RESTful 範例

by

Triton Ho 

前言

- 請配合 RESTful 教學淺談一起服用
- https://github.com/TritonHo/slides/blob/master/Taipei%202019-06%20talk/RESTful%20API%20Design-tw-2.2.pdf

今天範例

- 老闆：我想做一個讓人 online 玩 UNO 的系統
  - 謎之聲：請先跟 UNO 相關版權持有人談授權
  - 註：本文單純以 UNO 卡牌系統講解， UNO 商標和遊戲屬其相關持有人
- 補充：像 LOL / counter strike 這些 realtime action game，不是今天 RESTful 的討論範圍
  - 這些遊戲很多 UDP 路線
  - 對動作遊戲：動件連貫性和反應遠比資料正確重要

架構選擇 1

- 因為 client-side 需要接收其他人的出牌資訊，所以用 web-socket 來接收 service push
- 如果要 UserA 接收到 UserB 的出牌.....
  - 選擇 1A ：同一個牌局的用戶的 WebSocket / HTTPS request 全連上同一台 Application Server（簡稱 AS ）
  - 選擇 1B ：要麼用戶的通信能由不同的 AS 負責， AS 之間以 RabbitMQ 通信
  - 每秒 polling 問一下 server 最新狀態，在 2019 不是一個可被接受的方案

架構選擇 2

- 用戶已經建了 WebSocket 來接收卡牌更新
- 那麼.....
  - 選擇 2A ：用同一個 WebSocket 來出牌（或任何用戶行動）
  - 選擇 2B ：出牌用傳統的 RESTful API

同一 AS（看起來）好處

- 要麼同一個牌局的用戶的 WebSocket 全連上同一台 AS（簡稱 AS）
- 好處：
  - 不用 RabbitMQ 了， UserA 出牌了，直接把資訊經 WebSocket 傳給 UserB / C / D 就好了
  - 不用 Redis 做 caching ，反正會用上這牌局資料的用戶全是集中在同一台 AS 上，直接把牌局資料留在 AS 的 local memory 就好

同一 AS 壞處

- 強迫 User 只能進特定的 AS ，違反 RESTful 中 Stateless AS 守則
  - Stateless AS 不單單要求 AS 不能存放資料，也要求 Request 能讓任何一台 AS 來工作
- User X 只能由特定 AS 負責，那就代表 Load Balancer 那一層要紀錄 {UserX, AS123} 這樣子的 mapping
  - 這樣子的 LB 成本很貴的
  - LB 層如果要加開新機器，怎把這 mapping 資料同步？

同一 AS 壞處（續）

- 由單一 AS 負責一個牌局，那代表在這 AS 上的全部牌局都結束前，這台 AS 沒法 graceful shutdown
- 名句：死人是不能說「我死了」
  - 一台 AS 趴掉後，在 heartbeat 判定其真的死亡前，在這台 AS 上全部牌局都沒法轉給別的 AS 負責
  - 如果 heartbeat 設定太短，則容易有 false positive ，觸發不必要 AS 轉移
- 在 AS 轉移時，系統需要有一個 coordinator 來判定新的轉移目標，然後再更新 LB 層的 mapping
  - 這現實上很煩的

WebSocket 來出牌好處

- 你的 Application 層只有一個 Component
- 我想不出來了 ~

WebSocket 來出牌壞處

- 這違反了 single responsibility principle
- 如果 Websocket 只用來接收 server push ，那麼 websocket 可以非常簡單
  - 單純簡單用戶身份驗證後， listen RabbitMQ 特定 channel ，把收到的東西全部 relay 給用戶就好
  - 幾乎沒有 Application Logic ，所以極少改動
  - 不管你寫 UNO 還是飛行棋，這樣的 Websocket Component 都能無痛重用
- 如果 WebSocket 有 Application Logic
  - 那麼，你就會因為 App Logic Change / Bug Fix 而需要做 production deployment
  - 在 deployment 時，你要：
    - 等待這個 Server 全部的 Websocket 自然中斷？
    - 把用戶的 Websocket 強行中斷，叫他們連到新的 AS

    以上二者都不是好主意

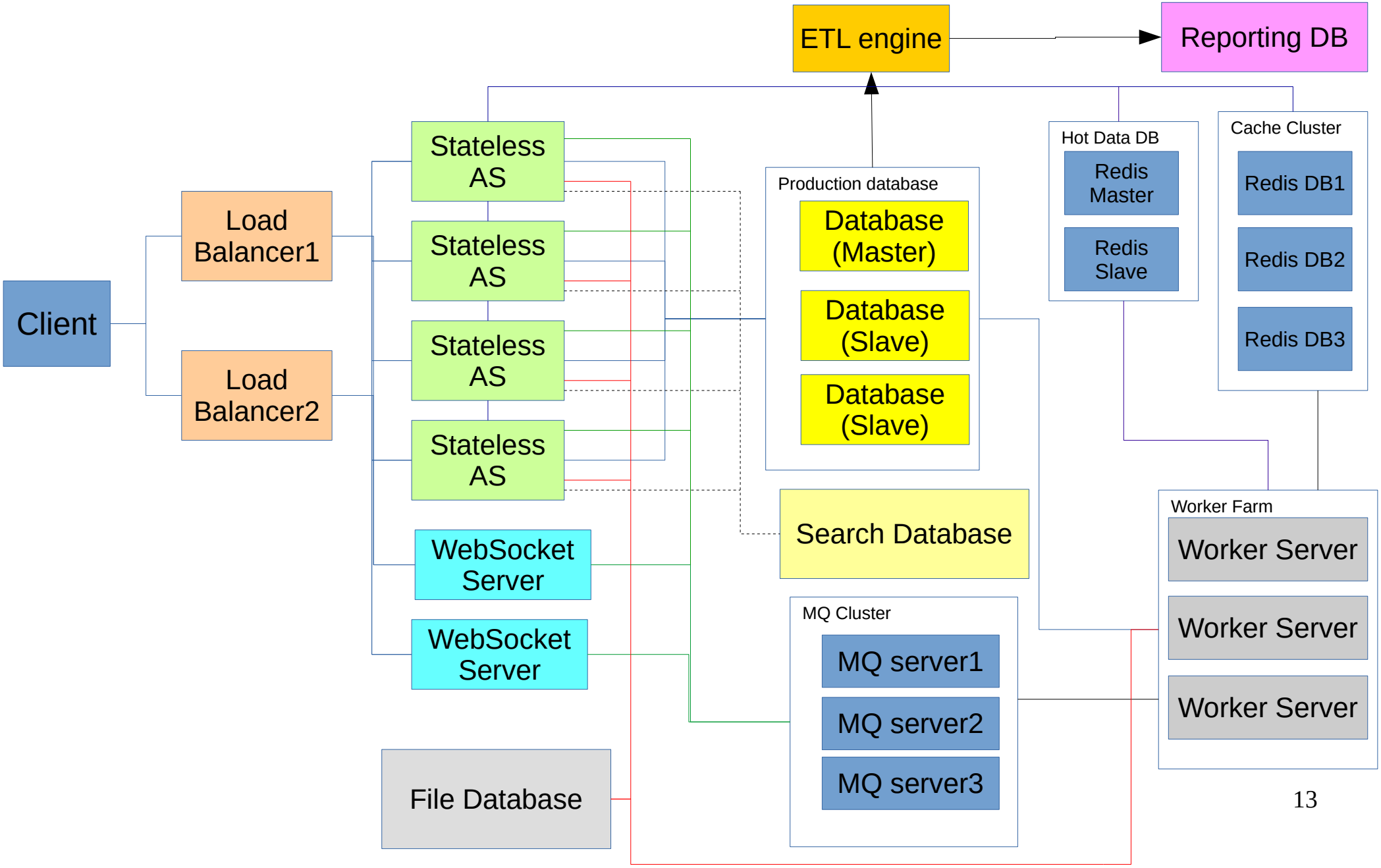
中場小感想

- RESTful 不是裝偉大裝潮流的哲學
- RESTful 是讓你省下麻煩，迴避潛在地雷的重要架構設計原則

合乎 RESTful 的架構選擇

- WebSocket 單純用來接收其他玩家的改動
- 要出牌是用傳統的 RESTful API
- AS 是 Stateless ，不同的用戶的 Websocket 會散落在不同的機器上

最終系統架構圖，不解釋～


ETL engine
Reporting DB
Client
Load Balancer1
Load Balancer2
Stateless AS
Stateless AS
Stateless AS
Stateless AS
Production database
Database (Master)
Database (Slave)
Database (Slave)
Hot Data DB
Redis Master
Redis Slave
Cache Cluster
Redis DB1
Redis DB2
Redis DB3
Search Database
WebSocket Server
WebSocket Server
MQ Cluster
MQ server1
MQ server2
MQ server3
Worker Farm
Worker Server
Worker Server
Worker Server
File Database

RESTful API endpoints

- /v1/games/{gameId} GET
  - 返回最新的遊戲狀態，包括：
    - 最新的 stepId
    - 自己的卡牌
    - 其他玩家狀態
- /v1/games/{gameId}/steps/{stepId} POST
  - 用來出牌的 endpoint

出牌流程

- User A 把 Request 丟到 AS
- AS 從 localcache / redis 拿回 (StepId - 1) 遊戲狀況
- AS 檢查和處理後，把出牌資訊存到 Redis（和資料庫）
- AS 把資訊丟到 RabbitMQ 上的 GameX channel
- User B / C / D 其 Websocket server ，一直在 listen GameX channel
  把從 GameX channel 收到的資訊轉給 client side

Websocket 通訊

- 在 User A 出牌時，其他用戶除了收到發出來的牌的資訊，還必須包含 StepId

小技巧：面對 WebSocket 斷線

- 流程：
  - 建立 Websocket 連線
  - 呼叫 /v1/games/{gameId} GET ，拿回遊戲狀態
  - 在拿到 Get response 前， WebSocket 所收到的東西全前保留下來
  - 拿到 Get response 後
    - WebSocket 收到的訊息，其 stepId 少於 GET response 的 StepId 全都丟掉不用管
      StepId >= GET response 的，則用來 replay 來建立現在最新狀態

小技巧：提升 local cache hit rate

- 今天的 LB 層，可以理解 Request 的 URI（並且維持便宜）
  - 所以 LB 層知道 Request 的 gameId 不是什麼困難事
- 小技巧： LB 層分派 Request 的手段不是 random ，而是基於 gameId 的 Consistent hashing
  - 同一 gameId 的 Request ，就會被丟到同一台 AS 上
  - Consistent hashing 不用把歷史存起來，不吃 Memory 的
  - 即使 AS 當掉， Request 自動丟到別的 AS 上

小技巧：面對 Conflict

- UNO 規則：跳插 (Jump-in)
  - 若玩者手中有牌與桌上的牌完全一模一樣（同色同字），則喊「 cut 」後可不依順序搶先出該牌
- 當 2 個玩家對 StepId = 87 發出 POST request 時
  - 先發者先勝（廢話），後發者拿到 HTTP 409(Conflict)
- 怎肯定只有一個人在 StepId = 87 成功出牌：
  - 把出牌資訊存到 Redis 時，用上
    SETNX gameX-87
    只有第一個人才能寫進去

完