國立政治大學資訊科學系

Department of Computer Science

National Chengchi University

碩士論文

Master's Thesis

Event Sourcing 與 CQRS 架構下的服務穩定機制

Improving Service Stability for Event Sourcing and CQRS Architecture

研 究 生：鄭宇軒

指導教授：廖峻鋒

中華民國 110 年 6 月

June 2021

DOI:10.6814/NCCU202101070

摘要

隨著雲端運算的興起，分散式系統的重要性逐漸受到重視，大量複雜的系統與服務逐漸成為企業的常態。而容器技術的普及，又為新一波軟體部署與管理方式帶來了前所未有的革命，其中以 Docker 和 Kubernetes 等平台的出現，讓微服務的設計概念得以落實。微服務的設計思維不僅能夠降低系統中服務與服務之間的耦合性，也能夠達成跨程式語言的開發模式。傳統資料存取模式(增刪修改)，在面對大量使用者同時寫入的情況下，經常造成單一失效及效能問題。使用命令與查詢責任分離模式 (CQRS, Command Query Responsibility Segregation) 與事件溯源(ES, Event Sourcing)架構，由軟體層面增加了系統的可擴張性，然而這些機制在基礎設施層面上目前仍欠缺完整的錯誤偵測與回復功能，且面對複數使用者同時對資料庫進行改寫時，亦會遇到一致性的問題。本論文採用 CQRS/Event Sourcing 架構的系統，設計一個完整且自主的錯誤偵測與回復機制，亦可處理並行讀寫的問題，並透過實作與數據分析驗證此設計機制的可行性。

關鍵字：Failure Recovery、Failure Detection、CQRS、Event Sourcing

DOI:10.6814/NCCU202101070

Improving Service Stability for Event Sourcing and CQRS Architecture

Abstract

As the Cloud computing raise rapidly, distributed system gradually attach great importance to a lot of software engineers around the world. Having a great range of complex business's logic system and services become enterprise's new normal condition. With the universal of container technology, the new wave of unprecedented revolution has spread into all kinds of software deployments and software managements. The appearance of Kubernetes and Docker platforms makes Microservice pattern have the chance to implement in enterprise's business system. The design concept of Microservice pattern not only can reduce the coupling between service and service but also can allow user to develop system in multi-platform environment. In the traditional method for data access(CRUD), when a lot of users' update data simultaneously, this will often cause single failure and low performance. Using Command Query Responsibility Segregation(CQRS) and Event Sourcing can enhance the scalability of system in software level. However, these mechanisms in infrastructure aspect still lack of complete failure detection and failure recovery. Additionally, when facing a huge amount of users update the database at the same time, will also encounter consistency problem. This essay will adopt the pattern of CQRS/Event Sourcing Architecture to design a complete failure detection and automatic failure recovery, this also can solve the problem of data access in parallel. We verify the availability of this system through implementation and data analysis.

Keyword : Failure Recovery、Failure Detection、CQRS、Event Sourcing

DOI:10.6814/NCCU202101070

致謝

又來到大家看論文最喜歡看到的致謝環節了 XD，說真的我去圖書館看別人的論文第一先看題目(廢話 ==!)再來看目錄，最後就是看我最喜歡的致謝環節(然後就把書闔上了 QQ)，如今沒想到換成自己來寫不免有些小激動哈哈。

首先來八股文一下好了，這次的論文能夠完成真的要非常感謝我的指導老師廖峻鋒教授給了我非常多的研究建議與討論，如果不是有老師的幫助我現在可能連半本論文都沒辦法完成，除了技術上的指導外還有每周 meeting 的盯進度都讓你會因為良心的譴責而試圖生出一下進度出來報，還有老師所提供的區塊鏈計畫也讓我學到不少有趣東西，也認識了許多高手，與大大智豪學長一起投稿的 TCSE 也是一場難忘的經驗(到嘉義旅遊 XD)。技術指導固然重要但我覺得收穫更大的是老師在 meeting 上所分享的一些待人處世的態度，以及工作技能外的軟實力培養的重要性，在論文的寫作指導上我也受到很多幫助，永遠沒辦法把話講清楚的我重修論文好幾次，老師還是不厭其煩地糾正，峻鋒老師真的是教授界裡的一股清流，教誨如春風、師恩似海深，祝福老師桃李滿天下。

在這兩年的時間裡我在 Lab 認識了很多大大，雖然大部分的人對本論文的產出沒有太多貢獻，但是沒有這些人我的研究生涯或許會過得挺無聊的(我可能會悶死在宿舍)，感謝俊安和宇凡大大的技術指導，在我剛進入 Lab 時幫助我適應新的環境，祝福你們能在資訊領域裡成為大神飛黃騰達，感謝昀臻大大每天抽空陪我在 Lab 裡講幹話，為 Lab 增添各式各樣的歡笑，祝福你的愛情能夠一帆風順。玠忠、奕修、韻淇、雅雯的學弟妹們這一年來和大家相處都很開心，雖然大家都超級安靜的 XD、也祝福各位能夠順利畢業!!，說真的這兩年真的又太多體驗了也認識了很多瘋狂的人就光是在街頭練習搭訕就是很新奇的體驗更不要說去夜店和酒吧那接經歷了，跨出自己的舒適圈給自己更多新的刺激點應該就是我現在最需要的別再繼續宅在家裡了。

最後給未來的自己一些期許好了，恐懼是天生不可避免的害怕未知是天性使然，我知道你有太多對恐懼的想像而造成自己不敢去行動，有很多時候那些恐懼都是虛假的若是不實際去嘗試那些害怕就是真的。我知道你有很多很棒的想法以及擁有嘗試一切的能力，而你最大的敵人就是害怕後的不行動，真相或許並沒有你想像的那麼糟糕，你也沒有自己想像的那樣差勁，想知道真相嗎?去收其他人的回饋吧，勇敢的去請教其他人吧。在街搭的挑戰中你已經實際體驗到了那些想法皆是自己內心的聲音皆是恐懼使然。你可是有無限的可能性的，相信自己，面對恐懼直視問題，你就會發現他沒那麼的可怕一切都是自己嚇自己而已，如果你能夠克服這點，我保證你已經成功了，獲得自己想要的生活了。人生是一場體驗，得到的感悟越多生命就越有意義，對了別忘記把這些體驗全部記錄下來喔!! (ᵐᵐ ºωº ᵐᵐ)，祝福你早日尋到真愛!!

最後送你一句話：世界是一面鏡子你對她笑他就對你笑

DOI:10.6814/NCCU202101070

目錄

DOI:10.6814/NCCU202101070

DOI:10.6814/NCCU202101070

圖目錄

DOI:10.6814/NCCU202101070

DOI:10.6814/NCCU202101070

第一章 緒論

1.1 研究背景

由於近年來雲端運算和大數據應用的普及化，要取得資料已經不像過去一樣困難，相對的人們對於網路的需求也更加龐大。例如：雙 11 單身購物節會在短時間內吸引大量使用者對購物網站系統進行瀏覽和交易，此時傳統式三層式架構將往往因大量涉及交易的資料寫入與讀取動作，造成大量資料被鎖定(Lock)，導致系統效能低落或單點失效系統當機。近年來許多企業利用 CQRS (Command Query Responsibility Segregation)[1]概念來緩解此問題，其核心概念是將「寫入(Command)」和「讀取(Query)」的交通分流，降低資料大量讀寫時所造成的效能低落。另一方面，近年來由於網路普及化大多數的人都會選擇在線上購物查詢資料處理生活中的大大小小事情，進而演變成許多企業的軟體系統必須有能力處理更多更複雜的商業邏輯，越複雜越龐大的系統維護起來就更加費勁，也因此微服務的概念從玄學漸漸地浮現至大眾的視線中。雖然微服務的設計架構正在快速普及化，但是在設計微服務架構時，分散式系統必須對於一致性（Consistency）、可用性（Availability）和分區容忍性（Partition）中做出取捨，另外在分散式的非同步環境下，資料一致性問題就成為不可被忽略的重要議題。

面對資料一致性問題 Event Sourcing 可能成為有效的解決方法，Event Sourcing 又稱為事件溯源，其主要機制是透過將所有對系統下達的指令轉換成事件型式，並將其全部儲存在事件資料庫(Event Store)，當系統出現不可預期的問題時，就可以透過 Event Store 的眾多事件進行回推作業，將像是時光倒流一樣，由於所有的操作過程皆被記錄下來，所以進行回溯並非難事，本研究透過 Event

DOI:10.6814/NCCU202101070

Sourcing 事件溯源概念創建一個具有權威的事件資料庫(Event store)，並透過這個資料庫將可以保證其他服務的資料保持在最終一致性，用以解決可能存在的資料一致性問題，再結合 CQRS 的分流機制開發出了一套基於歷史回溯機制的自動還原系統。然而，到目前為止，大部份 Event Sourcing/CQRS 架構在實際維運時，仍存在許多議題有待處理:

- **Event Sourcing/CQRS 基礎設施的穩定性:** 在設計微服務的自動還原系統當中要如何做到監控各自系統的健康狀況就成為了一個很重要的議題，現階段較為常見的監控方式就可分為兩類 1.監控系統主動(pooling)的向各個服務進行健康狀態的確認，不過這種方式就會增加系統連結的負擔。2.被動式接受服務給予的 heartbeats 訊息，此方法可以降低連結負擔但是相對的判斷服務是否存活就必須依靠時間來決定，這樣的方法會增加監控系統誤判服務是否失效的正確性。就目前常見的監控系統當中皆為集中式監控，以一個監控系統為核心來維持服務的穩定性，這樣的設計可能會遇到單一失效的問題，若是核心監控系統失效了，將無法繼續維持系統監控的穩定性，更不用提到要做到自動修復的功能。
- **Event Sourcing/CQRS 基礎設施失效之緩衝隔離機制:** 若想要增加系統的穩定度以及可用性就必須加入一些緩衝機制，當系統很不幸的發生失效的情況時，這個緩衝機制會暫時避免整個系統馬上失效，這也算是給各個服務的保護措施，透過這個機制也可有效增加系統處理前端需求的總吞吐量，而這項設計本研究將會利用斷路器機制(Circuit Breaker) [2]，它的原理和在電子電路當中常見的保護概念類似，可在部份系統發生災難時，提供緩衝隔離，避免其它部份系統失效。
- **避免讀寫分離所造成的 Race Condition:** Event Sourcing/CQRS 系統雖然增進了效能，但進行交易機制(Transactions)時，在系統對相同一群資料進行高密度的寫入時，會經常發生 Race Condition，這種情況下就有可能造成交易未

DOI:10.6814/NCCU202101070

寫入的發生，例如:有兩個人要搶五月天的票，在最終一致性的資料庫情況下他們兩個人都會看見只剩一張票，當兩人同時下單購買時，勢必會有一個人的指令被放棄掉，而這樣的情況在 Event Sourcing 系統當中是有可能發生為了解決這樣的問題，本研究所設計的 WatchDog 監控系統就會透過一些機制進行重新寫入已確保每筆交易都能成功被執行。

本研究將會定義系統失效(failure)：當節點在固定時間內無法對監控系統送出 heartbeats，並且對監控系統所送出的要求無法提供任何回應，亦或是監控系統監測到節點內部發生致命錯誤，無法發揮正常作業時都會被認定為失效。

1.2 研究動機

根據上述所講的情況，電商的購物平台時常會遇到短時間內大量的使用者，面對這樣的問題直觀上可以從兩個方向著手，第一種是從硬體上水平擴張增加伺服器的流量承受度，而另一種方式就是透過 CQRS 的方式分流 Command 和 Query 的使用者，此方法主要是應付大量 Query 使用者的流量，並且降低一直進入 Event store 改寫的動作，期望透過降低改寫次數能有效降低整個系統的流量負擔。

就本研究目前所知，目前較少在 Event Sourcing/CQRS 面向有關 failure detection 和 failure recovery 的研究和實作探討。所以本研究希望透過設計 CQRS 和 Event Sourcing 並且加上利用 Event Sourcing 的方式嘗試完成一個能夠做到 failure recovery 與 failure detection 機制，並同時設想加入擁有能達成集體共識的監控系統已增加整個系統在面對節點失效時的穩定度。

本研究透過 Event Sourcing/CQRS 所設計出來的網路購物平台的模擬系統，並從中發現一個靠微服務設計概念所設計出來的系統是需要一個自動化機制來穩定整個系統的運作，然而因分散式的特性想要做出這樣的機制並非是一件容易的事情，所以本研究希望以以下三個指標功能作為穩定機制代表包含 1.自動監控健康狀態 2.錯誤偵測 3.錯誤修復這三點特性。

DOI:10.6814/NCCU202101070

本研究透過 Chris Richardson 在 Microservices Patterns[1]此書中的設計思路，加入 Event Sourcing / CQRS 等設計概念來實驗並且測試由 Event Sourcing / CQRS 所設計的系統是否可以應付大量使用者對系統的負擔，並且加入自動偵測錯誤及自動修復功能及緩衝機制來自動修復系統所偵測出的錯誤。

1.3 研究貢獻

本研究的貢獻主要希望能夠解決三點問題 1.強化集中式監控系統可能造成單一失效問題 2.增強因單一失效而造成崩潰的分散式系統可用性 3.排除 Event Sourcing/CQRS 系統交易中可能出現的 Race Condition，除了解決這三個問題之外，本研究所設計的系統也會包含一些限制條件的自動修復功能，不過主要的亮點還是在解決前面那三大問題來達到整個系統的穩定度及提高自身的可用性。

1.4 論文架構

本論文的第二章主要會介紹關於 Event Sourcing/CQRS 的系統設計下的相關研究以及論文，當然也會包含錯誤回復及錯誤偵測，第三章將會解說關於本研究所使用的相關技術包含共識演算法的實現原理，Circuit Breaker 機制、CQRS 和 Event Sourcing 的實現方法及優勢，最後會帶到本研究分散式系統當中使用 MQTT (MESSAGE QUEUING TELEMETRY TRANSPORT)來進行訊息傳送，本論文的第四章節將會實際帶入本研究的實作架構包含 WatchDog 機制如何做到集體共識以及錯誤偵測，錯誤還原機制，Circuit Breaker 的運作流程模式解說，最後第五章將會帶入實驗設計以及實驗數據驗證本研究的設計是否有效，最後寫出本研究所得到的結論。

DOI:10.6814/NCCU202101070

第二章 技術背景與相關研究

2.1 命令與查詢責任隔離(Command Query Responsibility Segregation)

在傳統架構下的系統，修改以及查詢的指令皆是透過簡單基本的 CRUD 進行運作，在這模式下一般的系統執行時皆會得到不錯的表現，在撰寫軟體方面也是非常簡易直觀，但若是在更加複雜的應用程式下運行的話透過此方法就可能會變得相當不便，舉例來說:網路購物平台平常的時候人流量沒那麼大，上網查詢和下單指令都透過同一個窗口是可以應付的，但是若遇到雙 11 這樣特別的網購節日時一瞬間大量使用者的湧入是極有可能造成系統無法處理那麼多訊息，進而導致失效或是當機，在這樣的問題下就可以透過兩種方法進行修正，其一就是對硬體的加強，增購伺服器加強系統能夠應付的總處理速度上限，然而這個方式的缺點就是需要花更多的金錢來擴充硬體設備，而由於上述短時間急遽大量的使用者湧入的情況通常只會發生在一至兩天的時間中，擴充出來的效能也會因此浪費，就好比現今台灣發電廠的過度產電和浪費。

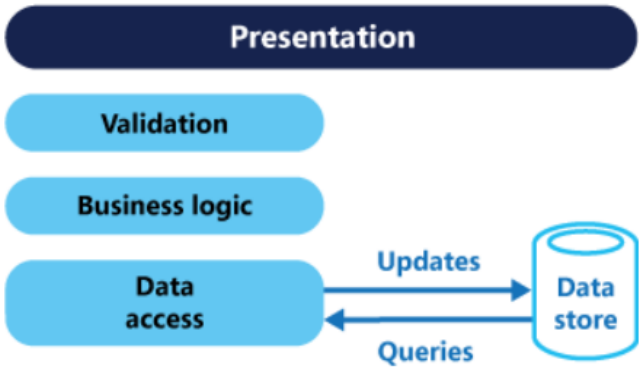


圖 1：傳統架構下的系統樣態[3]

DOI:10.6814/NCCU202101070

其二就是由軟體層面著手進行改進，在 1987 年 Betrand Meyer 的著作[4]中提到關於命令與查詢分離這套概念(CQS, Command Query Separation)，其原始概念是我們可以把物件操作分為命令（Command）和查詢（Query）兩種形式：1.修改命令(command):執行後將會改變物件的原始狀態 2.查詢(Query):查看物件的結果，此操作不會影響到物件的狀態換句話說就是只能看不能用。不過 CQS 只有在此書中描述的一種思維想法，而其背後最主要的觀點是一個方法應該改變一個狀態或是回傳一個結果，而不是兩者並行同時擁有，在此就呼應了書中[5]所提出的物件導向設計五個原則(SOLID)的其中一項 SRP(Single responsibility principle)的概念：「一個模組應只對唯一一個角色(actor)或利益相關者負責」不謀而合。

後來就由 Greg Young 在他自己寫的書籍[6]當中以此概念作為基礎提出了 CQRS(Command Query Responsibility Segregation)，將 CQS 的概念從方法論提升至模型的層面換句話說就是將上述的概念融入進系統的設計當中，此模型運行的方式是將 command 和 query 作為個別的服務並且分開進行運作，透過這樣的方

DOI:10.6814/NCCU202101070

法可以有效分流使用者們對資料庫的使用以及處理，大部分瀏覽網頁的人就會走 Query Mode 的面向，而真正需要下單購買的人就會走 Command Mode。

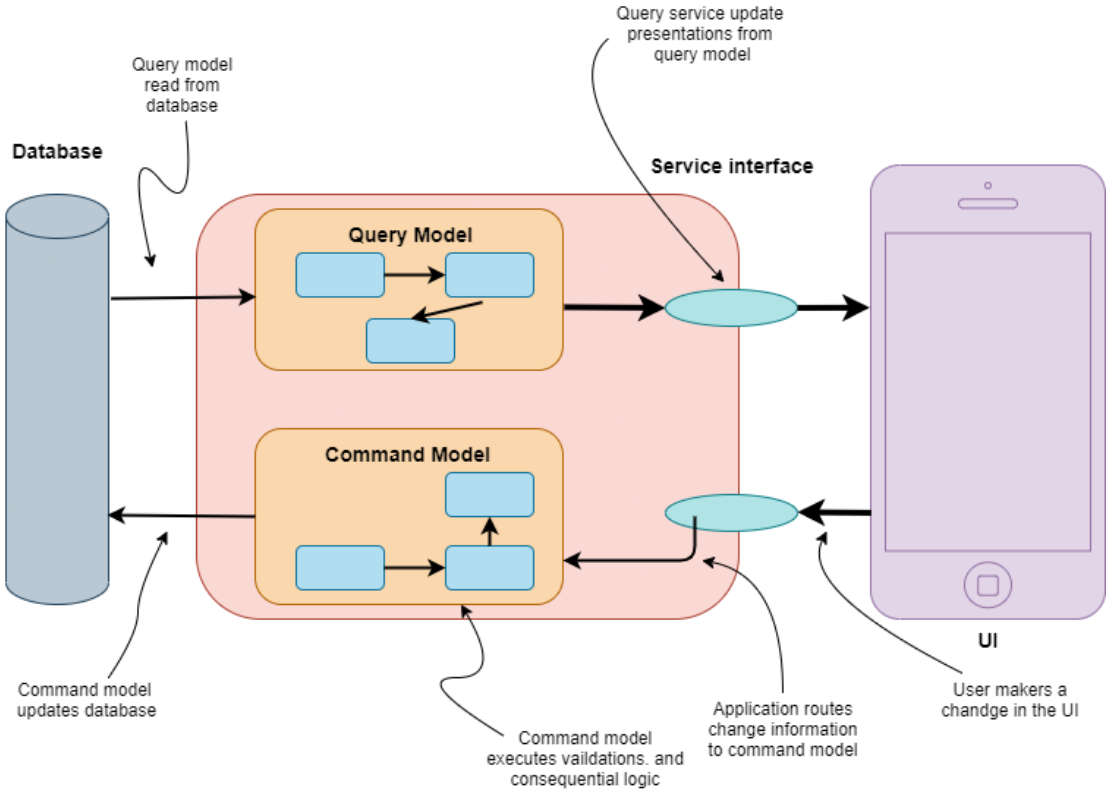


圖 2：CQRS 系統設計樣式(單一資料庫)

2.2 Event Sourcing

Event Sourcing 也叫做事件溯源，這個名詞是由 Martin Fowler 所提供的一種架構樣式，此系統有以下幾個特點 1.整個系統的運作皆是由事件驅動(event-driven system)完成的 2.事件的紀錄是最高級的，系統的數據皆以事件為基礎，而這些事件必須以某種形式加以保存。Martin Fowler 在關於 Event Sourcing 的文章中[7]是以船運作為使用 Event Sourcing 的範例，面對一般概念的資料儲存皆是以改寫資料庫的欄位為主，然而這個方法我們只能得到最後的結果，並無法得知任何中間過程的紀錄，就像是運送貨物的船我只能看見其最後到達的位置，但是無從得知中間過程停靠過那些港口，而這種歷史回溯機制在某些環境下是非常重要，例如智能合約的交易稽查機制，虛擬貨幣的交易流向等等。

DOI:10.6814/NCCU202101070

而 Event Sourcing 之所以會越來越受到關注的主要原因有以下幾點:

一、溯源與歷史重現簡單，不須另外操作。

二、保存修改資料的動作並轉換成事件，可以有效率知道目前系統的狀態以及系統是如何演變成現今的情況，在小型商業邏輯的情況下可能沒什麼感覺，但是在比較大型複雜的商業邏輯就會有強大的效果，即使利用現今所資料庫所擁有的 log 機制提供查詢分析也很難追溯其演變過程，這也是使用 Event Sourcing 的模型好處之一

三、事件回溯就像是跟五維度的生物看四維生物，時間的軸向是可以控制的，意思也就是說我可以來回穿梭在過去系統運行任何一個時間點甚至能夠複製跟分析歷史發生的過程。

四、若是發現一個錯誤時，可以直接擷取錯誤前區間的數據並進行修復，快速有效率，若是依照傳統的設計方法，修改錯誤得使用 SQL 指令並且手動一個一個修，若是這個錯誤存在的時間點較長且牽扯到的數據很龐大的話，這次的錯誤修復將會是一場噩夢。

五、透過事件回溯有助於發現問題的起始點。

六、在 Event Sourcing 設計下，事件的存取是一個一直疊加的新增事件群，並不用一直不斷更新修改的需求，有點類似像 stack 的概念，只需加入不用直接對資料庫進行修正。這樣的寫入方式可以在很多情況中提供較為優秀的修改性能，也可透過此操作強化系統中的事件吞吐量。

如果要想開發一個基於 Event Sourcing 設計模式下的分散式系統，最有效的嘗試就是與 CQRS 設計架構進行合併。修改指令的服務接收 Command 請求，觸發後端的商業邏輯服務，更新交易數據。然後再將這些事件透過 MQTT (MESSAGE QUEUING TELEMETRY TRANSPORT)發送。讀取服務監聽事件的發送，獲取事件，更新相對物化視覺圖的數據。同時所有的 Query 請求都由讀取

DOI:10.6814/NCCU202101070

服務處理並返還。 透過這樣的設計對於修改服務，它所接觸的數據都將會只有事件，是一個寫入操作的過程，事件寫入的方式也能夠提供較為方便的事件查詢系統。對於讀取服務來說，我們又可以部署多項服務，進一步提供大量使用者數據查詢的效能。如此可見，透過這樣的設計將能大大提高整個系統的性能。

2.3 Circuit Breaker

在分散式環境中，對遠端資源和服務進行的呼叫可能會因為暫時性錯誤而失敗，例如低速網路連線、逾時、資源過度認可或資源暫時無法使用等等。這些錯誤通常會在短時間內自行修正，而強大的雲端應用程式則會使用重試模式來準備處理這些錯誤。不過也可能發生因為非預期事件導致的錯誤，這可能需要更長的時間來進行修正。這些錯誤的嚴重性範圍包含失去部分連線到整個服務無法運作。在這些情況下，持續重試不太可能成功的作業是無意義的，相反地，應用程式應快速接受作業失敗，並且盡快處理此失敗事件。

此外，若是服務非常忙碌，則系統中某一部分的失敗可能會導致階層式連續失敗就像是骨牌效應一樣。 例如：被呼叫的服務可設定為能實作逾時，並在該服務無法在此期間內回應時，回覆失敗訊息。 不過，此策略可能會導致同個作業的許多並行要求遭到封鎖，直到逾時期限到期。這些已封鎖的要求可能會佔據重要的系統資源，例如記憶體、執行緒、資料庫連線等等。 因此，這些資源可能會被耗盡，導致其他必須使用相同資源但可能不相關的部分系統作業失敗。在這些情況下，作業最好能立即失效，並只嘗試呼叫可能成功的服務。 這邊要特別注意的一點，設定較短的逾時可能有助於解決此問題，但逾時不應該過短而造成作業一直失敗，即使服務要求最終會成功。

Circuit Breaker pattern [8]最先是由 Michael Nygard 在著作[2]中被提到，此模

DOI:10.6814/NCCU202101070

式可防止應用程式重複嘗試執行可能會失敗的作業。 過程中一旦判斷該錯誤會持續較長時間時，該模式會讓應用程式繼續執行，而不用等候錯誤修正或浪費 CPU 循環。 Circuit Breaker 也可讓應用程式偵測錯誤是否已解決。 如果此問題已獲得修正恢復正常，應用程式就會嘗試重新連結使用。

Circuit Breaker 的用途與重試模式不同。重試模式會在預期作業會成功的情況下，讓應用程式重試作業。 而 Circuit Breaker 則是會防止應用程式執行很可能會失敗的作業。 應用程式可以結合這兩種模式，透過 Circuit Breaker 使用重試模式來叫用作業。Circuit Breaker 有以下三種模式:

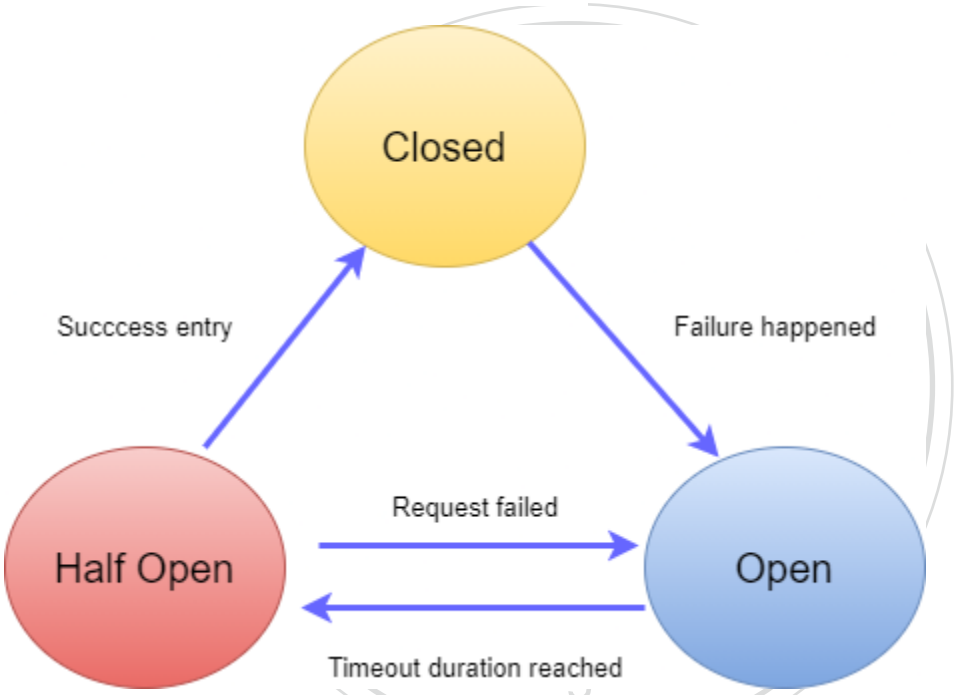


圖 3：Circuit Breaker 模式切換狀態

一、**Closed Mode**： Circuit Breaker 會更新最近的失敗計數，如果作業呼叫不成功時，Circuit Breaker 就會增加此計數。 如果最近的失敗數量已超出給定時間內的指定臨界值，則 Circuit Breaker 會轉換至 open mode。 此時的 Circuit Breaker 會啟動逾時計時器，當此計時器過期時，Circuit Breaker 會置於 Half-open mode 。逾時計時器的用途是讓系統有時間修正造成失敗的問題，然後才能允許應用程式嘗試再次執行作業。

DOI:10.6814/NCCU202101070

二、**Open Mode** ：來自應用程式的要求會立即失敗，並傳回例外狀況給應用程式。

三、**Half-open Mode**：允許來自應用程式的有限要求數量通過。如果這些要求成功，則會假設先前導致失敗的錯誤已修正，而且 Circuit Breaker 將會切換成 Close 狀態 (失敗計數器會重設)。 如果有任何要求失敗，Circuit Breaker 就會假設該錯誤仍然存在，因此還原成 open 狀態，並重新啟動逾時計時器，讓系統有更多時間從失敗中復原。Half-open mode 有助於防止復原服務時突然出現過多要求。在服務復原時可以支援有限的要求量，直到復原完成，但當復原正在進行時，大量的工作會造成服務逾時或再次失敗。

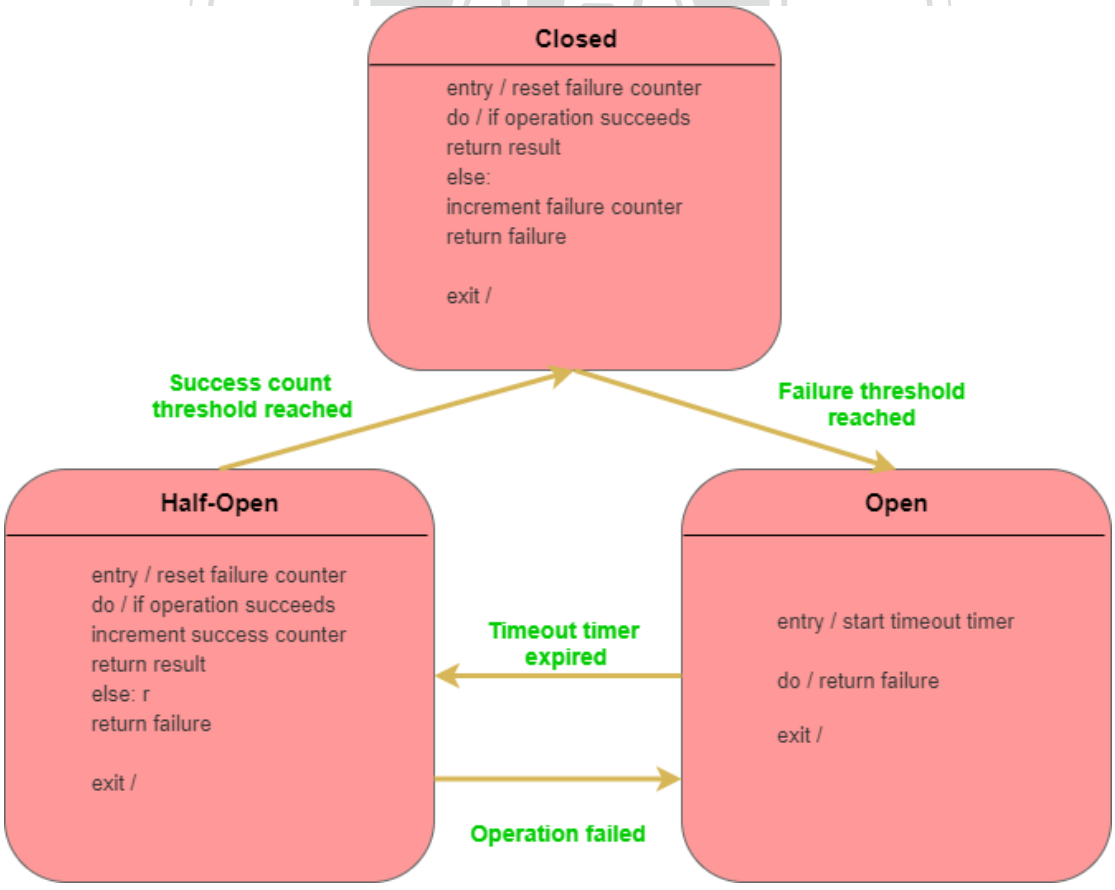


圖 4：Circuit Breaker 虛擬碼呈現

DOI:10.6814/NCCU202101070

2.4 Message-Oriented Middleware(MOM)

訊息導向中介軟體(Message-Oriented Middleware, MOM)是中介軟體中應用最廣泛的類型，適合應用在分散式系統間的訊息溝通。而與其他類型的中介軟體最大不同之處在於，MOM 採用鬆散耦合(Loosely Coupled)模型設計，透過非同步溝通的特性，不僅增加非同步系統之間整合的彈性，也增加溝通效能。例如：Publish/Subscribe Pattern 是一個可以將一份訊息，並同時送交到數千甚至數百萬的接收端，有助於高傳輸量(High Throughput)與低延遲(Low Latency)的應用

在訊息傳送的過程主要會有三種物件組成：1.傳送節點：負責傳送訊息 2.接收節點：訊息的接收端 3.訊息導向中介軟體（Message-Oriented Middleware, MOM）：中間訊息傳送媒介。整個運作流程如下所敘述，傳送節點會負責與 MOM 建立通道並且傳遞訊息，而接收節點會從通道中取得由傳送節點送出的訊息，傳送節點會將訊息傳送至 MOM，接收節點則會從 MOM 接收到訊息，而 MOM 將成為訊息的中間交換場所，其本身並不處理任何接收到的訊息，而是確保能提供雙方訊息的送達、負載平衡、容錯、以及交易的好仲介商。

MOM 最大的特色在於傳送節點和接收節點之間，並不需要透過任何方式知道彼此的存在，甚至連對方斷線也不會造成任何問題，MOM 回提供一個訊息的傳送通道，雙方節點只需要知道這個通道是什麼在哪個位置即可，當雙方節點已經連結到 MOM 所提供的通訊通道時就可以進行數據傳輸，這樣的模式會造成傳送節點與接收節點成為鬆耦合，任何一方進行修改或是更新重新連結時，都不會影響另外一方節點的運作，此外 MOM 也會提供非同步模式，讓訊息傳送後無需任何等待時間，節點可以繼續自己的流程。在訊息的傳送模式中主要以出版-訂閱（Publish-Subscribe）模式為主。

DOI:10.6814/NCCU202101070

Sub/pub 模式：

在 sub/pub 模式下，傳送節點將設為 publisher，而接收節點會被定義為 subscriber，MOM 提供的通道稱為主題(Topic)，傳送節點將訊息 pub 至 topic 上，接收節點訂閱感興趣的 topic。

Sub/pub 模式 publisher 採用 Push 模式，只要 topic 上有新訊息，subscriber 將馬上收到訊息，訊息發佈時，subscriber 不一定會在線上也不一定要在線上，神奇的是一旦連上線，就會馬上收到訊息在，而在本研究當中服務與服務之前的溝通手法就是透過 Sub/pub 模式進行溝通，最主要的目的就是需要 MOM 所擁有的耦合性。

。

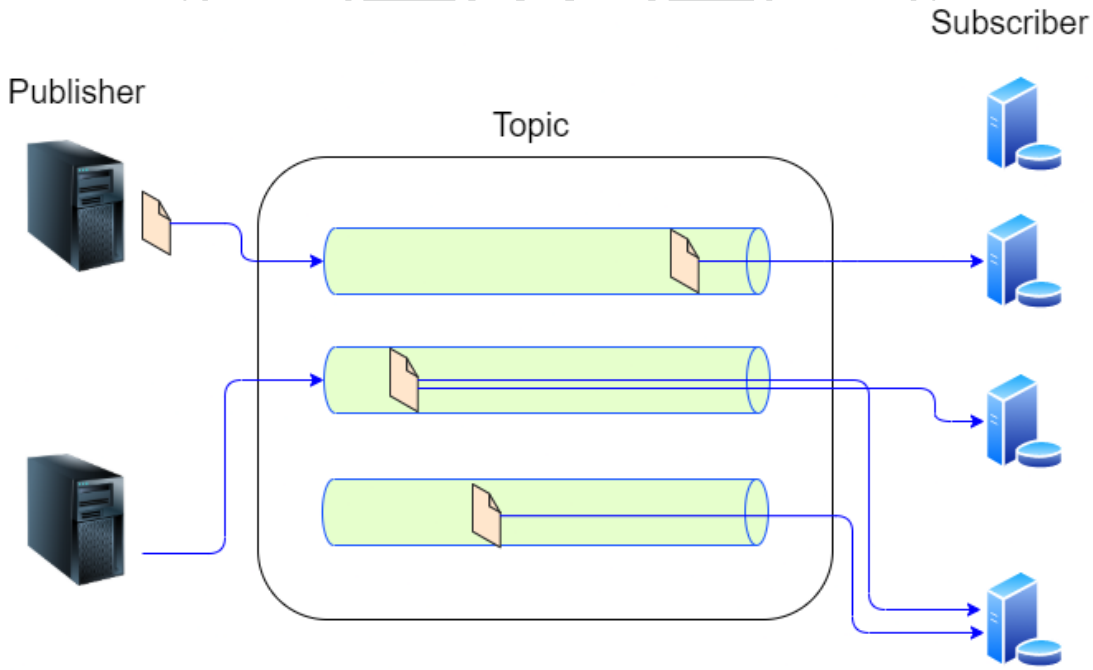


圖 5：Sub/pub 模式

DOI:10.6814/NCCU202101070

2.5 Raft 演算法

FLP 不可能原理是由 Fischer，Lynch 和 Patterson 三位科學家於 1985 年發表在論文[9] 中提出並且證明了在一個可靠網絡中，若是在允許節點失效（一個也算）的最小化非同步模型系統中，絕對不存在一個可以解決一致性問題的共識算法，簡單來說就是告訴了我們不應該浪費時間，試圖為非同步分散式系統設計一個可以在任何場景中都能使用的共識算法。實際上在面對允許節點失效情況下，純粹使用非同步系統是無法確保在有限的時間內能夠完成共識。即便是在非拜占庭將軍問題的情況下，也包括 Paxos、Raft 等演算法都無法在極端環境下達成共識，但是在工程的實踐中要出現極端環境的機率近乎微乎其微。即便是有 FLP 定理的提出，就工程學的角度來看，加上一些條件限制，在特定情況與環境下，還是可能實現分散式共識，也因此有誕生出了 CAP 原理[10]。

CAP 原理[10]的主要核心說明是分散式系統無法同時達成一致性（Consistency）、可用性（Availability）和分區容忍性（Partition）這三個特性，所以在設計中往往需要弱化對某個特性的需求，以達到在工程上面的需要。這三項具體說明如圖 6：

DOI:10.6814/NCCU202101070

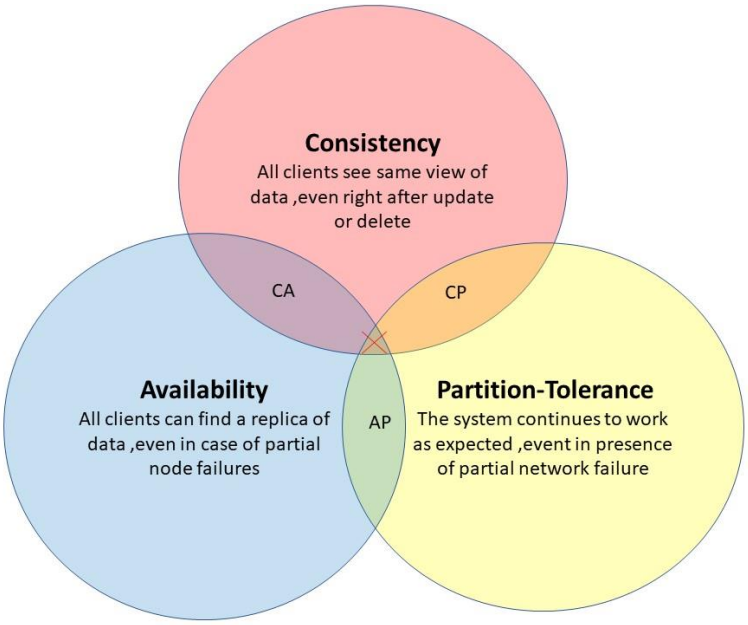


圖 6：沒有能夠完美符合三種條件下的系統

1. 一致性（Consistency）：任何事件應該都是原子的，所有副本上的狀態都是事件成功提交後的結果，保持強一致性。
2. 可用性（Availability）：系統（非失敗節點）能在有限時間內完成對操作請求的應答。
3. 分區容忍性（Partition）：系統網絡中可能發生分區故障（成為多個子網，甚至出現節點上線和下線），即節點之間的通信之間無法獲得保障。而這樣的網絡故障不應該影響到系統的正常服務。

在 CAP 定理認定中，分散式系統最多只能保證三種特性中的兩種特性。例如：當網路出現分區時，讓系統同時保持一致性與可用性是不可能的。其一可能是節點收到請求後，因為沒有收到其它節點的確認而不回應（犧牲可用性），其二可能是節點只能回應非一致的結果（犧牲一致性）。由於大部分時候網路都被預設為穩定的，因此網路系統基本可以提供令人安心的服務；若是當網路發生不穩定的情況時，系統就可能犧牲掉一致性（大多數都會選擇這個），不然就是犧牲掉可用性。像是 Event Sourcing/CQRS 的設計概念就是弱化一致性，可以允許

DOI:10.6814/NCCU202101070

在新版本上線前數據的不一致，需經過過一段時間後，資料庫才可以完成同步，但是期間內不保證一致同步。而像是 Paxos、Raft 的演算法就是弱化可用性，在演算法中可能存在著一些無法提供可運作節點的情況，另外也同時允許少數節點離線，但是可以達成所有節點的強一致性。

共識演算法主要應用於分散式系統中，一個集群中有多個節點組成，讓每個節點都維護相同的狀態，例如：在多種系統中都需要集群有單一個 Leader 存在，所有節點都必須承認這一個 Leader，否則多個 Leader 可能會導致 Split brain 等資料不一致等問題；但如何讓節點狀態一致是一件不簡單的事情，要考慮到節點可能失敗 / 網路封包延遲等等。

Raft 演算法是由 Diego.O and John.O 這兩位史丹佛大學的教授所提出來的一套由 Paxos 衍生出來的演算法，共識演算法 Paxos 之所有無法使用的主要原因，還是因為它太過於複雜，世界上真正了解的人沒有幾個，市面上的實作案例也分歧出非常多的不同實作版本，理論太過艱深導致實務上有很大的落差，所以 Raft 在設計時的一個核心理念是好懂，接下來將會詳細介紹關於 Raft 是如何在分散式系統中讓多節點在容忍錯誤下達到強一致性。

通常在 Raft 的集群中至少要由五個節點組成，可以容忍兩個節點失敗，而每個節點都將會有三種狀態 leader 、follower 、 candidate，通常情況下只會有一個 leader 其他人都是 follower。

- Follower： 被動的處理來自 leader 或 candidate 的請求。
- Leader：負責所有 client 的 request，並複製指令到 follower 中
- Candidate：follower 發現沒有 leader，設一個隨機 timeout 切換成 candidate 模式，準備要選新 leader。

在下面的示意圖當中可以知道 Followers 只接收來自 leader 的訊息，若是沒

DOI:10.6814/NCCU202101070

收到 Follower 就會轉變 Candidate 做選舉模式的準備，最後若是 Candidate 受到的回應最多時他將會成為新 Leader。

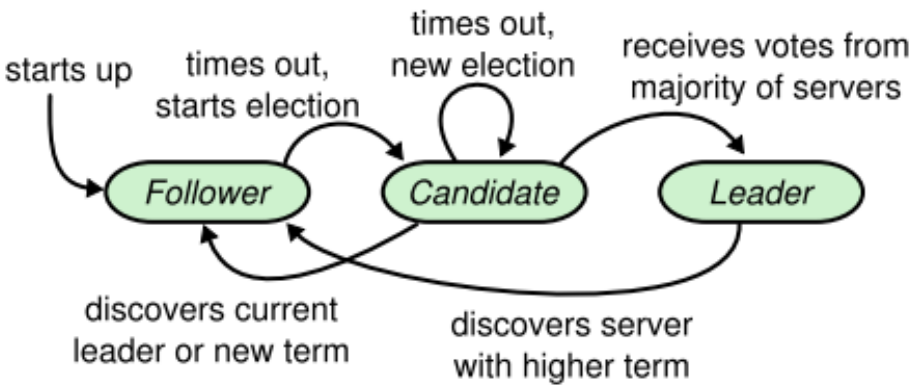


圖 7：角色轉換關係圖[11]

Raft 演算法會將時間切割成回合數(terms)，每一個回合代表著一次的選舉，也就是 candidate 去競選 leader 的過程，如果成功選出 leader 後，則每個節點紀錄這一個回合數(terms)；如果選舉失敗，則開啟新的回合直到有人成為 leader。

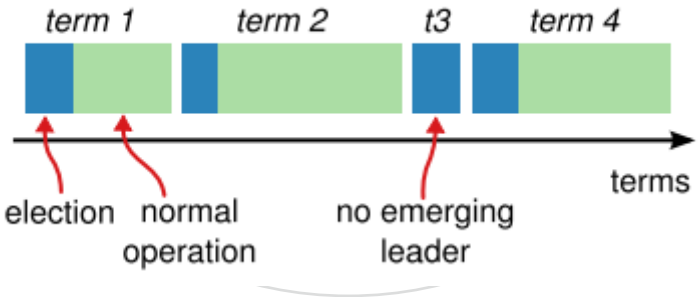


圖 8：回合機制[11]

回合(terms)本身是一個遞增數值，用來取代邏輯上的時間概念，在一些情況下有可能節點所觀察到的回合數跟其他節點不同，若是發生這樣的情況回合數小就是代表自身的資料過時，必須更新自己的回合數；例如說原本的 leader 可能斷線，其他節點推選出新的 leader 則會進到下一個回合數，原本的 leader 回歸後發現自己的回合數竟然比其他的 Follower 都還要小，這時就會主動變成

DOI:10.6814/NCCU202101070

Follower 如果節點收到請求時，發現回合數比自己小，則代表請求過期，將會直接拋棄該請求。

Raft 將會透過 heartbeats 來觸發 leader 選舉，當 leader 選上時，會在固定時間內發內部裡沒有 log 的 AppendEntries RPCs 的 heartbeats，如果 follower 超過 election timeout 後還是沒有收到來自 leader 的 heartbeats，則會直接進入選舉階段。Follower 會將現今的回合數加一後轉變為 candidate，並且在同一時間請求其他 Follower 透過 RequestVote 投票給他，在遇到以下三種情況 candidate 才會改變狀態。

1. 贏得選舉
2. 其他節點成為 leader
3. 超過一定時間都沒選出 leader

如果 candidate 拿下過半 Followers 的票數，其將成為新的 leader，而每一個 Follower 在同一個回合數中只會投票給請求最先到的那位 Candidate，設為了要避開選舉無效的情況發生，如果 Candidate 成為 leader，則開始發送 heartbeats。

如果 candidate 階段收到 heartbeats(AppendEntries)，則代表目前已經有了其他 leader 產生，這時 candidate 會去比對回合數，若是回合數大於它自身的回合數則轉變成 Follower，反之則繼續維持 candidate。

若是超過一定時間內還沒有選出來的話，timeout 後開始下一輪新的選舉 Raft 演算法會透過在一定時間內隨機 timeout (150-300ms)，避免所有的 Follower 同時進入選舉階段，這樣能加速 leader 的推選。

DOI:10.6814/NCCU202101070

2.6 相關研究

隨著近年來微服務 (Microservice) 架構日漸普及，許多相關架構設計的議題也逐漸受到重視，其中 CQRS 架構以容許短暫的資料 (讀取) 不一致性，將讀取與寫入的資料儲存處切分，來大幅提升讀取效能的優勢被廣為人知，而 CQRS 只是一個設計概念，其具體實現方法視運用場合而不同。

歷年來對於 CQRS 和 Event Sourcing 的相關研究大致可以分為兩類，一類討論了此架構下的效能分析以及優劣勢，其中以 Long [3] 點出了 CRUD 在效能上的四個缺點，內容主要是在講述當 CRUD 面對大量資料讀取時，是有很大的機會產生單點失效和效能不佳的問題，而 CQRS 可以有效分流讀寫端的吞吐量甚至 可以單獨強化兩端的資料負荷量，在 Miciel Overeem 等人[12]也提出在使用 Event Sourcing 儲存 event 時通常會以 schema-less data 的方式儲存，這樣的儲存 方式對於資料在轉換和事件更新的方面有些不足的地方，而作者在本論文中有提出幾個方法來做到更加高效率的資料轉換和事件更新，在分散式的 Event-sourced Systems 中並沒有一種系統性的效能檢測法用來評估系統，而 Dominik Meißner 等人[13]提出了六大步驟的方法論(Methodology)讓設計者評估自己的分散式系統的可用性。Jedrzej Rybicki[14]則是直接使用 Kafka 進行實作，實際探討使用 Event Sourcing 後可以得到什麼樣的後果和好處。第二類是實務場景應用，其中以 Han 與 Choi [15]對將 Event Sourcing 應用在 V2X 上進行事件回溯應用，希望將事件回溯機制應用在 5G 自駕車的技術上，將自駕車感測出來的所有事件全部儲存在透過機器學習的方式讓資料可以有效利用，而 Zhong 等人[16]將 Event Sourcing/CQRS 應用在交易系統裡面，在論文當中也可以看出在面對大量的使用者下 Event Sourcing/CQRS 系統的效能會明顯大於 CRUD。就目前所知，並沒有將 CQRS 與 Event Sourcing 的錯誤偵測(Failure Detection)與自我回復機制(Failure recovery)的相關學術文章發表，也沒有針對 Event

DOI:10.6814/NCCU202101070

Sourcing/CQRS 的架構設計出相關能夠提高系統穩定性的機制，透過以上的敘述本研究希望在設計出一套增強 Event Sourcing/CQRS 系統穩定程度的一連串監控、偵錯、復原機制。

DOI:10.6814/NCCU202101070

第三章 系統設計

在第一章的研究背景當中提到過本研究所要解決的三大問題: 1.強化集中式監控系統可能造成單一失效問題2.增強因單一失效而造成崩潰的分散式系統可用性3.排除 Event Sourcing 系統交易中可能出現的競爭危害，延續以上三點問題本研究將會在本章節詳細說明解決方案及設計考量。

本章節將會分成四個小節，第一小節將會是本章的總論，此小節會詳細說明本研究所模擬出來的微服務環境下全部的元件(component)以及其功能。4.2 將會敘述如何解決問題一：強化集中式監控系統可能造成單一失效問題的有效方法以及設計概念，透過看門狗監控系統(WatchDog)來達到類集中式監控機制。4.3 將會敘述如何解決問題二：增強因單一失效而造成崩潰的分散式系統可用性，本小節將詳細講解 Circuit Breaker 設計思維以及為什麼這個設計能夠增強整個分散式系統的可用性。4.4 則是敘述如何解決問題三：排除 Event Sourcing 系統交易中可能出現的競爭危害，本小節會說明在實作 Event Sourcing 當中發現的交易競爭危害現象，此現象的發生將可能造成交易事件的丟失。最後一小節則會詳細敘述當監控系統發現失效服務時如何做到錯誤回復機制，自動修復失效服務。

3.1 系統架構說明

事件導向微服務(Event-Driven Microservices)[17]的主要思維是讓每個服務都能達到透過事件接受來完成獨立自主的功能，並且降低服務與服務之間的耦合性。每個服務當中都會需要以下基本元件(在 Service 當中的單元都稱為是 component)。本研究將會實作一個模擬網路交易平台的微服務環境，搭配上 Event Sourcing/CQRS 的設計想法，整體架構將是採用 Javascript 以 Node.js 為主要開發

DOI:10.6814/NCCU202101070

環境使用，Message broker 方面會使用相對穩定的 Mosquitto，每個 Service 都會包成一個個的 container 模擬在微服務的環境下的使用場景，每個 Service 都會以 API(接收訊息)和 Event(訊息格式)的形式互相傳送訊息，每個 container 皆可用最適合的程式語言進行撰寫，整體系統的流程就如圖 9 所展示

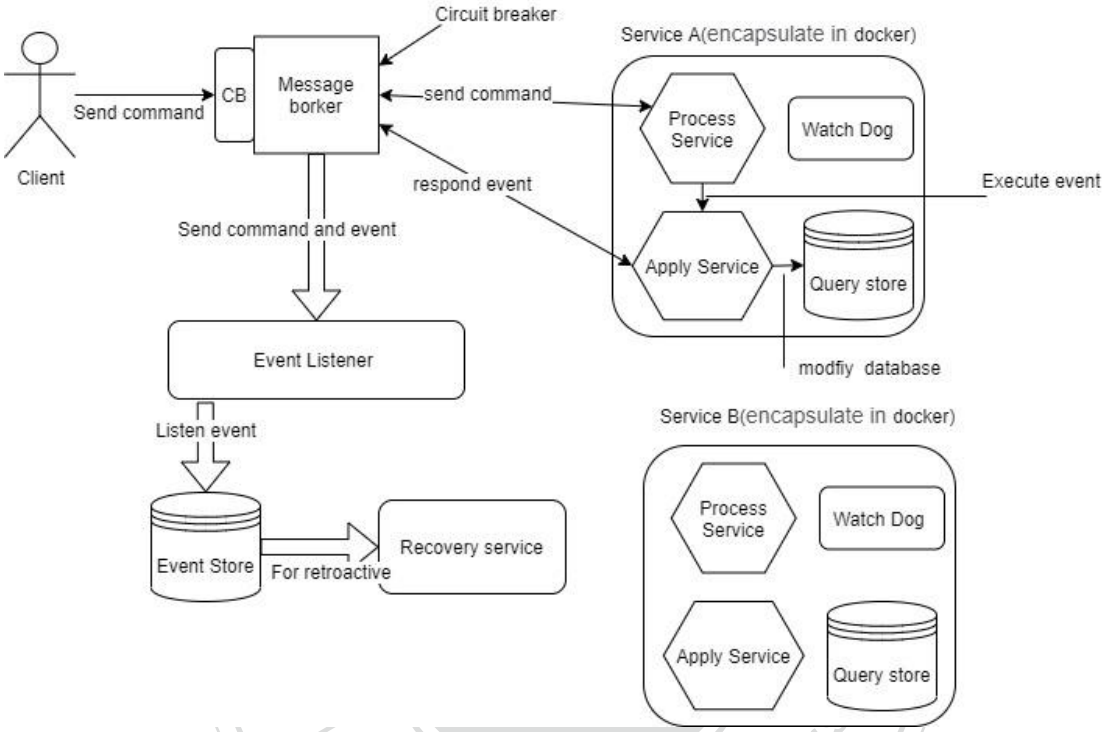


圖 9：整體 CQRS/Event Sourcing 的設計架構圖

3.1.1 系統架構

- 創建事件服務(Process Service)：由於使用者(client)所下的修改指令(command)並非是事件的格式，為了能將這些 command 傳送至基於 event-based 的系統當中並且運作就必須透過 Process Service 將其 command 轉化成複數個 event object。
- 執行事件服務(Apply Service)：從 Process service 當中收到的 event 尚未執行狀態，Apply service 的功能就是將這些已經產生的 event 實際執行，這些受

DOI:10.6814/NCCU202101070

到執行 event 結果將會反映到 Query store 當中，event 同時也會被標記為成功執行的狀態。

- 看門狗(WatchDog)：用來監控所有的服務(service)是否發生錯誤或是無法運行的狀態，若是發現了錯誤(failure)，就會開啟重啟回復機制(failure recovery)，試圖修復錯誤讓系統繼續穩定執行，而偵測錯誤的範圍就包含交易的 race condition 這部分將為在 4.4.1 的章節中詳細解說。
- 斷路器服務(Circuit Breaker)：當 service 中的無論是 Process service 或是 Apply service 中出現 failure 時用來暫時緩衝錯誤，主要目的是為了防止應用程式重複嘗試執行可能會失敗的作業，Circuit Breaker 會暫時保存 client 所傳過來的 command，並且不斷地重複嘗試連結後端服務，若是 command 的儲存量超出 Circuit Breaker 所能承受的數量時，就通知 client 系統已經出現問題。
- 查詢資料庫(Query Store)：儲存目前 service 版本的資料，在 client 送出 command 指令時的對照版本

此外還會需要另一個外部 Service 功能是用來進行回溯復原機制和建立權威資料庫。

- 事件監聽服務(Event Listener)：此服務的功能是用來完整監聽所有系統所產生的事件(包含 command,apply event,process event,etc)並保存至 Event Store 當中
- 事件資料庫(Event store)：保存所有 event，是最權威的資料庫版本。所有的服務回復都會透過此資料庫進行回溯還原。
- Recovery Service:啟用回溯機制時幫助 Service 還原至目前最新的 Query store 版本，本裝置會將重啟的服務透過 Event store 當中的歷史資料還原至最新版本。

DOI:10.6814/NCCU202101070

3.1.2 Service 端 & Watch Dog

一個 Service 當中將會包含四個必要的元件，前三個元件是用來模擬微服務當中各自 service 獨立的功能，本 Service 將是模擬未來在大型系統當中不同的商業邏輯例如:下單，轉款，信用卡支付，賣家網頁設計等功能。最後一個 WatchDog 則是本研究設計的亮點主要功能是作為錯誤偵測及錯誤回復的關鍵道具圖 4.2 將會詳細描述：

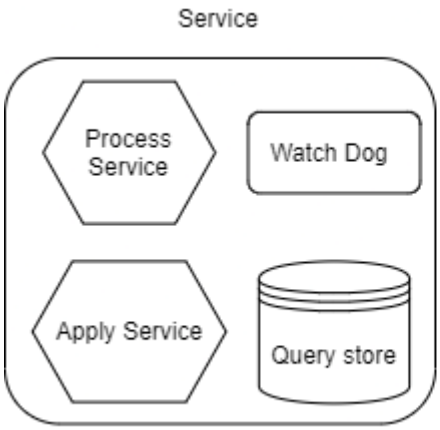


圖 10：Service 內部基本元件(模擬大型系統的商業邏輯)

根據 Chris Richardson 在其著作[1]中的第六章中提到過去傳統的資料儲存方式有以上四種缺點 1. 物件關系阻抗不匹配(Object-Relational impedance mismatch) 2. 缺乏歷史記錄功能 3. 執行數據軌跡(audit logging)時容易出錯 4. 傳統資料儲存系統並不支援 publish domain event。上述的四個缺點透過 Event Sourcing 的設計可以解決，而 Event Sourcing 的核心就是在於將 client 所下達的 command 轉換成 event 型式並且進行保存在章節 3.2 當中都有詳細說明 Event Sourcing 的優勢。

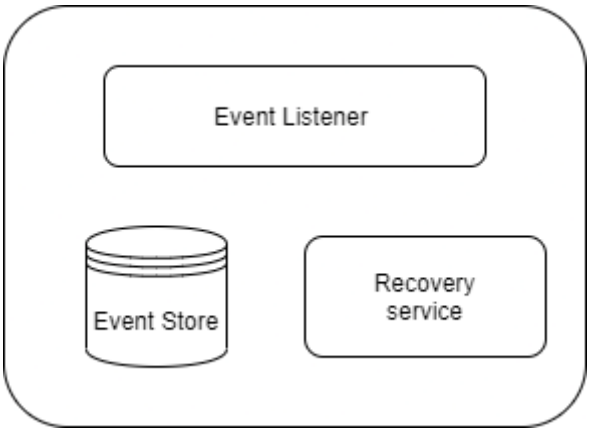
Process service 最主要的功能就是將使用者(client)所下的 command 指令(例如:線上轉帳，交易，存匯款，購物等)轉換成 event，Apply Service 的功能就是解析 event 並且執行 event，只有真正通過 Apply Service 的 event 才是真

DOI:10.6814/NCCU202101070

正被寫入執行的 event。不管是 Process Service 或是 Apply Service 是否成功執行都將會丟出執行 event(ProcessEvent,ApplyEvent)，這些 event 將會記錄 Service 的執行狀況，另外在執行 Apply Service 時又極大的機會遇到因為版本問題造成的交易外寫入的情況發生(在 4.4 中會詳細討論)，所有的 event 都將會被 Event Listener 監聽並被儲存在 Event Store 裡(在 4.1.3 中會詳細解說其功能)。

Query store 是用來儲存當下版本的數據紀錄，由於所有的 Service 皆為各自獨立運行，相對來說就會遇到分散式最常見的一致性(consistency)問題，FLP 不可能原理 (FLP impossibility)[9]詳細證明不應該浪費時間，試圖未非同步系統設計出適合任意場景的共識演算法，透過中 CAP 定理[10]我們得知對系統中的一致性（Consistency）、可用性（Availability）、分區容忍性（Partition）做適當的取捨，面對這樣的問題本研究所採取的最終一致性 (Eventual consistency)的方式來維持系統資料庫的一致性，但相對的這樣也會衍生出交易未寫入的問題(在 4.3.1 中會詳細討論)。Query store 主要是提供使用者(client)查詢使用的，透過 CQRS 的設計將可以有效分流寫入 (Command)、讀取 (Query)，有效避免單一失效，過多使用者進入導致系統當機的情況發生。

3.1.3 Event Listener, Event Store, 和 Recovery Service 機制



DOI:10.6814/NCCU202101070

圖 11：Event Listener、Event Store、Recovery Service 內部基本元件

Event Listener 以及 Event Store 的功能非常的簡單，一個負責監聽所有 event 一個負責儲存 event，比較特別的是 Recovery Service 主要有兩個功能，其一是負責處理交易未寫入的 drop event 將這些執行錯誤的 event 重新送回 Process Service，這個回復指令將會自動執行直至 Main Dog 下達暫緩指令(4.4.2)為止，其二為定期比對 Query store 是否出現錯誤並進行修正，由於 event store 的資料視為最權威資料所以若是發生不一致的問題將會以 event store 所回溯的資料為準。

3.1.4 CQRS 運作流程解說

這部分會詳細解說整個 CQRS 系統正常的運作流程，即為在沒有任何錯誤的情況時的運作模式，4.3 會解說若是遇到錯誤時該如何處理。

Client 訊息傳送 Command mode

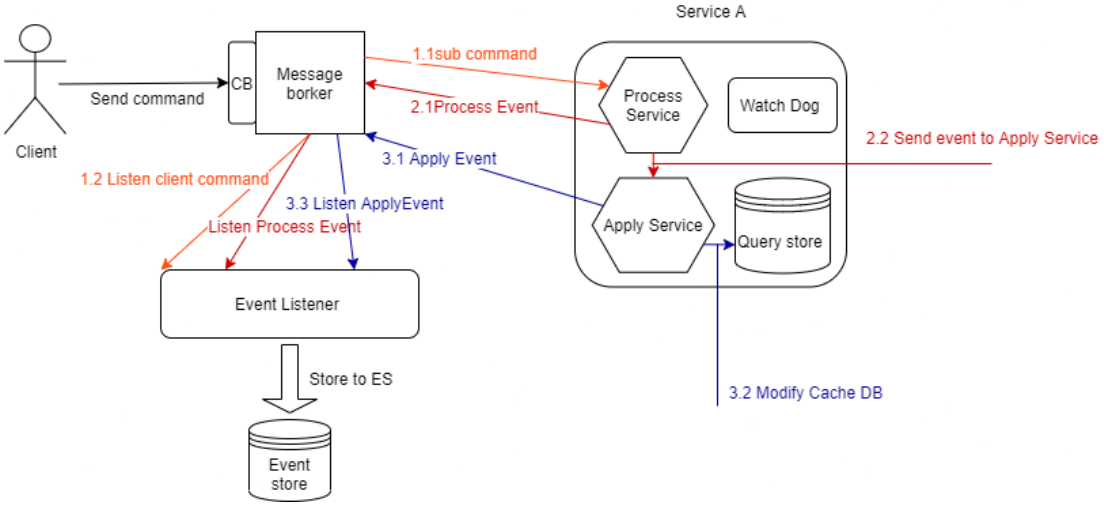


圖 12：Client 傳送 command 訊息

CQRS 的設計方式最大的特色就是將 Command 和 Query 的兩項功能進行分流，在一般的情況下大量的使用者基本上只會使用 Query 查詢的功能，所以將這兩項功能進行分離主要是為了將電腦效能運用在 Command 即為修改資料庫數據或是交易，本小節就是要探討 Command 端是如何執行

DOI:10.6814/NCCU202101070

面對 Client 所下達的 command 指令最先會經過 Circuit Breaker (4.3.1)把關，Circuit Breaker 在前面的章節中已經有提到過它的作用主要是用來預防當系統失效時使用者多次傳送無效指令給系統。再來就是透過 Message Broker 將訊息傳遞出去 Process Service 將會收到指定 Topic 的訊息。再來就是將收到的 command 指令轉換成 event type 這個工作將會交由 Process Service 負責，若是成功轉換完成時 Process Service 就會傳送出 ProcessEvent 並且標記此 Event 為成功轉換，相反的若是失敗的會就會標記成 drop event 兩者皆會被儲存在 Event Store 當中，成功經由 Process Service 轉譯的 event 則會經由 Apply Service 進行改寫，最後 Apply Service 會傳送出 ApplyEvent 並且標記此 Event 已經成功運行。

Client 訊息查詢 Query mode

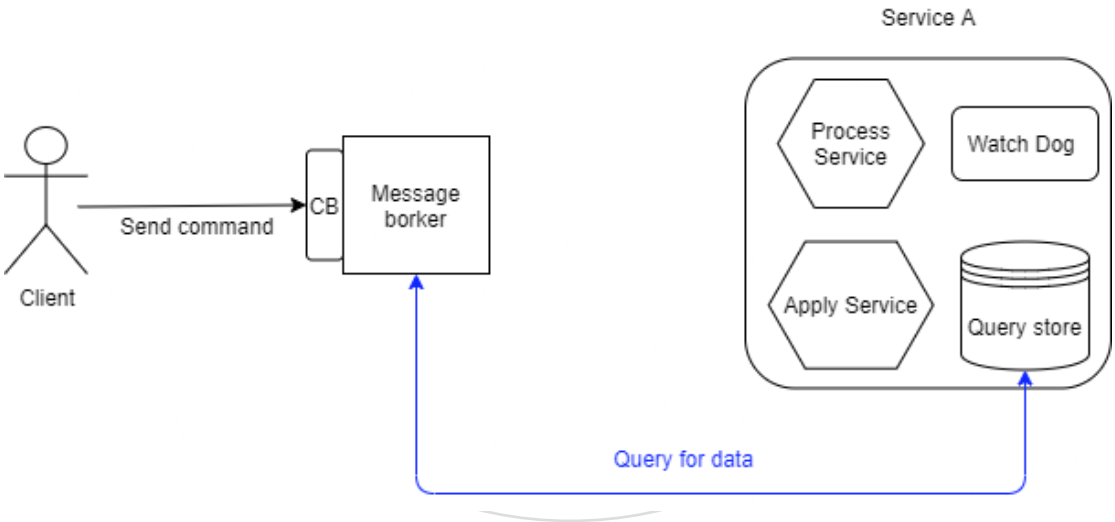


圖 13：Client 的 Query 路徑

Query mode 的運作很簡單收到的 Query 指令會直接連結到 Query store 並且取得 Service 中的暫存的資料，這些資料都是專門提供給使用者查詢的，透過這樣的方式使用 Query 端功能就不會用到 Process Service 和 Apply Service 這兩個資源，大大降低 Service 的工作量，然而缺點就是使用者看到的資料並非是最新最完整的資料，原因是因為使用者不可能每一次的 query 都直接連結到 Event Store 中將資料進行回溯驗證。

DOI:10.6814/NCCU202101070

3.2 強化集中式監控系統(WatchDog)

3.2.1 WatchDog 機制解說

現今常見的監控系統皆為集中式監控系統,也就是因為中心化的監控就極有可能造成單一失效問題,試想若是整個系統的監控核心突然失效,發生失效的服務無法及時被發現並加以修正,那還怎麼維持整個系統的穩定性呢?本章節將會介紹本研究所設計的看門狗機制(WatchDog),此監控系統者要是利用 Raft 共識演算法進行分散式系統的票選,當唯一的 leader 失效了,還能透過剩下的 WatchDog 進行下一輪的 leader selection 藉此維持系統的穩定性,預防單一 leader 失效所產生的整個系統崩潰。

3.2.1.1 Heartbeat 機制

Heatbeat 主要是用來監控在 Service 中的各個 component 是否是正常運作,方法會是由個別的 component 發送 heatbeat 訊息給 WatchDog service 證明自身是否是正常運作,使用的方法將會是被動式的單向傳輸 heatbeat(一般情況下),當 WatchDog 在一定時間內仍然尚未得到某個 component 的 heatbeat 訊息時 WatchDog 就會主動聯繫該 component 來確保 component 是否真的已經死亡(雙向確認),藉此來減少 UDP 的傳輸負擔,heatbeat 的傳輸方式會以 UDP 為主而並非 message broker,在這邊的考量是就算 message broker 因為某些原因而失效各個 component 還是能透過 UDP 進行控制與重啟。另外之前也曾嘗試過透過 ICMP (Internet Control Message Protocol)進行各 Service 的連結,不過由於 ICMP 只能 ping 固定 IP 不能針對 IP 中的其他 service 判斷失效的原因也無法傳送 ping 以外的資訊給其他 Service。

DOI:10.6814/NCCU202101070

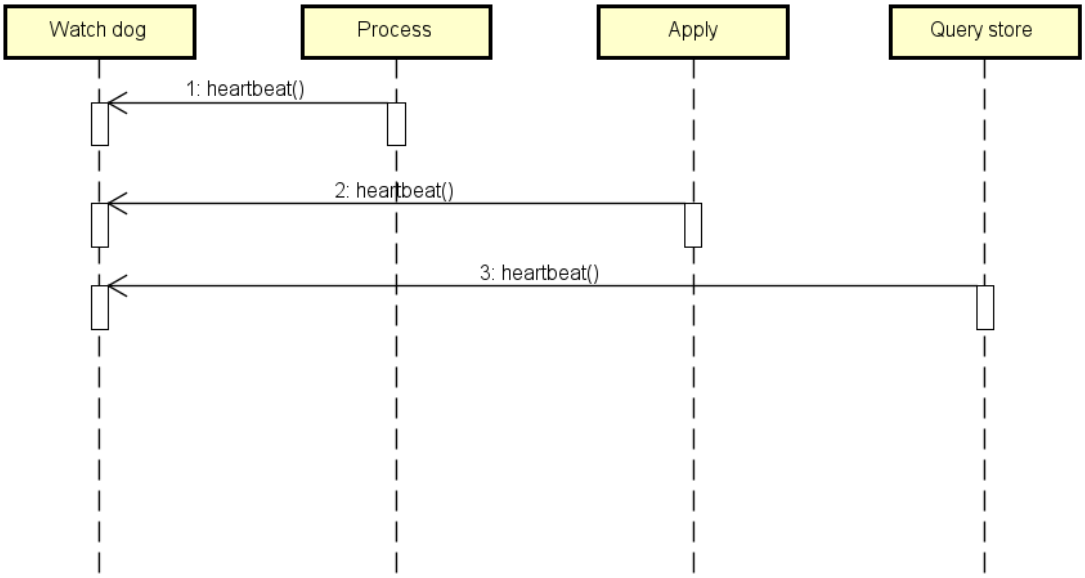


圖 14：所有 Service 中的 component 將會傳送 heartbeat 訊號給 WatchDog (內部監控)

3.2.1.2 MainDog 機制

每個 Service 中都有屬於自己的 WatchDog，但是實際運作的只有 Leader WatchDog(簡稱：Maindog)剩下的只做接收 MainDog 所發出的 heatbeats 來避免 MainDog 的失效，這裡就有包含 cold spare [18] 的概念，真正運行的 WatchDog 只有一個，其他的 WatchDog 則是持續處在待機狀態，而 MainDog 會以 multicast 的放是透過 UDP 傳送 heartbeat 的訊息，若是發生 MainDog 失效的話全部的 WatchDog 會再選出一個新的當作 MainDog，並且重啟原來失效的 MainDog，但是此時原來的 MainDog 就不會再是 MainDog 了，而會轉變成一般的 WatchDog，在失效的期間內所有的 WatchDog 監控系統會再次選出新的 Leader 轉換成 MainDog，以維持監控系統的穩定性。本研究使用 Raft 演算法當做是 WatchDog 們的共識演算法，WatchDog 之間的聯繫皆使用 UDP (User Datagram Protocol)作為通訊手段。其他在 Service 內的 Watchdog 則會監控 Service 內部的元件確保所有的元件的健康狀態都在可接受的範圍內，若是 MainDog 想要取得某 Service 內

DOI:10.6814/NCCU202101070

部狀態時就會要求節點中的 WatchDog 給予相應資料。

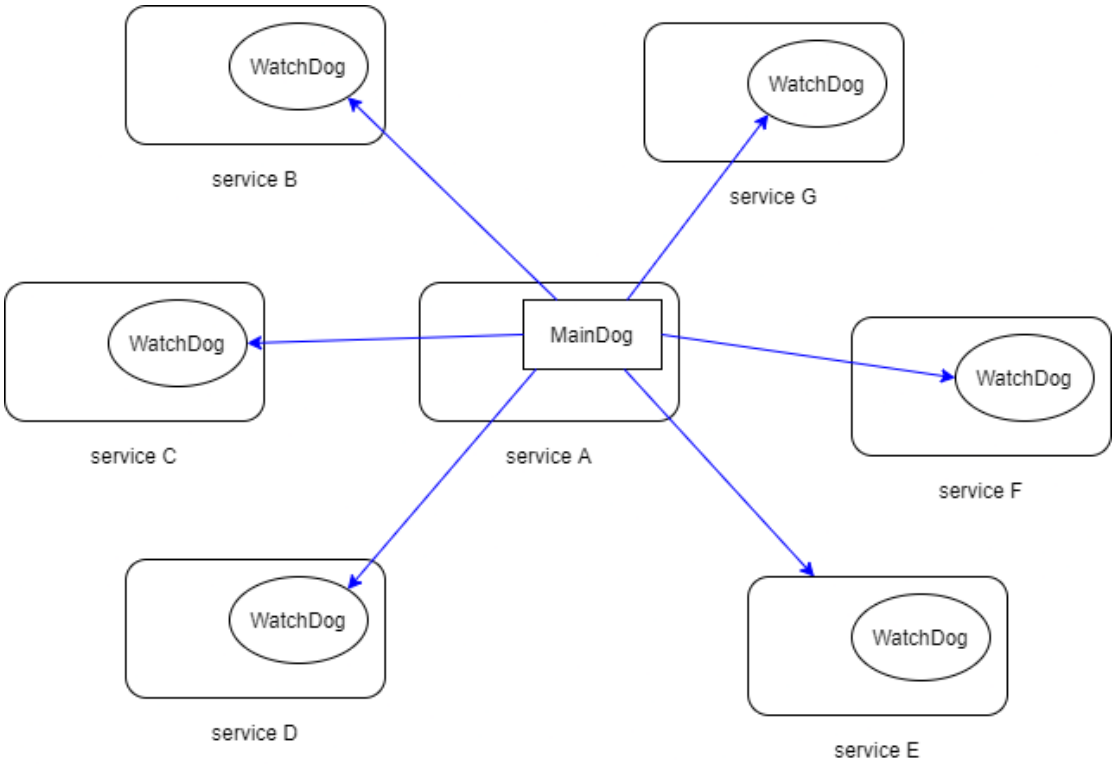


圖 15 ：Cold spare MainDog 群播示意圖

DOI:10.6814/NCCU202101070

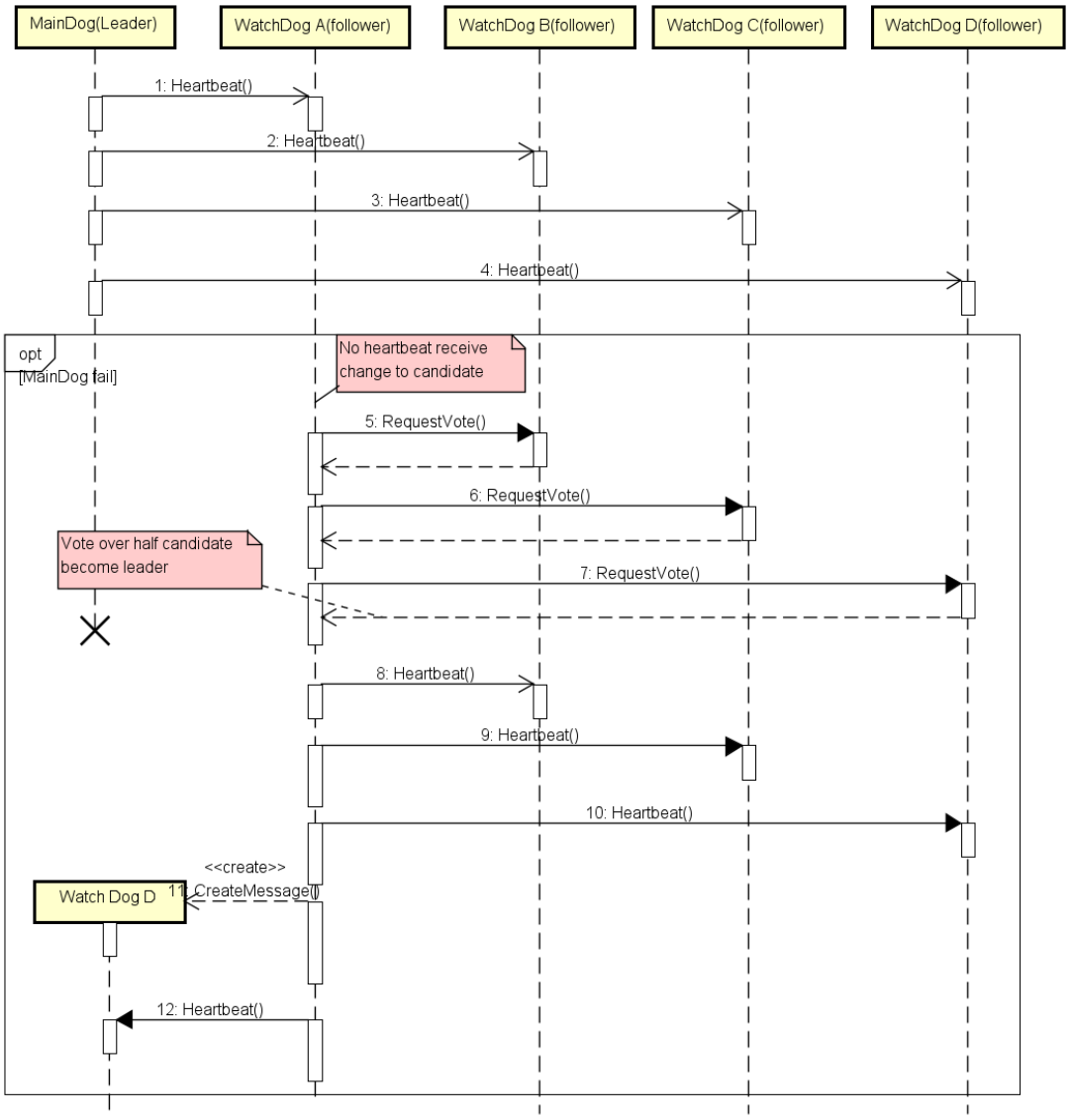


圖 16：Supervivor WatchDog 監控設計，當 MainDog 出現故障時該如何處理

3.3 增強因單一失效而造成崩潰的分散式系統可用性(Circuit Breaker)

本章節將會解說關於上述最開始所提到的問題二:增強因單一失效而造成崩潰的分散式系統可用性,在一般的系統缺乏面對其他服務失效時所造成的停滯無法處理,當系統失效後將造成Client端所下達的指令無法傳送到系統當中,在一般的情況下解決的方法就是將所有的 command 保存起來等待後台系統的回復，但是回復後的系統一瞬間面對排山倒海而來的 command 訊息

DOI:10.6814/NCCU202101070

極有可能再次引發崩潰，面對這樣的問題本研究設計了 Circuit Breaker 來當作一個緩衝機制來保護前後端的系統。

3.3.1 Circuit Breaker

Circuit Breaker 主要的目的是偵測後端系統的可用程度並且防止應用程式繼續發送失敗或是無效的指令給後端系統。Circuit Breaker 在 Cloud native system 當中已經成為現今熱門的 pattern 主要是由於在大量的分散式系統當中的服務，為了保護自己的服務系統不會受到其他服務的失效造成無法運作，進而設計出這套保護裝置，Netflix 所使用的 Hystrix 就是為了增加系統的 fault-tolerant 所設計出來的。

運行的過程中通常將會由 Client 端發送 command 指令，所有指令都會經過 Circuit Breaker 這個保護系統，Circuit Breaker 將會在三種模式中不斷切換運行下圖將詳細說明這三中模式:

Close mode:

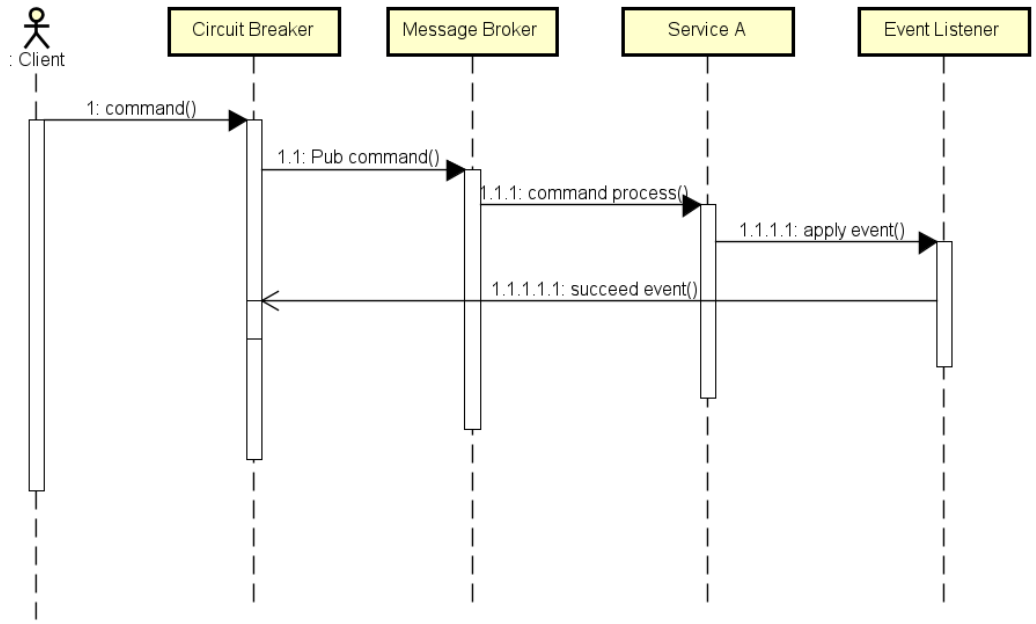


圖 17：Close mode 流程圖

Close mode 預設在所有的系統都可以正常運作的情況下的模式，所有的

DOI:10.6814/NCCU202101070

command 訊息皆能直接傳至 Service 中，暢行無阻。

Open mode:

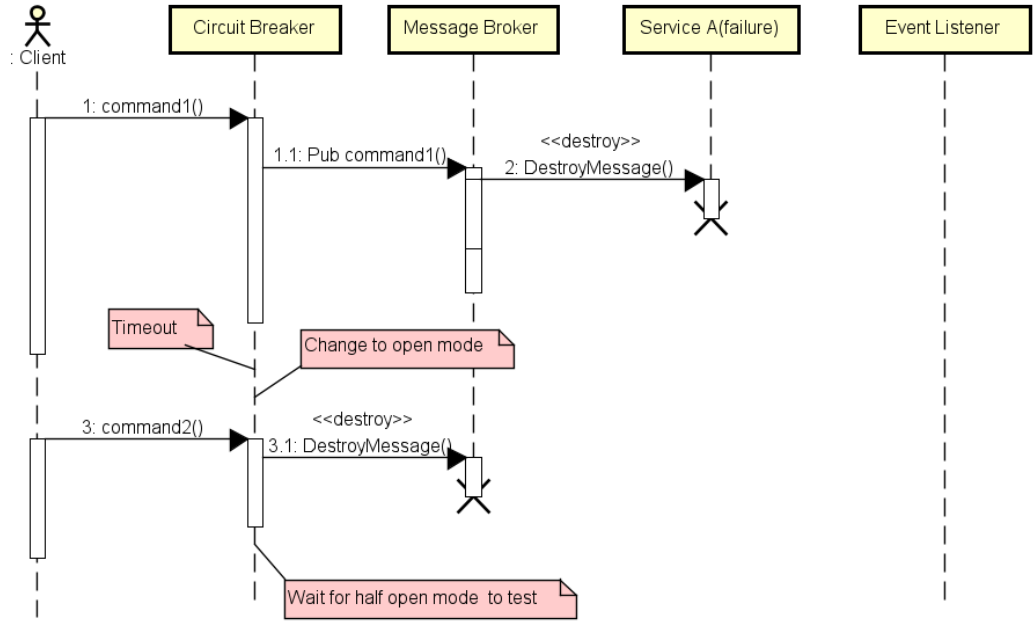


圖 18：Open mode 流程圖

Open mode 當系統處於正常無法運作的情況下的模式，所有 client 端所下達的 command 皆會被 Circuit Breaker 阻擋防止無效指令一直傳送。

DOI:10.6814/NCCU202101070

Half-Open mode:

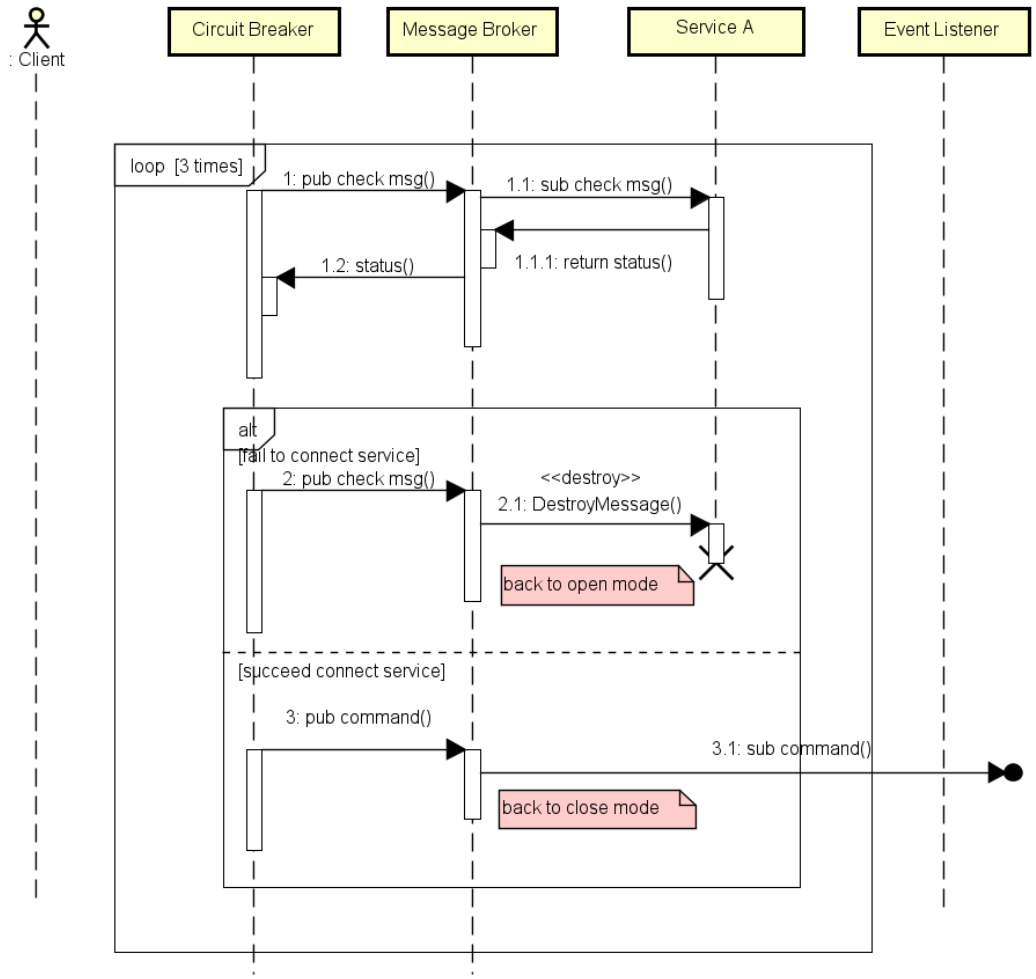


圖 19：Half-Open mode 流程圖

當 open mode 持續一段時間後將會進入半開模式，此模式主要目的是用來定期詢問後端系統是否已經恢復並且可以正常運作，半開模式下會傳送 check message，當後端接收到此訊息時將會把目前後端系統的整體情況通知出去，是否能夠運作也會包含在這之中，Circuit Breaker 將會重複傳送三次 check message 確認系統可以正常運作後轉換成 close mode

若在三次之中的任何一次出現斷線或是無回應的情況發生時，就會再度轉換成 open mode，等待下一次的詢問。

DOI:10.6814/NCCU202101070

3.4 排除 Event Sourcing 系統交易中可能出現的競爭危害

3.4.1 交易未寫入情況(Race condition problem)

由於最終一致性(eventual consistency)的關係使用者看到的資料會跟最新的資料有所出入，然而在一般流量不多的情況下發生同時修改同一筆資料的機會不多，但若是使用人數變多時就難免會遇到同時修改的情況，遇到這類狀況將可能造成交易未被寫入的情況發生。

在進行 command 轉換成 event 的過程時，將會發生的錯誤就是如果兩個 client 同時對同一筆資料進行改寫就會產生 dirty read 的情況發生[1]，也就是說 dirty read 的發生會造成原本所下達到交易未被寫入，解決這個問題的方法是每當 DataBase 受到改寫時就要更新版本，之後若是 event 跟這個版本產生衝突，就會標記狀態為 drop，並存至 event store，再重新發送 command，已確保 command 有被執行到。整個過程將以圖 20 為例：

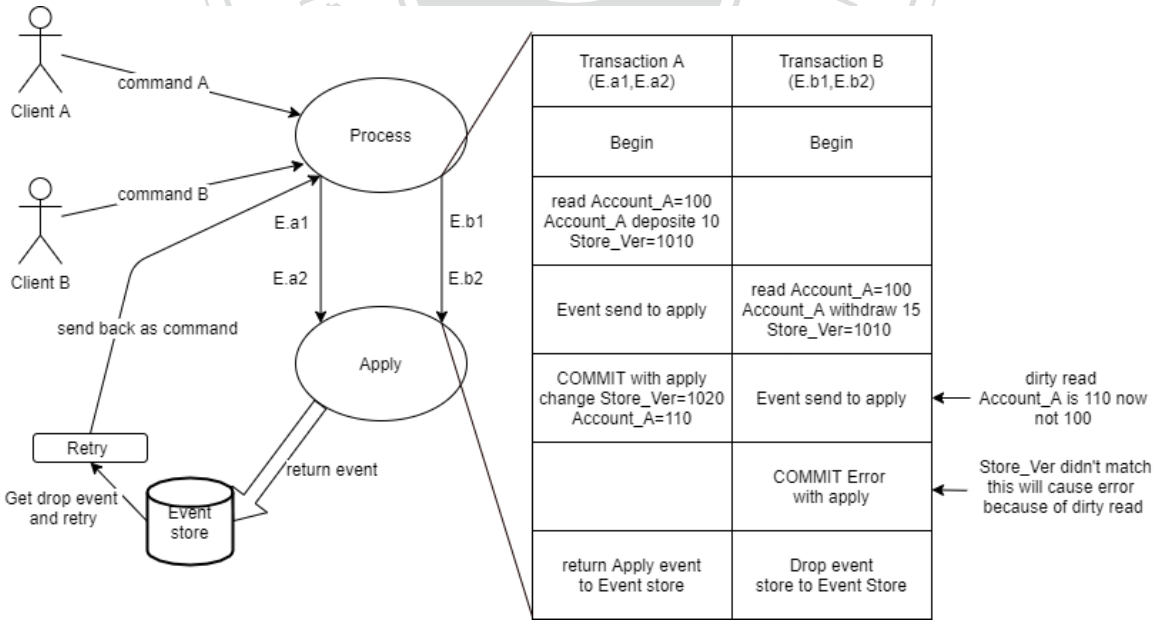


圖 20：潛藏的交易問題以及處理方法(樂觀鎖)

DOI:10.6814/NCCU202101070

3.4.2 交易 retry 機制

根據在 4.3.1 當中所提到的交易未寫入問題，其為寫入的主要原因是因為兩筆 command 同時要對同一筆資料進行修改，當第一筆指令修改資料後的資料版本就已經和當初的不相同，所以第二筆 command 的修改指令會因為版本不相同的關係被系統標記為 drop，drop event 被 Event Listener 間聽到後第一時間會被存入 Event Store 專門存 drop event 的地方，Event Listener 將會把 drop Event 重新丟回 Process 再次產生新的 event 確保此 command 可以成功運行，新的 event 會多出 drop_ID 的屬性，最後完成 commit 後的 ApplyEvent 會傳至 EventListener，經過對比後將 Event store 裡失敗的 drop event 從 drop list 中去除。

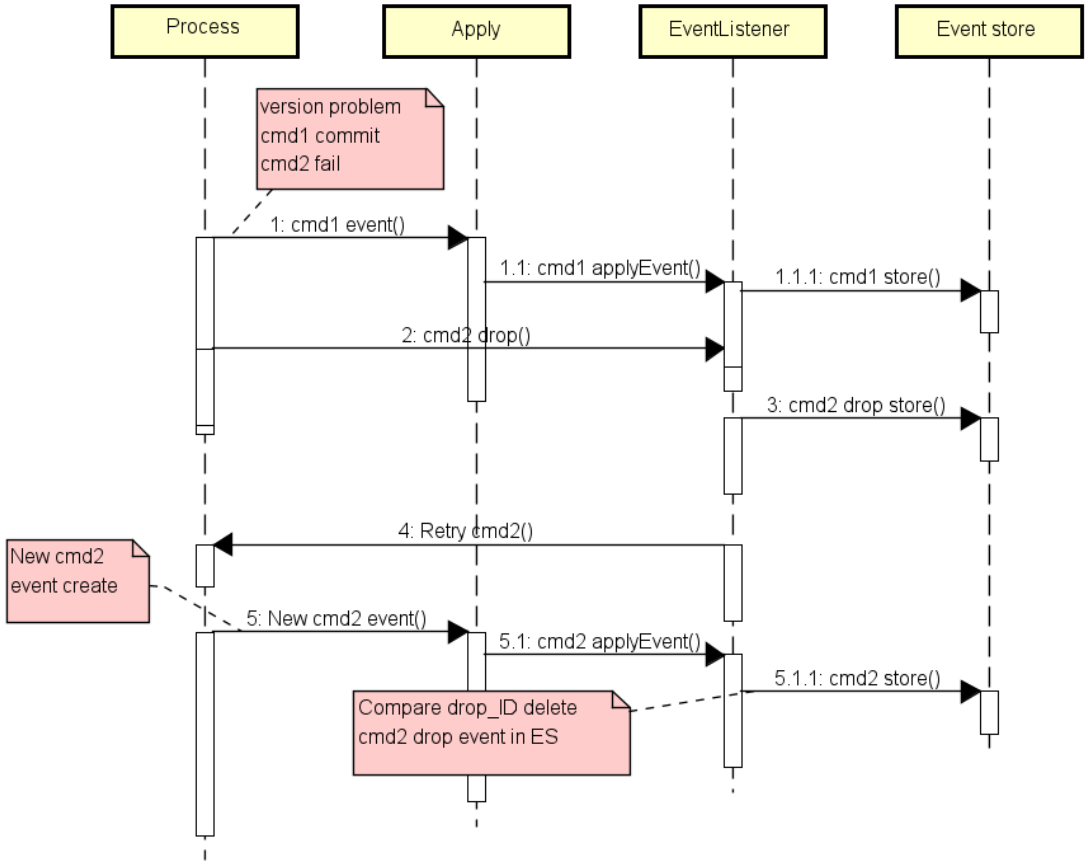


圖 21：Transaction retry 機制流程圖

DOI:10.6814/NCCU202101070

3.5 錯誤偵測及錯誤回復機制

3.5.1 Service 端未回應(Service A 失效)

情況一:

Service 完全喪失功能 MainDog 無法接受到來自 WatchDog 的 heartbeat 訊息，此時 MainDog 會主動傳送 Query message 進行二次確認若是有所回應就會回到正常模式，但若是依然沒有回應就會直接排除失效的 Service 後重新啟動新的 Service 進而維持系統的正常運作。

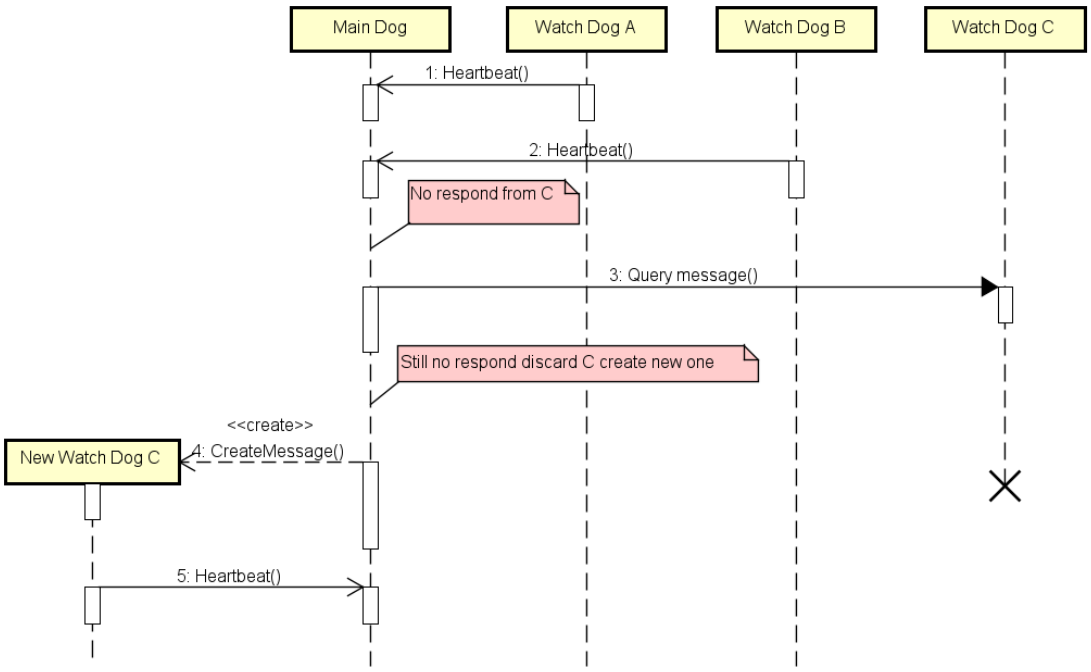


圖 22：當整個 Service 完全失去聯繫的情況下的復原機制

情況二:

失效的部分只有 Service 中的其中幾個元件，這時 WatchDog 會主動傳送錯誤訊息的詳細資料給 MainDog，經過 MainDog 的評估後決定是否能夠自行修復亦或是將失效的系統整個重啟。

DOI:10.6814/NCCU202101070

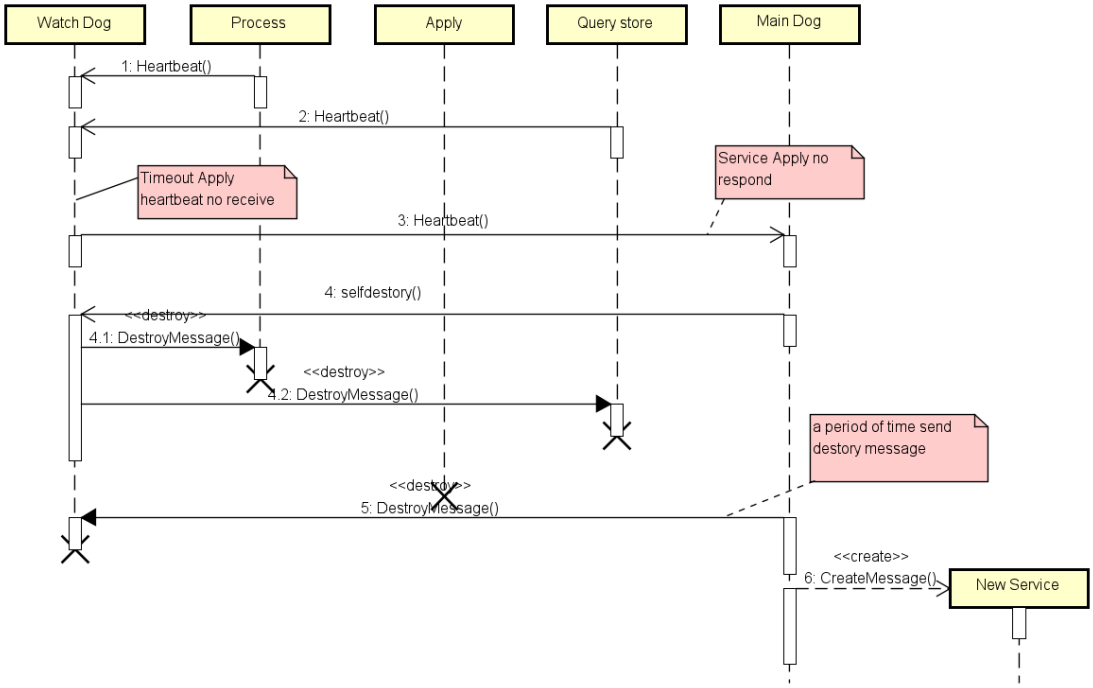


圖 23：當只有 Service 中的某個元件失去聯繫的情況下的復原機制

3.5.2 Service 重啟機制 (Failure recovery)

本小節將會探討若是 Service 本身無回應或是出現問題時系統會如何運作，本案例會假設一個 service 的失效會對整個系統造成甚麼問題。

當 Client 修改指令送出後將會有兩個元件會收到此 command 分別是 Event Listener 以及 Process Service，但如今 Service 已經失效相對的 ProcessEvent 並不會被傳遞出來，要判斷 Service 是否已經離線須滿足兩個條件 1. Event Listener 再一定時間內未受到來自 ProcessEvent. 2. Main Dog 在一定時間內並未收到來自 Service A WatchDog 的 Heartbeats 訊號，滿足上述兩個條件 MainDog 就會主動聯繫 Service A 的 WatchDog，若還是無回應就會啟動自動復原機制，同時送出的 Client command 會被標記為 drop。

此時無論 Service A 是否失效都會被遺棄，Main Dog 會直接重啟另一個相同功能的 Service 將取代，重新開啟的 Service 將會傳送 Restart message 將重啟的 IP 透過 UDP 傳送給 MainDog，MainDog 確認已經重啟 Service 後將會通知

DOI:10.6814/NCCU202101070

Recovery Service 將回溯資料傳遞至 Service 的 Query store。

而自動回覆機制的重啟指令是透過 MainDog 傳送 recovery 的 event，收到 event 的 Recovery service 會透過收到的 event 指令進去 event store 撈取特定區間的 event 並且進行回溯的動作，將執行完的結果也就是最新版本的傳送至給 process/apply 的 Query store 裡。

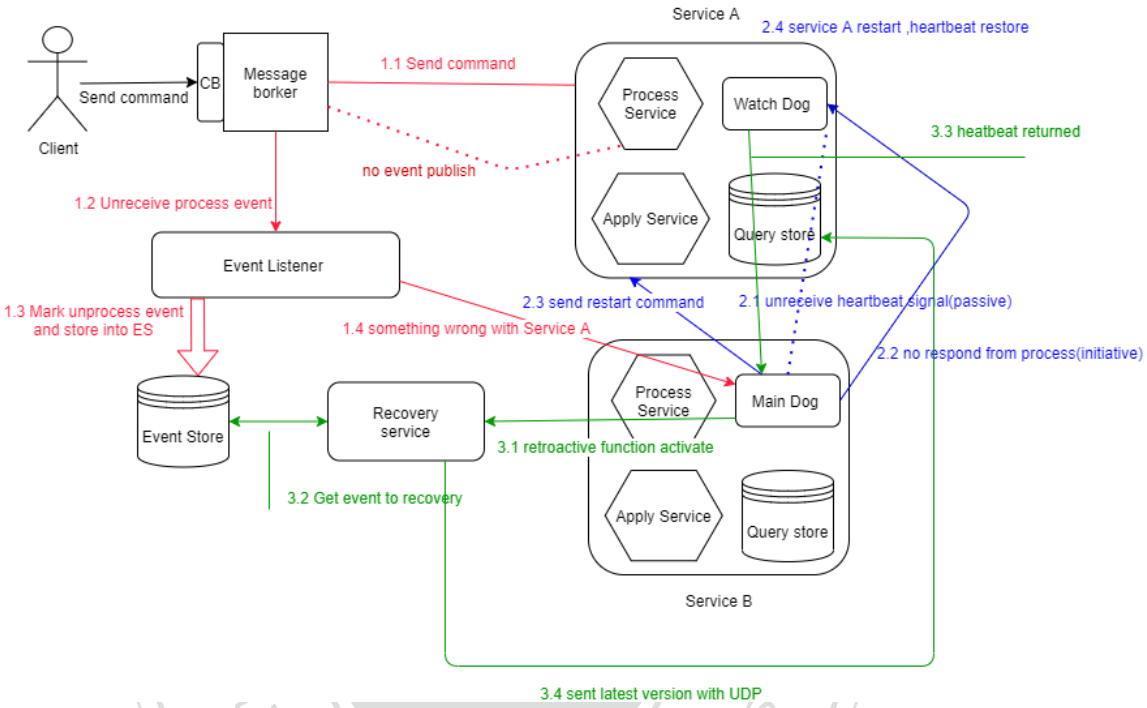


圖 24：遇到 failure 時的流程示意圖(以 failure 發生在 Service A 為例)

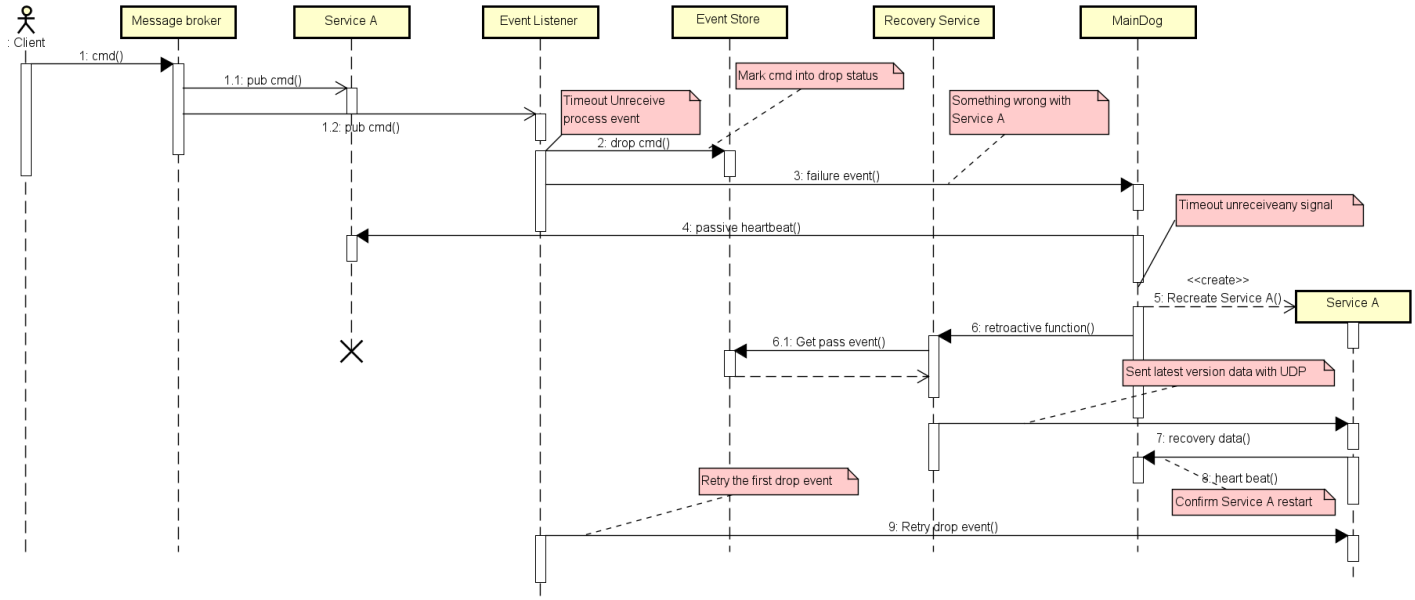


圖 25：回復機制的流程圖

DOI:10.6814/NCCU202101070

3.6 使用場景

本小節透過實際的場景演示出系統當中是如何處理 race condition 的問題。假設有極大量的歌迷要搶某演唱會的門票，但是門票現在只剩下 10 張，假設現在有 100 個人同時對系統進行下單的寫入指令，這些指令，在 CQRS/Event Sourcing 的架構上，會以 event objects 的型式送入系統，然而客端在送入這些 events objects 前，讀取到的票數餘額很可能是一樣的版本。為了遵循先到先處理的原則，速度最快的 event 將會最先被執行票數餘額就會減少，但此時在各個客端的認知中，票數餘額都是 10，而真實的情況是當第一個 event 被 commit 後數量就成 9 個。其他 event 所得到的訊息是過去的訊息這裡就會有 dirty read 的情況發生。遇到這樣的問題最壞的情況就是交易錯誤(庫存 10 個賣出 50 個)。所以本研究在這裡設計了 drop_event 以及 Store 版本標記，Store 版本標記將會讓系統對比目前讀取的資料是否為最新資料，而 drop_event 則會將那些不正確的資料隔離出來，這些被隔離的 drop_event 會按照時序存進 event store 當中保存。最終透過 Event Listener 將這些 drop_event 重新依照時序回傳至 Service 當中進行再一次的 commit。這次的讀取版本將會是 Query Store 中的最新版本。透過這樣的方法可以達成消費者的公平性(First in First out)以及防止交易錯誤的問題。

DOI:10.6814/NCCU202101070

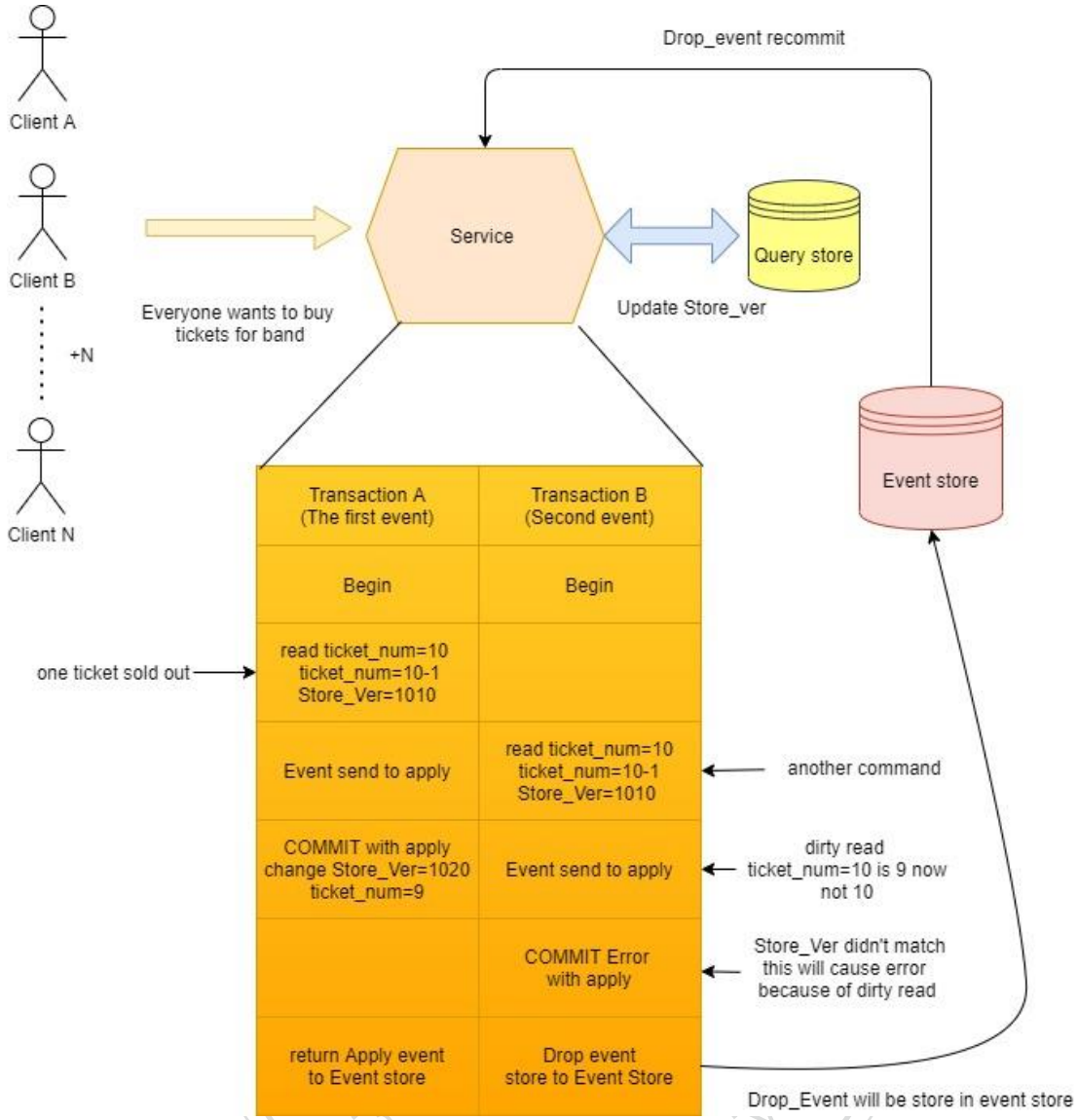


圖 26:Race condition 問題透過 drop_event 與版本標記得以解決

DOI:10.6814/NCCU202101070

第四章 實驗測試與討論

4.1 實驗總覽

此章節所設計的實驗將會是基於驗證本論文的貢獻問題以及測試本論文所設計出來的系統架構可用性，透過以下三個實驗測試本文所設計的系統是否能夠符合 1.多數 Service 毀損對比系統重新恢復所需要花費的時間 2. 當 Service 數逐漸增加的情況下 leader 失效後能夠重新選出新的 leader 所花費的時間對比 3. Client 端下達 command 的數量觀察系統處理效能對比完成時間。接下來將會以三個小節詳細解說關於實驗項目以及數據解釋。本論文所使用採用硬體設施包含 intel i7-9759H、16G RAM、而軟體開發測試層面則是 Docker v18.09.9、Node.js v11.15.0、Express v4.16.1、PM2 v4.5.6、Node-clinic v8.0.1、Wireshark v3.4.6、Mosquitto v2.0.10 進行程式開發封包抓取以及回應時間檢測。

4.2 系統錯誤回復效能

本小節實驗將會利用 Docker 部署 50 個相近的節點模擬分散式系統當中不同服務的連結，訊息傳送的部份會以 MQTT (MESSAGE QUEUING TELEMETRY TRANSPORT)做為主要通訊手段而監控系統的共識連結則會以 UDP 為主，本實驗將會運作兩種不同的系統其一是一般常見的的集中化式監控系統，這樣的系統將會把監控裝置獨自隔離出來，在透過 API 連結進行監控作業，然而這樣的設計有可能造成單一是失效的情況發生，若是想增強其穩定程度就必須確保集中式的

DOI:10.6814/NCCU202101070

監控系統能夠不出現任何問題。其二的系統是本論文所設計的 WatchDog 監控系統，此系統是透過 Raft 共識演算法讓分散式系統中的所有服務都能夠成為主要的監控裝置，透過共識演算法選出 Leader 其餘的就成為 Follower 並且進入休眠狀態。透過此方法可以允許主要的監控裝置發生錯誤，而發生的錯誤並不會影響到整個系統可以回復的可能性。

接下來的實驗將會讓這兩套系統做對比，本實驗將會把穩定度(Stability)定義為(成功次數/總次數*100%)，部屬 50 個不同的節點這些節點將會透過類似 chaos monkey 的自製程式進行隨機破壞，毀損比率將被定義成 50 個節點中會選出多少節點失效若是 50%的話就是有 25 個節點被破壞，本實驗將會重複 1000 次模擬測得最終穩定性的比率，

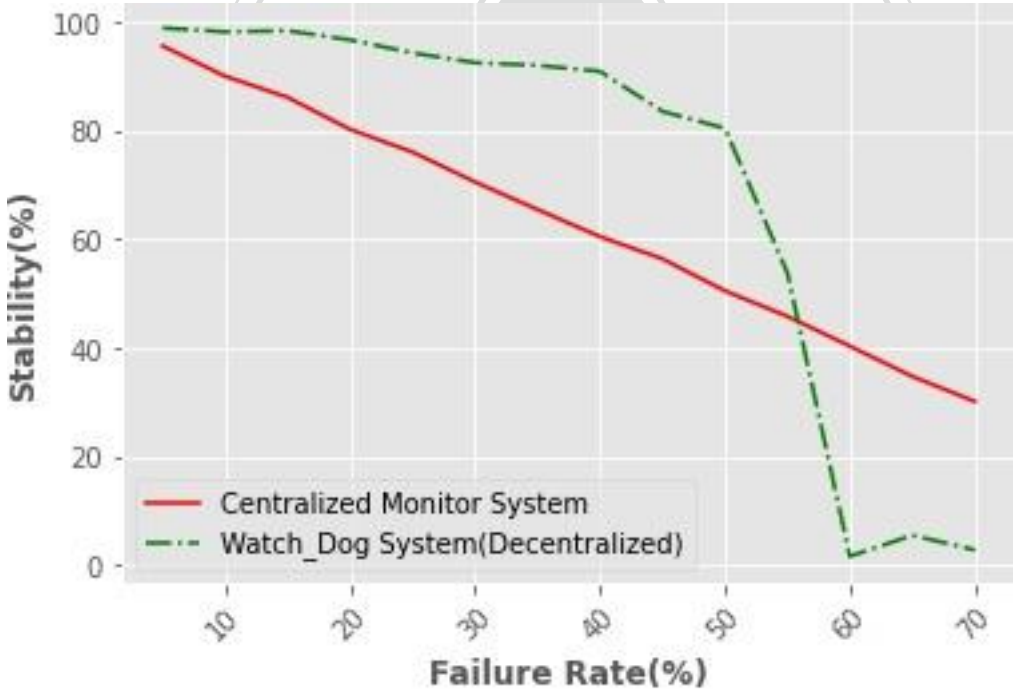


圖 27:集中式 VS WatchDog 監控系統穩定性對比

根據圖 26 中可以看出本研究所設計的系統(WatchDog)在面對毀損率低於 55%時的穩定程度是遠大於一般集中式的監控系統，而超過 55%後可以明顯發現穩定性的大幅降低，其主要原因是因為 Raft 共識演算法中當已經失去超過半數

DOI:10.6814/NCCU202101070

節點時將會無法票選出 Leader 進而導致監控系統的完全失效，試驗後發現若是可以將 WatchDog 設定在超過 55%後強制轉變成集中式的模式將可以有效避免55%後的單點失效問題更加確保系統上的穩定程度。

4.3 WatchDog 共識演算法恢復效能

本小節實驗將會測試 WatchDog 在進行共識時期恢復效能的數據，主要評估的依據將會是透過封包的數量大小以及重新共識所需花費的回復時間，對比節點數，可以預想的到當節點數愈高時整個共識效能就會越低，也就是說所產生的封包數以及共識回復時間將會增多，只是增加多少就需要做實驗才能夠測試得出結論。本實驗的設計會透過 wireshark 側錄即時封包數以及回復時間，同時也會使用類似 chaos monkey 的自製程式指定破壞處於任意處的 MainDog，觀察 WatchDog 共識演算法的修復所需時間及效能損耗，另外也會多加 WatchDog 共識時所消耗的封包量以及 heartbeat 所需的封包，本實驗將會從十至一百每十個為一單位逐一部署節點。

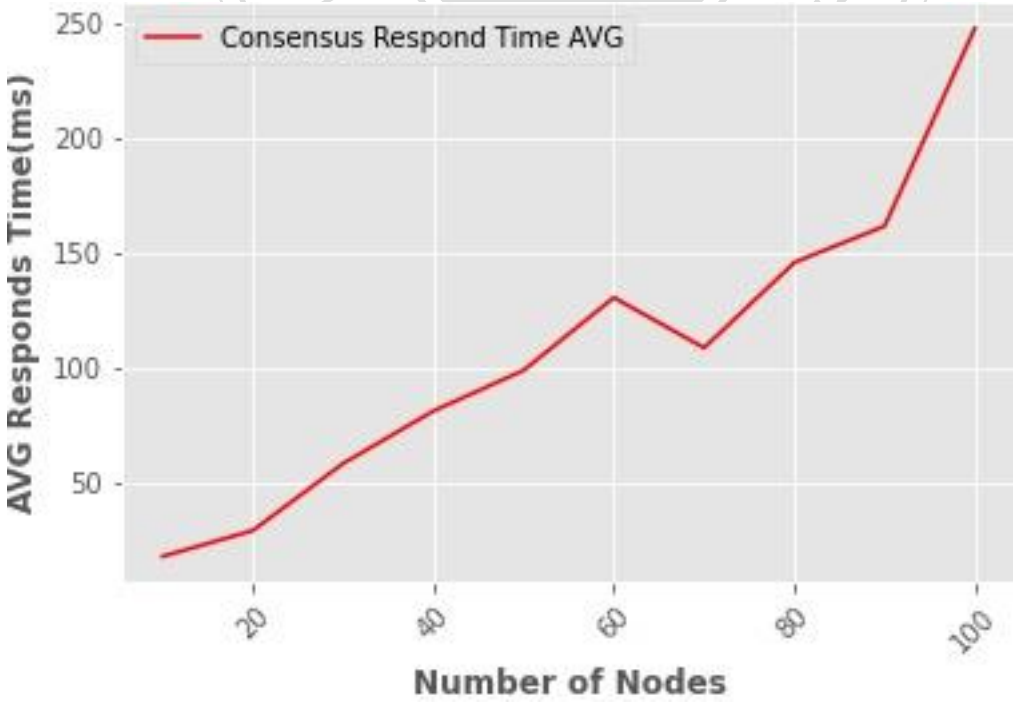


圖 28:共識節點數對比平均回應時間

DOI:10.6814/NCCU202101070

透過圖 27 可以清楚看出當節點數越高時所需的共識時間將會越長，不過由於每次的共識回應時間都會因為當時不同的情況而有不同，每個節點數之間的回應時間差異大，單看個節點數的平均回應時間並無法完整看出實驗結果，所以本研究特別重複對每個節點數測試 30 次的回應時間並且進行數據分析，透過數據分析得出圖 28 的箱形圖。

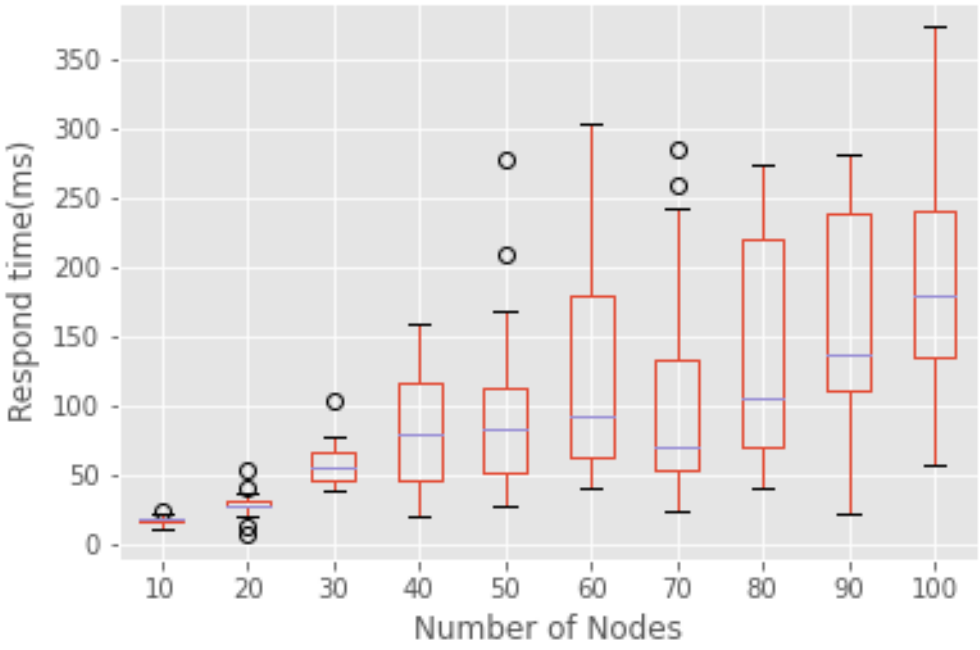


圖 29：共識所需的回應時間(箱形圖)

X 軸跟圖 27 一樣是節點數而 Y 軸是所需回應時間，從圖中可以觀察出在節點數在 40 節點前其回應時間的差異甚小，大約可以控制在 50ms 左右，若是超過 40 個節點數時共識回應時間產生巨大的差異，四分位間距大約都落在 100(ms)到 150(ms)左右，證明就算是相同節點數的情況下其共識的時間差異也是相當巨大的。在節點數逐漸增加的情況下是否能在一定時間內回覆共識將成為一個不確定的因素。

DOI:10.6814/NCCU202101070

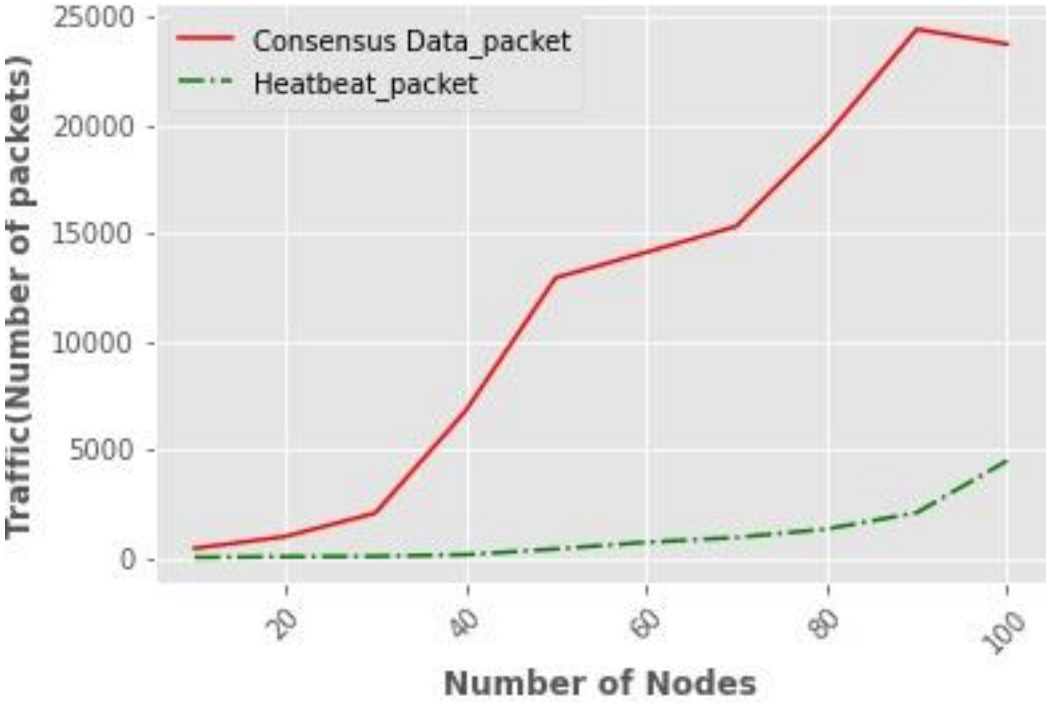


圖 30:複數節點所需的封包數

從圖中可以明顯看出當節點數愈大時所需要的共識封包數量超過比例的直線上升，在面對超過一百以上的節點時所消耗到封包數將會是一個犧牲效能的討論點，而 Heartbeat 的封包數量並沒有很明顯的暴增趨勢，對整體的系統的時間延遲不會造成太大的影響。

4.4 Race condition problem 解決效能

在 4.4 章節中探討到在使用 Event Sourcing 時的交易過程中可能會出現交易未寫入的情況發生，而本實驗希望透過高強度的壓力測試來觀察系統在面對高強度的 command 指令輸出的情況下是否能夠處理 race condition 的問題以及 Circuit Breaker 的加入是否能夠增強整個系統的穩定程度。本實驗將會透過 autocannon 做為高強度送出 command 的方法在短時間內盡量發送修改特定某個欄位的資料，進而模擬出人為產生的 race condition 環境，再來會利用 PM2 以及 Node-clinic 做為監測效能和計算回應時間的工具，在面對這種情況下系統一定會需要一定的時間內處理這些須修正的寫入資料，在尚未做出實驗時可以預期到系統會因為寫入

DOI:10.6814/NCCU202101070

的指令越多所需復原的回覆時間就會越大，但是當這些寫入指令達到一定的數量導致系統無法來得及處理這群有錯誤的寫入的指令時，整個系統就會崩潰並出現系統錯誤。

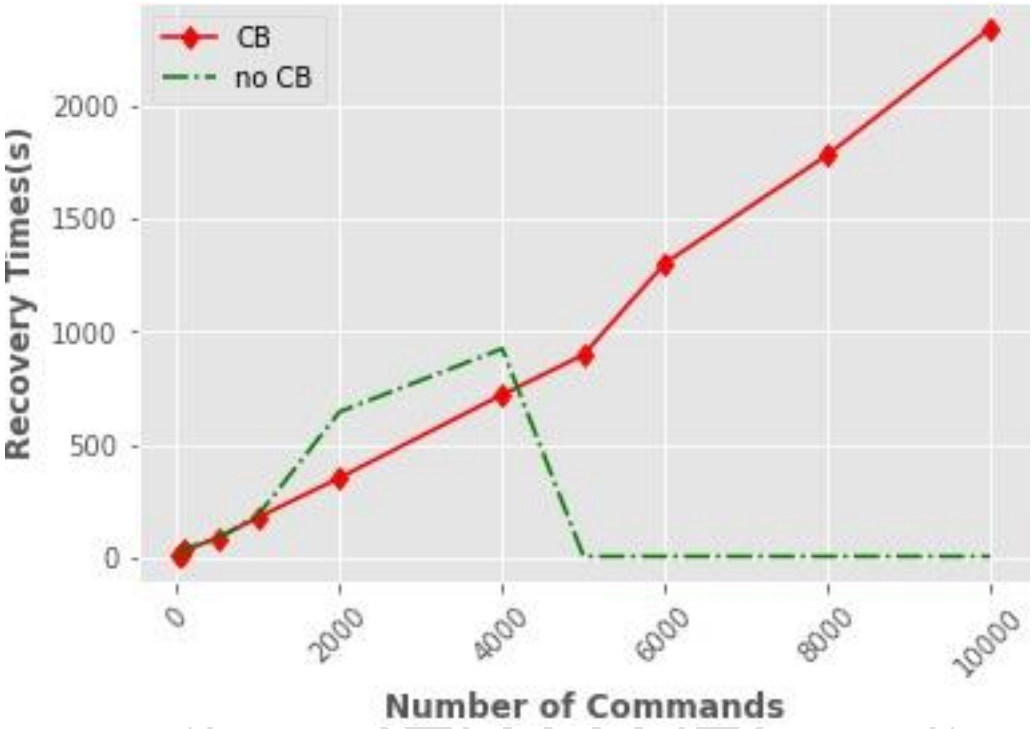


圖 31：是否使用 Circuit Breaker 的回復時間

在實驗中可得知使用 Circuit Breaker 在面對及高強度的寫入下不僅能夠正常運行更能增加處理寫入指令的吞吐量，在圖 30 中 X 軸是在一定的時間內高強度改寫資料庫的同一筆欄位資料的指令數量，而 Y 軸是系統處理所有因寫入的 race condition 而產生衝突的修改指令其回復所需要耗費的時間。由圖 30 可以得知兩者系統在同時寫入 1000 筆資料的情況下所得到的回復時間並沒有太多的差異，但是在面對 1000 筆以上的寫入時沒有 Circuit Breaker 的系統所需要的處理時間漸漸增加，而當寫入接近 4000 筆後系統將會呈現失效狀態，相反的有 Circuit Breaker 的系統雖然仍需花上一段時間才能夠回復正常狀態，但是本系統確實有能力可以修復過多流量所導致的失效情形。

DOI:10.6814/NCCU202101070

4.5 討論

透過上述三個小節的實驗說明可以明白在本研究所設計的 WatchDog 監控系統在面對低於系統 55%毀損率時將會比起集中式的監控系統穩定，可以從這當中看見本研究系統的優勢極大，但是這樣的優勢下必須提議下四點限制條件補足本研究不足的地方。下面將會詳細說明:

1. 有關 MainDog 的負載問題在本論文當中並沒有被討論到，在共識的監控系統下的 MainDog 必須完成許多任務包含監控所有節點的健康狀態，監聽 drop event 來解決交易 race condition 的問題，透過 event sourcing 重啟還原節點。要能完成這些龐大的任務是需要相當程度的效能，這樣的效能再節點當中將會被均分給節點本身的服務，這樣的現象將可能導致整個系統的效能低落。而本研究因時間關係所以並沒有完成透過機器學習的數據分析，解析出較為閒置的節點並讓此節點有更高的機會被抽選為 MainDog，本研究最終所使用的共識演算法將會公平地將機會均分給所有節點。這是第一項本研究所沒達成的地方
2. 在本研究當中所定義的失效(failure)定義為:當節點在固定時間內無法對監控系統送出 heartbeats，並且對監控系統所送出的要求無法提供任何回應，亦或是監控系統監測到節點內部發生致命錯誤，無法發揮正常作業時都會被認定為失效。然而在正常的節點當中節點的失效是有程度之分的。並不太會像本研究所假設的非黑即白。所以本研究在移植到其系統上使用時必須多加留意失效程度的問題。
3. 本研究所適用的 Circuit Breaker 是針對從 client 端中阻擋過多 command 指令傳送至後台的系統保護機制，是只針對 CQRS 架構中的 Command 端做保護，並非是分散式系統當中用來連結節點與節點之間的系統保護機制，這點必須在此多加說明。

DOI:10.6814/NCCU202101070

4. 本研究當中的CQRS/Event Sourcing系統是模擬電商平台的交易系統做模擬，由於本研究所寫的程式皆為自行開發所以並無法得知引用的其他微服務架構下的系統效能評估以及失效回復的可行性，本研究只是提出這樣的想法增強在 CQRS/Event Sourcing 系統下的穩定度，所有未來的研究可朝向將此方法引入到其他系統當中進行評估。

DOI:10.6814/NCCU202101070

第五章 結論

EventSourcing/CQRS 是一個很有潛力的軟體架構樣式，透過軟體層面的設計將可以得到相應的好處。然而在分散式系統當中，若能搭配有效的監控系統，則更能維持系統的穩定度。本論文透過去中心化的思維方式設計了利用共識演算法做出了 WatchDog 監控系統，有效的降低單一失效的問題，而 CricuitBreaker 的設計更能夠增加系統承受高流量的修改指令，增加整個機制的穩定性。關於資料 race condition 的問題也應該有更加完善的解決方式，本文目前考量較簡單的面向，用相對較基本的方法解決此問題。未來，WatchDog 系統上可以繼續研究若是在 55% 毀損率的情況下該如何提高整體系統穩定程度如何做到另一種達成共識的方法將會是本文未來的研究課題，更進一步優化整個系統的穩定程度。

經過第五章的三項實驗測試中可以得出一些結論，本研究所設計的無論是 WatchDog 監控系統或是 Circuit Breaker 的設計，以及整體系統架構的錯誤回復機制都有成功增加系統上的穩定性，不過在設計上還是有些功能需要改善，WatchDog 在進行集體共識時是需要達到半數以上節點的存在才能成功完成共識，改進的方向可以從超過半數節點失去聯繫後轉換成另一種模式，變成強制某個節點直接成為 leader 並執行回復機制，也就是集中式與非集中式的結合版本。讓眾多節點達成共識所犧牲的效能也是一大問題，從圖 27 以及圖 29 當中可以觀察到當節點數逐漸增加時，其所需的回應時間以及所需封包數並非式線性增加，意思也就是說若是節點數再從 100 往上繼續增加，勢必系統重新復原的時間就會加倍增長，封包量也會即速上升，造成系統上的效能降低。產生這樣的問題有可能是本研究所撰寫的程式品質問題，或者是透過達成節點共識勢必會去犧牲這樣的效

DOI:10.6814/NCCU202101070

能，就如同區塊鏈為了達成共識必然會犧牲相應運算能力一樣。未來將可對共識監控系統與集中式監控系統進行整體系統上效能的評估與其他相應的實驗設計，透過這樣的實驗就能比較全面了解集中與非集中的優劣勢，並進一步評估取捨。

DOI:10.6814/NCCU202101070

參考文獻

[1] C. Richardson, Microservices Patterns
With examples in Java. Published 2017 by Manning Publications, October 2018.

[2] M. T. Nygard, Release It!: Design and Deploy Production-Ready Software. Pragmatic Bookshelf, 2018.

[3] Z. Long, "Improvement and implementation of a high performance CQRS architecture," in 2017 International Conference on Robots & Intelligent System (ICRIS), 2017: IEEE, pp. 170-173.

[4] B. Meyer, Object-oriented software construction. 1997.

[5] R. Martin, Agile Software Development, Principles, Patterns, and Practices. Prentice Hall, 2002.

[6] G. Young, CQRS Documents. 2010.

[7] M. Fowler. "Event Sourcing." https://martinfowler.com/eaaDev/EventSourcing.html (accessed.

[8] M. T. Nygard, Release It!: Design and Deploy Production-Ready Software. 2007.

[9] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," Journal of the ACM (JACM), vol. 32, no. 2, pp. 374-382, 1985.

[10] S. Gilbert and N. Lynch, "Perspectives on the CAP Theorem," Computer, vol. 45, no. 2, pp. 30-36, 2012.

[11] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm (extended version)," ed: Tech Report. May, 2014. http://ramcloud. stanford. edu/Raft. pdf, 2013.

[12] M. Overeem, M. Spoor, and S. Jansen, "The dark side of event sourcing: Managing data conversion," in 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017: IEEE, pp. 193-204.

[13] D. Meißner, B. Erb, and F. Kargl, "Performance Engineering in Distributed Event-sourced Systems," presented at the Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems, Hamilton, New Zealand, 2018. [Online]. Available: https://doi.org/10.1145/3210284.3219770.

[14] J. Rybicki, "Application of Event Sourcing in Research Data Management," 2018 : The Fourth International Conference on Big Data, Small Data, Linked

DOI:10.6814/NCCU202101070

Data and Open Data, pp. 22-26, 2018.

[15] S. Han and J.-i. Choi, "V2X-Based Event Acquisition and Reproduction Architecture with Event-Sourcing," in Proceedings of 2020 the 6th International Conference on Computing and Data Engineering, 2020, pp. 164-167.

[16] Y. Zhong, W. Li, and J. Wang, "Using Event Sourcing and CQRS to Build a High Performance Point Trading System," in Proceedings of the 2019 5th International Conference on E-Business and Applications, 2019, pp. 16-19.

[17] A. Bellemare, Building Event-Driven Microservices: Leveraging Organizational Data at Scale. O'Reilly, 2020-07.

[18] M. Barnkob and J. Krukow, "Event Sourcing and Command Query Responsibility Segregation Reliability Properties," Computer Science University of Aarhus.—2018.—C, pp. 21-39, 2018.

DOI:10.6814/NCCU202101070