Exam, message queues, notifications, & step functions

COSC349—Cloud Computing Architecture

David Eyers

COSC349 Exam

- There are now four past papers available…
  - **Three hours**; **100 marks** across **8 questions**; answer all questions
  - **In-person** exam
- The 8 questions relate to different topic areas
  - Expect some alignment with the lecture structure
  - Each question has parts, and potentially subparts:
    - lots of small questions & no "Cloud computing. Explain. (20 marks)"s
- Exam is on lecture material, not labs, or assignments

Answering COSC349 exam questions

- Marking approach predictable from marks allocated:
  - "Describe three reasons for… (6)"—assume 2 marks per reason
- Structuring answers as bullet points rather than prose is OK provided that the linking to the question is clear
- Suggest you plan to do multiple passes through exam
  - Answer questions you're comfortable with first
  - Some questions are intended to be more challenging

Studying for COSC349

- Ensure you can answer the questions in the learning objectives presented at the start of each lecture
    - This should help highlight the key, core concepts
    - Some of the more detailed lecture material is provided for completeness, to provide context and for those interested
        - (… but some of it is very technically detailed)
- Come to tutorials if you're unsure about your answers to given learning objectives
    - I’m very keen to help, but I can’t form your answers for you

Learning objectives

- Understand that cloud applications will usually be built from **many different software components & services**

- Describe the role in building cloud applications of:
  - **Notification services**, *e.g.*, Amazon's Simple Notification Service
  - **Message queues**, *e.g.*, Amazon's Simple Queueing Service

- Illustrate how Amazon Step Functions provide support for **distributed transactions** in cloud applications

Cloud plumbing

- Ideally software components are **specialised**
    - Facilitates effective **separation of concerns**
    - Allows for broadest possible **reuse potential**
    - Scalability and elasticity can focus on **specific functionality**

- But applications need **interconnected components**
    - Want to avoid hard-coding component interactions
    - Interconnections are good **monitoring**, **logging**, **& audit** points
    - Often can use discrete messages rather than data streams

Component interaction queuing examples

- Workloads we've seen: **synchronous, 1–1 interactions**
  - Web + DB—web request initiates DB query; render DB response
  - S3 + Lambda—react when a particular bucket is changed

- Two different, useful **types of decoupling** are:
- **1** May have 1–n, *e.g.*, **allocate jobs to a pool of workers**
- **2** May have **disconnected targets**, *e.g.*, onsite database
  - *e.g.*, ensure that retry can occur, but is not sender's problem
  - Even within cloud services, batching can boost performance

Message queues and notification services

- **Notification services** and **message queues** factor out interconnection needs between software components

- Notification services—*e.g.*, **publish/subscribe paradigm**
  - Publishers **publish** messages on particular topics
  - Subscribers **subscribe** to those topics

- Message queues typically focused on **reliable delivery**
  - Temporary storage of messages is the focus

Message queues: key features

- Common case functionality of message queues is easy
  - Just a **buffer** between producer(s) and consumer(s)
  - … but buffer needs **persistency**; **high throughput**; **low latency**
    - (These requirements usually trade off against each other.)

- Message queues often provide further functionality
  - **Asynchronous delivery**—receiver need not be online
    - When sending, producer needs not consider receiver's status
  - **Reliable delivery**—retry after failures
  - **Dead letter queue** (DLQ)—messages go here after max. retries

Amazon Simple Queueing Service (SQS)

- SQS—decouples two applications
  - Producer **pushes** messages into a queue
  - Consumer **pulls** messages from the queue
  - Push/pull is analogous to pipes in Unix / WinNT OS kernels
  - Messages are **stored** for up to 14 days

- Two queue types: (for 64KB ‘chunks’)
  - **Standard**—may: deliver duplicates; out of order ($0.40/mil)
  - **FIFO**—no duplicates; first-in-first-out order; slower ($0.50/mil)

Amazon Simple Notification Service (SNS)

- SNS is a **topic-based publish-subscribe** system
  - Has multiple subscriber types:
    - **High-speed**: SQS; HTTP(S) POST to web-hooks; AWS Lambda
    - **Mobile**: email; SMS; iOS+Android push notifications
  - Fan out to multiple interested subscribers
- Subscribers can **set filters** on notifications
  - e.*g*., prefix matching on names; range matching on attributes
  - (SNS can then pass filtering close to source, which is optimal)
  - Topics can be set to deliver raw data: payload without JSON

Amazon Step Functions

- What if you want to use a **workflow with time delays**?
  - e.g., notify owner one month after they upload an S3 object
  - SQS / SNS can't help directly: don't support app-level timing
  - Can't usefully use Lambda, as function **would run for a month!**

- Amazon Step Functions handle this type of use case:
  - **Track states of execution**—you're only charged for transitions
  - Steps in parallel; serial; conditional; relative/absolute delay,…
  - Easy Lambda support; but doesn't work in AWS Academy

Distributed transactions

- Typical monolithic application design is **DB-backed**
  - Databases usefully support transactions
  - However databases are also a common bottleneck in designs
- **Saga pattern** is common micro-services alternative
  - (Actually published in 1987 by Garcia-Molina and Salem)
  - Saga is sequence of tasks that are in “transaction”
  - Each task must have a compensating action—an “undo”
    - These must be **idempotent**: saga rewind might need rerunning
  - Amazon Step Functions can orchestrate saga implementation