SYSTEM DESIGN – KWIPPER

Created by: Jidni Ilman

BACKGROUND

Kwipper is a fictional company that provides a learning management system for everyone. The user of this system primarily is learners and tutors. A tutor can make courses including materials such as Text, Video, and Multimedia files that learners can download. On the other hand, learners can subscribe to a course (with recurring billing) and can learn from Text, Video, and download Multimedia files from the course.

Kwipper serves 100M users across the world mainly in 4 countries: Indonesia (30M), Japan (30M), Philippines (30M), and Mexico(10M). Kwipper business development and strategic planning division have a roadmap to make Kwipper System more Highly Scalable, Highly Available, High Consistency, and Faster access for its users.

In case of an unwanted situation, the sustainability team requested the engineering team to make sure that Kwipper System has Disaster Recovery in case of sudden changes of governance regulation, data loss, sanctions, or trade embargo. The form of Disaster Recovery is not too complex and just provides important functionality: Backup and Replication.

REQUIREMENTS

FUNCTIONALITY

- Users have Roles: Tutor, Learner
- Users can Subscribe to the Courses
- Users can access the Courses materials
- Users can do Recurring Billing

NON-FUNCTIONALITY

- Highly Scalable
- Highly Available
- High Consistency
- Faster Access for user
- Disaster Recovery: Backup and Replication

ASSUMPTIONS

USERS

- 100M Users divided into 4 Service areas
- 100% Monthly Active Users (MAU) – 100M
- 20% Daily Active Users (DAU) – 20M
- We assume for User data in a row is 500 bytes (same as 500 characters), so it's ~50GB of data

COURSES

- Kwipper has 1K Course
- Each Course has 10 Course Item
- We assume for Course Item data in a row is 5000 characters (~5KB) (excluding Video and Multimedia Files)
- We assume for Course data in a table is 10*1000*5KB, so its ~50MB of data

CONTENTS

- Each Course Item has 100MB of data stored materials (90MB for 1 Video and 10MB for 1 other multimedia file)
- Each Video has 1800 seconds length
- We approximately need 10*1000*100MB = ~1TB storage for materials
- We assume for Content data in a row is 200 characters, so it's 200bytes*10*1000 = ~2MB of data

NETWORK

- 1 Course Item consumed per Users Daily
- 1 Course Item has an average of 2MB/s Network Throughput (Rate Limited) for its course materials
- 2*10*1000 = ~20 TB Network Throughput are needed daily by our system

SUBSCRIBE

- We assume every user subscribes to a new course each month
- We assume for Subscribe data in a row is 100 bytes, so it's 0.1KB * 100M *12 = 120GB of data every year

BILLING

- We assume every user has recurring billing every year
- We assume for Billing data in a row is 500 bytes, so it's 0.5KB*100M = ~50GB of data every year

OTHERS

- We assume the tutor only reviewed the running courses, not created more courses
- We assume the system didn't receive any newer users
- We only do the system design and skip the unnecessary business process
- We assume we are using GCP as our cloud platform, but we are using Terraform for the possibility to do a multi-cloud deployment to other cloud platforms such as AWS and Azure

SERVICES

- User – Manage Users including Roles
- Course – Manage Courses including Course Items
- Content – Manage Courses Video and Multimedia Contents
- Subscribe – Manage User-Course Subscription
- Billing – Manage User Billing and Payment including payment status

SOLUTIONS AT A GLANCE

- Microservices Architecture
- Isolated Database for each Services
- Database Sharding and Replication with Consistent Hashing (Hashing Ring)
- Modern Technology Stack with its specific Business Use Case

APPLIED PATTERNS

APPLICATION ARCHITECTURE PATTERN – Microservices Architecture

MESSAGING STYLE PATTERN – Publish-Subscribe (Pub/Sub) Messaging

SERVICE DISCOVERY PATTERN – 3rd Party Registration

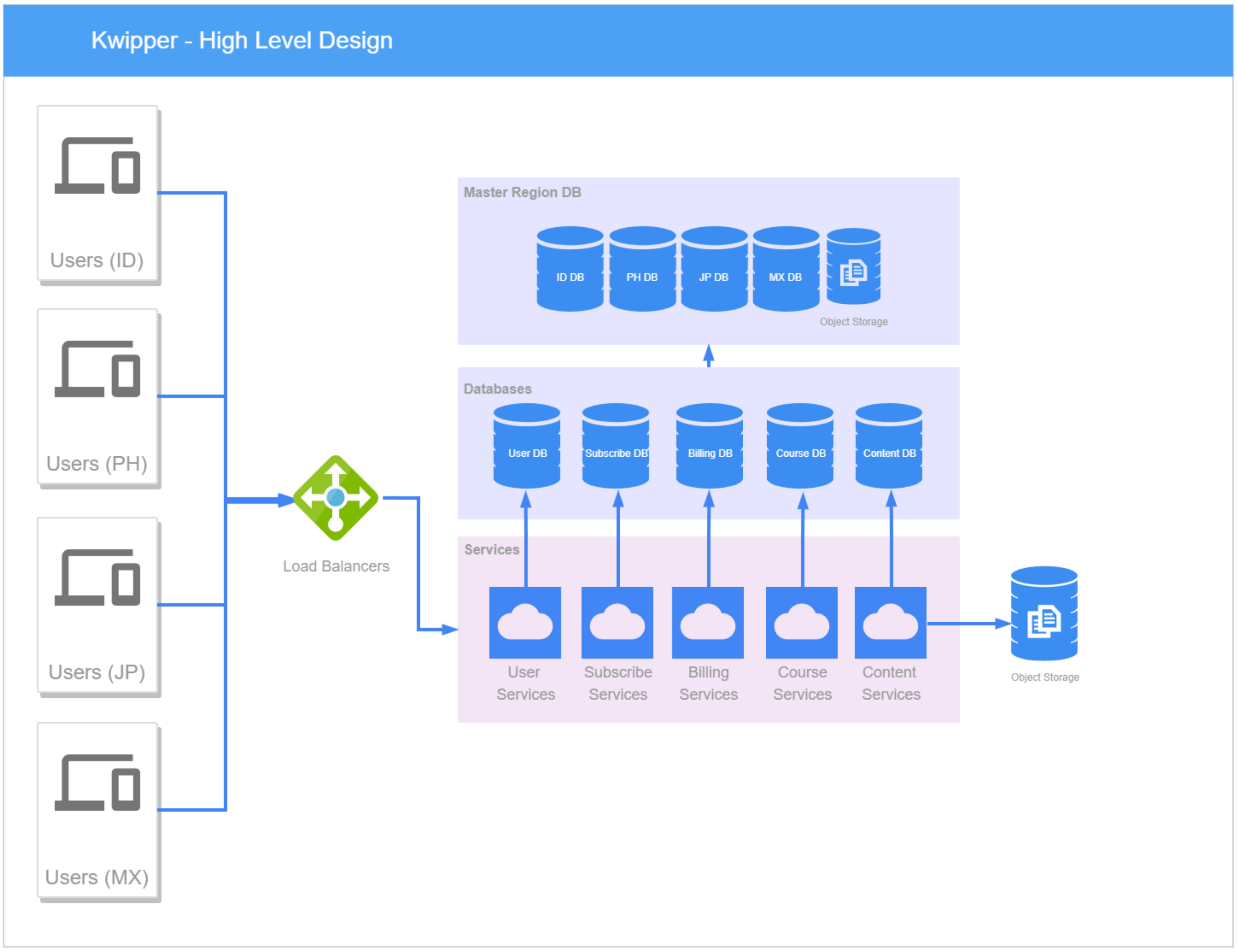
DATA CONSISTENCY PATTERN – Saga

EXTERNAL API PATTERN – API Gateway

OBSERVABILITY PATTERN – Distributed Tracing, Logging, and Health Check API

DEPLOYMENT PATTERN – Service as a Container

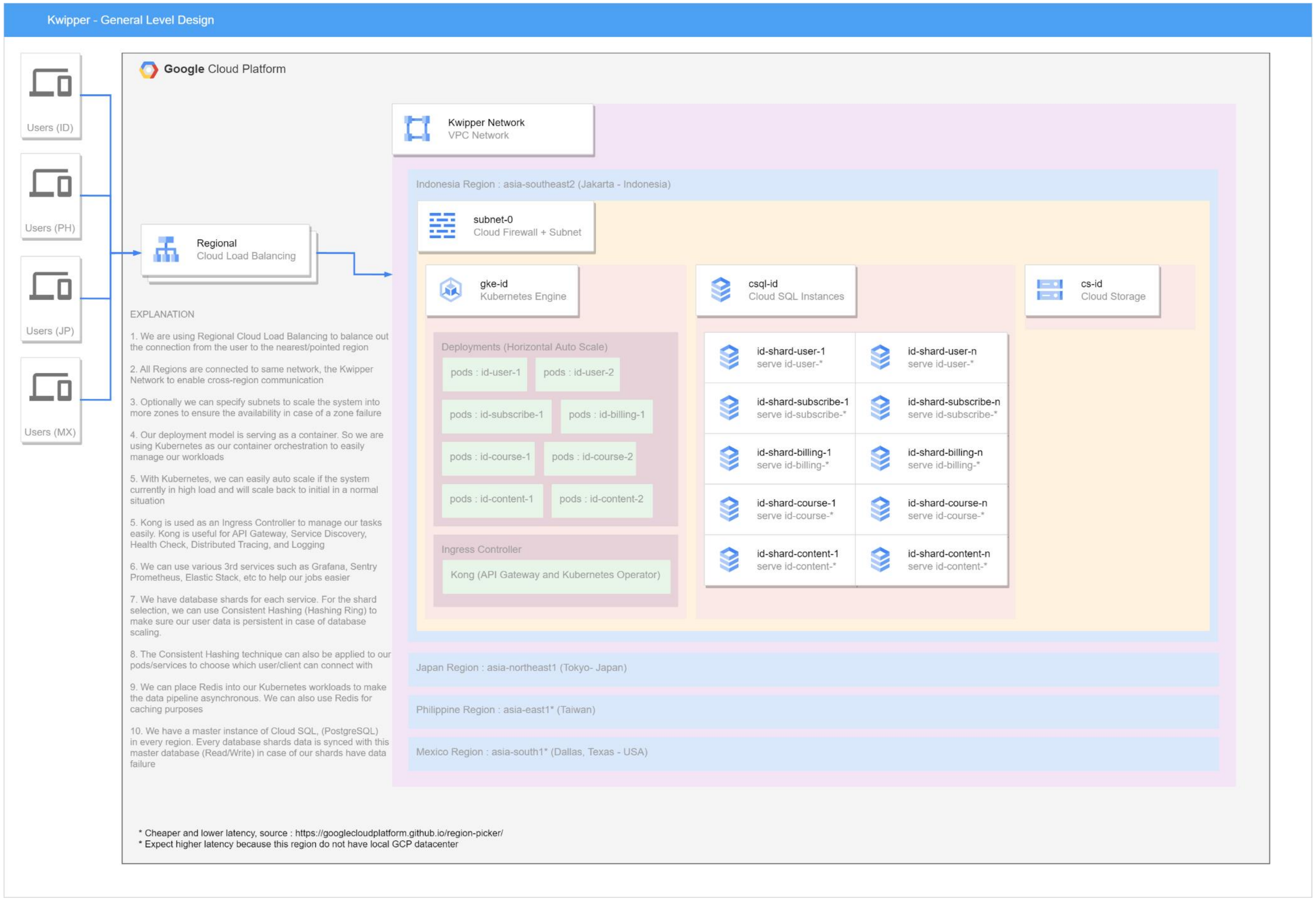
HIGH-LEVEL DESIGN



The basic high-level design of our kwipper system. We need load balancers, a service pool, databases, object storage, and a master database for each region. From this design now we can define and scale each part of system design requirements into our final and implementation design.

FINAL DESIGN

GENERAL LEVEL



EXPLANATION

1. We are using Regional Cloud Load Balancing to balance out the connection from the user to the nearest/pointed region

2. All Regions are connected to same network, the Kwipper Network to enable cross-region communication

3. Optionally we can specify subnets to scale the system into more zones to ensure the availability in case of a zone failure

4. Our deployment model is serving as a container. So we are using Kubernetes as our container orchestration to easily manage our workloads

5. With Kubernetes, we can easily auto scale if the system currently in high load and will scale back to initial in a normal situation

5. Kong is used as an Ingress Controller to manage our tasks easily. Kong is useful for API Gateway, Service Discovery, Health Check, Distributed Tracing, and Logging

6. We can use various 3rd services such as Grafana, Sentry Prometheus, Elastic Stack, etc to help our jobs easier

7. We have database shards for each service. For the shard selection, we can use Consistent Hashing (Hashing Ring) to make sure our user data is persistent in case of database scaling.

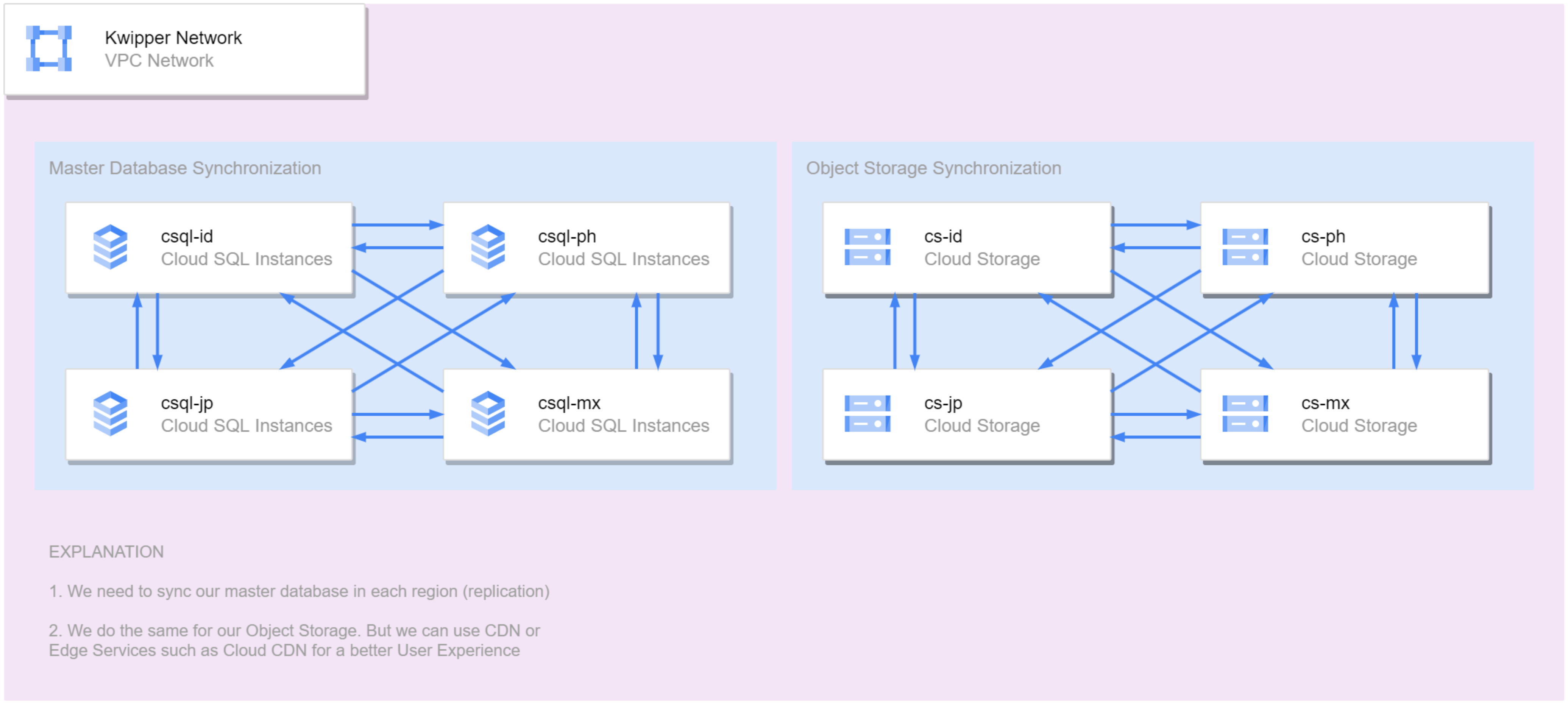
8. The Consistent Hashing technique can also be applied to our pods/services to choose which user/client can connect with

9. We can place Redis into our Kubernetes workloads to make the data pipeline asynchronous. We can also use Redis for caching purposes

10. We have a master instance of Cloud SQL, (PostgreSQL) in every region. Every database shards data is synced with this master database (Read/Write) in case of our shards have data failure

* Cheaper and lower latency, source : https://googlecloudplatform.github.io/region-picker/
* Expect higher latency because this region do not have local GCP datacenter

DATABASE SYNCHRONIZATION DESIGN



EXPLANATION

1. We need to sync our master database in each region (replication)

2. We do the same for our Object Storage. But we can use CDN or Edge Services such as Cloud CDN for a better User Experience

With this infrastructure scheme, we can make sure that if something unwanted happened in one region, another region can help as a temporary backup until a new instance is provisioned, a new instance is created, fetch data from a stable database region, and ready to serve services again.

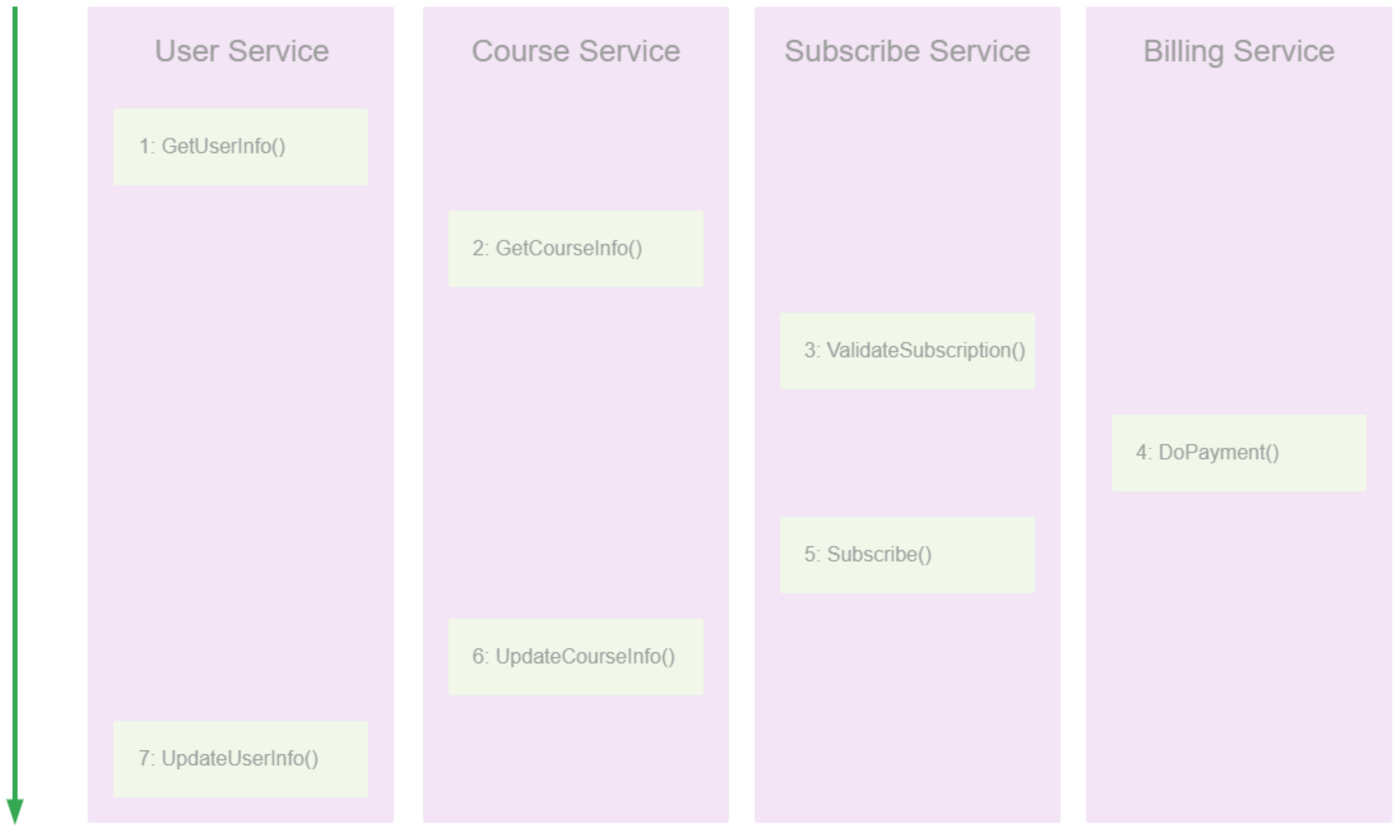
The synchronization process didn’t require to be real-time, with proper scheduling, we can cut more computing power and lower the cost.

In the next page, we can see the Saga Data Consistency Pattern in action.

SAGA DATA CONSISTENCY PATTERN

TRANSACTION

The transaction process as an example is **A user subscribing to a course and do the payment.** For this situation, the services that do the work are **User**, **Course**, **Subscribe**, and **Billing.** The task of each service is as follows:



COMPENSATION TRANSACTION

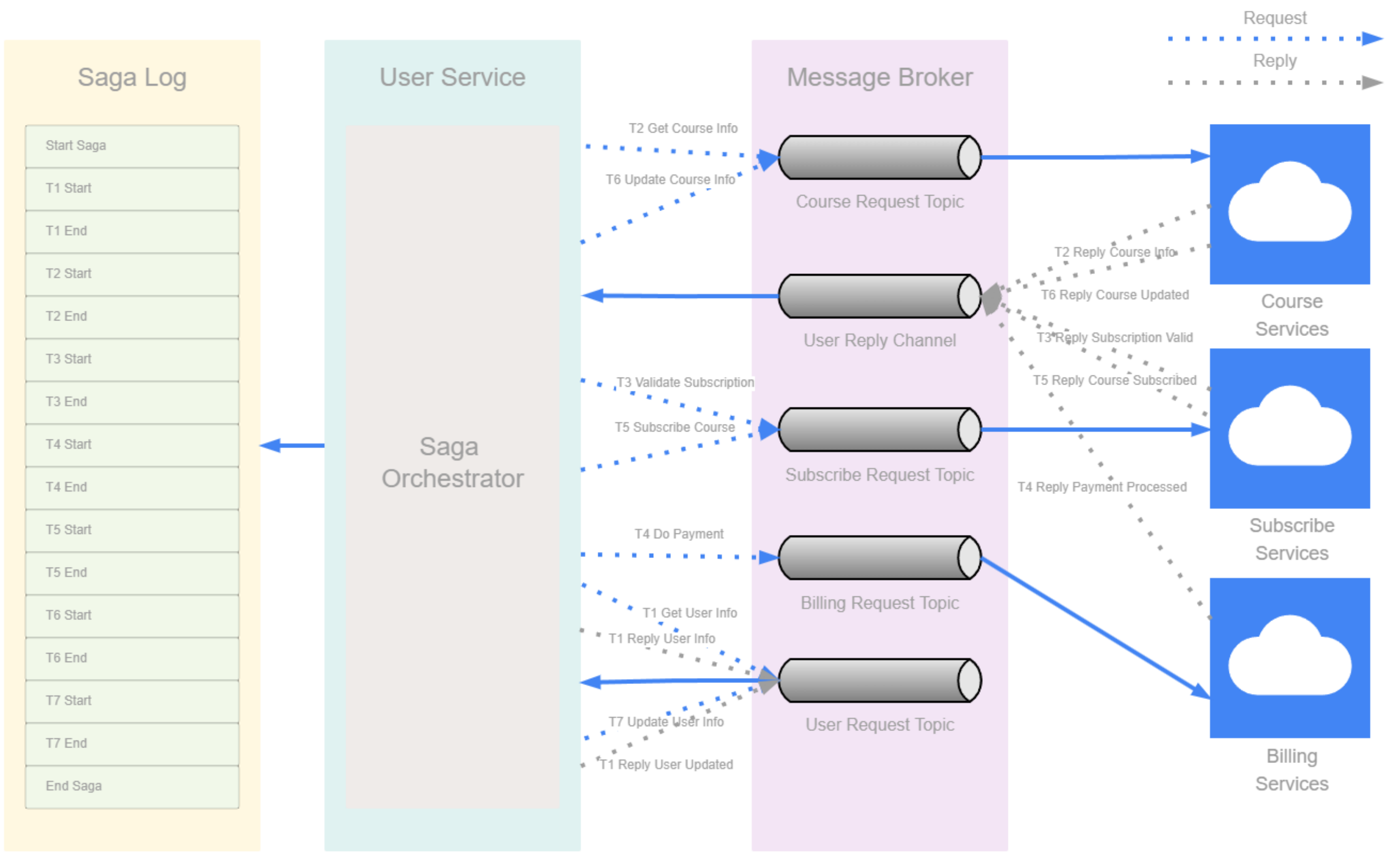
What happens if a local transaction resulting failure? For example, Credit Card Rejected? Or User issuing refund. This is the table for compensation transactions for each service.

Step	Service	Transaction	Compensating Transaction
1	User Service	GetUserInfo()	-
2	Course Service	GetCourseInfo()	-
3	Subscribe Service	ValidateSubscription()	-
4	Billing Service	DoPayment()	RefundPayment()
5	Subscribe Service	Subscribe()	Unsubscribe()
6	Course Service	UpdateCourseInfo()	RevertCourseInfo()
7	User Service	UpdateUserInfo()	RevertCourseInfo()

Compensation Transaction will make sure that the changes to the data are reverted in case of transaction failure. This gives us data consistency of our services. This is strengthened by the atomicity of our system, using a Pub/Sub messaging pattern as a way to communicate between services. The communication needs to be idempotent and retriable, in case of not responding service, we can request another new service from Service Registry for completing the transaction.

ORCHESTRATION SAGA

For this case, we are using the orchestration type of saga (centralized control). This is the saga of our transaction process.



RECOMMENDED TECHNOLOGY STACK

SERVICES LANGUAGE

Golang is the fast, simple, maintainable, and growing programming language designed best for concurrency. Many big names in tech use Golang, such as Kubernetes, Terraform, and Docker. The drawback of using Golang is the steep learning curve for the newer player entering the IT Industry, especially the concept of concurrency and pointer. As an ex-C Programmer, I found myself comfortable when entering Golang.

SERVICES FRAMEWORK

Labstack Echo, is a high-performance and minimalist framework with complete ready-to-use features. For more freedom and working specifically on Microservice Architecture, I'd recommend Go Kit and Go Micro. But personally, I found myself comfortable using Echo.

MESSAGE BROKER

Apache Kafka is my choice of a message queue and broker service. Kafka uses disk, despite others using RAM to store the data. One of the beauties of Apache Kafka is sequential I/O end-to-end message delivery and the zero-copy principle. I haven't used Kafka yet, but it is really interesting to learn. I have only used Redis and NATS in the past.

CONTAINER OS

Alpine Linux (Runtime Image) has a slogan of small, simple, and secure. And it's true! I've personally used Alpine for my docker container runtime OS. For builder OS, I've used the suitable OS for each tech I used, such as Golang Bullseye (Debian) for Golang.

IN MEMORY DATABASE

Redis is fast and powerful for caching. It's generally best practice to use it as a cache for database queries and storing frequently used user queries.

DATABASE

People use **PostgreSQL** for its Availability and Consistency. Strongly adapt the atomic, consistent, isolated, and durable (ACID) properties. PostgreSQL has a strong reputation for being reliable and for best performance. A lot of Cloud Native applications use and recommend PostgreSQL as the database of choice. I have experience using MongoDB, Firebase Firestore, Firebase Realtime Database, and MySQL. I can't find myself comfortable as I use PostgreSQL.

CokroachDB on other hand is a new contender in the relational database groups and relatively new. But I am kinda interested in its slogan: Scale Fast, Survive Disaster, Thrive Everywhere (as a true cockroach in mind).

CONTAINER

Docker is a king in container technology to make development efficient and predictable. I'm not going into docker deeply, just using docker as needed, craft Dockerfile or docker-compose, build, deploy, deploy into Kubernetes workload in GKE and that's it. But as of today, I'm still using docker as my primary container technology.

Kaniko is container builder using Dockerfile. Made by Google. I have zero experience in Kaniko but am willing to learn it when I need it.

ORCHESTRATION

Kubernetes or **K8s** is my favorite tool for orchestrating containerized apps. The useful features of K8s are Automatic Deployment, Auto Scaling, Workloads Management, Secret Management, Service Load Balancing, and Ingress.

INFRASTRUCTURE AS CODE (IAC)

Terraform can be used for automatic infrastructure deployment on any cloud. It is very useful when you often dynamically scale/deploy/modify your cloud infrastructure or use a multi-cloud deployment model for your system design.

CLOUD PLATFORM

Google Cloud Platform (GCP) is a little expensive compared to other competitors but it doesn't mean it losing its competitive power in the market. The solution suites provided by GCP really make your life easier. I have used Azure for a long time more than 4 years, AWS, Alibaba Cloud, and DigitalOcean. I'm still going back to GCP for my choice of the cloud platform. I've used GCP since 2019 and never had problems with it.

STORAGE SERVER

MinIO is S3 compatible object storage server. In my opinion, MinIO is the king of Object Storage solutions (even compared with Google Cloud Storage). Sometimes you need to isolate the Object Storage Technology from being too dependent on cloud platform-provided solutions such as Cloud Storage. I have experience with managing Object Storage locally (must be located in Indonesian Datacenter) due to government regulation and MinIO make this easier. At that time, GCP didn't have a data center in Indonesia. And waiting for that one built is not a good solution for my client. Migrating MinIO from GCP to Alibaba Cloud is easier thanks to their slogan being the Multi-Cloud Object Storage.

API GATEWAY

Kong is API Gateway and Service Conectivity Platform. You have your life easier managing microservice using Kong. Kong also supports the Service Mesh (1000+ Microservices) for your system. With Kong, you have Circuit Breakers, Health Checks, and Service Discovery for your services. You also have Distributed Tracing and Logging for your monitoring.

WEB SERVER (IF USING ANY)

Caddy is another powerful app written in Golang. Caddy is a powerful web server with automatic HTTPS. Caddy also has API Gateway, Reverse Proxy, Logging, Caching, Ingress, and WAF. In my opinion, it's a complete challenger to NGINX. I also found it easier to use caddy than NGINX when I tried it. I've using Caddy for one of the government projects that I manage. And when I train the government people about using Caddy, they can learn it easily.

CLOSING NOTES

- In the database design for each region, I'm using Database Sharding Technique. But to understand how each user/service connects to the correct shard is to implement consistent hashing.
- Consistent hashing is different from simple hashing that uses modulo (or another algorithm) to determine which shard the user/service should connect.
- Consistent Hashing gives us freedom of choice when to make a new instance of server/database whenever we want.
- You can check this for how to implement consistent hashing.
- In practical means, you can use a circle angle (360 degrees) for the hash. You can easily check what server/database is nearest with a service. With a circle, you can have theoretically unlimited hash nodes.
- I hope this system design document can give you a clear understanding of the system that I designed for Kwipper.
- I still needed to learn a lot, but I hope this system design document can meet the Quipper #2 Take Home Test criteria.