初探適用於物聯網環境的 MQTT 長期交易機制

賴冠瑜
資訊科學系
國立政治大學
laimark900317@gmail.com

廖峻鋒
資訊科學系/數位內容學程
國立政治大學
cfliao@nccu.edu.tw

摘要

MQTT (Message Queuing Telemetry Transport) 近年來在物聯網領域受到廣泛的注目與採用，交易技術是商業應用上不可或缺的重要基礎，目前在系統實作上，MQTT 仍欠缺完整的交易支援。本論文主要目的為初步探索適用於智慧環境/物聯網應用場域的 MQTT 訊息式長期交易機制 (Sagas)，因應此類環境的多變需求，支援多種交易模式與部署型式，為提高設計的通用性，我們設計兼容於原有規格的訊息式交易機制，藉由建構 MQTT 通訊的正規模型，並於其上定義了長期交易的語意，根據理論上的探討，基於「不處理」、「交易鎖 (lock)」與「Short-circuiting」等策略進行實作與實驗，驗證其功能完備性、效能及實務上之可行性。

關鍵字: 交易, Sagas, MQTT, 物聯網

I. 前言

近年來資通訊技術快速發展，各式運算裝置已微型化並可直接或間接與網際網路互連，促成智慧環境 (Smart Environments) 願景實現。智慧環境中的「智慧」是由具計算能力的裝置與各式軟體服務透過網路連結起來，形成的物聯網 (Internet of Things, IoT) 所實現: 由佈建在環境中，相互連接的微型感測裝置 (Sensors) 感知環境及使用者資訊，傳送到具備較強運算與儲存能力的邊緣主機 (Edge Server) 儲存，再藉由這些感測資料推測使用者意圖，最後由環境控制或資訊呈現來滿足使用者的需求。

在物聯網環境中，MQTT (Message Queuing Telemetry Transport) [1] 是一個受到廣泛採用的應用層通訊機制。MQTT 是 ISO[1]與 OASIS (Organization Advancement Structured Information Standards)[2]標準，由 Eclipse 基金會 M2M Industry Working Group 管理。目前業界以 MQTT 3.1.1 為應用主流，最新版本為 MQTT 5 [2]。MQTT 通訊協定近年來相當普及，物聯網環境中大部份裝置間資訊流通都是依賴 MQTT Broker 做為通訊骨幹，這類系統被稱為訊息式系統 (Message-Oriented Systems)。

同時，近年來物聯網技術也被大量運用在各式日常生活的商業服務之中，例如商場、租車、停車、物流追蹤等。在商用的資訊服務系統中，交易 (Transaction) 機制可說是最關鍵的一項技術。所謂「交易」指的是一連串不可分割系統行為所形成的基本單位 (Unit of Work)，不是全部成功，不然就是全部放棄，且通常須符合 ACID(Atomic, Consistent, Isolated, Durable) 屬性。以租車服務為例，選車、認證使用者、付費、解鎖等一系列步驟，就是一個交易，任何一個步驟失敗都必須徹消全部的動作。交易機制的處理大致可分為單機式交易與分散式交易。單機式交易的處理在資料庫系統中已非常成熟 [3]; 較單純的分散式交易已經有廣被採用的標準做法，稱為兩階段確認 (Two-Phase Commit, 2PC) [4]。在一般的商用資訊系統中，進行分散式交易時，交易參與者明確知道彼此位址，以同步 (Synchronized) 的方式進行，大部份情況下會預期短期內就可得知交易的結果，且 2PC 機制符合交易的 ACID (Atomic, Consistent, Isolated, Durable) 性質。在某些狀況下，參與單位比較複雜或是費時較久的交易稱為「長期交易 (Long Transaction)」，此類交易會透過稱為 Sagas [5] 的機制來處理。

在以 MQTT 為通訊骨幹的物聯網環境中，交易參與成員經常包含許多臨時加入的異質裝置 (如使用者的手機)，且由於中間透過 MQTT Broker，因此交易參與者通常不會得知對方位址，彼此溝通也以非同步方式 (Asynchronized) 進行，交易成員難以得知交易對象，也不能確保何時會有交易結果，因此，有許多「長期交易」型態的需求。然而，使用 Sagas 處理長交易，只能保證符合 ACID 中的 ACD，但無法保證 Isolation [5]。換句話說，交易在執行時可能因互相交錯 (interleaved)，而在對同一端點中資料做修改時，易形成 race condition，導致資料錯誤。此時，有一個直覺的解決方案是以應用程式層次的鎖 (Semantic Lock)，在交易完成前，將所有相關資料鎖定，確保資料一致。然對長期交易來說，長時間鎖定資料對效能會造成重大衝擊。因此，應該依應用場景的特性，藉由犧牲一定程度的 Isolation Level，以兼顧效能。在本論文中，我們將能在訊息式系統中支援此類交易的機制稱為「訊息式長期交易處理機制」，此類交易處理，有不少細節需要探究，然就我們所知，目前在這方面，研究數量較少、進展較慢，技術也較不成熟完備 [6]–[9]，尤其是像 MQTT 這類系統的訊息式交易 ACID 屬性，甚至直到 2018 年才被明確定義 [10]。

以 MQTT 為通訊骨幹的物聯網環境有其特殊性，只靠一般性原則並不足以因應，例如，在智慧境境/物聯網場域中，裝置運處與儲存能力的差異相當大，並非所有的端點 (裝置) 都有能力負擔與交易相關的運算與儲存空間，計算能力較弱的裝置，若無適當支援，根本無法參與交易。此外，部份物聯網裝置服務涉及環控，改變的是實體環境，沒辦法和資料庫的欄位值一般，可以立即修改、還原，這也是上述相關研究提出之機制所未能處理的。本論文主要目的為分享目前我們初步探索此一領域之研究進展，呈現了一個適用於物聯網應用場域的訊息式長交易處理機制，此交易機制因應物聯網應用的多變需求，支援長交易型式，同時，在設計時也考慮交易放棄或失敗後的交易補償 (compensation)、錯誤重試與服務衝突提供了相關的處理。為提高設計的通用性，我們基於 MQTT 5 加以擴充，提出兼顧相容性的訊息式交易機制，將研究成果實現為原型系統，並說明了初步效能測試的結果。

本論文受下列國科會研究計畫經費部份補助，編號:110-2221-E-004-001-與 111-2420-H-004-003-

[1]ISO/IEC 20922:2016 https://www.iso.org/standard/69466.html

[2]OASIS Message Queuing Telemetry Transport TC https://www.oasis-open.org/committees/mqtt/

II. 相關研究

交易機制是商用資訊系統不可或缺的重要技術，有關一般性 RPC 為主的交易機制的設計與實現，[3] 已提出非常完備的參考指引，在此書中有一節提到 Queued Transaction，然而它只描述的最基本的單節點與單服務間交易機制。相較之下，訊息式交易機制，一般來說研究數量較少且進展較慢。[9] 是相關議題較早期的研究，主要探討的重點是如何將 Messaging System 與分散物件交易系統整合，提出 MMT(Middleware Mediated Transactions) 機制 [7]。針對時間與空間的解耦這個經典問題，比較重大的突破是出自於 [6]，提出透過 Census 協商階段來解決無法認知交易對象問題，但它是植基於特定的產品所設計。[8] 延續此研究，提出了 TOPS(Transaction-Oriented Publish/Subscribe) 機制，它比較值得注意的改善是在於限制交易的參與者與交易的影響範圍 (Transaction Scope)，相關機制也是基於特定的產品所設計。基於問題的複雜性，相關研究在過去長期停留在實作性的探索，直到 2018 年才由 [10] 提出訊息式交易的 ACID 屬性的正規定義。本研究主要探索的範圍是適用於智慧環境/物聯網的訊息式交易，就我們所知，目前並沒有相關議題的研究成果被提出。

III. 系統設計

本節將說明目前初步設計，可以在 MQTT 之上進行長期交易的機制。為精確描述交易機制，我們首先定義了一組 MQTT 的描述表示式，在此表示式之上，我們再表述如何透過 MQTT 進行長期交易的精確語意。

A. MQTT 系統模型

以下我們首先介紹如何描述 MQTT 及其通訊行為。首先，令一 MQTT 系統以 Tuple: (E, B, T, M) 表示，其中 B 代表系統中 MQTT Brokers 所形成的集合，理論上一個系統中可有一至多個 Brokers，邏輯上可視為一個，我們用 b 代表，其中 $b \in B$; E 代表透過 b 彼此通訊的端點 (Endpoints)，其中，E^P, E^S 分別代表訊息發佈者 (Publisher, $e^P \in E^P$) 與訊息訂閱者 (Message Subscriber, $e^S \in E^S$)，同時訂閱和發佈訊息的端點稱為 Agent，以 e^A 表示，而其宇集合以 E^A 表示。由上可知，$E = E^S \cup E^P \cup E^A$。此外，$M = \{m_1, m_2, ...\}$ 代表透過 b 傳送的訊息集合，通常訊息會針對特定的主題 (topic) 來發佈與訂閱，我們用 $T = \{t_1, t_2, ...\}$ 來代表 topic 集合。其關係如圖 1所示，在圖中所呈現的系統包含三個主題 (t_1, t_2, t_3)、二個訂閱者 (e_1^S, e_2^S)、二個發佈者 (e_1^P, e_2^P) 與二個 Agents(e_1^A, e_2^A)。

接下來定義 MQTT 訊息格式，主要包含二種形式，第一種是單純只包含標頭 (H) 和 Payload(β)，可定義如下:

$$m(H, \beta) \equiv \kappa(H, \beta)$$

同義符號 $(\equiv)$ 的左方為應用程式、API 的觀點，我們稱為「設計層次」; 右方為實際傳送的訊息，屬於實作面、網路層次的表述，我們稱為「實作層次」。由這樣的高低層次並陳表述方式，可以表示出，一個「設計層次」的行為，在「實作層次」上，是如何具體透過 MQTT 端點間訊息交換來完成。上式中 κ 代表 MQTT 定義的 Control Packet Type，例如:PUBLISH、CONNECT 等。另一種 MQTT 訊息類型則額外包含回應碼 (reason code)，主要用於明確表述訊息解讀與執行的結果:

$$m'(H, \beta) \equiv \kappa'(r, H', \beta')$$

基於上面的定義，我們可以描述訊息傳送如何被實現。例如圖 1中，由端點 e_1^P 送訊息 m 給 Broker b，如式 (1)，

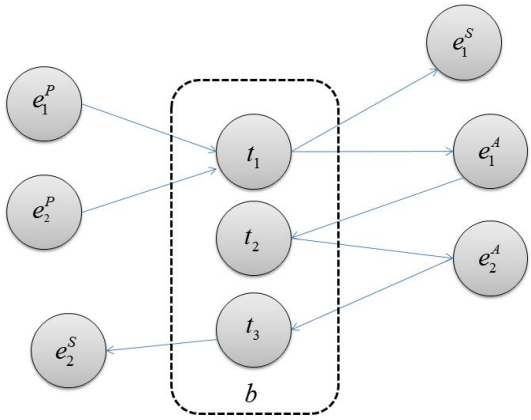

圖 1. MQTT 系統模型示意圖

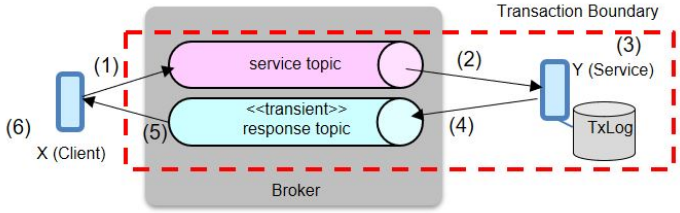

圖 2. 訊息式長期交易: 單一服務

接下來 b 再將此訊息轉送給 e_1^S，如式 (2)，與 e_1^A，如式 (3):

$$e_1^P.m(H, \beta) \rhd b \equiv e_1^P.\kappa(H, \beta) \rhd b \quad (1)$$

$$b.m(H, \beta) \rhd e_1^S \equiv b.\kappa(H, \beta) \rhd e_1^S \quad (2)$$

$$b.m(H, \beta) \rhd e_1^A \equiv b.\kappa(H, \beta) \rhd e_1^A \quad (3)$$

在需要回傳確認 (如 PUBACK) 的情況下，每次訊息傳送都需要二個步驟才能完成，其實作層次的定義如下:

$$e^P.\kappa(H, \beta) \rhd b \rightarrow b.\kappa'(r, H', \beta') \rhd e^P$$
$$b.\kappa(H, \beta) \rhd e^S \rightarrow e^S.\kappa'(r, H', \beta') \rhd b \quad (4)$$

其中，$\rightarrow$ 為時序運算元，例如 $x \rightarrow y$ 代表 x 先發生之後 y 才發生。

B. 單服務訊息式長期交易

有了上面的正規表示式，我們就能進一步描述如何在 MQTT 上進行交易。我們先從包含二個交易端點的狀況開始，如圖 2，我們定義參與交易的端點中，要使用服務的端點 X 稱為 Client，提供服務的 Y 端點稱為 Service。X(Client) 想要存取 Y(Service) 的服務，因此傳送訊息給 Y，這個訊息的內容包含了對 Y 所需提供服務的具體指示與描述，Y 根據訊息內容加以處理後，透過 response 回傳結果 (可能是成功或失敗)，上述過程形成一個交易，交易的範圍 (Transaction Boundary) 由虛線標示，在交易的範圍內，若何一個環節失敗 (Abort)，則之前的節點必須回復原始的狀態。具體而言，X 透過 service topic 傳送服務請求訊息到 Y，接下來 Y 執行邏輯後，將結果回傳到 response topic。

交易通訊機制將透過 MQTT 5 的 request-response 來實現，response topic 每次隨著 service topic 動態建立，所以

```
1  upon message<request> m do:
2    if(m.tx_id == tx_version + 1):
3      try:
4          results = prepare(m) // handle business
               logic, write to event log
5          commit(m)          // SAGAS: commit first
6          m' = createMessage(committed, results)
7      catch(e):
8        m' = createMessage(abort, e)
9      finally:
10       tx_version += 1
11       send(m')
12
13 upon message<abort> m do:
14   if(m.tx_id == tx_version && tx_mode == SAGAS):
15     compensate(m) // react to client abort
```

虛擬碼 1. 式 (5)-(7) 服務端點 (Y) 的虛擬碼

```
1  main:
2    send(createMessage(request) // create and send tx
         request to the service
3
4  upon message<abort> m do:
5    // deal with transaction failure
6
7  upon message<committed> m do:
8    send(createMessage(finish)) // or
         send(createMessage(abort)) if necessary
```

虛擬碼 2. 式 (5)-(7) 呼叫端點 (X) 的虛擬碼

在圖中標記為 transient。整個過程中可能發生 Abort 的環節包含:(1)X 到 Broker(2)Broker 到 Y(3)Y 收到訊息後處理的過程 (4)Y 到 Broker(5)Broker 到 X(6)X 收到回傳訊息後的處理。在 MQTT 中若以 QoS-2 傳送訊息，則可確保端點到 Broker 間的訊息傳送為 exactly once，如此一來，可能會 abort 的狀況只會發生在 (3)Y 收到訊息後處理的過程與 (6)X 收到回傳訊息後的處理。其中，在 (6) 的狀況下，若 X 傳入的訊息，是去修改 Y 的狀態，則此時需要讓 Y 進行回復，稱為補償動作 (Compensation)。其主要運作機制是 Y 進行變動前，將變動儲存在本地的 transaction log(TxLog) 中，X 發出 abort 後，Y 根據 event log 中找出上次儲存的變動，加以復原。根據以上的互動描述，其正規表示式如式 (5)-(7) 所示。

$$X.m^{request}(H,\beta) \rhd b^{Y} \rightarrow b^{Y}.m^{request}(H,\beta) \rhd Y \tag{5}$$

$$Y.m^{respeonse}(H',\beta') \rhd b^{Y'} \rightarrow b^{Y'}.m^{respeonse}(H',\beta') \rhd X \tag{6}$$

$$X.m^{finish}(H'',\beta'') \rhd b^{Y} \rightarrow b^{Y}.m(H'',\beta'') \rhd Y \tag{7}$$

式 (5)-(7) 中 X 與 Y 的交易處理邏輯整理如虛擬碼 1與 2，在此我們採用 [11] 提出的分散式演算法虛擬碼格式。其中，虛擬碼 1在第 2、10 與 14 行檢查 tx_version 主要目的為避免 consistency 問題，在未完成目前交易前，暫不參與後續交易。虛擬碼 1採用長期交易方式，也就是 Sagas，此種方式採取比較樂觀的機制，接受 partial commit，也就是所有服務接到訊息就 commit(虛擬碼 1, 4-6 行)，若之後有任何失敗，由 Client 統一要求所有服務進行補償機制 (compensation)(虛擬碼 1, 14-15 行)。在實際寫作程式時，尚應檢查每個進來的訊息是否來自同一個 Client，這部份必須與 MQTT 本身的認證機制結合，因為篇幅有限，並在呈現上聚焦於交易處理，虛擬碼 1並未呈現此一部份。

基於上述設計，我們可以進一步處理幾個特殊議題。首先是服務是否可補償 (compensable) 的問題。目前我們假設每一個服務在交易確認後，都可以藉由補償 (compensation) 回復，但實務上並不一定，例如租用物品一旦解鎖，就可能被使用或取走。在無法回復時，有二種可能的處理方法，首先是禁止 SAGAS 的使用，改採 Two phase commit (2PC)，如此一來會等到全交易確認完成才 commit，若採用此方法應修改虛擬碼 1的 4-6 行，遇到 Sagas 就改 2PC; 另一個可能性是仍允許 Sagas，但在補償需求發生時，發出系統警告，人為介入處理，此時就必須將 15 行置換為警告訊息的發出。第二個問題是服務是否可重覆執行 (idempotent)，在智慧環境中有些動作失敗時是可以重覆嘗試的，例如解鎖/上鎖，或是更新到同一個狀態。但有些動作是不可重覆執行 (non-idempotent) 的，例如環控的加溫、加濕、計數器、付費處理，這些動作在邏輯上做一次和做多次其結果是有區別的。在我們的設計上，服務若聲明是可重覆執行 (idempotent) 的動作，則可允許一定次數的重試，增加系統的強健性。在實作上，可在虛擬碼 1中的第 8 行加入 send 指令配合重試的計數器，在一定次數的嘗試後再提出 abort。

C. 多服務訊息式長期交易

接下來，我們將討論擴充到包含多個服務的訊息式交易。如圖 3，在這個場景中，X(Client) 想進行的交易包含 Y(Service1) 與 Z(Service2) 二個服務，虛線部份為交易範圍，交易範圍內若何一個環節失敗 (Abort)，則之前的節點必須回復原始的狀態。單一服務交易與多服務交易最大的不同在於 Client 端的控制邏輯會較為複雜。其訊息傳送的模式為:

$$\Sigma_{s_i \in S}[X.m_{s_i}^{request}(H,\beta) \rhd b^{s_i}] \rightarrow \Sigma_{t_i \in T}[b^{s_i}.m^{t_i.request}(H,\beta)] \rhd s_i \tag{8}$$

$$s_i.m_{s_i}^{respeonse}(H',\beta') \rhd b^{s_i'} \rightarrow b^{s_i'}.m^{respeonse_{s_i}}(H',\beta') \rhd X \tag{9}$$

$$\Sigma_{s_i \in S}[X.m_{s_i}^{finish}(H,\beta) \rhd b^{s_i}] \rightarrow \Sigma_{t_i \in T}[b^{s_i}.m^{t_i.finish}(H,\beta)] \rhd s_i \tag{10}$$

其中 $\Sigma_{s_i \in S}$ 代表 $\forall i, s_i \in S$ 都做 [...] 間的行為。$s_i \in S$ 代表參與交易的服務，$t_i \in T$ 代表每個服務的子交易，b^{s_i} 代表 broker 上第 i 個服務的 service topic，$b^{s_i'}$ 代表 broker 上第 i 個服務的 response topic。虛擬碼 3列出了多交易時 X 端的處理機制，在多服務交易的情況下，需要每個交易都成功才算是成功，因此較為複雜的部份在於一對多通知的處理，及服務交易狀態的等待，也就是需要統計比對是否所有交易已完成準備，才能送出 commit。在失敗時，也需要一併通知其它相關服務 abort 的訊息。Services 的部份則和單交易相同，因此可直接沿用虛擬碼 1的設計。

IV. 系統實作與實驗

基於上一節提出的交易設計與虛擬碼，我們開發出了一個系統原型，並在一個預先設定好的環境下進行了數項實驗。實驗環境架構與圖 3的多交易環境相同，交易包含 Service 1 與 Service2，其後端各包含一個資料庫。一個交易是在 Service 1 與 Service 2 都成功的狀況下才算是成功，反之就算是失敗，而失敗時，必須進行補償動作，以將系統回復到交易前狀態，一個 Saga 成功或失敗補償後，都算

```
main:
   tx = createTransaction(tx_id)
   tx.services = [request_1, request_2,...,request_n]
   for each request in services:
      send(createMessage(request, [service_topic_1,
          service_topic_2,...service_topic_n])

upon message<abort> m do:
   // deal with transaction failure
   ...
   // abort all participating transactions
   send(createMessage(abort, [service_topic_1,
       service_topic_2,...service_topic_n]))

upon message<committed> m do:
   send(createMessage(finish), m.service_topic)
   // or send(createMessage(abort, [service_topic_1,
       service_topic_2,...service_topic_n])) if
       necessary
```

虛擬碼 3. 多服務交易呼叫端點 (X) 的虛擬碼

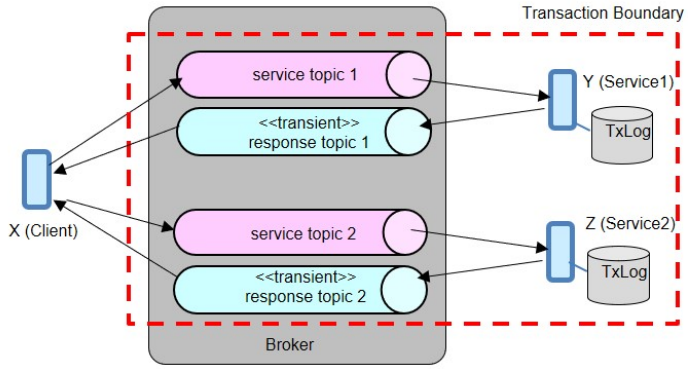

圖 3. 訊息式交易: 多服務

完成交易。整個過程是由 Orchestrator（協調器）元件，也就是圖 3中的 X，來主導。Orchestrator 與服務之間都是透過 Mosquitto MQTT broker 傳遞訊息，並且由 Orchestrator 暫存 Saga 狀態，包含補償所需資料、Sagas 當下的執行階段等，Orchestrator 與服務端點都是以 node.js 實作。

如同之前的討論，長期交易面臨 Isolation 無法確保的問題，在前一個長期交易 (Saga) 完成前，Orchestrator 可能又啟動另一個 Saga，造成有兩個 Sagas 同時間存在於系統裡，則稱這個狀況為 Interleaved Sagas。在遇到這種狀況時，有三種可能的因應策略，分別為「不處理」、「交易鎖 (lock)」與「Short-circuiting」。我們針對這三種狀況進行了不同的實作與實驗。一開始會由 Orchestrator 使用迴圈連續發出交易請求，並實驗在遇到 Interwoven Sagas 時，分別採取不處理、lock 與 short-circuiting 的應對方法下，在以下狀況發生時，完成不同交易數所花的時間，並計算它們每秒完成的交易數量 (transaction per second)。

- 所有服務正常 (normal)
- 所有服務失敗並且都補償 (all rollback)
- Service 1 失敗但沒有更動資料庫，所以只需補償 Service 2(Service 1 reject)
- Service 2 失敗但沒有更動資料庫，所以只需補償 Service 1(Service 2 reject)

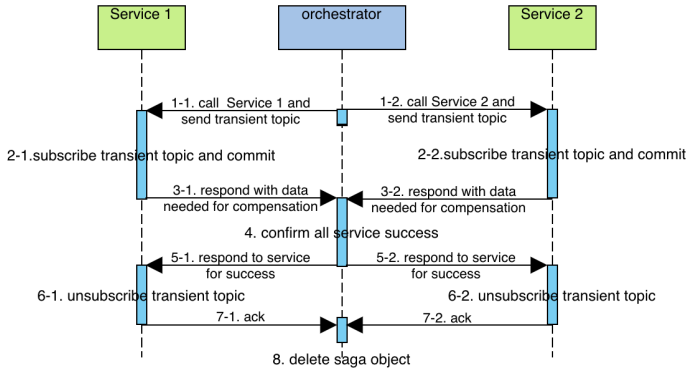

圖 4. 不處理 Interleaved Sagas: 成功

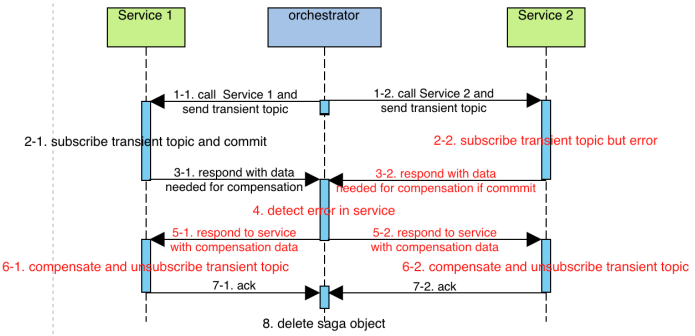

圖 5. 不處理 Interleaved Sagas: 失敗

- 全部都沒有更動資料庫，所以都不需要補償 (all reject)

A. 策略 1: 不處理 *Interleaved Sagas*

圖 2 是在不處理 Interleaved Sagas 的情況下，所有服務都成功的流程。一開始會由 orchestrator 先建構該次 Saga 與其相關資料。這些資料包含:

- State: 為該服務在該次 Saga 中的狀態，0 代表該服務還沒有回覆，1 代表成功 commit ，-1 代表該服務有錯誤發生。
- Ack: 負責儲存服務是否已經回傳 ack，false 表示未回傳，true 表示已回傳。
- Reject: 因為服務可能因錯誤而沒有更改到資料庫，因此就不需要補償，所以需要這個變數來確認有無這個狀況發生。
- TransientId: 因為端點之間溝通都是使用 MQTT，所以 orchestrator 需要知道 services 回傳的 topic，就利用這個 transientId 來儲存。
- compensate: 由這個陣列來儲存若服務發生錯誤時，需要用來補償的一連串操作，陣列的內容，由 orchestrator 和 service 之間協調共識即可，可以依照開發者自行定義。

在所有服務都成功的狀況下如圖 4，Orchestrator 會在 ack 完後刪除 Saga 物件。當有服務發生錯誤而失敗時，以 Service 2 發生錯誤為例如圖 5，orchestrator 在第 4 步中發現有服務失敗時，會利用存在 Saga 物件裡的從第 3 步拿到的補償服務所需資訊，在第 5 步中傳給服務端點進行補

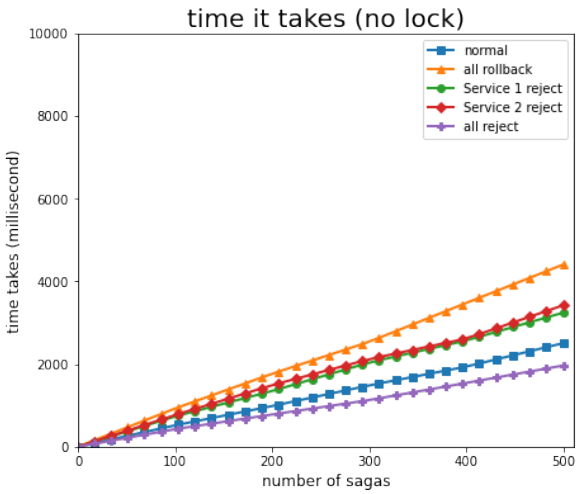

圖 6. 不處理 Interleaved Sagas: 交易時間

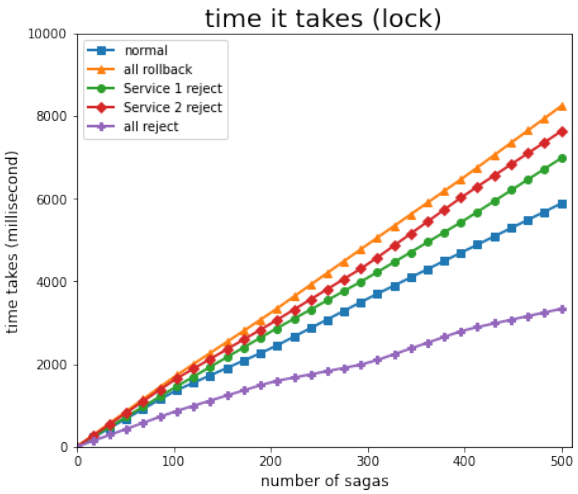

圖 8. 交易鎖: 交易時間

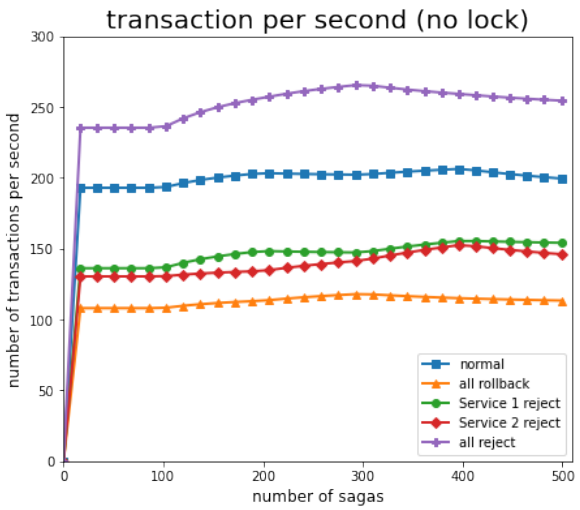

圖 7. 不處理 Interleaved Sagas:TPS(Transactions per Second)

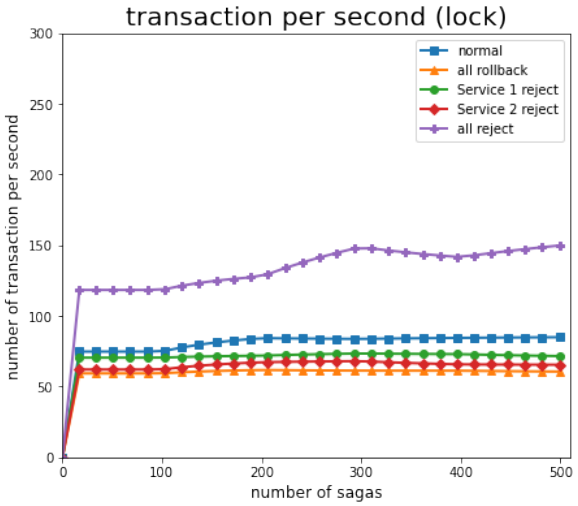

圖 9. 交易鎖:TPS(Transactions per Second)

償。若在第 2 步時，該服務因錯誤而沒有對資料庫進行更改，則在第 5 步時就不會回傳補償資料，第 6 步也不需要進行補償。

圖 6為完成指定數量的交易所需時間的實驗結果，而圖 7為每秒可完成的交易數量，可以發現，all reject 因為沒有動到資料庫，所以花的時間最少，normal 有動到資料庫，但因為不需補償，所以所花時間第二少。Service 1 reject 只需要補償 Service 2，而 Service 2 reject 只需要補償 Service 1，因為 Service 1 的服務是插入一行資料，補償時需刪除資料，Service 2 因為服務是更新一行資料的數字，補償只需要更新資料回過去的狀態即可，所以 Service 1 reject 所花時間第三少，Service 2 reject 所花時間第四少，all rollback 因為要補償所有的服務，對資料庫的更動最多，所以所花時間也最多。

B. 策略 *2:* 交易鎖

這個策略在服務成功與失敗時的流程以及 Saga 物件所儲存的資料，與圖 4是一樣的，但會在步驟 2 開始到步驟 7 結束前，使用 async-mutex module 的 mutex，在當次的 Saga 處理完後釋放 mutex，若有其他多個服務請求正在等候 mutex，它會優先讓較早 acquire 的服務存取，進行步驟 2 與隨後的步驟。即使服務端使用了 short-circuiting 或是 lock 的機制，在沒有 Interwoven Sagas 的正常情況下，在服務沒有出錯的流程則如圖 4，在有服務出錯的情況下的流程則如圖 5。

圖 8為在使用 lock 的情況下，完成指定數量的交易所需要的時間，而圖 9為在使用 lock 的情況下每秒可完成的交易數量，normal、all rollback、Service 1 reject、Service 2 reject、all reject 所需時間的排名，和無視 Interwoven Saga 時是一樣的，但是因為 Service 不能同時處理多個 Sagas，要等前一個 Sagas 所佔用的 lock 釋放，才可以處理新的 Saga 所以需要時間更長，每秒可以完成的交易數量較少。

C. 策略 *3:Short-circuiting*

Short-circuiting 機制是在有 Interleaved Sagas 時，把除了已佔有服務使用權以外的 Sagas 判定為失敗。以 Service 1 為例，在服務遇到 Interwoven Saga 時如圖 10，同樣會使用在 lock 機制時用到的 async-mutex module，利用 mutex.isLocked() 判斷是否有其他 Saga 正在服務上運行，若有，則會回傳失敗訊息給 orchestrator，並且直接判斷

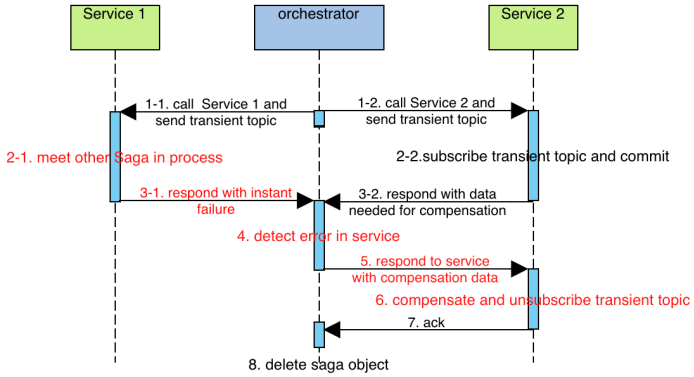

圖 10. Short-circuiting

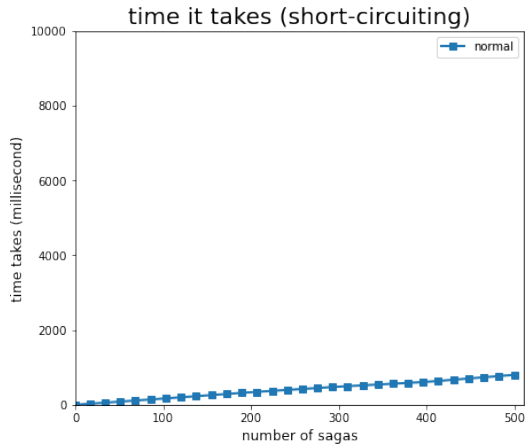

圖 11. Short-circuiting: 交易時間

這次的 Saga 失敗，並記錄到 Saga 物件中，隨後補償原本成功 commit 的 Service 2，但不會再和 Service 1 互動，orchestrator 等待 Service 2 回傳 ack 之後，同樣會刪除 Saga 物件。

圖 11為在使用 short-circuiting 的情況下，完成指定數量的交易所需要的時間，而圖 12為在使用 short-circuiting 的情況下每秒可完成的交易數量。因為只有第一個 Saga 會跟服務的資料庫互動，之後的 Sagas 都和第一個 Saga 交錯重疊 (Interwoven)，造成啟動 short-circuiting 的機制而提早結束，導致 normal、all rollback、Service 1 reject、Service 2 reject、all reject 等情況下，只有第一個 Saga 的時間是不一樣的，所以在這邊只有實驗 normal 的情況，此原因也導致使用 short-circuiting 的每秒完成交易數量較前述兩個方法多。

V. 結論

本論文藉由理論上的分析與實務上的實作驗證，初步探索 MQTT 之上進行長期交易的機制。首先我們建構了 MQTT 通訊的正規模型，並於其上定義了長期交易的語意。根據理論上的探討，基於「不處理」、「交易鎖 (lock)」與「Short-circuiting」，進行了一系列的實驗。由實驗結果，我們得知，交易因失敗而補償時，所需時間較長，而處理 Interwoven Sagas 的方式，也會影響整體機制的效能，使用 lock 時，可能會因為一個 Saga 的影響，造成其他的 Interwoven Sagas 特別慢; 而因為 short-circuiting 會直接使新的 Interwoven Sagas 失敗，若想完成該 Sagas，需要確認

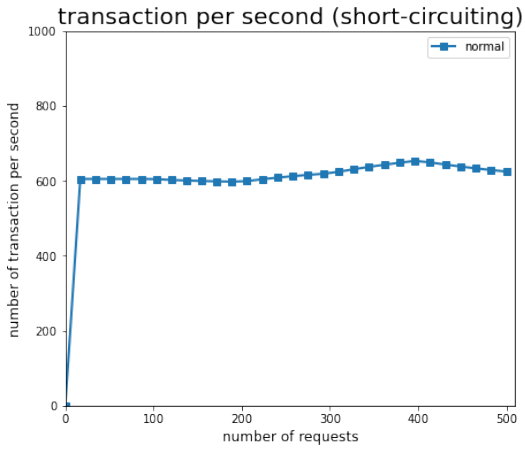

圖 12. Short-circuiting:TPS(Transactions per Second)

有無其他 Sagas 正在執行，或是讓 orchestrator 重新啟動 Sagas，也可能會造成效能的影響，所以須根據不同業務需求，進行機制的選擇。目前我們的交易模型相對單純且功能受限，未來將討論較複雜如巢狀交易的狀況，並擴大實驗的規模，以利進行更進一步的研究。

參考文獻

[1] A. Banks and R. Gupta, “Mqtt version 3.1.1,” *OASIS standard*, vol. 29, p. 89, 2014.

[2] A. Banks, E. Briggs, K. Borgendale, and R. Gupta, “Mqtt version 5,” *OASIS standard*, 2019.

[3] P. Bernstein and E. Newcomer, *Principles of Transaction Processing*, ser. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2009. [Online]. Available: https://books.google.com.tw/books?id=LmHgK5KKrQQC

[4] B. W. Lampson, “Atomic transactions,” in *Distributed Systems - Architecture and Implementation, An Advanced Course*. Berlin, Heidelberg: Springer-Verlag, 1981, p. 246–265.

[5] H. Garcia-Molina and K. Salem, “Sagas,” *ACM Sigmod Record*, vol. 16, no. 3, pp. 249–259, 1987.

[6] L. Vargas, L. I. Pesonen, E. Gudes, and J. Bacon, “Transactions in content-based publish/subscribe middleware,” in *27th International Conference on Distributed Computing Systems Workshops (ICDCSW’07)*. IEEE, 2007, pp. 68–68.

[7] C. Liebig and S. Tai, “Middleware mediated transactions,” in *Proceedings 3rd International Symposium on Distributed Objects and Applications*. IEEE, 2001, pp. 340–350.

[8] Y. Shatsky, E. Gudes, and E. Gudes, “Tops: A new design for transactions in publish/subscribe middleware,” in *Proceedings of the Second International Conference on Distributed Event-Based Systems*, ser. DEBS ’08, New York, NY, USA, 2008, p. 201–210.

[9] S. Tai and I. Rouvellou, “Strategies for integrating messaging and distributed object transactions,” in *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2000, pp. 308–330.

[10] M. Jergler, K. Zhang, and H.-A. Jacobsen, “Multi-client transactions in distributed publish/subscribe systems,” in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 120–131.

[11] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*. Springer Berlin Heidelberg, 2011. [Online]. Available: https://books.google.com.tw/books?id=Y8lHVFeJ6EQC