

POC OF KI-NGĀ-KŌPUKU SYSTEM

A THESIS SUBMITTED TO AUCKLAND UNIVERSITY OF TECHNOLOGY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF COMPUTER AND INFORMATION SCIENCES

Supervisors

Dr Alan T Litchfield

July 2017

By

Yuzhu Chen

School of Engineering, Computer and Mathematical Sciences

Copyright

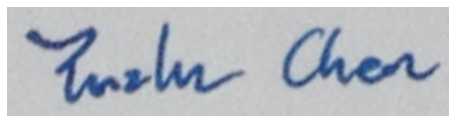
Copyright in text of this thesis rests with the Author. Copies (by any process) either in full, or of extracts, may be made **only** in accordance with instructions given by the Author and lodged in the library, Auckland University of Technology. Details may be obtained from the Librarian. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the Author.

The ownership of any intellectual property rights which may be described in this thesis is vested in the Auckland University of Technology, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

Further information on the conditions under which disclosures and exploitation may take place is available from the Librarian.

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the qualification of any other degree or diploma of a university or other institution of higher learning.

A photograph of a handwritten signature in blue ink on a light-colored surface. The signature appears to read "Zuzhi Chen".

Signature of candidate

Acknowledgements

I would first like to thank my supervisor Dr Alan Litchfield of the SECMS of the Auckland University of Technology, who is always helping me with my thesis whenever I ran into a trouble spot or had a question about my research or writing. I would also like to thank the expert who build up the research basement of my thesis: Monjur Ahmend, who is my research corporator and helped me during the research. Finally, I would like to thank my parents and the cast of thousands who supported me in my journey of discovery. Thank you.

Abstract

With the rapid development of technology, most of people and companies have chosen to develop services in the cloud environment. As the cloud computing technology become more and more popular, security and performance issues are considered to be important parts to be concerned. Applications that are implemented into the cloud environment still face many challenges, such as easy to be exposed and tracked, need extra cloud solutions for fault recovery, etc.

In this project, we want to proof the concept of a newly proposed distributed security system, and use the proofed concept to develop a decentralized distributed security system in the cloud environment, which brings absolute redundancy, availability, and fault-tolerant ability to user's applications. In this research, Design Science Research (DSR) methodology and Rapid Application Development (RAD) methodology are both used during the whole research process. Within the DSR research circle, RAD takes the role to lead the prototype development, and reflect the potential requirements and problems back to DSR for further theory support. This research covers cloud security problems, distributed systems, and also decentralized systems. Based on the research in these fields, a great theory background of this prototype system is established. This prototype system will act like a management framework when working with user's applications. The only thing that the developers should consider is how to fit their application into it. Moreover, since this framework aims to provide a generic solution for different purpose (security, serving customer applications, etc.), so there will be

less boundaries when developers try to choose what technologies should work with this framework. As considered to be a security system provided to users, it focuses on providing an extensible way to achieve application security and infrastructure security, which makes developers be eligible to implement any security mechanisms and solutions into it. As considered to be a decentralized distributed system, it is able to be resilient and no single point of failure, so that developers can focus on developing their products by following the rules of the system: Ki-Ngā-Kōpuku. The redundancy and availability of applications are handled by Ki-Ngā-Kōpuku automatically.

The limitations of this research are the big research scope and limited research time. Ki-Ngā-Kōpuku is a really big project, which requires enough time and various technologies to accomplish. It is hard to achieve the expected output within a limited time. Moreover, the selected research methodology (Design Science) doesn't fit this research perfectly. As a result, another research methodology (Rapid Application Development) is taken into this research and work with Design Science research methodology.

In general, Ki-Ngā-Kōpuku will distribute application's components into different network locations, and implement security mechanisms to the system. In this research, the system architecture of Ki-Ngā-Kōpuku has been designed, some basic and critical part of the system are also developed in Erlang programming language, such as component distribution, and component communication. Moreover, the ideal environment and limitations of Ki-Ngā-Kōpuku are also discussed in this research.

Contents

Copyright	2
Declaration	3
Acknowledgements	4
Abstract	5
1 Introduction	12
1.1 Background	12
1.2 Vision	13
1.2.1 Ki-Ngā-Kōpuku Basic Concept	14
1.2.2 Ki-Ngā-Kōpuku System Concept	15
1.3 Approach	15
1.4 Research Questions & Challenges	16
1.5 Contributions	17
1.6 Outline	18
1.7 Conclusion	19
2 Literature Review	21
2.1 Introduction	21
2.2 Cloud Computing Security	22
2.2.1 Security Requirements	22
2.2.2 Cloud Threats	23
2.2.3 Security Models in Cloud Computing	24
2.2.4 Data Security in Cloud Computing	26
2.3 Distributed System	33
2.3.1 Fault-Tolerant	35
2.3.2 Load Balancing	38
2.4 Decentralized System	42
2.5 Random Distribution	45
2.6 Conclusion	46

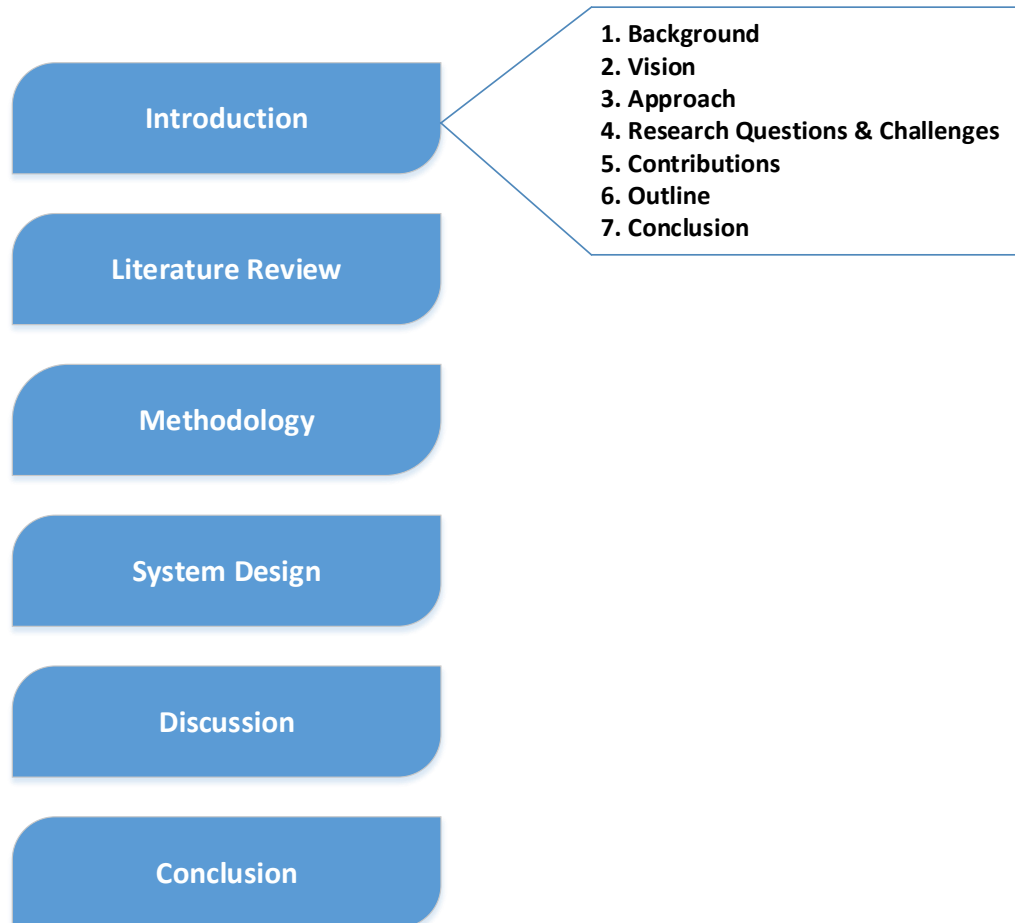
3	Method	49
3.1	Introduction	49
3.2	Methodology	50
3.2.1	Design Science Research Methodology	50
3.2.2	Rapid Application Development	52
3.3	Research Design	53
3.3.1	Research Goals	54
3.3.2	Research Process	54
3.3.3	Research Evaluation Criteria	56
3.4	Research Limitations	57
3.5	Hypothesis	59
3.6	Possible Solutions	60
3.7	Conclusion	62
4	System Design	64
4.1	Introduction	64
4.2	Ki-Ngā-Kōpuku Brief Introduction	65
4.2.1	System Features	65
4.2.2	System Requirements	65
4.2.3	Programming Language	66
4.3	Ki-Ngā-Kōpuku System Design	68
4.3.1	Ki-Ngā-Kōpuku System Cluster	68
4.3.2	Application Group	70
4.3.3	Network Topology	70
4.3.4	System Architecture	71
4.4	Ki-Ngā-Kōpuku System Interface	73
4.4.1	System Action	74
4.5	Traffic Handler	78
4.5.1	General Process	78
4.5.2	Message Pattern	78
4.6	Application Components	81
4.6.1	Component Definition	81
4.6.2	Component Storage	82
4.6.3	Component Action	84
4.6.4	Component Communication	88
4.7	Component Distribution	93
4.7.1	Distribution Factors	94
4.7.2	General Distribution Steps	95
4.7.3	Distribution Workflow	97
4.8	Conclusion	104

5	Discussion	107
5.1	Introduction	107
5.2	Summary of Related Work	108
5.2.1	Cloud Security Models	108
5.2.2	Data Security in Cloud Computing	109
5.2.3	Distributed System	110
5.2.4	Decenralized System	112
5.2.5	Random Distribution	113
5.2.6	Conclusion	113
5.3	Summary of Ki-Ngā-Kōpuku System	115
5.3.1	Erlang Programming Language	115
5.3.2	System Design	116
5.3.3	System Communication	117
5.3.4	Application Component	118
5.3.5	Component Distribution	119
5.3.6	Distribution Status	120
5.3.7	Conclusion	120
5.4	Research Questions	121
5.4.1	Hypothesis	121
5.4.2	Discussion	122
5.4.3	Conclusion	129
5.5	Design Evaluation	129
5.5.1	Goal Evaluation	130
5.5.2	Environment Evaluation	130
5.5.3	Structure Evaluation	131
5.5.4	Activity Evaluation	131
5.5.5	Evolution Evaluation	131
5.5.6	Evaluation Conclusion	132
5.6	System Limitations	132
5.6.1	System Environment	132
5.6.2	Server Network Interface Single-Point-Of-Failure (SPF) . . .	133
5.6.3	Distribution Times	134
5.6.4	Malicious Components	135
5.6.5	Communication Efficiency	135
5.7	Conclusion	136
6	Conclusion	138
6.1	Introduction	138
6.2	Research Summary	138
6.3	Future Work	140
6.4	Conclusion	141
	Appendices	143

List of Figures

3.1	Rapid Application Development Methodology (<i>Ramsoft Consulting</i> , n.d.)	53
3.2	Hierarchy of criteria for IS artifact evaluation (Prat, Comyn-Wattiau, & Akoka, 2014)	56
3.3	Evaluation Criteria	57
4.1	Ki-Ngā-Kōpuku Supervision Tree	68
4.2	Basic Network Topology	71
4.3	Basic System Architecture	72
4.4	Basic Communication Architecture	73
4.5	Brief Start Steps	76
4.6	Traffic Handler Process	79
4.7	Component Type Identification	85
4.8	Component Registry Approvement	86
4.9	Register Component via Types	89
4.10	Component Registry Message Processing	90
4.11	Component Communication	92
4.12	General Distribution Steps	95
4.13	Distribution Requestor Workflow	98
4.14	Distribution Sender Workflow	101

CHAPTER : INTRODUCTION



Chapter 1

Introduction

1.1 Background

Currently, cloud computing is playing an important role in our daily lives. Most of the services are moving to the cloud environment, and services are becoming increasingly convenient and intelligent due to the fast development of the cloud environment. Moreover, increasingly, companies and organisations are changing their service structures from locally based to cloud based, which are easy to manage. A number of key characteristics of cloud computing have been identified (Zissis & Lekkas, 2012):

1. Flexibility / Elasticity
2. Scalability of infrastructure
3. Broad network access
4. Location independence
5. Reliability
6. Economies of scale and cost effectiveness
7. Sustainability

Based on the benefits from cloud computing, according to (*Internet Security Threat*

Report, 2017), from July to December in 2016, the average number of cloud apps used per organization are 928, and 25% of data are broadly shared. However, the fast-growing cloud environment also faces more problems, such as security, unstable environment, and enormous cost. As reported from (*Internet Security Threat Report*, 2017), it takes only 2 minutes for an IoT device to be attacked, and the number of attacks against Symantec honeypot per hour is 9. Even more, (*Cisco 2017 Annual Cybersecurity Report*, 2017) shows 57% of security professionals concern about the cloud environment is the target of Cyber Attacks. (Koshan, 2015) also shows web application attacks, brute force attacks, and vulnerability scans each impact 44% of the cloud hosting customer impact.

Having applications become distributed is a trend in current computing field, a few of distributed computing frameworks are provided for developers, such as The Message Passing Interface (MPI), Hadoop, and Spark (maxdml, 2017). However, some of these frameworks can't guarantee a stable environment for users' applications because of its own design. Furthermore, cloud providers, such as Windows Azure, and Amazon Web Service (AWS), provide their own Hadoop solution as a distributed computing service for customers, which may arise cost concern when customers start using it.

In general, efficient tools that can bring security, high availability, and redundancy to applications are required by developers and customers.

1.2 Vision

To bring security, redundancy, and availability to user's applications, Ki-Ngā-Kōpuku is introduced in this research thesis. Ki-Ngā-Kōpuku is a decentralized distributed security system, which works like a framework that can distribute application components into different network locations, and help developers manage components in the whole network environment without centralised controllers. Ki-Ngā-Kōpuku also implements

its own security mechanisms to protect any system actions related to application components, preventing malicious operations affect the whole system. By implementing Ki-Ngā-Kōpuku, users' applications won't experience single point of failure, or hard to be being compromised.

This research thesis intends to proof the concept of Ki-Ngā-Kōpuku in both theoretical and technical way, and prove the value and availability to the cloud computing and industry environment. The concept of Ki-Ngā-Kōpuku is that distributing application components can improve application's fault tolerant, redundancy, and availability. In order to proof the concept, the application component concept has to be clarified first. Then each application component has to be able to be replicated and distributed to different places. After that, each component should work properly, and multiple replicas of each component should exist in the whole network. The most important part is that when a component fails, its replica can take over the task and carry on the computation task.

To better understand what Ki-Ngā-Kōpuku is, the next two subsections describe the theoretical backup concept of Ki-Ngā-Kōpuku, and also the concept that directs in the system development.

1.2.1 Ki-Ngā-Kōpuku Basic Concept

The idea of Ki-Ngā-Kōpuku comes from a distributed security model which is first addressed in Litchfield, Ahmed, and Sharma (2016). The addressed distributed security model illustrates one distributed security system which has great fault-tolerant ability and scalability by achieving random component distribution in the whole network.

In real life, a system is less prone to successful attack if it is hard to discern and difficult to take down. It is also a tough task for a system to perform online repair while it is being attacked. As a result of that, Ki-Ngā-Kōpuku focuses on maintaining stability

and performance of the overall system. To do that, the architecture is decentralized (avoiding single-point-of-failure problem), distributed (improving system performance and reducing computation resources pressure), highly redundant (meaning that failure problems of any part of the system won't affect the rest of the system), and components are randomly distributed (meaning that the system configuration is hard to discern).

1.2.2 Ki-Ngā-Kōpuku System Concept

As mentioned in the previous subsection, Ki-Ngā-Kōpuku is decentralized, and applying its own mechanism to distribute components to achieve the system goal. To bring this concept into production, the component mentioned in the concept will be application components, so that users can provide application's components to Ki-Ngā-Kōpuku, and it is able to randomly distribute application's components to make sure the redundancy of user's applications.

In general, using Ki-Ngā-Kōpuku can create multiple replicas of application's components in the network environment, therefore, applications won't have single-point-of-failure problems because if one component fails, then there will be another active component running. Moreover, security mechanisms implemented with in Ki-Ngā-Kōpuku is able to secure communication data between different components, and data that stored in network locations.

1.3 Approach

To develop a system that provides security, redundancy and availability to users' applications, the cloud environment is studied first. This includes the cloud infrastructure, cloud security issues, and existing cloud solutions. Then, distributed and decentralised systems are studied because Ki-Ngā-Kōpuku is supposed to be designed as a distributed system with decentralised system features. After that, Erlang programming language is

chosen to develop Ki-Ngā-Kōpuku. The reason for choosing Erlang as the development language is its value for the distributed system, and its usefulness for dealing with massive concurrency traffic. On top of that, users are required to undertake application componentisation before importing applications into Ki-Ngā-Kōpuku. After that, Ki-Ngā-Kōpuku can replicate application components and distribute components into different locations. Thus, it is possible for one application component to have multiple replicas in the network, and each component replica can serve application users, which gives applications redundant ability and high availability. For providing security to user's applications, encryption algorithms for transmission data, and data security architectures should be imbedded into Ki-Ngā-Kōpuku.

The Design Science Research (DSR) methodology is applied to the whole research process. Application domains are identified firstly, then relevant literature research fullfills the knowledge base of this project. After that, based on the existing knowledge, Ki-Ngā-Kōpuku system modules are developed. For developing Ki-Ngā-Kōpuku technically, the Rapid Application Development (RAD) methodology is chosen to be used. Problems and new findings coming from system development process is reflected to DSR, and being used in the evaluation process in DSR.

1.4 Research Questions & Challenges

The following are the research questions proposed in this research:

1. How to improve security among components?
2. How to design a component distribution mechanism to distribute components into several servers?
3. How to maintain communication between components which are located on different servers?

There are several challenges to researching the approach; these include:

1. How to use Erlang to design a distributed system?
2. How to address theoretical and technical gaps?
3. How to define replicas of application components in Ki-Ngā-Kōpuku?
4. How to make application components work as a whole application when each component is in different locations?
5. How to provide redundancy and availability to users' applications?
6. How to provide security mechanisms to user's applications?

Some challenges, such as using Erlang to design a distributed system, can be met with existing solutions such as using Erlang OTP design principles and nodes features to design an Erlang-based distributed system. However, while using one programming language, we still have to solve language problems. This paper also represents proof of concept research, which is based on existing theoretical models, so sometimes the current chosen technology tools are not able to achieve theoretical goals, and the research result is limited by the author's technical skills level or the overall project scope.

This research addresses the above challenges to move closer towards proving the concept and bringing the project to life.

1.5 Contributions

This research makes three contributions.

Firstly, This research proves the feasibility of Ki-Ngā-Kōpuku and its basic concept in both theoretical and technical way. Ki-Ngā-Kōpuku is based on a distributed security system which is proposed in Litchfield et al. (2016). The distributed security system uses component distribution to achieve redundancy and availability, which is totally

new to the cloud computing. In this research, the concept of the distributed security system and Ki-Ngā-Kōpuku are both proofed.

Secondly, this research designs Ki-Ngā-Kōpuku which is a decentralized distributed security framework for user's applications. It uses the distribution features of Erlang programming language to build up multiple self-manageable nodes, and group each Ki-Ngā-Kōpuku node to form a stable environment for user's applications. Each Ki-Ngā-Kōpuku node is able to exchange information with other nodes or its application components, and, for redundancy purpose, it can randomly distribute application components to other Ki-Ngā-Kōpuku nodes in different network locations. By using Ki-Ngā-Kōpuku, application components are replicated and randomly distributed among the network, which brings redundancy, availability and reliability to user's applications.

Thirdly, this research proposes using Linux ssh and rsync tools to transfer compiled code files to different network locations for achieving component distribution in Ki-Ngā-Kōpuku. The hash value of compiled code files will be checked to ensure the integrity.

1.6 Outline

The remainder of this research thesis is organised as follows:

Chapter 2 presents related work and overall background research for this thesis. This includes the cloud environment background, fault-tolerant technologies implemented in the cloud, cloud security issues, distributed systems, decentralised systems, and the theory that supports this research.

Chapter 3 describes the two research methodologies used in this research. One is the design science research methodology. Another is the rapid application development methodology, which is used for the technical aspects of the project. This chapter also proposes hypotheses related to the research questions and corresponding solutions.

Chapter 4 introduces details of Ki-Ngā-Kōpuku, which include the system design, system architecture, system actions, and system behaviours.

Chapter 5 is the discussion chapter, which discusses the major findings in chapter 4, literature review in chapter 2, and whether or not these findings can help solve research questions and prove the hypotheses. Moreover, contributions of this research thesis, system limitations and problems are also mentioned in this chapter.

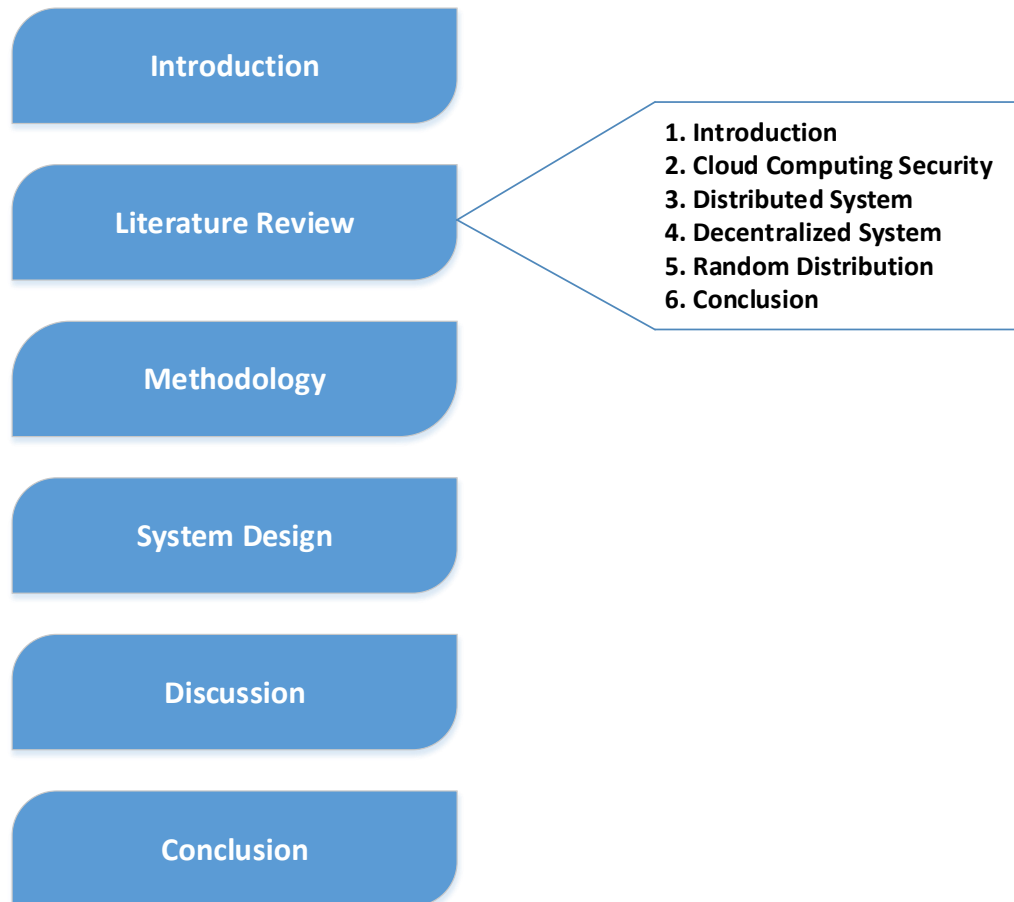
Finally, Chapter 6 summarises this research, and discuss extensions and future work.

1.7 Conclusion

This chapter has described the purpose of this research, the concept of Ki-Ngā-Kōpuku, approaches to achieve the system goals in both theoretical and technical ways, challenges that might have during the research, and contributions to the existing knowledge body.

To find possible solutions to research questions and challenges, the next chapter 2 has done the research on multiple related articles including cloud computing security, distributed system, decentralized system, and component distribution.

CHAPTER : LITERATURE REVIEW



Chapter 2

Literature Review

2.1 Introduction

In chapter 1, background of Ki-Ngā-Kōpuku and challenges are discussed briefly. As described before, Ki-Ngā-Kōpuku is based on one newly proposed system concept, which has no related models or productions invented before, so all aspects of Ki-Ngā-Kōpuku are researched for theoretical backup and seeking the success in technical achievement.

In this chapter, section 2.2 describes the security mechanisms in the cloud computing, and security solutions that are implemented in the cloud environment. In section 2.3, distributed system is studied, especially the common techniques that are used in the distributed system. Decentralized system is studied in section 2.4. In section 2.5, related distribution design and techniques are discussed. The last section 2.6 gives an overall conclusion of all contents mentioned in this chapter, and research questions are also proposed in this section.

2.2 Cloud Computing Security

This section describes the security of the cloud environment, including security requirements, security threats, security models, and data security.

Cloud computing is an advanced technology and its popularity makes it a target of network attacks. In that case, the security of cloud computing has to be researched in several aspects, such as security requirements, network attacks, resolutions for network attacks, and also security models implemented in the cloud environment. By researching the current security situation for cloud computing, Ki-Ngā-Kōpuku is able to use advantages of existing security solutions, and also find possible ways to deal with drawbacks of current cloud environment.

2.2.1 Security Requirements

Cloud computing is one popular technology that people commonly use in daily life. For developing systems on cloud environment, security requirements must be clarified in the first place for setting up minimal goals for the future system development. The security is the main factor in cloud environment. There are six security requirements (Lombardi & Di Pietro, 2011):

Identification and Authentication Assurance of identity of person or originator of data.

Authorization Unauthorised persons can't reach data.

Integrity Maintaining and assuring data consistency and accuracy.

Non-repudiation Originator of communications can't deny it later.

Availability Legitimate users have access when they need it

Access Control Legal users can be authorized by the owner to access the data.

Furthermore, identification, authentication, and authorization (IAA) is represented

as the cyclic security principles applied to service provisions, transactions and almost every action (Chang, Walters, & Wills, 2015).

Lombardi and Di Pietro (2011) has clarified the security requirements of cloud computing. Each of the system requirement leads the direction of each security solutions. As for Ki-Ngā-Kōpuku, the above six security requirements should all be satisfied, especially identification and authentication, integrity, and availability. Having the identification and authentication requirement satisfied can reduce malicious users. Integrity and availability are also the features that Ki-Ngā-Kōpuku must have.

2.2.2 Cloud Threats

After knowing what the security requirements the system must achieve, threats to the cloud environment have to be analyzed as well. Moreover, the cloud security protection is based on security issues in the Cloud Service Delivery Models (Fernandes & Soares, 2014), so that different security solutions are implemented according to different models. The three delivery service models are the Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

Indeed, normal cloud computing architecture will cause a lot of security problems. The Service Provider (SP) uses the resources provided by the Cloud Provider (CP), which leads to Service Users (SU) and SP have no physical control over cloud servers (Lombardi & Di Pietro, 2011). Possible attacks against the cloud system can be classified as follows (Smith et al., 2006):

1. Resource attacks against CPs
2. Resource attacks against SPs
3. Data attacks against CPs
4. Data attacks against SPs
5. Data attacks against SUs

Dan, Michael, et al. (2010) also mentioned seven top threats to cloud computing:

1. Abuse and nefarious use of cloud computing
2. Insecure interfaces and APIs
3. Malicious insiders
4. Shared technology issues
5. Data loss or leakage
6. Account or service hijacking
7. Unknown risk profile

Security protection methods can be different according to different service delivery models. IaaS is the bottom model which provides VMs to customers. Firewalls, Intrusion Detection Systems (IDS), Intrusion Prevention Systems (IPS), and Load Balancing are the basic security protection methods for IaaS. Instead, Virtual Machine Monitors (VMMs) is the critical component in cloud computing (Fernandes & Soares, 2014). The VMMs should be isolated from VMs. Moreover, Anything-as-a-Service (XaaS) is also one important part of service delivery model, but the security methods depend on what exactly the content of XaaS is.

The above literature all listed threats to cloud computing. However, there is one point that haven't take into consideration, which is the cloud providers itself. As described before, only cloud providers have the access to hardware resources, so there is no access limitation to cloud providers. As a result, users should put the cloud providers to untrusted party.

2.2.3 Security Models in Cloud Computing

In this section, cloud security models and other general security methods are discussed.

There are four cloud computing security models analyzed in Che, Duan, Zhang, and Fan (2011): the Cloud Multiple-Tenancy Model of NIST (Brunette, Mogull, et al.,

2009), the Cloud Risk Accumulation Model of CSA (Brunette et al., 2009), Jericho Forum's Cloud Cube Model (Formu, 2009), and the Mapping Model of Cloud, Security and Compliance (Brunette et al., 2009). Cloud Multiple-Tenancy Model of NIST allows multiple applications share the same computing resources, such as processes, storage, and memory. However, the technology difficulties of multiple-tenancy model are data isolation, architecture extension, configuration self-definition, and performance customization. The Cloud Risk Accumulation Model of CSA is a layer dependent model, which is consisted of SaaS, PaaS, and IaaS. SaaS is built on PaaS, and IaaS is the foundation of PaaS. As described in this research paper (Che et al., 2011), the lower service layer that a cloud service provider lies in, the more management duties and security capabilities that a customer is in charge of. As a result of that, this model doesn't guarantee the security on the cloud environment because it requires cloud providers and customers to make up the security gap in some circumstances. Jericho Forum's Cloud Cube Model is a description of security attributes information implied in the service and deployment models of cloud computing and the location, manager and owner of computing resources and so on. On the other hand, Jericho Forum's Cloud Cube Model uses multiple parameters to define the security level of cloud services in theoretical level (Formu, 2009). The Mapping Model of Cloud, Security and Compliance (Brunette et al., 2009) presents a good method to analyze the gaps between cloud architecture and compliance framework and the corresponding security control strategies. In general, the above four security models can be applied to evaluating the security level of cloud computing, and providing basic techniques to cloud development. The above four cloud computing security models have analyzed cloud security in multiple perspectives. However, there is no single model which combines these four security models, so that users can get all security information by applying one single model. Besides, (Che et al., 2011) also lacks experimental analysis or evaluation to each cloud computing security model.

Jing and Jian-jun (2010) introduced one security concept called Security Access Control Service (SACS). It includes Access Authorization, Security API, and Cloud Connection Security. User's identification is verified in Access Authorization layer; Security API keeps users use specific services safely after accessing to the cloud; Cloud Connection Security ensures the safety of resources provided by the bottom resource layer. In this case, the Access Authorization is located on the top of service layer, Security API lies between SaaS and PaaS in service layer, and Cloud Connection Security is placed between service layer and resource layer. This security model is ideal for the cloud computing because it provides three security layers and can have more stable performance. The SACS described in Jing and Jian-jun (2010) considers cloud security from higher level to lower level. However, only Security APIs can be used by users will not only limit system scalability, but also put more pressure on API key management. Furthermore, authors didn't consider API abuse and malicious usage can also harm the system, so monitoring user activities should be placed in this model. On the other hand, Cloud Connection Security may limit backend response time due to security check to both user identification and requested data.

2.2.4 Data Security in Cloud Computing

In this section, data security in cloud computing is discussed. Data security is the crucial part of cloud computing, as described in Sathyanarayana and Sheela (2013), data security becomes more important when using cloud computing at all "levels": IaaS, PaaS, SaaS. Furthermore, data security is the key component to ensure the popularity of cloud computing (Xin, Song-qing, & Nai-wen, 2012). Data can be divided into three types (Chang et al., 2015; "Enhanced data security model for cloud computing.", 2012): data at rest (storage data), data in transit (transmission data), and data in processing (processing data). For securing various types of data in the cloud environment, cloud providers use

symmetric encryption to encrypt data in storage, homomorphic encryption to encrypt processing data, and secret socket layer to encrypt transmission data. However, two issues about data security were issued in Mackay, Baker, and Al-Yasiri (2012):

1. The data encryption technology might affect the service performance
2. The security of shared resources

The above issues show the necessary to design efficient data encryption technology that cost few computing resources. However, data encryption and decryption don't consume too much resources, and the need for encrypting and decrypting data is not so frequent, so the first issue mentioned in Mackay et al. (2012) won't happen in common scenarios, which don't have so many data interaction. The second security issue of shared resources is really important. Protecting shared resources is efficient to guarantee data availability and data integrity. But restricting resources to be shared can reduce the security risk from the source.

Security for Data-at-rest

Chen and Zhao (2012) describes three security aspects that should be considered for the data stored in cloud storage: confidentiality, integrity, and availability. Data encryption is the common solution for data confidentiality. However, encryption algorithms, key strength, and key management have to be considered. Data integrity is challenged while users and cloud servers process massive data in the cloud storage. Data availability is limited by cloud services and providers because the cloud environment is not guaranteed to be safe and stable.

Data at rest normally means the data that is stored in a readable form on a cloud computing service (Sedayao, Su, Ma, Jiang, & Miao, 2009). This article also proposed using public key encryption to insure the confidentiality of data at rest. Its method is using a trusted collection agent's public key to encrypt data in memory layer, so that encrypted data is stored in storage. Once processes fetch encrypted data at rest, processes

use trusted collection agent's private key to decrypt data. The method mentioned in this article can protect data in some ways, but it requires some preconditions to be met in order to achieve data security. First of all, no snoop on memory content. If the original data in memory layer is modified by hackers before processes fetching, then this method won't work. Second precondition is that those trusted collection agents must be secure. If the private key of collection agents leaked out, then the data won't be secure. The third condition must be met is that hardware must have perfect performance. Everytime one process fetch the data, it always require keys to do encryption and decryption, which is a great pressure on CPU and I/O.

Wang, Wang, Ren, and Lou (2010) introduced a Third Party Auditor (TPA) which enables public auditability for cloud data storage. TPA combines the public key based homomorphic authenticator with random masking to audit the cloud data storage without bringing any burden to cloud services and users, and user data privacy is also guaranteed. Four algorithms (KeyGen, SigGen, GenProof, VerifyProof) are implemented in public auditing scheme. KeyGen is run by the users to setup the scheme. SigGen is used by the users to generate signatures. GenProof is used by the cloud servers to proof data storage correctness. VerifyProof is run by the TPA to audit the proof from the cloud server. By using this public auditing scheme, user's identification, data correctness, and auditing progress can be guaranteed. Yang and Jia (2013) introduced one improved privacy-preserving auditing protocol of the protocol in Wang et al. (2010). The improved protocol applies the Data Fragment Technique and Homomorphic Verifiable Tags in the method. Since the data fragment technique can reduce number of data tags, so that the performance is improved. Homomorphic Verifiable Tags is used to make the server only responses the sum of data blocks and the product of tags to the auditor, whose data block size is constant, so that the communication cost is reduced. Wang et al. (2010); Yang and Jia (2013) didn't mention much about the data availability within a specific time. In general condition, security mechanisms will get different results in different

time in order to be unpredictable. However, the above algorithms didn't take time into consideration.

Data segregation is the main security issue for data at rest in cloud computing Brodtkin (2008). Data segregation means data from different users may be stored in the same hard disk So (2011). Since data storage is shared, so problems of one set of data may affect other unrelated data that located in the same storage. Encryption is effective in protecting data, but cloud providers should provide evidence to users, and encryption schemes should be designed and tested by experienced professionals Singh and Chatterjee (2017).

Security for data in transit

Securing data using encryption technologies while transforming in the network is important in cloud computing. As proposed in Sreenivas, Narasimham, Subrahmanyam, and Yellamma (2013), an encryption algorithm is a statistical procedure used to encrypt data. As proposed in Ahmad and Habib (2010), symmetric encryption shares single key between sender and receiver to encrypt and decrypt data, and asymmetric encryption uses two keys, public key and private key between sender and receiver to encrypt and decrypt data. Abhishek and Yadav (2013) addressed symmetric key encryption is efficient for encrypting large amount of data. Example: Data Encryption Standard (DES), Triple DES (3DES), Advanced Encryption Standard (AES), and Blowfish. Asymmetric key encryption is slow and generally used for protecting the symmetric key. Example: RSA, Digital Signature Algorithm (DSA), and Diffie-Hellman. Furthermore, symmetric encryption algorithm runs faster and spend less memory than asymmetric encryption algorithm (Agrawal & Mishra, 2012). The difference between symmetric encryption and asymmetric encryption mentioned in Abhishek and Yadav (2013), in fact, based on different applications. Applications which require frequent interaction is recommended to use asymmetric encryption. Symmetric encryption can also encrypt data really well, but it requires advanced key distribution techniques to secure keys. (Agrawal & Mishra,

2012) analyzed the performance of symmetric encryption algorithms, but it lacks of the performance comparison between symmetric encryption algorithms and asymmetric encryption algorithms. It also don't have an evaluation plan for testing algorithms performance.

People prefer using strong encryption method to protect data in the public environment. Mouhib and Driss (2015) presents a framework using traditional encryption schemes (symmetric, asymmetric) and homomorphic encryption schemes (SHE and FHE). If users choose traditional encryption schemes, the cloud provider can use customer's private key to decrypt data and send it to customer. When homomorphic encryption is chosen to be used, the cloud provider will send the encrypted data to customer, and customer will decrypt the message. In Bansal and Singh (2015), a hybrid data encryption technique is developed. The hybrid encryption technique combines the process of RSA and blowfish to provide symmetric and asymmetric cryptography. As a result, it can use small key and execute really fast to provide uncrackable data due to the advantages of RSA and blowfish.

Khan, Mishra, Santhi, and Jayakumari (2015) has also developed an encryption technique that combines three encryption techniques: Fibonacci Series, XOR Cipher, and PN Sequence. The encryption technique uses segmenting key to split data into three parts, and different part uses different encryption techniques to encrypt, so that three encrypted result are generated. The same segmenting key is required in decryption process. Instead of combing multiple encryption schemes together, Ratha, Swain, Paikaray, and Sahoo (2015) propose an encryption technique using an arbitrary matrix with probabilistic encryption. It converts each value of the text file into a matrix, and multiply with another random matrix to generate encryption key. Encryption and decryption process use the generated encrypt key and data block. Multiple keys are generated for a single data block by modifying the arbitrary matrix. Unruh (2015) introduce the revocable quantum timed-release encryption and unknown recipient

encryption method that allows the sender to get back the encrypted data before the time T , and the encrypted data can't be leaked before time T . Moreover, the unknown recipient encryption allows users to send message to an unknown/unspecified recipient over an insecure network. Kocarev and Tasev (2003) propose a public key encryption algorithm based on Chebyshev maps. It generates one large integer and one random number from -1 to 1, then calculate the Chebyshev value of random number as the public key to encrypt data, and use large integer as the private key to decrypt data.

Evaluating the performance of each encryption algorithms is very important. In Nadeem and Javed (2005), DES, 3DES, AES (Rijndael), and Blowfish algorithms were chosen to be compared. The research used Java language as the platform, and execution time as the main comparison metric. The result is slightly different when dealing with block ciphers and stream ciphers. The performance results for block ciphers are shown as follows: 1. Blowfish (fastest), 2. DES, 3. AES, 4. 3DES (slowest). For stream ciphers, Blowfish is still the fastest, but 3DES is faster than AES. RSA, SHA1, and MD5 are also encryption technologies commonly used. As a result of that, Blowfish is an effective encryption algorithm as compared to other three algorithms. Sreenivas et al. (2013) get the result that RSA is a great encryption algorithm, which can generate more secure data and cost less time when uploading files on Windows Azure. Ratha et al. (2015) propose an optimized encryption technique using an arbitray matrixis, and also compared with DES, AES, and Blowfish. The execution time and throughout (in KB/sec) is better than DES and AES, but blowfish has better performance than the proposed encrypt technique.

Since the concept of public key encryption was proposed, key management has become important problem to consider. The key management is an important component, and is often used in any modern cryptographic protocol Shaheen, Yousaf, and Majeed (2015). It is the management of cryptographic keys in cryptosystem, which deal with key generation, key exchange, key storage, key usage, and key replacement.

In Pradeep and Vijayakumar (2015), different states for keys were mentioned. These states are: Generation, Activation, Suspension, Expiration, Destruction, and Archival. In the generation state, key pair is rendered. In the activation state, private or public key can be activated under specific conditions. If the status of key is unknown, or the owner of the key is not recognized, the key will be suspended in the suspension state. If the key is setup with crypto period, the key will be expired in the expiration state. When the key is not used, it will be demolished in the destruction state. In the archival state, key is archived after crypto process.

For securing the data in cloud computing environment, Xin et al. (2012) introduced the Cloud Computing Multi-Dimension Data Security model (CCMDSM) which uses three layers protecting data. These three layers are Authentication layer, File encryption and privacy protection layer, and File Fast Regeneration layer. The authentication layer is used to authenticate user by permissions or digital certificates. The second layer is designed to encrypt users' data, and the third layer is used to recover files. When intruders attempt to access users' files, the files will be locked and protected by privacy protection layer.

Furthermore, in "Enhanced data security model for cloud computing." (2012), one data security model with three layers similar with CCMDSM is introduced. This model consists of OTP authentication located in layer 1, data encryption, integrity, and private user protection in layer 2, and data fast recovery in layer 3. Instead, eight encryption algorithms were tested and evaluated to get the most appropriate algorithm in Amazon EC2 environment by using the security model. Eight encryption techniques namely RC4, RC6, MARS, AES, DES, 3DES, Two-Fish, and BlowFish. The evaluation is based on Statistical Tests. Moreover, P-value, rejection rate and time consuming are the three important metrics in comparison. The result shows the AES encryption technology is suitable for Amazon EC2 users with the lowest P-value and less encryption/decryption consuming time.

An authentication framework for Peer-to-Peer (P2P) cloud is addressed in Poh et al. (2013). The authentication model includes one security model and three phases of authentication. It assumes three entities in a P2P cloud system: users, resources hosts and trusted third party. On top of that, entity authentication and data origin authentication are the two main authentication requirements. Three phases of authentication are designed to meet the main authentication requirements. Bootstrapping Authenticated Users and Hosts is the first authentication phase which uses public key registration and symmetric key predistribution to authenticate users. Authenticated Routing, which is the second phase, uses digital signatures and MAC-based authentication to authenticate hosts. The third phase is the Authenticated Message Delivery, which implements signature with freshness token, MAC-based authentication token to ensure the authenticity of request message.

2.3 Distributed System

In this section, distributed system and techniques are discussed.

Distributed systems, cluster computing, and utility computing are relevant technologies to cloud computing (Khorshed, Ali, & Wasimi, 2012). The distributed system can be represented as one single system to users, but it has several computers connected together in different networks or physical regions. Two most well-known paradigms for distributed systems are clusters and grids (Fernandes & Soares, 2014). Clusters are designed in a more coupling and homogeneous approach with more power consumed, and grids consisted of typical machines work as slave computation node. The distributed systems have made cloud computing possible to offer storage resources and powerful processing ability with scalability as an on-demand service (Mackay et al., 2012). When dealing with large scale applications, using distributed systems is also a great solution to deal with huge volume of data (Hamdeni, Hamrouni, & Charrada, 2016).

However, it is hard to build and test distributed systems because of partial failure and asynchrony (Mccaffrey, 2016). Partial failure is the idea components can fail along the way, resulting in incomplete results or data. Asynchrony is the nondeterminism of ordering and timing within a system. Centralized systems have less scalability than distributed systems in system expansion, but when dealing with the state changes and policy changes, the centralized system is a good choice (Jain & Paul, 2013). Moreover, the distributed system has several problems and limitations. Mishra and Tripathi (2014) introduces three types of distributed systems: distributed software & hardware, distributed user, and distributed software hardware & user. This article also mentions issues and challenges of distributed software system, such as resource management, security and privacy, scalability, synchronization, and redundant testing during integration. In Afek (2013), more problems and issues are listed, such as atomic commit, atomic broadcast, and race condition. An atomic commit is an operation that applies a set of distinct changes as a single operation. An atomic broadcast is an operation that multiple processes deliver the same message in the same order. A race condition is the behaviour that output is dependent on the sequence or timing of other uncontrollable events.

Users various requirements are motivating the system to become reliable, flexible, and extendable. The distributed system should be able to save users' data, keep the interactions stable, and be always available to users (Mittal, Sangani, & Srivastava, 2015). Instead, the data processing ability is also another important metric to measure whether the system is good or not. However, cloud environment is very complex with multiple components, it finds really hard to evaluate and predict the system performance.

To have a deep learning about the reliability of distributed systems, Ahmed and Wu (2013) presents several models classified into three categories: User Centric, Architecture, and State based models. User centric approaches can be characterized as a multi stage problem solving processes where system is conceived in terms of user behavior.

Measuring user's behaviour and usage profile of the system can avoid difficulties in measuring complex systems. Architecture based approaches predict reliability at design phase in a Service Oriented Architecture (SOA). It uses data sharing approach to predict reliability for similar users and services based on past experience. State based approaches use Markov chain process to record transition from one state to another state. Various models use Markov chain to predict reliability according to the success rate or failure rate of each independent service. However, (Ahmed & Wu, 2013) also notify deficiency in these models: 1) Not all reliability factors are considered. 2) They have considered hardware failure as an important factor, but reliability of software application is ignored.

The following section describes some useful technologies used in distributed systems to enhance reliability and availability.

2.3.1 Fault-Tolerant

This section discuss fault-tolerant techniques used in the distributed system to improve the reliability of distributed systems.

Fault detection techniques play an important role in maintaining the health of a system. To achieve fault-tolerance, a distributed system architecture incorporates redundant processing components (Cristian, 1991). As stated in Sari and Akkaya (2015), replication is used for general fault tolerance method to protect against system failure. There are three major replication mechanism forms: The State Machine, Process Pair, and Roll Back Recovery. In the state machine mechanism, the process state of the system is replicated on autonomous system, and the input is sent to several replica nodes at the same time. The process pairs mechanism has a structure of master and slave. The master and slave is linked together and the master will send the same input to the slave. The roll back recovery is a check-point based mechanism, which can

use the check-point as a backup for the previous state. On the other hand, replicating data in diverse locations can increase fault tolerance, data reliability, availability, and accessibility (Hamdeni et al., 2016).

In order to design efficient fault tolerant system, Carlini, Ricci, and Coppola (2013) propose an architecture for Distributed Virtual Environment (DVE) integrating cloud and peer nodes to provide fault-tolerant for the P2P network. The architecture introduces a new kind of replica called backup Virtual Server (bVS). The bVS is located on servers that reside on the cloud, and it is used to backup data in peer servers. Peer servers are located in the P2P network. When peer servers encounter severe problems and cannot return to normal states, the bVS can detect the abnormality in peer servers, and change user connections to itself. This new architecture uses bVS to provide reliability to the system, but it assumes cloud nodes are absolutely safe and secure. However, security of cloud resources cannot be guaranteed. Borg, Baumbach, and Glazer (1983) introduce a message system supporting fault tolerance via using back process for each primary process. A primary process is a process which receives from and send to messages to users. For achieving fault tolerant message system, a backup process get replicated data from a primary process, so that the backup process can resume the computation once its primary process is down. Using backup process as the fault tolerant solution is acceptable, however, a primary process has to synchronise data to a backup process, therefore extra resources are consumed while processing messages in the system because of the data synchronous operation in processing.

Apart from using replication mechanism to provide fault-tolerant, Smara, Aliouat, Pathan, and Aliouat (2016) propose a fault detection technique that uses acceptance test in the action verification process, and build Fail-Silent cloud module. The Fail-Silent cloud module has the ability to do self-fault detection, and self-block after detecting fault situation. This module is based on component-based models, which uses the Behaviour, Interaction, Priority (BIP) tool, and the Acceptance Test is used as the

Self-Fault Detector. Moreover, the Fail-Silent system can provide the correct service, or block any services if fault situation is detected. For example, when the component receives one operation to make change, it will get the result and send the result to the Acceptance Test to verify the result, if the result is acceptable, the change will be saved and move to the next step, if the result is unacceptable, the change will not be saved, and the previous state will be retrieved, after that, the module will block the component to make sure it won't affect other components. Using acceptance test is effective in avoiding failures in the system, but more computing resources are required in the process, and it affects the response time when processing synchronous requests because of evaluation the acceptance of the result.

Instead of software or hardware problems causing system failures, message delivering problems can also cause system failures. Moser, Melliar-Smith, Agarwal, Budhia, and Lingley-Papadopoulos (1996) propose a fault-tolerant multicast group communication system called Totem. Totem is an ordered multicast group communication system which deals with problems in process message delivery within distributed systems. This system ordered the sequence of each message, and process messages according to sequence numbers. Within the Totem system, two protocols are used to manage the sequence of messages: Single-Ring Protocol, and Multiple-Ring Protocol. In Single-Ring Protocol, the system links several processes to a single circle, then it assigns a special token to one process, after that the process is allowed to send multicast message to the local area network (LAN). After the process finishes the message delivering task, the Totem system will pass that special token to another process and allow it to send multicast message later. The Multiple-Ring Protocol is providing the same service as the Single-Ring Protocol, but it is operated over multiple LANs interconnected by gateways, and it brings two reliable totally ordered message delivery services: agreed delivery, and safe delivery. The agreed delivery services guarantee that, when a processor delivers a message, it has already delivered all prior messages. The safe delivery

service guarantees, that before a processor delivers a message, it has determined that every other processor in its current configuration has received the message. The Totem system enables applications in distributed systems maintain the consistency of replicated information by providing totally ordered multicasting message. However, it is useful only when the system focuses on asynchronous messages. Moreover, the Totem system uses the user datagram protocol (UDP) to send multicast message, so that the ideal network environment is the LAN, which causes the working environment of the system is limited.

In general, computers cannot achieve the intended reliability without redundancy (Lyu et al., 1996). Software and hardware redundancy techniques are commonly used in building up fault tolerant distributed systems. Most of fault tolerant techniques are replicating data from the primary part to the backup part. However, the replication process security is not mentioned in these articles, and no effective replication methods are introduced. Furthermore, fault tolerant techniques, which are applied in asynchronous systems are limited because of process failures.

2.3.2 Load Balancing

This section discusses Load Balancing (LB) techniques in distributed systems to improve availability of services and system performance.

LB is designed to monitor traffic load of services and implement load balancing algorithms to spread the input or output pressure on each server. LB is often used to monitor service components continuously and redirect traffic when service components become non-responsive (Xu, 2012). In cluster computing, LB technology is used on high-traffic websites (Hussain et al., 2013). In Patel, Tripathy, and Tripathy (2016), four basic steps of load balancing process in Grids are mentioned: Load Monitoring

(Monitoring resource load and state), Synchronization (Exchanging load and state information between resources), Decision Making (Calculating the new work distribution and making work moment decision), and Job Migration (Data movement). In general, the goals of load balancing (Escalante & Korthy, 2011) are to: 1) Improve the performance. 2) Maintain system stability. 3) Build fault tolerance system. 4) Accommodate future modification.

Two types of load balancing algorithms (Desai & Prajapati, 2013) are static algorithm and dynamic algorithm. In static algorithm the traffic is divided evenly among the servers. The load balancing layer requires prior knowledge of server resources, so that each server can get the same traffic without difference. In dynamic algorithm the server with the lowest pressure in the whole network is preferred for balancing a load.

Shah and Farik (2015) point out strengths and weaknesses of static load balancing algorithms, such as Round-Robin, Weighted Round-Robin, Opportunistic Load Balancing (OLB), Min-Min, and Max-Min algorithms. The round-robin algorithm uses round-robin scheme for allocating jobs (Desai & Prajapati, 2013). It selects the first node randomly and then allocates jobs to all other nodes in a round robin fashion. As each task is assigned to processors on each node, so there is no starvation. However, the processing time of each processor is different, which makes some nodes become heavily loaded while others remain idle and under-utilized. Weighted round-robin assigns a weight to each server (Khiyaita, Bakkali, Zbakh, & Kettani, 2012), so that the server with the highest weight will receive more tasks. If the same weight value is assigned to each server, then all servers will receive balanced traffic. Opportunistic Load Balancing attempts to keep node busy. It does not consider the execution time of each server, which cause OLB assign tasks to random available nodes regardless of the node's current workload (Khiyaita et al., 2012). Min-min and Max-min load balancing algorithms are similar, which calculates the execution time of all tasks and assign the minimum or maximum time cost task to the corresponding server (Gupta & Sanghwan,

2015). The above static load balancing algorithms are useful in common situation, but each of them lacks the comprehensive design when dealing with traffic load and server states such as task execution time, server's availability, and server's performance.

As described in Alakeel (2010), the advantage of dynamic load balancing algorithms is that tasks can move dynamically from an overloaded server to an unoverloaded server according to the current state of the system. However designing and implementing dynamic load balancing algorithms is much more complicated and harder than implementing static load balancing algorithms. Milani and Navimipour (2016) discusses several popular dynamic load balancing algorithms: Tasked-based load balancing method (Ramezani, Lu, & Hussain, 2014), Honey bee behaviour inspired load balancing (LD & Krishna, 2013), Enhanced bee colony algorithm (Babu & Samuel, 2016), Agent-based load balancing (Gutierrez-Garcia & Ramirez-Nafarrate, 2015), and etc. A more comprehensive analysis to dynamic load balancing algorithms can be seen in Milani and Navimipour (2016). In conclusion, with dynamic load balancing mechanisms, the system response time can be improved significantly. Furthermore, flexibility and scalability of cloud can be guaranteed. However, decision-making for the selection of resources are essential, so that the run-time overhead is a major issue. Also, memory utilization and the complexity of dynamic load balancing algorithms are issues need to be considered as well.

Besides using load balancing algorithms balance traffic load, many techniques and systems are absorbing the idea of load balancing algorithms. The Hadoop MapReduce platform (Chang et al., 2015) are used when processing big data. It is scalable, fault-tolerant, and provide load balancing. The Hadoop process chunks data by using the name and data nodes and give out small pieces of result stored on each data node. In Hadoop, the name node receives user's requests and perform necessary indexing and searching by initiating a large number map and reduce processes. Once the operation is completed, the name node returns the output value to servers and clients. The load

balancer within Hadoop spreads the load equally across all available data nodes via comparing states of each data node within the name node. However, the name node is the entry point of load balancing, which makes load balancing useless when the name node fails. However, name and data node structure splits searching and storing function into different parts, so that Hadoop Mapreduce is efficient to deal with big data.

Sheng and Bastani (2004) mentions one load balancing technique which is based on spread cache on each node in cluster. Each independent node can ask for different parts of cache from the main cache server, and store in its own cache, so that the traffic load on the main web server can be reduced and spread on other nodes. The main server and nodes can also set up reference number which limits the total number of cache get from other servers. Spreading cache into multiple locations can reduce the pressure of the main server's cache, but it requires data synchronization between master and slave servers. On the other hand, spreading cache only optimize the performance in resources layer, load coming from the upper layer is not considered. Virtual Layer 2 (VL2) (Bagchi, 2015) is a network topology that describes the load balancing method implemented on physical switches. It can direct the traffic to a random core switch, and then forward the traffic to the real destination. As a result of that, once unpredictable traffic is detected, this routing method can deliver it to a random core switch as an intermediate destination. VL2 is a simple load balancing technique which focuses on the physical layer only, so that developers may not concern it as the main load balancing solution to their applications or services. Also, VL2 is limited by the amounts of physical hardware because if the physical layer has only one or two core devices, then the traffic will always flood into the core devices, which has no load balancing at all. Besides the hardware limitation, VL2 has to define whether the incoming traffic is predictable or unpredictable, and the traffic has to arrive to core devices before reaching to the real destination, which may cause massive delay for traffic transmission. However, VL2 uses core switches as a buffer for the system, which is useful to handle

massive incoming traffic.

2.4 Decentralized System

This section discusses basic concept of decentralized systems, and efficient design and security of decentralized systems.

Decentralized system is a system in which low level components operate on local information to accomplish global goals “Hierarchical Decentralized Systems and Its New Solution by a Barrier Method.” (1981). It is different from a centralised system which has the specific controller to control everything in the system. However, the decentralized system is known as distributed control Sandell, Varaiya, Athans, and Safonov (1978). As described in Feiker (1979), a distributed control system is a computerised control system for a process or plant, in which autonomous controllers are distributed throughout the system, but there is central operator supervisory control. In this case, if a processor fails, it will only affect one section of the plant process. Major differences between decentralized system, centralized system, and distributed system can be notified in various aspects (Goyal, 2015): maintenance, stability, scalability, ease of development, and evolution. The centralized system is much easier to maintain and develop because it has only one controller, but the distributed system and decentralized system can avoid single point of failure.

Autonomous Decentralized System (ADS) is an advanced system architecture to provide fault-tolerance, on-line expansion, and on-line maintenance (Coronado-Garcia, Gonzalez-Fuentes, Hernandez-Torres, & Perez-Leguizamo, 2011). The ADS is composed of modules and components that each of them is designed to perform their own functions independently to achieve the overall system goal (autonomous decentralized computer control systems). The Autonomous Decentralized Test System (ADTS) (Zhao & Xiao, 2015) fully describes the advantages of using ADS. The

ADTS consists of Autonomous Decentralized Test Units (ADTU) which are placed in the double-ring structure. The ADTU are autonomous and can finish assigned tasks independently. In order to do online system expansion, for example adding nodes, the ring will break at the target point, and the information transmission between other nodes will not be affected, after that, the ring will be restored when new nodes are added successfully. When dealing with the node failure, the ring will break and block the failed node. Moreover, one data protocol ADP is introduced to work with ADTS. The ADP contains five parameters: serial number (SN), data type (ICD), priority (PRI), duration time (DUR), and data (DATA). The network control processor (NCP) is a controller that controls the transmission line in the double-ring structure in decentralized system (Ihara & Mori, 1984). Two NCPs are connected to a host processor, and each NCP is independent to other NCPs, so that no receiver's information is required when the message is transmitted. Moreover, the ADS also has fault detection and fault recovery ability by using minor loop check signal and major loop check signal. Failed NCPs can be detected by signals, and the transmission route will be changed at the same time.

ADS doesn't have the central controller, but uses the Autonomous Control Processor (ACP) to control each subsystem. Each subsystem connects to each other through the Data Field (DF), which is used to send and receive messages. As stated in (Coronado-Garcia et al., 2011), the DF is designed to have private key and public key. When one DF tries to send the message, it uses receiver's public key to encrypt the message with the signature and one random number. The receiver DF will establish the connection only if the sender's identification is authenticated. Asymmetric key encryption is used in DF, but author didn't mention how to secure the communication happened within the subsystem. A secure message send methods in decentralized systems is introduced in Peyravian, Matyas, and Zunic (2002). The method generates a random secret key on the first node of node group, and distribute the random secret key to other remaining nodes, then the second node of the node group generates a random number and use a one-way

hash to create a working key. The working key is also distributed to other remaining nodes. As a result of that, all nodes can use the working key, the random number, and its signature to decrypt secured message. Besides the data security in autonomous decentralized systems, the network security is also one major problem. Generating key from the first node to the last node will cost a lot of time and lower the response speed. In Guo and Wang (2005), the Secure Connective System, Threat Defence System, and Trust and Identity Management System is mentioned. The Secure Connective System uses data encryption technologies such as SSL and IPsec to secure communication between each node. The Threat Defence System aims to provide protection mechanism to network attacks and unknown attacks. The Trust and Identity Management System manages user's privileges and permissions to specific business. The Secure Connective System, Threat Defence System, and Trust and Identity Management System seem professional to secure decentralized system. However, this solution requires various techniques to accomplish, which is a constrain to the solution.

Low latency for data querying is an important metric for cloud computing. In order to reach the high speed of read/write requests, cache technology is normally used on servers. In Takahashi, Mahmood, and Lakhani (2015), an autonomous decentralized system based URL filtering system is introduced. The filtering system uses L3 cache on each decentralized node which stores the relationship of text. The L3 cache consists of two modules: RAMDISK and Cache control. The RAMDISK is the virtual device driver for caching the most frequently used data, and the Cache control handles the incoming read/write request. Using cache can boost I/O speed, and this system uses two cache modules to manage data read and write, which is a great design. However, this system does not mention the backup solution. If one node is accidentally shut down, then all the record will be lost without backup. P2P routing protocol and P2P caching technology are great choice for improving querying data (Sheng & Bastani, 2004). Pastry is an example of P2P routing protocol. Each node in Pastry has a unique nodeID

with a key, and one routing table to store the information of other nodes. The proxy cache server is used in P2P caching technology. It is the centralized location of each node cache, if the requested data doesn't exist in the proxy server, the requestor will search the cache among other nodes, and the closet node is preferred. For Pastry, having one central cache location is not a good idea. If the centralized location is compromised, then all cache data will be lost. The routing table within each node is also vulnerable.

2.5 Random Distribution

In this section, similar component distribution services are researched.

Distributed system can improve the stability and performance because of the network load is spread into several server instances in the group or cluster. However, the services running on each server instance is previously designed, so that specific type of user request has to go to the specific server. Integrating random elements into the distributed system can be a good way to have uncertainty and strong security level for the whole system.

Assigning different parts in the system makes the overall architecture more complex. As part of the grid system, services are always expected to have nondeterministic behaviours (Zhang, Zagorodnov, Hiltunen, Marzullo, & Schlichting, 2004). In Carlini et al. (2013), the recent client/server DVE architecture was mentioned which split multiple Virtual Environments (VE) into several nodes randomly. But the problem was also notified that when several heavy-load VE were located on the same node, the node would be under severe pressure. The authors introduced the Positional Action Manager (PAM) and the State Action Manager (SAM) to manage the position of each VE and organize the state of each VE and node. GreenMap is a VM-based management framework which dynamically allocates and reallocates VM resources within a cluster Chang et al. (2015). One of the algorithms in the framework starts by randomly

generating new placement configuration in order to save more power when dealing with reallocating live VMs into physical resource. The previous papers only focused on VM-based resources allocation and distribution scenario, but the service-based random distribution is not covered.

The MAIS-IDS is a distributed intrusion detection system using multi-agent Artificial Immune System (AIS) approach Afzali Seresht and Azmi (2014). It is able to randomly select multiple agents to clone, and migrate cloned agents to new environment to aid scanning, which is similar to the service-based random distribution.

2.6 Conclusion

Ki-Ngā-Kōpuku is a new concept of system. It is a distributed security system, which is supposed to be highly available, and provide fault-tolerant ability in the system. As Ki-Ngā-Kōpuku is based on the cloud environment, so the common problems happened in the cloud environment are concerned. As a result of that, data security, system distribution, system performance and system availability are key problems to be solved.

Based on the research of literature review, cloud computing security can be achieved by implementing general security policy and data security mechanism. The general security policy is supposed to do user authentication and authorization. Data security is achieved by using encryption techniques and designing efficient architecture based on low level technologies such as hypervisor and hardware. On top of that, users can't control their own data on the cloud directly, because the data is stored on network resources which are provided by cloud providers, so it's not secure even though the security mechanism is implemented. Cloud environment is normally built up on the distributed system and the decentralized system, so that fault-tolerant and availability is guaranteed. Data replication is the common technology researched on both system, however how to check the integrity of data replicas and how to manage the access

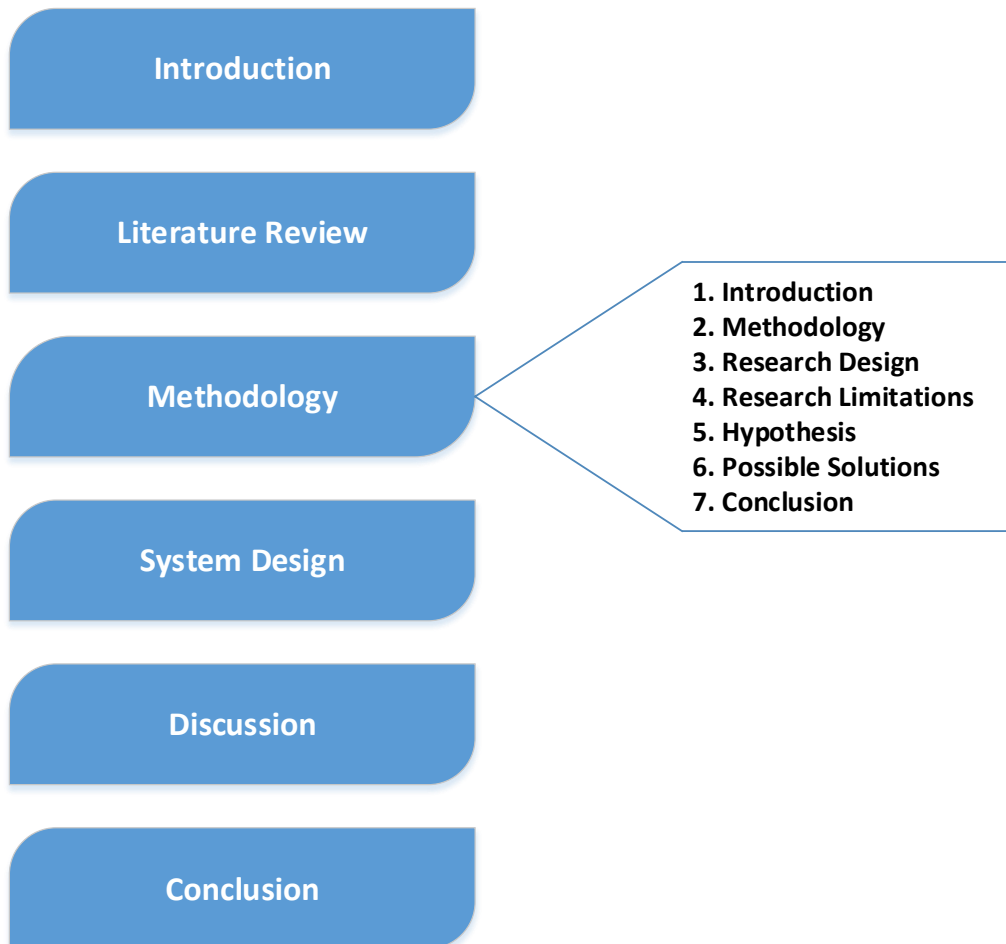
to data replicas were not mentioned. Besides, no articles were found to research the component random distribution which is a key feature in Ki-Ngā-Kōpuku to distribute components into multiple servers randomly. Private-public key pair can be used on user authentication and server authentication, but the key management is the critical problem. Designing key exchange mechanism and protocol for key management is important in key encryption research, but most of the research relies on extra key servers which is not secure if the key server is compromised. In order to evaluate the performance of system, several comprehensive evaluating plans should be made based on different scenarios and multiple metrics.

In general, challenges and chances are obvious in this research project. In order to complete this project, the following questions will be researched in this project:

1. How to improve security among components?
2. How to design a component distribution mechanism to distribute components into several servers?
3. How to maintain communication between components which are located on different servers?

The next chapter 3 will discuss what methodologies are used during the research. It also propose hypothesis for each research question.

CHAPTER : METHODOLOGY



Chapter 3

Method

3.1 Introduction

To achieve the research objectives, the appropriate research methodologies and plans should be addressed clearly. This research project not only focuses on the proof of concept of a new system concept, but also focuses on developing the system and bring out the prototype. As a result of that, the research methodology should be useful in both research and developing area. Besides knowing what kind of methodology should be followed during the research, operation details should also be clear so that the developing process can stick onto the research process. In this chapter, research questions from the previous chapter will be restated. Several hypothesis related to research questions will be mentioned.

Research Questions

1. How to improve security among components?
2. How to design a component distribution mechanism to distribute components into several servers?
3. How to maintain communication between components which are located on different servers?

The above three research questions were proposed according to the literature review. Ki-Ngā-Kōpuku is a decentralized distributed security system based on cloud environment, because the cloud is the public network environment, so the security is the first problem to be concerned. As the cloud resources are totally controlled by the cloud provider in the aspect of fundamental hardware, so 100% of security on the cloud is not granted. Being motivated by this security problem, the first research problem is proposed.

Component distribution is rarely seen on research papers. It is one important function in Ki-Ngā-Kōpuku. In this case, the second research question is proposed. Separating one application into multiple parts is the main goal in distributed computing, especially in designing the fault-tolerant system with high scalability. Splitting tasks into several parts and deliver each part into different servers can improve the system performance. However, systems are mostly designed with centralized controller, and component communication relies on the centralized controller, which makes it difficult to fully separate the system into parts and deploy into different machines. In order to keep the communication between components, most of systems deliver agendas with the component. As a result of that, if the agenda of one component goes down, that component cannot communicate with other components again. Ki-Ngā-Kōpuku is supposed to get rid of the agenda, and still have the ability to maintain communication between different components, and make all components look like one single application.

3.2 Methodology

3.2.1 Design Science Research Methodology

Design Science Research (DSR) is a set of analytical techniques and perspective for performing research in Information Systems A. Hevner and Chatterjee (2010). As

stated in Peffers, Tuunanen, Rothenberger, and Chatterjee (2007), three elements are included for a design science research: conceptual principles, practice rules, and a process for carrying out and present the research. Six steps of design science are also introduced in McPhee (1996): programming (to establish project objectives), data collection and analysis, synthesis of the objectives, data collection and analysis, synthesis of the objectives and analysis results, development (to produce better design proposals), prototyping, and documentation (to communicate the results).

DSR has three cycles: Relevance Cycle, Design Cycle, and Rigor Cycle. As proposed in A. R. Hevner (2007), the Relevance Cycle connects application environment to design science activities. The Rigor Cycle connects design science activities to knowledge base with science theories and methods. The Design Cycle connects evaluation activity to building design artifacts and processes activity. In more details, the environment contains application domain, which includes people, organizational systems, and technical system. It also has problems and opportunities. The environment defines basic milestone for the design science research activity. The Relevance Cycle passes requirements and field testing in the middle of Environment and Design Science Research. In this case, the Environment defines application problems and requirements. Design Science Research activity build the application according to requirements gathering from the Environment and evaluate whether it is possible to run. On the other hand, the Rigor Cycle brings theories and experience in design science research activity. It also update knowledge and experience getting from the research activity.

During the research, DSR methodology is used through the project. The reason to use DSR methodology is that the whole research part is mainly divided into three parts. One part is define system requirements, another part is develop and evaluation. The last part is experience management. In the whole research, three main research phases are implemented. Phase I is Define System. This phase includes identifying what system requirements should be achieved, finding out possible opportunities of the project, and

what problems happened during the system development. Phase II is developing system, and system evaluation. This phase is the most practical part of this project, and is the place to find out what should be improved of the system. Phase I will also be updated. Phase III is directly related to Phase II because this phase is the theories background of the whole project. Once the system needs improvements, the theories background should be updated first.

3.2.2 Rapid Application Development

While due to the time restriction, bringing out one running system prototype is the most important task, so the development methodology used in developing system phase is Rapid Application Development (RAD) methodology. As stated in Hough (1993), RAD puts more emphasis on processing and developing, rather than planning. Martin (1991) defines the key objectives of RAD as: high quality systems, fast development and delivery and low costs. As stated in Beynon-Davies, Carne, Mackay, and Tudhope (1999), RAD projects are typically small-scale and of short duration, which is about two to six months. Projects expected to finish in six years are available to use RAD methodology.

Except the time measurement, the type of project also affects the methodology usage. When the project is supposed to build up a prototype system with iterative process, RAD methodology is still a good choice. When develop a prototype, developers and users can have time to communicate to identify system requirements and user requirements.

Fig. 3.1 shows the process of RAD. First process is Analysis & Quick Design. This process defines system requirements, and gives priority to each task. The task with highest priority has to be completed first. The next cycle has three process: Develop, Demonstrate, and Refine. Once one part of the system is developed, demonstrate and review the part can help find problems. After the cycle, testing process bundle

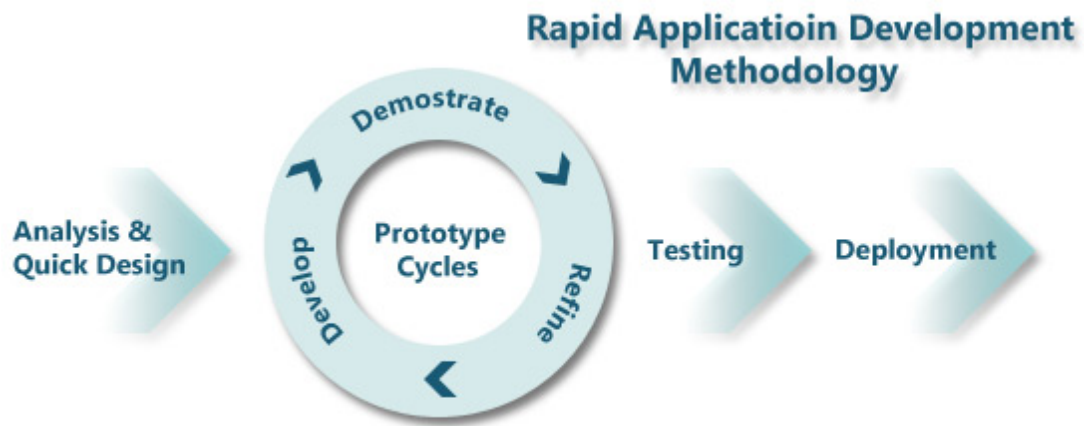


Figure 3.1: Rapid Application Development Methodology (*Ramsoft Consulting*, n.d.)

everything into one part and test it as one unit. If the testing result is good, then the prototype can be deployed.

The reason to use RAD as the methodology in developing system phase is that the project is supposed to build up one prototype, and only six months to finish the project. As a result of that, RAD is the right methodology to use.

3.3 Research Design

In order to achieve the research outcome, the research processes have to be designed. At this point, it is useful to follow the chosen research methodologies: DSR and RAD.

First of all, the overall research goals are divided into three levels:

- Level 1: A theoretical background for prototype system.
- Level 2: Executable units for specific purpose.
- Level 3: A compatible solution that meets theoretical and technical requirements.

The level 1 research goal is to make sure the whole research project has a strong theory background, so that it is achievable in the aspect of academic level. The level 2 research goal is for implementing this project in the aspect of technical level. Even

though some requirements can be proved by theories, but still need available techniques to prove it. The level 3 research goal intends to make this project become achievable not only in academic theory, but also in industry practical development.

3.3.1 Research Goals

To begin the research, research goals have to be clarified. Based on the literature review, research questions have to be analyzed during this research. Moreover, besides research questions, the research outcome should also be valuable for current or future computer science development. Hence, to understand the research goals, I must have a clear view about what the current industry needs and what can this research bring.

3.3.2 Research Process

The DSR methodology is chosen to be used during the whole research. Based on the research processes of DSR, the following research processes will be described:

1. Research Purpose.
2. Infrastructure and Applications.
3. Applicable Knowledge.
4. Develop / Build.
5. Justify / Evaluate.

The first research process is to clarify the purpose of this research. As 3.3.1 mentioned, the research purpose should be clear before starting the whole research. Due to a conceptual model has been created in Litchfield et al. (2016), as a project to proof this new model, implementing techniques to work as the conceptual model describes is the main purpose of this project. During the research, the research questions summarized from 2 should also be analyzed according to the findings.

The second research process is to define the system infrastructure and what the application should be presented after the research. Ki-Ngā-Kōpuku is designed to run on the cloud environment. Based on the conceptual model mentioned before, Ki-Ngā-Kōpuku must be distributed and decentralized. As the serving target of Ki-Ngā-Kōpuku is customer's application, so Ki-Ngā-Kōpuku should be flexible to implement and require less dependencies. Based on these considerations, the artifact type of this research will be Instantiation. Ki-Ngā-Kōpuku will be designed as a framework, and has the ability to handle unexpected conditions or errors without interfering the status of user's application.

In order for design science to achieve the objective of being rigorous, the research must draw on existing knowledge from a number of domains. Besides, DSR must also make a contribution to the archival knowledge base of foundations and methodologies (A. R. Hevner, 2007). In general, journal articles and conference papers will be searched. In addition, practical papers may offer even more specific and current knowledge. To start this research, the following areas will be researched for the knowledge base: Cloud Environment, Cloud Security, Distributed System, and Decentralized System. The point for cloud environment research is collecting cloud environment features and discover potential problems that may consider during the research. For example, the research on cloud security can identify what kind of security problems that the system will face. Distributed system and decentralized system are popular in cloud computing. As a result of that, studying on related literatures can get a clear view of useful design or architecture which might be helpful to design the system during research. The research on existing knowledge will meet the level 1 research goal.

In DSR methodology, it must produce and evaluate a novel artifact (A. R. Hevner, 2007). Moreover, the artifact in DSR does not have to be a fully functional system, so this process will fulfill the level 2 research goal. As a result of that, system modules will be the milestone of this research. In order to design the system module, firstly,

system architecture and features have to be designed. Then domains and objectives of each system module must be clarified (level 1 research goal). After that, the scope has to be defined, and choose appropriate techniques and tools to develop system modules.

After the system modules development, each system module has to be evaluated. In order to evaluate system modules, based on (Peffer, Rothenberger, Tuunanen, & Vaezi, 2012), the evaluation method within DSR is Prototype. A prototype is implementation of an artifact aimed at demonstrating the utility or suitability of the artifact (Peffer et al., 2012). In this case, the prototype will be composed of each system module. By evaluating the prototype, each system module is able to be evaluated according to its own purpose. This process will achieve the level 3 research goal.

3.3.3 Research Evaluation Criteria

According to 3.3.2, the evaluation method is prototype.

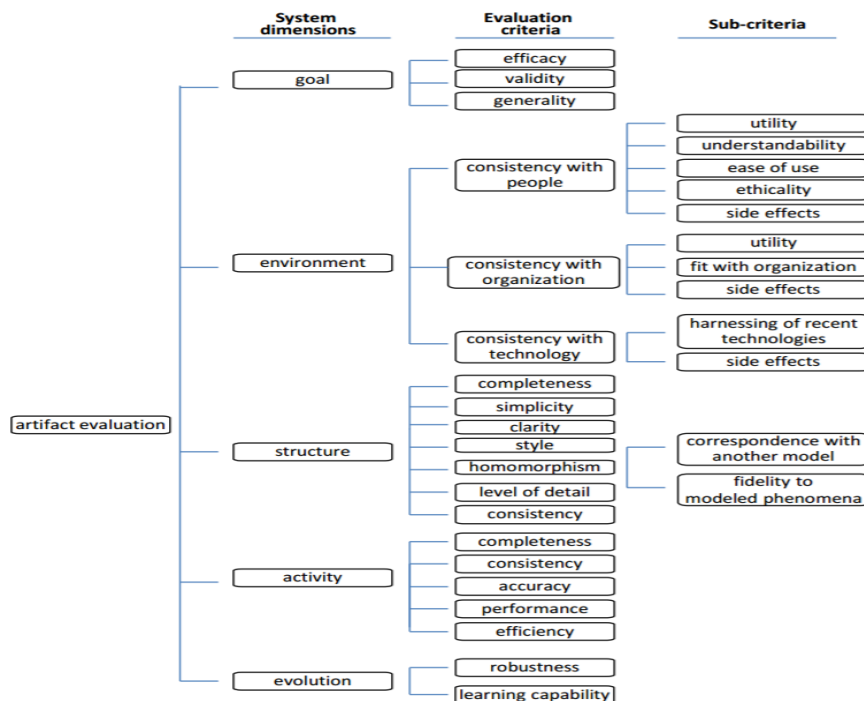


Figure 3.2: Hierarchy of criteria for IS artifact evaluation (Prat et al., 2014)

As shown in Fig. 3.2, there are five dimensions in artifact evaluation. In order to evaluate the prototype of this research, the evaluation criteria has to be defined as follows:

System Dimensions	Evaluation Criteria	Criteria Description
Goal	Efficacy	The prototype achieves research goals as expected.
		Parts of the prototype achieve research goals as expected.
	Validity	The prototype is able to handle each request correctly.
	Generality	The theory and prototype are valuable to current cloud computing environment.
Environment	Consistency with people	The prototype is easy to use and understand.
	Consistency with technology	The prototype is an implementation of new concept or new techniques.
		The prototype has less side effects.
Structure	Completeness	The prototype covers all related modules.
	Homomorphism	Each module in the prototype is flexible to work with other modules.
Activity	Completeness	Each module in the prototype is functional.
	Performance	The prototype is able to handle request and process data within a limited time.
Evolution	Robustness	The prototype is able to take actions to the change of environment.
	Learning capability	The prototype has the ability to learn from previous experience.

Figure 3.3: Evaluation Criteria

3.4 Research Limitations

The following list is the limitations of this research:

1. Research Scope.
2. Research Time.
3. Prototype Evaluation

The first limitation is the scope of this research. As mentioned in 3.3, this research needs to research various areas, and the whole project requires many techniques to accomplish. However, I am the only one who is responsible for prototype implementation, so it is a massive task. The project must cover techniques about distributed systems, decentralized systems, and also cloud computing. Within each of these areas, there are many constraints need to be considered, and also many techniques that require a lot of time to master. As a result of that, there will be a long period of time to go through the three cycles in DSR. On the other hand, this research is supposed to prove a new concept which is new to current existing knowledge. During the research, few related literatures were found, so it is difficult to update knowledge base within DSR.

The second limitation is the research time. As described before, this project requires a long period of time to finish. However, the given time is not enough to finish this research project, so DSR methodology is not suitable for the research which is required to have the output in a short time. In this case, RAD is used in the second research process (Infrastructure and Applications) and fourth research process (Develop / Build) within DSR. With the help of RAD, time can be saved in system modules development, and more time can be spared for prototype evaluation.

The third limitation is the prototype evaluation. The prototype is the final outcome of this research. However, within DSR, the prototype is evaluated when the design is updated or completed, which requires too much cost. In this case, by implementing RAD methodology, the prototype can be evaluated after system modules are fully functional.

3.5 Hypothesis

- Hypothesis 1: Independent temporary public-private key pair authentication can improve security among components.

Protecting network traffic is very important in the cloud environment. In normal situations, encryption techniques are always applied to encrypt data and authenticate communicate peers. However, in most of the systems, centralized key management server is always used to provide user authentication by using key pairs or other multi-factor authentication methods. If the centralized key management server is attacked by hackers, then user credentials will be lost. Besides, if the key management server contains keys related to servers under its network, then the whole network will be under danger.

Using temporary public-private key pair can reduce risks of key pairs leaking. Without centralized key management server, temporary key pair is useful to protect data integrity and secure system message. The reason is that key pair is always different from previous key pairs, and it won't last forever because it is temporary, which makes hackers hard to track.

- Hypothesis 2: Randomly distribute components into multiple network locations can improve the redundancy and performance level of overall system.

Replicating the service and store replicas on different servers or locations make the system become distributed. The performance and stability is greater than the centralized system. However, the service may have many components, so that replicating all services can be a great burden to other servers. Splitting services into multiple service components and distribute random number of components into random number of servers can be an efficient way to solve the problem. The size of each component is smaller than the original service, so less resources will be cost while transferring and running. On the other hand, because of

randomization, the amount of server and components can be various. In this case, randomly distribute components can improve redundancy and performance of overall system.

- Hypothesis 3: Using sockets and distribution feature of Erlang can maintain the communication between component.

Opening sockets on each server to communicate with outer world might be a good way. Sending and receiving data by using sockets is a common way to transfer data for the system. In the system design, Erlang nodes can listen to sockets to receive data from other servers, and use specific port to maintain communication between each node.

In normal condition, agents are used in distributed system to communicate with other agents on different servers. It is useful to manage and control the system. However, it is painful to install agents while installing system on each server, and hacking agents could cause serious problems in the system. As a result of that, using Erlang to build distributed system is a good choice, because Erlang nodes have the ability to communicate via different servers and network. One Erlang node is a small virtual machine running on the server, it has the remote procedure call function to call functions on other remote nodes, so that it is flexible to build distributed system and distribute services into multiple servers.

3.6 Possible Solutions

- Hypothesis 1: Independent temporary public-private key pair authentication can improve security among components.

Temporary public-private key pair can be used to encrypt data and authenticate components in the system. Once the connection is going to be established, random seed and current timestamp can be used to create public-private key pair. The

initiator and responder can exchange the public key, then encrypt message by using public key, and decrypt message by using private key.

- Hypothesis 2: Distribute service components into multiple servers can improve the redundancy and performance level of overall system.

Randomly select components to distribute can be achieved by implementing pseudo-random mechanism, and sorting components by total number that has been distributed into the system. Developers can design how many components to be distributed, and randomly amount numbers to be replicas. Because components and its number are not fixed, so it might be helpful, when nodes find one related node on local server, then the request can go to local components first.

- Hypothesis 3: Using sockets and distribution feature of Erlang can maintain the communication between each server.

For building up the communication channel between each node on different servers, sockets are opened to exchange message among servers. In this case, Ki-Ngā-Kōpuku listens to specific port, then send and receive message by using the specific port. However, for avoid system collision, Ki-Ngā-Kōpuku will be application-dependent, which means the relationship between Ki-Ngā-Kōpuku node and application is one to one, and only one single port is used for one specific application.

Ki-Ngā-Kōpuku node should be designed to listen the socket and responsible for handling traffic for system and application components. Application components can be designed as one single node, which makes it flexible to communicate with other nodes and be managed by system node. When an application component wants to communicate with components on other servers, that components can talk to system node first by using Erlang distribution feature, then Ki-Ngā-Kōpuku node talks to other servers by using specific opened port for communication.

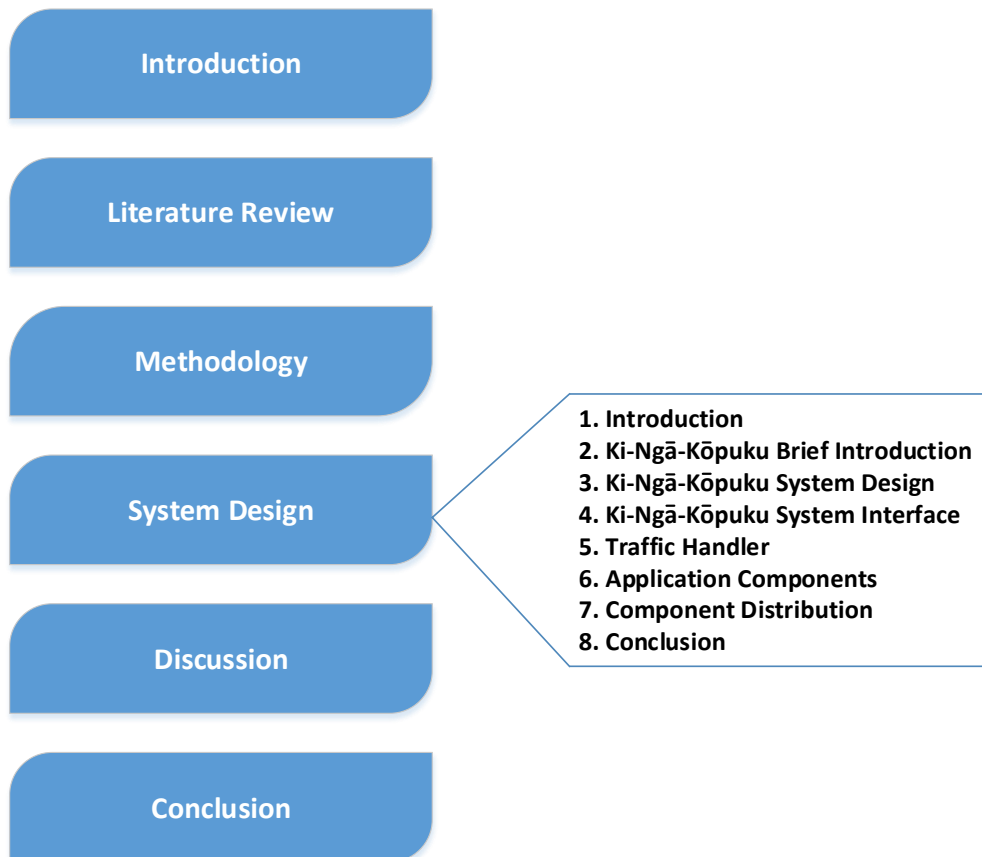
3.7 Conclusion

According to research questions from previous chapter, related hypothesis and possible solutions are declared. During the whole process of the project, Design Science (DS) Research Methodology is used to support the project, because DS has three cycles, one is identify and update system requirements and problems, another cycle is develop system and evaluate system. The third cycle is update theories and experience. Each cycle is important to this project. In more details, in the system develop process, Rapid Application Development methodology is used because of the project is prototype, and the time limit is six months, which is a short develop period of prototype.

By following the DSR methodology, at the very beginning of this research, the objectives of Ki-Ngā-Kōpuku is clarified in the Environment phase. The objectives of this research are designing a new system which is based on a new concept. According to the basic concept, security and distribution are two important parts that need to be achieved by Ki-Ngā-Kōpuku. In this case, related research questions are proposed, and articles are studied according to the research objective in Knowledge Base phase. After evaluating useful information of each article, the researched theories are applied to design the basic system architecture. Then a system prototype should be developed by following the basic system architecture. While developing the prototype, more advanced design or problems may be discovered. In that case, these new discovery will be reflected to the existing system design. For seeking the achievement of new discovery, related theory and technology will be researched in Knowledge Base phase, so that theoretical foundations will be updated. In the end, new ideas or solutions will be implemented into the prototype development.

Next chapter 4 describes details of Ki-Ngā-Kōpuku, including system architecture and system behaviours.

CHAPTER : SYSTEM DESIGN



Chapter 4

System Design

4.1 Introduction

The Methodology chapter 3 has introduced two methodologies used during the theoretical research and development. The design science research methodology is used to support the whole theory research process. For developing Ki-Ngā-Kōpuku in the correct way, the rapid application development methodology is used. Besides two methodologies, six research questions are proposed by researching related theories and reviewing related works. This research aims to proof the concept of Ki-Ngā-Kōpuku, so this chapter will discuss the system design of Ki-Ngā-Kōpuku in both theoretical and technical way,

This chapter describes details about Ki-Ngā-Kōpuku. In section 4.2, system features are described in subsection 4.2.1. System requirements for Ki-Ngā-Kōpuku is discussed in subsection 4.2.2. The subsection 4.2.3 describes the system architecture in the aspect of its programming language. In section 4.3, information details of Ki-Ngā-Kōpuku are described. The subsection 4.3.1 discuss the concept of Ki-Ngā-Kōpuku cluster, subsection 4.3.2 introduces a concept of application group in Ki-Ngā-Kōpuku. Subsection 4.3.3 discusses the network environment and topology of Ki-Ngā-Kōpuku.

System architecture is described in subsection 4.3.4, which describes system and communication architecture. Section 4.4 describes the functionality of the system interface of Ki-Ngā-Kōpuku. Next section 4.5 discusses the message delivery service in Ki-Ngā-Kōpuku. Section 4.6 discusses application components which are used in Ki-Ngā-Kōpuku. It first defines the concept of application components, then the component storage, actions, and communication methods are introduced. Next section 4.7 discusses component distribution service of Ki-Ngā-Kōpuku, which gives details about how to accomplish component distribution and requirements that have to be achieved for distribution.

4.2 Ki-Ngā-Kōpuku Brief Introduction

4.2.1 System Features

As Ki-Ngā-Kōpuku is designed as a framework that can be used by developers to provide redundancy and distribution to their applications, so Ki-Ngā-Kōpuku has to deliver related, flexible abilities for applications.

The following shows what Ki-Ngā-Kōpuku can do:

1. Receive traffic from application interface or user interface.
2. Record application's component information.
3. Automatic component distribution and implementation.
4. Prevent single points of failure for any application components.
5. Enable component intercommunication.

4.2.2 System Requirements

Before implementing and running Ki-Ngā-Kōpuku, some requirements have to be met.

1. Ki-Ngā-Kōpuku is written in the Erlang programming language, so Erlang has to be installed on the server.
2. Ki-Ngā-Kōpuku requires some shell script files for the specific task, so the ideal running environment is Linux.
3. Applications should be written in Erlang supported programming language
4. Currently, Ki-Ngā-Kōpuku can only be used within the Local Area Network (LAN).

4.2.3 Programming Language

Ki-Ngā-Kōpuku is written in the Erlang programming language, and its structure is designed to follow the Erlang Open Telecom Platform (OTP) structure. OTP is a set of modules and standards designed to help build applications. As described in (Mccaffrey, 2016; Afek, 2013), partial failure and asynchrony are typical problems for distributed systems. (Mittal et al., 2015) also propose the distributed system should be able to be reliable and flexible. Luckily, Erlang is able to use its OTP structure to solve these problems.

Erlang OTP has supervisor, `gen_server` behaviour. Each behaviour plays different roles in the Erlang project (*Building an Application With OTP*, 2017):

supervisor The supervisors are responsible for the monitoring and control of the child process. In normal conditions, if one process is unconditionally going down, other processes might be affected by the unconditional status. The system might encounter problems because of that. However, the Erlang supervisor can handle any crashes that happen with its child process, which means supervisors can prevent crash errors from affecting the whole system. A child process can either be another supervisor or worker process. Moreover, supervisors can define restart strategies and stop time intervals for the child process. By defining the child

process restart strategies, supervisors can manage the child process to restart when the child process is stopped, which makes the whole system remain healthy.

gen_server The Erlang `gen_server` works like a generic server in the Client-Server structure. It can receive messages from clients and computing the results. There are three types of request that can be received by the `gen_server`: synchronous requests, asynchronous requests, and normal requests. When the `gen_server` receives the synchronous request, it will send the result as soon as the process is finished. When the `gen_server` receives the asynchronous request, it will send the reply message `ok` immediately without waiting.

By comparing each behaviour in Erlang OTP, the supervisor and `gen_server` behaviours are mainly used in Ki-Ngā-Kōpuku. The Erlang `gen_server` is responsible for processing the user and system request, which acts like general servers, and the supervisor is responsible for monitoring the `gen_server` and handling errors coming from the child processes. The Erlang supervision tree describes this system action. Within the supervision tree, two factors are used: supervisors and workers:

supervisors Responsible for supervising workers and handling errors. If workers encounter problems, the supervisor can restart workers according to predefined restart strategy.

worker Workers are processes which perform computations, and responsible for performing specific user or system actions.

By following the Erlang supervision tree design, the Erlang programming language has fault-tolerant features that make Ki-Ngā-Kōpuku highly reliable and feasible.

The supervision tree is a basic system architecture implemented in the Erlang programming language. Fig. 4.1 shows the supervision tree of Ki-Ngā-Kōpuku. The circles represent supervisors, and the square boxes represent workers. In the supervision

tree, Ki-Ngā-Kōpuku is mainly separated into three parts: socket, main server, and local storage. The socket is used to communicate with the outside world. The main server holds critical functions for Ki-Ngā-Kōpuku. The local storage is responsible for storing local system information. As shown in Fig. 4.1, each worker is connected to a supervisor, which means that when any server encounters problems, its supervisor will handle the error and restart the server. As a result of that, the failure condition of one server won't affect other parts of the system.

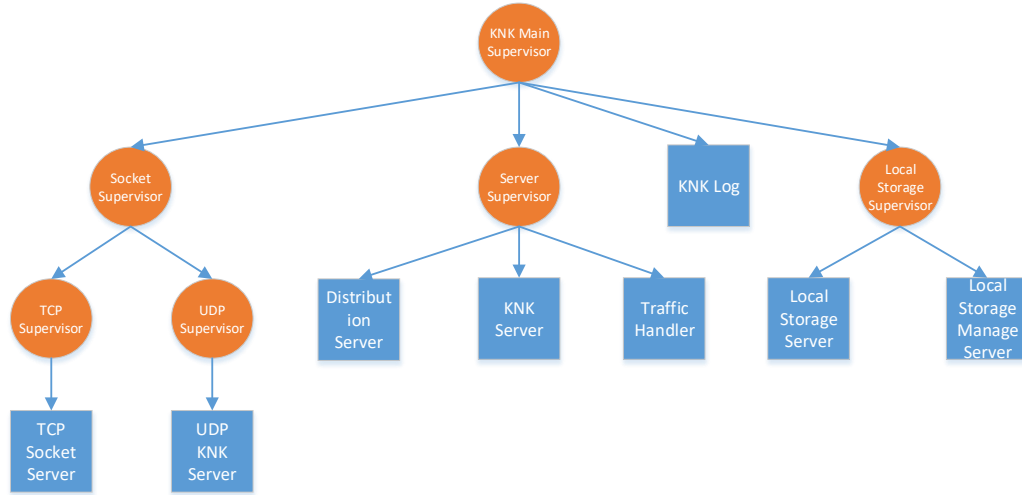


Figure 4.1: Ki-Ngā-Kōpuku Supervision Tree

4.3 Ki-Ngā-Kōpuku System Design

4.3.1 Ki-Ngā-Kōpuku System Cluster

Ki-Ngā-Kōpuku system cluster is a general group of Ki-Ngā-Kōpuku system nodes. Within the cluster, there are many application groups. Each application group consists of one single or multiple Ki-Ngā-Kōpuku system nodes, which only serve that particular application. As described in Brodtkin (2008); Singh and Chatterjee (2017), data

segregation is the main security issue for data at rest in cloud computing, so avoid multiple applications being hold by one node is helpful to reduce data segregation risk. Each Ki-Ngā-Kōpuku system node can only serve one particular application, and one single server can only host one Ki-Ngā-Kōpuku system node for one application or multiple system nodes for different applications. So the rule between application and server system nodes is, “One system node serves one application. One system node is hosted on one server. One server can hold multiple system nodes that work for multiple applications.”

The reason for a Ki-Ngā-Kōpuku system node serves one specific application is that Ki-Ngā-Kōpuku works like a typical Client-Server structure that provides ports for the application interface to connect. Moreover, a Ki-Ngā-Kōpuku system node can also become the Server for application components that belong to the same application. In which case, application components become Clients to a Ki-Ngā-Kōpuku node. Erlang’s `gen_server` behaviour is effective in Client-Server structure. Based on the ADS described in (Coronado-Garcia et al., 2011), each component within the system is independent, and each one of them is able to calculate the result by themselves. As a result, each Ki-Ngā-Kōpuku system node is the same as long as they are working for the same application. Consequently, developers can make their application interfaces connect to any Ki-Ngā-Kōpuku system node without worrying about what components are working under a Ki-Ngā-Kōpuku node. On the other hand, because Ki-Ngā-Kōpuku requires local component information for application message passing, managing components that work for the same application can reduce the complexity of the system, and the rest of the structure can’t be affected by another application’s failure.

4.3.2 Application Group

As mentioned in Ki-Ngā-Kōpuku system cluster, one application group is composed of one or more Ki-Ngā-Kōpuku system nodes that only serve that application. Group members can be hosted on one server or the whole network. In this research, Ki-Ngā-Kōpuku relies mostly on broadcasting traffic for communication, so one system port should remain open for Ki-Ngā-Kōpuku. Erlang programming language is great for handling massive concurrency traffic, one application group can be set to one specific port to receive broadcast traffic. Alternatively, the whole cluster can be made to listen to the same port, which sends and receives broadcast traffic through the whole Ki-Ngā-Kōpuku cluster.

4.3.3 Network Topology

Ki-Ngā-Kōpuku is a distributed system, so multiple servers should be included in the network design. As shown in Fig. 4.2, servers with Ki-Ngā-Kōpuku installed are called Ki-Ngā-Kōpuku group network. Servers within the group network handle requests from the outside world and play different roles according to the developer's design. The Ki-Ngā-Kōpuku group network can also be called Ki-Ngā-Kōpuku cluster. However, the cluster is different from other clusters in general purpose. The reason is that the general cluster is always designed to serve one specific purpose, such as cache cluster and database cluster. However, Ki-Ngā-Kōpuku cluster includes features of the general cluster because each single server in Ki-Ngā-Kōpuku cluster is highly flexible. Each server in Ki-Ngā-Kōpuku cluster can run one single part of a service of one application or multiple services of multiple applications at the same time, so that its serving ability, performance, and redundancy are stronger than common clusters.

Two general activities are also described in Fig. 4.2. As Ki-Ngā-Kōpuku cluster is flexible, it can make new servers join in once new servers have Ki-Ngā-Kōpuku

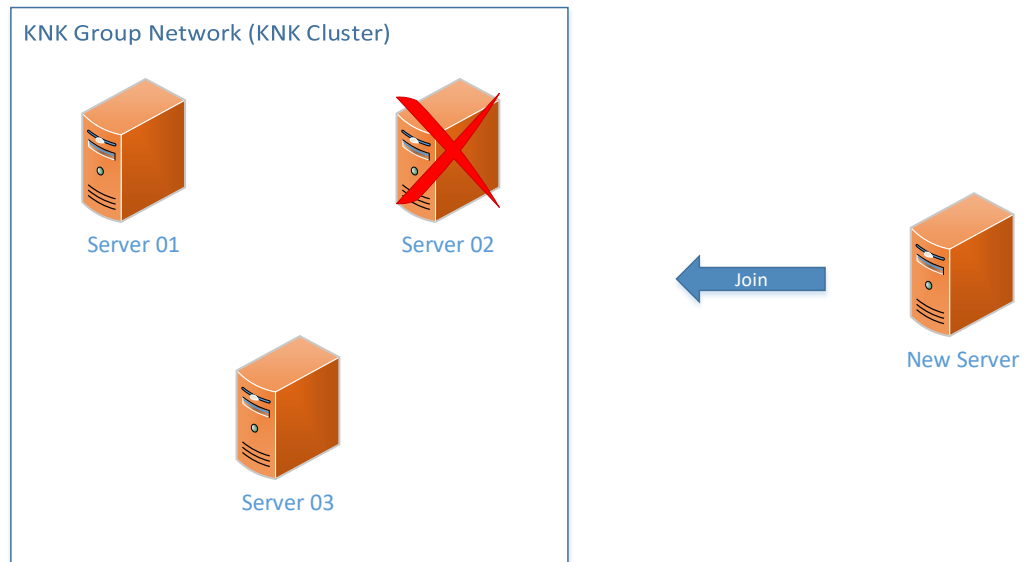


Figure 4.2: Basic Network Topology

installed. On the other hand, if users want to remove some servers, or some servers are undergoing severe unrecoverable problems, the cluster can delete those servers to keep the whole cluster healthy, which is like crossing out server 02 in the network topology figure.

4.3.4 System Architecture

Single Server Architecture

Ki-Ngā-Kōpuku on each server can be different because of the random ability. To give a general view of Ki-Ngā-Kōpuku on one single server, Fig. 4.3 below is used.

On one single server, there are two main Ki-Ngā-Kōpuku parts. One is Ki-Ngā-Kōpuku node; another is the application component. As shown in Fig. 4.3, Ki-Ngā-Kōpuku nodes and application components are connected to form one service on the server. Inspired by the decentralized system architecture in (Coronado-Garcia et al., 2011), Ki-Ngā-Kōpuku nodes act like autonomous controllers for each application

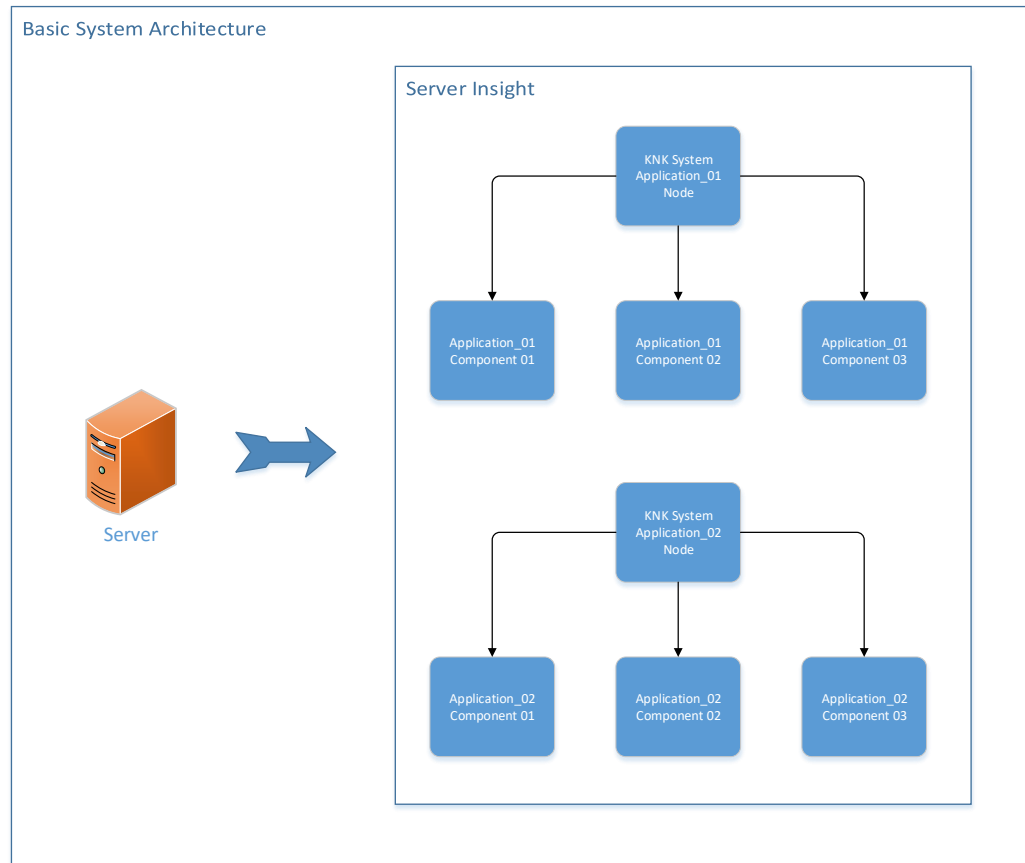


Figure 4.3: Basic System Architecture

component. Moreover, applications components can work together even if the location is different. One Ki-Ngā-Kōpuku system node is designed for one specific application, which means if one server wants to run five different application services, then that server must have five Ki-Ngā-Kōpuku system nodes for the applications. Application components have to be controlled by one Ki-Ngā-Kōpuku system node that belongs to the same application. By following this design, different applications are controlled by different system nodes, so that each application is not affected by other applications on the server.

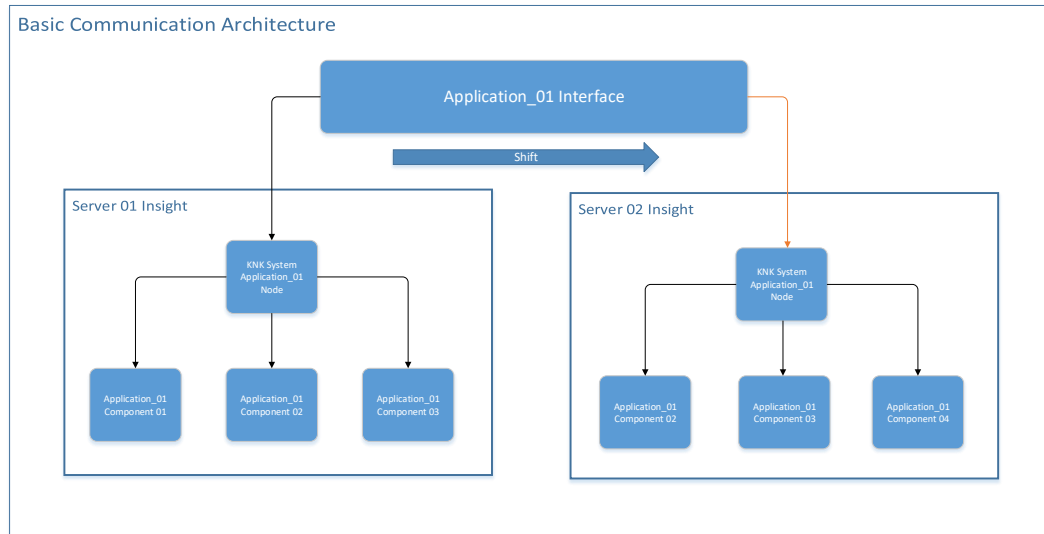


Figure 4.4: Basic Communication Architecture

Basic Communication Architecture

In Fig. 4.4, Ki-Ngā-Kōpuku lies between the application interface and application components. It is the pathway to pass application traffic between frontend and backend. However, each Ki-Ngā-Kōpuku system node plays an equal role for the application, which means that no matter which Ki-Ngā-Kōpuku system node the application interface connects to, the application communication won't be affected.

4.4 Ki-Ngā-Kōpuku System Interface

The system interface for Ki-Ngā-Kōpuku is the command-line interface, which has no GUIs. Since it is running in the Linux environment and using shell scripts, creating the system interface is a good option. In the shell script, the Linux environment can run the Erlang command to perform specific functions, such as starting the Erlang shell and Erlang node. On top of that, the shell script can also automate the process to compile all code in Ki-Ngā-Kōpuku and prepare the necessary working environment, so that

using the shell script can be very convenient for users.

4.4.1 System Action

As the system interface interacts with Ki-Ngā-Kōpuku, so the interface should provide all options. The following describes what users can do through the system interface.

1. Prepare Environment

Preparing the working environment for Ki-Ngā-Kōpuku is necessary before starting Ki-Ngā-Kōpuku. First, all source code related to Ki-Ngā-Kōpuku has to be compiled and stored in a specific directory. After that, the Erlang runtime system will read the configuration source file of Ki-Ngā-Kōpuku. The application configuration source file of Ki-Ngā-Kōpuku is called `knk_app.app`, which contains the necessary system parameters and the entry point to start the whole system.

The following parameters have to be configured in the application source file: `app`, `app_port`, `knk_port`, `log`, `data`, `app_dir`, and `nic`.

app The name of the application that Ki-Ngā-Kōpuku works for.

app_port Application port number which is used to connect to the application's interface to receive the application's data.

knk_port Ki-Ngā-Kōpuku port number which is used for system communication within Ki-Ngā-Kōpuku cluster.

log Directory to store log files.

data Data directory, which stores the ETS table and necessary system files.

app_dir Application directory, which stores application source code and relative files. The compiled application source file should be stored in this directory.

nic Network interface which used by Ki-Ngā-Kōpuku.

2. Start Ki-Ngā-Kōpuku

After the working environment is prepared, users can start Ki-Ngā-Kōpuku. There are three ways to start the Erlang application. One way is to enter the Erlang shell, and type `application:start(knk)`. Another way is typing a command on the Linux system: `$erl -pa ebin -eval "application:start(knk)"`. The third way is to use the shell script `still` to run the Erlang command. However, it is slightly different from the previous way. The third way is to type the command `$erl -pa ebin -s knk_app start` in the Linux system. In general, the last two ways are preferred in Ki-Ngā-Kōpuku interface because they are convenient for working with multiple, customised parameters because users can work with different parameters according to their requirements.

When Erlang is told to start an application, it will look for the application source file `knk_app.app` to find the entry point to start an application. In `knk_app.app`, the entry point is configured as `{mod, {knk_app, []}}`. It means Erlang will look for the module named `knk_app`, and run the `start()` function to start the whole application. To start Ki-Ngā-Kōpuku properly, the node name and node cookie have to be passed as parameters to the system interface. The complete command to start Ki-Ngā-Kōpuku will be like this: `erl -pa ebin $nodeName -setcookie $cookie -s knk_app start`.

‘nodeName’ and ‘cookie’ are the parameters that are passed to the shell script.

The Fig. 4.5 shows what will happen when starting Ki-Ngā-Kōpuku via system interface. If users want to start Ki-Ngā-Kōpuku, the `start` option has to be passed to the system interface. Once the interface receives the `start` option, it will create one Erlang node to start Ki-Ngā-Kōpuku application. Then another Erlang node, which is a local storage node, will be created to work with Ki-Ngā-Kōpuku node.

On the other hand, Ki-Ngā-Kōpuku will be initialised when the system has started. Three factors will be initialised while starting the system: application,

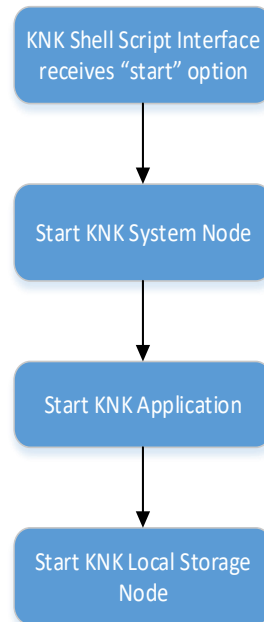


Figure 4.5: Brief Start Steps

component storage, and system status. `Application` refers to the application that Ki-Ngā-Kōpuku works for. This factor is used to distinguish different applications and related application components. As described in the system design, one Ki-Ngā-Kōpuku node has to work for one application, and nodes working for the same application can only work together. By doing it this way, the application traffic can't be affected by other applications, which reduces the burden on Ki-Ngā-Kōpuku unless cross application communication is required. Besides restricting communication between different applications, specifying the application can also help prevent unwanted application components, so that the component environment is clean.

Component storage is another factor that needs to be initialised when the system fully starts. In fact, the local storage node works with Ki-Ngā-Kōpuku node. The system node and local storage node build up the entire system. The reason to

have local storage for Ki-Ngā-Kōpuku node is to prevent shared data storage for the whole Ki-Ngā-Kōpuku cluster. As one local storage node only works for one system node, if something goes wrong within the system it won't affect other system nodes and data. Furthermore, other system nodes can't connect to the local storage node. Only the system node that it works with can modify the data. The last factor is system status. There are three kinds of system status used in Ki-Ngā-Kōpuku: `new`, `active`, and `running`. The following indicates details of each system status:

New System can run, but has no application components.

Active System can run, but its registered application components are not running.

Running System is running with registered components.

In this case, the system status will become `new` when it is started for the first time.

3. Stop Ki-Ngā-Kōpuku System

When users want to stop Ki-Ngā-Kōpuku, the option `stop` has to be passed to the system interface. Once the `stop` option is received, then the shell script will tell the Erlang system to run the `application:stop(knk)` command to stop the whole Ki-Ngā-Kōpuku.

As Ki-Ngā-Kōpuku uses memory storage to store necessary data in the system, so data will be lost when the system is shutdown. To avoid data loss when stopping the system, the local storage node, which holds necessary data for Ki-Ngā-Kōpuku, will save data on the disk and flush the memory before the system shuts down. In this case, Ki-Ngā-Kōpuku can still work properly when it restarts.

4. Request Components

Typing command `request_comp` in the system interface starts the request for

application components. When the interface receives `request_comp`, it will first check the system status. Only systems with `new` system status are allowed to request application components from the whole network.

4.5 Traffic Handler

Handling traffic between the different nodes and maintaining communication with the other systems are very important to Ki-Ngā-Kōpuku. Due to Ki-Ngā-Kōpuku distribution feature, multiple nodes with different usage purpose are bundled together to make the system operate successfully. As a result of that, managing and maintaining traffic among the different nodes should be well designed.

The traffic handler is receiving socket message or node message from other servers or nodes and then taking operations to the traffic according to specific rules or patterns. Even more, the traffic handler can also send messages to one specific destination, or broadcast message to all servers in the whole network.

4.5.1 General Process

As shown in Fig. 4.6, the traffic handler will receive any incoming data and parse data according to any pre-defined message patterns or routing rules. The traffic handler is the message hub for Ki-Ngā-Kōpuku and it manages all process behaviours and traffic directions in the system. Consequently, traffic coming from the socket or node should be passed to the traffic handler.

4.5.2 Message Pattern

Message patterns are critical in the Erlang programming language when dealing with messages (*Concurrent Programming*, 2017). Messages are the communication traffic

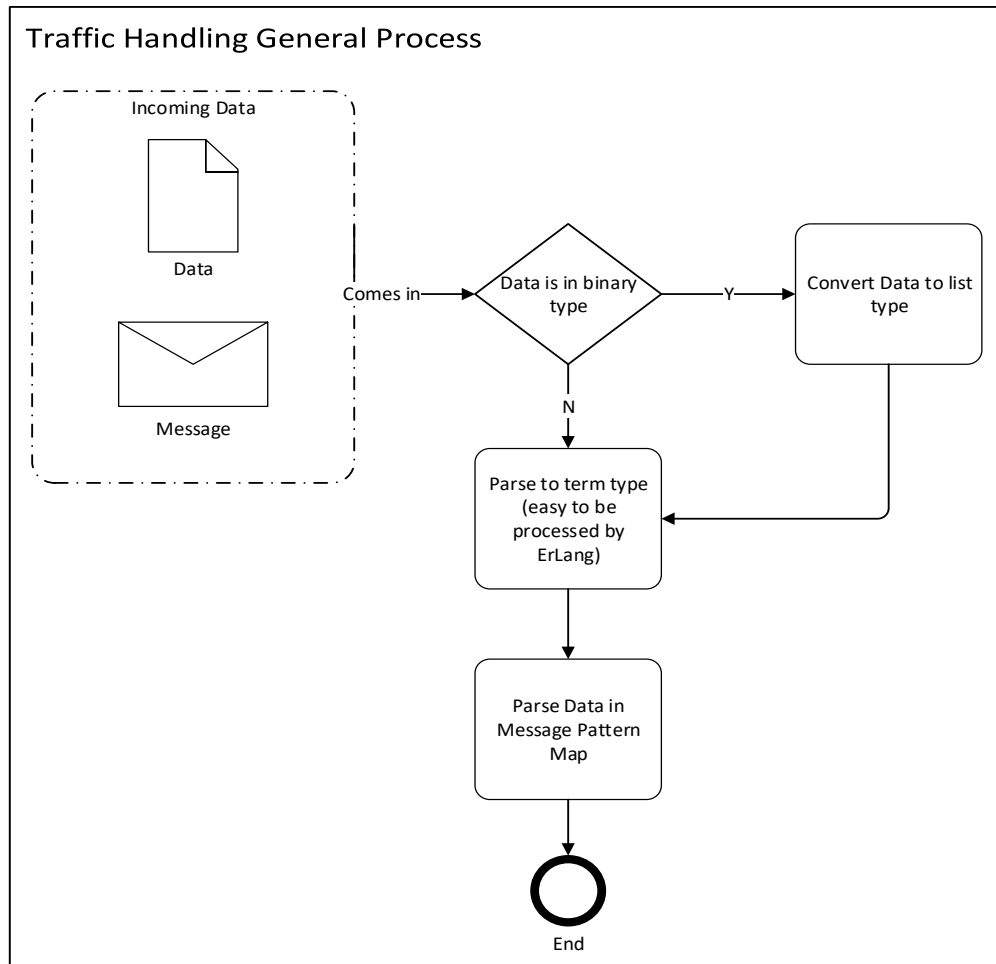


Figure 4.6: Traffic Handler Process

between processes. When a message arrives at one process, that process can be used to pre-define message patterns to quickly process data and operations. In addition, message patterns limit the kind of data that the process receives. This means the process is set to receive specific messages that can reduce the pressure of developers or processes handling unexpected messages.

The traffic handler processes two main types of message. One type is `sys`, another type is `socket`. The `sys` type message is the system traffic, which can be traffic between nodes or traffic within the system itself. The `socket` type message comes

from the port that Ki-Ngā-Kōpuku listens on. The following example code shows how the traffic handler processes incoming data:

```
message_map(Data) ->
  {DataType, DataContent} = Data,
  case DataType of
    sys ->
      [DataRequest, DataInfo] = DataContent,
      case DataRequest of
        register_comp ->
          handle_register(DataInfo);
        request_distribute ->
          knk_socket:send_bcast({socket, [request_distribute,
            DataInfo]});
        tar_transferred_info ->
          knk_socket:send_ucast({socket, [tar_transferred_info,
            DataInfo]});
      end;
    socket ->
      [DataRequest, _DataInfo] = DataContent,
      case DataRequest of
        request_distribute ->
          handle_distribute_request(DataContent);
        tar_transferred_info ->
          handle_tar_transferred_info(DataContent);
      end;
  end.
```

As shown in the example code, the traffic handler uses the `message_map` function to receive requests. The message type will be checked, and then the traffic handler will take actions according to message content. For example, if the traffic handler receives a request `{sys, request_distribute}`, it will broadcast, or distribute, the request among all of the network. The reason that the request has `sys` tag is that the distribution request is generated by the system itself. After the traffic handler directs the distribution request to its socket process, the request will be wrapped with the `socket` tag because it is going to be sent to the outside world. Once the traffic handler for other Ki-Ngā-Kōpuku systems receives the distribution request from the socket, it will

process the distribution request.

4.6 Application Components

In this research, Ki-Ngā-Kōpuku is designed to provide redundancy and availability to the user's applications. To achieve these goals, it requires users to undergo application componentisation before fully implementing their applications in Ki-Ngā-Kōpuku. As Ki-Ngā-Kōpuku is written in the Erlang programming language, each application component will play a different role when integrated with Erlang to build up the distributed system. The role for each component is Node.

Each node in Erlang is a single unit working in the whole system, and the communication between each node is convenient and flexible. In this case, the user's application should be able to work with Erlang, and each application component should work like a Node in Ki-Ngā-Kōpuku.

4.6.1 Component Definition

Application componentisation involves separating one single application into several small pieces, and each piece can effect computations by using its functions or methods. Consequently, one single application component can finish one task without any help from other components. To work with Ki-Ngā-Kōpuku, each application component should have a node name and their own major process. The node name is the identity of each application component, which should be unique on one single server because Ki-Ngā-Kōpuku will use the node name to find specific components (*Distributed Erlang*, 2017). On the other hand, the major processes running on each component will be used to exchange messages between Ki-Ngā-Kōpuku and other application components. In general, one single application component can work like a mini-server, and Ki-Ngā-Kōpuku plays a role to establish the connection between these mini-servers.

Erlang programming language has already provided APIs for multiple programming languages, the communication between different languages won't be a problem. If the application is written in Erlang, then it's a matter of simply using the command

```
erl -name nodename@ip_address
```

to start one Erlang node. If the application is written in Java, then users can import Jinterface java library, and use

```
OtpNode myOtpNode = new OtpNode("server", "mycookie");
```

to start one node.

4.6.2 Component Storage

Component storage stores all information about application components work under Ki-Ngā-Kōpuku framework. In Ki-Ngā-Kōpuku, the Erlang ETS table is used for creating the local component storage. The reason to store information locally is that distribution requires sharing information between different servers or hosts, which means the location of Ki-Ngā-Kōpuku node is known by other servers. Consequently, if hackers compromise the shared information, then all servers in the distributed system will be in danger. On the other hand, component storage can be stored with the Erlang hidden node. Common nodes in Erlang can share node information as long as they are connected to each other. However, Erlang hidden nodes can hide their information when nodes connect to them. In this case, using hidden nodes can prevent the node information from being exposed to unnecessary operations.

-hidden option is required to start hidden node, and the command is

```
erl -name nodename@ip_address -hidden
```

There are four ETS tables used to store application component information: App table, Component table, Component Attributes table, and Component Distribution Count

table. The App table stores the name of the application that Ki-Ngā-Kōpuku works for. The component table stores a list of components that are working under Ki-Ngā-Kōpuku. The Component Attributes table stores the name of application components, but also the node name, the functions within it, the name of the directory holding the component source file, and even the command to start the application components.

1. Table Structure

- App Table Data Structure: {app, {AppName}}.
- Component Table Data Structure: {comp, [CompName]}.
- Component Attributes Table Data Structure: {comp_atts, CompName, CompNode, [{FuncName, N}], CompSrc, RunEnv}.
- Component Distribution Count Table Data Structure: {comp_dist, {CompName, CountNumber}}.

2. Component Attribute Format Example

To give a more direct view of component attributes, the `calculator` application is used in this section. In the real world, one calculator can receive numbers typed by users and calculate results according to different operations. In this example, addition and subtraction are used. The addition function will receive two integers and add both numbers. The subtraction function is similar to the addition function; however, instead, it will use one integer to minus another one. The addition and subtraction function in Erlang will be like this `add(X, Y) -> X + Y.; minus(X, Y) -> X - Y..`

Addition and subtraction are two different functions, so that `calculator` application can be separated into an addition and subtraction component. Then the App Table data structure will be like this, {app, {calculator}}. The Component Table data structure will be like this, {comp, [calculator_addition,

calculator_subtraction]}. The Component Attributes Table data structure is

```
{comp_atts,  
calculator_addition,  
'calculator_addition@192.168.1.1',  
[{add, 2}],  
"/usr/local/calculator/src/calculator_addition.beam",  
"erl -s calculator_addition start"}
```

4.6.3 Component Action

Component Type Identification

Identifying types of application components are necessary for the component registry. There are four types of application components for Ki-Ngā-Kōpuku. As shown in Fig. 4.7, the component table, component attributes table and component distribution count table are checked to identify component types. In general, if one component can't be found in any of these tables, then it is very new to the cluster. If one component can't be found in the component table and component attributes table, but the distribution count table has a record of it, then the component is new to the local Ki-Ngā-Kōpuku system. The reason is that the component and component attributes table are local tables that store component information in local environments. However, the distribution count table records application components information among the whole network, so components recorded in the distribution count table indicate that these components exist outside of a local environment. If a component can be found in all these tables, then its attributes will be checked because Ki-Ngā-Kōpuku doesn't want to have duplicate application components running under its management.

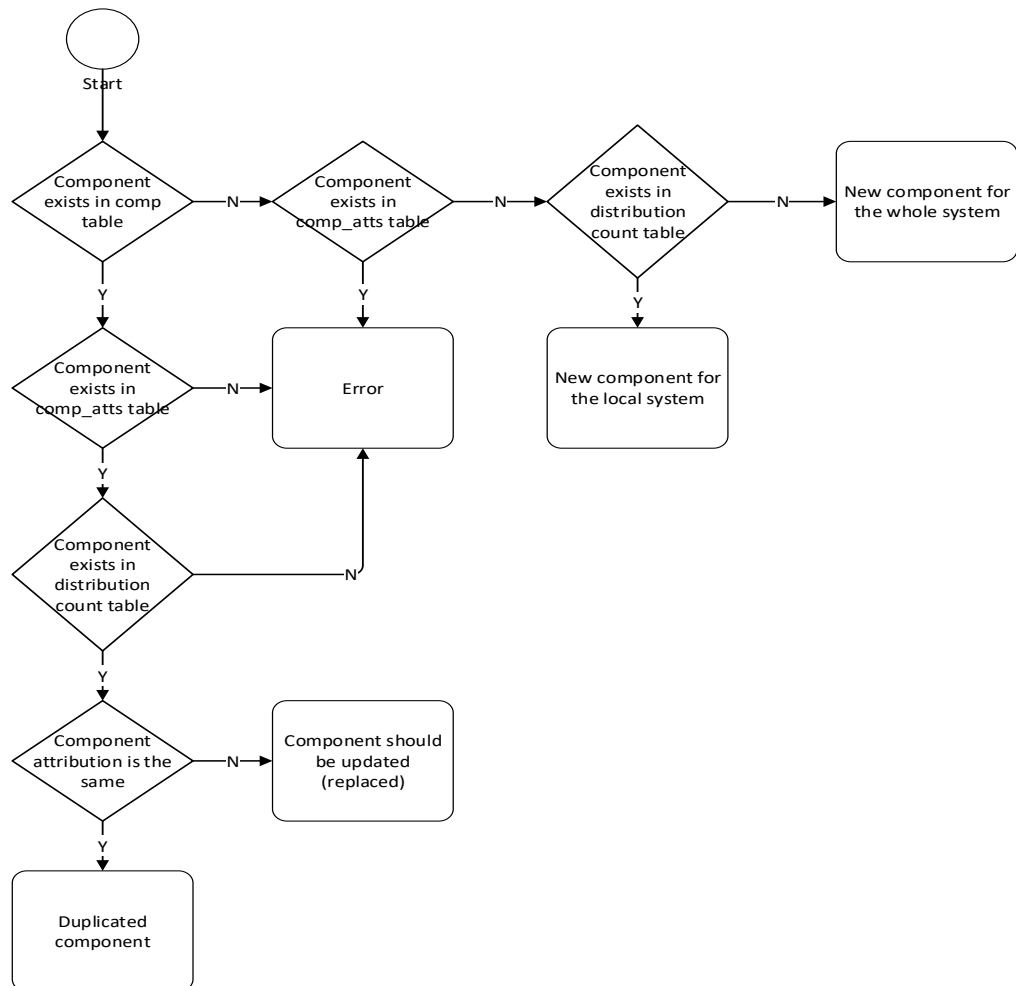


Figure 4.7: Component Type Identification

Component Registry

The component registry is the process to get component information from application components and store component information into the local component storage. Components have to register their attributes in a Ki-Ngā-Kōpuku node, and then Ki-Ngā-Kōpuku can store component information in the local storage node. This process is used when users use Ki-Ngā-Kōpuku for the first time. When Ki-Ngā-Kōpuku is initialised, it doesn't have component information. In this case, users have to add

application components into Ki-Ngā-Kōpuku manually. Consequently, registering applications component is required. On the other hand, component registry is also the final approval step for component distribution. Once application components are distributed to new destinations, Ki-Ngā-Kōpuku will need to register component information in its local storage. As shown in Fig. 4.8, application components have to be registered after distributing and manually adding components.

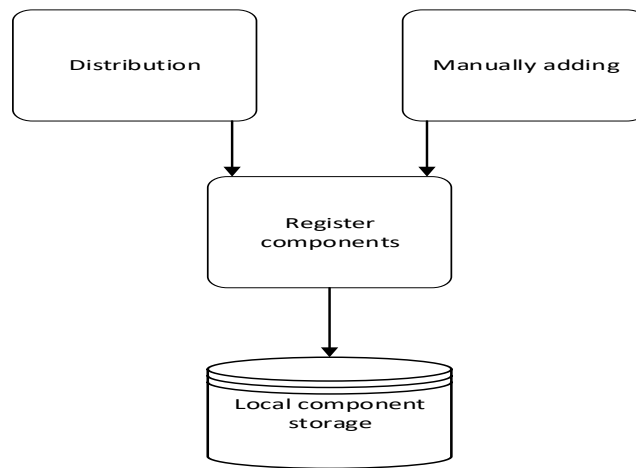


Figure 4.8: Component Registry Approval

For registering components in Ki-Ngā-Kōpuku, the component has to send a message to Ki-Ngā-Kōpuku socket, which is listening to a specific UDP port. Registered message sent from application components contain the name of the application that it belongs to, the properties of functions within it, the source file location, the node name, and a script for starting itself. Once Ki-Ngā-Kōpuku receives component information, it will test the basic connection of application components. If the connection fails, then Ki-Ngā-Kōpuku will report the error and reject this registration process.

The component register message structure is

```
{register_comp, [AppName, CompName, [{Fun_01, N}, {Fun_02, N}], NodeName, CompSrc]}.
```

The register message contains the message type: `register_comp`, and a list of component attributes. Ki-Ngā-Kōpuku will identify the message type first. If the message type equates to `register_comp`, then it will receive the component attributes one by one. Ki-Ngā-Kōpuku will check the `AppName` in component storage to make sure of the existence of the application. If the application does not exist in the storage, then Ki-Ngā-Kōpuku will ignore the registration message. If `AppName` can be found in the component storage, then Ki-Ngā-Kōpuku will check any component duplication according to `CompName`. If the component name is duplicated, then Ki-Ngā-Kōpuku will ignore the message and report duplication status. Furthermore, the node name duplication will also be checked. When `NodeName` already exists in the component storage, the system will report duplication status and ignore the registration message. The reason for rejecting the registration message is because of node duplication and the regulation that each component or node must be unique in Ki-Ngā-Kōpuku. The last component attribute `CompSrc` is the file directory, which stores the component source file. This attribute is used by Ki-Ngā-Kōpuku to validate the existence of the source file. If the source file can't be found in `CompSrc`, Ki-Ngā-Kōpuku will report an error and notify users to check the existence of the component source file.

As mentioned above, Ki-Ngā-Kōpuku needs to make sure that the system node can connect application components. To test the connection status, the temporary node will be used. The temporary node defines basic communication behaviours between components and Ki-Ngā-Kōpuku. For enabling a connection between nodes, the node cookie has to be the same on both nodes. The `net_adm:ping()` function in Erlang can be used to test basic connection between nodes. If the connection is established successfully, this function will return, `pong`; otherwise, the return message will be, `pang`. Instead of just building a connection between the components and Ki-Ngā-Kōpuku, it is also very important to test the main process availability of application

components. To do that, the node template pre-defines one simple connect function called `conn_test`, which can send and receive specific message patterns. By doing that, the temporary node can use the connect function to exchange messages with the main process running on application components. After the temporary node has finished its task, it will be shut down and deleted.

The temporary node is the proxy for a Ki-Ngā-Kōpuku node. It uses basic functions to communicate with application components and get the result required by a Ki-Ngā-Kōpuku node. Using the temporary node not only reduces the burden of the system node but also improves the security of Ki-Ngā-Kōpuku because it is the gateway to blocking unhealthy or malicious components.

Fig. 4.9 shows application components registry process according to component types.

Fig. 4.10 shows how Ki-Ngā-Kōpuku processes application component registry messages. Once Ki-Ngā-Kōpuku receives the registration request from application components, it will start the component registry process. If the register is successful, then the system will send message to all other Ki-Ngā-Kōpuku systems notifying them to update the distribution count table because it is related to the component distribution status in the cluster.

4.6.4 Component Communication

Since the user's application is separated into different components that individually hold different functions, then the communication between each component is key to making those components work as one whole application. A communication method is introduced in Peyravian et al. (2002). The method is to generate keys and relative signature on all nodes, so that other nodes can use these key information to decrypt data sent from that node. Since Ki-Ngā-Kōpuku is implemented between user interfaces

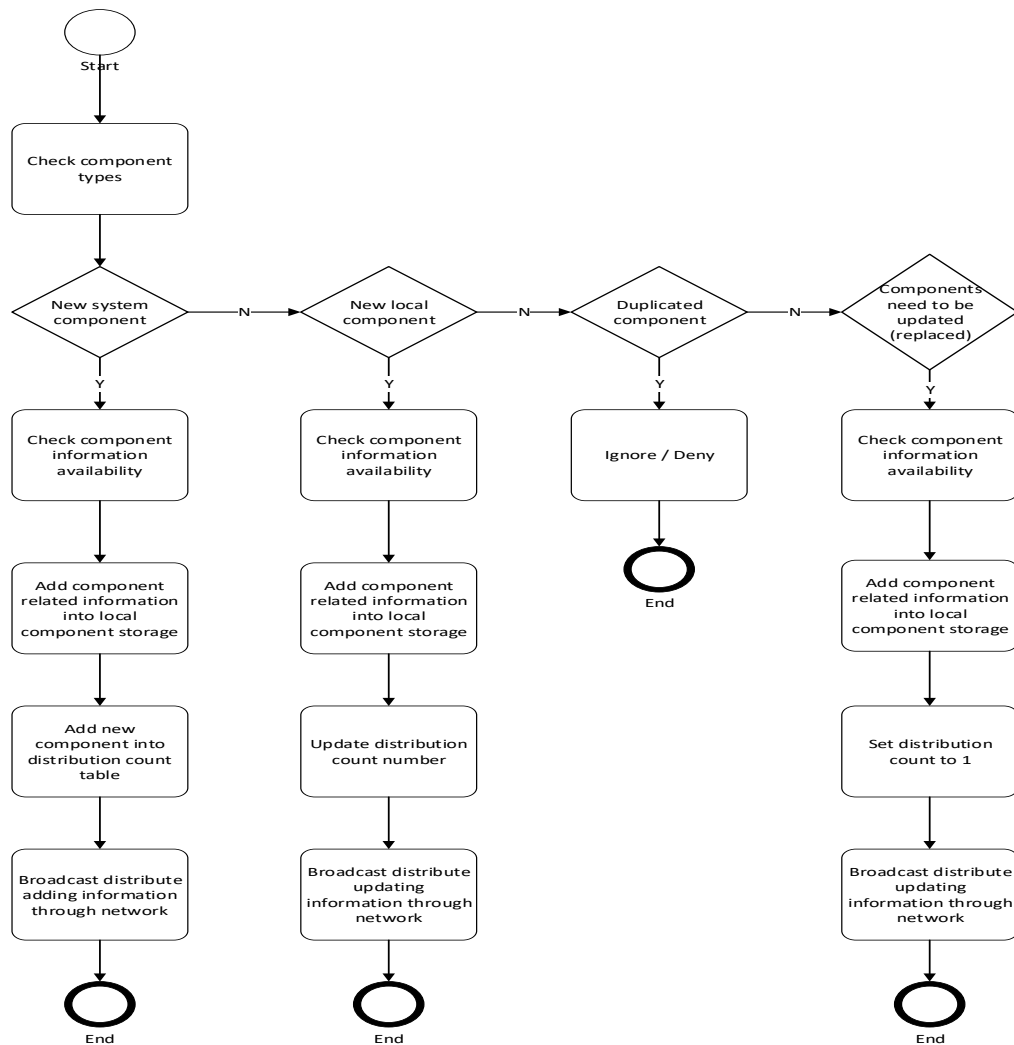


Figure 4.9: Register Component via Types

and actual application services (components), and one Ki-Ngā-Kōpuku system can hold multiple application components at one time on one single server, and it is not necessary to waste resources on other nodes, so a Ki-Ngā-Kōpuku system has to build some communication tunnel with other Ki-Ngā-Kōpuku systems on other servers.

For security reasons, each server is isolated from each other. This means that one Ki-Ngā-Kōpuku system on one server shouldn't know what components are working on other servers. However, the user's request coming from the outside world might need multiple application services and functions for support because of complex business

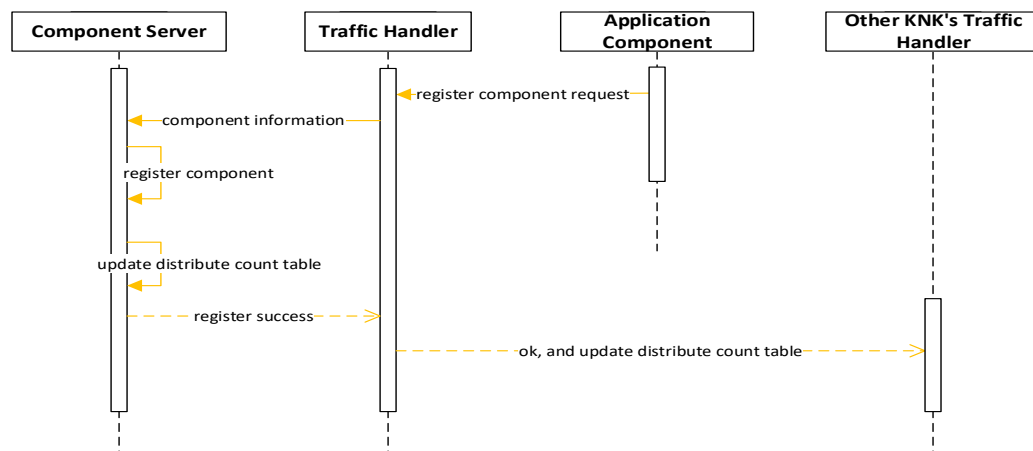


Figure 4.10: Component Registry Message Processing

logics. It is highly possible that one server will not have all application components on it. Hence, Ki-Ngā-Kōpuku has to be able to find those required components in the whole network and transfer data to them to continuously process the user's request. For the redundancy of applications components, it is possible to handle multiple instances of the same application component on one single server, but this requires Ki-Ngā-Kōpuku uses the inter-balancing techniques or more advanced algorithms to pick up one component to use among all related component instances, which makes Ki-Ngā-Kōpuku complex. As a result of that, in the current phase, no duplicated application components should be existed on a server, so that the redundancy of application components are relied to other servers in the network.

Locating Components

A distributed system has to be fast to react to user's request Ahmed and Wu (2013). The Erlang programming language uses nodes to build up a distributed system. When a node connects to another node, it will not only know the information of the connected node but also knows the existence of other nodes that connect to that node. Since

Ki-Ngā-Kōpuku system nodes have to require information from the outside world and other Ki-Ngā-Kōpuku system nodes, it is highly risky to keep Ki-Ngā-Kōpuku nodes connected to each other. If each Ki-Ngā-Kōpuku system node is connected, then the connection information will be exposed if hackers compromise one of Ki-Ngā-Kōpuku nodes.

Given that Ki-Ngā-Kōpuku systems are not connected to each other, Ki-Ngā-Kōpuku implements two ways to help locate components in the whole network. One is `local first`; another one is `UDP broadcasting`. In situations, when a request comes from a user interface, a Ki-Ngā-Kōpuku node will find relative functions that are required to process the request. If the component that holds the required function on the local server is detected, the request will be directed to that local component. This is called `local first`. If local components don't have required functions, then Ki-Ngā-Kōpuku will send broadcast message through the UDP port and try to find required functions on other servers in the network. If other servers happen to have the required functions, then remote application components will receive the request and continue to parse it.

Component Intercommunication

Application components don't talk to each other directly because Ki-Ngā-Kōpuku itself holds all of the information about components. In this case, a Ki-Ngā-Kōpuku node is the only bridge for application components to talk to the outside world.

Ki-Ngā-Kōpuku has to provide APIs that can be used by application components to pass traffic to a Ki-Ngā-Kōpuku node so that application components can exchange messages and work as a whole application. Each component in Ki-Ngā-Kōpuku uses its node name to represent its unique network location. As a result of that, the node name are used in for component communication. Moreover, calling functions on a remote node normally requires the name of the remote component and the remote function

name.

As shown in Fig. 4.11, application component A wants to get a result from application component B, which is located in another Ki-Ngā-Kōpuku system. Thus, component A has to tell its Ki-Ngā-Kōpuku system node to find the location of component B. To tell a Ki-Ngā-Kōpuku node to find required components; application components can use Remote Procedure Call (RPC) in the code. RPC is used when an operation has to be executed for a remote address. In these cases, application components can use RPC call-specific functions in Ki-Ngā-Kōpuku.

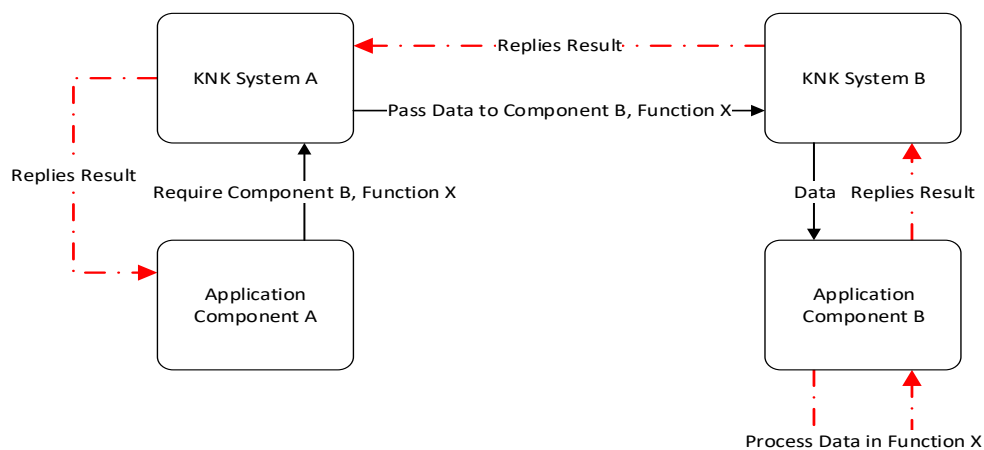


Figure 4.11: Component Communication

The following code shows how to require functions on a remote node in Erlang:

```
require_fun(ServerNode, CompName, FunInfo, Data) ->
rpc:call(ServerNode, simple_handler, require_comp,
[node(), CompName, FunInfo, Data]).
```

The `require_fun` function requires four parameters: the name of its Ki-Ngā-Kōpuku system node, the name of application components that it wants to communicate with, the name of functions that it wants to use, and the data passed to functions. This function uses the `rpc` module in Erlang to call the `require_comp` function in the

`simple_handler` module which runs on a Ki-Ngā-Kōpuku node. When a Ki-Ngā-Kōpuku node receives information from this function, it will search components and pass data to them.

On the other hand, each Ki-Ngā-Kōpuku system spawns multiple temporary processes to handle traffic from application components. When a component requires information from a remote application component, a Ki-Ngā-Kōpuku node will use a temporary process to handle the traffic so that the request status won't affect the node itself.

4.7 Component Distribution

In Ki-Ngā-Kōpuku, distribution happens when the system receives distribution requests from other Ki-Ngā-Kōpuku systems. Distribution functions in Ki-Ngā-Kōpuku not only deal with code but also take actions on the data that is stored in local storage. The general purpose of distribution is to deliver the components of a user application to a new destination, then make the new destination act like a part of the user application. In this case, when some servers for that application can't work as usual, other servers may still be alive that hold the same function of those broken servers so the whole application won't be affected by server failure.

Since the user's application can be written in different programming languages, and distributing application components is a difficult task to do, so Ki-Ngā-Kōpuku focuses on providing a general way for distributing compiled source code for different programming languages.

4.7.1 Distribution Factors

Distribution Count Table

Ki-Ngā-Kōpuku uses one table to count distribution numbers for each application component. This table only stores the component name and total number that the component has been distributed. When Ki-Ngā-Kōpuku receives a distribution request, it will use the distribution count table to decide which components should be distributed. Components are chosen by sorting the number of each component in the distribution count table. Components with lower number will be chosen for distribution. The reason is that lower number means the component has fewer replicas in the whole cluster so that distributing those components with lower numbers can guarantee the balance between each component. When Ki-Ngā-Kōpuku decides to distribute one component, and that chosen component is successfully distributed, then the count number of that component will be updated on all Ki-Ngā-Kōpuku systems.

Local Storage Node

The local storage node stores the information from each local application component. Once the distribution request is received, Ki-Ngā-Kōpuku will retrieve the component information from the local storage node. In current research, there are no efficient ways to transfer components to different destinations because applications can be written in different programming languages. In this context, transferring compiled files can be a solution for component distribution. Besides preparing components for distribution, the local storage node will also store component information once new application components are registered.

Linux Tools

For distributing application components, some Linux tools are used to perform critical tasks in the distribution process. These Linux tools are SSH, MD5, TAR, and RSYNC. MD5 is used to calculate the hash value for each component. TAR will compress components into one file so that they are easy to manage and can reduce the bandwidth burden. SSH and RSYNC are used to transfer compressed component tar file to new servers. Currently, a password-less SSH connection has to be established so that there is no need to type in a login password when using the SSH command.

4.7.2 General Distribution Steps

As shown in Fig. 4.12, there are seven steps for the applications component distribution process.

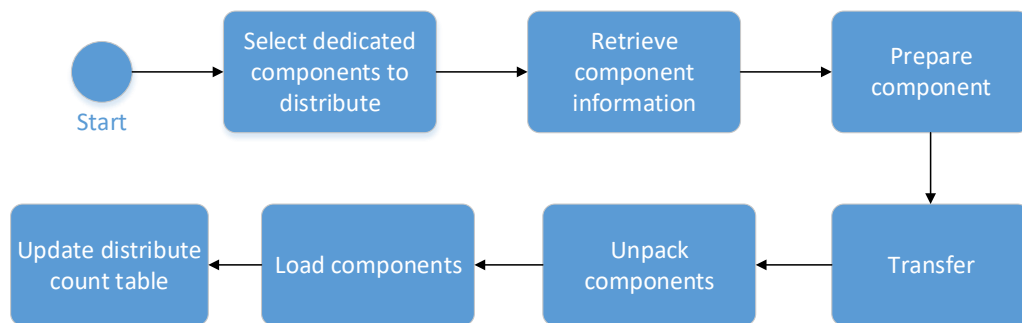


Figure 4.12: General Distribution Steps

The following shows details of seven general steps in the distribution process.

1. Select dedicated components to distribute.

Ascending the distribution count table, and gather all components with the lowest (or lower) number. Then ascertain the total number of components that should be distributed. The distributed number (N) is defined by users, which means Ki-Ngā-Kōpuku will get the first N components to distribute.

2. Retrieve component information.

Search the component name in component storage, and retrieve the source directory of each component and compiled code (for the current version). If no components are found in local storage, then the request is ignored.

3. Prepare component.

Compress the necessary component files as well as the required boot up script or procedure used for checking the integrity of components, starting components, and registering components.

4. Transfer.

For the current version, SSH and RSYNC should be used. SSH is used for remote access, and Rsync is used for transferring component files.

SSH needs servers to establish the password-less SSH connection between destination servers. This requires both servers to provide the username and password or certificate for SSH.

5. Unpack components.

Unzip component files, and check the MD5 hash value. If the hash value is the same, then the component is ready to load and start. If the hash value is not the same, then the component files are dropped and return message are sent to other servers.

6. Load components (depends on Step 5)

Following the load steps to start distributed application components.

7. Update distribute count table.

If distributed components successfully load on the new server, then distributed components should be registered in the local component storage. Then the distribution count table will be updated. The component receiver will send broadcast messages notifying that distribution has succeeded. After that, other Ki-Ngā-Kōpuku system nodes will receive distribution success messages and

update their own distribution count table.

4.7.3 Distribution Workflow

Two distribution workflows are used for the distribution requestor and distribution sender. The distribution requestor is Ki-Ngā-Kōpuku that sends distribution requests to the whole network. The distribution sender is Ki-Ngā-Kōpuku that receives distribution requests and transfers application components to the requestor.

Distribution Requestor

As shown in Fig. 4.13, the distribution process starts when the requestor sends a distribution request to the whole network while the server is new to Ki-Ngā-Kōpuku cluster or attempting to request application components from other servers. After sending the request, the requestor has to wait for senders to send compressed application component files. However, it is hard for requestors to detect new files coming to the system because requestors won't know the exact time that those compressed files came to the system. On the other hand, because the distribution request is broadcast to the whole network, the requestor will receive multiple compressed component files sent from different Ki-Ngā-Kōpuku systems. The requestor won't use all of those files; it will randomly select one file to use. However, the requestor won't know how many compressed files will come, and when those files will come.

To be able to detect new compressed application components files the requestor will start one process to check one ETS table, and another process to monitor the check result. The ETS table checked by the process is called `selection_pool`, which is used to receive messages from distribution senders. The message contains the information about the compressed file that is transferred by distribution senders, which means when the sender successfully transfers the file to the requestor, it will send the file information

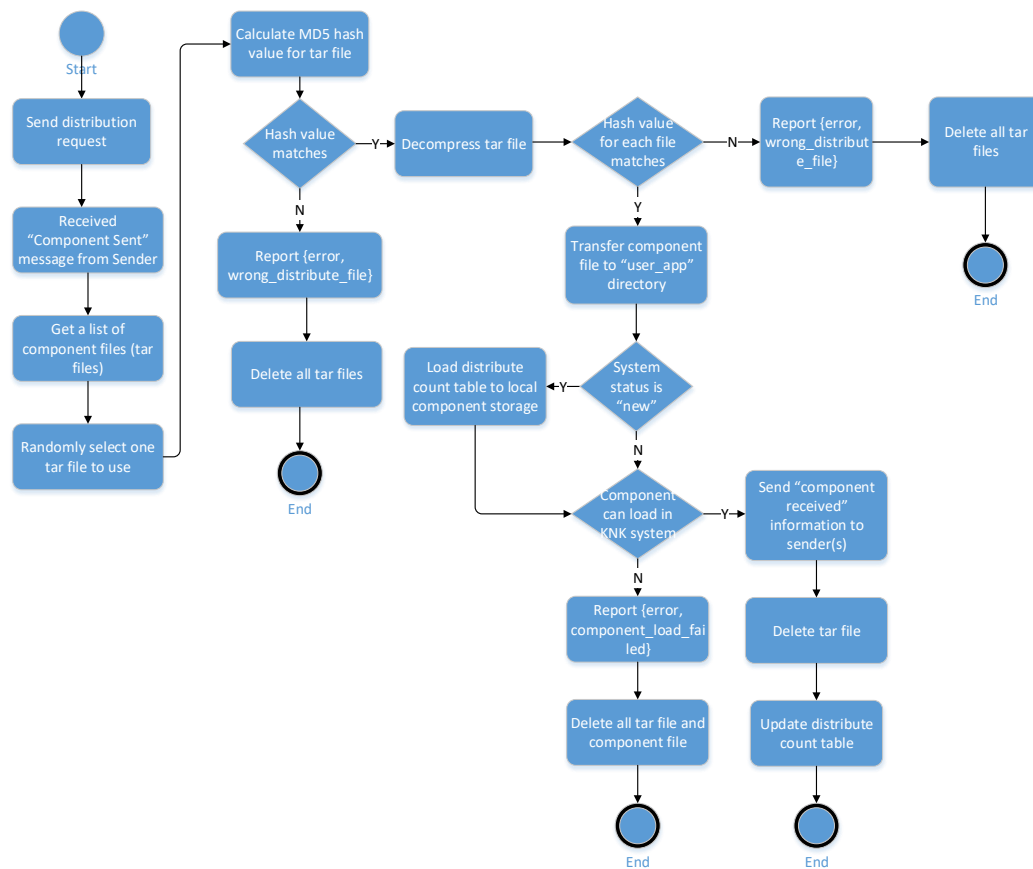


Figure 4.13: Distribution Requestor Workflow

and the requestor will store the information in the `selection_pool` table. In this case, the requestor won't have to worry about how many files it will receive and where to find them.

The following code describes the monitor process and table checking process.

Starting monitor process and table checking process:

```

Pid = spawn_link(?MODULE, monitor_loop, [0]),
register(monitor_loop, Pid),
TabChPid = spawn_link(?MODULE, table_check_loop, []),
register(table_check_loop, TabChPid).

```

The above Erlang code shows how to start the monitoring and table checking process.

To find these processes easily, the monitor process is locally registered with the name

monitor_loop, and table checking process is named as table_check_loop.

Table checking process:

```
table_check_loop() ->
  receive
    check_table ->
      Result = gen_server:call(knk_etssrv, check_select),
      monitor_loop ! Result,
      table_check_loop();
    stop ->
      exit(normal)
  end.
```

As shown in the above code, the table checking process can receive two messages: check_table and stop. When the process receives the check_table message, it will check ETS table selection_pool, and applies the result to the monitor process monitor_loop.

Monitor process:

```
monitor_loop(N) ->
  receive
    start ->
      table_check_loop ! check_table,
      monitor_loop(N);
    '$end_of_table' ->
      send_check_signal(),
      monitor_loop(N);
    _Other ->
      if
        N < 3 ->
          send_check_signal(),
          monitor_loop(N + 1);
        true ->
          %% tells distribute process that files are coming
          send_notification(),
          table_check_loop ! stop,
          exit(normal)
      end
  end.
```

The monitor process is named as `monitor_loop` in the system, which can receive `start`, `'$end_of_table'`, and other messages. When it receives `'$end_of_table'` message from the table checking process, it means the `selection_pool` table is still empty, and no files are coming now. If the process receives messages other than `start` and `'$end_of_table'`, it means that there are files that have been transferred to the server. In this case, the monitoring process will still run for a while to give more time to the system to receive more files. If the monitoring process reaches the limit, then it will stop itself and send a signal to the table checking process to stop as well. Consequently, Ki-Ngā-Kōpuku will randomly select one file from the `selection_pool` table to use.

After selecting the compressed application components file to use, the requestor will calculate the hash value of that file and request the hash value of the selected file from the sender to check the file integrity. If the hash value of the compressed file doesn't match, then the file will be deleted, and the whole distribution process will be stopped. If the hash value matches, then it will be uncompressed, and Ki-Ngā-Kōpuku will check the integrity of component files within it. When all files pass the integrity check, the requestor is ready to start those components and register the component information in its local storage node; then the whole distribution process is finished.

Distribution Sender

The distribution workflow used for the distribution sender is different from the requestor because senders have to prepare the application components files and necessary elements for use by the requestors.

Fig. 4.14 describes the distribution process for senders. In general, two main tasks have to be done from the sender side. One is selecting which components should be distributed to the requestor; another task is transferring those application components files to destinations.

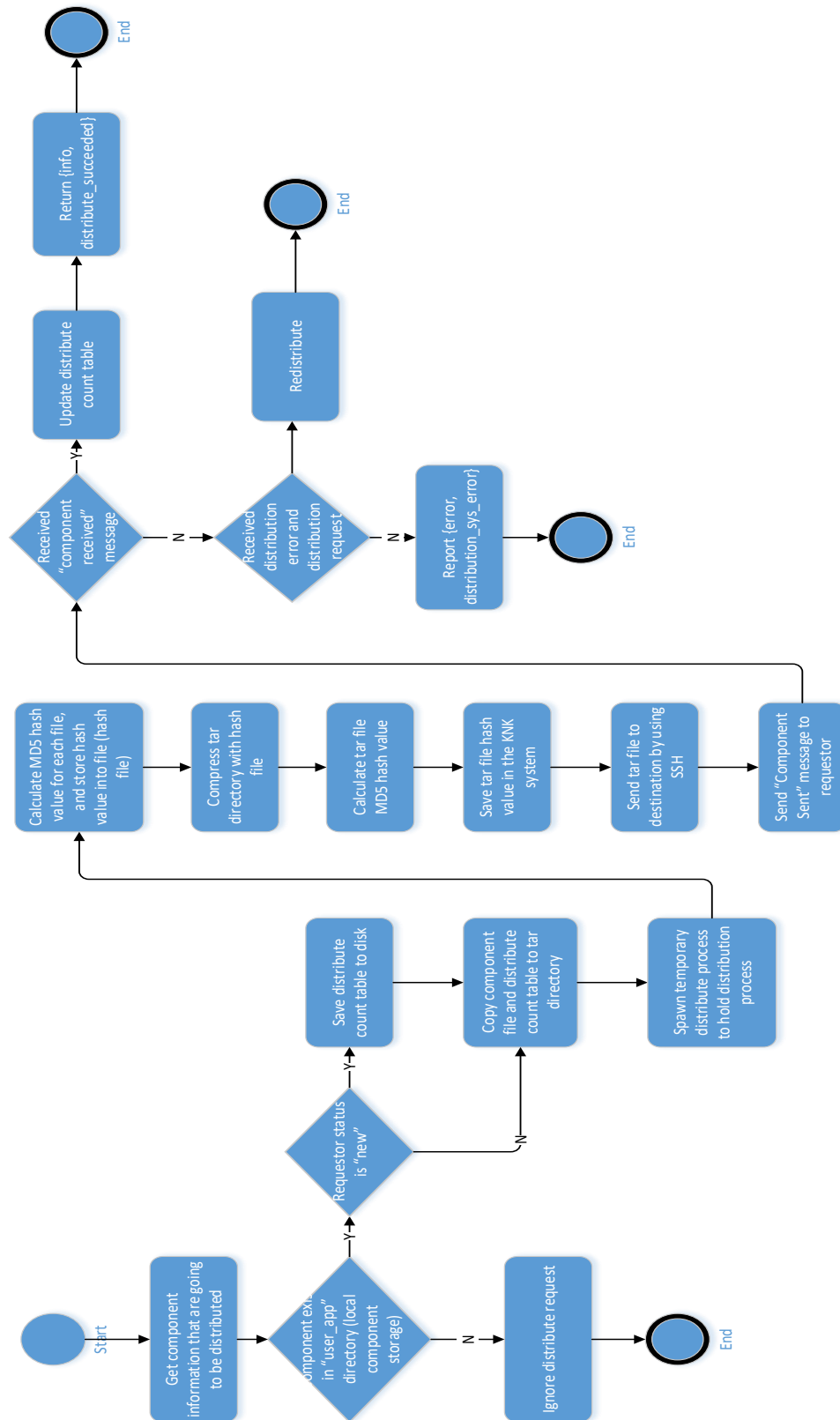


Figure 4.14: Distribution Sender Workflow

Once Ki-Ngā-Kōpuku receives a distribution request from the other Ki-Ngā-Kōpuku system, it will first check the distribution status of each application component from the distribution count table. As described in the distribution count table Section, components with lower distribution count numbers will be selected by Ki-Ngā-Kōpuku.

1. Component Selection.

The Ki-Ngā-Kōpuku system first retrieves all components' distribution status recorded in the distribution count table:

```
full_distribute_list() ->
  case ets:first(comp_dist) of
    '$end_of_table' -> empty_dist_count;
  HComp ->
    full_distribute_list(HComp, [])
  end.

full_distribute_list(HComp, RList) ->
  HCompCountL = ets:lookup(comp_dist, HComp),
  case ets:next(comp_dist, HComp) of
    '$end_of_table' -> RList ++ HCompCountL;
  NComp ->
    full_distribute_list(NComp, RList ++
      HCompCountL)
  end.
```

The above code is used to check the ETS table `comp_dist` and replies to a list as a result. The list contains the component name and its related count number, so that the list structure will be like this `[{CompName, CountNumber}]`.

After getting a full list of application components, Ki-Ngā-Kōpuku has to sort the list to distribute candidate application components. The following code shows how to sort the list in ascending order:

```
sort_distribute_list(RANGE) ->
  case full_distribute_list() of
    DistL when is_list(DistL) ->
      [X || {X, _} <- lists:sublist(
        lists:keysort(2, DistL), RANGE)];
```

```
        empty_dist_count -> no_count_list
    end.
```

The RANGE parameter passed in the above function is defined by users, which is then used by Ki-Ngā-Kōpuku to select a range of application components for distribution.

2. Component Preparation

When Ki-Ngā-Kōpuku has decided which application components should be distributed to the destination, the system will prepare the component files for component transfer. First, Ki-Ngā-Kōpuku has to make sure those selected components are running in its local environment. If those components can't be found in its local storage node, then Ki-Ngā-Kōpuku will ignore the distribution request because it doesn't want those application components. Instead, if Ki-Ngā-Kōpuku finds component information in the local storage node, then it will check whether or not those component source files exist.

In the current situation, Ki-Ngā-Kōpuku uses the shell script to prepare component files. During the preparation process, the hash value of each application component file and its distribution count table will be calculated. After that, Ki-Ngā-Kōpuku will compress those files into one tar file. The following command calculates the hash value for each component file and creates one compressed file:

```
# Calculate hash value for each component file
filename=$(echo $filePath | awk -F"/" '{print $NF}')
```

```
md5sum $filename > $metadata
datavalue=$(cat $metadata | cut -d " " -f 1)
dataname=$(cat $metadata | cut -d " " -f 3)
echo $datavalue,$dataname >> $hashFile
# Calculate hash value for distribution count table
md5sum $distname > $metadata
echo $tabhash,$tabname >> $hashFile
# Create tar file
```

```
tar -zcf $tarName *
```

3. Component Transfer

For transferring components to the destination, the password-less SSH is used.

The following shell script command uses SSH and RSYNC to transfer the tar file to the destination:

```
rsync -auth -e ssh $tarName $user@$DstHost:$storePath
```

\$tarName, \$user, \$DstHost, and \$storePath are parameters that should be passed to this command. \$tarName is the name of the tar file that will be transferred by the distribution sender. \$user is the system user name of distribution requestor. \$DstHost is the IP address of the destination server. The last parameter \$storePath is the directory where the tar file should be send to.

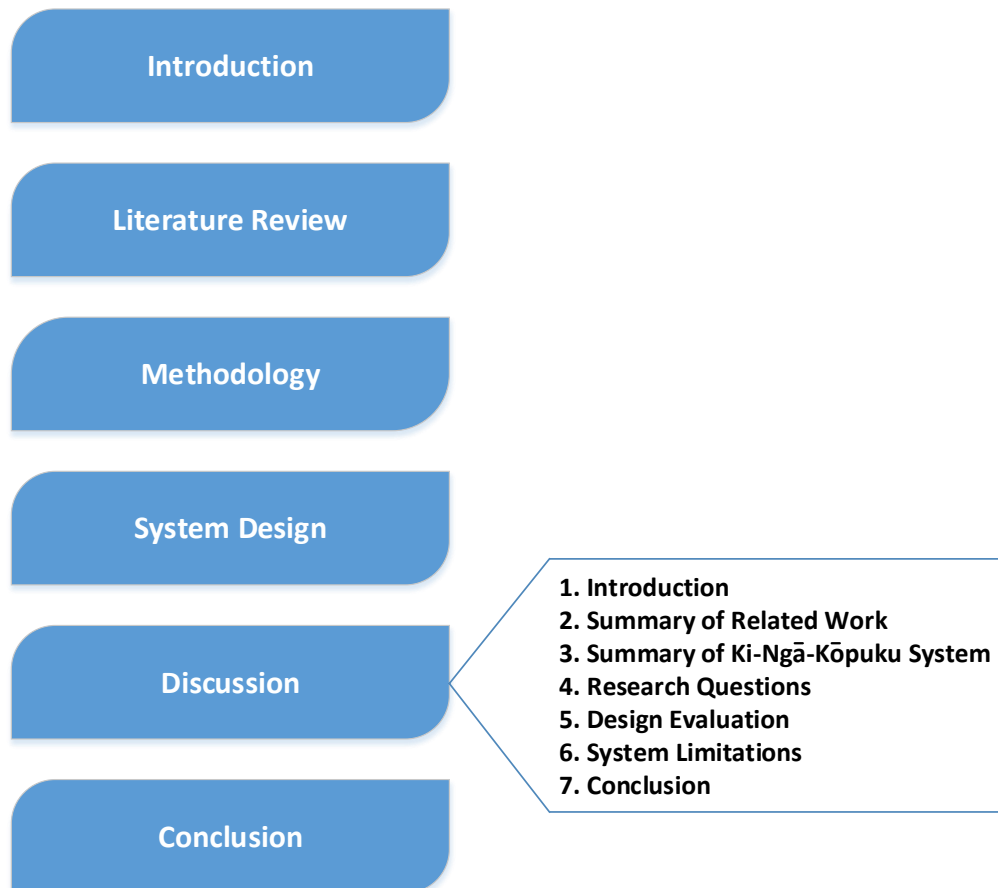
4.8 Conclusion

This chapter discussed details of system design of Ki-Ngā-Kōpuku. The Ki-Ngā-Kōpuku system is used as a distributed security system for user's applications. It is efficient to provide absolute redundancy and availability to user's applications by randomly distribute application components to different locations, so that user's application will always have active application components running and serving for users. For taking operations on user's applications, Ki-Ngā-Kōpuku requires developers to componentize the applications first to generate application components. Then, developers have to use the built-in function of Ki-Ngā-Kōpuku to register component information, so that Ki-Ngā-Kōpuku is able to taking actions on application components. Communications between Ki-Ngā-Kōpuku and application's components are achieved by using remote procedure call (RPC) or service port. Application's components can use RPC to retrieve necessary data from Ki-Ngā-Kōpuku system nodes or other application's

components. The specific UDP port is used to establish connections between different servers or Ki-Ngā-Kōpuku system nodes. In that case, different Ki-Ngā-Kōpuku system nodes can exchange necessary information between each other, which brings much more flexibility to the whole Ki-Ngā-Kōpuku cluster. The Ki-Ngā-Kōpuku system uses SSH and RSYNC to securely transfer component compiled files to achieve component distribution. In this case, as long as Ki-Ngā-Kōpuku can get the path of component's compiled source file, application components will always be suitable to be distributed.

The next chapter 5 will discuss the result of literature review, and proposed research questions.

CHAPTER : DISCUSSION



Chapter 5

Discussion

5.1 Introduction

The System Design chapter 4 has introduced the specific system design information of Ki-Ngā-Kōpuku, which includes system architecture, system features, traffic handler, and component distribution. For better evaluating the result of the previous chapter, this chapter will discuss the literature review, and proposed research questions. Then, more detailed information will be discussed according to comparing the researched theoretical articles and technical solution that has been described in chapter 4.

This chapter first discuss the literature review to summarize what can be learned from these articles. Then an overview of Ki-Ngā-Kōpuku system design is given to summarize what has been done in chapter 4. After that, the proposed research questions are discussed by evaluating the result of this research. At the end of this chapter, the limitations of Ki-Ngā-Kōpuku system are discussed to provide a basic direction for further research.

5.2 Summary of Related Work

This section discusses what has been done in the Literature Review chapter 2, and reflections on the literature that helps research and design Ki-Ngā-Kōpuku.

First of all, the concept of Ki-Ngā-Kōpuku is introduced: Ki-Ngā-Kōpuku is designed to secure the cloud environment, and bring absolute redundancy and availability to user's applications. It is a decentralized distributed security system, which is consisted of multiple independent self-controllers, and implement security mechanisms and distribute user's application components to different network locations.

5.2.1 Cloud Security Models

By studying the cloud security models, the basic design concept of Ki-Ngā-Kōpuku is set up. Servers are the work environment of Ki-Ngā-Kōpuku. In the aspect of cloud computing, Ki-Ngā-Kōpuku lies in the IaaS and PaaS layer. The Cloud Risk Accumulation Model of CSA (Brunette et al., 2009) describes the security risks will be accumulated when the service is located in lower layer. Cloud users have to put more efforts into the security if low layer services are used. As a result of that, Ki-Ngā-Kōpuku has to consider more security protections to protect the system itself. Moreover, as described in the Cloud Multiple-Tenancy Model of NIST (Brunette et al., 2009), the cloud providers allow multiple applications use the same computing resources, Ki-Ngā-Kōpuku should have the idea of isolation in the system design, which makes it hard to be affected by unrelated problems.

Jing and Jian-jun (2010) introduced one security concept called Security Access Control Service. It verifies user's identification in Access Authorization layer; user's access is secured by Security API; Cloud Connection Security ensures the safety of resources provided by the bottom resource layer. Based in this model, each layer in Ki-Ngā-Kōpuku should also implement security mechanisms, especially authenticating

identifications, and resources verification.

5.2.2 Data Security in Cloud Computing

Securing the data in cloud environment is one of the purpose of Ki-Ngā-Kōpuku. Ki-Ngā-Kōpuku can use encryption algorithms to protect data in the cloud environment. In Sedayao et al. (2009), third party's attributes are imported to encrypt and decrypt data, it also assumes the third party is secure and isolated from the cloud environment. However, this condition is hard to meet and the security of third party is not guaranteed, so Ki-Ngā-Kōpuku should use its own public or private key information to encrypt and decrypt data. Wang et al. (2010); Yang and Jia (2013) propose a third party auditor to protect data. It uses algorithms to generate signatures, proof data correctness, and audit the proof from the cloud server. This audit model provides a way for Ki-Ngā-Kōpuku to secure data from user access, data origin, and also the audit environment itself. Brodtkin (2008); So (2011); Singh and Chatterjee (2017) all mentioned imperfect design of data segregation is easy to cause security issues for data in storage in cloud computing. In this case, Ki-Ngā-Kōpuku should design advanced data segregation mechanism, and distribute data into different locations can also reduce the risk.

In Mouhib and Driss (2015), a framework is introduced that provides two ways for users to protect data. First way is that cloud providers use user's private key to decrypt data, and send decrypted data to users. This method is not acceptable, because user's private key is exposed. Second way is that cloud providers send encrypted data to users, and users use their private key to decrypt data. This method can be used in Ki-Ngā-Kōpuku, but selecting and extracting wanted data is a difficult task because data is encrypted. Khan et al. (2015) developed an encryption technique combines three encryption techniques. It splits the data in three parts, and different part uses different encryption techniques to encrypt. If Ki-Ngā-Kōpuku use different encryption algorithms

to encrypt different parts of the data, then it is going to consume lots of computing resources to encrypt and decrypt, the system also has to consider data processing. Unruh (2015) propose a time-release encryption method. This method allows the sender to get back the encrypted data before the time T , and the encrypted data can't be leaked before time T . Setting up a time limit for the encrypted data is a good idea. However, Ki-Ngā-Kōpuku is possible to handle massive size of data, which affects the time of data decryption. In this case, the time limit has to be set to a large value, otherwise the data won't have enough time to decrypt.

The Cloud Computing Multi-Dimension Date Security System is proposed in Xin et al. (2012). It has three layers: Authentication layer, File encryption and privacy protection layer, and File Fast Regeneration layer. The first layer authenticate user's identification. The second layer encrypts data. The third layer is used to recover files. This model has a buff area for the data recovery in case of the data is compromised. However, if the buff area is not secure, then it is highly possible to lose the data. But it is useful for Ki-Ngā-Kōpuku to set up a special area for accessing data. A similar model is also proposed in "Enhanced data security model for cloud computing." (2012). It uses OTP authentication in the first layer, and the last two layers are similar with the Cloud Computing Multi-Dimension Date Security System.

5.2.3 Distributed System

Mccaffrey (2016) mentions partial failure is one difficult part for building distributed system. Partial failure results in incomplete results. Jain and Paul (2013) discusses the state changes and policy changes are difficult in distributed system. Mishra and Tripathi (2014) mention issues and challenges of distributed software system, such as resource management, security and privacy, scalability, synchronization, and redundant testing during integration. In Afek (2013), message process and delivery are issues

in distributed system. Considering these problems, Ki-Ngā-Kōpuku should design a fault-tolerant mechanism. In this case, if one part of the system fails, the rest of system won't be affected. State and policy change in Ki-Ngā-Kōpuku should be restricted, and synchronising change operations within the system is useful. In order to process messages correctly, Ki-Ngā-Kōpuku should design message protocols or message brokers.

Fault-tolerance is one of the purpose of Ki-Ngā-Kōpuku. As stated in Sari and Akkaya (2015), replication is used for general fault tolerance method to protect against system failure. Carlini et al. (2013) propose an architecture for Distributed Virtual Environment. It combines Peer-to-Peer network with cloud computing. This architecture replicates servers in Peer-to-Peer network, and transfer the replicas to the cloud environment. Ki-Ngā-Kōpuku is software focused, so replicating servers or virtual machines are not useful for Ki-Ngā-Kōpuku. Moreover, this method assumes the cloud environment is absolutely secure, which doesn't fit the system. Borg et al. (1983) uses backup processes to achieve fault-tolerant in the aspect of processors. This method is useful to Ki-Ngā-Kōpuku. Backup processes can synchronise data from the primary processes, and take over the communication once the primary one are failed. However, Ki-Ngā-Kōpuku has to consider the synchronisation mechanism between primary and backup processes. Backup processes should also be secured. For achieving fault-tolerant, Ki-Ngā-Kōpuku can also evaluate user's actions or manage messages sequence. In Smara et al. (2016), a fault detection technique that uses acceptance test in the action verification process is introduced. It first check the result of actions, and evaluate the results to define whether the results are acceptable to the system. If the result is unacceptable, then the fail won't affect other parts. The Totem is a fault-tolerant ordered multicast group communication system. It manages the message sequence, so that messages can be processed properly.

Load balancing is a technique to improve performance and availability of distributed

systems. There are two types of load balancing techniques: static and dynamic. As described in Shah and Farik (2015), static load balancing algorithms lack resource monitoring and task management. Most of static algorithms don't check the execution time of tasks and the availability of resources. However, in Alakeel (2010), the advantage of dynamic load balancing algorithms is that tasks can move dynamically from an overloaded server to an unoverloaded server according to the current state of the system, but it is harder to implement dynamic load balancing algorithms. In the aspect of Ki-Ngā-Kōpuku, dynamic load balancing algorithms can be used, because the traffic destination can be changed via evaluating the state of resources.

5.2.4 Decentralized System

As described in Sandell et al. (1978), the decentralized system is known as distributed control. A distributed control system is a computerised control system for a process or plant, in which autonomous controllers are distributed throughout the system, but there is central operator supervisory control. Ki-Ngā-Kōpuku is supposed to be developed as a decentralized system, so autonomous controllers should be the basic component that builds up Ki-Ngā-Kōpuku. Each controller should also have fault-tolerant ability, so that the failure of a controller can't affect other active controllers.

Zhao and Xiao (2015) introduces the Autonomous Decentralized Test System. Double-ring structure is used in this model, Each test unit in the double-ring structure can work independently and complete assigned tasks. Once adding/removing test unit, then structure is going to breakup, and find place for new test unit. From this research, each Ki-Ngā-Kōpuku system node should work together in special scenarios, and also provides a layer that blocks underlying infrastructure information from users. In Coronado-Garcia et al. (2011), public-private key pair is implemented in the Data Field, which is used to send and receive message in the decentralized system. Message

sender uses receiver's public key and random number to encrypt data. The receiver will establish connection only if the sender's identification is authenticated. Since Ki-Ngā-Kōpuku is a decentralized system with multiple autonomous controllers, so the communication between each controller have to be encrypted. Using public-private key pair is useful in Ki-Ngā-Kōpuku.

5.2.5 Random Distribution

Software based component distribution research are not found, but literature that have similiar ideas have been discussed.

Carlini et al. (2013), the Distributed Virtual Environment architecture is discussed. It is able to split multiple virtual machines into different location. Even though this architecture doesn't relate too much with the distribution feature in Ki-Ngā-Kōpuku. From this research, Ki-Ngā-Kōpuku should distribute components, and manage component's information. The reason is that if multiple components are distributed into the server, then it is highly possible to make servers overloaded. Afzali Seresht and Azmi (2014) propose a distributed intrusion detection system called MAIS-IDS. This system has multiple security agents, and it is able to randomly select multiple security agents migrating to new environment. The MAIS-IDS is useful to Ki-Ngā-Kōpuku development. The idea of migrating security agents are similiar to Ki-Ngā-Kōpuku distribute user's application compnents.

5.2.6 Conclusion

The whole section 5.2 has discussed several theoretical and technical ways that are related to Ki-Ngā-Kōpuku. By analyzing the findings of the literatures, advantages and disadvantages of each literature are discussed. Moreover, the literature review findings are compared with the objectives of Ki-Ngā-Kōpuku. Findings unrelated to

Ki-Ngā-Kōpuku objectives will be ignored.

Based on the research on cloud security models, Ki-Ngā-Kōpuku should have advanced security mechanism implemented within it. In more detail, system isolation and data security are the main parts that should be considered in the aspect of security. From the literature, system isolation can be achieved by storing data in different locations or making the system become distributed. Placing access control in each system layer and securing the data can also improve the overall security level of Ki-Ngā-Kōpuku. Most of the literature have mentioned applying encryption technique are the direct way to secure data. In this case, Ki-Ngā-Kōpuku should use advanced encryption mechanism in the system. However, the concept of Ki-Ngā-Kōpuku is new by comparing to other systems, so the encryption mechanism should be designed to fit the architecture of Ki-Ngā-Kōpuku. As proposed in many literature about fault-tolerant system, hardware and system level fault-tolerant technique are always discussed. Due to Ki-Ngā-Kōpuku is only a system running on cloud environment, which has no direct access to physical resources, so Ki-Ngā-Kōpuku can use system level technique to achieve fault-tolerance, such as using backup process or evaluating user's actions. Load balancing technique can also be implemented into the system. The reason is that Ki-Ngā-Kōpuku is have to handle user's requests from outside world, static or dynamic load balancing technique can help reduce system burden.

Ki-Ngā-Kōpuku is also a decentralized system. From the literature, the decentralized system is known as distributed control, so Ki-Ngā-Kōpuku should use autonomous controllers to manage different parts of the system. Moreover, each controller should be possible to work with other controllers, so that the system scability can be guaranteed. Besides fault-tolerant ability of the system, literature also talk about communication security between each part of the decentralized system, which should also be concerned. By learning literature about component distribution, advanced distribution mechanism should be used in Ki-Ngā-Kōpuku, otherwise the server is possible to face overload

problem.

In general, as Ki-Ngā-Kōpuku is a decentralized distributed security system, which runs on the cloud environment, so cloud security, data encryption, distributed system, decentralized system, and component distribution have to be researched.

5.3 Summary of Ki-Ngā-Kōpuku System

As described in the System Design chapter 4, Ki-Ngā-Kōpuku is a decentralized distributed security system which brings security, absolute redundancy and availability to user's applications. This research aims to proof the concept of Ki-Ngā-Kōpuku in both theoretical and technical way, so this section discusses the major findings of this research.

5.3.1 Erlang Programming Language

First of all, the Erlang programming language is selected to develop Ki-Ngā-Kōpuku. The main reason to use Erlang is that it is useful to build up distributed system because of its node feature. Each node can be designed as an independent working unit, so that each node can work on different computation task, or all nodes work together to achieve the same goal. Moreover, Erlang's message passing technique makes the system be strong enough to do concurrent computing, which is ideal to serve massive requests. In the aspect of development, Erlang can develop the prototype in a really short time, and developers can get the benefit from Erlang's hot-swap feature, which makes developers easy to replace code without shutting down or restarting the service. In general, Erlang is an ideal programming language to develop Ki-Ngā-Kōpuku.

5.3.2 System Design

Ki-Ngā-Kōpuku is designed to be implemented between frontend interfaces and backend servers. Its function is to build up an infinite layer that lies on the top of user's applications, and provide stable services for users. In which case user's applications can always be available to users.

The building block of Ki-Ngā-Kōpuku is the Erlang node, which is the basic component that builds up Erlang's distributed system. Each Erlang node is an independent Erlang runtime system, which is able to hold multiple Erlang applications. For implementing Ki-Ngā-Kōpuku with user's applications, users have to compontenize their applications first. Each application component can be a service or some functions that are used within the application. As a result of that, Ki-Ngā-Kōpuku doesn't have to distribute the whole application in the distribution process.

The cluster and application group is introduced in Ki-Ngā-Kōpuku. Ki-Ngā-Kōpuku cluster is an organization of all Ki-Ngā-Kōpuku application groups in the whole network. An application group is the combination of Ki-Ngā-Kōpuku nodes and user's application components. For isolating unrelated traffic between different applications, application components that work for the same application are grouped in the same application group. Moreover, components that reside in different network locations are also belong to the application group. Therefore, a Ki-Ngā-Kōpuku cluster is consided of one or more application groups, components that are located in different network locations can also be in the same application group as long as they are working for the same application, one application can only have one application group in the whole network, and the application group will exist as long as there are components running in the network.

A Ki-Ngā-Kōpuku node is an independent self-management controller to user's application components. Due to the application group concept, the system node can

work with application components only if both serves the same application. Also, only one system node that works for an application is allowed to run in the network location, so that if a network location attempts to run multiple applications, then it has to run multiple system nodes, and each system node works for only one specific application. The reason to avoid multiple system nodes working for the same application in one network location is that the application's components management will be easier, and no collisions will happen between components.

While developing Ki-Ngā-Kōpuku, Erlang OTP design principles and Erlang supervision tree is used. Erlang OTP describes each system role in particular behaviours, which brings convenient while developing. The supervision tree is a design structure. It has two roles: supervisor and worker. A worker is a specific part of the system that do computation task. A supervisor is a monitor of workers. It is able to monitor the behaviour of each worker, and handle errors that happened within the worker, so that other parts of the system won't be affected, and only the problem worker is catered by supervisors. In general, the supervision tree gives Ki-Ngā-Kōpuku an absolute fault-tolerance ability.

5.3.3 System Communication

For using Ki-Ngā-Kōpuku with user's applications, the system has to provide communication methods for both user's applications and the system itself. Due to Ki-Ngā-Kōpuku is supposed to be an infinite layer for frontend interfaces and backend servers, so Ki-Ngā-Kōpuku node is the element to build up the infinite layer. Application interfaces or components can exchange data with the system node, then the system node is able to do corresponding tasks according to each operation. Moreover, each system node is the same, so no matter which system node that user's applications are connecting to, the system node will always provide appropriate services.

Erlang's distribution feature makes each node be able to communicate with other nodes among the network. However, this opens up a door for security threats. Once a node is compromised by hackers, then other the information of other nodes will be under danger. To prevent this security threat, Ki-Ngā-Kōpuku chooses to isolate each server among the network, and use system port for the system communication. Ki-Ngā-Kōpuku uses UDP port for transferring traffic to application's components. A Ki-Ngā-Kōpuku node uses the broadcast feature of UDP to send broadcast messages among the network, which makes other system nodes able to receive the message, so that each system node can still communicate with each other. Ki-Ngā-Kōpuku also uses its own message management service called traffic handler. The traffic handler can receive traffic from the system port, and analyze the traffic according to message patterns, then encapsulate the traffic in the pre-defined data format, after that the system node will send the new traffic to real destinations. A Ki-Ngā-Kōpuku node also uses UDP broadcast messages to find the wanted communication targets. The wanted communication targets can be a set of functions or services within the application. Each system node uses the same finding mechanism and follows the "local first" rule to manage the traffic processing order within Ki-Ngā-Kōpuku. "local first" rule refers to the local components have the highest priority to be selected to process the traffic if the local components are in the list of wanted components. Otherwise, the system node will try to find wanted components in other network locations via communicating with other system nodes.

5.3.4 Application Component

Ki-Ngā-Kōpuku uses its local storage to store the information of application's components. The local storage is another kind of Erlang node running on the system. Each system node has a local storage node. Four types of application components are defined

in Ki-Ngā-Kōpuku: new component for the whole system, new component for the local system, component that requires update, and duplicated component. Since an application component has to be identical to its corresponding system node, so the system node has to take actions on components by defining component types. However, storing component information within the local storage node is a potential security risk for the system. When the storage node is intruded by hackers, the local component information will be exposed, which is against the system requirement that the system information should be hard to determine.

5.3.5 Component Distribution

Ki-Ngā-Kōpuku distributes application components to different network locations to achieve the redundancy of user's applications. Once a Ki-Ngā-Kōpuku node receives distribute a request from other system nodes, it checks the component distribution status in the whole system first to identify which components should be distributed according to the distribution record stored in the local storage. Then the system node checks the existence of distribution candidate components in the local environment. If the system node detects the existence of application components that should be distributed, then it will prepare these components. Otherwise, the system node will ignore the distribute request.

For distributing application components to destinations, the system node requires users provide the directory of component's compiled source file while storing component information into the local storage. As a result of that, Ki-Ngā-Kōpuku node can use Linux commands to transfer component's compiled source file to the destination, and the distribute receiver can use the compiled source file directly to start the component. The distribute request is broadcast, so that each Ki-Ngā-Kōpuku node can react to the request. Once the components are selected to be distributed, the system node calculates

the hash value of each component file, which ensures component integrity. The receiver will randomly pick up application's components coming from a system node, and update the distribution status.

5.3.6 Distribution Status

Once a Ki-Ngā-Kōpuku node receives an application component, the types of components will be identified. The reason is that Ki-Ngā-Kōpuku records the distribution status of each application component in the whole network. The distribution record has two attributes: component name, and count number. If the component is new to the whole system, then a record of the new component will be added in all system nodes; if the component is new to the local system, then the count number will add one, and all other system nodes will update the count number together; if a new version of component is received, then the count number will be set back to one; if the component is duplicated, the distribution record will not be updated.

5.3.7 Conclusion

This research has researched Ki-Ngā-Kōpuku in the aspect of concept and technical part. Erlang programming language is used to develop Ki-Ngā-Kōpuku. Its special features make the system become distributed and decentralized. Using UDP within the system makes each system node plays the same role within the cluster. Before using Ki-Ngā-Kōpuku, users have to separate their applications into several components. A component can be a set of functions or a single service. In order to distribute application components, Ki-Ngā-Kōpuku uses Linux tools to create and transfer the compiled source file of components. When the system receives a new component, it will check the type of the new component first by checking its local storage and distribution status. The distribution status is stored in each system node that belongs to the same application

group. The status reflects the number of a component has been distributed in the whole network. Adding or removing components can change the distribution status in each system node. In general, Ki-Ngā-Kōpuku can bring redundancy and availability to user's applications.

5.4 Research Questions

The research questions that have been proposed in chapter 2 are as follows:

1. How to improve security among components?
2. How to design a component distribution mechanism to distribute components into different network locations?
3. How to maintain communication between components which are located on different network locations?

These three research questions are proposed during defining what exactly Ki-Ngā-Kōpuku is. Due to Ki-Ngā-Kōpuku is a decentralized distributed security system which runs on the cloud environment, so this research has to focus on cloud environment, and the functionality of the system itself.

5.4.1 Hypothesis

This hypothesis that have been proposed in chapter 3 are as follows:

1. Independent temporary public-private key pair authentication can improve security among components.
2. Distribute service components into multiple servers can improve the redundancy and performance level of overall system.
3. Using sockets and distribution feature of Erlang can maintain the communication between each server.

In the following section, the research questions and hypotheses are addressed.

5.4.2 Discussion

This section discusses the relevance between the literature review and the system design. Research questions and relative hypothesis will also be researched to ensure questions and hypothesis are helpful to the overall research.

The first view of Ki-Ngā-Kōpuku is having distributed application components among the network location. Ki-Ngā-Kōpuku encourages the developers to make decoupled and independent components or services of applications. In developers' perspective, this may lead to more hard working on software and architecture design. However, designing distributed decoupled components makes applications easy to scale and manage. As for the cloud providers, deploying this framework makes the platform itself be easy to recover from disasters. Moreover, efficient process can be taken when users want to scale their applications, such as adding more website instances, and setting up a new application server in a different country.

Component Security

According to Ki-Ngā-Kōpuku concept, security is the first part that need to be concerned. As can be seen in the previous literature research, most of cloud security models and methods are effective to secure the cloud environment. However, most of security mechanisms assume some parts of it are absolutely secure, such as cloud environment, and their own local computing environment. Since the cloud environment is not absolutely secure in the real life, and Ki-Ngā-Kōpuku consists of distributed components, so this research thesis propose the research question: *1. How to improve security among components?*

The related hypothesis is: *1. Independent temporary public-private key pair authentication can improve security among components.*

The public-private key pair can ensure identities at both ends, and the temporary feature makes it hard for hackers to track and copy information. It is a good idea to use the temporary public-private key pair for Erlang's distributed system and component authentication. Erlang's distributed system mainly relies on a trustworthy network environment, however, once this system is implemented on the cloud or any other open network, Ki-Ngā-Kōpuku will become weak. In this case, implementing a temporary public-private key pair can be a good way to improve data security and prevent malicious activities.

In fact, Ki-Ngā-Kōpuku provides a great environment for implementing temporary public-private key pairs. This is not because Ki-Ngā-Kōpuku environment is weak on security when it is working on an open network; rather, the real reason is the unique feature of nodes in Ki-Ngā-Kōpuku. Each Erlang node must have a unique node name to build up one distributed system so that it is identical for each node working in the same system. In this case, each node can use local information to create its temporary public and private key pair. For example, one node can use its node name and its current system time to create a random seed, and then use pseudorandom mechanisms to create random seed, and then use the random seed to create a key pair. The following code shows how Erlang uses unique node names, system times, and random numbers to create random seed:

```
random:seed(erlang:phash2([node()]),  
erlang:monotonic_time(),  
erlang:unique_integer()).
```

However, using the temporary public-private key pair in Ki-Ngā-Kōpuku is not

considered here because the objective of this research is to develop one prototype of Ki-Ngā-Kōpuku, and the larger project will need more developers and time for completion. On the other hand, Ki-Ngā-Kōpuku mainly focuses on providing redundancy and availability to applications, so the security issues for communication traffic are not considered the main problems for this research to solve.

Component Distribution

Ki-Ngā-Kōpuku is designed to distribute application's components into different network locations, so that multiple component replicas exist in the network to achieve redundancy for user's applications. As described in the Literature Review, not much literatures research on the service-focused components.

Since component distribution is the key point for Ki-Ngā-Kōpuku to provide applications redundancy, so the research question is: 2. *How to design a component distribution mechanism to distribute components into different network locations?* The relative hypothesis is: 2. *Randomly distributing application components can improve the redundancy and performance level of overall system.*

However, while developing the system, randomly distributing application components are not the best solution to the problem. The reason is that the random can't be achieved in computer science at all, which means the system has to do some specific computation to get the so called random result. While randomly choosing the component to distribute, there is a chance for some components not being selected by the system, which can cause unbalanced distribution status among all application components in network. In this case, Ki-Ngā-Kōpuku has to evaluate the distribution status of each component in the whole network.

In Ki-Ngā-Kōpuku system design, application components are distributed randomly. Ki-Ngā-Kōpuku first defines a range of components according to component distribution status, then it randomly pickings up components from the range to distribute. In the

distribution process, Ki-Ngā-Kōpuku should know which components have the least number of replicas in the whole network, and also what components that the local system is capable to distribute. In this case, two kinds of information are used: 1) the overall distribution status of each component in the system. 2) information about local application components. The overall distribution status will define which application component has lower number of replicas as compared to other components in the network, then Ki-Ngā-Kōpuku will know which component should be distributed, so that the distribution status of each component can be balanced. The local component information records the components that exist in the local system. It helps Ki-Ngā-Kōpuku define which component can be distributed according to its own situation.

In more detail, three questions have to be considered for the distribution of application components. These questions are: What to distribute? How to distribute? How to run application components?

What to distribute?

In this research, the component distributed by Ki-Ngā-Kōpuku is compiled code file. The reason is that most programming languages can directly run compiled code files in its own language environment. These two languages, for example, Java and Erlang, can transfer compiled code files to a different location and then run compiled files directly once Java and Erlang environment have been installed. Moreover, if the application is written in Erlang itself, then the components can be transferred by using the function `c:nl(Module)`. This function can load modules directly to other nodes or destinations, even though other nodes don't have the source code for that module. However, Ki-Ngā-Kōpuku is not only serving the Erlang programming language, so transferring compiled code files is a general way to distribute components.

How to distribute?

In the current research, SSH and RSYNC are used to transfer application components. Ki-Ngā-Kōpuku chooses to use file transmission for component distribution,

so the size of each application component must be considered. In that case, Ki-Ngā-Kōpuku will compress component files into one tar file, and transfer the tar file to destinations. When Ki-Ngā-Kōpuku is ready to transfer the tar file, it will use the password-less SSH for file transmission. However, password-less SSH still requires details of login users and a password so that each server can login to the other without providing user names and passwords, which is a serious security problem for the whole system. However, this research aims to achieve system functionality. Moreover, password-less SSH and Ki-Ngā-Kōpuku are suitable for working in a safe environment, so it is still reasonable to use SSH to transfer application components.

How to run application components?

Once Ki-Ngā-Kōpuku receives application components from other servers, the system should be able to run application components automatically. For doing this, application components have to provide start up script or command to Ki-Ngā-Kōpuku so that the system can use these mechanisms to start application components automatically. In this research, Ki-Ngā-Kōpuku uses function `os:cmd(Command)` to run the start up script for application components.

The component distribution mechanism used in Ki-Ngā-Kōpuku makes sure each component is balanced in the whole network, so that no components will be missed in the distribution process. Since Ki-Ngā-Kōpuku limits the amount of components that to be distributed, so the server won't be overloaded. In general, this mechanism guarantees the balance of overall component distribution status, but it still need more improvement because of the type of the components it distributes are limited.

Component Communication

Ki-Ngā-Kōpuku is designed as a layer on top of the application components. As a result of that, the system should be able to exchange information with application components. On the other hand, Ki-Ngā-Kōpuku also distribute components into different locations,

so that each component should be able to communicate with others.

The research question is: *3. How to maintain communication between components which are located on different network locations?* The relative hypothesis is: *3. Using sockets and distribution feature of Erlang can maintain the communication between each component.*

In Ki-Ngā-Kōpuku, the system node is responsible for communicating with outside world. It opens two ports for communication. One port is defined by users and is used to talk to the application's interface. Another port is used for Ki-Ngā-Kōpuku itself, which is used to send a message to other system nodes. In more detail, the traffic coming from application's interface won't go directly to application components. The traffic will be received by the system node, then the system node will parse the traffic to find out the destination, which is the application component. If the component doesn't exist in local system, then the system node will send messages to all other system nodes in the network to find the required component.

In fact, the main reason to use socket for component communication among different network locations is for securing the system. Since Ki-Ngā-Kōpuku is developed in Erlang, every element of the Erlang distributed system has to be trusted. However, if one part of the system is compromised, then it is highly possible to expose the information of other parts. So in the system design, Erlang's distribute features are only allowed in local environment, and socket is used to send and receive messages. There are two general scenarios for using the socket for node communication:

Scenario One: Requesting Application Components

Ki-Ngā-Kōpuku is an empty framework when it is first initialised. To serve user's applications, the system has to request application components from other Ki-Ngā-Kōpuku. Since cluster members are unknown to Ki-Ngā-Kōpuku, new Ki-Ngā-Kōpuku will use UDP broadcast features to broadcast requests among all networks, so that the network destination can be detected. In that case, all system nodes that are in the

same network can receive the request and prepare application components for new Ki-Ngā-Kōpuku.

Scenario Two: Finding Application's Components

Application components are supposed to work like a complete application under Ki-Ngā-Kōpuku. However, the application is separated into several pieces, and it is possible for each Ki-Ngā-Kōpuku node to hold different application components, so application components need a way to exchange messages with other application components. However, each Ki-Ngā-Kōpuku system node is not allowed to talk to each other directly for security reasons. This means a Ki-Ngā-Kōpuku node can broadcast UDP traffic to each system node and get the information that its components want. Moreover, if a Ki-Ngā-Kōpuku system node wants to build a point-to-point communication, it can use unicast UDP traffic for one specific destination.

Components in the same local environment can use Erlang features to perform component communication. In Erlang programming language, it is possible to use RPC functions to make synchronous or asynchronous calls. This is important for application components because the applications have to have different operations according to different requests or strategies. Besides using RPC to talk directly to the remote location, Erlang also has its message-passing technique for distributed systems. Erlang uses `Pid ! Msg` and `{Pid, NodeName} ! Msg` to send a message to one process. In the first example, `Pid` is the process id or identifier for where the message is going to be sent. This method is used to send messages to a locally registered process. If the message has to be sent to a process that runs on a remote node, then the second example can be used.

In general, using socket and Erlang distribute features can maintain component communication. However, Ki-Ngā-Kōpuku should consider providing efficient communication protocols for components because the system uses UDP broadcast messages to find the message destination. If the system receives complex traffic which requires

multiple components to corporate, then whole Ki-Ngā-Kōpuku will face broadcast flood.

5.4.3 Conclusion

Based on the Literature Review and System Design for Ki-Ngā-Kōpuku, three proposed research questions are analyzed. The first question is about component security. However, in current situation, this research haven't apply all security mechanisms in Ki-Ngā-Kōpuku. The second research question is about component distribution. In current research, randomly distributing application components by comparing the distribution status of each component can make sure each component is balanced. However, the current distribution mechanism doesn't have enough scalability, which are not convenient for some programming languages. As a result of that, the distribution mechanism needs further improvement. The last research question is about component communication. Ki-Ngā-Kōpuku uses socket to build up communication channel between the system and outside world. It also uses Erlang message passing techniques to maintain the component communication in local system. But it also need improvent in the future.

5.5 Design Evaluation

As mentioned in 3, the prototype developed during this research will be evaluated by a set of evaluation criteria. However, the expected prototype is not totally finished when this research hits the deadline. As a result of that, only parts of the prototype will be evaluated by the evaluation criteria.

5.5.1 Goal Evaluation

In `Goal` dimension, three criteria will be evaluated: efficacy, validity, and generality. Efficacy means the prototype achieves research goals as expected. The goal of this research is to prove a new concept by building a prototype with technique implementation. The prototype has to provide redundancy, availability, and fault-tolerant ability to user's application. As mentioned before, some parts of the prototype are able to provide redundancy and availability 4.7, but the fault-tolerant ability 4.2.3 lacks experimental test. Moreover, a research question about component security is not fully analyzed during this research. As a result, the efficacy of this research is not perfect.

The validity evaluation means the prototype should be able to handle each request correctly. This research has developed a system module to handle coming request and interact with backend services 4.5, and a system module to manage component distribution 4.7. However, it doesn't cover all scenarios about traffic communication. As a result, the validity of this research is not perfect.

The generality evaluation is to measure the value of the prototype to current theory and techniques. Based on 4, the concept and the architecture is totally new, so this research and the prototype are valuable to current theory and techniques.

5.5.2 Environment Evaluation

In `Environment` dimension, the prototype has to be evaluated by two criteria: consistency with people, and consistency with technology. In the aspect of consistency with people, the prototype has to be easy to use and understand. This research tries to provide a command line interface of the prototype 4.4, so that users can manage the whole system by using the command line interface. The prototype also use Ki-Ngā-Kōpuku system cluster concept 4.3.1 and application group concept 4.3.2 to explain how the whole system works. As a result of that, the prototype is consistent to people.

The prototype is also consistent to technology. As described before, the concept is new to current theories and technologies. And there won't be any side effects.

5.5.3 Structure Evaluation

In `Structure` dimension, two criteria will be evaluated: completeness, and homomorphism. The completeness is to evaluate whether the prototype covers all related modules. In fact, the prototype is composed of socket module, distribution module, traffic handler, log module, and storage module. However, only socket module, distribution module and traffic handler are developed. As a result, the prototype is not fully completed.

The homomorphism means each module is capable of working with other modules without errors. Ki-Ngā-Kōpuku works like a framework for user's application, so each module of Ki-Ngā-Kōpuku itself shouldn't be separated. By following the design in 4.2.3, the prototype itself is able to consume errors occurred within the system.

5.5.4 Activity Evaluation

Completeness and performance are the criteria in `Activity` dimension. The completeness in this dimension focuses on module's functionality. In this case, the completeness is not perfect, because some of the modules are not fully developed. In the aspect of performance, the evaluation result is also not good. The reason is that the prototype is not fully developed, and it requires additional applications to join in the evaluation.

5.5.5 Evolution Evaluation

Two evaluation criteria in `Evolution`: robustness, and learning capability. The robustness evaluates the ability to take actions to the change of environment. In current design, Linux is the only environment for Ki-Ngā-Kōpuku. Another environment that Ki-Ngā-Kōpuku relies on is port. However, the prototype is able to change system

configuraiton according to user's need, and it has the ability to handle errors. As a result, it is robust.

Learning capability evaluates the prototype's ability to learn from previous experience. In current design, the learning capability is bad.

5.5.6 Evaluation Conclusion

Based on the above evaluation, the current prototype should do the following:

1. Fully develop necessary system modules.
2. Test prototype with users' applications.
3. Define metrics for performance evaluation.

5.6 System Limitations

It is clear from discussing research questions and research results that Ki-Ngā-Kōpuku still has limitations and problems. This section will discuss the limitations that might affect Ki-Ngā-Kōpuku.

5.6.1 System Environment

Ki-Ngā-Kōpuku has to be run in the trusted network. The reason is that the Erlang programming language is used to develop Ki-Ngā-Kōpuku. In Erlang distributed system, node is the basic part in the system. However, the security mechanism implemented in nodes are weak, so that Erlang distributed systems have to stay in the trusted environment. On the other hand, Ki-Ngā-Kōpuku sends UDP broadcast messages to communicate with other Ki-Ngā-Kōpuku. Since broadcast is only useful in the local area network (LAN), so that Ki-Ngā-Kōpuku has to run in the closed environment.

5.6.2 Server Network Interface Single-Point-Of-Failure (SPF)

Ki-Ngā-Kōpuku is designed to provide redundancy and availability for the user's application. However, Ki-Ngā-Kōpuku is a framework that focuses on guaranteeing an application's functionality. The reason for putting more attention on the server side instead of the network interface is that there are tools and technologies to help solve the problem.

The ideal usage scenario for Ki-Ngā-Kōpuku is to put it between client interfaces and actual servers. The client interface can connect to the server that runs Ki-Ngā-Kōpuku nodes; then Ki-Ngā-Kōpuku becomes one kind of gateway for passing messages for not only client interfaces but also backend servers. On the other hand, if a Ki-Ngā-Kōpuku node that the client interface connects to is failed, then it won't affect the whole system because other Ki-Ngā-Kōpuku nodes can do the same work and be connected by client interfaces to serve users continuously. However, the network interface of servers can affect the whole Client-Server structure because it is on the top level of backend servers.

In general, Client-Server architecture always faces a problem with the connection between frontend and backend. If the frontend needs to retrieve data from the backend, it has to know where the backend server is located. So, in normal situations, domain names or static public IP address are used to represent the location of servers. Moreover, the TCP and UDP connection also need to know the specific destination address. Consequently, if the client interface can't connect to the backend servers' address, then the whole Client-Server structure will go down. This represents a network problem that sometimes needs help from other tools, software, or even hardware.

There are ways to help reduce the damage of this problem. Firstly, putting more public static IP addresses under one domain name, or using multiple domain names for Client-Server connections. This is always implemented by the ISP, which guarantees that if one destination is unreachable, the ISP will always direct messages to those

still alive at the destination. However, if the frontend uses IP address instead of using domain names as the destination, it will need to change the destination. Secondly, some keep-alive tools can be used on the server side. Keep-alive tools mainly use the floating virtual IP address technology. This transfers the same IP address to another connectable destination if the original one goes down so that the top level won't notice the destination change. Despite all these possible solutions, it is still not possible to completely get rid of this problem with Ki-Ngā-Kōpuku because it works behind the network interface. If the network connection is unstable or stopped, then Ki-Ngā-Kōpuku won't work as well as planned. Consequently, Ki-Ngā-Kōpuku works for backend servers and assumes that the network interface is healthy.

5.6.3 Distribution Times

Distributing application components is a special feature of Ki-Ngā-Kōpuku. Each Ki-Ngā-Kōpuku can request application components from other Ki-Ngā-Kōpuku. However, it is also highly possible for one Ki-Ngā-Kōpuku to obtain all application components if the system sends multiple distribution requests. In that case, if hackers compromise one Ki-Ngā-Kōpuku then they can obtain all of the application components and endanger the application.

Setting up distribution times for Ki-Ngā-Kōpuku might help resolve this problem. However, if new application components are joined, this might affect the current working system. The reason is that applications are always changing, so application components have to change as well. No matter whether components are new to the system or need to be updated, Ki-Ngā-Kōpuku still has to consider the redundancy and availability of these components. If distribution times are limited, then Ki-Ngā-Kōpuku might not be well prepared for guaranteeing components' redundancy and availability.

5.6.4 Malicious Components

Application components are supposed to work under Ki-Ngā-Kōpuku. However, it is hard to control the components. Currently, Ki-Ngā-Kōpuku can receive any users' application components. This means that if one malicious component joined Ki-Ngā-Kōpuku, it is not possible to detect it.

To reduce the risk coming from malicious components, Ki-Ngā-Kōpuku should have an authentication process for users' applications. This can be done by generating a unique fingerprint for the application once the system is first initialised. If one application component wants to join, it must provide the application fingerprint to prove that it is a part of the application.

5.6.5 Communication Efficiency

In the current research, Ki-Ngā-Kōpuku uses the socket and Erlang message passing techniques to establish node communication. It is designed as a framework that can be used for any applications. Consequently, Ki-Ngā-Kōpuku does not contain any application business logic, which means the system doesn't know the logic to process each request. In the case of a banking system application example, when a user wants to login the banking system must first verify the user's existence then make sure the user's identification is correct. However, Ki-Ngā-Kōpuku only provides APIs to search the required functions for the application interface and components. Thus, it is not so efficient for application communication because the system will spend much time searching the required functions and components when no application logic controllers are set in Ki-Ngā-Kōpuku.

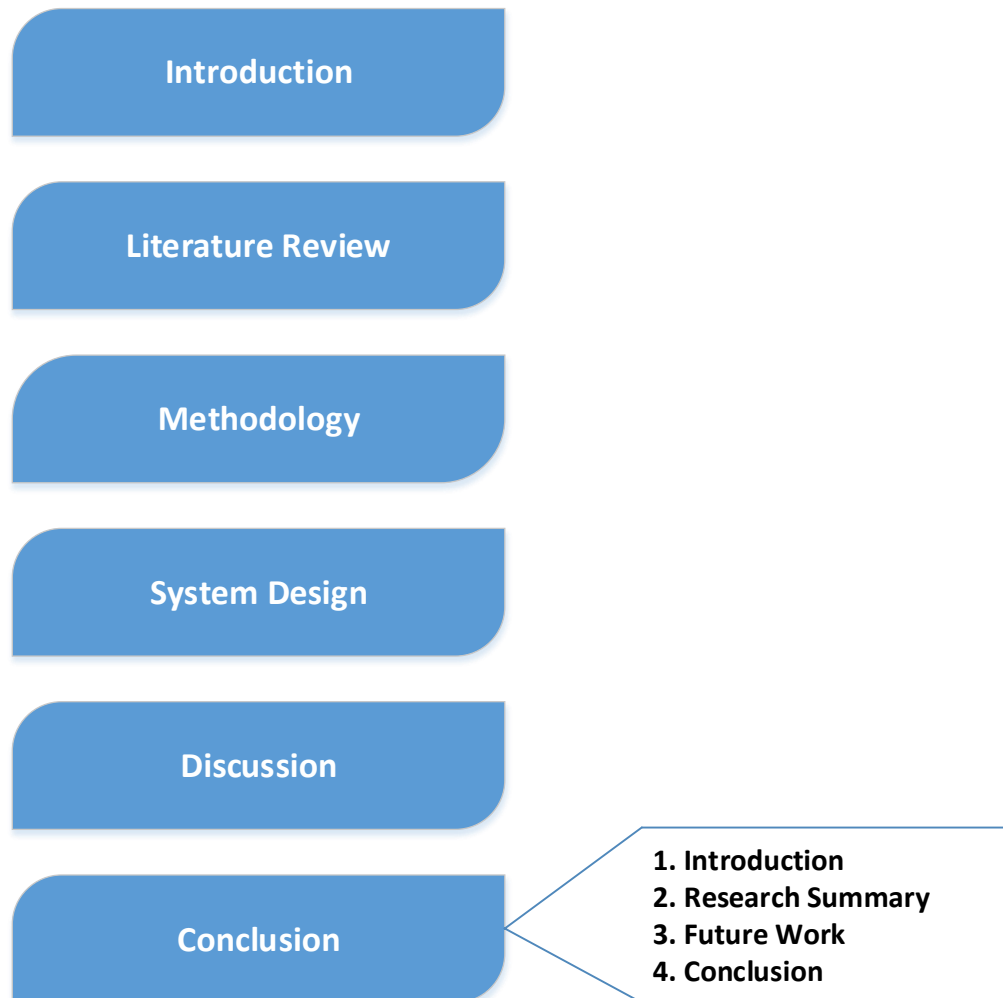
In general, designing a more efficient way to establish the communication bridge between each application component is an important task for Ki-Ngā-Kōpuku.

5.7 Conclusion

This chapter has discussed the Literature Review and what should be kept in mind based on existing literatures. The system design of Ki-Ngā-Kōpuku, is also been discussed. Moreover, this chapter evaluates the current prototype by following evaluation criteria, and also trying to find possible solutions for the proposed research questions based on the literatures and current Ki-Ngā-Kōpuku. System limitations are also discussed.

The next chapter 6 is Conclusion. It gives a conclusion of what this research thesis is about, what has been done in this research, and the further research.

CHAPTER : CONCLUSION



Chapter 6

Conclusion

6.1 Introduction

This chapter makes a conclusion about this whole research thesis. The section 6.2 makes a conclusion about what is this thesis, what is this research supposed to do, what has been done in the thesis, and whether the research has achieved the ideal goals.

6.2 Research Summary

The cloud environment is a great platform for implementing the distributed system, but the cloud can't guarantee applications can work in a safe environment. The aim of this research thesis is proof of concept of a new system called Ki-Ngā-Kōpuku, and develop a prototype of the system. This system is a decentralized distributed security system, which provides redundancy, availability, and fault-tolerant to user's applications. In order to proof the new concept, this thesis researched related literatures, such as cloud security, distributed systems, and decentralized systems.

To develop the prototype, and step closer towards to the existing system concept, several research questions were proposed:

1. How to improve security among components?
2. How to design a component distribution mechanism to distribute components into different network locations?
3. How to maintain communication between components which are located on different network locations?

This research follows the above research questions, then using design science research methodology for the overall research, and rapid application development methodology for technical development development. For better understand and solve research questions, relative hypothesis are proposed:

- Independent temporary public-private key pair authentication can improve security among components.
- Distribute service components into multiple servers can improve the redundancy and performance level of overall system.
- Using sockets and distribution feature of Erlang can maintain the communication between each server.

Ki-Ngā-Kōpuku is developed in the Erlang programming language, which is designed for building distributed systems. Hence, Erlang is an ideal language to design such a system running in the cloud environment. According to Ki-Ngā-Kōpuku concept, the system can provide redundancy and availability to applications by taking actions on application components. Application components are important for Ki-Ngā-Kōpuku. In the current research, components can be one single file running one single service, or one single file running multiple services for one application. During the research period, research questions about component distribution and component communication were partly solved by using Erlang and other Linux software. The critical part of component communication has been proven, but the solution still needs further improvement. For

the component distribution research question, Ki-Ngā-Kōpuku evaluates the overall distribution status of application components in the network, then compare with the local components in itself to randomly decide which components can be distributed. On the other hand, the form of application components may not work for some programming languages because Ki-Ngā-Kōpuku transfers compiled code file, which is not secure and flexible. The current distribution problem of this solution is that Ki-Ngā-Kōpuku has to read specific record to get the overall component distribution status, which makes Ki-Ngā-Kōpuku less secure. For solving the component communication problem, Ki-Ngā-Kōpuku uses UDP socket and Erlang message passing features to transfer data. UDP is normally used to send broadcast messages in the whole network. However, massive broadcast messages will be a great burden for the whole network. The last research question is about securing application components. In Ki-Ngā-Kōpuku design, application's name, system node's name are used to authenticate users, applications, and network traffic. But the temporary public-private key mechanism is not used in current research.

Based on the current research, Ki-Ngā-Kōpuku can perform critical functions on users' applications. Even though some research questions couldn't be completely answered, some advice was given for improvement, and possible system limitations were listed for further research.

6.3 Future Work

Application Authentication

Each component is considered to belong to the same application as Ki-Ngā-Kōpuku. Even though Ki-Ngā-Kōpuku has authentication methods for each application component, the method is not strong enough to prove the relationship between components

and applications. In the future, it will be possible to develop one authentication mechanism to prove that each component belongs to one specific application. Any unrelated components should not be used in Ki-Ngā-Kōpuku.

Security Components

As Ki-Ngā-Kōpuku works in the cloud environment, the system can implement security solutions for itself or users' applications. The Ki-Ngā-Kōpuku system is consisted of several Erlang nodes, so the system can have specific components that run firewalls, IDS, IPS, or other security solutions. The Ki-Ngā-Kōpuku system can distribute these security components along with user's application components so that other systems or servers will also be protected.

Application Components Distribution

The compiled source code file is now the application component that is used by Ki-Ngā-Kōpuku for distribution. Transferring the compiled file is a general way to transfer one application component to another location. However it is neither safe nor convenient. On the other hand, most code files require dependencies to run on the system, so Ki-Ngā-Kōpuku should also have the ability to deal with components and their dependencies while implementing component distribution.

In current design, Ki-Ngā-Kōpuku will only distribute components once it receives distribution request from other systems. Since the distribution request is sent manually by typing command, so the overall distribution process is not good enough. The ideal distribution process for Ki-Ngā-Kōpuku is random and automatic. So the system should be able to automatically distribute components once the requestor is detected.

6.4 Conclusion

This research is proof of concept of Ki-Ngā-Kōpuku. Ki-Ngā-Kōpuku is a brand new architecture in the cloud computing, which is designed to make user's applications

become highly available and stable. In this research thesis, some critical parts of Ki-Ngā-Kōpuku have been proved not only in theoretical way, but also in technical way, such as the distribution mechanism of application components, and traffic communication between each application component. Even though the system security is not achieved, but the prototype is able to perform simple authentication to network traffic and components.

Ki-Ngā-Kōpuku is a large project, it includes many interesting ideas and cool technologies. What I've done in this research is only a small part of the system. I hope Ki-Ngā-Kōpuku become strong, and powerful. And one day, Ki-Ngā-Kōpuku can change the world.

Appendix A

Glossary

Application User-defined applications that work with Ki-Ngā-Kōpuku. Applications can be a bank system, an ecosystem, a calculator application, etc.

Application Component Modules separated from user-defined applications. Each component should have less dependencies and be able to perform task independently.

Application Group The collection of application components that belong to one specific application.

Application Port The port that Ki-Ngā-Kōpuku system uses to communicate with user-defined applications.

Availability Availability is the probability that a system will work as required when required during the period of a mission.

Broadcasting It is a method of transferring a message to all recipients simultaneously.

Erlang It is a general-purpose, concurrent, functional programming language, as well as a garbage-collected runtime system.

Erlang OTP It is a collection of useful middleware, libraries, and tools written in Erlang programming language.

Erlang OTP gen_server It is a behavior module provides the server of a client-server

relation.

Erlang OTP supervisor It is a behavior module provides a supervisor, a process that supervises other processes called child processes. A child process can either be another supervisor or a worker process.

Fault-tolerant Fault tolerance is the property that enables a system to continue operating properly in the event of the failure of (or one or more faults within) some of its components.

Ki-Ngā-Kōpuku System The system that is designed to bring redundancy, availability, and fault-tolerant to user-defined applicaitons.

Ki-Ngā-Kōpuku System Cluster A group of system nodes that run in the same network environment.

Worker Process It is the processes which perform computations, and responsible for performing specific user or system actions.

Redundancy Redundancy is a system design in which a component is duplicated so if it fails there will be a backup.

Runtime System A runtime system is a collection of software and hardware resources that enable a software program to be executed ona computer system.

Node An independent runtime system which is written in Erlang.

Server Devices that host specfic OS. Such as physical servers, virtual machines.

System Node Nodes that work for Ki-Ngā-Kōpuku system and user-defined applications. Manage application components that belong to the same application.

System Port The port that each system node uses to communicate with other nodes.

TCP Transmission Control Protocol. It is one of the main protocols of the Internet protocol suite.

Traffic Handler A process that runs within Ki-Ngā-Kōpuku system. It is responsible for receiving incoming traffic and deliver the result to the right destinations.

UDP User Datagram Protocol. It is one of the core members of the Internet protocol

suite.

References

- Abhishek, H., & Yadav, J. (2013). Data encryption techniques commonly used algorithms and their security issues. *International Journal of Research in Information Technology*, 1, 186–193.
- Afek, Y. (2013). *Distributed computing : 27th international symposium, disc 2013, jerusalem, israel, october 14-18, 2013. proceedings*. Heidelberg : Springer, 2013.
- Afzali Seresht, N., & Azmi, R. (2014). Mais-ids: A distributed intrusion detection system using multi-agent ais approach. *Engineering Applications of Artificial Intelligence*, 35(1), 286–298. Retrieved from <http://www.sciencedirect.com/science/article/pii/S0952197614001444>
- Agrawal, M., & Mishra, P. (2012). A comparative survey on symmetric key encryption techniques. *International Journal on Computer Science and Engineering*, 4(5), 877. Retrieved from <https://pdfs.semanticscholar.org/20f3/dd8943a17138c3eefa4258aa1b6837ffcb59.pdf>
- Ahmad, N., & Habib, M. K. (2010). Analysis of network security threats and vulnerabilities by development & implementation of a security network monitoring solution. Retrieved from <http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A832701&dswid=-4344#sthash.1dvoeMva.dpbs>
- Ahmed, W., & Wu, Y. W. (2013). A survey on reliability in distributed systems. *Journal of Computer and System Sciences*, 79, 1243–1255. Retrieved from <http://www.sciencedirect.com/science/article/pii/S0022000013000652>
- Alakeel, A. M. (2010). A guide to dynamic load balancing in distributed computer systems. *International Journal of Computer Science and Information Security*, 10(6), 153–160.
- Babu, K. R., & Samuel, P. (2016). Enhanced bee colony algorithm for efficient load balancing and scheduling in cloud. In *Innovations in bio-inspired computing and applications* (pp. 67–78). Springer.
- Bagchi, S. (2015). *Emerging research in cloud distributed computing systems*. Hershey, Pennsylvania (701 E. Chocolate Avenue, Hershey, PA 17033, USA) : IGI Global, [2015]. Retrieved from <https://books.google.co.nz/books?hl=en&lr=&id=U4EfCgAAQBAJ&oi=fnd&pg=PR1&dq=Emerging+research+in+cloud+distributed+computing+systems&ots=sInpLd1N9o&sig=vLcPZ3o90fope9YS3P0rK-tdRww#v=onepage&q=Emerging%20research%20in%20cloud%20distributed%20computing+systems>

- 20distributed%20computing%20systems&f=false
- Bansal, V. P., & Singh, S. (2015, Dec). A hybrid data encryption technique using rsa and blowfish for cloud computing on fpgas. In *2015 2nd international conference on recent advances in engineering computational sciences (raecs)* (p. 1-5). doi: 10.1109/RAECS.2015.7453367
- Beynon-Davies, P., Carne, C., Mackay, H., & Tudhope, D. (1999). Rapid application development (rad): an empirical review. *EUROPEAN JOURNAL OF INFORMATION SYSTEMS*, 8(3), 211 - 223. Retrieved from <http://link.springer.com/article/10.1057/palgrave.ejis.3000325>
- Borg, A., Baumbach, J., & Glazer, S. (1983). Message system supporting fault tolerance. In *Operating systems review (acm)* (17th ed., Vol. 17, p. 90-99). Auragen Systems Corp, Fort Lee, NJ, USA. Retrieved from <http://www.andrew.cmu.edu/course/15-749/READINGS/optional/borg-1983.pdf>
- Brodkin, J. (2008). Gartner: Seven cloud-computing security risks. *Infoworld*, 2008, 1-3.
- Brunette, G., Mogull, R., et al. (2009). Security guidance for critical areas of focus in cloud computing v2. 1. *Cloud Security Alliance*, 1-76.
- Building an application with otp*. (2017). <http://learnyoussomeerlang.com/building-applications-with-otp>. (Accessed: 2017-10-28)
- Carlini, E., Ricci, L., & Coppola, M. (2013). Flexible load distribution for hybrid distributed virtual environments. *Future Generation Computer Systems*, 29(Including Special sections: High Performance Computing in the Cloud & Resource Discovery Mechanisms for P2P Systems), 1561 - 1572. Retrieved from <http://www.sciencedirect.com/science/article/pii/S0167739X1200177X>
- Chang, V., Walters, R. J., & Wills, G. (2015). *Delivery and adoption of cloud computing services in contemporary organizations*. Hershey : Information Science Reference, [2015]. Retrieved from <https://books.google.co.nz/books?hl=en&lr=&id=AWfCCAAQBAJ&oi=fnd&pg=PP1&dq=Delivery+and+Adoption+of+Cloud+Computing+Services+in+Contemporary+Organizations&ots=n4Wwc4nBsT&sig=HeepZAoAzY7IvcwjEx5S2qyiie0>
- Che, J., Duan, Y., Zhang, T., & Fan, J. (2011). Study on the security models and strategies of cloud computing. *Procedia Engineering*, 23(PEEA 2011), 586 - 593. Retrieved from <http://www.sciencedirect.com/science/article/pii/S187770581105394X>
- Chen, D., & Zhao, H. (2012). Data security and privacy protection issues in cloud computing. In *Proceedings - 2012 international conference on computer science and electronics engineering, iccsee 2012* (1st ed., Vol. 1, p. 647-651). College of Information Science and Engineering, Northeastern University. Retrieved from <http://ieeexplore.ieee.org/abstract/document/6187862/>
- Cisco 2017 annual cybersecurity report*. (2017). Cisco. Retrieved from <http://b2me.cisco.com/en-us-annual-cybersecurity-report-2017>

- Concurrent programming*. (2017). http://erlang.org/doc/getting_started/conc_prog.html. (Accessed: 2017-10-28)
- Coronado-Garcia, L. C., Gonzalez-Fuentes, J. A., Hernandez-Torres, P. J., & Perez-Leguizamo, C. (2011, March). An autonomous decentralized system architecture using a software-based secure data field. In *2011 tenth international symposium on autonomous decentralized systems* (p. 331-334). doi: 10.1109/ISADS.2011.50
- Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2), 56–78.
- Dan, H., Michael, S., et al. (2010). Top threats to cloud computing v1. 0. *Cloud Security Alliance*, 2, 2012.
- Desai, T., & Prajapati, J. (2013). A survey of various load balancing techniques and challenges in cloud computing. *International Journal of Scientific & Technology Research*, 2(11), 158–161.
- Distributed erlang*. (2017). http://erlang.org/doc/reference_manual/distributed.html. (Accessed: 2017-10-28)
- Enhanced data security model for cloud computing. (2012). *2012 8th International Conference on Informatics and Systems (INFOS), Informatics and Systems (INFOS), 2012 8th International Conference on*. Retrieved from <http://ieeexplore.ieee.org/document/6236556/>
- Escalante, D., & Korty, A. J. (2011). Cloud services: policy and assessment. *Educause Review*, 46(4).
- Feiker, G. E. (1979, NOV). *Distributed control system*. Google Patents. (US Patent 4,173,754)
- Fernandes, D. A., & Soares. (2014). Security issues in cloud environments: a survey. *International Journal of Information Security*, 13(2), 113–170.
- Formu, J. (2009). Cloud cube model: selecting cloud formations for secure collaboration. *Google Scholar*. Retrieved from https://collaboration.opengroup.org/jericho/cloud_cube_model_v1.0.pdf
- Goyal, S. (2015). *Centralized vs decentralized vs distributed*. Retrieved 2015-07-01, from <https://medium.com/@bbc4468/centralized-vs-decentralized-vs-distributed-41d92d463868>
- Guo, Y., & Wang, C. (2005, March). Autonomous decentralized network security system. In *Proceedings. 2005 IEEE networking, sensing and control, 2005*. (p. 279-282). doi: 10.1109/ICNSC.2005.1461201
- Gupta, S., & Sanghwan, S. (2015). Load balancing in cloud computing: A review. *International Journal of Science, Engineering and Technology Research (IJSETR)*, 4(6).
- Gutierrez-Garcia, J. O., & Ramirez-Nafarrate, A. (2015). Agent-based load balancing in cloud data centers. *Cluster Computing*, 18(3), 1041–1062.
- Hamdeni, C., Hamrouni, T., & Charrada, F. B. (2016). Data popularity measurements in distributed systems: Survey and design directions. *Journal of Network and Computer Applications*, 72, 150 - 161. Retrieved from <http://www.sciencedirect.com/science/article/pii/S1084804516301205>

- Hevner, A., & Chatterjee, S. (2010). Design science research in information systems. In *Design research in information systems* (pp. 9–22). Springer. Retrieved from <https://pdfs.semanticscholar.org/408c/622746e3c70297613167a960d00cc9a212d7.pdf>
- Hevner, A. R. (2007). A three cycle view of design science research. *Scandinavian journal of information systems*, 19(2), 4.
- Hierarchical decentralized systems and its new solution by a barrier method. (1981). *IEEE Transactions on Systems, Man, and Cybernetics, Systems, Man and Cybernetics, IEEE Transactions on, IEEE Trans. Syst., Man, Cybern*(6), 444. doi: 10.1109/TSMC.1981.4308712
- Hough, D. (1993). Rapid delivery: An evolutionary approach for application development. *IBM Systems Journal*, 32(3), 397–419.
- Hussain, H., Malik, S. U. R., Hameed, A., Khan, S. U., Bickler, G., Min-Allah, N., ... Rayes, A. (2013). Review: A survey on resource allocation in high performance distributed computing systems. *Parallel Computing*, 39, 709 - 736. Retrieved from <http://www.sciencedirect.com/science/article/pii/S016781911300121X>
- Ihara, H., & Mori, K. (1984, Aug). Autonomous decentralized computer control systems. *Computer*, 17(8), 57-66. doi: 10.1109/MC.1984.1659218
- Internet security threat report* (Vol. 22). (2017). Symantec. Retrieved from https://s1.q4cdn.com/585930769/files/doc_downloads/lifelock/ISTR22_Main-FINAL-APR24.pdf
- Jain, R., & Paul, S. (2013). Network virtualization and software defined networking for cloud computing: A survey. *IEEE COMMUNICATIONS MAGAZINE*, 51(11), 24–31. Retrieved from <http://ieeexplore.ieee.org/abstract/document/6658648/>
- Jing, X., & Jian-jun, Z. (2010, Aug). A brief survey on the security model of cloud computing. In *2010 ninth international symposium on distributed computing and applications to business, engineering and science* (p. 475-478). doi: 10.1109/DCABES.2010.103
- Khan, M. A., Mishra, K. K., Santhi, N., & Jayakumari, J. (2015, April). A new hybrid technique for data encryption. In *Communication technologies (gcct), 2015 global conference on* (p. 925-929). doi: 10.1109/GCCT.2015.7342801
- Khiyaita, A., Bakkali, H. E., Zbakh, M., & Kettani, D. E. (2012, April). Load balancing cloud computing: State of art. In *2012 national days of network security and systems* (p. 106-109). doi: 10.1109/JNS2.2012.6249253
- Khorshed, M. T., Ali, A. S., & Wasimi, S. A. (2012). A survey on gaps, threat remediation challenges and some thoughts for proactive attack detection in cloud computing. *Future Generation Computer Systems*, 28, 833 - 851. Retrieved from <http://www.sciencedirect.com/science/article/pii/S0167739X12000180> doi: 10.1016/j.future.2012.01.006
- Kocarev, L., & Tasev, Z. (2003, May). Public-key encryption based on chebyshev maps. In *Circuits and systems, 2003. iscas '03. proceedings of the 2003 international symposium on* (Vol. 3, p. III-28-III-31 vol.3). doi: 10.1109/ISCAS.2003.1204947

- Koshan, M. (2015). *Cloud security report: Honeypot findings*. Retrieved 2015-05-05, from <http://www.cwps.com/blog/cloud-security-report-honeypot-findings>
- LD, D. B., & Krishna, P. V. (2013). Honey bee behavior inspired load balancing of tasks in cloud computing environments. *Applied Soft Computing*, 13(5), 2292–2303.
- Litchfield, A., Ahmed, M., & Sharma, C. (2016). A distributed security model for cloud computing. In *22nd americas conference on information systems* (pp. 1–10). San Diego, CA.
- Lombardi, F., & Di Pietro, R. (2011). Secure virtualization for cloud computing. *Journal of Network and Computer Applications*, 34(1), 1113 - 1122. Retrieved from <http://www.sciencedirect.com/science/article/pii/S1084804510001062>
- Lyu, M. R., et al. (1996). *Handbook of software reliability engineering*. Retrieved from http://s3.amazonaws.com/academia.edu.documents/24969624/56257_handbook_of_software_reliability_engineering_00_content_and_preface.pdf?AWSAccessKeyId=AKIAIWOWYYGZ2Y53UL3A&Expires=1500821366&Signature=%2Fy5bleelxGDbPDXf8%2BwyXYHYwB0%3D&response-content-disposition=inline%3B%20filename%3DHandbook_of_software_reliability_engineer.pdf
- Mackay, M., Baker, T., & Al-Yasiri, A. (2012). Security-oriented cloud computing platform for critical infrastructures. *Computer Law & Security Review*, 28(6), 679–686.
- Martin, J. (1991). *Rapid application development*. Macmillan Publishing Co., Inc.
- maxdml. (2017). *An overview of distributed computing frameworks*. Retrieved 2017-10-01, from <https://users.cs.duke.edu/~maxdml/drupal/?q=distributed-computing-intro>
- Mccaffrey, C. (2016). The verification of a distributed system. *Communications of the ACM*, 59(2), 52 - 55. Retrieved from <http://dl.acm.org/citation.cfm?id=2844108>
- McPhee, K. (1996). *Design theory and software design*. Retrieved from <https://era.library.ualberta.ca/files/m613n029s/TR96-26.pdf>
- Milani, A. S., & Navimipour, N. J. (2016). Review: Load balancing mechanisms and techniques in the cloud environments: Systematic literature review and future trends. *Journal of Network and Computer Applications*, 71, 86 - 98. Retrieved from http://www.sciencedirect.com.ezproxy.aut.ac.nz/science/article/pii/S1084804516301217?_rdoc=1&_fmt=high&_origin=gateway&_docanchor=&md5=b8429449ccfc9c30159a5f9aeaa92ffb doi: 10.1016/j.jnca.2016.06.003
- Mishra, K. S., & Tripathi, A. K. (2014). Some issues, challenges and problems of distributed software system. *International Journal of Computer Science and Information Technologies. Varanasi, India*, 7, 3.
- Mittal, M., Sangani, R., & Srivastava, K. (2015). Testing data integrity in

- distributed systems. In *Procedia computer science* (45th ed., Vol. 45, p. 446-452). Information Technology Department, DJSCOE. Retrieved from <http://www.sciencedirect.com/science/article/pii/S1877050915003130> doi: 10.1016/j.procs.2015.03.077
- Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A., Budhia, R. K., & Lingley-Papadopoulos, C. A. (1996). Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4), 54–63.
- Mouhib, I., & Driss, E. (2015). Enhanced data security approach for cloud environment based on various encryption techniques. *Journal of Theoretical and Applied Information Technology*, 80(3), 439-446. Retrieved from <http://search.proquest.com/openview/23e78381fcb4dcbbf7c7c7f8f8f06101/1?pq-origsite=gscholar&cbl=2040122>
- Nadeem, A., & Javed, M. Y. (2005, Aug). A performance comparison of data encryption algorithms. In *2005 international conference on information and communication technologies* (p. 84-89). doi: 10.1109/ICICT.2005.1598556
- Patel, D. K., Tripathy, D., & Tripathy, C. (2016). Review: Survey of load balancing techniques for grid. *Journal of Network and Computer Applications*. Retrieved from <http://www.sciencedirect.com/science/article/pii/S1084804516000953> doi: 10.1016/j.jnca.2016.02.012
- Peffer, K., Rothenberger, M., Tuunanen, T., & Vaezi, R. (2012). Design science research evaluation. *Design science research in information systems. Advances in theory and practice*, 398–410.
- Peffer, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of Management Information Systems*, 24(3), 45 - 77. Retrieved from <http://www.tandfonline.com/doi/abs/10.2753/MIS0742-1222240302>
- Peyravian, M., Matyas, S., & Zunic, N. (2002, March 26). *Decentralized systems methods and computer program products for sending secure messages among a group of nodes*. Google Patents. Retrieved from <https://www.google.com/patents/US6363154> (US Patent 6,363,154)
- Poh, G. S., Mohd Nazir, M. A. N., Goi, B.-M., Tan, S.-Y., Phan, R. C.-W., & Shamsudin, M. S. (2013). An authentication framework for peer-to-peer cloud. In *Proceedings of the 6th international conference on security of information and networks* (pp. 94–101).
- Pradeep, K., & Vijayakumar, V. (2015). Survey on the key management for securing the cloud. In *Procedia computer science* (50th ed., Vol. 50, p. 115-121). VIT University. Retrieved from http://www.sciencedirect.com.ezproxy.aut.ac.nz/science/article/pii/S1877050915005736?_rdoc=1&_fmt=high&_origin=gateway&_docanchor=&md5=b8429449ccfc9c30159a5f9aeaa92ffb doi: 10.1016/j.procs.2015.04.072
- Prat, N., Comyn-Wattiau, I., & Akoka, J. (2014). Artifact evaluation in information systems design-science research-a holistic view. In *Pacis* (p. 23).

- Ramezani, F., Lu, J., & Hussain, F. K. (2014). Task-based system load balancing in cloud computing using particle swarm optimization. *International journal of parallel programming*, 42(5), 739.
- Ramsoft consulting. (n.d.). <http://www.ramsoft.com.au/methodology.php>. (Accessed: 2016-11-28)
- Ratha, P., Swain, D., Paikaray, B., & Sahoo, S. (2015). An optimized encryption technique using an arbitrary matrix with probabilistic encryption. *Procedia Computer Science*, 57(3rd International Conference on Recent Trends in Computing 2015 (ICRTC-2015)), 1235 - 1241. Retrieved from <http://www.sciencedirect.com/science/article/pii/S1877050915019511>
- Sandell, N., Varaiya, P., Athans, M., & Safonov, M. (1978). Survey of decentralized control methods for large scale systems. *IEEE Transactions on automatic Control*, 23(2), 108–128.
- Sari, A., & Akkaya, M. (2015). Fault tolerance mechanisms in distributed systems. *International Journal of Communications, Network and System Sciences*, 8(12), 471.
- Sathyanarayana, T. V., & Sheela, L. M. I. (2013, Dec). Data security in cloud computing. In *2013 international conference on green computing, communication and conservation of energy (icgce)* (p. 822-827). doi: 10.1109/ICGCE.2013.6823547
- Sedayao, J., Su, S., Ma, X., Jiang, M., & Miao, K. (2009). *A simple technique for securing data at rest stored in a computing cloud*. (Vol. 5931 LNCS). Intel Corporation: Springer. Retrieved from <http://barbie.uta.edu/~hdfeng/CloudComputing/cc/cc15.pdf> doi: 10.1007/978-3-642-10665-1_51
- Shah, N., & Farik, M. (2015). Static load balancing algorithms in cloud computing: Challenges & solutions. *International Journal Of Scientific & Technology Research*, 4(10). Retrieved from <http://www.ijstr.org/final-print/oct2015/-Static-Load-Balancing-Algorithms-In-Cloud-Computing-Challenges-Solutions.pdf>
- Shaheen, S. H., Yousaf, M., & Majeed, M. Y. (2015, Dec). Comparative analysis of internet key exchange protocols. In *2015 international conference on information and communication technologies (icict)* (p. 1-6). doi: 10.1109/ICICT.2015.7469595
- Sheng, B., & Bastani, F. B. (2004, April). Secure and reliable decentralized peer-to-peer web cache. In *Parallel and distributed processing symposium, 2004. proceedings. 18th international* (p. 54-). doi: 10.1109/IPDPS.2004.1302976
- Singh, A., & Chatterjee, K. (2017). Cloud security issues and challenges: A survey. *Journal of Network & Computer Applications*, 79, 88 - 115. Retrieved from http://www.sciencedirect.com.ezproxy.aut.ac.nz/science/article/pii/S1084804516302983?_rdoc=1&_fmt=high&_origin=gateway&_docanchor=&md5=b8429449ccfc9c30159a5f9aeaa92ffb doi: 10.1016/j.jnca.2016.11.027

- Smara, M., Aliouat, M., Pathan, A.-S. K., & Aliouat, Z. (2016). Acceptance test for fault detection in component-based cloud computing and systems. *Future Generation Computer Systems*. Retrieved from <http://www.sciencedirect.com/science/article/pii/S0167739X16302151> doi: 10.1016/j.future.2016.06.030
- So, K. (2011). Cloud computing security issues and challenges. *International Journal of Computer Networks*, 3(5), 247–55.
- Sreenivas, V., Narasimham, C., Subrahmanyam, K., & Yellamma, P. (2013, July). Performance evaluation of encryption techniques and uploading of encrypted data in cloud. In *Computing, communications and networking technologies (icccnt), 2013 fourth international conference on* (p. 1-6). doi: 10.1109/ICCCNT.2013.6726514
- Takahashi, H., Mahmood, K., & Lakhani, U. (2015, March). Autonomous decentralized semantic based url filtering system for low latency. In *2015 ieee twelfth international symposium on autonomous decentralized systems* (p. 9-16). doi: 10.1109/ISADS.2015.35
- Unruh, D. (2015). Revocable quantum timed-release encryption. *Journal of the ACM*, 62(6), 49 - 49:76. Retrieved from <http://dl.acm.org/citation.cfm?id=2817206>
- Wang, C., Wang, Q., Ren, K., & Lou, W. (2010, March). Privacy-preserving public auditing for data storage security in cloud computing. In *2010 proceedings ieee infocom* (p. 1-9). doi: 10.1109/INFCOM.2010.5462173
- Xin, Z., Song-qing, L., & Nai-wen, L. (2012, Aug). Research on cloud computing data security model based on multi-dimension. In *Information technology in medicine and education (itme), 2012 international symposium on* (Vol. 2, p. 897-900). doi: 10.1109/ITiME.2012.6291448
- Xu, X. (2012). From cloud computing to cloud manufacturing. *Robotics and computer-integrated manufacturing*, 28(1), 75–86.
- Yang, K., & Jia, X. (2013). An efficient and secure dynamic auditing protocol for data storage in cloud computing. *IEEE transactions on parallel and distributed systems*, 24(9), 1717–1726.
- Zhang, X., Zagorodnov, D., Hiltunen, M., Marzullo, K., & Schlichting, R. D. (2004, Sept). Fault-tolerant grid services using primary-backup: feasibility and performance. In *Cluster computing, 2004 ieee international conference on* (p. 105-114). doi: 10.1109/CLUSTER.2004.1392608
- Zhao, X., & Xiao, M. (2015, Nov). Autonomous decentralized test system to enhance ats's survival probability and online maintainability. In *Ieee autotestcon, 2015* (p. 324-328). doi: 10.1109/AUTEST.2015.7356510
- Zissis, D., & Lekkas, D. (2012). Addressing cloud computing security issues. *Future Generation computer systems*, 28(3), 583–592.