from real hardware to virtual servers changed the way of software application deployment. To make things easier, Docker has been introduced which makes the deployment very easy and saves a lot of time for the developers. Setting up new hardware resources has never been easy, they usually took a couple of days; to increase the efficiency virtual servers were introduced. With virtualization, the time of setup went down from several hours to just a couple of minutes. With Docker, within some seconds everything can be up and running as shown in the Table 4-1.

	Shipping	Deployment (Manual)	Deployment (Automated)	Boot
Real hardware	Days	Hours	Minutes	Minutes
Virtual Machine	Minutes	Minutes	Seconds	Minutes
Container	Seconds	Minutes	Seconds	Seconds

Table 4-1: Deployment Time Comparison

Use of Docker for Software Developers:

- Consistent development environments for the entire team.
- Development Environment is same as the production environment.
- Docker makes build (code-compile) easy.
- Easy configuration: No need to install any language environment on the machine.
- Deployment is easy. The image is simply pushed to any environment, if it's working in the development environment, it will work anywhere.
- Simplified testing

4.2 Design Patterns for Cloud-based Application

Design patterns are considered as a general solution to a commonly occurring problem. In software design, patterns provide a description of solving a problem that can be implemented in various ways as per need. The design patterns suited to the thesis work were analyzed and presented in this section.

4.2.1 Availability Design Patterns

The proportion of time, any system is functional and working correctly is termed as availability. The complex cloud applications face many challenges in terms of availability. The solution to the problem of application downtime is illustrated in this section in the form of design patterns.

4.2.1.1 Health Endpoint Monitoring Pattern

Monitoring is a crucial service when dealing with distributed systems and the cloud. Supervision of an application is a good practice and critical as a business requirement- to monitor the web applications and middle-tier and shared services, and confirms that they are available and working properly. The failure of an application depends on many factors such as network latency, problems in the storage system or network bandwidth, etc. Therefore applications should be verified at regular intervals that they are performing correctly [34].

Implementation:

Send requests to an endpoint on the application which needs to be checked, on receiving the request; the application should perform some necessary checks stated and return a response stating its status shown in figure 4-5.

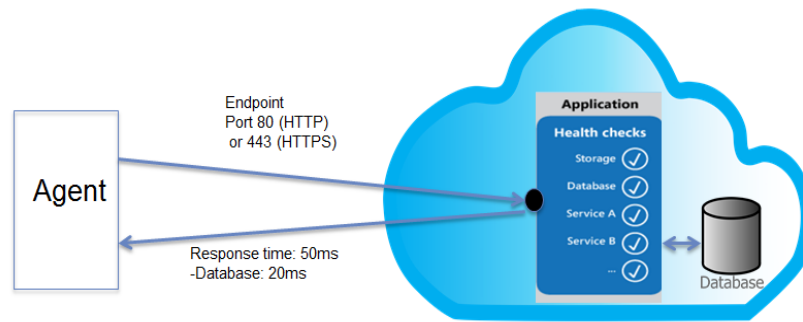


Figure 4-5: Health Endpoint Monitoring Pattern

- Send request to the application endpoint and verify the results.
- Analysis of the results is performed by using some tool or framework.
- Check cloud storage or a database for availability and response time.
- Validate the response, for example, HTTP response 200 (OK) indicates that there is no error. Other key considerations include measuring the response time and checking latency issues, checking resources or service located outside the application, checking SSL certificates expiration or validating the URL returned by the DNS lookup to ensure correct entries.

When to use this pattern:

This pattern is used for:

- Monitoring websites and web applications to check availability.
- For checking correct operations.
- Monitoring middle-tier components and to detect the point of failure to isolate it.

The work of [35] showed there are two types of monitoring: push-based monitoring and polling-based monitoring. The latter is common and used quite often. It involves a set of measuring controllers which send an echo-signal periodically to the hosted applications. The check can be directed to the application through a communication protocol like HTTP in the case of the web application or to the operating system that hosts the applications through network protocols like ICMP or Simple Network Management Protocol [36].

In Push-based monitoring, an application or a monitoring agent is responsible for sending messages to the measuring controller when a meaningful change occurs in the monitored application. Both the approaches can be implemented in a cloud environment. Chan and Chieu [35] used a polling mechanism to check periodically for host failures, whereas in push based monitoring, agents running on the host push notification to the monitoring controller.

4.2.1.2 Queue-Based Load Leveling Pattern

In a cloud environment, if a service is subjected to high loads, it may cause performance or reliability issues. Flooding a service with a huge number of concurrent requests may also

result in complete service failure. The solution is to introduce a message-queue between the task and service. The process where tasks and services run asynchronously, a task posts a message containing the data required by the service to a queue. The queue stores the messages and allows the service to retrieve and process them. This design pattern allows a service to work on its own pace and can get rid of concurrent requests [24].

Implementation

Refactor the solution by implementing a queue between the task and the service. The task and service can now run asynchronously.

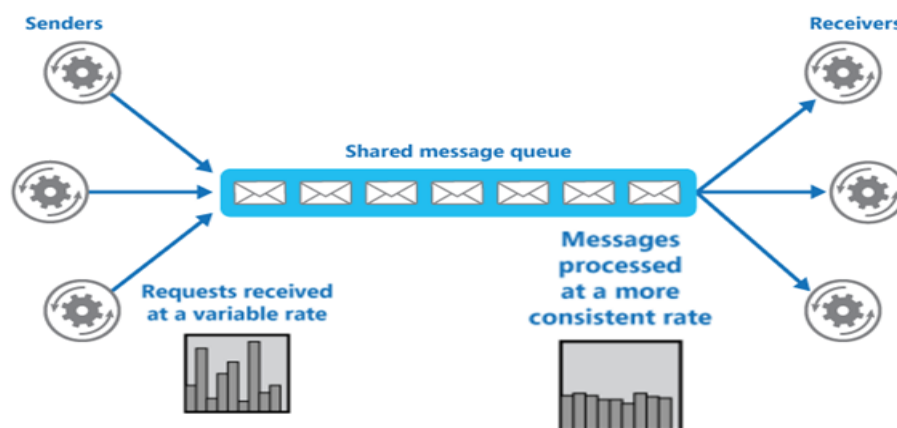


Figure 4-6: Queue-Based Load Leveling Pattern [24]

- The task posts a message containing the data required by the service to a queue. The queue works as a buffer and stores all the messages until the service retrieves it, as shown in figure 4-6.
- The pattern maximizes availability because of message queue storing the messages; the task can continue to post messages even if the service is busy or not available.
- It can also help to maximize scalability because the number of queues and services can be increased or decreased with respect to the traffic.

Example of working

As discussed above this pattern is useful in distributed architecture. For example, in the figure 4-7 many services are accessing the database simultaneously. When many services were interacting with the storage, some failed due to timeout or other reasons and causes problem; the solution is to introduce a message queue which can deal with concurrent requests. The incoming messages can be stored in a message queue, and as per the availability, the messages can be forwarded.

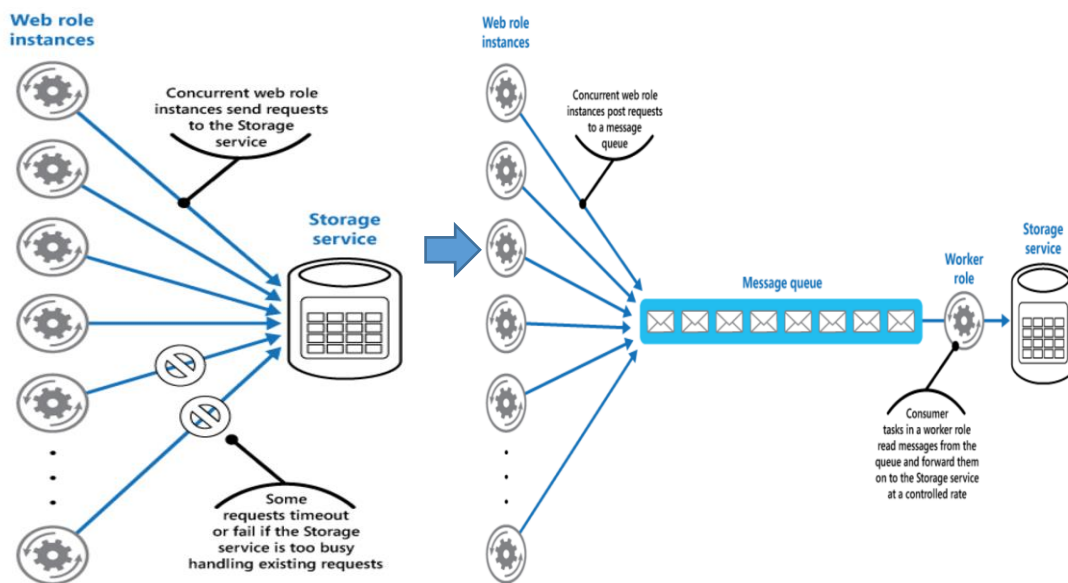


Figure 4-7: Design Pattern Working [24]

4.2.1.3 Throttling Pattern

This pattern allows the system to continue working, even when an increase in demand results in an extreme load on resources. In the case of a sudden heavy load on the resources, if the processing requirements of the system exceed the available resource capacity, the service suffers from poor performance and may even fail completely. One of the common ways to handle this type of problem is auto-scaling; an alternative strategy is to allow applications to use resources only up to some limit, and then throttle them when the limit is reached [34].

Implementation

This pattern allows implementing several throttling strategies such as:

- Disabling or degrading the functionality.
- Rejecting requests from an individual user who has already accessed system APIs more than some limited time.
- Deferring operations which are performed on behalf of lower priority applications or tenants.

Working

Throttling can help to limit the number of requests from each user. For example as shown in the figure 4-8, for a cloud application, the limit of receiving request is 100 requests per second. Now in a multi-tenant system when many users send requests to the application, it checks if the limit is reached or not. On reaching the limit, the application blocks these requests.

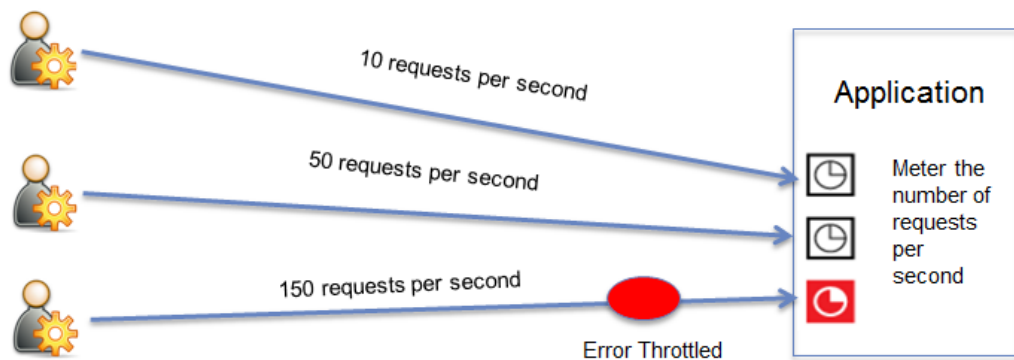


Figure 4-8: Throttling Pattern [34]

4.2.2 Resiliency Design Patterns

The ability of a system to handle and recover gracefully from failure is a meaning of the term resiliency in software design. It is necessary to detect failures soon and isolate them to stop cascading failures. The design patterns for resiliency are discussed in this section.

4.2.2.1 Retry Pattern

The pattern allows an application to handle temporary failures and can improve the stability of the application. The faults are typically self-correcting, the request that triggered a fault when repeated after a certain delay is likely to be successful. As the name suggest, retry pattern suggest to wait and try again, for example, suppose an application invokes an operation on a hosted service, but the request fails, the application should wait for some interval and try again. The pattern is beneficial because of its easy implementation and works well for a small application. However, the pattern faces some issues such as in a highly aggressive retry policy with minimal delay between attempts, a significant number of request retries can further degrade a busy service and can cause cascading failures [34].

Implementation:

This pattern introduces some strategies to handle failure. The strategies are as follows:

- **Cancel:** The application should terminate the operation if the fault indicates that the failure is not transient and occur again.
- **Retry:** if the fault reported is rare or unusual, it may be possible that it has been caused due to unusual circumstances such as a network problem. In this scenario, the application could retry the failing request again immediately.
- **Retry after delay:** It is also a good practice to retry the service after a short break. In case of some network issue, it is always good to give it some time to recover and then try again.

This policy should be used keeping in mind the business requirements of the application and the nature of the failure.

When to use this pattern

In the case of transient faults, this pattern is useful as it interacts with the remote services. This pattern might not be useful in case when a fault is likely to be long lasting. Trying again might waste the resources and time.

4.2.2.2 Leader Election Pattern

As the name suggests, the principle behind it is to coordinate the actions performed by a collection of collaborating instances in a distributed application by electing one instance as the leader that is responsible for managing others [37]. The main benefit is that instances never conflict with each other and cannot interfere with the work that other are performing.

Implementation:

In a cloud-based system scaling is a common practice. In case of multiple instances of the same task running at the same time serving different users, it's likely to face some problem. It is necessary to coordinate their actions and make sure the problem of overwriting doesn't appear.

- A single instance of the task should be elected to act as a leader who is responsible for managing other instances.
- Implement leader election algorithms such as the Bully Algorithm or the Ring Algorithm [38].
- It must be considered that maybe at sometime the leader can fail or has become unavailable. In this case, it is important to detect the failure quickly and appoint a new leader.

When to use this pattern:

This pattern is well suited for a cloud hosted solution where a careful coordination is needed to reach a common goal in a distributed environment.

According to [37] *“Avoid making the leader a bottleneck in the system. The purpose of the leader is to coordinate the work of the subordinate tasks, and it doesn't necessarily have to participate in this work itself—although it should be able to do so if the task isn't elected as the leader.”*

The pattern is not useful if:

- There is already a dedicated process which acts as a leader and is predefined.
- The coordination between tasks can be achieved using other methods which are more lightweight. Example: Use of locking mechanism to control access.

4.2.2.3 Compensating Transaction Pattern

The basic principle behind this pattern is to undo the work performed by a series of steps when one or more of the steps fail. Cloud-based applications modify data frequently, and it is quite common that the data sources are held in different geographic locations. In a distributed

environment, an application should not try to provide strong transactional consistency; rather it should implement eventual consistency. This means that a typical operation consists of a series of separate steps which results into a consistent view of the system [39].

The main challenge is to handle the situation when one of the steps fails. One of the ways is to roll back the steps, but in a distributed system it is likely that the data in the previous steps are already changed.

The solution is to implement a compensating transaction. A common approach is to use a workflow where the system records information about each step, and in the case of failure, the step can be undone. In order to understand the principle of this pattern we can think of a reservation portal to book a flight ticket, where the steps followed by the users are recorded so in case of any error or modification, the user can revert to the previous step and can continue the work instead of following the whole procedure again.

4.2.2.4 Timeouts

Timeouts are used very commonly and are used at lower levels of abstraction. Applications should always set a timeout for remote calls. In a normal application without a timeout, a network failure or a remote system being down could result in the failure of the complete application. Timeouts should be implemented in a correct fashion; otherwise, they can cause further problems. For example waiting too long for a reply, can slow down the whole system. On the other hand, if a timeout happened very quickly, it may ignore a response that would otherwise be received [40].

4.2.2.5 Circuit Breaker pattern

Outside the software world, a circuit breaker is an electrical component which on detecting excess load or usage, opens the circuit to cut out the flow and prevent appliances from being damaged. The use of circuit breaker at the application level is similar to the one in an electric circuit. When a failure occurs, the circuit is opened so that no more requests can be forwarded to the failed part of the application. In the distributed environment, calls to remote resources and services can fail due to transient faults, such as slow network, timeouts, resource unavailability, etc. There can be certain situations where the faults can take a long period of time to get fixed or there can be a situation where the failure of one service can result in a whole system breakdown. In order to prevent cascading failure, the circuit breaker pattern is used. This pattern can prevent cascading failure by preventing an application from repeatedly trying to execute an operation which is likely to fail. Circuit breaker prevents cascading failures as shown in figure 4-9.

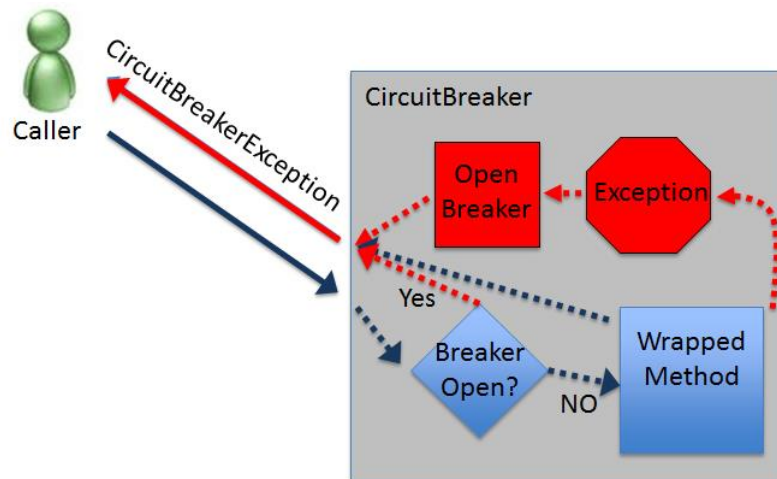


Figure 4-9: Circuit Breaker Working [41]

According to Mike Wasson [41], “The purpose of the Circuit Breaker pattern is different than the Retry pattern. The Retry pattern enables an application to retry an operation in the expectation that it’ll succeed. The Circuit Breaker pattern prevents an application from performing an operation that is likely to fail. An application can combine these two patterns by using the Retry pattern to invoke an operation through a circuit breaker. However, the retry logic should be sensitive to any exceptions returned by the circuit breaker and abandon retry attempts if the circuit breaker indicates that a fault is not transient.”

The circuit breaker behaves like a proxy for operations that might fail. The pattern is more discussed in the upcoming section 4.3.1.

4.2.2.6 Bulkhead

Bulkhead is a common word with context to ships. It is a concept where a ship is divided into separate areas to ensure that a single penetration of the hull does not necessarily sink the ship. With proper partition, a ship can face difficult situations. In the case of flood or some distress, hatches are closed, and bulkheads prevent water from flooding to other parts of the ship so that only a certain section is affected. In software systems, the same concept is used to partition a system and prevent failure in one component from affecting other components which may result in a complete failure of the whole system.

One of the common practices for the implementation of bulkhead pattern is by using separate connection pools for each connection. So if one pool faces some failure, the others can still operate [40].

4.2.2.7 Fail Fast

“Waiting longer for a response is bad but waiting for failure is a waste of time.” In case of failure, applications should be able to fail fast instead of wasting time and resources in waiting for a response. In distributed system, an application should respond to failure quickly,

instead of waiting it should fail fast in order to save resources and maintain capacity while the system is under a heavy load.

4.3 Fault Tolerance Libraries for Resilient Applications

The patterns which are discussed in the previous section 4.2 are implemented by various libraries. In the following section, the fault tolerance library Hystrix along with other useful libraries are introduced.

4.3.1 Hystrix

In a distributed environment, failures are inevitable. To control the interactions between distributed services and preventing them from failure, Hystrix library is designed to provide greater tolerance of latency and failure. Hystrix achieves resiliency by stopping cascading failures and isolating points of access between distributed services. It also provides a fallback option which enhances the overall resiliency of system.

Hystrix library came into existence in 2011 by the Netflix API team working on resilience engineering. Since then the team tried and implemented the library in their own system and saw a dramatic improvement in uptime. Today according to Netflix, *“Tens of billions of thread isolated and hundreds of billions of semaphore-isolated calls are executed via Hystrix everyday at Netflix. [42]”*

In a distributed architecture like microservices discussed in section 4.1.2, services rely on each other to achieve a common goal; in microservices, one service may require using other services as dependencies to get some result for the work. When services rely on each other for work, it might be possible that at a certain point any service may fail and result in the failure of the entire application. If not failed, they can also result in increased latencies between the services, which is, even more, worse than a failure. For example: A user is more keen to get an error when it tries to access a website rather than keep on waiting for the response. On failure, users can be notified and repair work can be started accordingly but if the service is slow, the user keeps on waiting and face unsatisfactory experiences.

Hystrix is an open source library which follows the principles stated below in protecting the system when new failures inevitably occur:

- Stop cascading failures
- Fail fast and recover rapidly
- Fallback and degrade gracefully
- Isolate client network interactions using circuit breaker pattern
- Real-time monitoring, alerting, and operational control

Example: Suppose there is a service running on Tomcat which opens up two connections to two services, in this scenario if one service takes more time than expected to send back the

response, the other service will be waiting which means one thread pool is doing nothing and just waiting for an answer from the slow service. This type of scenario is not acceptable when the amount of traffic is high and may cause saturation of all the services and block the server causing complete failure of the service application. In a connected car where several critical applications are involved, this problem is a serious threat and must be handled [43]. Applications in Distributed Architecture comprise of dozens of dependencies, and there is always a risk of failure in any of these at any time. If the host application is not isolated, these external failures can bring the whole application down.

Scenario: An application depends on 30 services, each service has availability (uptime) of 99.99%.

$$99.99\%^{30} = 99.7\% \text{ uptime}$$

$$0.3\% \text{ of } 1 \text{ billion requests} = 3,000,000 \text{ failures}$$

2 + hours of downtime/month (if all dependencies have an excellent uptime)

In real world scenario, the situation is even worse. Therefore need arises to make the application resilient and thus improves the overall availability of it.

4.3.1.1 Hystrix – How it works

Previous sections explained the distributed architecture and the problems associated with it. Applications have dozens of dependencies, and there is always a risk of failure in any of them.

- In a normal workflow, when everything works correctly, the flow may look like to the diagram shown below in figure 4-10.

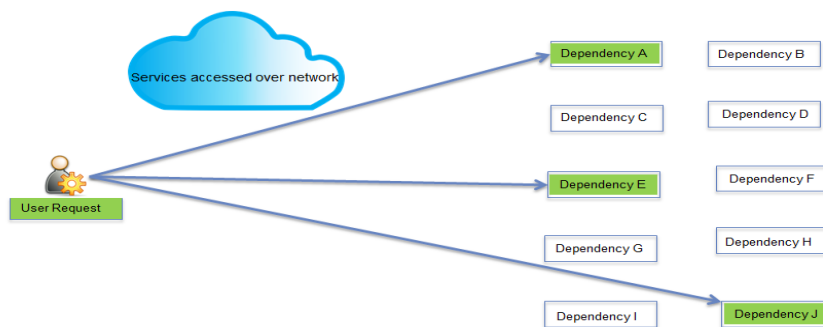


Figure 4-10: Working of Hystrix- Normal Workflow

- Dependency-J failed, causing blockage to an entire user request as shown in figure 4-11. Latency in a single network call can block user request. Latency proved to be a bad experience for the user; the user should be provided with some fallback or degraded result.

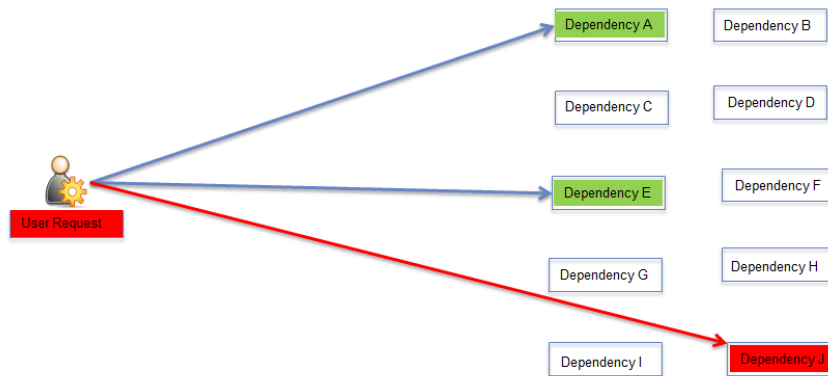


Figure 4-11: Working of Hystrix- Failed Dependency

- A single backend dependency-J becoming latent under heavy traffic volume can cause all other resources to become saturated in a very short time. All the user requests to dependency J becomes latent and results in increased latencies between services, which backs up queues, threads, and other resources ultimately causing cascading failures across the complete system as shown in the figure 4-12.

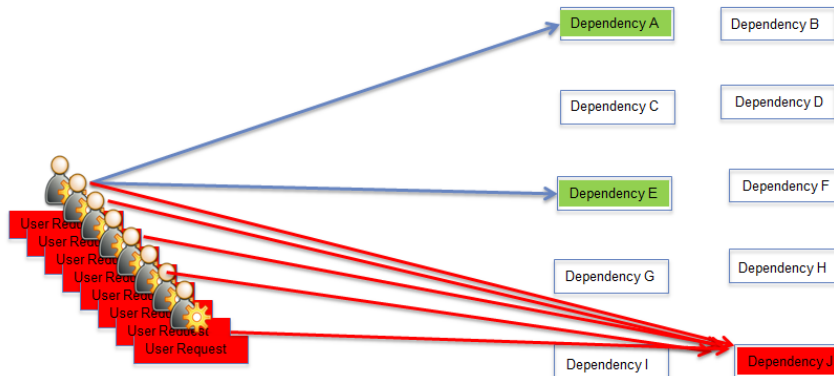


Figure 4-12: Working of Hystrix

Hystrix tries to fulfill the requirement to isolate and manage failures and latency issues so that failure in a single dependency cannot take down the entire application. Hystrix library has the capabilities to stop cascading failure, and it works by following the measures stated below:

- Prevent any single dependency from using all the container user thread.
- Isolate failures by using bulkhead and circuit breaker pattern.
- Optimization for discovering failures by using real-time metrics.

4.3.1.2 Hystrix Functionality & Implementation

Hystrix is a Java based library which wraps all the calls to external systems or dependencies in a HystrixCommand object. The code to be isolated is wrapped inside the run() method of a HystrixCommand.

Hystrix library provides an abstract class called `HystrixCommand` which needs to be extended for each remote service method. The remote call which is also called as a potentially dangerous code needs to be put to the overridden `run()` method; this method is triggered by Hystrix unless and until a circuit breaker is open. Then the command is executed using `execute()` method on its instance.

```
public class ServiceCommand extends HystrixCommand<String>
{
    private static final Service = new Service();
    private final String param;

    public ServiceCommand(String param) {
        super(HystrixCommandGroupKey.Factory.asKey("Service"));
        this.param = param;
    }

    @Override
    protected String run() {
        return service.call(param);
    }
}
```

Code Snippet 1: Hystrix Command

- Construct a `HystrixCommand` or a `HystrixObservableCommand` object. If the dependency is supposed to return a single response, use `HystrixCommand` object and if the dependency is supposed to return an `Observable` that emits responses, use `HystrixObservableCommand` object.
- The object created needs to be executed, which can be done in four ways. Hystrix provides four methods named `execute()`, `queue()`, `observe()`, and `toObservable()`. Out of these four methods, `execute`, and `queue` methods are not available for `HystrixObservableCommand`.
- On executing the command, Hystrix checks if the circuit is open.
- Hystrix reports failures, success, timeouts, and rejections to the circuit breaker, which maintains the statistics.
- On command execution failure, Hystrix reverts to the fallback. On success, it will return the response.

Adopting fault tolerance mechanism using `HystrixCommand` implementation is very flexible and easy process; however, a new command has to be prepared for every single method of

each remote service as well as a new command instance needs to be created for every method call that means it involves writing a lot of code [44].

Hystrix Circuit Breaker

Hystrix implements the circuit breaker pattern. The diagram below shows the interaction of HystrixCommand or HystrixObservableCommand with a HystrixCircuitBreaker and the logic behind it.

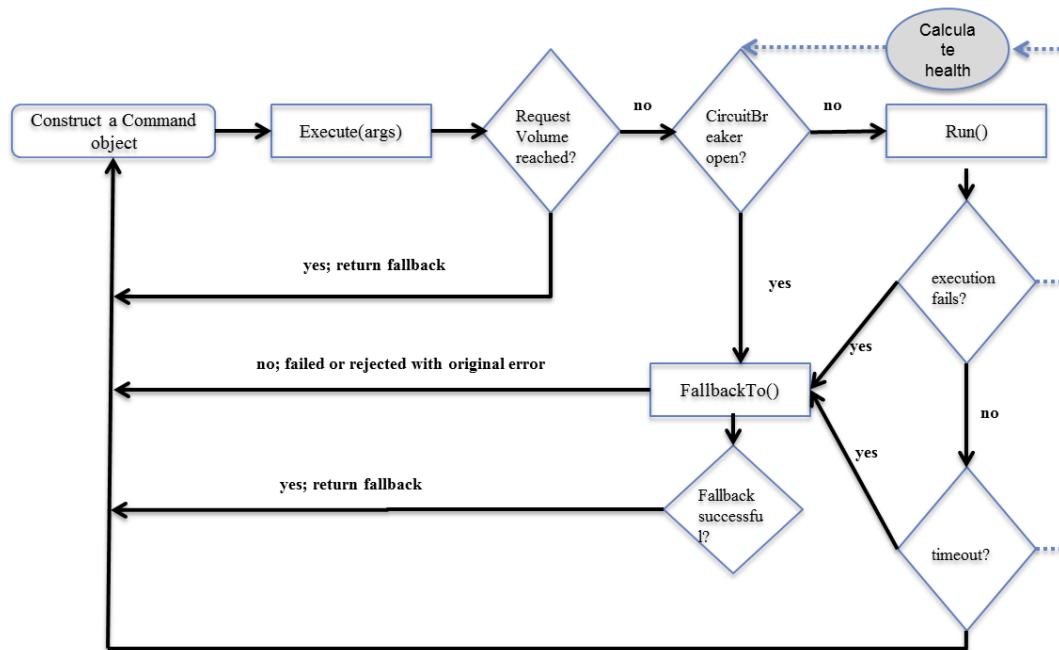


Figure 4-13: Hystrix Circuit Breaker Working

The flowchart in the figure 4-13 shows the detailed working of Hystrix. The circuit breaker is implemented to prevent cascading failure. Circuit breaker performs its operation like:

- On smooth operation, the circuit is closed allowing the request to pass. On meeting certain threshold (`HystrixCommandProperties.circuitBreakerRequestVolumeThreshold()`), or in the case of error percentage exceeds the threshold error percentage (`HystrixCommandProperties.circuitBreakerErrorThresholdPercentage()`), the circuit breaker transitions from closed to open and terminates all the requests made against that circuit breaker.
- After some time it allows a single request to check the current status (`HystrixCommandProperties.circuitBreakerSleepWindowInMilliseconds()`) (circuit half-open). On getting a successful response, the circuit transition from open to closed and allows requests otherwise it remains open.

Hystrix Isolation

Hystrix implements bulkhead pattern to isolate dependencies and to limit the concurrent access to any of them.

Hystrix makes use of separate per-dependency thread pools to achieve the mechanism of isolation. Some of the main reasons for doing so are stated below:

- Each service has a client library which changes over time including the logic behind them.
- Applications usually execute dozens of different backend service calls.
- If the client does not change, sometimes the service itself can change over the time.
- Client side code can also face latency or any types of failures.

Hystrix tackles these problems by using thread pools which have many advantages, some of them are as follows:

- The application is protected from runaway client libraries and can also accept new libraries at lower risk.
- In the case of any issue or failure in client library, it is isolated, and when it becomes healthy, the thread pool will clear up, and the application can resume with a healthy performance.

The isolation provided by thread pools allows s for the changing and dynamic combination of client libraries without causing outages [42].

Hystrix Threads & Thread Pools

Hystrix allows clients to execute on separate threads. Hystrix thread pools are used to constrain any dependency so that latency on the execution can saturate the available threads only in that thread pool and does not cause any failure outside that pool.

Thread pool enhances the functionality of the Semaphore mechanism. With semaphore implementation, concurrent requests can be restricted to any given underlying system.

Example:

Semaphore- Suppose the Tomcat system can handle hundreds of incoming requests. There are many systems underneath which depend on this tomcat system. Semaphore are used so that no one single system occupies the entire tomcat system. For a certain system semaphore is configured allowing ten requests to go through and after the limit of ten, the semaphore will reject any further requests made to the Tomcat system.

Thread Pools- Thread pool is also used to shed load if any backend system is going bad. However, it allows timeouts also which is needed if the client is not trusted.

The client libraries in the distributed systems which are always changing and are dynamic in nature can be handled gracefully using Hystrix threads pools mechanism. The only drawback of using thread pool is the computational overhead, which involves queuing, scheduling and context switching adding up while running a command on a separate thread.

Hystrix Fallback

Hystrix provides a fallback method for graceful degradation. In the case of failure, Hystrix will call the fallback method to obtain a default value and process that. With a simple HystrixCommand fallback can be implemented using `getFallback()` which can be used for any type of failures such as runtime failure, timeout, thread pool or semaphore rejections. It also works for circuit breaker mechanism by Hystrix.

```
public class CommandHelloFailure extends HystrixCommand<String>
{
    private final String name;

    public CommandHelloFailure(String name) {
super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        this.name = name;
    }

    @Override
    protected String run() {
        throw new RuntimeException("this command always fails");
    }

    @Override
    protected String getFallback() {
        return "Hello Failure " + name + "!";
    }
}
```

Code Snippet 2: Hystrix Fallback

Whenever the command `run()` will fail, the caller will be able to receive the value returned by the command `getFallback()` method instead of receiving an exception [42].

4.3.1.3 Hystrix Dashboard and Monitoring

Hystrix Dashboard helps to monitor Hystrix metrics in real time. The dashboard helps to monitor all real time actions taking place in the application and discover operational events. According to the Netflix development team “*When the Netflix development teams started using Hystrix dashboard for monitoring, the operations improved by reducing the time needed to discover and recover from operational events. [45]*”. With the use of Hystrix, the duration of the incidents can be made short by detecting them fast. The impact of the failure

can be minimized significantly by using Hystrix library features that stop cascading failures, and with the dashboard, the real-time system behavior can be observed which gives much useful information, making it easy for the teams to monitor and detect failures.

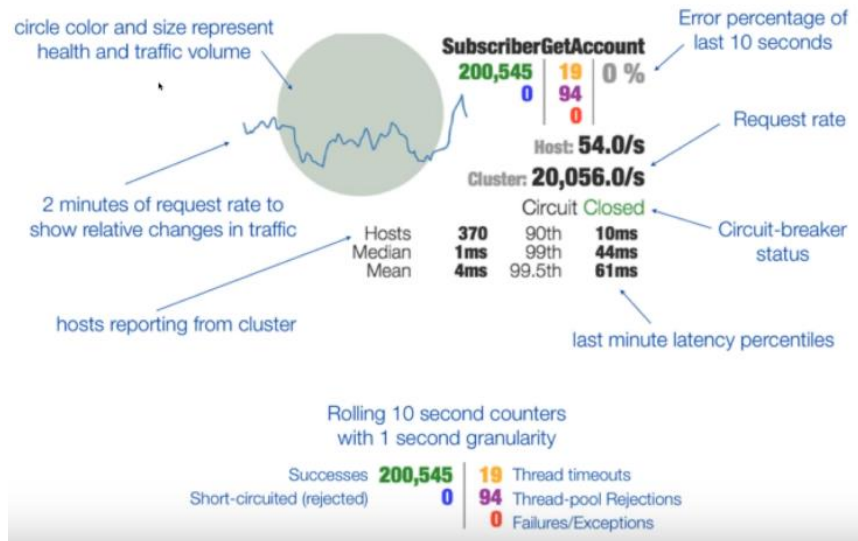


Figure 4-14: Hystrix Dashboard

Hystrix dashboard provides much information as shown in the figure 4-14 which is very useful for developers and operations teams and also provides a graphical representation which is easy to understand and saves time to detect any failure behavior.

Metrics and Monitoring

Implementation of Hystrix generates metrics. These metrics tell about the latency and execution outcomes. One of the benefits is that Hystrix offers metrics per command key and to fine granularities in the order of seconds.

Achieving HA using Hystrix FT library is the focus of the thesis. NTT Data in near future will use this library for the upcoming project to make applications resilient and ensure availability. There are other libraries which show similar characteristics and are open-sourced. Two of them are described in the sections below:

4.3.2 JRugged

JRugged is a library that provides circuit breaker implementation and also monitoring capabilities. It provides straightforward add-ons to existing code to make it more robust and easier to manage [46]. The three most important mechanisms provided by JRugged to make applications robust and fault tolerant are:

- **Initializers:** Provides a way to decouple service construction from initialization and allows the latter to run in the background. For the implementation of this mechanism, the service implements the `Initializable` interface and then passes it to an initializer. The work of the initializer is to keep trying initializing the services in a background thread. This mechanism is very useful in cases where services do not have

all the resources needed for initialization available in the beginning but will eventually get them at some point.

```
public interface Initializable {
    public void tryInit() throws Exception;
    public void afterInit(); }
```

Code Snippet 3: Initializable Interface

- **Circuit breakers:** Circuit breaker can be used to throttle traffic to a failed system by wrapping the service calls by `CircuitBreaker`. They are beneficial particularly in those cases where monitoring is difficult, such as a peer system which can be only accessed over the network. When application operates normally, the `CircuitBreaker` is `CLOSED` and allows calls to pass. When a call fails, `CircuitBreaker` changes its state from `CLOSED` to `OPEN` and does not allow client calls to pass through. As the failed service recovers, the `CircuitBreaker` remains open and after a cool-down period, it switched to a half-closed state where a single call is let through in order to test the service. If the call succeeds, the state of `CircuitBreaker` moves to closed otherwise remains opened.

```
CircuitBreaker circuitBreaker = new CircuitBreaker();
circuitBreaker.invoke (() -> service.call());
```

Code Snippet 4: JRugged Circuit Breaker

- **Performance monitors:** It provides a way to monitor runtime behavior of a service and also provides a way to wrap a service to collect a series of useful statistics such as information about its latency and throughput. `JRugged` provides plugins such as `Hyperic`, used to visualize runtime statistics.

`JRugged` provides a very simple way to implement circuit breaker but does not provide any fallback options. `JRugged` also lacks to provide bulkhead pattern implementation which is required to make an application fault tolerant [46].

4.3.3 Javaslang

It is a lightweight fault tolerant library inspired by `Hystrix` but designed for functional programming. `Javaslang` is an object-functional language extension to Java 8, which aims to reduce lines of code and increase code quality [47]. `Javaslang` provides many mechanisms to make application resilient such as – circuit breaker, retry, rate limiter, fallback, and caching. Implementation of the circuit breaker in `Javaslang` is quite similar to the one in `Hystrix`. Here, there is no need to create commands for different service methods. Instead, `Javaslang` provides an abstract way to implement them [48].

```
CircuitBreakerRegistry circuitBreakerRegistry =
CircuitBreakerRegistry.ofDefaults();
CircuitBreaker circuitBreaker =
```

```

circuitBreakerRegistry.circuitBreaker("Service");

Try.CheckedRunnable checkedRunnable =
CircuitBreaker.decorateCheckedRunnable(() ->
service.call(), circuitBreaker);
Try result=Try.run(checkedRunnable);

```

Code Snippet 5: Javaslant Circuit Breaker

Javaslant CircuitBreaker is a more lightweight library having fewer dependencies, in comparison with Hystrix. Mechanisms like rate limiter and retry are new and are not provided by Netflix Hystrix Library. However, it does not provide bulkheads or requests collapsing.

Comparison between Hystrix, JRugged and Javaslant:

Mechanisms Available	Hystrix	JRugged	Javaslant
Bulkhead	✓	✗	✗
Circuit Breaker	✓	✓	✓
Fallback	✓	✗	✓
Lightweight	✗	✗	✓
Rate Limiter	✓	✗	✓
Retry	✓	✗	✓
Monitoring tool	✓	✓	✗

Table 4-2: Comparison of FT libraries

The comparison shows Hystrix and Javaslant has most of the features to support FT. JRugged lacks to provide fallback and bulkhead mechanism which are essential to make application resilient. Javaslant is designed only for functional programming and therefore Hystrix will be used for the work in this thesis. The support and help sources are enough for the Hystrix as compared to other FT libraries. In this thesis work Hystrix will be used and focus will be given to implement it to make resilient applications.

4.4 Resilient Platforms for Microservices

The previous section 4.3 described and compared some popular libraries which implement circuit breaker pattern, which is used to make applications resilient and fault tolerant resulting in high availability. The platforms which offer the functionality for high availability are discussed in this section.

4.4.1 Spring Boot

Fault tolerance mechanisms such as circuit breaker are added to Spring Boot applications using Spring Cloud Netflix project [27] that provides various Netflix OSS feature

integrations, including Hystrix. It uses Hystrix annotations to configure Hystrix, and there is no need to create many implementations of `HystrixCommand` class. The benefit of using the platform is that it is not necessary to do any additional configuration to implement Hystrix, just an annotation named `@EnableCircuitBreaker` on a class that serves as an application entry point. This class now implements Hystrix mechanism and can be monitored. In the same way `@HystrixCommand` is applied to provide `fallbackMethod`.

4.4.2 Wildfly Swarm

The platform integrates fault tolerance mechanism where Hystrix components work with Ribbon to provide the circuit breaker functionality. It allows enabling fallback option by satisfying the requests even if the remote services are unavailable. Wildfly swarm implements Hystrix and therefore is treated as a platform which provides fault tolerance capabilities to make applications highly available [49].

4.4.3 Vert.x

The platform provides the circuit breaker pattern implementation making it a fault tolerant platform. It monitors the number of failures and tracks their numbers to see if the number of failures reaches the threshold. Once the threshold is reached, it opens the circuit and does not allow the request to hit the failed service. Implementation of the circuit breaker pattern is performed by adding a dependency and a circuit breaker is created with the required configuration [50].

```
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-circuit-breaker</artifactId>
  <version>3.4.3</version>
</dependency>
```

Code Snippet 6: Dependency for Circuit Breaker

4.5 Testing

In software engineering, testing is an essential part which is described as a process of checking the implemented solution to match its initial requirements. As the applications get more complex involving a large number of platforms and methodologies, it is more important than ever to find a robust methodology for making sure every aspect of the application is fully tested to meet their specified requirements and can successfully operate in the targeted environment.

During the thesis work, the work included learning and finding of various methodologies which help to achieve high availability. As already discussed throughout the previous sections, testing distributed applications is hard, and therefore a new approach is needed for

testing. In this section, a new framework for testing distributed system named Chaos Monkey is analyzed.

4.5.1 Testing Distributed Systems

As described in Section 3.3, cloud-based applications are generally based on distributed system architecture. The components within distributed systems interact with each other to reach a common goal. This increases the complexity and makes testing difficult. Considering a non-distributed system system being tested under the best of circumstances, and no matter how deep the system is tested, the bug can still get through. Now consider testing in distributed system, multiply the complexity of testing taking into consideration different platforms running multiple processes written in multiple languages and running on various operating systems.

Testing distributed systems are hard; some of the reasons for it are as follows:

- Massive changing data sets
- Web-scale traffic
- Complex Interactions and Information flows
- Asynchronous requests on massive scale
- Third party service involvements
- Changing environment
- Failures happen in an unpredictable way

4.5.2 Chaos Monkey

The previous section gave an overview of the complexity in distributed system and how hard it is to test them. To assure availability, the system is tested and tested more. Before the explanation of Chaos Monkey testing, the basic types of testing mechanism are stated below. Every application is gone through a set of unit tests followed by integration tests where units are combined and tested. Stress testing and load testing ensures the reliability as well as availability of the application.

- Unit Testing
- Integration Testing
- Stress Testing
- Exhaustive test suites to simulate and test all failure mode

However, for most large-scale systems it is almost impossible to simulate and test all the failure modes. To overcome the challenges, Netflix [51] introduces an another way of testing which suggests some mechanisms listed below:

- Cause failure to validate resiliency
- Test the design assumptions by stressing

- Instead of waiting for random failures, insert them periodically

Chaos Monkey testing, a methodology coined by the Netflix team proposed a mechanism of forcing failures to the production system to introduce errors in random components in a controlled fashion.

Failures are inevitable and occur all the time, especially in a distributed architecture. In a production environment, if the service fails in the night or during the holidays, it is very hard for the engineers to correct it. Distributed system failures are unpredictable, and therefore engineers should be prepared for any failure beforehand. In complex and dynamic architectures, the simple fix is not sufficient, and it may cause undesired consequences. There are many situations in the real world where it is hard to avoid failures, and so the chaos monkey methodology is used to get prepared for them.

In distributed system testing, it is hard to answer these questions:

- Are load balancers working correctly in case of failures?
- How easy is it to rebuild the failed system?
- Does a simple fix or quick patch worked reliably and sufficiently?

The phenomenon of chaos monkey tries to answer these questions. The testing team can test the system without waiting for it to fail, instead it tries to cause it to fail. Chaos monkey helps the team to get prepared for handling new errors and make their system resilient to gain high availability.

Case Study of practicing Chaos Monkey methodologies at Netflix Amazon Web Services to demonstrate the usefulness of Chaos Monkey in the real-world scenario.

In the initial days of implementing Chaos Monkey, the Netflix team faced some problem with their Stack Exchange. According to the Netflix engineers working on the problem, “Every few days, one of the servers in the web farm would simply stop responding to all external network requests” [52]. The team spent months chasing the problem caused by Chaos Monkey where some servers were shutting down randomly. According to the testing report, the team took different steps to resolve the issue, some of which were as follows:

- Swapping network ports
- Using a different switch
- OS and driver level network setting
- Replacing network cables
- Kernel Hotfixes
- Taking high-level vendor support

The team after trying so many measures and steps to cure the problem realized the positive side of this activity. They identified the loopholes in their system infrastructure and also came to know the ability of their engineers to face random failures. They took steps such as:

- Where there was only one server performing an essential function, they switched to two.
- Created fallbacks where they were missing.
- Removing dependencies, and minimizing where required.

The basic principle of Chaos Monkey


It is a service whose job is to kill instances and services within the production environment randomly. The process helps to constantly test the ability and resiliency of the system to succeed despite facing failures, and get prepared in the event of an unexpected outage or failure. This service operates at a controlled time, i.e. the service kills another service in operational time and not on weekends or during holidays. The reason behind this is to prepare engineers to respond and correct the failure during business hours.

Monitoring of Chaos Monkey Events:

- REST: Netflix came up with a simple REST interface that allows querying about the termination events. Chaos Monkey causes random failure, so it is important to keep track on its events, so if anything goes wrong this API is used to get notifications and information of the terminations.
- SERVO: It is an application monitoring solution which keeps track on what is happening to the application at runtime.
- Inspired by the usefulness of Chaos Monkey, Netflix introduced Simian Army which includes many more services based on the philosophy of introducing failures to the system.

4.5.3 Simian Army by Netflix

Netflix created the Simian Army shown in Table 4-2, which consists of a series of tools and services known as monkeys that deliberately inject failure into the production environment where services and systems are working [53].

Official logo of the Service (Pictures Source:Netflix OSS)	Name of the Service	Impact	Analysis
	Simian Army	Introduce failures and interrupt the normal functioning of the system.	System behavior, resiliency, and availability.




	Chaos Monkey	Instance, dependency or service failure.	Make services stateless, run services on clusters, manage latent services, make application resilient, isolate failures & implement fallback.
	Chaos Gorilla	Lots of instances fail. Impact on the availability zone (shuts down).	Large scale events are hard to simulate, smooth recovery is a challenge, automatic scalability.
	Latency Monkey	Slow down the services, introduce latency and errors.	Stop cascading failures, introduce resiliency by configuring circuit breaker and fallbacks.

Table 4-3: Simian Army [54]

Conclusion:

This chapter summarized the analysis done on different mechanisms including design patterns, FT libraries, and Testing, discussed its findings and contributions. Before going to the next chapter, below is the overview of analysis results.

Service Architecture: Monolithic and Microservice architecture were analyzed and microservices are used to gain HA. Microservices are used in most of the applications on which NTT Data is working, however the reason for using were described.

Design Patterns: To gain HA at application level, different cloud based design patterns were studied and explained which gives different type of approaches in the direction of HA. The design patterns named Circuit Breaker, Bulkhead and Fail Fast were analyzed more closely and they can be used for further implementation.

Fault Tolerant Library: The thesis work focusses mainly on the use of Hystrix Library which was analyzed along with all its features. The library is implemented and results out to be very useful for making applications resilient which helps applications to gain high availability. It was analyzed that the library can be implemented to new Java based application which is a requirement of NTT Data upcoming projects and also possible to integrate it into the existing applications. The features were analyzed which were used for the concept and implementation. While studying about the Hystrix library two other FT libraries were discussed which works somehow similarly but lacks many features found only in Hystrix library. This proves that Hystrix is the most advance library till now and can be used to gain HA. The section also discussed about the resilient platforms for microservices and found out that the discussed thee platforms can be used along with FT libraries to make applicationn highly resilient.

Testing: The need for testing mechanisms for distributed systems were discussed and a new mechanism of Chaos Monkey testing was analyzed. Chaos Monkey was analyzed to figure

out how the mechanism can be used for the applications hosted on OpenShift which is used by NTT Data. It was observed that Chaos Monkey methodology can be implemented along with Hystrix integrated applications. The technologies and methodologies were gathered, studied and analyzed so that they can be used together to make a highly available application. The observations from this chapter further outlined the direction for the design and concept part which will be considered in the next chapter.

5 Concept and Design

This chapter describes the main concept of the thesis and designs a solution for the implementation by integrating fault tolerance mechanisms within an application. The goal is to use fault-tolerant library Hystrix within an application to make it resilient. The concept is made to show how Hystrix works when something fails in the application. The concept is made using a scenario where one of the service goes down or is very slow and is not responding and thereby causing problem for the whole application. The problem is then handled by using Hystix which is described in this section. The concept is created for the teams at NTT Data to understand how Hystrix can help in this scenario. The concept is made so that it can work as base for the implementation part where the same scenario will be implemented and Hystrix effect can be observed. The chapter starts describing the main concept by explaining the problem first and then pointing out some important requirements which need to be fulfilled, to make the application resilient. To fulfill these requirements, the design of the solution is proposed, and then the architecture is explained which then can be tested to achieve high availability.

The Figures 5-1 and 5-2 illustrates the problem scenario.

- The client makes a request to the research web to perform some task.
- The research web relies on many dependencies on getting the content requested by the client.

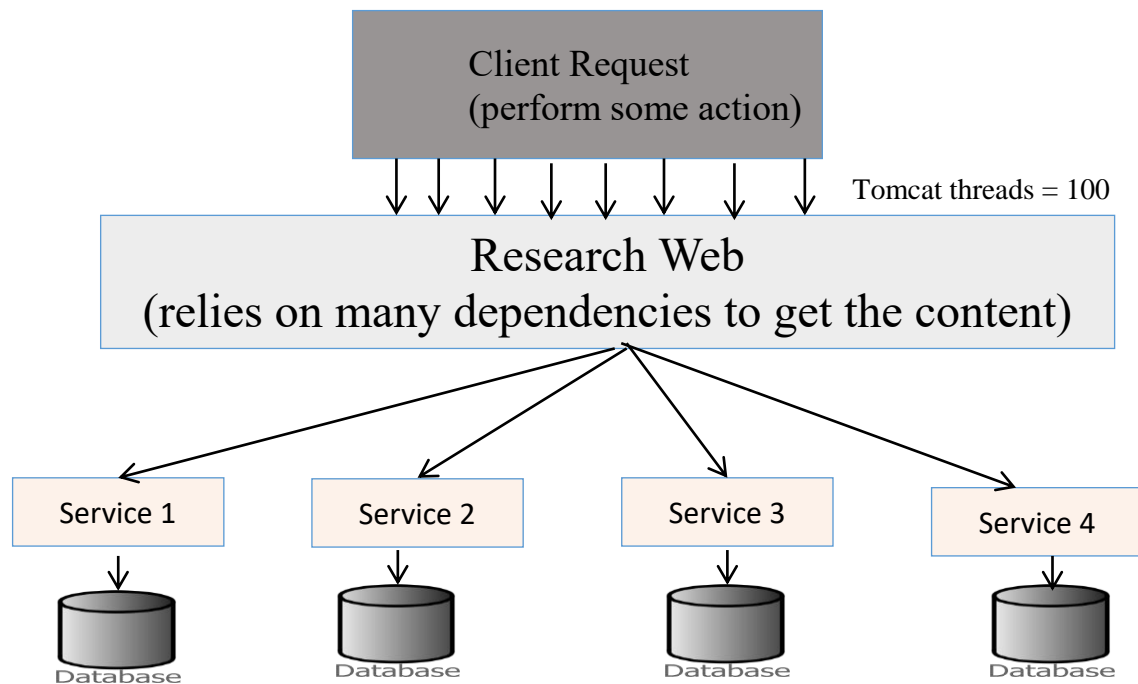


Figure 5-1: Concept - Normal Execution

- One of the services (Service 3), when called by the research web, does not respond.

- The requests keep on hitting the failed or slow service without giving it time to recover, and all the threads of the application are used up waiting for the response from the service.
- As a result, the user keeps on waiting.
- All the threads are used up by the faulty service resulting in complete application failure as shown in Figure 5-2.

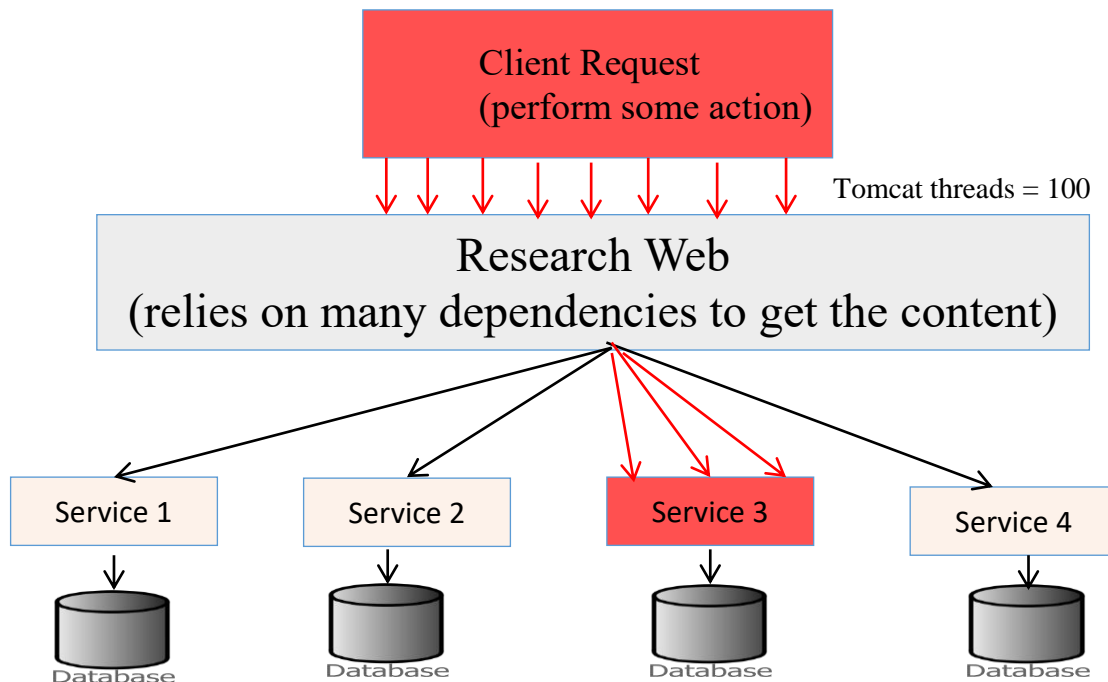


Figure 5-2: Concept- (Problem Scenario)

Implementation of timeouts:

- Now, with the implementation of timeouts, the client does not have to wait for long.
- With the implementation of timeouts, the client gets the notification (fallback).
- But on the server side, the request keeps on hitting the slow or failed service.
- If the issue persists long enough, the same issue will arise as discussed in the previous scenarios where all the threads are used up.
- This can result in complete failure of the application.

From the above-mentioned scenario, it is clear that failure in one service can result in cascading failure. Timeouts do not solve the problem completely, and therefore it needs a solution, and therefore the next step is to figure out the requirements which should be considered when proposing a required solution.

5.1 Requirements

As discussed in the section 4.1.2, microservice architecture gives a direction to achieve high availability but to enable these services to communicate safely with the remote services, fault tolerant mechanism is used which can handle any type of error or failure. The main

requirements to make a service resilient resulting into high availability are summarized below:

Circuit Breaker & Fallback

Circuit breaker mechanism can be used directly in the code where a remote service is called. A seamless integration of circuit breaker pattern is required which is easily implemented by the developer and add a little overhead to service method calls. The solution should not increase the complexity by adding unnecessary dependencies to the application.

Less code- Less Complexity

The application should use predefined FT library. It should be possible to use annotations to configure and implement fault tolerance mechanisms. Using annotations, developers can easily configure useful mechanisms without changing any business logic of the application design.

Monitoring

Failures will happen for sure, so to detect and correct them, real-time monitoring is required. On the application level, the solution is needed which can monitor different services running on a single application so that any failure to service can be detected quickly.

Testing

Testing of distributed systems is difficult so a mechanism is required which can work in parallel to the traditional testing methodologies. Instead of waiting for a failure to happen, introduce errors into the system and be prepared with a solution.

5.2 Design of solution

Fault tolerance library Selection:

There are several libraries analyzed in section 4.3, and it was observed that Hystrix provides much more functionality than other libraries. Therefore Hystrix library is used to make applications resilient and achieve high availability.

- Set thread pool limits for each dependency, if a dependency requests for more than what is allocated, Hystrix will break the circuit (State of the circuit changes from Closed to Open).
- Implement circuit breaker and fallback logic.
- Isolate point of access to services and stop cascading failures across them.
- Improve overall resiliency of the application.

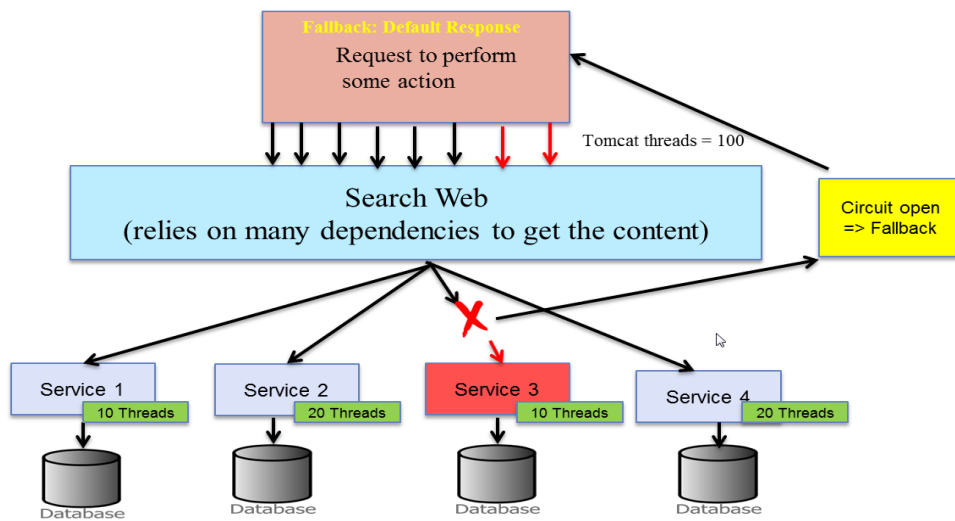


Figure 5-3: Design- Use of Hystrix

- Service 3 is down as shown in figure 5-3, and Hystrix on observing the failure changes its circuit status from closed to open (threadpool limit reached, or error frequency threshold reached).
- No more requests are sent to the service 3, giving it time to recover.
- Circuit breaker allows a request to hit the service again after some time to check the current status of the service.
- If the request is served, and the response is received, the status of the circuit changes from open to closed and allow the normal functioning of the system.
- If the service is still down and no response is received, circuit keeps its status as Open and fallback method is called.

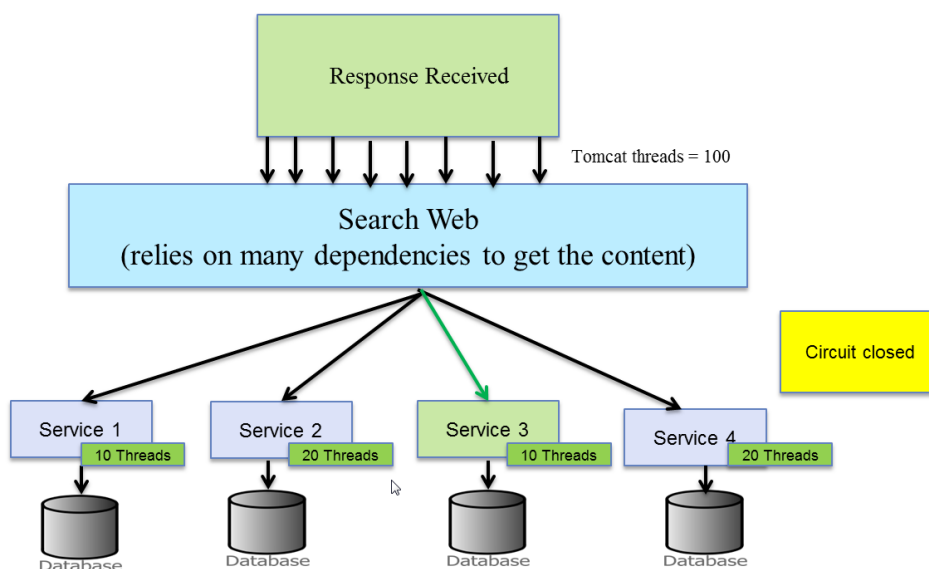


Figure 5-4: Design- Problem solved with Hystrix

The design of the proposed solution makes use of Hystrix library features as shown in figure 5-4, which fulfill the requirements and prepare an application for failures. The solution also proposes a fault injection testing mechanism to test the resiliency of the application.

5.3 Architecture

After gathering the requirements in the previous section and proposing a concept, the architecture design as shown in figure 5-5 is proposed in this chapter. Microservice architecture is more useful with context to availability when compared to a monolithic approach with regards to high availability. The library which fulfills the requirement and implements design pattern needed for scalability, resiliency and high availability is Hystrix which is widely used and fulfills the required mechanisms.

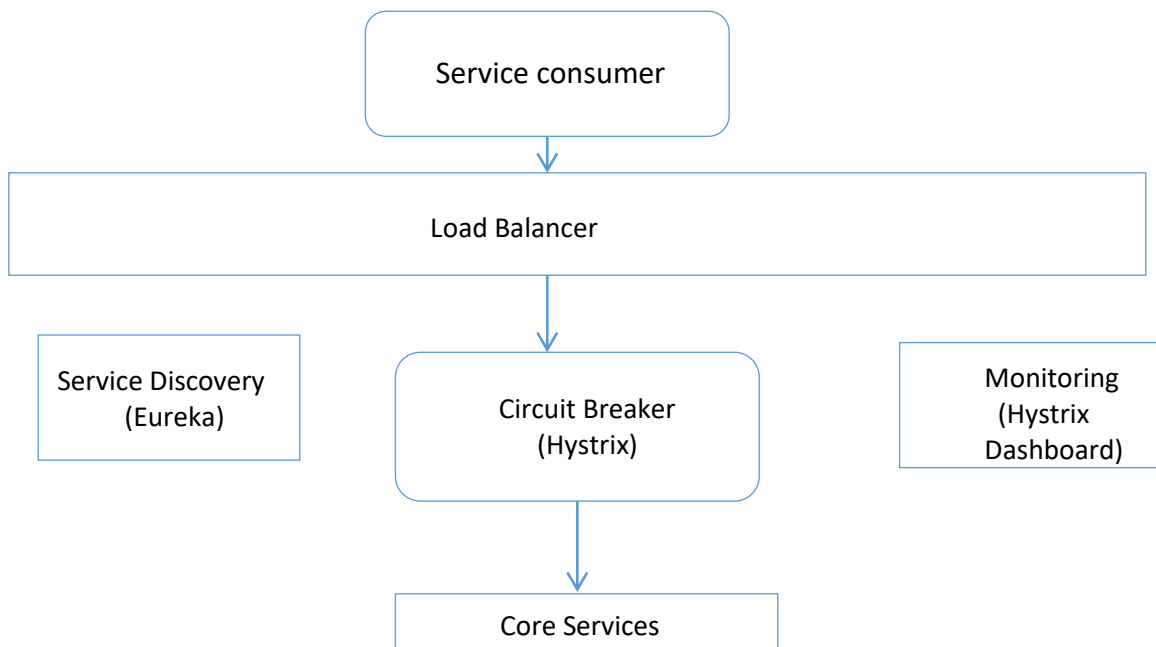


Figure 5-5: Architecture design at the application level

The architecture for the application is designed keeping in mind the goal to achieve high availability and also finding an open source solution as per the requirement of the company's project scope. The application architecture is discussed below:

- The application follows microservice architecture which is already discussed in the previous section 4.1.2.
- Ribbon library is used as a load-balancer. It also provides features for fault tolerance, caching and batching. Adding ribbon is easy by simply adding its dependency.
- Eureka Service Discovery is a REST based service which is also used as a load balancer. Cloud-based applications consist of different services that communicate with each other, and the location of these services may change with time. Service discovery allows the communication between the various services by keeping track of each

service location. The service which needs to communicate can look for the address of another service in a service discovery and can interact with it. Eureka also manages the services and could be helpful in case of any modifications.

- Monitoring is essential when working with microservices. Hystrix provides an integrated monitoring tool known as Hystrix dashboard which gives a graphical visualization of the real-time state of services.
- The application is hosted on cloud PaaS (discussed in the section 3.5.2) solution.
- Load and stress test is performed to check the availability of the application. Chaos Monkey methodology is recommended to test the application running on OpenShift PaaS cloud platform.

Conclusion:

The concept is to introduce Hystrix library to the application and test it using Chaos Monkey methodologies. The application structure is inspired by the projects at NTT Data. NTT Data needs a concept which shows that how microservice architecture along with fault tolerant libraries can improve the availability for a cloud based application. The library Hystrix is never used in any projects till now at NTT Data and therefore a general concept was made to show how the mechanisms discussed in the analyses chapter 4 can work together.

The testing methodology inspired by Chaos Monkey is also introduced to the concept, the goal is to make the application resilient and then test it using Chaos Monkey methodology to test its availability. The concept shows how Hystrix can be used and gives an architecture for the application design.

The concept throws light on the application level, the design of application which can lead to high availability. The concept is made by grouping together mechanism like Hystrix and Chaos Monkey using microservice architecture to test the advantage of it. NTT Data needs solution which uses open sourced tools and therefore the tools and libraries were used which were open sourced by Netflix OSS.

6 Implementation

This chapter describes the phase of implementation. It is outlined in the form of use cases to differentiate the implementation approach. The first use case describes the scenario where a normal application is running, and results were taken. In the second case, timeouts are used which helps the application to react on slow or failed services but does not serve the functionality needed to achieve high availability by making the application resilient in a distributed system. In the third case, Fault tolerant library Hystrix is implemented, and results were compared with the other two cases, which shows the advantages of Hystrix and fulfill the requirements mentioned in section 5.1.

The scenario of the application:

For the demonstration that how Hystrix works and how it makes an application resilient, a Java application is developed and used as a demo application for the implementation part. The Java-based application is composed of two parts where two services interact with each other. For better understanding, the two services were designed in such a way that one of them acts as a client sending requests and the second service behaves as a server processing these requests.

Service 2 which behaves as a server and has a REST endpoint with two resources and consumes the API requests made by the client. The client will launch 20 threads (user defined) at the slow endpoint, and the effect is recorded.

The application is tested under three scenarios which are discussed below. The behavior of the application is demonstrated when one resource is taking too much time.

6.1 Normal Execution

The first scenario shows the effect of latent or failed service on the whole application. The normal execution is on the application having no FT mechanism. The table 6-1 and 6-2 gives a clear picture of the goal and procedure of the execution.

Outline	How the application fails when a service or a resource is slow or unavailable.
Summary	Two services in the application interact with each other, one of the services will face some error and gets slow. The impact of the failure is analyzed.
Procedure	<ul style="list-style-type: none">• The application has two typical APIs, in which one of them

	<p>consist of sleep function (to show what happens when a service is slow) and the other API when called displays current time and status.</p> <ul style="list-style-type: none"> • The service 1 which behaves as a client will send the request to the other service which behaves as a server and will launch a number of threads (for demonstration number of threads = 20). The requests will hit the API which is slow (sleep = 30 seconds). • The other service 2 which behaves as a server will receive the requests and wait for 30 seconds to respond (slow endpoint). • The behavior of the application is demonstrated for this scenario.
Preconditions	<ul style="list-style-type: none"> • Services should be up and running. • Exception handling.
Output	<p>The client creates a number of threads (20 threads) and hits the slow endpoint (30 seconds sleep). All threads are lined up and wait for the slow service to respond. When a huge number of requests tries to access the slow API, the server side will be loaded with a huge amount of threads making the whole application slow and resulting into cascading failure. Due to heavy load on the server side, the whole application stops working.</p> <ul style="list-style-type: none"> • The client keeps on waiting for the slow service to respond. • Server side became slow and loaded with a huge number of open threads. • The overall application becomes unavailable.
Functional Description	<p>In the application, one slow or failed service may lead to cascading failure and affect the whole application.</p>
Open Issues	<p>Stop sending a request to the slow service using some mechanism such as timeouts.</p>

Table 6-1: Implementation - Normal Execution

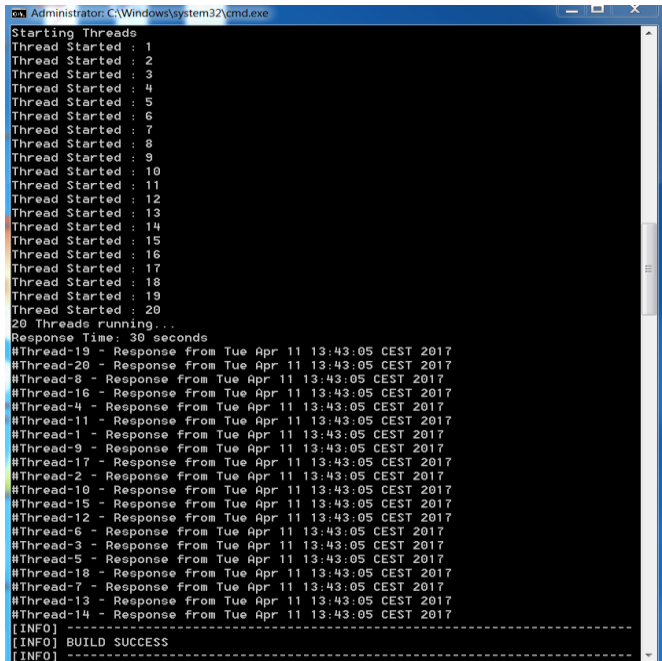
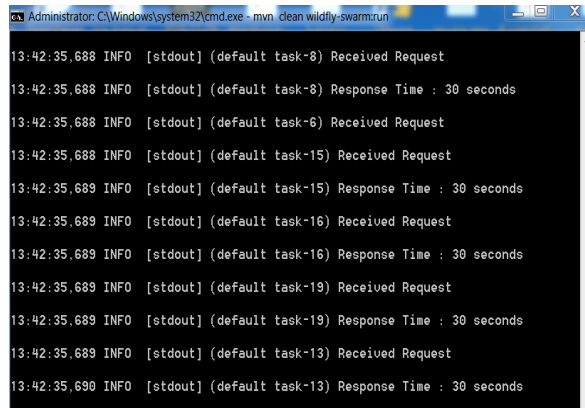

Service 1 (Client)	Service 2 (Server)
	
<p>The client started threads and waits for the slow service to respond. After 30 seconds client received the responses.</p> <p>The client keeps on waiting if the endpoint service has failed (for the demonstration in the implementation sleep is implemented).</p>	<p>Threads opened at the server side waiting for the slow endpoint to respond.</p>  <p>The server processed the request after 30 seconds.</p>

Table 6-2: Screenshot- Normal Execution

6.2 Timeout Execution

In this scenario, timeout function is implemented. When a request goes to a service which is slow or failed, the timeout function waits for the response for a defined time. If the response is not received the timeout function will be called and the no more requests will be forwarded to the failed service. The scenario is described in the Table 6-3 and 6-4.

Outline	Use of timeout when a service is slow or unavailable.
Summary	Two services in the application interact with each other, one of the services will face some error and gets slow. In this case, timeout is introduced. The client will wait for a response only up to a defined time. Timeout is beneficial because the client will be notified and doesn't have to wait for the response from a slow service.

Procedure	<ul style="list-style-type: none"> • The application has two API's, in which one of them consists of a sleep function (to show what happens when a service is slow) and the other API displays current time and status when it is called. • The service 1 which behaves as a client will send the request to the other service which behaves as a server and will launch a number of threads (for demonstration number of threads = 20). The requests will hit the API which is slow (sleep = 30 seconds). • The other service 2 which behaves as a server will receive the requests and wait for 30 seconds to respond (slow endpoint). • Configure a timeout; for example, the request should wait for one second for the response. • With no response within one second, throw an exception.
Preconditions	<ul style="list-style-type: none"> • Services should be up and running. • Exception handling. • Timeout function
Output	<p>The client creates a number of threads defined and hits the slow API (30 seconds sleep). The service is slow, and no response is given back; therefore, after waiting for a second, timeout function is called, and notification is sent to the client. Client (service: sending requests) gets the notification and does not have to wait. Timeout helps the client to get the notification, and there is no need to wait, but at the server side (service: receiving requests), the threads are still waiting for the slow service to respond. The server when loaded with many requests becomes slow as so many threads are queued and finally stops working (the endpoint API which displays current time does not respond and is blocked).</p>
Functional Description	<p>Timeout does not make the application fully resilient.</p>
Open Issues	<p>The challenge is to make an application resilient is still open.</p>

Table 6-3: Implementation - Use of Timeouts

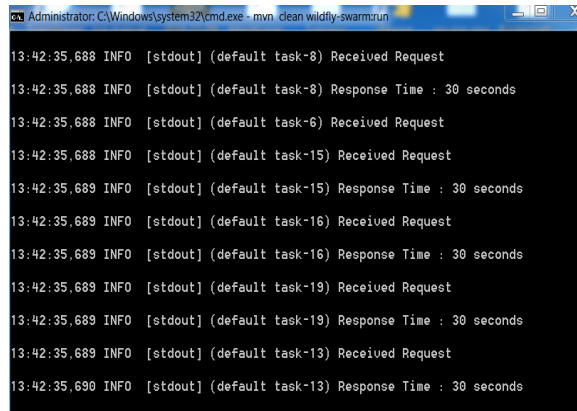
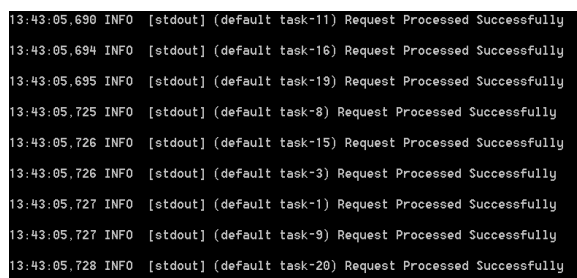
Service 1 (Client)	Service 2 (Server)
<pre>Starting Threads 20 Threads running... Timeout: 1 second #Thread-11 - Response message (Fallback) #Thread-2 - Response message (Fallback) #Thread-10 - Response message (Fallback) #Thread-6 - Response message (Fallback) #Thread-3 - Response message (Fallback) #Thread-7 - Response message (Fallback) #Thread-14 - Response message (Fallback) #Thread-13 - Response message (Fallback) #Thread-4 - Response message (Fallback) #Thread-9 - Response message (Fallback) #Thread-5 - Response message (Fallback) #Thread-1 - Response message (Fallback) #Thread-8 - Response message (Fallback) #Thread-12 - Response message (Fallback) #Thread-15 - Response message (Fallback) #Thread-16 - Response message (Fallback) #Thread-17 - Response message (Fallback) #Thread-18 - Response message (Fallback) #Thread-19 - Response message (Fallback) #Thread-20 - Response message (Fallback)</pre>	
<p>The client started threads and waits for the slow service to respond. A timeout of a second is implemented, on receiving no response from the slow endpoint client receives fallback.</p>	<p>Threads opened at the server side waiting for the slow endpoint to respond.</p>  <p>The server processed the request after 30 seconds. The client received the fallback after 1 second, but the server is queued with requests till 30 seconds. During 30 seconds when several more requests were hit to the slow API opening a lot of threads, the application becomes slow and then was unavailable for a while.</p>

Table 6-4: Screenshot- Timeout Implementation

6.3 Hystrix Execution

In this scenario, Hystrix is introduced in the application. The circuit breaker and fallback are implemented which helps to achieve the goal and make application resilient. The effect of Hystrix is described in the table 6-5 and 6-6.

Outline	How Hystrix helps when a service is slow or unavailable and how it can stop cascading failures.
Summary	Two services in the application interact with each other, one of the

	services will face some error and will get slow. The impact of the failure is analyzed.
Procedure	<ul style="list-style-type: none"> • The application has two API's, in which one of them consist of sleep function (to show what happens when a service is slow) and the other API displays current time and status when it is called. • The service 1 which behaves as a client will send the request to the other service which behaves as a server and will launch a number of threads (for demonstration number of threads = 20). The requests will hit the API which is slow (sleep = 30 seconds). • The other service 2 which behaves as a server will receive the requests and wait for 30 seconds to respond (slow endpoint). • Implement Hystrix and use its fault tolerant mechanisms to solve the problem.
Preconditions	<ul style="list-style-type: none"> • Services should be up and running. • Exception handling. • Hystrix Implementation.
Output	The requests made by the client hit the slow API and Hystrix on observing that there is no response from the endpoint, opens the circuit and does not allow more requests to hit the slow or failed service. The circuit breaker will function in this case and observe the failed service and isolate the failure by opening the circuit and thus stops cascading failure. Fallback is configured at the client, and at the server side only a few requests passed, and the circuit is opened which does not allow any more request to hit the slow service endpoint, giving it time to recover.
Functional Description	Hystrix circuit breaker stops cascading failure. The circuit breaker, fallback, and fail-fast features make the experience better for the client and server services. The mechanism allows the slow or failed endpoint to take time and recover. In time to time, Hystrix checks the status of the failed service and closes the circuit when the service is up again.
Open Issues	Use of annotations to implement Hystrix.

Table 6-5: Implementation - Use of Hystrix

Service 1 (Client)

```
#Thread-11 - Response message (Fallback)
#Thread-2 - Response message (Fallback)
#Thread-10 - Response message (Fallback)
#Thread-6 - Response message (Fallback)
#Thread-3 - Response message (Fallback)
#Thread-7 - Response message (Fallback)
#Thread-14 - Response message (Fallback)
#Thread-13 - Response message (Fallback)
#Thread-4 - Response message (Fallback)
#Thread-9 - Response message (Fallback)
#Thread-5 - Response message (Fallback)
#Thread-1 - Response message (Fallback)
#Thread-8 - Response message (Fallback)
#Thread-12 - Response message (Fallback)
#Thread-15 - Response message (Fallback)
#Thread-16 - Response message (Fallback)
#Thread-17 - Response message (Fallback)
#Thread-18 - Response message (Fallback)
#Thread-19 - Response message (Fallback)
#Thread-20 - Response message (Fallback)
```

The client got the immediate fallback because the server endpoint is slow and taking more time than expected.

- Immediate fallback is provided; the client doesn't have to wait for the response.

Service 2 (Server)

```
Administrator: C:\Windows\system32\cmd.exe - mvn clean wildfly-swarmrun
15:31:00,314 INFO [stdout] (default task-4) Response Time : 30 seconds
15:31:00,314 INFO [stdout] (default task-9) Received Request
15:31:00,315 INFO [stdout] (default task-9) Response Time : 30 seconds
15:31:00,315 INFO [stdout] (default task-2) Received Request
15:31:00,315 INFO [stdout] (default task-2) Response Time : 30 seconds
15:31:00,316 INFO [stdout] (default task-6) Received Request
15:31:00,316 INFO [stdout] (default task-6) Response Time : 30 seconds
15:31:00,316 INFO [stdout] (default task-1) Received Request
15:31:00,316 INFO [stdout] (default task-1) Response Time : 30 seconds
15:31:00,318 INFO [stdout] (default task-3) Received Request
15:31:00,318 INFO [stdout] (default task-7) Received Request
15:31:00,319 INFO [stdout] (default task-7) Response Time : 30 seconds
15:31:00,319 INFO [stdout] (default task-3) Response Time : 30 seconds
15:31:00,319 INFO [stdout] (default task-5) Received Request
15:31:00,320 INFO [stdout] (default task-5) Response Time : 30 seconds
15:31:30,316 INFO [stdout] (default task-8) Request Processed Successfully
15:31:30,318 INFO [stdout] (default task-9) Request Processed Successfully
15:31:30,320 INFO [stdout] (default task-1) Request Processed Successfully
15:31:30,322 INFO [stdout] (default task-2) Request Processed Successfully
15:31:30,325 INFO [stdout] (default task-7) Request Processed Successfully
```

A very little number of requests are queued up because Hystrix opens the circuit breaker by observing that the endpoint is not responding. After a while when the service recovered, the requests were processed and the circuit is closed to receive more requests.

- Circuit breaker cuts the connection and does not allow requests to hit the fail or slow endpoint.
- The failure at one slow endpoint is isolated.

Table 6-6: Screenshot- Hystrix Implementation

The response time of the application is recorded. In normal and timeout execution scenario, the load is applied to the server side. The server was queued up with the requests, and without giving it time to process those requests, more number of requests are sent to the server side. The application is called again and again by refreshing it, and it was observed that the response was slow, and after a while it became unavailable.

The table 6-7 shows the reaction time of the application without Hystrix implementation. The application response time, when called for the first time is recorded. Now when one of the service becomes slow and the server side overhead increased, the application is called again and the time is recorded. It was observed that the time taken by the application to respond was 01:72:540. The time will be compared with the Hystrix enabled application to see the effect of Hystrix mechanism.

Response time on first request: 16:09:32:247
Response time on second request: 16:11:04:787

To demonstrate the effect of the slow endpoint on the overall application, the application was refreshed and the time for the response is recorded. It was observed that when the load at server side increased the overall application became slow and with too much of load, the application became unavailable.

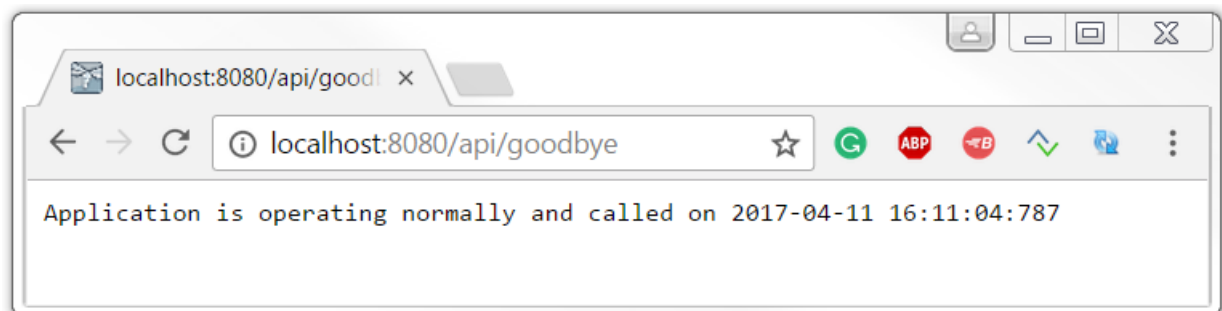
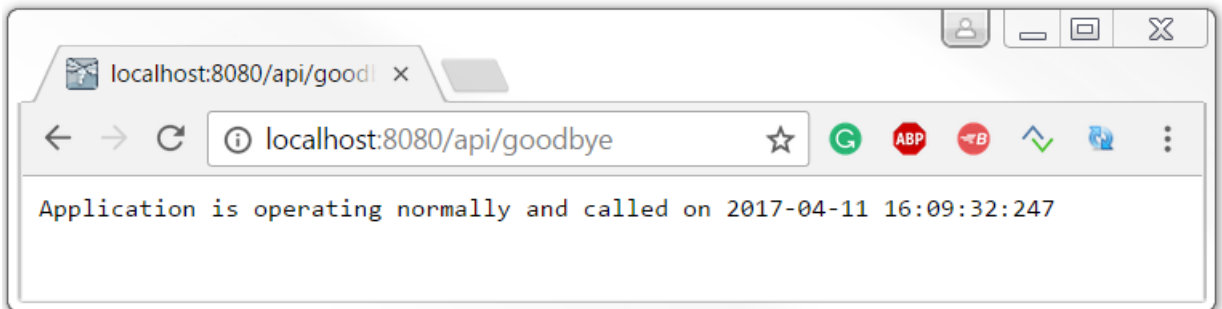
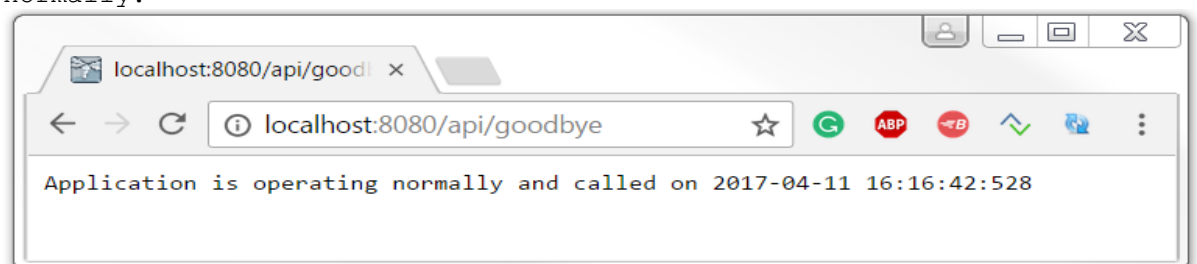


Table 6-7: Screenshot to show availability (Timeout)

In the case of Hystrix implementation scenario, the circuit opens up and doesn't allow the requests to queue up at the server side, and therefore the application responded quickly. It was observed that in spite of one slow endpoint, Hystrix makes it possible for an application to run making it resilient.

Response time on first request: 16:16:42:528
Response time on second request: 16:16:43:763
In spite of slow/failed service, the application is running and replying normally.



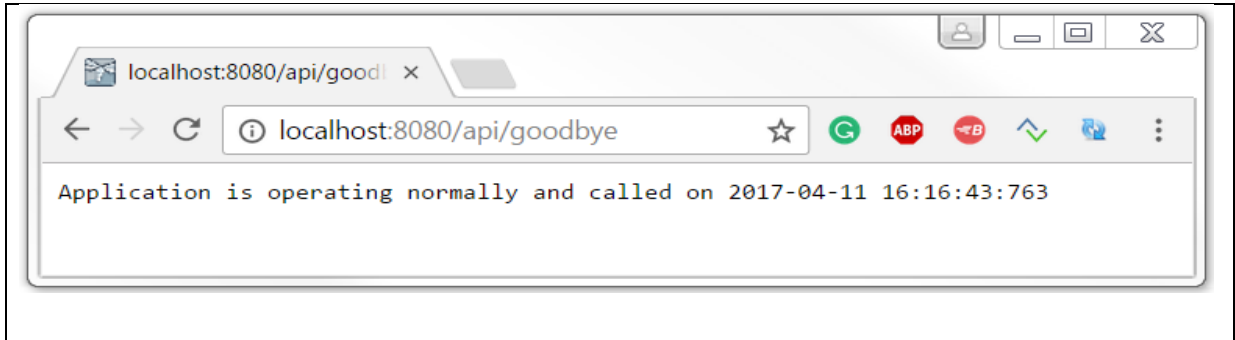


Table 6-8: Screenshot to show availability (Hystrix)

Here the application is called and time is recorded shown in the table 6-8. Since the server overhead is low because of circuit breaker implementation therefore the response time is very minimal. The response time of 01:235 (1second and 235milisecond) cannot be detected by the user and the user does not experience any anomaly.

The three scenarios discussed above clearly shows the benefits of Hystrix, and how the availability of an application can be improved by making an application resilient and fault tolerant. It was seen that Hystrix does not allow the requests to hit the slow or failed endpoint, however it proved a fallback to the user. It gives time to the failed endpoint to recover and works again normally. The time recorded shows that Hystrix allows the service to recover and the application responded quickly when compared with the application with timeout. In timeout enable application the overhead on the failed endpoint is higher and no time is given to the endpoint to recover and therefore the application responded after a long time. This implementation shows that how Hystrix can make an application resilient and improves its overall availability.

6.4 Monitoring with Hystrix Dashboard

This section describes the implementation of Hystrix using microservice architecture. The Hystrix implementation can be done in two ways: with direct Hystrix command based approach and with the use of annotations. The easiest way is to annotate the remote call with `@Hystrixcommand` annotation with the basic configuration to fallback in case a remote call failed using “fallBackCall.”

The table 6-9 below shows the actual components used for the implementation of the design concept described in section 5.3.

Operational Components	Netflix, Spring
Service Discovery server	NetflixEureka
Dynamic Routing and Load Balancer	Netflix Ribbon
Circuit Breaker	Netflix Hystrix
Monitoring	Netflix Hystrix Dashboard
Central Configuration server	Spring Cloud Config Server

Table 6-9: Components used for the implementation

- **Infrastructure:** Netflix Eureka is a service discovery server which allows the microservices to register themselves at runtime. To enable the server `@EnableEurekaServer` annotation is used. To implement the Eureka Server the first step is to create a new Maven project and add the dependencies to it. For this project `spring-cloud-starter-parent` was imported. After adding the dependencies the main class for the application is defined as shown below and finally the application port is configured using `application.yml` configuration file:

```
// Annotation for standard Spring Boot application
@SpringBootApplication
//Eureka Annotation
@EnableEurekaServer
public class EurekaApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaApplication.class,
args);
    }
}
```

Code Snippet 8: Annotations use for spring cloud applications

- Microservices will be auto-registered with Eureka by adding a `@EnableDiscoveryClient` annotation to the application.
- Below is the screenshot of the Eureka server. The first screenshot in figure 6-1 shows the standard interface and the second screenshot in figure 6-2 shows the state when services are registered at the runtime.

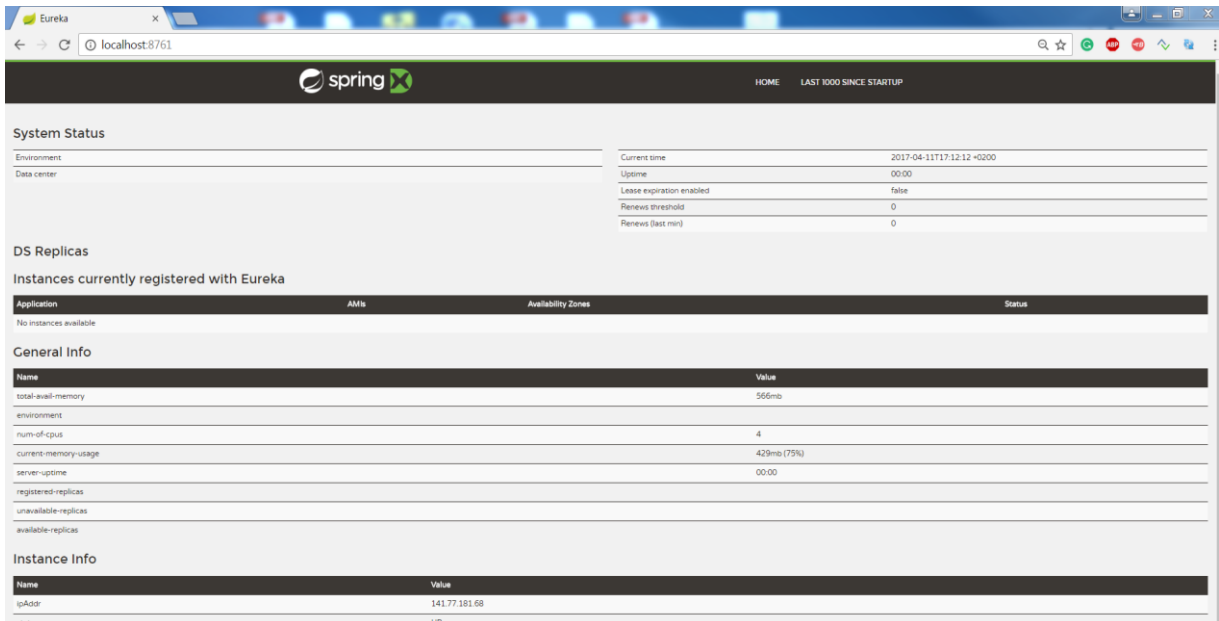


Figure 6-1: Eureka server when microservices are not running

Application	AMIs	Availability Zones	Status
SAMPLE-HYSTRIX-AGGREGATE	n/a (1)	(1)	UP (1) - PC46338
SAMPLE-HYSTRIX-CONFIG	n/a (1)	(1)	UP (1) - PC46338
SAMPLE-HYSTRIX-GATEWAY	n/a (1)	(1)	UP (1) - PC46338
SAMPLE-HYSTRIX-MONITOR	n/a (1)	(1)	UP (1) - PC46338
SAMPLE-HYSTRIX-SERVICE	n/a (1)	(1)	UP (1) - PC46338

Figure 6-2: Registration of Services on runtime

- **Hystrix:** It is enabled using the `@EnableCircuitBreaker` annotation with the spring boot application. Hystrix monitors the method which is annotated by `@HystrixCommand`.
- **Fallback:** In the case of any error, a fallback method is used. The fallback method is called in case a timeout occurs, a call to service fails, or the circuit is open.
- **Hystrix Dashboard:** Once all the services are working correctly, the functionality of Hystrix circuit breaker is observed. When one of the services fails, the circuit breaker detects a problem and opens the circuit. The caller of the service gets a fallback. On increasing the error frequency over the limits, Hystrix opens the circuit to fail fast and thus stops cascading failure. Figure 6-3 is a screenshot taken from the Hystrix dashboard. It shows that how the circuit breaker is working in the application. On the left the circuit is closed for the service but the service named `FallbackCommand` failed and therefore its circuit is opened.



Figure 6-3: Screenshot from Hystrix Dashboard

- The color of the circuit changes to depict a circuit is failing. It changes color from green to yellow to red. The transition of the circuit from closed to open is shown in the figure 6-4. For a single service the transition is shown to clarify the working of circuit breaker. The service is running with no error and thus the circuit is closed with green color. At certain point the error percentage is increased to 34% and the color is changed to show that it's not functioning normally and need attention. The error percentage is still lower than the threshold assigned and therefore the circuit is still closed and the service is still running.

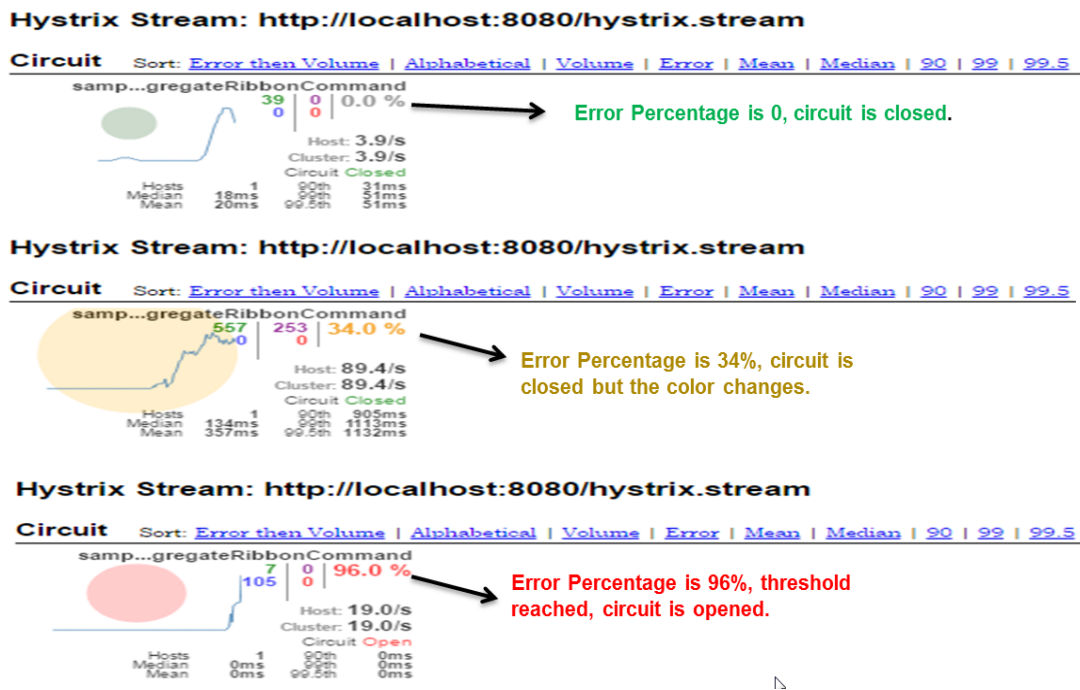


Figure 6-4: Screenshot of Circuit Breaker in action

Hystrix can be also embedded into existing Java based applications. By using one of the existing Spring application, this was tested. The solution found to integrate Hystrix into existing Spring application is with the help of Spring Aspect Oriented Programming (AOP). The aspect oriented programming complements Object Oriented Programming (OOP) but as the key unit of OOP is class, in AOP it is Aspects. The aspect is a module which provides cross-cutting requirements [55]. The interface or gateway where Hystrix needs to be introduced can be made as protected and monitored by using `@CircuitBreaker` annotation. An Advice is created to be executed at a pointcut annotated with `@CircuitBreaker`. The thesis gives more emphasis on implementing Hystrix for NTT Data upcoming projects and therefore this way of integration was only tested to get an idea and to know is it possible to integrate Hystrix into existing applications. This was tested with the help of already existing projects which is open sourced [56].

Hystrix library can be used as a circuit breaker to efficiently handle cascading failures. A failing microservice can cause a full system outage due to propagating errors. Hystrix isolates the failure, fail fast and provide a fallback.

6.5 Testing

Distributed systems are complex and does not behave as expected. As the system grows, it suffers from different dependability issues, and the presence of a fault at any component level can result in the failure of the entire system. In distributed systems, dependability is an

important issue especially when the service is a critical safety system; faults can propagate and impact other components resulting in complete failure. Furthermore, availability of services is a critical aspect at the moment, especially when talking about connected car services and applications [57]. In order to validate and test these types of services, fault injection method has been widely used. As the name suggest, in this approach faults are introduced to the distributed system environment to validate the availability and resiliency of the application. Fault injection can be performed at two levels, the first is at the hardware level, and the other one is at the software level.

In this section, the test methodology adopted for testing Distributed Systems is described. The application is tested using traditional methods, and then the fault injection mechanism is adopted and tested based on Chaos Monkey testing methodology.

The traditional way of Testing (Application level):

On the application level, the unit testing and integration testing has been performed followed by the performance tests. Unit tests help to verify that a single component works as expected and when the components interact with each other, integration testing helps to verify that the interaction between them is successful and the components behave as expected. The application was then tested under heavy load to discover the problems application can face. Two scenarios have been created to differentiate between the behavior of an application under test; one application does have fault tolerance mechanisms (Hystrix enabled) and the other does not. The test plan execution along with the results is shown in the figure 6-5. The test execution is planned keeping in mind the goal to achieve resiliency leading to HA. The tests were performed manually and results were recorded.

Test Number	Test Scenario	Description	Pre-condition	Expected Results	Actual Result	Status
1	Without FT	Client side started 100 threads to the slow/failed endpoint.	Timeout of 5 seconds configured	Client notified when timeout is triggered, the overall application still runs.	Client notified after 5 seconds, Application is still running but 100 threads are still queued up at the slow endpoint side.	PASS
2	Without FT	Client requests (refreshed) 5 times opening 500 threads.	Timeout of 5 seconds configured	Client notified when timeout is triggered, the overall application still runs.	Client notified after 5 seconds, 500 threads queued up at the slow endpoint giving no time to recover: Application went down.	FAIL
3	Hystrix Enabled	Client requests (refreshed) 5 times opening 500 threads.	Hystrix configured	Client notified when timeout is triggered, the overall application still runs.	Circuit breaker triggered to open, observing slow/ failed endpoint, requests to the endpoint are stopped giving it time to recover. Application still runs and client got the default response(fallback)	PASS

Figure 6-5: Test Execution Plan 1

Chaos Monkey Testing:

In distributed large scale applications, services are being updated and deployed hundreds of times a day. Microservice-based architectures running on cloud technology necessitates a new approach for testing the resiliency of these applications, especially in a production

environment. The methodologies of Chaos Monkey were adopted and tested. The steps are described below:

The scenario of the Test Experiment:

The java based demo application has been deployed to Openshift (cloud: platform as a service), the Openshift platform is used in NTT Data but due to security reasons the application was not tested on the companies Openshift environment. The Openshift was installed on the local computer using virtual machine. The application can be deployed to Openshift using Github. The application used for testing is deployed using the environment for Java based applications (OpenShift provides environment for Php, NodeJs,etc.) and chaos monkey service has been introduced to the environment. Chaos Monkey service introduced random failure which was observed, and the behavior of the overall application is noted down. Chaos Monkey is a methodology which gives an idea on how to test distributed systems by introducing failures in it. The service developed by Netflix team is open sourced but can only run with Amazon cloud environment. For the thesis work an open source service is used which works on the principle of Chaos Monkey. The service [58] is analyzed and approved to be used at NTT Data for the OpenShift platform. The service can be deployed on the Openshift just like a normal application and on starting the service, it starts to kill other services randomly.

The experiments were repeated, and after each experiment which is described below, the impact was noted down, and the discovered flaws were fixed. The execution of test plans is shown in figure 6-6.

Test Number	Test Scenario	Description	Observations & Flaws	Measure Taken	Status
1	Normal application without FT running on openshift.	Chaos monkey service kills the DB dependency.	The database dependency was called constantly even when it was down, giving no time to recover.	Circuit breaker is implemented in the application.	Fixed
2	Normal application with Hystrix circuit breaker implementation	The dependency was killed manually to check the fix (database shutdown)	Circuit breaker worked and no more request passed to failed dependency, No fallback.	Hystrix fallback method is implemented, default response when circuit is open	Fixed
3	Hystrix Enabled	Performance test was done, application tested under heavy load	On reaching the defined threshold, circuit opens.	Openshift provides scalability feature, 2 instances are created to serve heavy load	Fixed
4	Hystrix Enabled	Chaos monkey kills random service.	Continous testing is required, chaos monkey creates random failures everytime it runs.	At application level, hystrix provides resiliency	more testing

Figure 6-6: Test Execution Plan 2

In the world of distributed system development, any change or update in a component can cause a chain of errors affecting the whole system. Chaos Monkey testing prepares for these types of failures. In the implementation, the resiliency of an application is tested by

introducing errors and observing whether the application is able to survive it. The testing was done, and the loopholes were fixed, it was noted that at every step of testing new bugs were found and fixed which made the application more resilient and moreover highly available.

Chaos Testing Results:

The Chaos Monkey Test were runned around 10 times in each turn and only the errors corresponding to the application is recorded. Sometimes the errors were raised for scalability and load balancers which were neglected as the main focus was to test the effect of Hystrix on the application. It was recorded that each time the sequence of failure is different but the errors were same. The environment used for the experiment was not so complex therefore the errors were same. On scaling the services, the errors differs but the type of error remains the same. As the service was open-sourced the behaviour was predefined to kill the services randomly. When the Chaos Monkey service deployed alone on the Openshift, on scaling it started killing each other.

Chaos Monkey methodology allowed to automate the testing process. Injecting failures to the system identifies the weak spot of the architecture and improvements can be done easily. Detecting unexpected faults and discovering the bottlenecks of the software architecture saves a lot of time and also money for the testing teams. The testing teams now don't have to manually plan for the testing of possible failures instead they just have to monitor, detect and improve the architecture.

The results which were taken can be summarized as shown in the table 6-10:

Chaos Monkey Error	Current State of the Application	Effect on the Application	Availability (low or High)
Service killed	Without Hystrix	Application downtime	50%
Service killed	Scaling done	Application on high load stopped working	70% (Scaling increase cost)
Service killed	Hystrix Enabled	Application runs(falllback)	80%
Service killed	Scaling of Hystrix Enabled Service	Aplication runs	90%

Table 6-10: Test Results

Availability is measure in terms of percentage. For the thesis work availability is measure in these categories. The errors associated to application design were only taken.

- 50% : Chaos Monkey runs 10 times and 5 times the application went down.
- 70%: Chaos Monkey runs 10 times and 3 times the application went down.
- 80%: Chaos Monkey runs 10 times and 2 times the application went down.
- 90%: Chaos Monkey runs 10 times and 1 time the application went down.

It was observed that resiliency increases overall availability of an application. Hystrix enabled applications are resilient and face less downtime when compared with the normal application. The Chaos Monkey testing helps to find errors early instead of waiting for them to happen. As already discussed distributed systems are complex which makes testing very difficult. It takes a lot of time and resources to plan the tests and estimate that what can go wrong. Errors are unpredictable and can happen any time so instead of waiting for them to happen, Chaos Monkey caused errors so that they can be fixed. In the thesis work a web application is hosted on the cloud, keeping in mind that the application is functioning as expected. The unit tests were performed and there was no problem to run it. On running the Chaos Monkey testing, it was observed that scaling needs to be done and timeout should be embedded to the application. On performing more tests, the need of circuit breaker arised and therefore Hystrix was introduced. The Chaos Monkey testing helped to identify the errors and they were corrected which was very beneficial both for th developers and the testing teams. Testing teams just have to run it, identify the cause of error and fix them. It saves a lot of resources and helps to figure out errors in the early stage. To use this mechanism for the production environment proper planning should be done, i.e. the test should be carried out in controlled fashion. The errors generted should be examined properly and should be fixed quickly so that it does not affect the whole system negatively. At the application level the errors should be fixed in a proper manner so that same error does not exist again. The Chaos Monkey testing is not an alternative to traditional testing, it is mainly used to find errors for the distributed systems running on the cloud. The next chapter gives a conclusion of the thesis and describes the observation made from the thesis work.

7 Conclusion

This chapter consolidates the work and evaluates the research findings. The observations gathered during the work is illustrated in this section along with the benefits of the findings. In the end, an outlook on future research will be presented which were observed during the work.

7.1 Observations

The research done on the topic clearly gives an idea that how cloud technology has transformed the landscape of businesses and will continue to do so. In the connected car market, key applications must be accessible at all times and meeting these higher demands for availability require not only a strategy that accounts for enterprise infrastructures but also focusing on designing and delivering highly available applications.

It has been observed that resilient applications improve the overall availability of a system. The figure 7-1 shows the graph of availability vs. complexity and cost. The graph is drawn by gathering data from the company's experience and the experience of development and architecture teams. The availability increases without increasing the cost at the application level. With the introduction of microservices-based Distributed System architecture, the availability increases, but on the same time, the complexity of the application also increases. Some of the factors which contribute to the complexity are as follows:

- Developers deal with the additional complexity of the microservice architecture.
- Creation and handling of distributed systems is difficult.
- Testing is difficult in distributed system which involved inter-service communication mechanisms.
- The increase in deployment and operational complexity.
- Increased memory consumption and overhead are higher as each service runs on its own.

The complexity is decreased by using some mechanisms and technologies discussed throughout the thesis work. Some of these mechanisms are listed below:

- The solution proposed in the thesis for making an application highly available includes the use of a fault tolerant library named Hystrix which decreases the complexity & cost and helps in creating fault tolerant and resilient distributed systems.
- Chaos Monkey methodology makes testing robust and easy in distributed systems.
- Docker makes deployment and development of microservices easy.

Microservices developed along with the proposed mechanisms develops a highly available application.

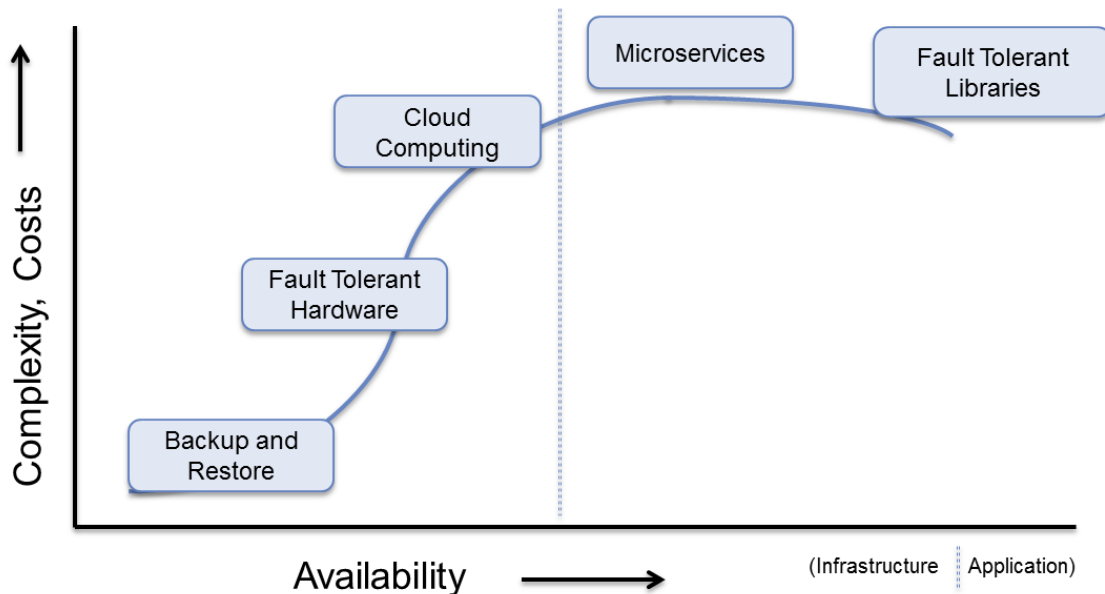


Figure 7-1: Availability vs. Complexity, Cost

Some of the key points observed during the work are listed below:

- Although technologies have improved greatly during the past decade at the infrastructure level, reaching high availability is still fraught with high costs and complexity.
- Application level architecture and design can also contribute to avail availability.
- Hystrix fault tolerant library, when implemented with microservice architecture, enhances the availability of the system.
- The need for real-time monitoring for distributed services is fulfilled by the Hystrix dashboard.
- Design patterns such as circuit breaker, bulkhead, fail fast, and fallback are easy to implement with the help of Hystrix library which adds no cost to the system.
- Hystrix provides different ways for implementation which can be easily configured for a new application and can also be configured to the old applications in the enterprise.
- Hystrix is very flexible, and any java based application can be easily integrated with it.

Testing is an integral part of software development and contributes towards the goal of reaching availability. Firstly, the system was tested using traditional methods of manual testing. The discussion was made on the possible failures and what could go wrong. The failure scenarios were determined, and manual testing was performed by executing corresponding failure tests. Soon it was observed that the space of possible failures is huge when dealing with distributed systems based on microservice architecture. In-depth manual testing is tough and conducting random search and test would take too long.

Since the distributed system grows exponentially over the time, so detecting the faults as early as possible saves resources. Injecting failures and figuring out the problem increases the confidence level of testing and also improves the code quality. The code can be reviewed when error is found which can improve it. The developer will debug the error and have the opportunity to improve the code as well. Consequently, the overall system can be enhanced and customer satisfaction can be increased.

By adopting and analyzing new mechanisms and technologies at the application level, it was observed that availability could be increased without increasing a lot of complexity and cost. The overall concept which came out after all the observations made throughout the work is shown in the figure 7-2.

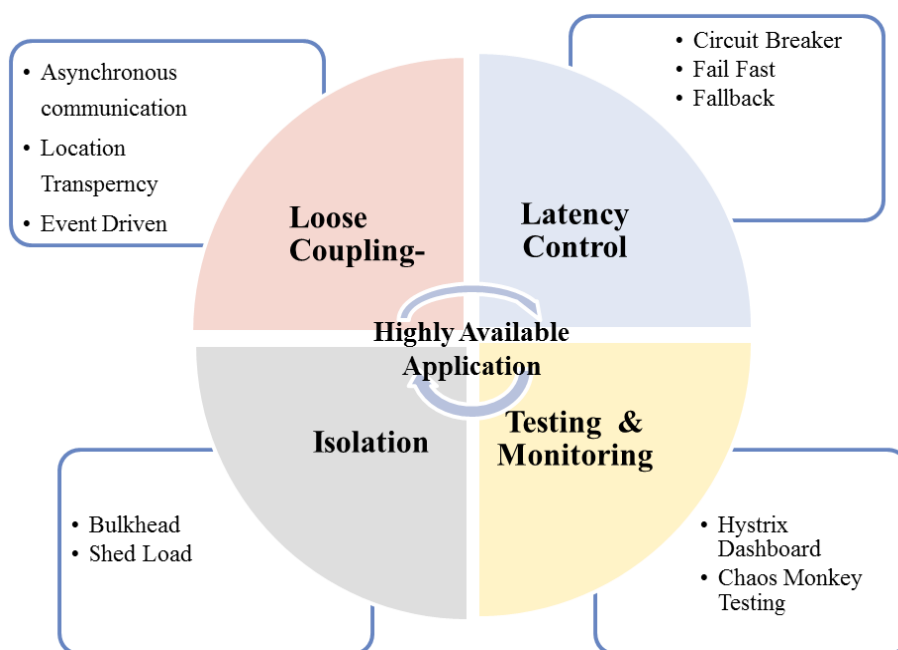


Figure 7-2: Result- HA application

Different patterns and libraries were discussed in the previous chapters but they were never tested together. NTT Data worked on microservices but never implemented Hystrix library to it. The thesis goal is to give a proof that all the technologies can work together. The main properties needed to make application highly available are Loose-coupling, Latency Control, Isolation and Testing along with Monitoring. These four essential properties cover the scope of work only for this thesis. This is the concept on which NTT Data will start building applications which are more resilient and available. The work was to test the Hystrix and Chaos Monkey frameworks functionality which was done successfully by using demo applications and a concept is prepared which is ready to be implemented in upcoming projects for the clients. The connected car backend applications can be designed using the concept of the thesis. The backend applications on which NTT Data will work are the based on the distributed systems. The thesis describes about the challenges in distributed systems and how

they can be tackled at the application level. The backend applications for the connected car will be developed using Hystrix by NTT Data for the future projects. The thesis covers the topic in a general way, the company did not allow to use any application use case or ongoing projects because of the restrictions from the client Daimler.

7.2 Summary

The topic of the thesis deals with the concept of achieving high availability. The implementation of fault tolerance mechanisms in microservice platforms has been achieved by integrating Hystrix library. Hystrix proves to be the most advanced library for fault tolerance until now. The implementation of Hystrix with the easily configurable annotations makes it very simple to use. Testing distributed systems with a new type of chaos engineering and monitoring the real-time action of circuit breakers with Hystrix dashboard makes the proposed solution very robust and ready to be used for the projects at NTT Data.

The work analyzed the mechanism to inject failure to the system deliberately and test for failures. The implementation of the mechanisms proves the effectiveness of using Hystrix and Chaos Monkey methodologies for gaining high availability.

The main concept is build based on the problems that are found with the current distributed applications which results into downtime and the need for testing distributed systems for cloud-based applications. The challenges in distributed systems were found, and new mechanisms were analyzed to face them. Conceptualization was focussed on building a highly available application architecture, which combines the advantages of design patterns and fault tolerant libraries. Once this was achieved, a mechanism was built that mainly focused on testing the distributed systems.

The most important part of the result was that the concept developed is extremely generic. This can help in a lot of ways, and the results can be scaled to powerful levels in the time ahead.

It has been seen that making application resilient improves the overall availability of the system. The powerful features provided by Hystrix to accomplish the goal are circuit breaker pattern, failure isolation, thread pools, fallback, and dashboard. Implementing these features in a microservice architecture makes the system highly available. The Chaos Monkey testing methodology helps to tackle the complexity of testing a distributed system. The automated tests of chaos engineering can introduce failure and creates a scope for improvement in the system.

7.3 Outlook

Although the current solution offers rich configuration options and makes testing easy, there is still some room for improvement. The fault tolerant libraries other than Hystrix needs

enhancement which is under heavy development. Overall, the solution developed hints at promising concepts of development for software development for distributed systems.

In the current solution, Hystrix dashboard allows to watch the services and figure out failures when anything goes wrong. In future work, there should be a solution which can automatically detect errors and try to debug it and give a report on whether a service is resilient rather than relying on a human to keep an eye on the dashboard to figure it out. For some problems only a reboot to servers works out which can be automated, Hystrix dashboard gives result in the metric form which can be used to get an idea about the error which can be debugged automatically for certain situations which are not so critical.

Netflix is updating the technology rapidly so in future work the chaos testing can be integrated with cloud continuous delivery platform to make the automatic start of the process. On the application level, solutions can be built which can introduce failures by sending random messages or delaying messages.

The availability factor is so important that new technologies and mechanisms keeps on improving, cloud computing providers have different ways of dealing with downtime problem and platforms like Windows Azure, Heroku, Google AppEngine provide different ways to achieve high availability, so in the future there is a need to test Hystrix enabled application with them and do the comparison in terms of availability and make use of their benefits.

The mechanisms for reaching high availability both at the application level and infrastructure level can be combined to see the overall effect on the availability of distributed applications.

Finally, it would be interesting to involve more companies in the study for making a more general approach to gain High Availability for connected car applications and produce more reliable and general solutions.

Bibliography

- [1] N. Data, "Connected Car Report," NTT Data, Stuttgart, 2015.
- [2] T. C. Co., "The 10 Biggest Cloud Outages of 2016," CRN, [Online]. Available: <http://www.crn.com/slide-shows/cloud/300083247/the-10-biggest-cloud-outages-of-2016.htm/pgno/0/1>. [Accessed 12 January 2017].
- [3] F. Khendek, M. Hormati and M. Toeroe, "Software Reliability Engineering: Towards an evaluation framework," p. IEEE International Symposium On. IEEE. pp 43, 2014.
- [4] A. Kanso and W. Li, "Comparing Containers versus Virtual Machines," *Achieving High Availability*, 2014.
- [5] Li Wubin, A. Kanso and A. Gherbi, "Leveraging Linux Containers," *Achieve High Availability for Cloud Services*, 2015.
- [6] A. Basiri, A. Blohowiak and C. Rosenthal, "A Platform for Automating Chaos Experiments," *Netflix*, 2017.
- [7] H. Tomasson, "Distributed Testing of Cloud Applications," 2011.
- [8] N. Data, "Connected Car Image Gallery," NTT DATA, Stuttgart, 2016.
- [9] ITILv3, "Knowledgetransfer.net," Knowledgetransfer, 2011. [Online]. Available: <http://www.knowledgetransfer.net/dictionary/ITIL/en/Availability.htm>.
- [10] W. Torell and V. Avelar, "MTBF: Explanation and Standards," *Mean Time Between Failure*, no. 78, 2010.
- [11] E. Watanabe, "Survey on Computer Security," Japan Information Development, Tokyo, 1986.
- [12] J. Gray and D. P. Siewiorek, "High Availability Computer Systems," *High Availability Paper for IEEE Computer Magazine Draft* .
- [13] J. Gray and D. P. Siewiorek, "High Availability Computer Systems," Digital Equipment Corporation, San Francisco, CA, 2010.
- [14] NTT, "Distributed System," NTT, Stuttgart, 2017.
- [15] P. Bailis, "ACM Queue," *Tracing and Debugging Distributed Systems*, vol. 15, no. 1, p. 11, 2017.
- [16] W. Vogels, *Tweet: Everything fails, all the time*, San Francisco: Twitter, 2015.
- [17] B. Kirsch, "TechTarget," March 2016. [Online]. Available: <http://searchservirtualization.techtarget.com/tip/Plan-for-hardware-failure-because-you-cant->

avoid-it. [Accessed 17 December 2016].

- [18] N. Slack, *Operations Management*, London: Prentice Hall, 2001.
- [19] EventHelix, "EventHelix.com," [Online]. Available: http://www.eventhelix.com/RealtimeMantra/FaultHandling/reliability_availability_basics.htm. [Accessed 21 November 2016].
- [20] F. Piedad and M. Hawkins, *High Availability: Design, Technique, and Processes*, Upper Saddle River: Prentice Hall PTR, 2001.
- [21] D. Linthicum, "Cloud Adoption's Tipping Point Has Arrived," *InfoWorld*, 2013. [Online]. Available: <http://www.infoworld.com/article/2611402/cloud-computing/cloud-adoption-s-tipping-point-hasarrived>.
- [22] T. Mell and G. Grance, "Cloud Computing," NIST, 2011.
- [23] f. Liu, T. Jin and J. Mao, "NIST Cloud computing reference Architecture," *NIST Special Publication*, pp. 500-292, 2011.
- [24] Microsoft, "Developer Network," [Online]. Available: <https://msdn.microsoft.com/en-us/library/dn568099.aspx>. [Accessed 08 March 2017].
- [25] MuleSoft, "Microservices architecture vs monolithic architecture," MuleSoft, INC., 2017. [Online]. Available: <https://www.mulesoft.com/resources/api/microservices-vs-monolithic>.
- [26] J. Evans, *Mastering Chaos - A Netflix Guide to Microservices*, https://www.infoq.com/presentations/netflix-chaos-microservices?utm_source=youtube&utm_campaign=newcircleexperiment&utm_medium=link, 2016.
- [27] Netflix, *Spring Cloud Netflix*, <http://cloud.spring.io/spring-cloud-netflix/spring-cloud-netflix>.
- [28] Microsoft, "Designing Web applications," [Online]. Available: <https://msdn.microsoft.com/en-us/library/ee658087.aspx>. [Accessed 27 April 2017].
- [29] Fowler, Martin, "Microservices," 2014.
- [30] M. L. Abbott and M. T. Fisher, *THE ART OF SCALABILITY*, Boston: Pearson Education, 2009.
- [31] M. Fowler, "MartinFowler.com," Thought Works, 25 March 2014. [Online]. Available: <https://martinfowler.com/articles/microservices.html>. [Accessed 05 02 2017].

- [32] D. Inc, "Docker," Docker, [Online]. Available: <https://www.docker.com/>. [Accessed 21 February 2017].
- [33] M. Green, "The Blueberry Insights Blog," Blueberry Consultants, [Online]. Available: <https://www.bbconsult.co.uk/blog/docker-containers-used-software-developers>.
- [34] Microsoft, "Microsoft Library," 2016. [Online]. Available: <https://docs.microsoft.com/en-us/azure/architecture/patterns/health-endpoint-monitoring>. [Accessed 15 December 2016].
- [35] H. Chan and T. Chieu, "An approach to high availability for cloud servers with snapshot mechanism," 2013. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2405152>. [Accessed 1 March 2017].
- [36] D. Alexandrov T, "Software availability in the cloud.," ACM, 2013.
- [37] M. Narumoto and C. Babbage, "Microsoft Azure Documentaiton," [Online]. Available: <https://docs.microsoft.com/en-us/azure/architecture/patterns/leader-election>. [Accessed 26 March 2017].
- [38] "Algorithm," [Online]. Available: <http://web.cs.iastate.edu/~cs554/NOTES/Ch6-Election.pdf>.
- [39] A. Homer, J. Sharp, L. Brader, T. Swanson and M. Narumoto, "Cloud Design Patters," 2014.
- [40] S. Newman, Building Microservices, O'Reilly Media, 2015.
- [41] M. Wasson, "Cloud Design Pattern," no. Microsoft Library, 2017.
- [42] "Github Hystrix Wiki," [Online]. Available: <https://github.com/Netflix/Hystrix/wiki/How-it-Works>.
- [43] Netflix, "Netflix Hystrix wiki," Github Netflix, [Online]. Available: <https://github.com/Netflix/Hystrix/wiki>. [Accessed 22 February 2017].
- [44] "Github- javanica," Github open source, [Online]. Available: <https://github.com/Netflix/Hystrix/tree/master/hystrix-contrib/hystrix-javanica>. [Accessed 11 February 2017].
- [45] Netflix, "Github Hystrix Netflix," Netflix, 2016. [Online]. Available: <https://github.com/Netflix/Hystrix/wiki/Dashboard>. [Accessed 10 02 2017].
- [46] Comcast, "GitHub," [Online]. Available: <https://github.com/Comcast/jrugged/wiki>. [Accessed 24 March 2017].

- [47] Github, "JavaSlang," [Online]. Available: <https://github.com/javaslang>.
- [48] Github, "Resilience4j," Github Open Source, [Online]. Available: <https://github.com/resilience4j/resilience4j>. [Accessed 15 December 2016].
- [49] GitHub, "GitHub," [Online]. Available: <https://github.com/siamaksade/wildfly-swarm-hystrix-example>.
- [50] "Vert.X circuit breaker," [Online]. Available: <http://vertx.io/docs/vertx-circuit-breaker/java/>.
- [51] N. Team, "Netflix Open Source Software Center Insight, Reliability and Performance," [Online]. Available: <https://netflix.github.io/>.
- [52] N. Blog, *Chaos Monkey Netflix*, Netflix, 2011.
- [53] E. B. Boyter, *Chaos Test Engineering*, Konstantin Kovshenin, 2016.
- [54] Netflix, "Simian Army," Netflix Official, 2017.
- [55] Spring.io, "Aspect Oriented Programming with Spring," [Online]. Available: <http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html>.
- [56] M. Sertic, "Hystrix Spring," Github, [Online]. Available: <https://github.com/mirkosertic/HystrixSpring>.
- [57] Ericsson, "High Availability is more than five nines," 2014. [Online]. Available: <http://www.ericsson.com/real-performance/wp-content/uploads/sites/3/2014/07/high-availability.pdf>, 2014..
- [58] J. M. Parrilla, "Monkey-Ops: <https://github.com/Produban/monkey-ops>".



TECHNISCHE UNIVERSITÄT
CHEMNITZ

Studentenservice – Zentrales Prüfungsamt
Selbstständigkeitserklärung

Name: Yadav	Bitte beachten:
Vorname: Arpit	1. Bitte binden Sie dieses Blatt am Ende Ihrer Arbeit ein.
geb. am: 24.01.1989	
Matr.-Nr.: 366669	

Selbstständigkeitserklärung*

Ich erkläre gegenüber der Technischen Universität Chemnitz, dass ich die vorliegende **Masterarbeit** selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keinem anderen Prüfer als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

Datum: **28.04.2017**

Unterschrift: *Arpit Yadav*

* Statement of Authorship

I hereby certify to the Technische Universität Chemnitz that this thesis is all my own work and uses no external material other than that acknowledged in the text.

This work contains no plagiarism and all sentences or passages directly quoted from other people's work or including content derived from such work have been specifically credited to the authors and sources.

This paper has neither been submitted in the same or a similar form to any other examiner nor for the award of any other degree, nor has it previously been published.