การออกแบบและพัฒนาเฟรมเวิร์คสำหรับระบบไมโครเซอร์วิสแบบกระจาย

โดย

นายภากร คูกรินทร์รัตน์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตร
วิทยาศาสตรมหาบัณฑิต (วิทยาการคอมพิวเตอร์)
ภาควิชาวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์และเทคโนโลยี มหาวิทยาลัยธรรมศาสตร์
ปีการศึกษา 2588

การออกแบบและพัฒนาเฟรมเวิร์คสำหรับระบบไมโครเซอร์วิสแบบกระจาย

โดย

นายภากร คูกรินทร์รัตน์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตร
วิทยาศาสตรมหาบัณฑิต (วิทยาการคอมพิวเตอร์)
ภาควิชาวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์และเทคโนโลยี มหาวิทยาลัยธรรมศาสตร์
ปีการศึกษา 2588

Design and Implementation of a Decentralized Microservices Framework

BY

MR. Pakorn Kookarinrat

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF SCIENCE (COMPUTER SCIENCE)

DEPARTMENT OF COMPUTER SCIENCE

FACULTY OF SCIENCE AND TECHNOLOGY

THAMMASAT UNIVERSITY

ACADEMIC YEAR 2016

มหาวิทยาลัยธรรมศาสตร์
คณะวิทยาศาสตร์และเทคโนโลยี

วิทยานิพนธ์

ของ

นายภากร คูกรินทร์รัตน์

เรื่อง

การออกแบบและพัฒนาเฟรมเวิร์คสำหรับพัฒนาระบบไมโครเซอร์วิสแบบกระจาย

ได้รับการตรวจสอบและอนุมัติ ให้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตร
วิทยาศาสตรมหาบัณฑิต (วิทยาการคอมพิวเตอร์)

เมื่อ วันที่ 6 กรกฎาคม พ.ศ. 2559

ประธานกรรมการสอบวิทยานิพนธ์ _____
(ดร. ประภาพร รัตนธำรง)

กรรมการและอาจารย์ที่ปรึกษาวิทยานิพนธ์ _____
(รองศาสตราจารย์ ดร. เยาวดี เต็มธนาภัทร์)

กรรมการสอบวิทยานิพนธ์ _____
(ผู้ช่วยศาสตราจารย์ ดร. ทรงศักดิ์ รองวิริยะพานิช)

กรรมการสอบวิทยานิพนธ์ _____
(ผู้ช่วยศาสตราจารย์ ดร. ชลเมธ อาปณิกานนท์)

คณบดี _____
(รองศาสตราจารย์ ปกรณ์ เสริมสุข)

หัวข้อวิทยานิพนธ์	การออกแบบและพัฒนาเฟรมเวิร์คสำหรับพัฒนาระบบไมโครเซอร์วิส
ชื่อผู้เขียน	นายภากร คูกรินทร์รัตน์
ชื่อปริญญา	วิทยาศาสตรมหาบัณฑิต
สาขาวิชา/คณะ/มหาวิทยาลัย	สาขาวิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์และเทคโนโลยี มหาวิทยาลัยธรรมศาสตร์
อาจารย์ที่ปรึกษาวิทยานิพนธ์	รองศาสตราจารย์ ดร. เยาวดี เต็มธนาภัทธ์
ปีการศึกษา	2558

บทคัดย่อ

ในปัจจุบันสถาปัตยกรรมซอฟต์แวร์แบบใหม่ที่ชื่อ microservices กำลังได้รับความนิยม เนื่องจาก microservices สามารถแก้ไขปัญหาความซับซ้อนของโปรแกรมได้เป็นอย่างดี และส่งเสริมรูปแบบการพัฒนาที่มีการเปลี่ยนแปลงบ่อย รูปแบบของสถาปัตยกรรมเป็นการแบ่งซอฟต์แวร์ขนาดใหญ่ออกเป็น service ขนาดเล็กที่แยกจากกันอย่างเด็ดขาดหลาย service แต่ละ service จะติดต่อกันผ่าน remote call ทำให้การเปลี่ยนแปลงในแต่ละ service จะไม่กระทบกับ service อื่น อย่างไรก็ตาม การติดต่อกันผ่าน remote call โดยตรงจะทำให้เกิดการผูกติดกันระหว่าง service เพื่อกำจัดการผูกติดกันระหว่าง service งานวิจัยชิ้นนี้นำเสนอการใช้ message bus แบบกระจายเป็นเครื่องมือใช้ในการสื่อสารระหว่าง service ใน microservices มีส่วนประกอบดังต่อไปนี้ public API, messaging, load-balance และ service discovery ใน public API จะใช้ HTTP และ RESTful ในการติดต่อกับ service ในขณะที่ service discovery จะเป็นแบบกระจายเพื่อป้องการปัญหา single point of failure เราได้ทำการทดสอบ message bus ด้วยการนำไปพัฒนาระบบจำลองที่มีรูปแบบการทำงานที่ส่วนใหญ่มักจะเกิดขึ้นกับ microservices ซึ่งระบบสามารถทำงานได้อย่างถูกต้อง และเรายังทำการทดลองความสามารถในการขยายตัวของ message bus โดยการเพิ่มจำนวนของ service อย่างต่อเนื่องแล้วทำการตรวจวัดปริมาณการใช้ทรัพยากรในระบบของ message bus ซึ่ง CPU และ memory มีการใช้งานและอัตราการเพิ่มขึ้นของการใช้งานเพียงเล็กน้อยในขณะที่ อัตราการใช้งาน network มีอัตราการเพิ่มขึ้นเป็นแบบ linear

คำสำคัญ: microservices, message bus, สถาปัตยกรรมซอฟต์แวร์

Thesis Title	Design and Implementation of a Decentralized Microservices Framework
Author	Mr. Pakorn Kookarinrat
Degree	Master of Science
Major Field/Faculty/University	Computer Science Faculty of Science and Technology Thammasat University
Thesis Advisor	Associate Professor Yaowadee Temtanapat
Academic Years	2015

ABSTRACT

A new software architecture, known as microservices, becomes rapidly popular recently. Microservices could help developers cope well with the problems of software complexity and demands on an adaptive development process that needs to respond to changes quickly. In this architecture, a single monolithic large application would be divided into small multiple isolated services. They are separately deployed and communicated to other services via remote calls. This architectural style allows any changes on one service not affecting the others. However, if services directly make remote calls, it would create interdependencies and tight couplings between them. To remove such problem, this paper proposes a decentralized message bus to use as a communication tool between services. Our message bus provides a framework for services to collaborate. It divides into four main components, public API, message bus, messaging and service discovery. The API uses the HTTP and RESTful style of communication. We use decentralized service discovery to avoid a single point of failure of the system. The messaging uses a simple TCP connection with only a header and body in its message. We also define three necessary communication messages for the services, viz. request/response, notification and publish/subscribe. The proposed framework is implemented and tested with a real-world scenario. It works correctly without any problem. Also, to

realize how it could be scaled, we run the system continuously with incremental services and traffics. From the observation on the resource consumption of CPU, memory and network I/O, we found that the network consumption grows linearly while the CPU and memory usages have little change in consumption

Keywords: microservices, message bus, software architecture

กิตติกรรมประกาศ

วิทยานิพนธ์ฉบับนี้สำเร็จลุล่วงได้เป็นอย่างดี เนื่องจากได้รับความอนุเคราะห์จากรองศาสตราจารย์ เยาวดี เต็มธนาภัทธ์ ซึ่งเป็นอาจารย์ที่ปรึกษาและควบคุมวิทยานิพนธ์ เป็นผู้จุดประกายหัวข้อวิทยานิพนธ์ ให้คำปรึกษาแนะนำ และแนวคิด ช่วยแก้ไขภาษาอังกฤษ ตลอดจนให้กำลังใจแก่ผู้วิจัยเสมอมา รวมถึงคอยช่วยสนับสนุนแนะนำผู้วิจัยในเรื่องของการลงทะเบียนเพื่อนำเสนอผลงานวิจัยในงานประชุมวิชาการนานาชาติ และขอขอบพระคุณ ดร. ประภาพร รัตนธำรง ผู้ช่วยศาสตราจารย์ ดร. ทรงศักดิ์ รองวิริยะพานิช และ ผู้ช่วยศาสตราจารย์ ดร. ชลเมธ อาปณิกานนท์ ที่ช่วยมาเป็นกรรมการการสอบ อีกทั้งยังช่วยให้คำแนะนำในการแก้ไขวิทยานิพนธ์ให้มีความสมบูรณ์ยิ่งขึ้น ผู้วิจัยซาบซึ้งในความกรุณา และขอขอบพระคุณทุกท่านเป็นอย่างสูงไว้ ณ โอกาสนี้ด้วย

นายภากร คูกรินทร์รัตน์

สารบัญ

สารบัญตาราง

สารบัญภาพ

บทที่ 1
บทนำ

1.1 ปัญหา

Microservices architecture style (Fowler & Lewis, 2014) เป็นรูปแบบการพัฒนาระบบที่เพิ่งเกิดขึ้นใหม่เมื่อไม่นานมานี้ แต่ถูกกล่าวถึงอย่างแพร่หลาย สไตล์ของสถาปัตยกรรม Microservices ใช้การแบ่งระบบใหญ่ออกเป็น service เล็ก ๆ หลาย services โดยไม่ได้กำหนดหรือนิยามวิธีการแบ่งไว้ อย่างไรก็ตามแนวทางการแบ่งที่นิยม ได้แก่ การแบ่งระบบตาม context ของ Context boundary ใน Domain driven design หรือ การแบ่งด้วยการให้แต่ละ service มี SRP (Single responsibility principle) คือ service มีหน้าที่ที่ทำงานในลักษณะเดียวกัน ทำให้แต่ละ service สามารถถูก deploy แยกกันได้ (self-container) แต่ยังคงการสื่อสารกันได้ผ่าน lightweight protocol ที่กำหนด

ในปัจจุบันสถาปัตยกรรม microservices ได้รับความนิยมเพิ่มขึ้น น่าจะมีสาเหตุจากการที่ microservices ช่วยลดความซับซ้อนการพัฒนาระบบลง และสอดรับกับรูปแบบการพัฒนาซอฟต์แวร์แบบ Agile ในด้านความซับซ้อนการพัฒนา จากแนวคิดการแบ่ง service เป็นระบบย่อยที่เบ็ดเสร็จสามารถ deploy แยกจากกัน แต่ยังคงทำงานร่วมกันได้ ทำให้ service ย่อยที่เป็นบริการขนาดเล็กมีความซับซ้อนของการพัฒนาลดลงตามไปด้วย ดังจะเห็นได้จากวิธีการวัดระดับความซับซ้อนของระบบ (McCabe, 1976 and Henry & Kafura, 1982) ที่มักสัมพันธ์กับการวัดขนาดของโปรแกรม เช่น วิธีการวัดความซับซ้อนของระบบจากจำนวนบรรทัด จำนวนการทำงาน จำนวนเส้นทางการเดินของโปรแกรม เป็นต้น ดังนั้นเมื่อแยกระบบเป็น service ขนาดเล็กทำหน้าที่เพียงบริการเฉพาะเรื่องลักษณะเดียว ขนาดของโปรแกรมจึงเล็กลง ความซับซ้อนจึงลดลงตามไปด้วย

นอกจากนี้การที่ service ของ microservices นั้นสามารถ deploy แยกออกจากกันได้ ช่วยให้เกิดความยืดหยุ่น และง่ายในการเปลี่ยนแปลง service โดยไม่ส่งผลกระทบกับ service อื่น ๆ จึงสอดคล้องกับรูปแบบการบริหารจัดการซอฟต์แวร์แบบ Agile ที่เป็นที่นิยมในปัจจุบัน โดยเฉพาะอย่างยิ่งในกลุ่มบริษัท startup เนื่องจากแนวคิดการพัฒนาของ Agile เน้นการ release product ให้รวดเร็ว ต่อเนื่อง เพื่อให้ตรงตามความต้องการของผู้ใช้งาน ในการพัฒนาระบบแบบ Agile จึงใช้วิธีการพัฒนาระบบแบบทยอยสร้าง (incremental build) คือการสร้างระบบไปทีละส่วน ซึ่งระบบไม่จำเป็นจะต้องเสร็จทั้งหมดจึงจะถูก deploy ให้ผู้ใช้ใช้งาน จึงได้รับข้อมูล feedback จากผู้ใช้อย่างรวดเร็ว ช่วยให้สามารถปรับเปลี่ยนระบบให้ตรงกับความต้องการของผู้ใช้งานได้ทันท่วงที ทำให้การปรับเปลี่ยนระบบเกิดขึ้นบ่อย จึงต้องการวิธีการพัฒนาที่ตอบสนองต่อการเปลี่ยนแปลงได้อย่าง

ยืดหยุ่น รองรับการปรับเปลี่ยนบริการได้ง่าย และบ่อยกว่ารูปแบบการพัฒนาแบบดั้งเดิม ซึ่งสอดคล้องโดยตรงกับสถาปัตยกรรมของ Microservices ที่เกิดจากการประกอบกันของ service ขนาดเล็กที่เป็นอิสระต่อกันหลาย service ทำให้การทยอยสร้างระบบเป็นไปได้ง่าย อีกทั้ง service สามารถเปลี่ยนแปลงได้โดยที่ไม่กระทบกับบริการอื่น ๆ จึงง่ายต่อการเปลี่ยนแปลงแก้ไข

แม้ Microservices จะมีความเหมาะสมหลายประการในแง่การพัฒนาซอฟต์แวร์ในปัจจุบัน แต่ Microservices ก็เป็นระบบแบบกระจาย (Distributed system) จึงมีความยุ่งยากแบบเดียวกับระบบการกระจายตัวทั่วไป โดยอาจสรุปข้อด้อยซึ่งเกิดจากระบบการกระจายและเกี่ยวข้องกับการใช้ microservices ได้เป็น 4 ข้อดังนี้

1. เมื่อระบบถูกแบ่งออกเป็นหลาย service ทำให้จำนวนเครื่องในระบบเพิ่มขึ้นจึงยากต่อการจัดการดูแล (Hoff, 2015) และหากทำด้วยคนจะใช้เวลานาน ตัวอย่างเช่น งาน update service งานติดตั้ง (configuration) ระบบ งานกำหนดการเชื่อมต่อระหว่างระบบ ทำให้การทำ automation นั้นมีความจำเป็นสำหรับระบบ แต่การทำระบบให้เป็น automation จะเพิ่มความซับซ้อนของการดำเนินการ ดังนั้นจึงต้องการเครื่องมือมาช่วยในการลดความซับซ้อนของการดำเนินการที่เพิ่มขึ้น เช่น การบริการ service discovery เพื่อช่วยทำให้ service แต่ละ service รู้จักกันอัตโนมัติ หรือ configuration management tools ซึ่งจะช่วยให้เราสามารถสร้าง environment ของระบบได้แบบอัตโนมัติ เครื่องมือที่กล่าวมาเหล่านี้จะช่วยทำให้การทำระบบ automation เป็นไปได้ง่ายขึ้น

2. Communication การเปลี่ยนแปลงระบบจาก local function call มาเป็น remote call ทำให้ประสิทธิภาพของระบบลดลงเนื่องจาก latency ของ network (Richardson, 2015) แต่การพัฒนาระบบในการทำ remote call ให้มีประสิทธิภาพนั้นเป็นเรื่องที่ยุ่งยากและต้องการความเข้าใจในด้าน network ดังนั้นการใช้ lightweight communication infrastructure เป็นเรื่องที่จำเป็นเพื่อให้ผู้พัฒนาสามารถสนใจแค่การพัฒนาระบบที่ตอบสนองกับความต้องการทางธุรกิจได้

3. การผูกติดกันของ service การเปลี่ยน local function call เป็น remote call จะทำให้เกิดการผูกติดกันระหว่าง service (Mason, 2011) เพราะการทำ remote call จะทำโดยการ fix static IP ทำให้เกิด static connection ระหว่าง service และรูปแบบการเรียกใช้ service แต่ละ service ที่อาจจะไม่เหมือนกันก็จะทำให้ service ผูกติดอยู่กับ service ที่เรียกใช้ ทำให้การเปลี่ยนแปลงระบบเป็นเรื่องยาก เนื่องจาก service แต่ละ service มีการผูกติดกันที่สูง และ flexibility ของระบบลดลง ทำให้ระบบต้องการสิ่งที่มาจัดการกับการผูกติดกันระหว่าง service ที่เกิดจากการใช้ remote call

4. Availability และ Scalability เป็นสิ่งที่สำคัญกับระบบในยุคปัจจุบันโดยเฉพาะระบบทาง Internet ที่ความลื่นไหลในการใช้งานนั้น มีผลกับผู้ใช้เป็นอย่างมาก โดยส่วนใหญ่การเพิ่ม

Availability และ Scalability เป็นการเพิ่มเครื่องที่มีการทำงานเหมือนกันเข้ามาช่วยกันทำงานหรือทำงานทดแทนกัน แต่การที่ microservices แบ่งระบบใหญ่ออกเป็น service ย่อยหลาย service นั้นทำให้การเพิ่ม Availability และ Scalability เป็นเรื่องที่ทำได้ยาก ยกตัวอย่างเช่น จากที่เคยทำ load balance กับระบบใหญ่อยู่ระบบเดียวอาจต้องทำ load balance ให้กับทุก service ทำให้ระบบต้องการเครื่องมือหรือวิธีการที่ช่วยในการเพิ่ม Availability และ Scalability ของระบบที่ใช้ microservices

จากปัญหาของการสื่อสารของ microservices จะเห็นว่าลักษณะการสื่อสารระหว่าง service ของ microservices มี 2 รูปแบบคือ point-to-point และ message bus ในกรณีของ point-to-point เป็นการสื่อสารระหว่าง service แบบเชื่อมต่อตรง ซึ่งทำให้เกิดการผูกติดกันของ service การขยายตัวจึงเป็นไปได้ยาก ในขณะที่การสื่อสารอีกรูปแบบหนึ่งคือ message bus ซึ่งส่วนมากใช้ centralized message queue เป็นตัวกลางในการสื่อสาร ทำให้ลดการผูกติดกันของ service แต่จะมี single point of failure อีกทั้งยังทำให้ service ต้องส่งข้อมูลบน network มากกว่าหนึ่งครั้งสำหรับหนึ่งการสื่อสาร จนส่งผลให้ประสิทธิภาพลดลงจาก delay ของ network เพื่อแก้ไขปัญหาดังกล่าว งานวิจัยนี้จึงนำเสนอวิธีการแก้ไขปัญหา ด้วยการออกแบบ message bus เพื่อใช้ในการสื่อสารระหว่าง service ใน microservices ซึ่งลดการผูกติดกันของ service ในขณะเดียวกันก็ใช้ architecture แบบ decentralized เพื่อกำจัด single point of failure อีกทั้งยังแก้ปัญหาการส่งข้อมูลบน network มากกว่าหนึ่งครั้งในหนึ่งการสื่อสาร ที่มาจากการใช้ centralized message bus โดยการสื่อสารระหว่าง service จะอยู่ในรูปแบบ single network communication ทำให้ประสิทธิภาพที่ได้ไม่ต่างจากการสื่อสารแบบ point-to-point มากนัก และสุดท้ายเพิ่ม availability และ scalability ของระบบที่เป็นเรื่องสำคัญสำหรับการพัฒนาในยุคปัจจุบัน

1.2 วัตถุประสงค์

ออกแบบและพัฒนา framework ที่ใช้ในการพัฒนาระบบแบบ microservices architecture style ที่มีลักษณะดังต่อไปนี้

1. สนับสนุนการทำ automation
2. ลักษณะการสื่อสารเป็นแบบ single network communication
3. ลดการผูกติดกันระหว่าง services
4. สนับสนุน Scalability และ Availability

1.3 ขอบเขต

งานวิจัยชิ้นนี้เป็นการออกแบบระบบ message bus และ decentralized service discovery มาใช้ร่วมกันเพื่อช่วยในการพัฒนาระบบแบบ microservices เพื่อให้การพัฒนาระบบเป็นไปอย่างมีประสิทธิภาพโดยมีขั้นตอน วิธีการและกรอบการทดลองดังต่อไปนี้

1. ศึกษางานที่เกี่ยวข้องและความรู้ที่ใช้ในการพัฒนาระบบ decentralized
2. ออกแบบ decentralized message bus เพื่อเป็นเครื่องมือที่ช่วยในการพัฒนาระบบ
3. พัฒนาระบบ decentralized message bus โดยใช้ภาษา Golang ในการพัฒนา
4. ออกแบบและทดสอบระบบที่ได้จัดเตรียมไว้เข้ามาใช้ในระบบที่สร้างขึ้นซึ่งจะอยู่ในเครือข่ายวงเดียวกัน แล้วจัดเก็บข้อมูล
5. วิเคราะห์ข้อมูลที่ได้จากการทดสอบทั้งในส่วนของประสิทธิภาพการขยายตัว ประสิทธิภาพของการส่งข้อมูลระหว่าง service และความสามารถในการทำงานตามรูปแบบการทำงานที่มักจะเกิดขึ้นของระบบ microservices

1.4 ประโยชน์ที่ได้รับจากงานวิจัย

1. ลดความยุ่งยากในการจัดการระบบที่เกิดจากจำนวนระบบของ microservices โดยสร้างระบบ service discovery ให้แก่ผู้พัฒนา เพื่อการสร้างระบบ automation ทำได้ง่ายขึ้น
2. ลดการผูกติดกันของระบบย่อยจากการติดต่อสื่อสารกันของ service ด้วย remote call ใน microservices โดยใช้ message bus ทำให้ service สามารถถอดเข้า-ออกจากระบบได้ง่ายทำให้เหมาะสมกับ agile process development
3. ลดงานของผู้พัฒนาที่ต้องทำในส่วนของการติดต่อกันระหว่าง service ช่วยทำให้ผู้พัฒนาสามารถมุ่งเน้นเฉพาะในส่วนของการสร้าง service ได้
4. เพิ่มประสิทธิภาพของระบบ microservices ในด้าน Scalability และ Availability

บทที่ 2
วรรณกรรมและงานวิจัยที่เกี่ยวข้อง

งานวิจัยนี้เป็นการออกแบบการสร้าง microservices โดยใช้ decentralized message bus เป็นระบบสื่อสาร โดยให้ความสนใจเกี่ยวกับ distributed system และรูปแบบการสร้างการสื่อสารที่มีประสิทธิภาพเป็นหลัก เพื่อให้ microservices มีความคงทนต่อความล้มเหลวและลด operational complexity และปัญหาจากการสื่อสารของ service ในบทนี้จึงนำเสนอวรรณกรรมและงานวิจัยที่เกี่ยวข้อง ดังต่อไปนี้

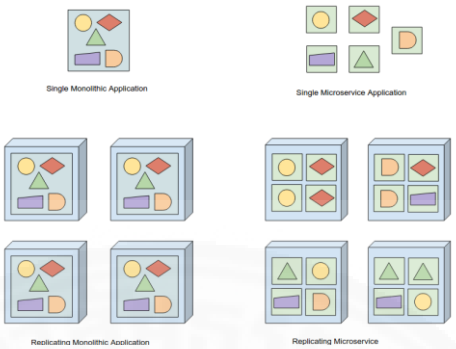
1. Microservices
2. Service discovery tools
3. SWIM Protocol
4. Message bus
5. Service-oriented architecture
6. RESTful
7. งานวิจัยที่เกี่ยวข้อง

2.1 Microservices

Microservices เป็น architecture style แบบใหม่ที่เริ่มได้รับความสนใจเมื่อประมาณปี 2013 โดยเริ่มต้นจากบริษัท Netflix microservices เป็น architecture style ที่แบ่ง single complex application ออกเป็นหลายๆ service โดยแต่ละ service ที่ถูกแบ่งออกมาจะแยกจากกันอย่างเด็ดขาด เวลาติดต่อกันจะติดต่อกันผ่านทาง remote API และเวลา deploy จะ deploy แยกกัน microservices นิยมนำมาเปรียบเทียบกับ Monolithic application ซึ่งเป็นรูปแบบการสร้าง application ที่ถูกใช้มาตั้งแต่แรก นั่นคือการสร้าง application ที่ทำงานอยู่บน process เดียวกัน โดย Monolithic application เป็นสาเหตุของการเกิด microservices เพราะ complexity ที่มาจาก monolithic application มีสูงมากเพราะตัวระบบมีขนาดใหญ่ และเนื่องจากขนาดของระบบที่มีขนาดใหญ่จึงทำให้ใช้เวลาในการ deploy ระบบนาน และเมื่อต้องการเปลี่ยนแค่บางส่วนของระบบจำเป็นต้อง deploy ใหม่ทั้งระบบ

เนื่องจาก Microservices เป็น architecture style ที่เพิ่งเกิดใหม่ จึงยังไม่มีรูปแบบและลักษณะการสร้างที่ชัดเจน แต่สามารถสรุปตามรูปแบบการใช้งานของหลายบริษัท ที่มีการเปิดเผยเป็นลักษณะของ microservices โดยแบ่งตาม Martin Fowler และ James Lewis (Folwer &

Lewis, 2014) ซึ่งเป็นผู้เชี่ยวชาญด้าน Software design และผ่านการสร้าง Software ด้วย Microservices architecture มาหลายระบบ ได้ดังต่อไปนี้



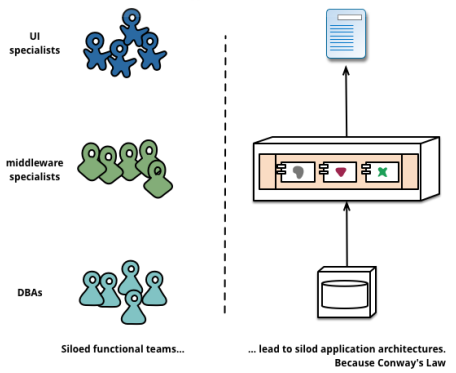
รูปที่ 2.1 Monolith & Microservices (Folwer & Lewis, 2014)

2.1.1 Componentization via service

การพัฒนาระบบหนึ่งในรูปแบบที่นิยมใช้ในการพัฒนาคือ componentization ซึ่งถูก microservices นำมาใช้ในการพัฒนาระบบ โดยระบบจะถูกสร้างด้วยการต่อ component เข้าด้วยกัน และจะสื่อสารกันผ่าน interface ซึ่งใน microservices component จะอยู่ในรูปแบบ service ที่สื่อสารกันผ่านรูปแบบการติดต่อสื่อสารกลาง ข้อดีของระบบ componentization คือ ระบบจะมีความยืดหยุ่นสูง เนื่องจาก component สามารถถอดเข้าออกได้ง่าย ทำให้ระบบสามารถปรับเปลี่ยนได้อย่างรวดเร็ว โดย microservices มีข้อดีเหมือนที่กล่าวไว้ข้างต้นด้วย

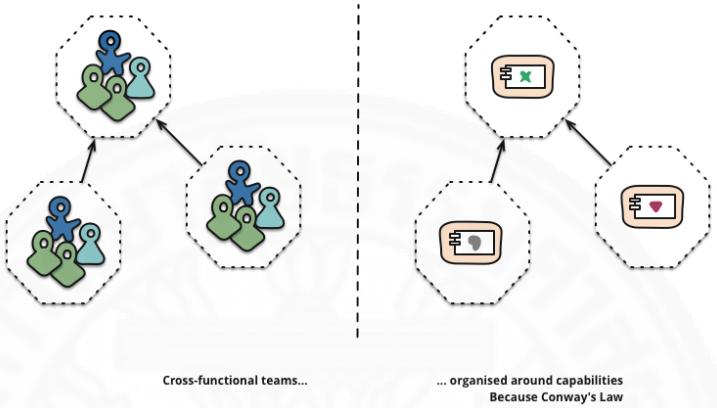
2.1.2 Organized around business capabilities

รูปแบบการบริหารจัดการระบบ software ในปัจจุบันส่วนใหญ่ถูกแบ่งตามชั้นของเทคโนโลยี ดังรูปที่ 2.2 ตัวอย่างเช่น ทีม UI ทีม server และทีมฐานข้อมูล ซึ่งรูปแบบการจัดการแบบนี้ การเปลี่ยนแปลงเพียงเล็กน้อยจะทำให้เกิดการสื่อสารระหว่างทีม เนื่องจากการเปลี่ยนแปลงระบบมักต้องการความรู้ความเข้าใจในระบบ การสื่อสารระหว่างทีมมักทำให้เกิดความล่าช้าในการพัฒนา



รูปที่ 2.2 รูปแบบการแบ่งทีมตามเทคโนโลยี (Folwer & Lewis, 2014)

ในขณะที่ทีมในระบบ microservices จะถูกแบ่งตาม service ใน microservices ดังรูปที่ 2.3 โดยแต่ละทีมจะมีนักพัฒนาครบทุกตำแหน่งที่จำเป็นต่อการพัฒนา ทำให้การเปลี่ยนแปลงใน service เกิดการสื่อสารระหว่างทีมเท่านั้น ยกเว้นเกิดการเปลี่ยนแปลง service ที่กระทบกับ service อื่น จึงจะทำให้เกิดการสื่อสารข้ามทีม แต่การเปลี่ยนแปลงแบบนี้เกิดขึ้นไม่บ่อย ซึ่งการสื่อสารภายในทีมจะมีความรวดเร็วกว่าการสื่อสารข้ามทีม ทำให้การพัฒนาระบบเป็นไปอย่างรวดเร็ว



รูปที่ 2.3 รูปแบบการแบ่งทีมตามความสามารถทางธุรกิจ (Folwer & Lewis, 2014)

2.1.3 Products not projects

รูปแบบการพัฒนาระบบในปัจจุบัน โดยมากเมื่อระบบถูกพัฒนาเสร็จทีมที่พัฒนาระบบจะส่งต่อให้แผนกอื่นดูแลต่อไป ทำให้ระบบมักขาดการดูแล หรือแม้กระทั่งต้องสร้างระบบใหม่ทดแทนระบบเดิม เนื่องจากทีมที่ดูแลไม่มีความรู้ความเข้าใจในตัวระบบเดิม แต่ใน microservices นั้นทีมพัฒนาจะทำหน้าที่เป็นเจ้าของ service ซึ่งมีหน้าที่ดูแล service ตั้งแต่เริ่มสร้างจนถึงการหยุดการใช้งาน ทำให้การปรับปรุงหรือเปลี่ยนแปลงแก้ไข service เป็นไปอย่างรวดเร็ว เนื่องจากนักพัฒนามีความรู้และความเข้าใจในตัว service

2.1.4 Smart endpoints and dumb pipes

Service ใน microservices จะติดต่อกันผ่านทาง lightweight protocol ซึ่งปกติแล้วจะนิยมใช้ RESTful HTTP ติดต่อระหว่าง service โดยตรง โดยส่งข้อมูลในรูปแบบ JSON อีกวิธีหนึ่งที่นิยมใช้เป็นช่องทางการสื่อสารของ service ใน microservices นั้นคือการใช้ lightweight message bus ตัวอย่างเช่น RabbitMQ หรือ ZeroMQ ซึ่งมีความสามารถมากกว่าการติดต่อโดยตรงระหว่าง service จึงช่วยลดการผูกติดกันระหว่าง service ในระบบและระบบจะมีข้อตกลงกลางในการสื่อสารระหว่าง service ทำให้ service ง่ายต่อการถอดเข้าและถอดออกจากระบบ อีกทั้งเมื่อข้อมูลทุกข้อมูลต้องผ่าน message bus ทำให้สามารถที่จะ monitor ข้อมูลในระบบได้ง่ายกว่าวิธีการแรก แต่จะไม่ใช้ Enterprise message bus หรือ ESB ซึ่งเป็น message bus ที่นิยมในการสร้างระบบ SOA เพราะไม่ต้องการเพิ่มความซับซ้อนในการใช้งานและการดูแลให้กับระบบ

2.1.5 Decentralized Governance

การที่ service ใน microservices เป็น self-container ทำให้ service มีอิสระในการเลือกเครื่องมือในการสร้างระบบ ส่งผลให้ service ใน microservices สามารถเลือกเครื่องมือในการสร้างระบบที่เหมาะสมกับการพัฒนาระบบได้ ในการพัฒนาระบบนั้น เครื่องมือที่เหมาะสมกับระบบส่งผลต่อประสิทธิภาพในการพัฒนาระบบอย่างมาก ตัวอย่างเช่น Node.js เหมาะกับระบบประเภทที่เน้นการทำงานกับ IO แต่ Golang เหมาะกับระบบประเภทที่ต้องการพลังในการคำนวณ ทำให้การพัฒนาระบบแบบ microservices สามารถพัฒนาระบบได้อย่างมีประสิทธิภาพมากกว่าระบบแบบ monolith ที่อยู่ในลักษณะระบบใหญ่ระบบเดียวจึงไม่สามารถที่จะเลือกเครื่องมือในการพัฒนาให้เหมาะกับทุก module ในระบบได้

2.1.6 Decentralized data management

Microservices service มักจะมีฐานข้อมูลเป็นของตัวเองไม่ได้ใช้ฐานข้อมูลรวมกันเหมือนในระบบแบบ monolith เพราะในแต่ละ service จะมีลักษณะความหมายของข้อมูลที่ไม่เหมือนกัน คำบางคำชื่อเรียกอาจจะเหมือนกันแต่ความหมายอาจจะต่างกันออกไป และลักษณะข้อมูลแต่ละแบบก็จะเหมาะสมกับฐานข้อมูลที่ต่างกันออกไป การใช้ฐานข้อมูลที่เหมาะกับลักษณะข้อมูลจะทำให้ระบบสามารถดึงข้อมูลอย่างมีประสิทธิภาพมากกว่าการใช้ฐานข้อมูลที่ไม่เหมาะสมกับรูปแบบข้อมูล ดังนั้นในแต่ละ service ใน microservices มักจะใช้ฐานข้อมูลแยกออกจากกัน โดยเลือกฐานข้อมูลที่เหมาะสมกับลักษณะของข้อมูลที่ service เก็บเพื่อประสิทธิภาพของระบบ

2.1.7 Infrastructure Automation

Microservices นั้นระบบจะถูกแบ่งออกเป็น service ขนาดเล็กจำนวนมากทำให้จำนวนเครื่องที่ต้องดูแลมีจำนวนมากตามไปด้วย ทำให้การดูแลระบบใช้เวลาและทรัพยากรคน อีกทั้งการที่ระบบ microservices นั้นมีลักษณะเป็น distributed system ทำให้เพิ่มความยุ่งยากในการดูแลระบบ ดังนั้นระบบ microservices มักจะถูกออกแบบให้เป็นระบบแบบ automation เพื่อลดความยุ่งยากในการดูแลและจัดการระบบ และด้วยความต้องการ automation ระบบมักจะถูกสร้างขึ้นบนระบบ cloud ที่สามารถบริหารจัดการเครื่องคอมพิวเตอร์จำลองได้อย่างง่ายดาย ส่งผลให้การสร้างระบบแบบ automation ง่ายขึ้น

2.1.8 Design of failure

การเปลี่ยนจาก single application มาเป็น multiple service ทำให้มี service ที่ต้องดูแลเป็นจำนวนมาก โอกาสที่ service จะหยุดทำงานจากสาเหตุ hardware หรือ software มีมากขึ้น ทำให้การออกแบบระบบ microservices จำเป็นต้องมีการนำเอาวิธีการต่างๆที่ช่วยลดโอกาส

การล่มของระบบลง ตัวอย่างเช่น การทำ replication service การทำ real time monitoring เพื่อดูความผิดปกติของ service เป็นต้น

2.1.9 Evolutionary Design

การเปลี่ยนจากระบบ Monolithic เป็น microservices มักจะเปลี่ยนแปลงทีละส่วน โดยส่วนที่ถูกแก้ไขบ่อยจะถูกเปลี่ยนเป็น microservices ก่อน เพราะทำให้สามารถแก้ไขส่วนนั้นได้ง่ายไม่กระทบกับทั้งระบบ

2.2 Service discovery

Service discovery หรือ locating service เป็นส่วนประกอบที่สำคัญของระบบ microservices เพราะในระบบ microservices นั้น service มีโอกาสเปลี่ยนแปลงที่อยู่ตลอดเวลา ดังนั้นระบบจึงต้องการเครื่องมือที่สามารถจัดหา IP และ port ของ service ที่เราต้องการติดต่อ นั่นคือ service discovery ซึ่งในปัจจุบัน service discovery มีหลายตัว บางตัวถูกสร้างมาเพื่อทำ service discovery โดยเฉพาะ ในขณะที่บางตัวไม่ได้สร้างมาเพื่อเป็น service discovery โดยตรง แต่ด้วยความสามารถที่เหมาะกับการทำ service discovery จึงถูกนำมาใช้งานเป็น service discovery โดยสามารถแบ่งออกเป็นกลุ่มตามรูปแบบ architecture ได้ดังต่อไปนี้

2.2.1 Centralized service discovery

Service discovery รูปแบบนี้จะมีส่วนประกอบสำคัญสองส่วนคือ service registration และ service discovery โดย service registration จะทำหน้าที่เก็บข้อมูลของ service ทั้งหมดใน cluster ในขณะที่ service discovery ทำหน้าที่จัดหาที่อยู่ service ให้กับ service ที่ร้องขอ โดยการอ่านค่าที่อยู่จาก service registration จากรูปแบบ architecture นี้ service registration จึงเป็นจุดให้บริการเชื่อม (broker) ทำให้เกิด single point of failure ซึ่งมีโอกาสที่ระบบจะทำงานไม่ได้เมื่อจุดเชื่อมต่อหยุดทำงาน แต่ข้อดีของการที่อ่านข้อมูลมาจากจุดเชื่อมต่อจุดเดียว จะทำให้ข้อมูลที่อยู่ของ service นั้นถูกต้องตลอดเสมอ รูปแบบประเภทนี้จึงนิยมใช้เมื่อต้องการความถูกต้องของข้อมูลสูง เครื่องมือที่เป็นรูปแบบ centralized service discovery นั้นมีอยู่สองรูปแบบดังต่อไปนี้

2.2.1.1 General purpose service registration

General purpose service registration เป็นเครื่องมือที่ถูกสร้างมาเพื่อใช้ในการจัดเก็บข้อมูลทั่วไปไม่ได้เจาะจงเพื่อทำ service discovery แต่ด้วยความสามารถที่เหมาะในการทำ service registration ด้วย จึงถูกนำมาใช้อย่างแพร่หลาย แต่ต้องพัฒนาเพิ่มเติมเพื่อให้เหมาะสมกับการใช้งานเป็น service discovery โดย Zookeeper (Zookeeper, 2010) เป็นหนึ่งในเครื่องมือที่ได้รับความนิยม การเก็บข้อมูลของ Zookeeper มีลักษณะเป็น node hierarchy โดย

จุดเด่นของมัน คือความถูกต้องของข้อมูล (consistency) ข้อมูลที่ได้จาก server ใน cluster จะมีค่าเท่ากันหมด การติดต่อกับ Zookeeper จะติดต่อผ่านทาง client ที่จัดเตรียมไว้ให้ อีกหนึ่งเครื่องมือที่ถูกนำมาใช้เป็น service discovery ได้แก่ Etcd (CoreOS, 2016) โดยที่ Etcd เก็บค่าของข้อมูลอยู่ในรูปแบบ key-value โดยสามารถติดต่อ Etcd ผ่านทาง HTTP + JSON API สนับสนุน Availability ด้วยการใช้ replication ในรูปแบบ master-salve โดยจุดเด่นของ Etcd อยู่ที่ความถูกต้องของข้อมูลเช่นเดียวกับ Zookeeper แต่สามารถอ่านข้อมูลของ service ได้จาก server ที่ไม่ใช่ master ทำให้ประสิทธิภาพในการอ่านข้อมูลดีกว่า Zookeeper

2.2.1.2 Single purpose service discovery

เป็นเครื่องมือที่สร้างมาเพื่อทำ service discovery โดยเฉพาะ ดังนั้นเครื่องมือจะถูกเตรียมทุกอย่างไว้ให้ครบเรียบร้อย โดยส่วนมากเป็นการประกอบเครื่องมือหลายตัวเข้าด้วยกันเพื่อสร้าง service discovery เครื่องมือที่อยู่ในกลุ่มนี้มีดังนี้ SmartStack (Igor & Martin, 2013) และ Eureka (Netflix, 2016) SmartStack เป็น service discovery ที่สร้างด้วยบริษัท Airbnb เป็นการผสมผสานกันระหว่าง Nerve (Airbnb, 2016), Synapse (Airbnb, 2016) และ Zookeeper โดย Zookeeper จะทำหน้าที่เป็น service registry ของระบบและ Nerve จะทำหน้าที่เป็นส่วนให้ client ติดต่อกับ service discovery ผ่านทาง http API ในขณะที่ synapse ทำหน้าที่เป็น load balance เพื่อเพิ่มประสิทธิภาพให้กับระบบ ข้อดีของ SmartStack คือ client ไม่จำเป็นต้องใช้ library ของ service discovery แต่จะติดต่อผ่านทาง sidekick process ของ Nerve ซึ่งเป็น process ที่ทำงานอยู่บนเครื่องเดียวกับ client ทำให้ client ไม่ต้องผูกติดกับ service discovery

Eureka ถูกสร้างด้วยบริษัท Netflix Eureka ถูกสร้างจาก Java เป็น service discovery ที่ทำงานอยู่บน AWS cloud ซึ่งมีส่วนประกอบอยู่ 2 ส่วนคือ Eureka server และ Eureka client โดย Eureka server ทำหน้าที่เป็น service registration โดย server สามารถสร้างเป็น replication set ได้เพื่อเพิ่ม Availability ให้กับระบบ Eureka client ทำหน้าที่ช่วย service ติดต่อกับ service เป้าหมาย การทำงานของ Eureka จะเน้นไปที่ availability ของระบบมากกว่า consistency ของข้อมูล อีกทั้งยังสามารถทำงานต่อได้เมื่อระบบเกิด partition และจะฟื้นฟูระบบกลับคืนมาเมื่อระบบกลับมาเป็นปกติ

2.2.2 Decentralized service discovery

เป็น service discovery แบบกระจาย แต่ละ server สามารถทำงานทดแทนกันได้หมด ทำให้ระบบไม่มีจุดเชื่อมต่อ (centralized server) โดยเครื่องมือที่อยู่ในกลุ่มนี้คือ Serf (HashiCorp, 2016) ทำหน้าที่จัดการบริหารสมาชิกและตรวจหาสมาชิกที่ล้มเหลว ซึ่งอาศัย SWIM protocol ที่พัฒนามาจาก Gossip protocol ในการทำ failure detection โดย Serf จะทำงานใน

รูปแบบ sidekick process ที่ทำงานอยู่บนเครื่องเดียวกันกับ service ที่ต้องการ service discovery โดยการติดต่อกับ Serf จะติดต่อผ่านทาง library หรือทาง RPC protocol

ในงานวิจัยชิ้นนี้ต้องการ service discovery ที่มี availability ที่สูงเพื่อให้ระบบ microservices มี availability ที่สูงตามเพราะการล่มของระบบอาจจะส่งผลต่อธุรกิจของบริษัท แต่ยอมให้ข้อมูลของ service อยู่ในรูปแบบ eventually consistency คือข้อมูลที่ได้รับอาจไม่ตรงกันแค่ช่วงเวลาระยะสั้นระยะหนึ่ง แต่สุดท้ายจะเกิดการเชื่อมต่อข้อมูลให้ตรงกัน เพราะความไม่เท่ากันของข้อมูลจะเพียงทำให้ service ไม่สามารถติดต่อได้ชั่วขณะเป็นช่วงระยะเวลาสั้น ซึ่งสามารถยอมรับได้ ดังนั้น Serf จึงถูกเลือกขึ้นมาใช้ในงานวิจัยชิ้นนี้ เพราะ Serf เป็น service discovery ในรูปแบบ decentralized ทำให้ไม่มี single point of failure ดังนั้น availability จึงสูงกว่าแบบ centralized และ data consistency ของ Serf เป็นแบบ eventually consistency ซึ่งตรงกับที่งานวิจัยชิ้นนี้ต้องการ โดย Serf ใช้ SWIM protocol ช่วยในการตรวจสอบหาส่วนที่ล้มเหลวของระบบซึ่งจะถูกอธิบายในหัวข้อถัดไป

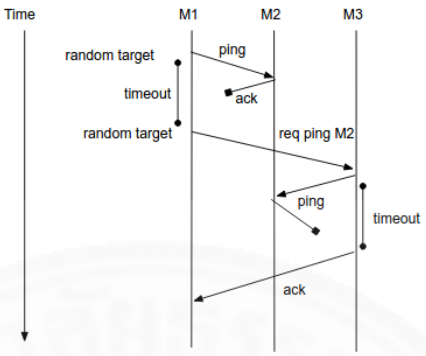
2.3 SWIM Protocol

SWIM (Scalable Weakly-consistent infection-style Process Group Membership) (Das & Gupta & Motivala, 2002) คือ protocol ที่ใช้ในการบริหารจัดการสมาชิกของกลุ่ม process และทำ failure detection ในแบบกระจาย ไม่มีจุดกลางที่ทำหน้าที่เป็นจุดเชื่อมต่อ SWIM protocol นั้นถูกพัฒนาต่อมาจาก gossip protocol ซึ่งอาศัยการกระจายข่าวในแบบสังคมของมนุษย์ คือ กระจายจากปากต่อปากไปเรื่อยๆ SWIM protocol ประกอบไปด้วยสองส่วนประกอบหลักคือ failure detector ใช้ในการตรวจหาสมาชิกที่หยุดทำงาน และ dissemination คือการกระจายข้อมูลออกไปยังสมาชิกในกลุ่ม การบริหารสมาชิกในกลุ่มนั้น สามารถทำโดยการให้สมาชิกทั้งหมดถือรายการข้อมูลสมาชิกของกลุ่ม เมื่อมีการตรวจพบการหยุดทำงาน จาก failure detector หรือการเพิ่มสมาชิกหรือการขอออกจากกลุ่ม ระบบก็จะส่งข้อมูลด้วย dissemination ไปยังทุกสมาชิก เพื่อเปลี่ยนแปลงรายการที่สมาชิกเก็บอยู่ ทำให้ทุกสมาชิกมีข้อมูลที่ถูกต้องตรงกันทั้งกลุ่ม โดยรายละเอียดของส่วนประกอบต่างๆ จะอธิบายดังต่อไปนี้

2.3.1 SWIM Failure detector

SWIM หาสมาชิกที่หยุดทำงานด้วยการให้สมาชิกทั้งหมดส่ง ping ไปหาสมาชิกอื่น M^i ทุกช่วงเวลา T' โดยเลือกจากการสุ่ม ถ้าไม่มีการตอบกลับมาในระยะเวลาที่กำหนดก็จะส่งคำสั่งไปหาสมาชิกใหม่ M^k ที่มาจากการสุ่มเช่นเดียวกัน เพื่อให้ส่ง ping ไปหาสมาชิก M^i โดยถ้าไม่ได้รับการ

ตอบกลับจากสมาชิก M^k เช่นกันก็จะกระจายข้อมูลว่ามีสมาชิกที่หยุดทำงานด้วย แต่ถ้าได้รับการตอบกลับก็จะส่งผลลัพธ์ไปยัง M^i ว่า M^k ยังทำงานอยู่ปกติ

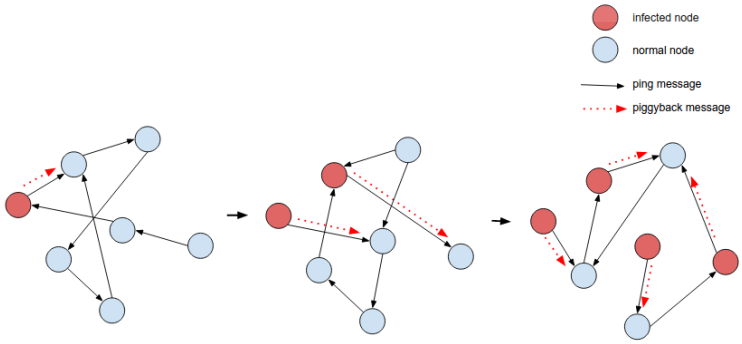


รูปที่ 2.4 วิธีการทำงานของ SWIM Failure detector

2.3.2 Dissemination

Dissemination เป็นการสื่อสารข้อมูลระหว่างสมาชิกในกลุ่ม มีการสื่อสารอยู่สองชนิดคือ multicast และ piggyback การสื่อสารข้อมูลที่เกี่ยวกับการมีสมาชิกใหม่เพิ่มเข้ามาหรือมีสมาชิกเก่าออกไปแล้วนั้น SWIM จะใช้ multicast เป็นตัวกระจายข้อมูลให้ทุกสมาชิกได้รับรู้เพื่อลบหรือเพิ่มสมาชิกจากรายการ แต่การกระจายแบบ multicast นั้น จะทำให้ข้อมูลวิ่งอยู่บนระบบเครือข่ายมากเกินไป ทำให้ SWIM พัฒนาวิธีการที่เรียกว่า Infection style dissemination ขึ้นมาเพื่อลดข้อมูลที่วิ่งอยู่บน network

Infection style dissemination หรือ piggyback message เป็นการอาศัย ping ที่ปกติจะต้องส่งไปหาสมาชิกในทุกช่วงเวลา T' อยู่แล้ว ช่วยกระจายข้อมูลให้ โดยแนบข้อมูลไปกับ ping เมื่อสมาชิกได้รับข้อมูลนี้ช่วงเวลาที่ต้องส่ง ping ก็จะแนบข้อมูลชุดนี้ไปกับ ping ด้วย ทำให้ได้มาซึ่งวิธีการ infection style ทำให้ข้อมูลที่วิ่งอยู่บน network ไม่เยอะมากจนเกินไปเหมือน multicast แต่ข้อมูลจะค่อยๆกระจายตัวออกไปเหมือนไวรัสและในที่สุดทุกสมาชิกจะได้รับข้อมูลเหมือนกันหมด



รูปที่ 2.5 การทำงานของ piggyback message

2.3.3 พัฒนาความถูกต้องของ SWIM Failure detector

การตรวจจับสมาชิกว่าหยุดทำงาน อาจเกิดข้อผิดพลาดของการตรวจจับได้เช่น กรณีที่สมาชิกที่ไม่ตอบสนอง ping อาจเป็นเพราะมีปัญหาเกิดขึ้นชั่วคราวหรือข้อมูลที่ถูกส่งออกไปนั้นหายกลางทางจากปัญหาของเครือข่าย ซึ่งทำให้สมาชิกนั้นต้องออกจากระบบไปเพราะการตรวจจับที่ผิดพลาด ดังนั้น SWIM ลดปัญหานี้โดย ระบุในรายการสมาชิกที่ไม่ตอบ ping ในครั้งแรกนั้นให้ถือว่าเป็นรายการน่าสงสัย แล้วกระจายข้อมูลนี้ออกไปยังทุกสมาชิก สมาชิกใดๆได้รับข้อมูลนี้แล้วจะบันทึกไว้ในรายการเช่นกัน เมื่อต้อง ping ไปยังสมาชิกที่ถูกสงสัยว่าหยุดทำงานแล้วจริง และไม่ได้รับการตอบกลับ จึงตัดสินว่าสมาชิกนั้นได้หยุดทำงานแล้ว จากนั้นจะกระจายข้อมูลให้สมาชิกอื่นๆได้รับรู้ แต่ถ้าได้รับการตอบกลับจากสมาชิกที่ถูกสงสัยก็จะลบออกจากรายการน่าสงสัย และกระจายข้อมูลให้ทุกสมาชิกได้รับรู้เช่นเดียวกัน วิธีการนี้จะช่วยลดปัญหาของการตรวจจับผิดพลาดได้อย่างมาก ช่วยทำให้ SWIM มีความถูกต้องที่มากขึ้น

2.4 Message bus

ในระบบตั้งแต่ขนาดกลางมักจะประกอบไปด้วยโปรแกรมมากกว่าหนึ่งโปรแกรมทำงานอยู่ข้างใน message bus เป็นเครื่องมือหนึ่งที่ใช้ในการสื่อสารกันระหว่างโปรแกรมในระบบ (Gregor & Boboy, 2003) โดยจะสื่อสารกันผ่านทาง message bus แทนที่จะติดต่อสื่อสารกันโดยตรง ซึ่ง message bus นั้นเหมือนระบบ bus ใน mainboard ซึ่งจะช่วยในเรื่องของการผสานกันของส่วนประกอบที่มีความต่างกันในเรื่องของเทคโนโลยีที่ใช้ในระบบและสิ่งแวดล้อมของระบบ ด้วยการกำหนดรูปแบบการติดต่อกลาง ที่ทุกส่วนประกอบจะต้องใช้เมื่อต้องการจะติดต่อสื่อสารกันผ่านทาง message bus การใช้ message bus ซึ่งมีรูปแบบกลางในการสื่อสารมีประโยชน์ดังต่อไปนี้

1. ลดการผูกติดกันของโปรแกรมในระบบ โดยโปรแกรมจะสื่อสารกันผ่านทาง message bus แทนที่จะสื่อสารการโดยตรงที่ต้องรู้ที่อยู่และรูปแบบการติดต่อสื่อสารของแต่ละฝ่าย
2. ลดจำนวน connection ที่จะเกิดขึ้น ยกตัวอย่างเช่นในระบบที่มีโปรแกรมอยู่ทั้งหมด 10 โปรแกรม ถ้าโปรแกรมทั้ง 10 โปรแกรมต้องสามารถติดต่อสื่อสารกันได้หมดจะทำให้เกิด 100 connection ในระบบ แต่ถ้าผ่านทาง message bus โปรแกรมจะสร้าง connection เพียงแค่ 1 connection กับ message bus เท่านั้น โดยการลดจำนวน connection นี้จะช่วยลดความยุ่งยากในการรักษาระบบและช่วยทำให้ระบบสามารถแก้ไขได้ง่ายขึ้นอีกด้วย

3. ช่วยเพิ่มความยืดหยุ่นในการเปลี่ยนแปลงหรือแก้ไขส่วนประกอบในระบบ เพราะในการติดต่อสื่อสารมีรูปแบบกลางในการติดต่อทำให้สามารถเพิ่มและลดส่วนประกอบในระบบเป็นไปได้ง่ายขึ้น

2.5 Service-oriented architecture

Service-oriented architecture (SOA) เป็น architecture style ที่ใช้แนวคิดในการสร้างระบบจากการประกอบกันของหลาย service โดย service ถูกมองเป็นฟังก์ชั่นงานทางธุรกิจที่มีความเป็นอิสระ ไม่ขึ้นกับ service อื่นและอาจจะเกิดจากการประกอบกันของหลาย service โดย protocol หลักที่ SOA ใช้ในการติดต่อสื่อสารกันระหว่าง service คือ SOAP ซึ่งเป็น protocol ที่ใช้ XML เป็นรูปแบบข้อมูลที่ใช้ในการสื่อสาร โดย SOA ถูกออกแบบอยู่บนองค์ประกอบต่อไปนี้

- Service loose coupling คือทุก service ต้องไม่มีการผูกติดกัน
- Service reusability คือ service ต้องสามารถถูกเรียกใช้ได้จากทุก service
- Service abstraction คือ service ต้องซ้อนรูปแบบการทำงานของ service แสดงเฉพาะรูปแบบการติดต่อเท่านั้น
- Standardized service contract คือทุก service ต้องใช้รูปแบบการสื่อสารเดียวกัน รูปแบบการสื่อสารจะถูกตกลงล่วงหน้า
- Service statelessness คือ service ต้องเก็บข้อมูลให้น้อยที่สุด
- Service discoverability คือ service ต้องสามารถถูกค้นพบได้ง่าย โดยการติดต่อ service มากจะต้องอาศัยเครื่อง service discovery ในการติดต่อ
- Service granularity คือการทำงานของ service ต้องจบในตัว service เอง
- Service composability คือ service สามารถเกิดจากการประกอบกันของหลาย service ได้โดยไม่สนใจความซับซ้อนของการประกอบกันของ service

เพื่อให้ SOA มีองค์ประกอบดังที่กล่าวไปแล้ว จึงต้องการส่วนประกอบหลายอย่างในการส่งระบบ ยกตัวอย่างเช่น

- Service discovery เป็นส่วนที่ทำหน้าที่บริหารการติดต่อ service ในการลงทะเบียน service เพื่อทำให้ระบบสามารถค้นหา service ได้ง่าย
- Messaging ใช้ในส่งข้อมูลเพื่อทำการสื่อสารกันระหว่าง service โดยมีรูปแบบการสื่อสารกลาง

- Service Orchestration ใช้ในการควบคุมการ business processes ของระบบ

ใน SOA ระบบการสื่อสารเป็นหนึ่งในส่วนประกอบที่สำคัญในระบบ ทำให้ระบบการสื่อสารนั้นถูกออกแบบมาเพื่อรองรับรูปแบบการสื่อสารหลายรูปแบบ โดยระบบการสื่อสารจะถูกแบ่งออกเป็นส่วนประกอบดังต่อไปนี้

- Message type pattern คือรูปแบบของข้อมูลที่ใช้ในการส่งยกตัวอย่างเช่น command, document, request/reply
- Message channel pattern คือช่องทางที่ใช้ในการสื่อสารยกตัวอย่างเช่น point-to-point, publish-subscribe, message bus
- Routing pattern คือรูปแบบการในการขนส่งข้อมูลจาก service ต้นทางไปยัง service ปลายทาง ยกตัวอย่างเช่น pipe and filter, content aggregator
- Service consumer pattern คือรูปแบบพฤติกรรมของ client ของ service ยกตัวอย่างเช่น polling consumer, event-driven, message façade
- Message construction pattern คือรูปแบบในการสร้างข้อมูลที่ใช้ในการสื่อสารในระบบ ตัวอย่างเช่น message sequence, message expiration
- Transformation pattern คือรูปแบบการเปลี่ยนแปลงรูปร่างของข้อมูล ยกตัวอย่างเช่น envelop wrapper, content filter

ดังจะเห็นได้จากที่กล่าวไปแล้ว SOA เป็น architecture ที่ค่อนข้างมีรายละเอียดที่ซับซ้อนและมีส่วนประกอบจำนวนมากทำให้อาจไม่เหมาะกับระบบบางประเภท ยกตัวอย่างเช่น ระบบที่ต้องการการเปลี่ยนแปลงที่บ่อย หรือระบบขนาดเล็กถึงขนาดกลางที่ไม่ต้องการเครื่องมือในระดับ SOA ซึ่งอาจเป็นการเพิ่มภาระให้ระบบโดยไม่จำเป็น

2.6 RESTful

เป็น software architecture style ที่อยู่ในแบบ network-base โดยเกือบทั้งหมดใช้ HTTP เป็น protocol หลักในการสื่อสาร ซึ่ง RESTful (Fielding, 2000) จะไม่กำหนดวิธีการสร้างของส่วนประกอบในระบบ วิธีการติดต่อและรูปแบบข้อมูลในระบบ แต่จะกำหนดขอบเขตและข้อจำกัดแทน โดยขอบเขตและข้อจำกัดมีดังต่อไปนี้

- Client-server คือ client และ server จะต้องทำหน้าที่ต่างกันและไม่ผูกติดกัน ยกตัวอย่างเช่น client จะไม่เก็บข้อมูล แต่จะปล่อยให้เป็นหน้าที่ของ server

- Stateless คือทุกการร้องขอของ client จะต้องไม่มีข้อมูลของ client ถูกเก็บไว้ใน server ข้อมูลจะถูกส่งไปเท่าที่จำเป็นสำหรับการร้องขอเท่านั้น
- Cacheable ข้อมูลที่ได้จากการร้องขอจะต้องสามารถนำมาใช้ใหม่ได้
- Layered system client จะต้องไม่สามารถระบุได้ว่าการร้องขอเป็นการติดต่อกับ server โดยตรงหรือไม่ โดยคำร้องขอนั้นอาจจะผ่านระบบ load balance หรือระบบอื่นก่อนก็ได้
- Uniform interface รูปแบบการเข้าถึงทรัพยากรใน server จะต้องมีรูปแบบเดียวกัน โดยมีข้อกำหนดดังต่อไปนี้
    - Identification of resources รูปแบบการเข้าถึงข้อมูลในระบบจะต้องแตกต่าง ยกตัวอย่างเช่น ระบบที่ใช้ URL ในการเข้าถึง ทรัพยากรของระบบ URL ที่ใช้ในการเข้าถึงทรัพยากรที่ต่างกันจะต้องมีรูปแบบ URL ที่ต่างกัน
    - Manipulation of resource through representations รูปแบบที่ใช้ในการจัดการกับทรัพยากรในระบบจะทำผ่าน metadata ยกตัวอย่างเช่นใน HTTP จะใช้ HTTP method ในการระบุรูปแบบของการจัดการทรัพยากรในระบบ
    - Self-descriptive message ทุกข้อความจะต้องมีข้อมูลที่อธิบายรูปแบบของข้อความเพียงพอต่อการใช้งาน
    - HATEOAS ในการติดต่อสื่อสารกับ server ของ client จะใช้รูปแบบ hypermedia

2.7 งานวิจัยที่เกี่ยวข้อง

งานวิจัยที่เกี่ยวข้องกับ Microservices นั้น ยังมีจำนวนน้อย เพราะ Microservices เพิ่งเป็นที่รู้จักเมื่อไม่นานมานี้ งานวิจัยที่ค้นพบมีดังนี้ Serfnode (Stubbs, 2015) , Self-managing microservices (Toffetti & Brunner & Blochlinger & Dudouet & Edmonds, 2015) และ Iris (Szilagyi, 2014) ทุกงานวิจัยมีเป้าหมายในการแก้ไขปัญหาของ microservices โดยสามารถสรุปตาม feature ทั้งหมดที่ทุกงานวิจัยพัฒนาได้ดังต่อไปนี้ Service discovery, Load balance, Service orchestration และ Auto recovery

2.7.1 Service discovery

Service discovery เป็นปัญหาที่ทุกงานต้องการการแก้ไข เพราะเป็นปัญหาสำคัญที่จะทำให้การเปลี่ยนระบบเป็น automation นั้นประสบผลสำเร็จ เมื่อสามารถเปลี่ยนระบบให้เป็น automation ได้ปัญหาเรื่องความยุ่งยากในการจัดการของ microservices จะลดลงเป็นอย่างมาก

โดย service discovery tool ที่ถูกนำมาใช้งานในทุกงานวิจัยจะมีอยู่สองรูปแบบคือ แบบมีจุดเชื่อมต่อ (centralize) และแบบไม่มีจุดเชื่อมต่อ (decentralize)

2.7.1.1 Centralization

Self-managing microservices เป็นงานวิจัยที่ใช้รูปแบบจุดเชื่อมต่อ (Centralized server) โดยใช้ Etcd เป็นเครื่องมือในการเก็บข้อมูลให้อยู่ในรูปแบบ key-value สามารถเก็บไว้บนหลาย server ได้ โดยที่ consistency ของข้อมูลยังคงอยู่ แต่ Etcd นั้นไม่ได้สร้างมาเป็นเครื่องมือทำ service discovery โดยเฉพาะ เพราะฉะนั้นจึงยังมีส่วนที่ต้องทำเพิ่มอีก ข้อดีของ centralize service discovery ที่เห็นได้ชัดคือเรื่อง consistency ของข้อมูล ข้อเสียคือ single point of failure

2.7.1.2 Decentralization

งานวิจัยที่ใช้รูปแบบไม่มีจุดเชื่อมต่อนั้น มีงานวิจัย Serfnode และ Iris โดยงานวิจัย Serfnode ใช้ Serf ซึ่งเป็นเครื่องมือ maintain membership and failure detection ที่มีจุดแข็งในด้าน availability ที่สูงมากโดยแลกกับ consistency ของข้อมูลที่เป็นแบบ eventually consistency คือข้อมูลจะเกิด consistency แต่ต้องอาศัยระยะเวลา

งานวิจัย Iris ไม่ได้เป็นงานวิจัยที่เกี่ยวข้องกับการทำ service discovery โดยตรง แต่เนื่องจากพัฒนาจาก Pastry และ Scribe ซึ่งเป็น Dynamic hash table หรือ DHT สามารถส่งข้อมูลไปยัง node ที่ไม่รู้ที่อยู่ได้ จึงสามารถปรับใช้กับงานในเชิง service discovery ได้ ถึงแม้จะขาดคุณสมบัติบางประการ ตัวอย่างเช่น การตรวจหา service ล้มเหลว เป็นต้น

DHT คือการส่งข้อมูลอยู่ในรูปแบบ key-value ซึ่ง key คือผลลัพธ์ของการ hash ค่าที่อยู่ของ node เป้าหมาย ส่วน value คือข้อมูลที่ต้องการส่ง โดยจะมีการกำหนด keyspace ซึ่งเป็นช่วงค่าที่เป็นไปได้ทั้งหมดของ key เพื่อใช้ในการกำหนด node ที่ข้อมูลจะถูกส่งไป ด้วยการกำหนดช่วงของ keyspace ให้กับทุก node เมื่อต้องการส่งข้อมูล DHT จะ hash ค่าที่อยู่ของ node เป้าหมายด้วย consistent hashing เพื่อให้ได้ key ที่จะใช้ในการส่งข้อมูล จากนั้นจะเลือก node ที่ใกล้เคียงที่สุด โดยดูจากช่วง key ของ node เพื่อนบ้านที่ถูกเก็บไว้ในทุก node ว่าช่วง key ใน node ไหนใกล้เคียงกับ key ของข้อมูลที่จะส่งมากที่สุด จากนั้นจะส่งข้อมูลไปยัง node นั้น ถ้า node ที่ได้รับข้อมูลมีช่วงของ key ไม่ตรงกับ key node จะหา node ที่ใกล้เคียงใหม่ และส่งข้อมูลต่อไป ทำวนซ้ำจนกระทั่งเจอ node ที่มีช่วง key ตรงกับ key ของข้อมูล

2.7.2 Load balance

Load balance เป็นเรื่องสำคัญของการใช้ microservices เนื่องจาก load balance สามารถเพิ่มทั้ง Availability และ Scalability ของระบบได้ แต่ microservices นั้น แบ่งระบบออกเป็นหลาย service ทำให้การทำ load balance ยากขึ้นเนื่องจากต้องทำให้ทุก service ซึ่ง

เป็นงานที่ใช้ทั้งคนและเวลา ดังนั้นการมีคุณสมบัติของ load balance ใน microservices จึงเป็นเรื่องสำคัญ ดังจะเห็นจากทุกงานวิจัยมีการทำ load balance ทั้งหมด โดยใช้วิธีการที่ใกล้เคียงกันคือการตั้งกลุ่มของ service ที่เหมือนกันให้เป็นหน่วยที่เล็กที่สุดที่ service สามารถเลือกใช้ส่งข้อมูลได้ จากนั้นระบบจะทำการเลือกว่าควรเป็น service ไหนที่ได้รับข้อมูล

2.7.3 Service orchestration

งานวิจัยเดียวที่เลือกพัฒนา service orchestration คือ Self-managing microservices โดยให้นักพัฒนาประกาศโครงสร้างของระบบผ่านทาง Etcd จากนั้น ระบบจะสร้างระบบตามโครงสร้างที่นักพัฒนาประกาศไว้ โดยระบบจะต้องทำงานอยู่บนเครื่องมือ virtualization และระบบจะสร้างข้อมูลของ node ที่เกิดขึ้นจริงในระบบเพื่อใช้การในการทำ self-healing ต่อไป การที่ service orchestration มีงานวิจัยทำแค่งานเดียวอาจเป็นเพราะความซับซ้อนของการทำ service orchestration ที่อาจทำให้ระบบมีขนาดใหญ่และยุ่งยากในการจัดการ จึงไม่สอดคล้องกับลักษณะการใช้ microservices

2.7.4 Communication

Communication มีเพียงงานวิจัย Iris ที่เลือกแก้ไขปัญหานี้ด้วยการพัฒนา message bus โดยรูปแบบการส่งข้อมูลใน Iris นั้น มีสามรูปแบบคือ request/response, publish/subscribe และ broadcast โดยใช้ Pasty ซึ่งเป็น peer-to-peer protocol สำหรับกลุ่มการสื่อสารขนาดใหญ่ ยกตัวอย่างเช่น WAN หรือ Internet Iris ใช้วิธีการส่งต่อข้อมูลผ่าน node ใกล้เคียงจนกระทั่งถึง node เป้าหมาย ทำให้การส่งข้อมูลไม่จำเป็นต้องรู้จักข้อมูลของทุก node ส่งผลให้การยึดติดกันระหว่าง node ลดลง

2.7.5 Auto recovery

Auto recovery ช่วยให้ Availability ของระบบสูงขึ้น เนื่องจากสามารถฟื้นฟูตัวเองได้ โดยวิธีการที่นิยมใช้คือ monitor ระบบค้นหา service ที่หยุดทำงาน เมื่อเจอจะสร้าง service ใหม่ขึ้นมาแทนที่ service ที่หยุดทำงานไปแล้วให้เพิ่มเข้ามาในระบบ เนื่องจากเป็นวิธีที่ทำง่ายแต่วิธีนี้เหมาะสมกับระบบที่ deploy เป็น cloud computing เท่านั้น โดยมีสองงานวิจัยที่ใช้แนวทางนี้ คือ Serfnode ซึ่งใช้รูปแบบ parent and child node ทำให้ parent สามารถตรวจได้ว่า child node ยังทำงานอยู่หรือไม่ ถ้าไม่ทำงานแล้วจะสร้างขึ้นมาใหม่ โดยที่ทุก node สามารถทำหน้าที่ parent ได้หมดจึงไม่มี single point of failure และอีกงานวิจัยหนึ่ง Self-managing microservices ซึ่งในงานวิจัยไม่ได้ให้รายละเอียดในการทำ Auto recovery

ตารางที่ 2.1 เปรียบเทียบคุณสมบัติของงานวิจัยที่เกี่ยวข้องตาม feature ในหัวข้อ 2.7

	Serfnode	Self-managing microservice	Iris	Our model
Service discovery	√	√	√	√
Load balance	√	√	√	√
Service orchestration	x	√	x	x
Communication	x	x	√	√
Auto recovery	√	√	x	x

ตารางที่ 2.1 เป็นตารางที่สรุปและเปรียบเทียบคุณสมบัติของงานวิจัยชิ้นนี้กับงานวิจัยที่ใกล้เคียง โดยการตัดสินใจเลือกคุณสมบัติของงานวิจัยชิ้นนี้ มาจากการที่ communication ของ microservices นั้น ควรที่จะ lightweight และความซับซ้อนต่ำ ทำให้คุณสมบัติอย่างเช่น service orchestration และ auto recovery ที่มักจะเพิ่มความซับซ้อนของระบบมากขึ้นนั้นถูกตัดออก ยกตัวอย่างเช่น service orchestration จะทำให้ระบบผูกติดกับรูปแบบที่มักจะถูกกำหนดขึ้นเพื่อให้ระบบเกิดการสร้างแบบอัตโนมัติได้ ทำให้ระบบที่มีความต้องการการเปลี่ยนแปลงบ่อยจะไม่เหมาะ และมีการทำงานที่ไม่จำเป็นเกิดขึ้น ในขณะที่ auto recovery นั้น เป็นที่ต้องการในกรณีของระบบที่ต้องการ availability ที่สูงมาก ซึ่ง load balance มีการเพิ่ม availability ในระดับหนึ่งอยู่แล้ว ดังนั้นในระบบทั่วไปแล้วจึงมักจะไม่มีความจำเป็นที่จะต้องพัฒนา auto recovery เพราะจะทำให้ระบบมี overhead ที่สูงขึ้น ดังนั้นในงานวิจัยชิ้นนี้จะเน้นไปที่การทำ service discovery, load balance และ communication

จากตารางที่ 2.1 จะเห็นได้ว่างานวิจัยนี้มีลักษณะที่คล้ายกับ Iris ทั้งในด้านที่เป็น decentralized message bus เหมือนกันและมีคุณสมบัติอื่นที่เหมือนกัน แต่ก็ยังมีข้อแตกต่างดังต่อไปนี้ ระบบการสื่อสารของงานวิจัยนี้ ทำการแบ่งชั้นของการติดต่อสื่อสารและ service discovery ออกจากกันทำให้การสื่อสารจะเป็นแบบ single network communication แต่ในขณะที่ Iris จะรวมการสื่อสารและการทำ service discovery เข้าด้วยกัน โดย protocol DHT ที่ Iris เลือกใช้ ทำให้เกิดจำนวน network communication ในหนึ่งการสื่อสารเท่ากับ log(n) โดย n เป็นจำนวน node ซึ่งจำนวน network communication ที่สูงจะส่งผลกระทบทำให้ประสิทธิภาพในการส่งข้อมูลลดลง และข้อแตกต่างอีกข้อหนึ่งคือ protocol ที่ Iris เลือกใช้ในการทำ service discovery

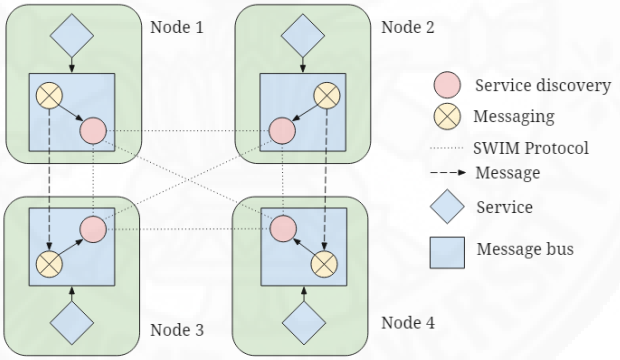
ไม่สามารถตรวจหา node ที่ล้มเหลวได้ ทำได้เพียงค้นหา node เป้าหมายเท่านั้น แต่ในขณะที่ SWIM protocol ที่ใช้ในการทำ service discovery ที่ใช้ในงานวิจัยชิ้นนี้ สามารถทำการตรวจหา node ที่ล้มเหลวได้ จึงเหมาะกับการนำมาใช้เป็นเครื่องมือในการทำ microservices มากกว่าเพราะลักษณะหนึ่งของ microservices คือต้องการเครื่องมือที่ใช้ในการ monitor หา service ที่ล้มเหลวได้ ซึ่ง SWIM protocol สามารถรองรับความต้องการนี้ได้โดยที่ไม่ต้องอาศัยเครื่องภายนอกเพิ่มเติม

บทที่ 3
วิธีการวิจัย

3.1 การออกแบบ

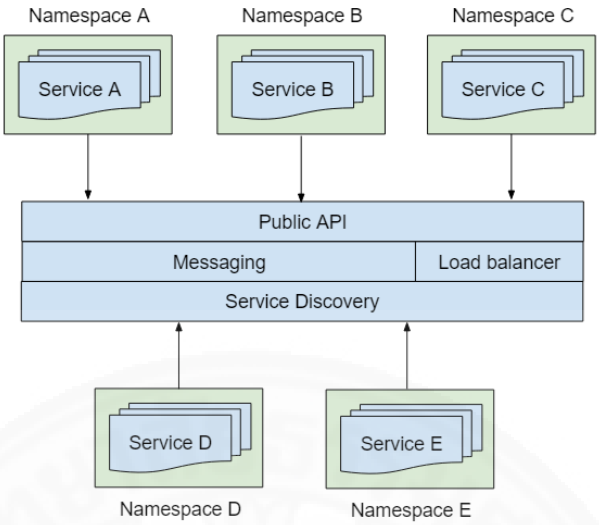
จากปัญหาของ microservices ในเรื่องการสื่อสารระหว่าง service งานวิจัยชิ้นนี้จึงออกแบบ message bus เพื่อใช้เป็นเครื่องมือติดต่อสื่อสารระหว่าง service ซึ่งช่วยแก้ปัญหาการผูกติดกันของ service เพราะ message bus เปลี่ยนการติดต่อสื่อสารของ service จาก point-to-point มาเป็นแบบมาติดต่อผ่าน message bus ซึ่งไม่ระบุเป้าหมาย ในขณะเดียวกัน message bus ยังถูกออกแบบให้มีคุณสมบัติเพิ่มเติมเพื่อให้เหมาะกับการใช้ใน microservices architecture

ระบบ message bus ในงานวิจัยชิ้นนี้จะเป็นระบบกระจาย คือไม่มีจุดเชื่อมต่อของระบบ (Decentralized server) โดย message อยู่ในรูปแบบ sidekick process ที่ทำงานอยู่บนเครื่องเดียวกันกับ service ดังรูป 3.3 คุณสมบัตินี้ทำให้ message bus ไม่มี single point of failure เหมือนใน centralized message bus ที่นิยมใช้ใน microservices จึงทำให้ระบบมี availability ที่สูงขึ้น



รูปที่ 3.1 รูปแบบการทำงานของ message bus

การติดต่อสื่อสารระหว่าง message bus แบ่งออกเป็น 2 ช่องทางคือ messaging และ failure detection ดังจะเห็นได้จากรูปที่ 3.1 โดย messaging จะทำหน้าที่ส่งข้อมูลที่ service ต้องการส่งไปยัง service เป้าหมาย ในขณะที่ failure detection เป็นช่องทางที่ใช้ส่งข้อมูลเพื่อทำการตรวจหา service ที่ล้มเหลว ของ message bus การแยกช่องทางการส่งข้อมูลและการตรวจหา service ที่ล้มเหลวออกจากกัน จะทำให้ง่ายต่อการจัดการ และ protocol ที่ใช้ในการติดต่อสื่อสารจะมีความเรียบง่ายเนื่องจาก ความชัดเจนและขนาดของหน้าที่ของ protocol ที่ลดลง นอกเหนือจากนี้ การแยก messaging ออกจาก failure detection ยังทำให้การส่ง message อยู่ในรูปแบบ point-to-point ซึ่งมีประสิทธิภาพมากกว่าการผ่านตัวกลาง โดยส่วนประกอบที่สำคัญของ message bus เป็นไปดังรูปที่ 3.2 โดยมีรายละเอียดดังต่อไปนี้



รูปที่ 3.2 ส่วนประกอบใน message bus

3.1.1 Public API

Public API เป็นส่วนประกอบที่สร้างและทำงานเป็น interface ที่อนุญาตให้ระบบภายนอกหรือ service ใช้ในการติดต่อกับระบบ message bus โดย HTTP ถูกเลือกขึ้นมาเป็น protocol ที่ใช้ในการติดต่อสื่อสารกับ Public API สาเหตุในการเลือก HTTP เป็น protocol หลักที่ใช้ในการติดต่อสื่อสาร เพราะ HTTP เป็นที่รู้จักอย่างกว้างขวางและมีอยู่ในทุกระบบและเครื่องมือที่เป็นที่นิยม ซึ่งจะทำให้ Public API สามารถเข้าถึงได้จากทุกระบบ โดย interface ของ public API ถูกออกแบบตาม RESTful style ซึ่งจะใช้ URL ในการระบุทรัพยากรในระบบที่ต้องการเข้าถึง และใช้ HTTP method ในการระบุลักษณะการกระทำกับทรัพยากร รูปแบบของข้อมูลที่ใช้ในการรับส่งจะเป็นคนละรูปแบบกับทรัพยากรจริงที่อยู่ในระบบ รูปแบบข้อมูลที่นิยมใช้คือ JSON, XML, HTML ในระบบนี้เลือกใช้ JSON (JavaScript Object Notation) เป็นรูปแบบการรับส่งข้อมูล

Method: HTTP method, i.e., *GET, POST, PUT, DELETE*

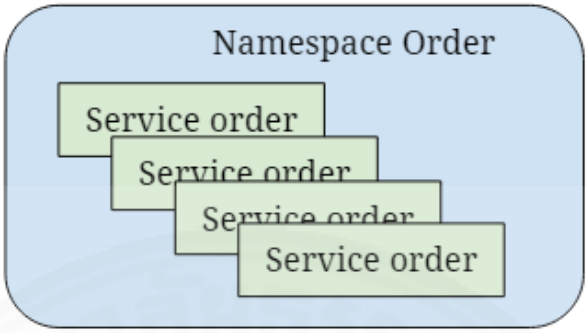
URI: http://msg_bus_addr/resources

Request body อยู่ในรูปแบบ JSON format โดยเกี่ยวเนื่องกับ ***resource_type***

รูปที่ 3.3 รูปแบบการสื่อสาร Service API

รูปแบบกลางของการรับส่งข้อมูลของ message bus เป็นไปตามรูปที่ 3.3 โดยจะเห็นได้ว่า URL ถูกแบ่งออกเป็น 2 ส่วนหลักดังต่อไปนี้ 1) msg_bus_addr คือที่อยู่ของ message bus

โดยมีค่าตั้งต้นเป็น 127.0.0.1 2) resource คือทรัพยากรในระบบ message bus ที่อนุญาตให้ระบบภายนอกติดต่อได้ โดยมี 3 ทรัพยากรดังต่อไปนี้ /namespace, /topic และ /message



รูปที่ 3.4 Namespace

/namespace เป็นทรัพยากรที่เก็บข้อมูลของ namespace ทั้งหมดในระบบ โดย namespace คือ service name ที่เก็บกลุ่มของ service ชื่อเดียวกัน ซึ่งลงทะเบียนกับ message bus ดังรูปที่ 3.4 namespace จะช่วยในเรื่องของการทำ load balance ของ service การลงทะเบียน service ใน namespace จะทำผ่าน URL path */namespace* โดย JSON data ที่ใช้ในการลงทะเบียนมีรูปแบบดังต่อไปนี้ *{ namespace: string, contactPoint: string_url }* ซึ่ง namespace คือชื่อของ namespace ที่ต้องการลงทะเบียน และ contactPoint คือ URL ที่ message bus ใช้ในการติดต่อกลับ service เมื่อมีข้อมูลส่งมาถึง

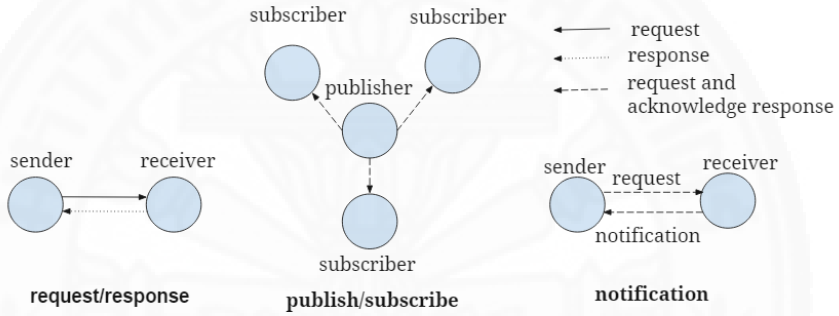
/topic เป็นทรัพยากรที่เก็บข้อมูล topic และ service ซึ่ง subscribe กับ topic ในระบบ การ subscribe topic กับ message bus จะทำผ่าน URL path */{topic_name}/subscribers* โดย topic_name เป็นชื่อของ topic ที่ service ต้องการ subscribe จะไม่มีการส่งข้อมูล JSON สำหรับการ subscribe topic

/message เป็นทรัพยากรที่ใช้ในการส่งข้อมูลกันระหว่าง service โดยรูปแบบของการส่งข้อมูลมี 3 รูปแบบด้วยกันคือ request-response (*/reqRes*), publish-subscribe (*/pubSub*) และ notification (*/noti*) รูปแบบข้อมูลที่ใช้ในการสื่อสารคือ JSON โดยรูปแบบการติดต่อสื่อสารของ message bus ถูกเลือกมาจาก รูปแบบการติดต่อสื่อสารระหว่าง service ใน microservices ที่มี 2 ลักษณะสำคัญ (Richardson, 2015) ดังต่อไปนี้ 1) รูปแบบการเชื่อมต่อของ client-server เป็นแบบ one-to-one หรือ one-to-many 2) ลักษณะการส่งข้อมูลเป็นแบบ synchronous หรือ asynchronous โดยจากลักษณะการติดต่อสื่อสารที่กล่าวมา ทำให้เกิดรูปแบบการติดต่อสื่อสารตามตารางที่ 3.1 ซึ่งเป็นรูปแบบการติดต่อสื่อสารของงานวิจัยชิ้นนี้

ตารางที่ 3.1 รูปแบบการติดต่อสื่อสารโดยแบ่งตามลักษณะการติดต่อสื่อสาร

	Synchronous	Asynchronous
One-to-one	Request/response	Notification
One-to-many	X	Publish/subscribe

จากตารางจะเห็นได้ว่าการสื่อสารแบบ one-to-many และ synchronous ไม่มีรูปการส่งข้อมูลเพราะผู้ส่งและผู้รับไม่รู้จักกัน จึงทำให้ผู้รับข้อมูลไม่สามารถที่จะส่งผลลัพธ์กลับไปยังผู้ส่งไม่ได้ เมื่อทำงานเสร็จแล้ว ทำให้รูปแบบการส่งข้อมูลที่มีลักษณะเช่นนี้เป็นไปไม่ได้ ในขณะที่รูปแบบการติดต่อสื่อสารที่เป็นไปได้ มีลักษณะของการไหลของข้อมูลตามรูปที่ 3.5



รูปที่ 3.5 รูปแบบการส่งข้อมูลในระบบการติดต่อสื่อสาร

3.1.2 Load balancer

Load balancer เป็นส่วนประกอบที่ช่วยในการกระจายข้อมูลไปยัง service โดยอาศัย namespace ซึ่งมี service ประเภทเดียวกันอยู่หลาย node หรืออีกนัยหนึ่ง service ที่เป็นชนิดเดียวกันจะอยู่ใน namespace เดียวกัน เพื่อทำให้การกระจายข้อมูลเป็นไปอย่างรวดเร็วและสามารถตอบสนองได้กับระบบหลายรูปแบบ round-robin จึงถูกเลือกมาเป็น algorithm ที่ใช้ในการกระจายข้อมูล เพราะ round-robin นั้นมีลักษณะที่ง่ายและรวดเร็ว เนื่องจากมีความซับซ้อนต่ำ

3.1.3 Messaging

Messaging เป็นส่วนประกอบที่ดูแลเรื่องการส่งข้อมูลระหว่าง service โดย protocol ที่ใช้ ในการสื่อสารคือ TCP เพราะ TCP ไม่มี overhead ในการส่งข้อมูลเมื่อเปรียบเทียบกับ HTTP protocol และรองรับการส่งข้อมูลในรูปแบบ asynchronous และ broadcast ได้ดีกว่า protocol ตัวเลือกอื่น โดยข้อมูลจะแบ่งออกเป็นสองส่วนคือ header และ body ใน header จะเก็บข้อมูลที่เป็นลักษณะของการส่งข้อมูล ยกตัวอย่างเช่น การส่งข้อมูลแบบเป็น synchronous หรือ asynchronous เป็นต้น ในขณะที่ body เก็บข้อมูลที่ service ต้องการส่งไปยัง service เป้าหมาย ในกรณีการส่งข้อมูลแบบ asynchronous เมื่อ message bus ได้รับข้อมูลจาก message bus ต้นทาง

แล้วจะตอบกลับทันทีว่าได้รับข้อมูลแล้วในขณะที่ synchronous จะรอข้อมูลตอบกลับจาก service แล้วจึงส่งข้อมูลกลับไป

3.1.4 Service discovery

Service discovery เป็นส่วนประกอบที่ทำหน้าจัดหาที่อยู่และข้อมูลของ service โดยใช้ Serf เป็นเครื่องมือในการจัดการข้อมูลของ service ซึ่ง serf ใช้ SWIM protocol เป็น protocol ในการจัดการสมาชิกแบบ decentralized ดังได้กล่าวไว้แล้วในบทที่ 2 โดย SWIM protocol นั้นเร็วกว่า DHT Protocol ที่ Iris ใช้ในแง่ของจำนวน hops ที่ต้องใช้ในการส่งข้อมูล

3.2 ขั้นตอนการทำงานของระบบ

ระบบ distributed system สถานการณ์ปรกติที่เกิดขึ้นกับระบบมีดังต่อไปนี้ เริ่มจากสร้างระบบจากนั้น service จำนวนหนึ่งจะถูกเพิ่มเข้ามาในระบบ เมื่อมี service ในระบบจะเกิดการสื่อสารระหว่าง service และทำงานไประยะหนึ่งระบบจะถูกเปลี่ยนแปลงหรือแก้ไขเพื่อเพิ่มความสามารถ ทำให้เกิดการเปลี่ยนหรือแก้ไข service ในขณะที่ระบบ distributed system มีโอกาสที่ service จะหยุดทำงานหรือ network มีปัญหา ซึ่งจะส่งผลกระทบทำให้ระบบอาจหยุดการทำงาน จากสถานการณ์ที่พบได้ทั่วไปสามารถสรุปเป็นขั้นตอนการทำงานได้ 4 ประเภทดังต่อไปนี้

1. ขั้นตอนการเพิ่ม service เข้ามาในระบบ
2. Communication ระหว่าง service
3. ขั้นตอนทำการอัพเดท service ใหม่เป็น version ใหม่
4. ขั้นตอนเมื่อ service หยุดทำงาน

3.2.1 เริ่มเพิ่ม service เข้ามาในระบบ

การสื่อสารผ่านระบบ message bus ของงานวิจัยนี้ service ที่ต้องการใช้ระบบ message bus จำเป็นต้องลงทะเบียนกับ message bus พร้อมกับระบุ namespace ที่ service ต้องการรับข้อมูล เพื่อจะสามารถสื่อสารกับ service อื่นได้ โดยวิธีการลงทะเบียนคือ service ส่ง HTTP request มายัง message bug โดยผ่าน public API ซึ่งควบคุมรูปแบบการติดต่อสื่อสารกลางระหว่าง service ดังที่ได้กล่าวแล้วในบทที่ 3.1 โดยมีรูปแบบการส่งดังรูปที่ 3.6

Method: PUT

URL: http://{message_bus_address}/namspace

Request body: {

name: {namespace}

contactPoint: {contact_point_path}

}

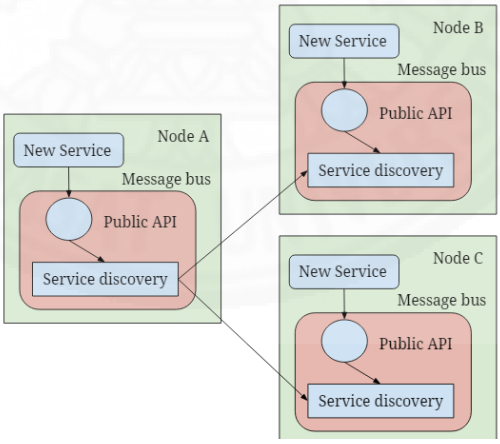
message_bus_address = URL ที่ใช้ในการติดต่อ message bus ค่าตั้งต้น คือ 127.0.0.1:21001

namespace = ชื่อของ namespace ที่ service ต้องการลงทะเบียน

contact_point_path = URL ที่ message bus จะส่งข้อมูลให้เมื่อ service ถูกเลือกจาก load balance ให้ทำการประมวลผลข้อมูล

รูปที่ 3.6 รูปแบบการลงทะเบียนบน message bus

เมื่อ service ทำการลงทะเบียนกับ message bus ผ่านทาง public API จากนั้นจะส่งให้ service discovery ซึ่งทำหน้าที่บริหาร service ทั้งหมดในระบบ ที่กล่าวในบทที่ 3.1 กระจายข้อมูลของ service ที่ลงทะเบียน ไปยัง message bus อื่นในระบบ และทำการส่งผลลัพธ์ให้กับ service โดยดูได้จากรูปที่ 3.7



รูปที่ 3.7 ขั้นตอนลงทะเบียน service

3.2.2 เกิด communication ระหว่าง service

3.2.2.1 Request/response

Request/response เป็นรูปแบบการติดต่อสื่อสารที่มีผู้ส่งและผู้รับอย่างละหนึ่งเท่านั้น โดยการส่งข้อมูลจะเป็นแบบ synchronous คือผู้ส่งจะรอให้ผู้รับข้อมูลตอบกลับมาจึงจะสามารถไปทำงานอื่นได้ ทำให้เกิดการปิดกั้นการทำงานของผู้ส่งระหว่างรอข้อมูล ตัวอย่างของ

รูปแบบการส่งชนิดนี้คือ HTTP ที่เป็นรูปแบบการสื่อสารของระบบ internet โดยการทำงานจะเกิดขึ้นสองฝั่ง ทั้งผู้รับและผู้ส่ง ฝั่งผู้รับทำงานเริ่มจาก service ที่ต้องการส่งข้อมูล จะส่ง HTTP request ไปที่ public API ของ message bus โดยมีรูปแบบการส่งดังรูปที่ 3.8

เมื่อ public API ได้รับข้อมูล จะขอรายชื่อ service ทั้งหมดใน namespace จาก service discovery เพื่อส่งข้อมูลต่อไปยัง load balance ที่ทำหน้าที่เลือก service ใน namespace ที่เหมาะสมจะรับข้อมูล หลังจากได้ service เป้าหมายจาก load balance แล้ว public API จะร้องขอไปยัง messaging module ให้ส่งข้อมูลไปยัง service เป้าหมาย เมื่อ messaging module ได้รับคำร้องขอ จะส่งข้อมูลไปยัง service เป้าหมาย และรอการตอบกลับของ service เป้าหมาย เมื่อได้รับการตอบกลับ จะส่งผลลัพธ์กลับไปยัง service ที่ส่งข้อมูล ผ่านทาง HTTP response ซึ่งถือเป็นการจบการทำงาน

Method: PUT

URL: http://{message_bus_address}/message/reqRes/{namespace}

Request body: {message}

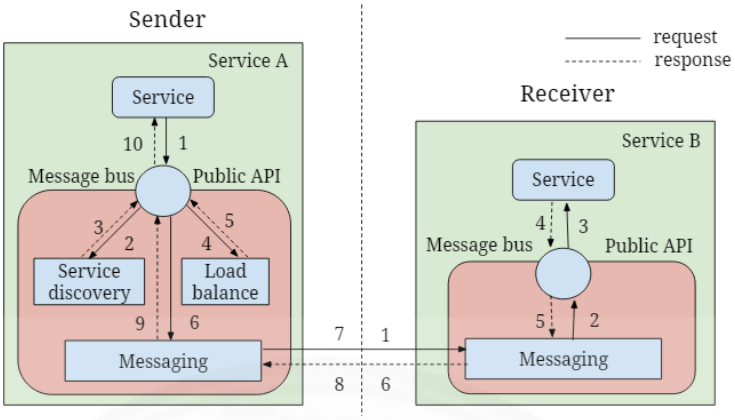
message_bus_address = URL ที่ใช้ในการติดต่อ message bus ค่าตั้งต้น คือ 127.0.0.1:21001

namespace = ชื่อของ namespace ที่ service ต้องการลงทะเบียน

message = ข้อมูลที่ต้องการส่งไปยัง service ที่อยู่ใน namespace เป้าหมาย

รูปที่ 3.8 รูปแบบการส่งข้อมูลแบบ request/response ผ่านทาง public API

ฝั่งผู้รับเมื่อ message bus ของ service เป้าหมายได้รับข้อมูลจากฝั่งผู้ส่งแล้ว จะส่งข้อมูลต่อไปยัง public API เพื่อติดต่อไปยัง service เป้าหมายตาม contactPoint ที่ได้ระบุไว้ในขั้นตอนการลงทะเบียน เมื่อ service ตอบกลับมา public API จะส่งผลลัพธ์กลับไปยัง messing module เพื่อตอบกลับฝั่งผู้ส่งข้อมูล โดยขั้นตอนการทำงานของ request/response ทั้งหมดจะเป็นไปตามรูปที่ 3.9



รูปที่ 3.9 ขั้นตอนการทำงานลอง request/response

3.2.2.2 Publish/subscribe

Publish/subscribe เป็นรูปแบบการสื่อสารที่ผู้รับสามารถมีมากกว่าหนึ่งรูปแบบการส่งข้อมูลเป็นแบบ Asynchronous การส่งข้อมูลผู้ส่งจะระบุหัวข้อ (topic) มากับข้อมูล ผู้รับที่ระบุรับข้อมูลที่มีหัวข้อตรงกับข้อมูล จะได้รับข้อมูลไปประมวลผลต่อ ด้วยวิธีนี้ผู้รับและผู้ส่งจะไม่ต้องรู้จักกัน ทำให้ไม่เกิดการผูกติดกันระหว่างผู้รับและผู้ส่ง โดยการทำงานจะแบ่งออกเป็นสองฝั่ง คือ publish และ subscribe โดยการทำงานมีรายละเอียดดังต่อไปนี้

3.2.2.2.1 Publish

Publish จะเป็นการกระจายข้อมูลไปยังทุก namespace ที่ subscribe หัวข้อที่ publisher ส่งข้อมูล Publish เริ่มจากการส่ง HTTP request ไปที่ public API ของ message bus ด้วยรูปแบบในรูปที่ 3.10

Method: PUT

URL: http://{message_bus_address}/message/pubSub/{topic_name}

Request body: {message}

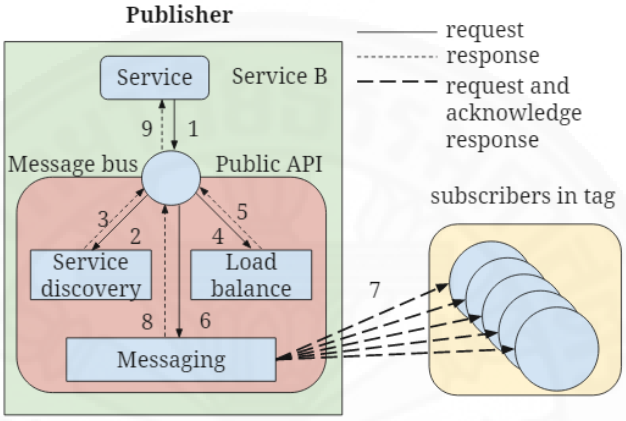
message_bus_address = URL ที่ใช้ในการติดต่อ message bus ค่าตั้งต้น คือ 127.0.0.1:21001

topic_name = ชื่อของหัวข้อที่ service ต้องการ subscribe

message = ข้อมูลที่ต้องการส่ง

รูปที่ 3.10 รูปแบบการ publish ข้อมูลผ่านทาง public API

จากนั้น public API จะขอที่อยู่ service ใน namespace ทั้งหมดที่ subscribe หัวข้อที่จะส่งจาก service discovery และจะส่งต่อให้กับ load balance เพื่อทำการเลือก service ที่เหมาะสมในที่จะรับข้อมูล เมื่อได้รับที่อยู่ service ที่จะส่งทั้งหมด จึงส่งต่อให้กับ messaging module ส่งข้อมูลกระจายไปยังเป้าหมายทั้งหมดแบบ request and acknowledge response นั่นคือส่งข้อมูลไปแล้วเป้าหมายตอบกลับทันทีเมื่อได้รับข้อมูล โดยไม่รอให้ประมวลผลข้อมูลเสร็จก่อน โดยขั้นตอนทั้งหมดเป็นไปดังรูปที่ 3.11



รูปที่ 3.11 ขั้นตอนการ publish ข้อมูล

3.2.2.2.2 Subscribe

Subscribe จะเป็นการ subscribe namespace ของ service กับหัวข้อ ส่งผลให้ service ที่อยู่ใน namespace เดียวกันกับ service ที่ subscribe มีโอกาสได้รับข้อมูลของหัวข้อเช่นกัน การ subscribe โดยเริ่มจาก service ส่ง HTTP request ไปยัง public API ของ message bus เพื่อขอ subscribe หัวข้อที่ต้องการรับข้อมูล โดยมีรูปแบบการส่งดังรูปที่ 3.12 ซึ่ง message bus จะปฏิเสธการ subscribe กรณีที่หัวข้อยังไม่ถูกสร้างโดย publisher

Method: PUT

URL: http://{message_bus_address}/topic/{topic_name}/subscriber

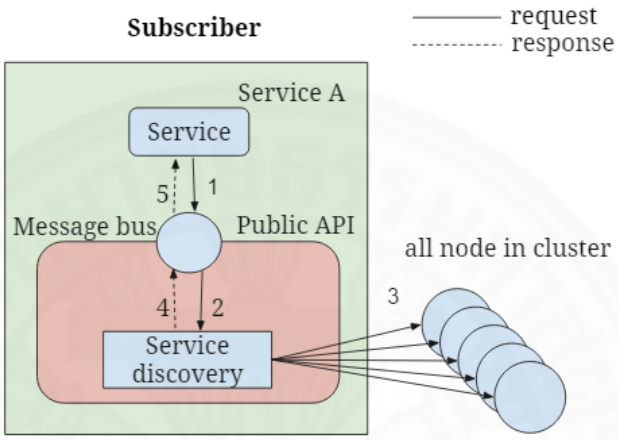
Request body: {}

message_bus_address = URL ที่ใช้ในการติดต่อ message bus ค่าตั้งต้น คือ 127.0.0.1:21001

topic_name = ชื่อของหัวข้อที่ service ต้องการ subscribe

รูปที่ 3.12 รูปแบบการ subscribe ผ่านทาง public API

เมื่อ public API ได้รับ request จาก service แล้วจะส่งข้อมูลไปยัง service discovery เพื่อกระจายข้อมูลการ subscribe ไป message bus อื่นในระบบ เมื่อ message bus อื่นในระบบได้รับข้อมูล จากนั้นส่งผลลัพธ์การ subscribe ไปยัง service ผ่านทาง public API ดังรูปที่ 3.13



รูปที่ 3.13 ขั้นตอนการ subscribe

3.2.2.3 Notification

Notification หรือ request/asynchronous response เป็นรูปแบบที่มีผู้ส่งและผู้รับอย่างละหนึ่งเหมือน request/response ต่างกันคือผู้ส่งจะไม่รอให้ผู้รับตอบกลับมาทำให้จะไม่มีการปิดกั้นการทำงานของผู้ส่ง ซึ่งมีรายละเอียดดังต่อไปนี้

Method: PUT

URL: http://{message_bus_address}/message/noti/{namespace}

Request body: {message}

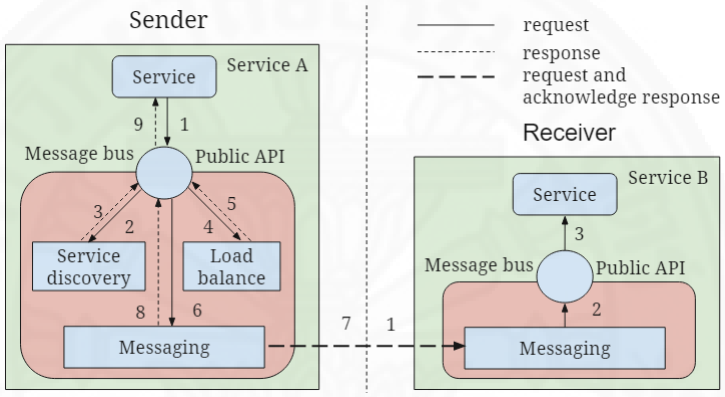
message_bus_address = URL ที่ใช้ในการติดต่อ message bus ค่าตั้งต้น คือ 127.0.0.1:21001

topic_name = ชื่อของหัวข้อที่ service ต้องการ subscribe

message = ข้อมูลที่ต้องการส่ง

รูปที่ 3.14 รูปแบบการส่ง request ของ notification

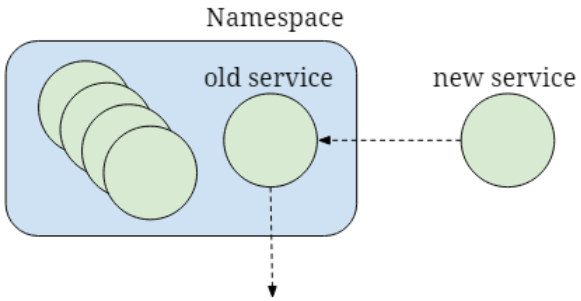
Service จะส่ง HTTP request ไปยัง public API ของ message bus โดยมีรูปแบบการส่งดังรูปที่ 3.14 เมื่อ public API ได้รับ request จะขอที่อยู่ของ service ใน namespace ทั้งหมดจาก service discovery แล้วส่งต่อให้ load balance จัดการเลือก service เป้าหมาย จากนั้น public API จะเรียก messaging module ให้ทำการส่งข้อมูลแบบ request and response acknowledge ไปยัง message bus เป้าหมาย เมื่อได้รับการตอบกลับ จะส่งผลลัพธ์การส่งไปยังผู้ส่งผ่านทาง public API ในขณะที่ message bus เป้าหมายจะส่งมูลไปยัง service ผ่านทาง public API โดยใช้ contactPoint ที่ได้จากตอนลงทะเบียน service โดยขั้นตอนทั้งหมดเป็นไปดังรูปที่ 3.15



รูปที่ 3.15 ขั้นตอนการ request ใน notification

3.2.3 ขั้นตอนทำการอัพเดท service ใหม่เป็น version ใหม่

การเปลี่ยนแปลง service ในงานวิจัยชิ้นนี้นั้น สามารถทำได้โดยการถอด service เก่าออกแล้วใส่ service ใหม่เข้าไปแทนดังรูปที่ 3.16 แต่สิ่งที่ผู้ใช้ควรระวังคือ ลำดับขั้นตอนของการเปลี่ยน service มีผลกระทบที่แตกต่างกันออกไป โดยขั้นตอนการเปลี่ยน service สามารถทำได้สองรูปแบบคือ ถอด service เก่าออกแล้วใส่ service ใหม่เข้ามา และการถอด service เก่าออกก่อนจะทำให้ service ที่ให้บริการอยู่ใช้งานไม่ได้ชั่วขณะ ในกรณีระบบมีการรองรับที่ดีสำหรับเหตุการณ์ที่ service อยู่ทำงานชั่วคราว วิธีนี้จะเหมาะที่จะใช้ในการอัพเดท service ส่วนอีกวิธีคือการ เพิ่ม service ใหม่เข้ามาก่อนแล้วจึงถอด service เก่า วิธีนี้จะทำให้บริการของ service ไม่หยุดชะงัก แต่ข้อควรระวังคือ เรื่อง consistency ของข้อมูลใน service ซึ่งการใช้งาน service ที่ต่าง version เพิ่มกันอาจทำให้ข้อมูลในฐานข้อมูลไม่ตรงกัน ซึ่งอาจจะส่งผลเสียต่อระบบ



รูปที่ 3.16 วิธีการอัพเกรด service ในระบบ

โดยการเพิ่ม service เข้ามาในระบบนั้นได้พูดถึงในบทที่ 3.2.1 แล้ว ดังนั้นในบทนี้จะพูดถึงส่วนที่เหลือคือการถอด service ออกจากระบบ ซึ่งจะเริ่มจาก service ส่ง HTTP request ไปยัง message bus ผ่านทาง public API โดยมีรูปแบบการส่งดังรูป 3.17

Method: DELETE

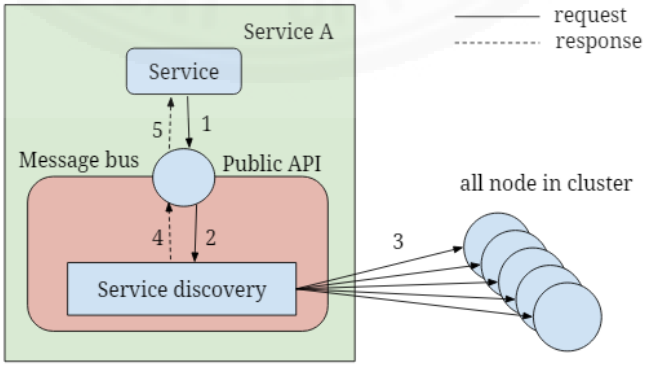
URL: http://{message_bus_address}/namspace/{namespace}

message_bus_address = URL ที่ใช้ในการติดต่อ message bus ค่าตั้งต้น คือ 127.0.0.1:21001

namespace = ชื่อของ namespace ที่ service ต้องการลงทะเบียน

รูปที่ 3.17 รูปแบบการถอด service ออกจากระบบ

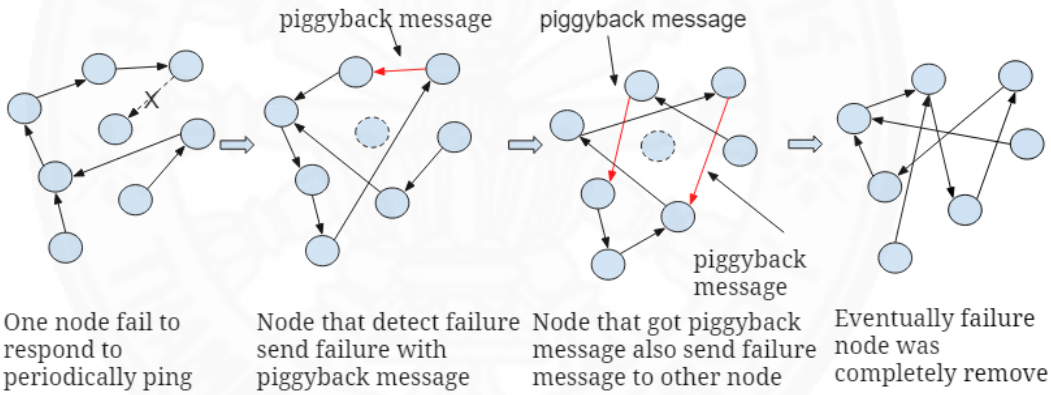
เมื่อ public API ได้รับ request จะส่งข้อมูลการถอด service ออกจากระบบไปที่ service discovery เพื่อกระจายข้อมูลไปยังทุก message bus ในระบบเพื่อทำการถอด service ออกจากข้อมูลของ message bus จากนั้นก็ส่งผลลัพธ์กลับไปยัง service โดยขั้นตอนเป็นไปดังรูปที่ 3.18



รูปที่ 3.18 ขั้นตอนการถอด service ออกจากระบบ

3.2.4 ขั้นตอนเมื่อ service หยุดทำงาน

งานวิจัยชิ้นนี้ message bus ซึ่งเป็นเครื่องมือที่ใช้สื่อสารกันระหว่าง service อยู่ในรูปแบบ decentralized นั้นจะมี message bus หลายตัวที่หน้าที่เหมือนกันสามารถทดแทนกันได้ อาศัยอยู่เป็น sidekick process ของ service ซึ่ง message bus แต่ละตัวจะรู้จักกันได้ต้องอาศัย service discovery ซึ่งในงานวิจัยชิ้นนี้เลือกใช้ Serf ที่ใช้ SWIM protocol เป็นเครื่องมือที่ช่วยในการจัดการสมาชิกในกลุ่ม cluster และตรวจหาส่วนที่หยุดทำงานของสมาชิกกลุ่ม ซึ่งใน SWIM protocol อาศัยการส่งข้อมูลขนาดเล็กเป็นช่วงเวลา ด้วย UDP ไปยังสมาชิกที่สุ่มขึ้นมา ถ้าไม่ได้รับการตอบกลับในระยะเวลาที่กำหนด จะกระจายข้อมูลของสมาชิกที่หยุดทำงาน ด้วยวิธี piggyback message ซึ่งเป็นการแนบข้อมูลไปกับการส่งข้อมูลที่ใช้ในการเช็คว่าสมาชิกยังทำงานอยู่ เมื่อสมาชิกได้รับข้อมูลจะทำการลบข้อมูลของสมาชิกที่ล้มเหลวออก และส่งข้อมูลต่อไปด้วย piggyback message จนในที่สุดทุกสมาชิกได้รับข้อมูลครบ ดังรูปที่ 3.19



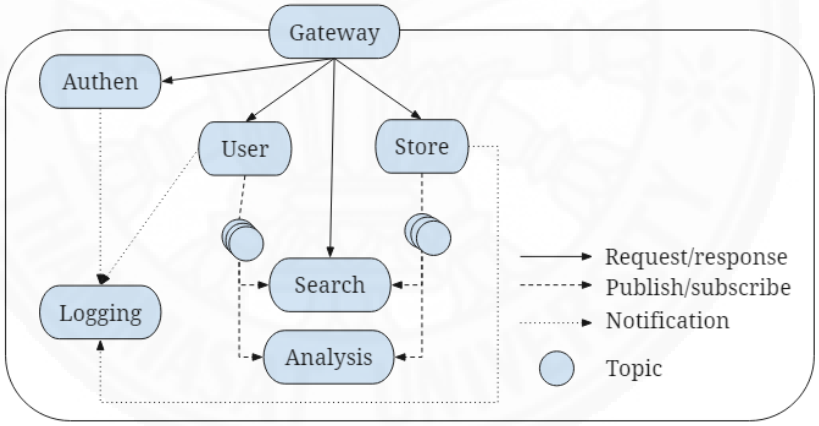
รูปที่ 3.19 ขั้นตอนการตรวจหา service ที่ล้มเหลว

บทที่ 4
ผลการวิจัยและอภิปรายผล

เพื่อทดสอบความถูกต้องของงานวิจัย งานวิจัยนี้จะทำการทดสอบระบบ โดยแบ่งออกเป็นสามด้านคือ ความสามารถในการรับรองระบบในรูปแบบการใช้งานทั่วไป ระดับการใช้ทรัพยากรของ message bus และประสิทธิภาพของการส่งข้อมูล

4.1 ความสามารถในการรับรองระบบในรูปแบบการใช้งานทั่วไป

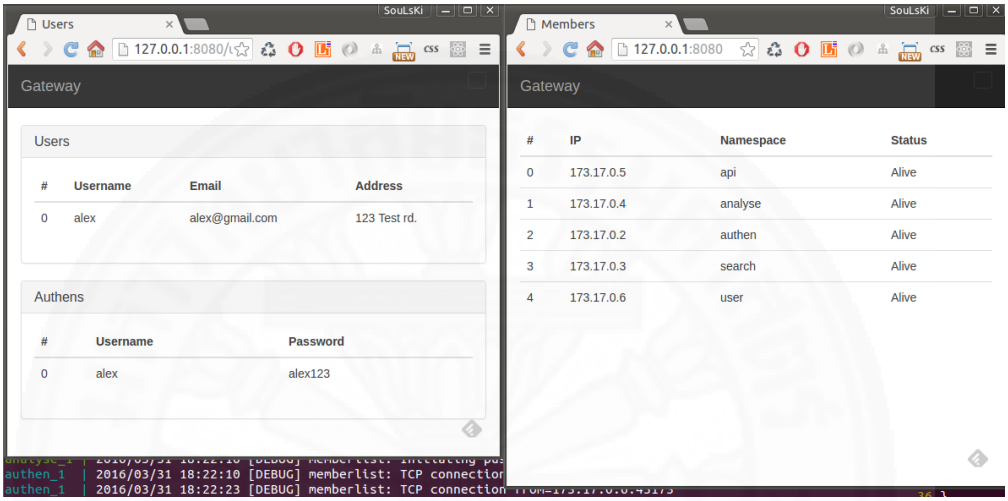
การทดสอบความสามารถในการรับรองระบบ จะทำการจำลองตัวอย่างระบบที่มีอยู่จริงในปัจจุบัน เพื่อแสดงให้เห็นว่า message bus สามารถใช้ในระบบทั่วไปได้จริง ระบบที่งานวิจัยชิ้นนี้เลือกขึ้นมาทดสอบคือ ระบบ e-commerce เพราะเป็นระบบที่เป็นที่รู้จักกันเป็นอย่างดี สามารถเข้าใจได้ง่าย โดยระบบจำลองจะประกอบไปด้วย service ที่สัมพันธ์กัน ดังรูปที่ 4.1 ในแต่ละ service ทำหน้าที่ดังต่อไปนี้



รูปที่ 4.1 service ในระบบจำลอง

Gateway ทำหน้าที่เป็นจุดที่ระบบภายนอกใช้ในการติดต่อกับระบบภายใน โดย gateway จะทำหน้าที่รับคำขอจากระบบภายนอก จากนั้นจะติดต่อขอข้อมูลไปยัง service อื่นในระบบและรวบรวมข้อมูลเพื่อทำการส่งกลับไปยังระบบภายนอกที่ร้องขอข้อมูล โดย gateway ได้จัดเตรียม interface ที่ใช้ในการดูข้อมูลของระบบซึ่งอยู่ในรูปแบบเว็บไซต์ ดังตัวอย่างในรูปที่ 4.2 โดยข้อมูลที่ gateway จัดเตรียมไว้มีดังต่อไปนี้

- ข้อมูลสถานะของ node ทั้งหมดในระบบ
- ข้อมูลของผู้ใช้งานทั้งหมดในระบบ
- ข้อมูลของสินค้าทั้งหมดในระบบ
- ข้อมูลของเหตุการณ์ที่เกิดขึ้นในระบบ
- ข้อมูลของระบบ search และ analysis



รูปที่ 4.2 ตัวอย่างของ interface ของ gateway

รูปแบบการสื่อสารที่ gateway ใช้ติดต่อกับ service ทั้งหมดคือ request/response เนื่องจาก gateway ต้องการข้อมูลตอบกลับจาก service เป้าหมายเพื่อส่งข้อมูลผลลัพธ์กลับไปยังระบบภายนอก

Authen ทำหน้าที่จัดการข้อมูลที่เกี่ยวกับ authorization และ authentication ยกตัวอย่างเช่น เพิ่มผู้ใช้งานที่มีสิทธิ์เข้าสู่ระบบ ตรวจสอบสิทธิ์ของผู้ใช้งาน โดยทั้งหมดนี้จะให้บริการผ่านทางการติดต่อสื่อสารแบบ request/response

User ทำหน้าที่บริหารจัดการข้อมูลของผู้ใช้งานทั้งหมดในระบบ โดยจะให้บริการผ่านการติดต่อสื่อสารแบบ request/response user จะมีการ publish ข้อมูลที่เกี่ยวข้องกับการจัดการ user ยกตัวอย่างเช่น การสร้าง user และการลบ user เพื่อกระจายข้อมูลให้กับระบบที่ต้องการ ดังจะเห็นได้จากรูปที่ 4.1 search และ analysis ทำการ subscribe ข้อมูลจาก user เพื่อนำข้อมูลไปใช้ในระบบ

Store ทำหน้าที่บริหารจัดการสินค้าทั้งหมดในระบบ โดยให้บริการผ่านการติดต่อสื่อสารแบบ request/response store จะมีการกระจายข้อมูลที่เกี่ยวข้องกับการจัดการสินค้าในระบบผ่านการติดต่อสื่อสารแบบ publish/subscribe ยกตัวอย่างเช่น เพิ่มสินค้าเข้ามาใหม่ ทำการลบสินค้า โดย service ที่ subscribe กับ store มีอยู่สอง service คือ search และ analysis

Logging ทำหน้าที่จัดเก็บข้อมูลเหตุการณ์ที่เกิดขึ้นในระบบตัวอย่างเช่น ข้อผิดพลาดในระบบ ข้อมูลของผู้ใช้ที่ทำการ login เข้าระบบ โดย logging จะได้รับข้อมูลเหล่าที่จาก service ในระบบผ่านทางการติดต่อสื่อสารแบบ notification เพราะข้อมูลเหล่านี้ไม่ต้องการการตอบกลับแต่ต้องการความรวดเร็วในการติดต่อสื่อสาร

Search ทำหน้าที่ให้บริการค้นหาข้อมูลแบบ full-text search ทั้งระบบโดยข้อมูลจะมาจากการ subscribe topic ของ service user และ store โดย search จะให้บริการค้นหาข้อมูลผ่านทางการสื่อสารแบบ request/response

Analysis ทำหน้าที่จัดเก็บข้อมูลการใช้งานของผู้ใช้ในระบบ ยกตัวอย่างเช่น จำนวน user ที่เข้ามาในระบบ ข้อมูลของ user ในระบบและข้อมูลของสินค้าในระบบ เพื่อแปลงเป็นข้อมูลที่เป็นประโยชน์ในเชิงธุรกิจ โดยข้อมูลจะได้รับมาจากการ subscribe topic ของ service user และ store ข้อมูลที่ถูกวิเคราะห์แล้วจะถูกให้บริการผ่านทางการสื่อสารแบบ request/response

การสร้างระบบจำลอง งานวิจัยชิ้นนี้ได้เลือกใช้เครื่องมือดังต่อไปนี้เป็นตัวช่วยในการสร้างระบบจำลอง ระบบจำลองถูกสร้างขึ้นจากภาษา Golang และใช้ docker เป็นเครื่องมือในการจำลองคอมพิวเตอร์หลายเครื่องขึ้นมาบนเครื่องเดียว และเพื่อให้ระบบสามารถสร้างขึ้นใหม่ได้ง่าย สะดวกต่อการทดลอง docker-compose จึงถูกนำมาใช้เพื่อเป็นเครื่องมือในการทำ automation system โดย docker-compose จะสร้าง service ทั้งหมดตามที่ระบุใน docker-compose file และใช้ docker network จำลอง virtual network สำหรับ service โดย containers ทั้งหมดจะถูกแจกจ่าย IP ให้อยู่ใน CIDR 173.17.0.0:24 ฐานข้อมูลที่ service ทั้งหมดใช้คือ sqlite ระบบทั้งหมดสามารถถูกสร้างจากเริ่มต้นจนเสร็จสมบูรณ์ด้วยการใช้ Makefile โดย source code ทั้งหมดได้ถูกนำขึ้นไปวางบน github โดยสามารถเข้าไปได้ที่ URL https://github.com/soulski/dmp-scenario

การทดสอบความถูกต้องในระบบ เราได้ทำการทดสอบสองเรื่องคือ ความถูกต้องในการส่งข้อมูลและความสามารถในการจัดการเมื่อเกิดความล้มเหลวในระบบ ในการตรวจสอบความถูกต้องของการส่งข้อมูลนั้น เราจะทำการทดสอบการส่งข้อมูลทั้งสามรูปแบบดังตารางที่ 4.1

ตารางที่ 4.1 วิธีการทดสอบความถูกต้องของการส่งข้อมูลของ message bus

รูปแบบการส่งข้อมูล	ทดลอง	ผลลัพธ์
Request/response	ส่งคำสั่งไปยัง gateway คือทำงานในระบบ	Gateway จะส่งข้อมูลไปยังระบบปลายทางได้อย่างถูกต้อง
Publish/subscribe	สั่ง user ทำการสร้างข้อมูลผู้ใช้งานในระบบ	User ทำการ publish ข้อมูลไปยัง subscriber ซึ่งในที่นี้คือ Search, Analyze
Publish/subscribe	สั่ง store ทำการสร้างข้อมูลสินค้า	Store ทำการ publish ข้อมูลไปยัง subscriber ซึ่งในที่นี้คือ Search, Analyze
Notification	ทำการส่งข้อมูล login ที่ไม่มีถูกต้อง	Authen ทำการส่งข้อมูลแบบ notification ไปยัง Logging
Notification	ทำการส่งข้อมูลในการสร้าง user ที่ไม่ถูกต้อง	User ทำการส่งข้อมูล user ด้วย notification ไปยัง Logging
Notification	ทำการส่งข้อมูลในการสร้าง store ที่ไม่ถูกต้อง	Store ทำการส่งข้อมูลสินค้าด้วย notification ไปยัง Logging

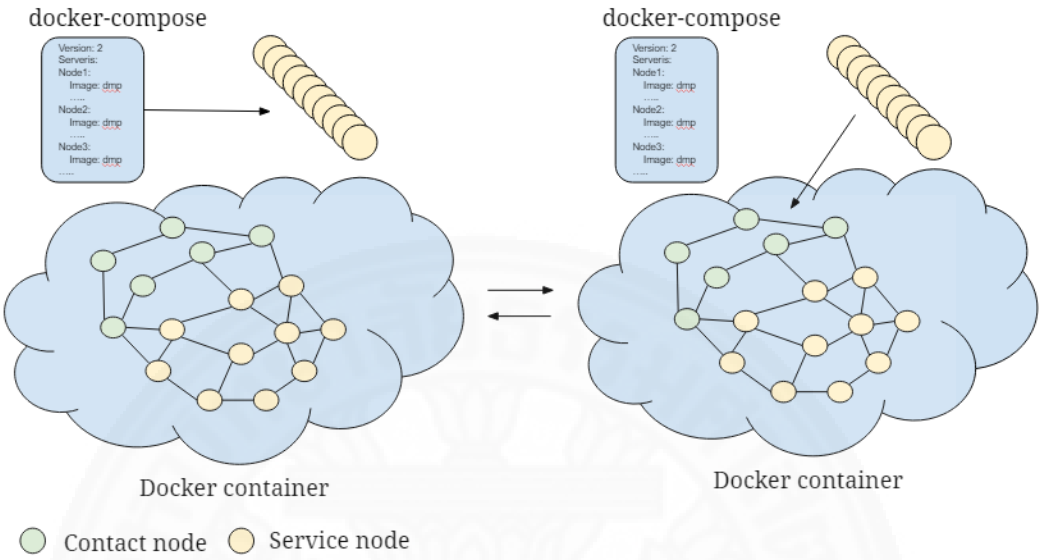
ระบบ Gateway จะมีหน้าแสดงข้อมูลของระบบทั้งหมดอยู่ เพื่อใช้ในการตรวจสอบข้อมูล ผลลัพธ์ที่ได้จากการทดลองคือทุกกรณีสามารถทำงานได้อย่างถูกต้อง ด้านความสามารถในการจัดการเมื่อเกิดความล้มเหลวในระบบนั้น จะทำการปิด docker ของ service แล้วทำการเรียกขอดูข้อมูลของ namespace ที่ service ลงทะเบียนว่ายังมี service อยู่หรือไม่ อีกทั้งยังทำการทดลองด้วยการส่งข้อมูลไปยัง namespace ที่ service ลงทะเบียนซึ่งผลลัพธ์คือ ระบบสามารถแสดงข้อมูลของ namespace ที่ไม่มี service ได้อย่างถูกต้อง และระบบยังสามารถส่งข้อมูลไปยัง namespace ที่ service ถูกปิดได้อย่างถูกต้อง

จากที่กล่าวมา จะเห็นได้ว่าระบบจำลองสามารถทำงานได้อย่างถูกต้อง แม่นยำ มีรูปแบบการส่งข้อมูลที่เพียงพอต่อความต้องการของระบบทั่วไป จึงแสดงให้เห็นว่า message bus สามารถนำไปใช้งานกับระบบทั่วไปได้

4.2 ระดับการใช้ทรัพยากรของ message bus

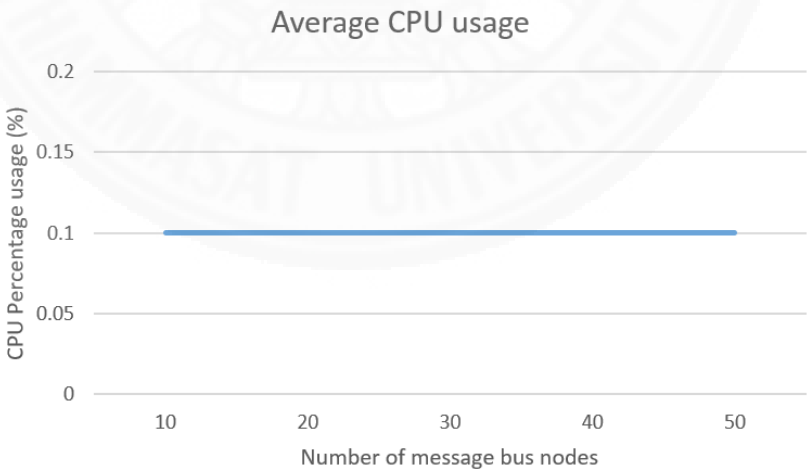
เพื่อทดสอบความเหมาะสมในการทำงานในระบบขนาดใหญ่ เราจึงทำการออกแบบการทดสอบด้วยการสร้าง docker-compose file ที่ใช้เพื่อทำการสร้าง docker ที่มี message bus

ทำงานอยู่ รอบละ 10 containers โดย message bus จะมีการตั้งค่าให้เข้ากลุ่ม cluster โดยร้องขอไปที่ message bus กลุ่มแรกที่ถูกสร้างดังรูปที่ 4.3

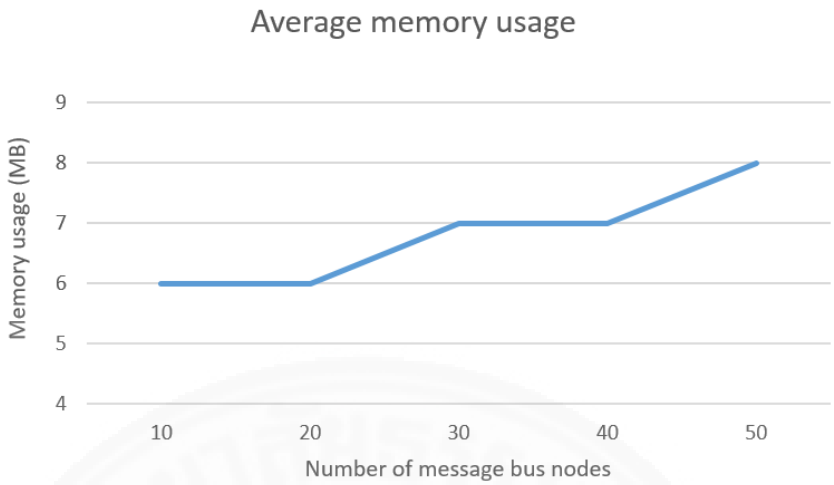


รูปที่ 4.3 วิธีการทดลองการใช้ทรัพยากรของ message bus

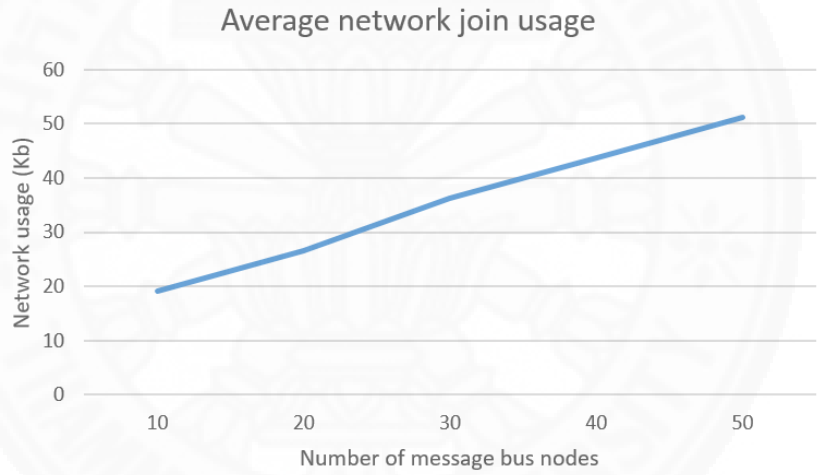
การทดลองจะสั่งให้สร้าง docker จาก docker-compose ทั้งหมด 4 รอบ โดยแต่ละรอบจะมีการวัด CPU, memory, network i/o และ network ที่ใช้เพื่อเข้าร่วม cluster ของ message bus ที่ใช้ในการทำงาน ผลลัพธ์ที่ได้ออกมาดังรูปที่ 4.3, 4.4, 4.5, 4.6



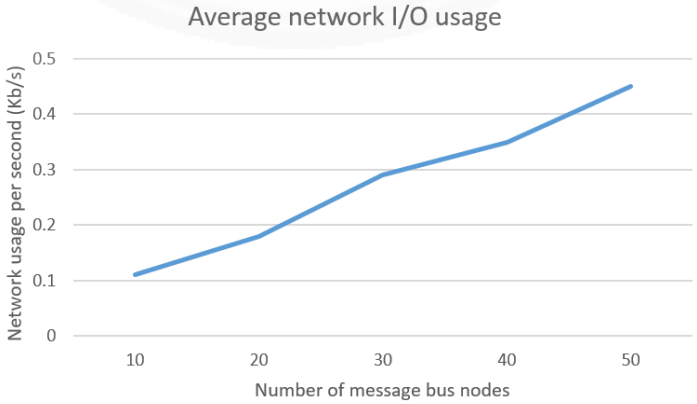
รูปที่ 4.4 อัตราการใช้ CPU ของ message bus



รูปที่ 4.5 อัตราการใช้ Memory ของ message bus



รูปที่ 4.6 อัตราการใช้ Network เพื่อเข้าร่วม cluster ของ message bus

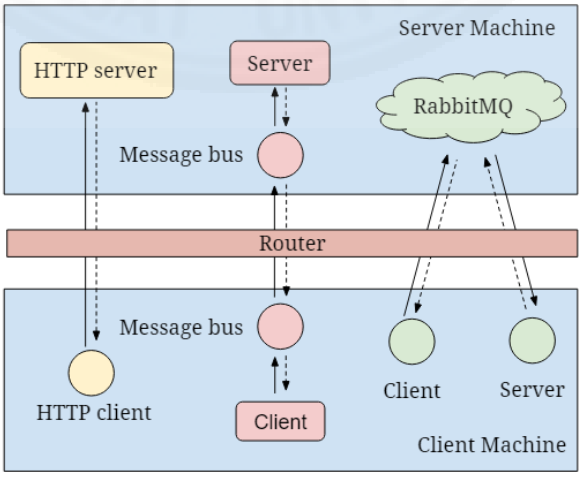


รูปที่ 4.7 อัตราการใช้ Network I/O ของ message bus

จากกราฟแสดงให้เห็นว่า message bus ใช้ CPU เพิ่ง 0.1% ในอัตราคงที่ แต่ความจริงแล้วยังมีการเพิ่มของการใช้งาน CPU อยู่ แต่เป็นอัตราที่ต่ำมาก ส่วน memory กราฟออกมามีลักษณะเป็นขั้นบันได ซึ่งการจากเครื่องมือที่ใช้ในการวัดอัตราการใช้งานพื้นที่ memory นั้นมีหน่วยเป็น megabyte แต่อัตราการเพิ่มของการใช้พื้นที่ memory นั้น มีหน่วยเพียง kilobyte ทำให้อัตราการเพิ่มของกราฟจะมีลักษณะเป็นช่วง จากที่กล่าวมาเห็นได้ว่าจำนวน node ใน cluster นั้นส่งผลเพียงเล็กน้อย ต่ออัตราการใช้งานพื้นที่ memory และมีอัตราการเพิ่มขึ้นแบบ linear ในขณะที่ network ที่ใช้ในการเข้าร่วม cluster และ network I/O เพิ่มขึ้นเป็นแบบ linear ซึ่งเป็นเพราะเมื่อจำนวน node เพิ่มขึ้นจำนวน network ที่ต้องใช้ในการสื่อสารเพื่อหา node ที่ล้มเหลวก็มากขึ้น แต่ขนาด network I/O ที่ใช้ก็ยังอยู่ในอัตราที่ไม่ได้สูงมากจนเกินไป โดยจากกราฟทั้งหมดแสดงให้เห็นว่า message bus มีอัตราการใช้งานทรัพยากรบนเครื่องเมื่อจำนวน node เพิ่มขึ้นเพียงเล็กน้อย ทำให้สามารถใช้งานกับระบบที่มีขนาดเล็กถึงกลางหรือแม้กระทั่งขนาดใหญ่ได้เป็นอย่างดี

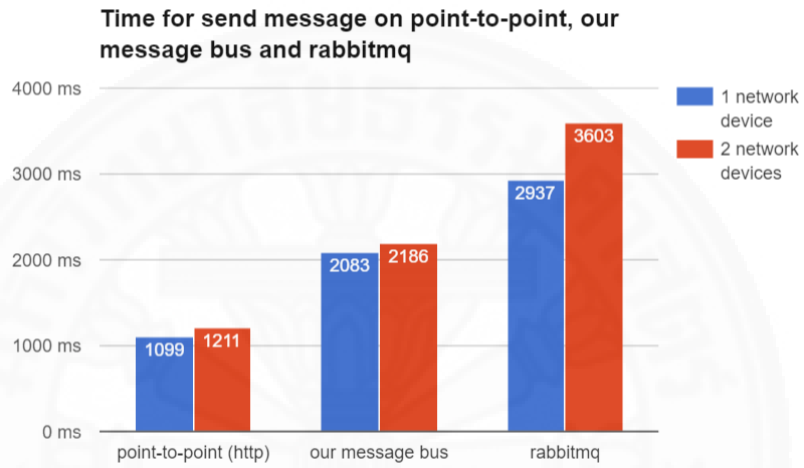
4.3 ประสิทธิภาพของการส่งข้อมูล

เพื่อทดสอบประสิทธิภาพการส่งข้อมูลระหว่าง service ด้วย message bus เราจึงทำการทดสอบด้วยการเปรียบเทียบกับเครื่องมือที่นิยมใช้ในการสื่อสารระหว่าง service ใน microservices 2 ชนิด นั้นคือ HTTP และ message queue โดย message queue เราจะใช้ RabbitMQ ซึ่งเป็น message queue ที่ได้รับความนิยม โดยวิธีการทดสอบนั้นจะเริ่มด้วยการสร้าง server สำหรับรับข้อมูลจากทั้ง HTTP, RabbitMQ และ message bus และตอบกลับ จากนั้นสร้าง client ที่ใช้ในการส่งข้อมูลโดยจะวัดประสิทธิภาพของการส่งข้อมูลที่ฝั่ง client ทั้งหมด รูปแบบของการทดสอบเป็นดังรูปที่ 4.7



รูปที่ 4.8 โครงสร้างการเปรียบเทียบประสิทธิภาพการส่งข้อมูล

ทุกรูปแบบการส่งข้อมูล client จะอยู่บนเครื่องคอมพิวเตอร์เดียวกันและ server จะอยู่ที่อีกเครื่องคอมพิวเตอร์หนึ่ง โดยคอมพิวเตอร์จะทำการเชื่อมต่อกันผ่าน router ยกเว้น RabbitMQ เนื่องจาก message queue เป็นระบบที่มีการเชื่อมต่อ 3 ฝั่ง ดังนั้นเพื่อจำลองให้เหมือนระบบจริง client และ server จึงอยู่บนเครื่องเดียวกัน โดยมี RabbitMQ อยู่อีกเครื่อง โดยจะทดสอบด้วยการส่งข้อมูลจาก client ไปยัง server แบบ synchronous จำนวน 1000 requests และเริ่มวัดเวลาตั้งแต่ส่ง request แรกถึง request สุดท้าย จนได้รับผลลัพธ์กลับมา ผลลัพธ์ที่ได้เป็นไปตามรูปที่ 4.8



รูปที่ 4.9 เปรียบเทียบระยะเวลาที่ใช้ในการส่งข้อมูลของ HTTP, message bus and rabbitmq

จากกราฟแสดงให้เห็นว่า message bus ทำได้ไม่ดีเท่า HTTP ถึงแม้จะเกิดการส่งข้อมูลแค่ 1 communication ซึ่งอาจจะเกิดจาก overhead ที่มาจากการส่งข้อมูลระหว่าง process ของ message bus และ service แต่สามารถส่งข้อมูลได้เร็วกว่า message queue ที่มีคุณสมบัติใกล้เคียงกัน อีกทั้งอัตราการเพิ่มของระยะเวลาที่ใช้ในการส่งข้อมูลเมื่อจำนวน network device ที่ข้อมูลต้องวิ่งผ่านในที่การทดลองนี้คือ 1 และ 2 ก็เพิ่มน้อยกว่า ซึ่งเป็นเพราะเมื่อจำนวน communication มากขึ้นจำนวน network device ที่ต้องวิ่งผ่านก็ย่อมมีมากขึ้น ทำให้ประสิทธิภาพในการส่งข้อมูลยิ่งต่ำลง ดังที่เกิดกับ message bus แบบ Iris ที่ใช้ protocol DHT ในการส่งข้อมูลทำให้จะเกิด communication hop log(n), n คือจำนวนของ node ทั้งหมดในระบบ (Szilagyi P, 2014) ส่งผลให้ประสิทธิภาพในการส่งข้อมูลของ Iris นั้นจะตกลงเมื่อมีจำนวน node ที่มากขึ้น ดั้งนั้น message bus ในงานวิจัยนี้เป็นเครื่องมือที่เหมาะสมในการใช้งานกับระบบ microservices มากกว่า message queue ในเชิงความเร็วของการส่งข้อมูล

บทที่ 5
สรุปผลการวิจัยและข้อเสนอแนะ

ในปัจจุบัน microservices ซึ่งเป็น software architecture style กำลังได้รับความนิยม เนื่องจาก microservices ช่วยลด complexity ของระบบ ซึ่งเป็นปัญหาสำคัญในการพัฒนา software โดย microservices จะทำการแบ่งระบบขนาดใหญ่ออกเป็น service ขนาดเล็กหลาย service โดยแต่ละ service จะมีลักษณะเป็น self-container ดังนั้นการ deploy ระบบของแต่ละ service จะทำแยกกัน ทำให้ระบบที่พัฒนาด้วย microservices รองรับต่อการเปลี่ยนแปลงที่บ่อยได้ ทำให้เหมาะกับกับระบบการพัฒนาแบบ agile ที่เน้นการเปลี่ยนแปลงระบบบ่อยเพื่อให้ตรงกับความต้องการของลูกค้า

แต่อย่างไรก็ตามเมื่อ service มีลักษณะเป็น self-container ดังนั้น function call ระหว่าง service ถูกเปลี่ยนมาเป็น remote call service ทำให้เกิดการผูกติดกันของ service ซึ่งจะส่งผลให้ความยืดหยุ่นของระบบลดลง โดยก็ได้มีความพยายามในการแก้ไขด้วยการใช้ message queue ซึ่งจะเป็นตัวกลางในการติดต่อสื่อสาร ทำให้ทุกการติดต่อจะต้องผ่าน message queue ส่งผลให้ประสิทธิภาพในการสื่อสารลดลง

โดยงานวิจัยชิ้นนี้จึงนำเสนอ decentralized message bus ที่มุ่งเน้นไปที่การช่วยลดการผูกติดกันของ service และช่วยลด operational complexity ด้วยการใช้ service discovery message bus มี architecture แบบ decentralized ไม่มีจุดศูนย์กลางในการติดต่อ แต่ละ node สามารถทำหน้าที่แทนกันได้ โดย message bus จะอยู่ในรูปแบบ sidekick process ซึ่งจะอาศัยอยู่ในเครื่องเดียวกับ service รูปแบบการสื่อสารใน message bus ได้ถูกจัดเตรียมไว้ให้ 3 รูปแบบคือ request/response, publish/subscribe และ notification อีกทั้งช่วยสนับสนุนการเพิ่ม scalability และ availability ในระบบให้ง่ายขึ้น ด้วยการจัดทำ auto load-balancer ผ่านทาง concept namespace ทำให้ผู้ใช้งานไม่ต้องทำ manual configuration โดยจะการวัดความสามารถในการทำงานของ message bus ด้วยการใช้ message bus สร้างระบบจองลอง e-commerce และตรวจสอบความถูกต้องของระบบ อีกทั้งยังตรวจความสามารถในการขยายตัวด้วยการวัดอัตราการใช้ทรัพยากรเมื่อจำนวน message bus ในระบบเพิ่มขึ้น สุดท้ายวัดประสิทธิภาพของการส่งข้อมูล ด้วยการเปรียบเทียบกับเครื่องมือสื่อสารที่เป็นที่นิยมใน microservices อย่าง HTTP และ message queue ซึ่งสามารถสรุปผลได้ดังนี้

5.1 ความสามารถในการทำงาน

จากการทดลองสร้างระบบจำลอง e-commerce ด้วย message bus ซึ่งมีระบบรูปแบบการใช้งานที่หลากหลาย ทั้งระบบการค้นหา ระบบสินค้า ระบบการติดตามผู้ใช้งาน และระบบการตรวจจับความผิดปรกติของระบบ พบว่า message bus สามารถทำงานได้อย่างถูกต้องทั้งในการส่งข้อมูล การตรวจจับความล้มเหลวของ service ในส่วนการกระจายโหลดนั้นยังมีข้อจำกัดในส่วนของข้อมูลแต่ละ load-balance ไม่มีการเชื่อมต่อกันทำให้อาจจะเกิดการส่งข้อมูลไปยัง service เดียวซ้ำกันได้ โดยลักษณะของ service ที่เหมาะสมกับ message bus ในงานวิจัยนี้คือ stateless service เนื่องจากการทำงานแบบกระจายของ message bus และคุณสมบัติของบริการใน microservices ที่มีลักษณะแบบ isolate จึงทำให้การทำ service ที่มีลักษณะ stateful จะไม่เหมาะกับ message bus ในงานวิจัยชิ้นนี้ ซึ่งแสดงให้เห็นว่า message bus มีความสามารถในแง่ของการนำไปใช้งานในระบบทั่วไปได้อย่างมีประสิทธิภาพ

5.2 ความสามารถในการขยายตัว

จากผลการทดลองจะเห็นได้ว่าอัตราการใช้ทรัพยากรในระบบอย่างเช่น network I/O มีอัตราการเพิ่มขึ้นเป็น linear ตามจำนวน node ที่เพิ่มขึ้น ในขณะที่ CPU และ memory ก็ใช้เพียงเล็กน้อยและเพิ่มขึ้นน้อยมากเมื่อจำนวน node เพิ่มขึ้น ซึ่งแสดงให้เห็นว่า message bus สามารถที่จะขยายตัวได้ดี จำนวน node ในระบบส่งผลกระทบไม่มากนักกับประสิทธิภาพของการขยายตัวของระบบ

5.3 ประสิทธิภาพของการสื่อสาร

ในส่วนของประสิทธิภาพในการส่งข้อมูล เนื่องจากเกิด communication แค่ 1 hop ในการสื่อสารทำให้ message bus สามารถทำงานได้ดีกว่า message queue ซึ่งจะเกิด communication มากกว่า 1 hop แต่ก็ยังทำงานได้ไม่ดีเท่า point-to-point communication อย่าง HTTP แต่อย่างไรก็ตาม message bus ก็มีคุณสมบัติที่มากกว่า HTTP

5.4 ข้อเสนอแนะในการทำวิจัยต่อไป

เนื่องจากงานวิจัยชิ้นนี้มุ่งเน้นไปที่ระบบการสื่อสาร และการทำ service discovery ทำให้ยังมีบางจุดที่ยังต้องการการปรับปรุงเพื่อให้ระบบสามารถทำงานได้ดีขึ้น โดยจุดที่อาจจะเพิ่มเติมได้มีดังต่อไปนี้

- รองรับการทำงานบนระบบการสื่อสารแบบเชื่อมโยงระยะไกล ตัวอย่างเช่น WAN หรือ Internet เนื่องจาก SWIM protocol ที่งานวิจัยนี้ใช้ในการทำ failure detection ยังมีจุดอ่อนอยู่ที่การทำงานบนระบบการสื่อสารแบบ WAN หรือ Internet เนื่องจาก protocol ที่ใช้ในการ detect failure เป็น UDP ซึ่งการทำงานบนระบบสื่อสารทางไกลมีโอกาสที่ข้อมูลจะสูญหายกลางทางสูง ทำให้อาจจะต้องการ protocol แบบ TCP ซึ่งสามารถทำงานได้ดีกับระบบการสื่อสารแบบทางไกล แต่ก็ยังต้องทำให้ระบบการทำ failure detection ยังคง lightweight ไม่ทำให้ระบบการสื่อสารเกิด network congestion หรืออีกทางเลือกหนึ่งคือทำระบบแบบ hybrid ระหว่าง decentralized และ centralized เมื่อต้องการทำ failure detection บนระบบ WAN หรือ internet จะใช้ระบบแบบ centralized แทน
- รองรับการ upgrade service ให้ง่ายขึ้น โดยอาจจะใช้การทำ alias namespace ของ service ซึ่ง alias คือ namespace ที่ไม่ได้ผูกติดกับ service แต่จะผูกติดกับ namespace ด้วยกันเองเมื่อ service ต้องการส่งข้อมูลมายัง alias namespace ระบบจะทำการขอ service จาก namespace ที่ alias ผูกติดอยู่ด้วย โดยสามารนำมาใช้ในการช่วยในการ upgrade service ได้โดยตัวอย่างเช่น alias namespace user ถูกผูกกับ namespace user_v_1 เมื่อต้องการ upgrade service ผู้ดูแลระบบจะสร้าง namespace user_v_2 เมื่อทุกอย่างเสร็จสิ้นจะผูก alias user เข้ากับ user_v_2 ทำให้การเปลี่ยน namespace เป็นไปได้อย่างราบรื่นขึ้น
- รองรับการ monitor service ทั้งระบบในระดับที่ละเอียดขึ้น อาจจะสามารถทำได้เมื่อมีการตอบกับการ ping จะแนบข้อมูลสถานะของ service เป้าหมายกลับไปด้วย เพื่อให้ message bus เก็บไว้ เมื่อเกิดการร้องขอข้อมูลของ service ทั้งหมด message bus จะดึงเอาข้อมูลจากกลุ่ม message bus จำนวนหนึ่งมาเลือกข้อมูลที่ใหม่ที่สุดแล้วส่งกลับไปยังระบบที่ต้องการ

รายการอ้างอิง

หนังสือและบทความในหนังสือ

Gregor H, Boboy W. Enterprise Integration Patterns. 1st ed. Wesley; 2003.

วิทยานิพนธ์

Stubbs J, Moreira W, Dooley R. Distributed Systems of Microservices Using Docker and Serfnode. IWSG ‘7. 2015:34-39.

Toffetti G, Brunner S, Blochlinger M, Dudouet F, Edmonds A. An architecture for self-managing microservices. AIMC ‘15. 2015:19-24

Szilagyi P. Iris: A decentralized approach to backend messaging middlewares. Computer Science and Information Systems, 2014;(11)2:549–567.

Henry S, Kafura K. Software structure metrics based on information flow. IEEE, Transactions on Software Engineering, 1982:510–518

McCabe T. A Complexity Measure. IEEE, Transactions on Software Engineering, 1976 Dec;SE-2(4):308-320

Das A, Gupta I, Motivala A. SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol. DSN 2002. 2002:303-312

Fielding R. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000.

สื่ออิเล็กทรอนิกส์

Fowler M, Lewis J. Microservices [Internet]. Chicago: Martin Fowler; 2014 [updated 2014 March 25; cited 2016 Jan 16]. Available from: http://martinfowler.com/articles/microservices.html.

Richardson C. Introduction to Microservices [Internet]. San Francisco: Nginx. 2015 May - [cited 2016 Jan 16]. Available from: https://www.nginx.com/blog/introduction-to-microservices/

Mason R. ESB or not to ESN Revisited [Internet]. San Francisco: MuleSoft. 2011 June - [cited 2016 Jan 16]. Available from: http://blogs.mulesoft.com/dev/news-dev/esb-or-not-to-esb-revisited-part-1/

Hoff T. Microservices – Not A Free Lunch [Internet]. C2015 - [cited 2016 Jan 16]. Available from: http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html

Betts M. The No. 1 Cause of IT Failure: Complexity [Internet]. 2009 Dec - [cited 2016 Jan 16]. Available from: http://www.computerworld.com/article/2550521/enterprise-applications/the-no--1-cause-of-it-failure--complexity.html

Charette R. Why Software Fails [Internet]. 2005 Sep - [cited 2005 Sep 2]. Available from: http://spectrum.ieee.org/computing/software/why-software-fails

HashiCorp. Serf [Internet]. 2016 [cited 2016 Jan 30]. Available from: https://www.serfdom Airbnb. Nerve [Internet]. GitHub, Inc; c2016 [cited 2016 Jan 30]. Available from: https://github.com/airbnb/nerve.io/

Igor S, Martin R. SmartStack: Service Discovery in the Cloud [Internet]. Airbnb, Inc; 2013 [cited 2016 Jan 30]. Available from: http://nerds.airbnb.com/smartstack-service-discovery-cloud/

Airbnb. Nerve [Internet]. GitHub, Inc; c2016 [cited 2016 Jan 30]. Available from: https://github.com/airbnb/nerve

Airbnb. Synapse [Internet]. GitHub, Inc; c2016 [cited 2016 Jan 30]. Available from: https://github.com/airbnb/synapse

Netflix. Eureka [Internet]. GitHub, Inc; c2016 [cited 2016 Jan 30]. Available from: https://github.com/Netflix/eureka

Apache. Zookeeper [internet]. The Apache Software Foundation; c2010-2016 [cited 2016 Jan 30]. Available from: https://zookeeper.apache.org/

CoreOS. etcd [Internet]. CoreOS, Inc; 2016 [cited 2016 Jan 30]. Available from https://coreos.com/etcd/

ประวัติผู้เขียน

ชื่อ	นาย ภากร คูกรินทร์รัตน์
วันเดือนปีเกิด	20 เมษายน 2527
ตำแหน่ง	Senior Software Developer บริษัท Agoda Company Pte. Ltd

ผลงานทางวิชาการ

Pakorn Kookarinrat, Yaowadee Temtanapat. (2016). Design and Implementation of a Decentralized Message Bus for Microservices. 2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE). 13-15 July 2016

ประสบการณ์ทำงาน

2559 – ปัจจุบัน Senior Software Developer
บริษัท Agoda Company Pte. Ltd
2554 – 2559 Senior Software Developer
บริษัท ShopSpot Pte. Ltd
2550 – 2553 Software Developer
บริษัท MFEC Company