Self-Healing of Distributed Systems

A dissertation submitted for the Degree of Master of Computer Science

G.P. Sanjeewa
University of Colombo School of Computing
2019

Declaration

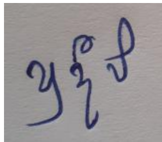
The thesis is my original work and has not been submitted previously for a degree at this or any other university/institute.

To the best of my knowledge it does not contain any material published or written by another person, except as acknowledged in the text.

Student Name: G.P. Sanjeewa

Registration Number: 2015/MCS/069

Index Number: 15440691



____

Signature: Date: 2020/06/20

This is to certify that this thesis is based on the work of

Mr. G. P. Sanjeewa

under my supervision. The thesis has been prepared according to the format stipulated and is of acceptable standard.

Certified by:

Supervisor Name: Dr. Kasun Karunanayake

_____________________ 21/06/2020

Signature: Date:

Abstract

Self-healing is a buzzword in distributed computing domain since last decade. During this study, the evolution of distributed systems with respect to self-healing was studied. Some of the common architectures used for self-healing of distributed sytems is presented. With the inception of cloud computing, cloud computing platforms have vastly adopted self-healing methodologies when running their large scale server farms.

The study was inspired by the project which the author has been working at his company. In the upcoming sections author has proposed some automated strategies to heal a distributed system replacing the manual process which involves human intervention. For the implementation of the suggested methodology, author has proposed a solution based on *docker* and *kubernetes*. Self-healing solution has been implemented on a WSo2 EI cluster. The research has been able to achieve the self-healing of a VM cluster in VM level. Going further from initial objectives, auto scaling has been achieved in the VM cluster as well.

Table of Contents

List of Figures iii

List of Tables iv

Abbreviations iv

Chapter 1: Introduction 1

1.1 What is a distributed system? . . . 1

1.2 What is a self-healing system? . . . 2

1.3 Motivation . . . 2

1.4 Aims and Objectives of the Project . . . 3

1.5 Scope . . . 3

Chapter 2: Literature Review 4

2.1 Related research work . . . 4

2.1.1 Maintaining system health . . . 4

2.1.2 Identification of faulty states . . . 7

2.1.3 Recovery of system from faulty state . . . 8

2.2 Current research limitations and research gap analysis . . . 9

Chapter 3: Methodology 10

3.1 Monitoring system health . . . 10

3.2 Identification of faulty states . . . 11

3.3 Recovery of system from faulty state . . . 12

Chapter 4: Proposed Solution 15

4.1 Selected Tools and software components . . . 15

4.1.1 Tools . . . 15

4.1.2 Software Components . . . 15

4.2 Building the Solution . . . 15

4.2.1 Mock API . . . 16

4.2.2 Pod definition 17
4.2.3 Deployment definition 18
4.2.4 Node-port service definition 19
4.2.5 HPA definition 19
4.3 How the initially defined objectives were achieved 21

Chapter 5: Evaluation and Results **22**
5.1 Analysis of implemented solution 22
5.2 Excising Methodologies 24
5.3 Comparison of proposed solution and existing solution 25
5.3.1 Time to recover 26
5.3.2 Human intervention 26
5.3.3 Root cause analysis 27

Chapter 6: Conclusion and Future Work **28**
6.1 Future work 28
6.2 Conclusion 28

References **29**

List of Figures

1.1 Simplified Apigate API gateway server setup 1

2.1 Architecture based monitoring detailed setup [10] 5
2.2 Externalized Dynamic Adaptation Infrastructure [28] 6

3.1 Monitoring and healing model 11
3.2 Fault identification model 11
3.3 Healing agent architecture 13

5.1 Nodes in kubernetes cluster 22
5.2 Initial pod in kubernetes cluster 22
5.3 Node-port service 22
5.4 Replica set 23
5.5 Horizontal pod auto-scaler 23
5.6 Deleting a pod 23
5.7 Apache jmeter set up to apply load 24
5.8 Apache jmeter set up to apply load 24
5.9 Initializing a new pod 25
5.10 Initializing a new pod 25
5.11 Initializing a new pod 25
5.12 Deleting pods 26
5.13 Deleting pods 26

Abbreviations

ADL	Architecture Description Language	1, 8
AM	API Manger	1, 15
API	Application Programming Interface	1
DEP	Digital Enablement Platform	1, 12
EI	Enterprise Integrator	i, 1, 12, 15, 21, 22
HPA	Horizontal Pod Autoscaler	ii, 1, 19
HTTP	Hypertext Transfer Protocol	1, 19
HTTPS	Hypertext Transfer Protocol Secure	1, 19
IGW	Internal Gateway	1
IS	Identity Server	1, 15
KM	Key Manager	1, 12
REST	REpresentational State Transfer	1, 15
SLA	Service-level Agreement	1, 3
VM	Virtual Machine	1, 19

Chapter 1: Introduction

Self-healing of distributed systems is a popular topic in the industry today. When enterprise software systems increase in complexity, rectification of system faults and recovery from malicious attacks become more difficult, labor-intensive, expensive and error-prone. These factors have actuated research dealing with the concept of self-healing systems [12].

1.1 What is a distributed system?

A set of networked computer(server) nodes which collectively perform one complex computational task is defined as a distributed system. These computers communicate with each other by passing messages between them[26].

Apigate API gateway is a good example for a distributed system.

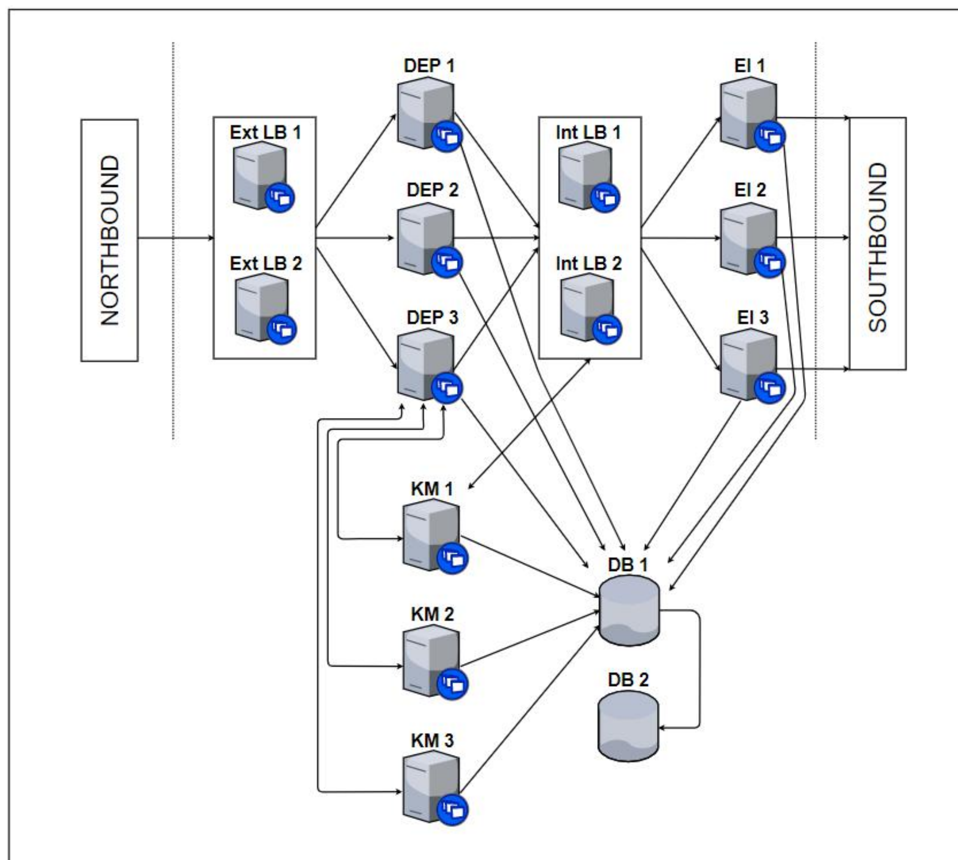


Figure 1.1: Simplified Apigate API gateway server setup

Here, each server node has a dedicated task while the whole system collectively operates as an API gateway.

1.2 What is a self-healing system?

It is a property of the system that it can identify when it is not operating correctly and ability to make necessary adjustments to restore itself (or with human intervention) to normalcy. Just like a biological systems heal a wound, self-healing systems heal themselves[12]. Self-healing needs to be happened by the system itself or with some human intervention. Dynamically adapting to the changes in the environment is expected from a self-healing system. At run-time, system should be able to deal with the possibilities which were not anticipated in design-time[25].

There are another two terms going closely with self-healing systems; fault-tolerant systems and survivable systems. In fault-tolerant systems, backup systems stay stand by to take the position of the main system in case of a malfunction of the main system. This is required to happen without any service interruption[23]. This standby system is also known as a ’mirror’ systems [12]. Survivable systems are an extension to the fault-tolerant systems. Survivable systems keep the system secure in the presence of malicious or arbitrary fault[20].

Self-healing systems consider about taking the current working system back to normalcy rather than depending on a backup standby system. In this study, our focus is to come up with a solution which can detect the faulty states and recover from faulty states autonomously without human intervention.

The term "self-healing" is originated from the natural biological system of humans[12]. Our body has the ability to heal itself when it is damaged. Let it be a wound, food poisoning, body tries to recover itself. Human immune system plays a vital role in self-healing in human body. This has opened a new area in computer science research called artificial immunology[7].

1.3 Motivation

Apigate API gateway is run on a distributed system. At work, we often face scenarios where the system runs into some of the unintended states. In those states, we have often noticed high transaction time or complete traffic failure. Then, human interaction is required to get the system back to a working state. In most of the scenarios, what needs to be done to recover the system is straight forward. Only problem is the inability of the system to identify that it is in a unintended

state and it doesn't know how to recover from that state. In such a situation human interaction is required to resolve the issue. Human interaction always come up with financial cost and time. That affects the overall performance of the system and sometimes we are unable to meet the SLA provided to the partners who are using API gateway. Therefore, we thought of coming up with a solution to that problem we are facing in the industry.

1.4 Aims and Objectives of the Project

Main objective of this research is to come up with a solution to self-healing of distributed systems. Going forward solution will be implemented on Apigate IGW. To come up with a novel solution, similar approaches will be studied. The knowledge generated from previous researches will be used to come up with a better solution. A new solution will be developed based on proposed methodology.

1.5 Scope

Scope of the research was limited to look for a solution to self-healing in distributed systems. System quality wise parameters like response time was not taken in to consideration in this study. In this research, I did not consider about self-healing of security threats like phishing attacks, malware attacks etc. In this research we have considered only on the server level self-healing. We did not look into application failures. During implementation, self-healing methodologies were only applied to internal load balancers, external load balancers, database, digital enablement platforms(DEPs), key managers(KMs) and enterprise integrators(EIs) of Apigate IGW environment as they are the most critical server nodes. Python script can be used to automate the solution deployment process.

Chapter 2: Literature Review

Researches in self-healing of computer systems go down till 70's[2]. When reading the literature, it is obvious that the problem has been there since the inception of computers, but the problem has taken more attention at the beginning of this century when the computer systems started getting complex.

2.1 Related research work

The related literature can be summarized into three main categories.

1. Maintaining system health
2. Identification of faulty states
3. Recovery of system from faulty states

Following sub subsections will describe the related research work in detail classifying them to above three categories.

2.1.1 Maintaining system health

Monitoring the system health to maintain proper system health plays a vital role in distributed systems. Strategies followed in literature to monitor and maintain system health will be described in this subsection.

Some self-healing distributed systems maintain an agent who will keep monitoring the system health. In most of the systems which we describe below, follow this agent architecture. Monitoring process can be continuous or periodical depending on the criticalness of the system. There are different strategies in literature which are used to monitor and maintain system health of a distributed system.

2.1.1.1 Maintaining redundant components

Replicating system components to maintain system health is a costly but popular concept. Negpal et al. [22] has presented a programming methodology for self-assembly of complex structures

using a technique inspired by biology. In this research, author has described the mechanism of deploying multiple identical components to facilitate the redundancy of a distributed system.

Huhns et al. [15] has proposed a different type of redundancy policy for multi-agent systems. In this approach, two types of decision making algorithms have been used for pre-processing and post-processing decision making. In this approach, redundant agents have been used to keep the system running with different algorithms.

2.1.1.2 Using probes

Probing is commonly used to monitor system health. In literature there are several strategies used to maintain system health using probing.

1. Architecture based monitoring

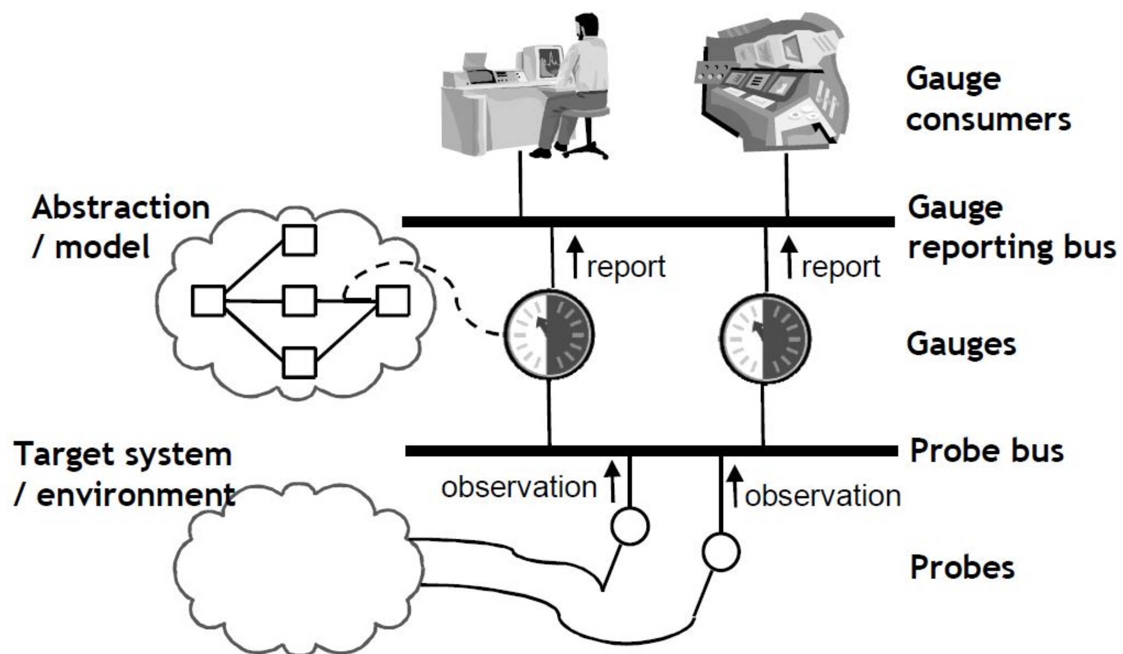


Figure 2.1: Architecture based monitoring detailed setup [10]

Garlan et al.[10] suggest a monitoring system with 3 layer architecture as mentioned in figure 2.1. In this research, probes are deployed in actual system which keeps monitoring system parameters. Probes send information of the system to "Probe bus". Gauges in level two use the information sent from "probes" and interpret the received data in higher level model properties. Extracted information is sent to "Gauge reporting bus". Gauge consumers are in the third layer. This layer uses the gauge values to take system level decisions like display warnings and alerts, decision on system repair or to get the status

of the system.

2. Decision and control layer

   A similar approach has been suggested by Kaiser at el. [16]. to add self-healing features to legacy systems. Here, they have used gauges and probes to obtain system properties and interpret the same in higher level model properties. Apart from that they have used an extra layer called "Decision and control layer" which obtain information from gauge reporting bus to optimize the deployment of gauges and probes.

3. Feedback control loop

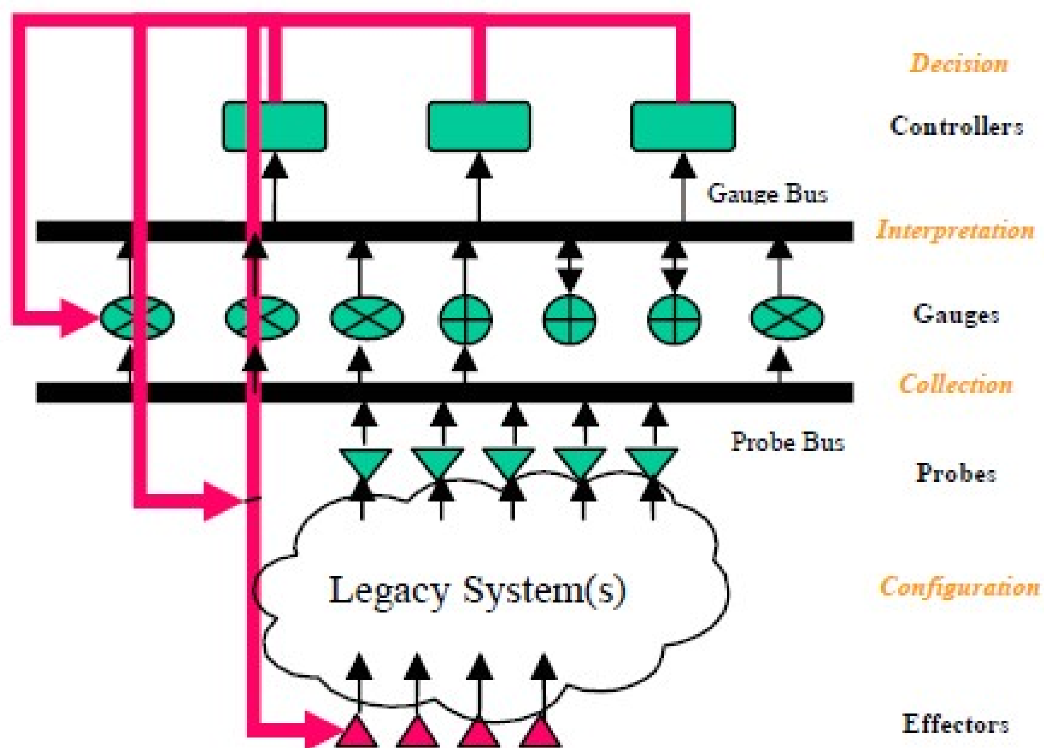


Figure 2.2: Externalized Dynamic Adaptation Infrastructure [28]

Valetto at el. [28] have proposed the same "probes and gauges" approach. Apart from that, they have introduced a feedback control loop for automatic reconfiguration of probes and gauges. This can be implemented on any legacy system. The model introduced here is independent from the running system. When designing probes and gauges, the architecture of the system must be closely followed. A probe here is an individual sensor attached to running program or a component of running program. Feedback system controls the parameters of running system as well. So, there is some dependency caused by self-healing mechanism. Anyway, the researchers have kept the interruption at a

minimal level.

4. Adaptive mirroring

   Combs el al. [5] propose a mirroring architecture for self-healing of distributed systems. This assumes there are substitutes available for the failing systems. In this paper, authors have used "probe" concept to get the information of failing components.

5. Gathering data from functional layers using sensors

   Two types of sensors are introduced here to collect the information of the system. State sensors collect information about the state of the system while analysis sensors gather information about messages flowing through the system.

2.1.1.3 Log analysis

Normally, most of the computer systems generate logs of different types. There are application specific logs as well as the logs related to server health. In literature, we can find researches which analyze real time logs to identify the problems in live systems.

Michael et. al [17] propose a performance monitoring system for Windows servers. They have come up with a service called *WatchTower* which monitors all the running terminals. This service is similar to UNIX daemon. User interaction is not required to start the service and WatchTower service starts automatically at system start up. WatchTower analyses all the open command lines and consoles to get the current status of the system.

In literature related to fault-tolerant computing, there are researches which are using error logs of the system to identify the failures of the system [24][3][24][18][19][27].

Kishor et. al [14] present a software rejuvenation model which monitors for application level failures of a system. In this paper various techniques to follow in design, test, debug and operation phases of the software are described. Parameters which vary with software aging are also taken into consideration. In this approach, the rejuvenation process identifies the issue and tries to restart only few processes to resolve the issue. If that fails, it can even restart the whole system to heal it.

2.1.2 Identification of faulty states

The next famous topic when it comes to self-healing systems is identification of faulty states in the system. The data received through system level probes, log analysis needs to be gauged

properly to identify whether the system has reached the threshold. Like in a biological system, failure in one component should be able to recover without affecting other components of the system.

Identification of faulty states can be classified into two subtopics according to the literature analysis; identification of missing components or missing messages from the system and system monitoring models which has the sense of system architecture.

2.1.2.1 Identification of missing components or missing messages from the system

Negpal et al. [22] have come up with a self-repair architecture based on agents. Here they have proposed a monitoring model which it can sense the missing of neighbour components of the system. The architecture of the software system is agent based. Every agent is replicated and surrounded by these replicated agents. When the replicated agents sense their neighbour is missing, one of the replicated agents take the place of the dead agent. George et al. [11] have also pointed out that any failure in system can be identified by missing of messages from the failed component.

Aldrich et al. [1] point out their strategy to identify missing components is, not receiving the response to a query. In this approach they have embedded two different methods to identify system availability; periodic announcements from systems and retries to connect to component.

Dabrowski et al.[6] mention that missing of scheduled announcements indicate that the announcing component is malfunctioning or a network failure.

2.1.2.2 System monitoring model

Garlan et al. [9] mention that monitored values of the system can be mapped to the architectural properties of the system to identify whether the components are functioning properly. This is the concept adopted by most ADLs [4][8][28]. Merideth et al. [21] propose when a fault is detected continuously probing the system can improve the survivability of the system.

2.1.3 Recovery of system from faulty state

In previous sections, we discussed about different monitoring mechanisms available and how to identify whether the system is in an unhealthy state. In this section we will analyze the literature

related to recovery of the system from faulty states. Any self-healing system should consists of policies which heal the system when it is in an unhealthy state. The literature related to recovery mechanisms for self-healing can be classified into two main categories; maintaining redundancy and usage of architectural models to heal the system.

2.1.3.1 Maintaining Redundancy

Nagpal et. al. [22] have mentioned about self-assembly in self-healing systems. In this approach, system maintains an agent architecture. Each agent has several replicated agents around it. When an agent is dead, one of the replications take the place of the agent to maintain proper system functionality. George et al. [11] state that failure can occur in network path or the component itself. In this research they have followed the analogy of a biological system. There are several replicated nodes which keeps transmitting their existence. Once that signals are not received, a replicated node will take up the operations.

2.1.3.2 Architectural models to heal the system

Cheng et. al. [4] propose a repair strategy using information provided by gauges. This method initially identifies the cause of problem and then repair the system. This repair strategy is tightly coupled with the system architecture. Valetto et al. [28] have proposed a high level repair strategy consisting a feedback loop. Feedback loop gets the input from gauges and adjust the system accordingly to maintain proper system functionality.

2.2 Current research limitations and research gap analysis

Most of the systems described above are not implemented in commercial systems. For an example, the methodology described by Cheng et al. [4] and Valetto et al. [28] are hardly bind with the architecture of the system. So, implementing self-healing in these methods require a good understanding of the system architecture. In general there are legacy systems in industrial distributed systems. So, trying to understand those systems is bit challenging. So, generally industry will not accept such systems if it is not really required.

Most of the researches were targeting on probing the system. There were several approaches in the literature for probing computer systems. Gauging and healing actions were not that attracted by researchers. In our study we will focus more on gauging and healing actions.

Chapter 3: Methodology

There are lots of examples out there when it comes to industrial distributed systems. Telephone networks, computer networks(world wide web), wireless sensor networks are popular among them.

In this research we are planning take an API gateway as the distributed system to apply self-healing. To be more specific, we will be using Apigate API gateway as the system to implement self-healing. Simplified network diagram of Apigate API gateway can be found in figure 1.1.

3.1 Monitoring system health

Defining the correct behaviour and faulty behavior of a system is challenging. There is no clear cut boundary between faulty behaviour and the accepted behaviour. There is kind of a fuzzy state where it is not clear whether the system is in faulty state or not. So, we are required to define a threshold to distinguish two behaviours of the system. System health check can be done continuously or periodically. Anyway this health check mechanism needs to be able to identify correctly the fuzzy separation zone.

It is required to have an agent to monitor the system health. Agent can be a component of the system or an independent agent. Agent needs to monitor the component-wise system health periodically or continuously depending on the criticalness of the system.

High level implementation plan can be seen in figure 3.1.

As described earlier, there is an agent deployed in the distributed system as the monitoring/healing agent. In this implementation, we are using an already available server node as the agent for monitoring and healing.

In this monitoring and healing node, we have installed nagios. Nagios is a free and open-source software application which can monitor systems, networks and infrastructure. Nagios can monitor all server nodes in distributed system.Specifically, it can monitor memory usage, CPU usage of all server nodes in the network and the network connectivity to each server node. Furthermore, it can monitor disk usage of each partition in the distributed system. So, basically nagios is acting as the monitoring agent in the system.

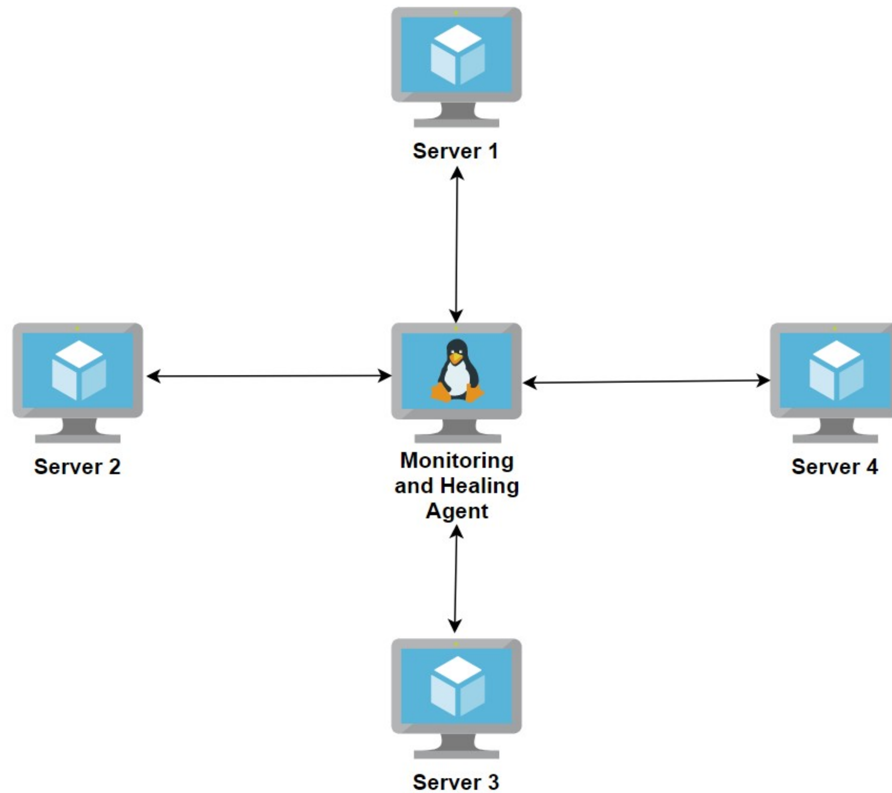


Figure 3.1: Monitoring and healing model

3.2 Identification of faulty states

In literature review section, several identification methodologies to grab the faulty states were discussed. Usage of gauges and usage of architectural models were the main identification methodologies used in literature. In this implementation, we will be using a methodology which is closely related to gauges.

Figure 3.2: Fault identification model

As discussed in the previous section, nagios will be used as the monitoring service. There is a separate python script running along with nagios service in the same server node. This python script is capable of accessing nagios using the APIs exposed by nagios service. There is a gauge

implementation done in this python script which can raise a flag when a parameter value exceeds a pre-defined threshold. So, identification of faulty states can be done using this python script which acts as the agent for identifying the faulty states.

3.3 Recovery of system from faulty state

The faulty component is required to be healed. In a biological system, when there is a wound, it will recover without disturbing normal body functions. The same is expected from the distributed system. It is required to heal without disturbing other functioning components. To assist this process, there needs to be adequate redundant components to take over the system in case of a failure of a component.

The recovery action is always attached to the architecture of the distributed system. This required to be planned with a thorough analysis of the system. As discussed earlier, there needs to be an agent who will monitor the component heath and perform the recovery process. It can be a different server node or a part of the distributed system. It is mentioned in literature[13] that the characteristics of a self-healing system usually match with software agent architecture, which is a multi-agent design style; and can perceive their environment.

Apigate API gateway has already implemented redundancy to some extend(check figure 1.1). As you can see in figure 1.1, there are two nodes of external load balancers, internal load balancers and and databases. Pure intention of having two nodes is to take the operation by stand by node when the main node is out of service. DEP, EI and KM nodes have three from each. That is intended to improve the capacity of API gateway since one node cannot handle the traffic of API gateway alone. Apart from handling the traffic, having multiple nodes also enable the redundancy of the the system. When one node is failed, other two nodes can continue the operation until the faulty node joins the operation after recovery.

In ExtLB, IntLB and DB nodes, there is a service called keepalived, which enables the fault-tolerant capability. With keepalived service, one virtual ip is exposed to the rest of the nodes in the distributed system. When a node is out of operation, virtual ip will route the requests to stand-by node and the situation is handled without any service impact.

So, Apigate API gateway has already implemented fault-tolerant features in some layers(IntLB, ExtLB and DB). For other layers(DEP, EI and KM), we can implement fault-tolerant features with the available resources. But that is not the goal of this research. When a system is equipped with fault-tolerant features, it doesn't mean the system is self-healing. Having fault-tolerant

ability is a good feature for a system. Still, having self-healing ability is the icing on the cake.

In previous two sections, we discussed on probing the system using nagios and gauging the values received from probes using custom python script which acts as the healing agent is shown in figure 3.3.

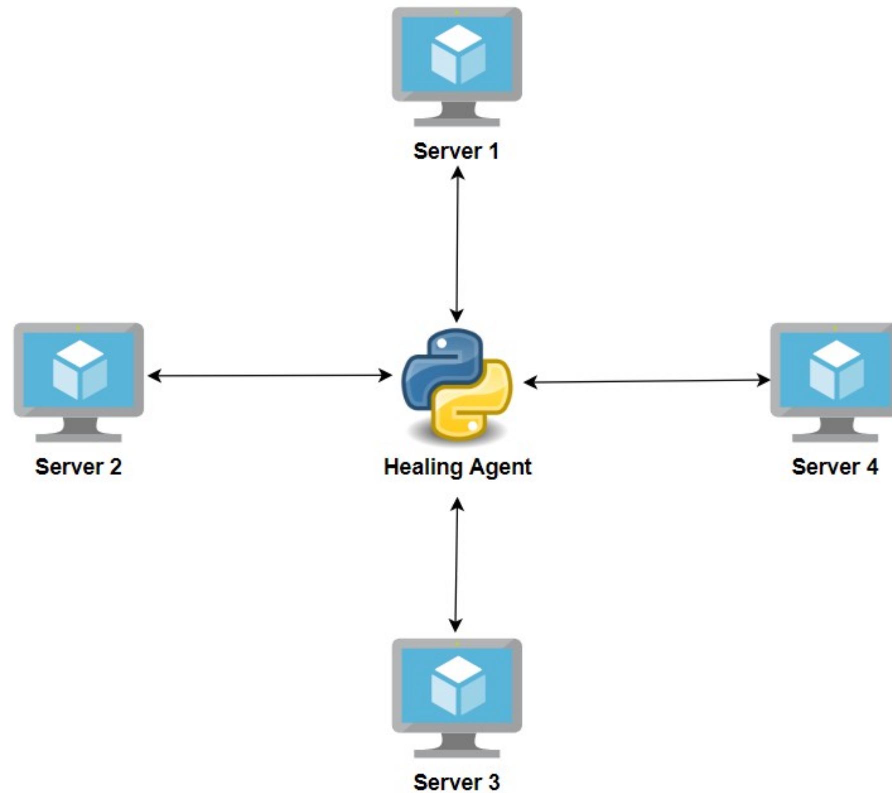


Figure 3.3: Healing agent architecture

Python script has defined the gauges for system parameters. When the parameters reach the gauge, healing agent can act as specified in the program. As mentioned in the figure 3.3, healing agent can directly control all server nodes in distributed system. If the agent identifies a known error behavior it will act as it is programmed. At the initial phase we will identify high memory usages, high CPU consumption of server nodes. When the memory usage or CPU consumption is high, healing agent will heal the node with following algorithm.

1. When the affected agent is IntLB, ExtLB or DB

   keepalived service switches the operation to stand by node in this scenario. Once the operation is switched to stand by node, affected node will be restarted by the healing agent. Then keepalived service will switch back the operation to this healed node.

2. When the affected agent is KM, DEP or EI

   These nodes are load balanced through load balancers as mentioned in figure 1.1. So,

once a node doesn't function properly, healing agent will detach the node from related load balancer. Then healing agent restarts the affected node and attach it back to respective load balancer.

Chapter 4: Proposed Solution

This research was inspired by the work currently I'm carrying out in my company. So, I thought of implementing the solution on a component of the system which my team manage currently. As I described in chapter 1, my team is managing an API gateway which consists of several layers of open-source software like WSo2 AM, WSo2 EI, WSo2 IS, nginx load balancers and mySQL databases. For the solution implementation I selected WSo2 EI since we can simply deploy an REST API using WSo2 EI.

4.1 Selected Tools and software components

For the solution implementation I used several open-source tools and software components.

4.1.1 Tools

As tools, I used docker and kubernetes to develop the solution. Docker was used to pull the docker container images from docker hub which is the largest open platform for containerized images. After deploying the simple REST API, again docker was used to bundle the software stack back as another docker image. Kubernetes was the container orchestration platform which was used to deploy and orchestrate the containers.

4.1.2 Software Components

As software components, the only software component which was used is the WSo2 EI docker image which is publicly available in docker hub. Docker image was downloaded from docker hub. WSo2 recommends to use the docker image from their private repository hosted at wso2.com for production environments. Still I used the image from docker hub since this is just a demonstration and not a production grade deployment.

4.2 Building the Solution

The solution is implemented using docker and kubernetes. To build the solution, there were five major components I had to look at.

1. Mock API
2. Pod definition
3. Deployment definition
4. Node-port service definition
5. HPA definition

4.2.1 Mock API

Mock API mocks a real API behaviour. Without calling an actual back-end, the response is hard coded in a mock API. In my solution I have used a mock API to put load on WSo2 EI. Mock API code is taken from `http://lahiruwrites.blogspot.com`

```
<api xmlns="http://ws.apache.org/ns/synapse"\
name="SimpleAPI" context="/simple">
   <resource methods="GET">
      <inSequence>
         <payloadFactory media-type="xml">
            <format>
               <Response xmlns="">
                  <status>OK</status>
                  <code>1</code>
               </Response>
            </format>
            <args/>
         </payloadFactory>
         <respond/>
      </inSequence>
   </resource>
</api>
```

This API is configured inside WSo2 EI and a modified docker image is used for the demonstration.

Below is the test API call.

```
curl --location --request \
GET 'http://192.168.8.111:30010/simple/'
```

Sample response is as below.

```
<status>OK</status>
```

4.2.2 Pod definition

Pod definition is the smallest element in a kubernetes deployment. It is the wrapper on top of a docker container. So, to wrap up the WSo2 EI image, we have to define a pod. Here is the pod definition file.

```
apiVersion: apps/v1
kind: Pod
metadata:
  name: ei-pod
  labels:
    type: ei
    version: 6.6.0
spec:
  containers:
  - name: integrator
    image: pradeepsanjeewa/wso2eiwithsimplemockapi:latest
    resources:
            limits:
                    cpu: 500m
            requests:
                    cpu: 200m
```

Here, I have used the modified docker image (pradeepsanjeewa/wso2eiwithsimplemockapi:latest) for the pod. CPU resources are limited to 500 millicpu cores.

4.2.3 Deployment definition

Even though we defined the pod in previous sub section, that won't be directly used in the kubernetes deployment. Pod definition is embedded inside the deployment definition.

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: ei-deployment
    labels:
        type: ei-dep
spec:
    template:
        metadata:
          name: ei-pod
          labels:
            type: ei
            version: 6.6.0
        spec:
          containers:
          - name: integrator
            image: pradeepsanjeewa/wso2eiwithsimplemockapi:latest
            resources:
                    limits:
                            cpu: 500m
                    requests:
                            cpu: 200m
    replicas: 1
    selector:
        matchLabels:
            type: ei
            version: 6.6.0
```

Here, I have defined the kubernetes deployment embedding the pod definition. In the deployment definition, we can mention the number of replicas to be available at a given time.

So, once that is defined, even if a pod starts to malfunction, another pod is initiated by kubernetes platform to maintain the defined number of replicas.

4.2.4 Node-port service definition

To expose the ports of a pod to host computer(or VM) I'm using the node-port service. Node-port service definition is as below.

```
apiVersion: v1
kind: Service
metadata:
  name: ei-service
spec:
  type: NodePort
  ports:
    - targetPort: 8280
      port: 8280
      nodePort: 30010
      name: http
    - targetPort: 9443
      port: 9443
      nodePort: 30011
      name: https
  selector:
    type: ei
    version: 6.6.0
```

Here I have exposed two ports in pod; 8280 and 9443. 8280 is the port which accepts HTTP requests. So, to hit API calls, we need to keep open this port. 9443 is the HTTPS carbon console port where we can view all the deployed carbon apps(API files).

4.2.5 HPA definition

HPA definition file defines all the matrices it needs to monitor to trigger a new pod. If the defined matrices are exceeded, a new pod will be initialized. Below are some matrices HPA can monitor.

1. Pod CPU usage
2. Pod memory usage
3. custom matrices

Under custom matrices related to this study, we can use parameters like response time from down-stream, log pattern monitoring and response code monitoring, For this study I'm using only CPU usage.

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-deployment
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: ei-deployment
  minReplicas: 1
  maxReplicas: 6
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 200
```

Here we have defined the matrices to consider for scaling up the pods in the cluster. Average CPU utilization is provided as 200% for a pod. If that limit is exceeded spinning up new pods starts. Here we have defined *minReplicas* to two and *maxReplicas* to six. So, HPA can scale the cluster up to six replicas and it will keep minimum of one replica at a given time.

4.3 How the initially defined objectives were achieved

The initial objective of the research was to come up with a better methodology for self-healing of distributed systems. In the implementation I have mentioned how we can come up with a self-healing distributed system for a WSO2 EI layer. When we define the number of *replicas* in deployment definition, it makes sure it always keep the defined number of pods in the environment. If a pod malfunctions, platform spin up a new pod to match the number of pods defined in the deployment definition. In this way, self-healing part of the distributed system can be achieved.

Going forward from the initially defined objectives, we can define a HPA which monitors the system matrices and custom matrices to scale up the number of nodes in cluster to serve traffic spikes or high traffic periods like promotions. HPA overrides the *replicas* configuration in deployment definition file and increase the number of pods (replicas) if the matrices are hit. In this way, the distributed system can auto scale to serve more traffic without degradation of the performance like response time.

Chapter 5: Evaluation and Results

This chapter evaluates the pros and cons of the suggested self-healing and scaling method with the existing methods. Furthermore, achieved results for the implementation will be described here.

5.1 Analysis of implemented solution

As described in the previous section, a deployment definition is used to initialize the kubernetes deployment. Once the deployment is created, one pod with WSo2 EI is deployed in a worker node. Here I have used three VMs for the implementation; one kubernetes master node and two worker nodes.

```
root@kubemaster:/etc/sudoers.d# kubectl get nodes
NAME          STATUS   ROLES    AGE   VERSION
kubemaster    Ready    master   28d   v1.18.3
kubenode01    Ready    <none>   28d   v1.18.3
kubenode02    Ready    <none>   28d   v1.18.3
```

Figure 5.1: Nodes in kubernetes cluster

Following is the deployment running in kubernetes cluster.

```
NAME                                READY   STATUS    RESTARTS   AGE
pod/ei-deployment-757665bdf5-98684  1/1     Running   0          57m
```

Figure 5.2: Initial pod in kubernetes cluster

As I described in the previous chapter, a node-port service is used to expose the ports of the pod through the ports of kubernetes nodes.

```
NAME                 TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)                         AGE
service/ei-service   NodePort   10.109.195.30   <none>        8280:30010/TCP,9443:30011/TCP   4d20h
```

Figure 5.3: Node-port service

Since the deployment definition defined the number of replicas as one, one replica of the pod is available.

```
NAME                                    DESIRED   CURRENT   READY   AGE
replicaset.apps/ei-deployment-757665bdf5   1         1         1       4d20h
```

Figure 5.4: Replica set

HPA is defined to spin up new pods in case the cpu utilization exceeds 200%.

```
NAME                                                REFERENCE                  TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/ei-deployment   Deployment/ei-deployment   4%/200%   1         4         1          4d20h
```

Figure 5.5: Horizontal pod auto-scaler

Deployment definition has explicitly defined the number of replicas as one. Therefore it tries to keep live one replica/pod at any given time. If the pod starts to be unresponsive, stuck or destroyed, a new pod will be initiated by kubernetes. This is demonstrated in the figure 5.6. When we try to delete a pod, it automatically spin up a new pod. In that way, self-healing can be achieved in VM level.

```
NAME                                 READY   STATUS        RESTARTS   AGE
pod/ei-deployment-757665bdf5-98684   1/1     Terminating   0          177m
pod/ei-deployment-757665bdf5-j99w7   1/1     Running       0          23s
```

Figure 5.6: Deleting a pod

Going further from self-healing, there is a HPA defined as shown in figure 5.5. Using the defined HPA, auto scaling of the cluster can be achieved. We can consider matrices like,

1. CPU utilization
2. Memory utilization
3. Custom matrices

For demonstration purposes CPU utilization has been used as the matrix. To increase resource usage load needs to be applied on the kubernetes cluster. Apache jmeter has been used for that as shown in figure 5.7 and 5.8.

When the load is applied on the system, CPU utilization increases. When the CPU utilization exceeds 200%, new pods are initialized as shown in figures 5.9, 5.10, 5.11.

When the load applied through jmeter is stopped, hpa removes the extra pods and take the number of pods back to minimum number of pods defined in the hpa definition.

In this way, auto scaling can be achieved in the defined kubernetes environment.

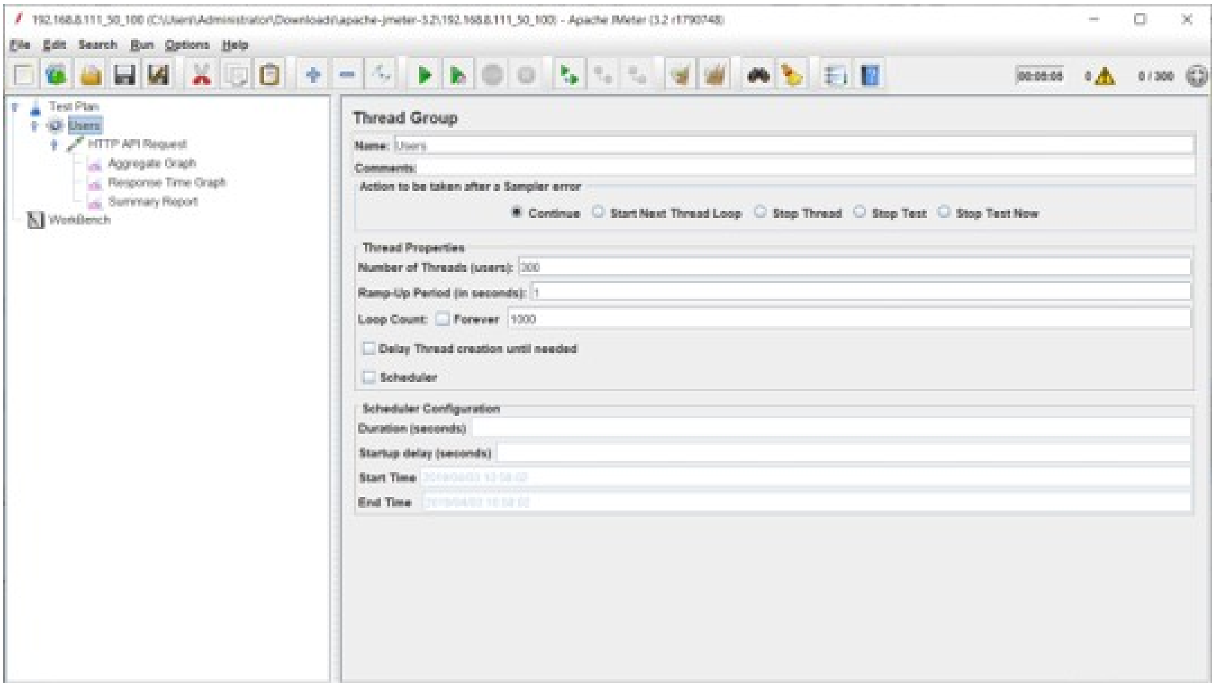


Figure 5.7: Apache jmeter set up to apply load

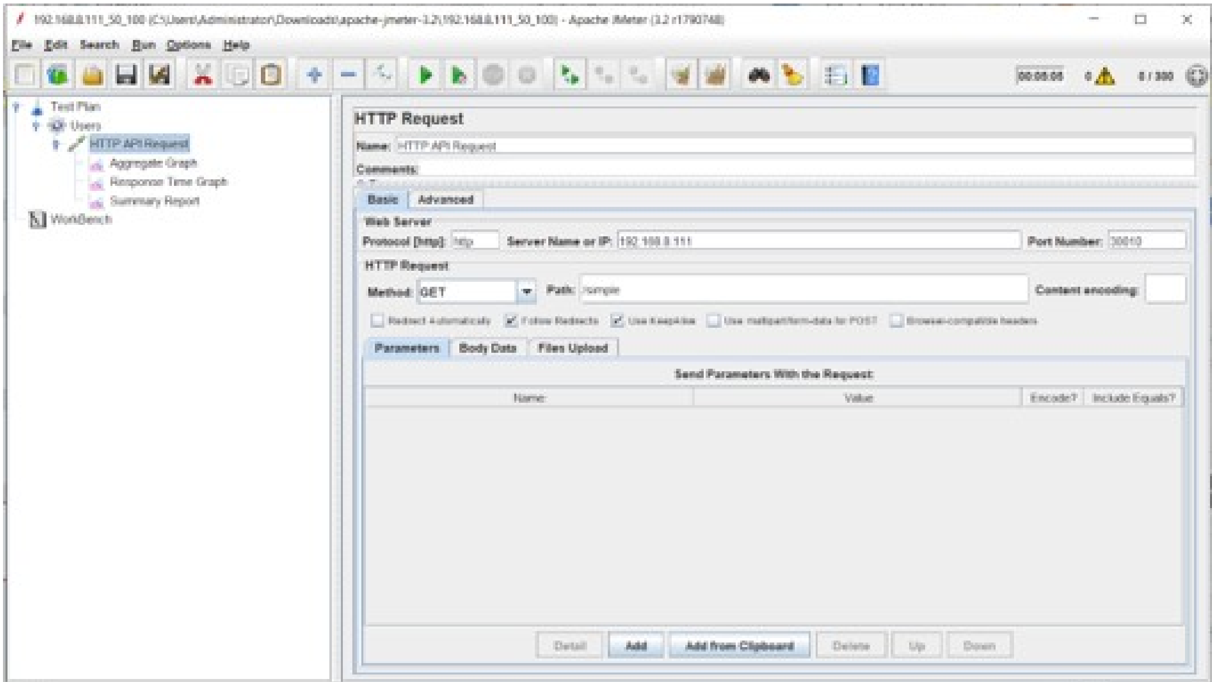


Figure 5.8: Apache jmeter set up to apply load

5.2 Excising Methodologies

The research was inspired by the current work my team is carrying out in my company. When there is an issue with an excising server, that will be identified through the monitoring system like kibana or a complaint may be received from a customer. Then DevOps team log into servers and look for the server which is malfunctioning. Once the server is identified, that server is

```
Every 2.0s: kubectl get all                                                                                    kubemaster: Fri Jun  5

NAME                                   READY   STATUS              RESTARTS   AGE
pod/ei-deployment-757665bdf5-72wng     0/1     ContainerCreating   0          5s
pod/ei-deployment-757665bdf5-85jjr     1/1     Running             0          4h28m

NAME                    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                         AGE
service/ei-service      NodePort    10.107.137.57   <none>        8280:30010/TCP,9443:30011/TCP   4h38m
service/kubernetes      ClusterIP   10.96.0.1       <none>        443/TCP                         12d

NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/ei-deployment   1/2     2            1           4h28m

NAME                                       DESIRED   CURRENT   READY   AGE
replicaset.apps/ei-deployment-757665bdf5   2         2         1       4h28m

NAME                                                REFERENCE                  TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/ei-deployment   Deployment/ei-deployment   246%/200%  1         4         1          146m
```

Figure 5.9: Initializing a new pod

```
Every 2.0s: kubectl get all                                                                  kubemaster:

NAME                                   READY   STATUS              RESTARTS   AGE
pod/ei-deployment-757665bdf5-574zt     0/1     ContainerCreating   0          5s
pod/ei-deployment-757665bdf5-72wng     1/1     Running             0          66s
pod/ei-deployment-757665bdf5-85jjr     1/1     Running             0          4h29m

NAME                    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                         AGE
service/ei-service      NodePort    10.107.137.57   <none>        8280:30010/TCP,9443:30011/TCP   4h39m
service/kubernetes      ClusterIP   10.96.0.1       <none>        443/TCP                         12d

NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/ei-deployment   2/3     3            2           4h29m

NAME                                       DESIRED   CURRENT   READY   AGE
replicaset.apps/ei-deployment-757665bdf5   3         3         2       4h29m

NAME                                                REFERENCE                  TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/ei-deployment   Deployment/ei-deployment   243%/200%  1         4         2          147m
```

Figure 5.10: Initializing a new pod

```
Every 2.0s: kubectl get all                                                                          kubemaster: Fri Jun

NAME                                   READY   STATUS    RESTARTS   AGE
pod/ei-deployment-757665bdf5-574zt     1/1     Running   0          80s
pod/ei-deployment-757665bdf5-72wng     1/1     Running   0          2m21s
pod/ei-deployment-757665bdf5-85jjr     1/1     Running   0          4h31m
pod/ei-deployment-757665bdf5-xqq2x     1/1     Running   0          19s

NAME                    TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                         AGE
service/ei-service      NodePort    10.107.137.57   <none>        8280:30010/TCP,9443:30011/TCP   4h40m
service/kubernetes      ClusterIP   10.96.0.1       <none>        443/TCP                         12d

NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/ei-deployment   4/4     4            4           4h31m

NAME                                       DESIRED   CURRENT   READY   AGE
replicaset.apps/ei-deployment-757665bdf5   4         4         4       4h31m

NAME                                                REFERENCE                  TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/ei-deployment   Deployment/ei-deployment   239%/200%  1         4         4          148m
```

Figure 5.11: Initializing a new pod

removed from the cluster manually. Then DevOps team analyze logs to identify the problem with the server, fix it and attach back to cluster. That is the normal procedure used in any of the *on-prem* server farm / distributed system.

5.3 Comparison of proposed solution and existing solution

There are several matrices to consider when comparing the two systems for pros and cons.

```
Every 2.0s: kubectl get all                                                                                    kubemaste

NAME                                READY   STATUS        RESTARTS   AGE
pod/ei-deployment-757665bdf5-574zt   1/1     Terminating   0          11m
pod/ei-deployment-757665bdf5-72wng   1/1     Running       0          12m
pod/ei-deployment-757665bdf5-85jjr   1/1     Running       0          4h40m
pod/ei-deployment-757665bdf5-xqq2x   1/1     Terminating   0          10m

NAME                 TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                         AGE
service/ei-service   NodePort    10.107.137.57   <none>        8280:30010/TCP,9443:30011/TCP   4h50m
service/kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP                         12d

NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/ei-deployment   2/2     2            2           4h40m

NAME                                       DESIRED   CURRENT   READY   AGE
replicaset.apps/ei-deployment-757665bdf5   2         2         2       4h40m

NAME                                                 REFERENCE                  TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/ei-deployment   Deployment/ei-deployment   3%/200%   1         4         2          158m
```

Figure 5.12: Deleting pods

```
Every 2.0s: kubectl get all                                                                                    kubemaster

NAME                                READY   STATUS        RESTARTS   AGE
pod/ei-deployment-757665bdf5-72wng   1/1     Terminating   0          13m
pod/ei-deployment-757665bdf5-85jjr   1/1     Running       0          4h41m

NAME                 TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)                         AGE
service/ei-service   NodePort    10.107.137.57   <none>        8280:30010/TCP,9443:30011/TCP   4h51m
service/kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP                         12d

NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/ei-deployment   1/1     1            1           4h41m

NAME                                       DESIRED   CURRENT   READY   AGE
replicaset.apps/ei-deployment-757665bdf5   1         1         1       4h41m

NAME                                                 REFERENCE                  TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
horizontalpodautoscaler.autoscaling/ei-deployment   Deployment/ei-deployment   4%/200%   1         4         2          159m
```

Figure 5.13: Deleting pods

5.3.1 Time to recover

Time for the recovery is the most crucial factor when it comes to an outage of a production system. When look at this from client's perspective, this is leakage of revenue. From manage service team's perspective, this can impact the impact the SLAs and finally result in penalties. When the proposed solution is used, system recovers within less than a minute(only depends on the time to spin up a new pod).

When it comes to an manually managed typical server farm, this will take 10 minutes or more for the recovery.

So, there is a good advantage to apply the proposed solution when considered from the perspective of time to recovery.

5.3.2 Human intervention

In a production outage human intervention is not required when the suggested solution is applied. If the manual method is used, human intervention is needed to login to the system. Human

intervention comes with a cost as well. So, with the suggested solution, we can get rid of human intervention to attend to the issue and saves revenue as well.

5.3.3 Root cause analysis

Once there is an production outage, manage service team is required to come up with the root cause. Root cause analysis is mostly carried out though analyzing logs. Identify and distinguish between logs is not straightforward in a kubernetes setup if that in not previously defined.

When it comes to on-prem servers, there are log locations within the servers. Even if the logs are rotated, they are properly structured and easy to find. In that way the root cause analysis may be easy with on-prem servers while it is bit challenging with kubernetes.

Chapter 6: Conclusion and Future Work

6.1 Future work

When it comes to future work, there is a lot remaining in this area when going forward. In my study I have considered an *on-prem* distributed system. I didn't touch cloud computing much in this research. So, in future someone can start from cloud computing which is getting much popularity these days.

In my research I considered from the perspective of the virtual machine(container /pod). May be in the next phase, application level analysis like log analysis, status code analysis can be added for identifying systems state. That would be much accurate rather than depending of the system matrices like cpu usage and memory usage.

How to determine self healing without using kubernetes will be a good area to look at. May be someone can come up with a better tool to address the self-healing problem.

6.2 Conclusion

As described in chapter 5, we can come up with a self healing vm (container) cluster with the proposed kubernetes setup. So, here we have achieved the main objective. Taking few steps forward from the initially set goal, we have achieved an auto scaling cluster of vms which has the ability to scale up and down depending on the matrices set like cpu usage or memory usage.

Furthermore I'm planning to apply the same for the project I'm currently working at company to achieve better results in term of recovery time when there is an issue in a server.

References

[1] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Architecture-centric programming for adaptive systems. In *Proceedings of the first workshop on Self-healing systems*, pages 93–95, 2002.

[2] A. Avizienis, G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr, and D. K. Rubin. The star (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design. *IEEE Transactions on Computers*, 100(11):1312–1321, 1971.

[3] M. F. Buckley and D. P. Siewiorek. Vax/vms event monitoring and analysis. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 414–423. IEEE, 1995.

[4] S.-W. Cheng, D. Garlan, B. Schmerl, P. Steenkiste, and N. Hu. Software architecture-based adaptation for grid computing. In *Proceedings 11th IEEE International Symposium on High Performance Distributed Computing*, pages 389–398. IEEE, 2002.

[5] N. Combs and J. Vagle. Adaptive mirroring of system of systems architectures. In *Proceedings of the first workshop on Self-healing systems*, pages 96–98, 2002.

[6] C. Dabrowski and K. Mills. Understanding self-healing in service-discovery systems. In *Proceedings of the first workshop on Self-healing systems*, pages 15–20, 2002.

[7] S. Forrest, S. A. Hofmeyr, and A. Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, 1997.

[8] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[9] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, pages 27–32, 2002.

[10] D. Garlan, B. Schmerl, and J. Chang. Using gauges for architecture-based monitoring and adaptation. 2001.

[11] S. George, D. Evans, and S. Marchette. A biological programming model for self-healing. In *Proceedings of the 2003 ACM workshop on Survivable and self-regenerative systems: in association with 10th ACM Conference on Computer and Communications Security*, pages 72–81, 2003.

[12] D. Ghosh, R. Sharman, H. Rao, and S. Upadhyaya. Self-healing systemssurvey and synthesis. *Decision Support Systems*, 42:2164–2185, 2007.

[13] E. Grishikashvili. Investigation into self-adaptive software agents development. *Distributed Multimedia Systems Engineering Research Group Technical Report*, 2001.

[14] Y. Hong. Closed loop design for software rejuvenation. In *Proc. Workshop on Self-Healing, Adaptive, and Self-Managed Systems, 2002*, 2002.

[15] M. N. Huhns, V. T. Holderfield, and R. L. Z. Gutierrez. Robust software via agent-based redundancy. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 1018–1019, 2003.

[16] G. Kaiser. Autonomizing legacy systems. In *invited talk at the Almaden Institute Symposium on Autonomic Computing, April*, pages 10–12, 2002.

[17] M. Knop, J. Schopf, and P. Dinda. Windows performance monitoring and data reduction using watchtower. In *11th IEEE Symposium on High-Performance Distributed Computing (HPDC11)*, volume 35, 2002.

[18] I. Lee, R. K. Iyer, and D. Tang. Error/failure analysis using event logs from fault tolerant systems. In *[1991] Digest of Papers. Fault-Tolerant Computing: The Twenty-First International Symposium*, pages 10–17. IEEE, 1991.

[19] T.-T. Lin and D. P. Siewiorek. Error log analysis: statistical modeling and heuristic trend analysis. *IEEE Transactions on reliability*, 39(4):419–432, 1990.

[20] M. G. Merideth. Enhancing survivability with proactive fault-containment. In *DSN Student Forum, Citeseer*, volume 20. Citeseer, 2003.

[21] M. G. Merideth and P. Narasimhan. Proactive containment of malice in survivable distributed systems. In *Security and Management*, pages 3–9. Citeseer, 2003.

[22] R. Nagpal, A. Kondacs, and C. Chang. Programming methodology for biologically-inspired self-assembling systems. In *AAAI Spring Symposium on Computational Synthesis*, pages 173–180, 2003.

[23] V. P. Nelson. Fault-tolerant computing: Fundamental concepts. *Computer*, 23(7):19–25, 1990.

[24] R. K. Sahoo, M. Bae, R. Vilalta, J. Moreira, S. Ma, and M. Gupta. Providing persistent and consistent resources through event log analysis and predictions for large-scale computing systems. In *Workshop on Self-Healing, Adaptive, and Self-Managed Systems*, 2002.

[25] B. Satzger. Self-healing distributed systems. 2008.

[26] A. S. Tanenbaum and M. Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[27] M. M. Tsao. Trend analysis and fault prediction. 1984.

[28] G. Valetto and G. Kaiser. A case study in software adaptation. In *Proceedings of the first workshop on Self-healing systems*, pages 73–78, 2002.