COMSATS University
Islamabad
Abbottabad Campus

Name: Taifoor Farid Siddiqui

Reg #: FA22-BSE-051

Course: Software Design and Architecture.

Example Number 1:

Project Name: Twitter

Architecture Problem: Scalability issues with the monolithic architecture.

Issue:

In 2008, Twitter experienced frequent crashes, symbolized by the "Fail Whale" error message. This was because the system's monolithic design couldn't handle the rapidly increasing number of users and their activity. A monolithic architecture means all parts of the application (like handling tweets, showing timelines, searching) are tightly coupled and run as a single unit. When one part gets overloaded, the entire system is affected. It's like having one big server trying to do everything at once.

Solution:

Twitter moved to a microservices architecture. This means they broke down the large, single application into smaller, independent services. Each service is responsible for a specific function. The key components of this solution include:

- **Service Separation:** Functions like delivering tweets, generating user timelines (the stream of tweets you see), and performing searches were separated into individual services. This means each of these functions runs independently. If one service experiences high load, it doesn't necessarily impact the others.
- **Caching:** They implemented caching. Caching stores frequently accessed data in a fast access location (like memory). When a user requests data that is cached, the system retrieves it from the cache instead of going to the slower main database. This greatly reduces database load and speeds up response times.
- **Asynchronous Messaging:** They used asynchronous messaging. In a synchronous system, a request waits for a response before continuing. In an asynchronous system, a request is sent, and the sender doesn't wait for an immediate response. The response is handled later. This allows the system to handle many requests concurrently without blocking. This is important for handling large volumes of tweets and other actions.

In simpler terms: Instead of one big program doing everything, they created many smaller, independent programs that each do one specific thing. This allowed them to scale each part of the system independently and handle more users and activity. Caching and asynchronous messaging further optimized the performance and responsiveness of the platform.

Example Number 2:

Project Name: Netflix

Architecture Problem: Dependency issues in a monolithic system.

Issue:

Initially, Netflix used a monolithic design for its DVD rental service. As they transitioned to streaming, the system struggled with reliability and often failed under high demand. Because all parts of the application were tightly coupled in the monolithic architecture, a single failure in one part of the system affected the entire service. This meant that if, for example, the recommendation engine had a problem, it could bring down the entire streaming experience.

Solution:

Netflix shifted to a microservices architecture. This involved breaking down the large, single application into smaller, independent services. Each service is responsible for a specific function, allowing them to operate and scale independently. The key components of this solution include:

- **Service Separation:** Functions such as user authentication, video recommendations, video playback, billing, and metadata management were separated into individual services. This means if the recommendation service has an issue, it doesn't necessarily impact the user's ability to watch a video.
- **Chaos Engineering:** Netflix pioneered the use of Chaos Engineering. This practice involves deliberately injecting failures into the system in a controlled environment to identify weaknesses and vulnerabilities. By proactively causing failures, they could identify and fix potential problems before they impacted real users. This approach helped them build a more resilient and fault-tolerant system.

In simpler terms: Netflix moved from one large, interconnected program to many smaller, independent programs. Each of these smaller programs handles a specific task, like suggesting movies or playing videos. This change meant that if one part had a problem, the other parts could continue to work. They also started intentionally breaking parts of their system in a safe way to find and fix potential weaknesses before they caused real problems for users.

Example Number 3:

Project Name: Amazon

Architecture Problem: Bottlenecks in a centralized database.

Issue:

Early on, Amazon relied on a monolithic architecture with a single, centralized database to handle orders, inventory, customer data, and other critical information. This setup created bottlenecks, especially during peak shopping periods like holidays. The single database couldn't handle the massive influx of requests, leading to slowdowns and impacting user experience. This meant slow loading times, difficulty placing orders, and other performance issues.

Solution:

Amazon migrated to a service-oriented architecture (SOA). This involved splitting the monolithic system into smaller, independent services. Each service was responsible for a specific business function. The key components of this solution include:

- **Service Decomposition:** Functions like order processing, inventory management, customer accounts, and product catalogs were separated into individual services. This allowed each service to operate and scale independently.
- **Decentralized Databases:** Each service was given its own database. This decentralized approach eliminated the bottleneck of the single, centralized database. Each database could be optimized for the specific needs of its corresponding service.
- **DynamoDB Introduction:** Amazon developed and introduced DynamoDB, a NoSQL database, to manage high-speed, scalable data operations. DynamoDB is designed to handle massive amounts of data and high traffic volumes, making it ideal for services that require extreme scalability and low latency (delay).

In simpler terms: Amazon initially used one big database to store all its information. This became a bottleneck when many people tried to use the website at the same time. They then switched to using many smaller databases, each dedicated to a specific part of the business, like managing orders or tracking inventory. This allowed each part of the system to handle its own workload without being slowed down by other parts. They also created a special type of database, DynamoDB, designed to handle very large amounts of data and traffic quickly.

Example Number 4:

Project Name: Uber

Architecture Problem: Challenges with real-time data handling in microservices.

Issue:

As Uber transitioned from a monolithic architecture to a microservices architecture, they encountered challenges related to inter-service communication, especially concerning real-time data. Features like location tracking, which require constant and immediate updates, suffered from latency (delays) and inconsistencies. This meant that the information displayed to users (driver locations, estimated arrival times) wasn't always accurate or up-to-date, leading to a poor user experience. The problem stemmed from the difficulty of efficiently sharing and updating real-time data across many independent services.

Solution:

Uber implemented solutions to address these real-time data handling challenges. The key components of their solution include:

- **Apache Kafka for Event Streaming:** They implemented Apache Kafka, a distributed streaming platform. Kafka allows for high-throughput, fault-tolerant, real-time data streaming. This means that events (like a driver's location update) are published to Kafka as a stream, and other services can subscribe to this stream to receive the updates in real time. This enabled efficient and reliable communication of real-time data between services.
- **Ringpop for Service Discovery and Load Balancing:** They introduced Ringpop, a library that provides consistent hashing and membership for distributed applications. Ringpop helps services discover each other and efficiently distribute load across instances of each service. This optimized communication pathways and prevented any single service instance from becoming overloaded, further improving performance and reducing latency.

In simpler terms: When Uber switched to using many smaller programs instead of one big program, they had trouble keeping all the programs updated with real-time information, like driver locations. To solve this, they used a system called Apache Kafka, which acts like a fast and reliable messaging system for real-time updates. They also used a tool called Ringpop to help the smaller programs find each other and share the workload efficiently, ensuring everything runs smoothly and quickly.

Example Number 5:

Project Name: Facebook

Architecture Problem: Database scaling issues.

Issue:

As Facebook's user base grew rapidly, they faced significant challenges with database scaling. Their initial database system struggled to handle the increasing number of read and write operations required to manage user data, connections, and interactions. Issues with database replication (copying data to multiple servers) and data consistency (ensuring all copies of the data are the same) became major bottlenecks, impacting performance and user experience. The system simply couldn't keep up with the sheer volume of data and activity.

Solution:

Facebook implemented several key solutions to address their database scaling challenges. These included:

- **TAO (The Association and Objects):** They developed TAO, a custom data store specifically designed to efficiently manage social graph data. A social graph represents the relationships between users (friends, connections, etc.). TAO is optimized for read-heavy workloads, which are typical in social networks where users frequently view profiles and connections.
- **Sharding:** They implemented sharding, a technique that involves partitioning the database into smaller, more manageable pieces called shards. Each shard holds a subset of the data, and these shards are distributed across multiple servers. This distributes the database load and improves performance by allowing concurrent access to different parts of the data.
- **Caching:** They heavily utilized caching techniques. Caching stores frequently accessed data in a fast access location (like memory). When data is requested, the system checks the cache first. If the data is found in the cache, it's retrieved much faster than from the database. This significantly reduces database load and improves response times.
- **Distributed Architecture:** They moved to a distributed architecture, where the application and its data are spread across multiple servers. This allowed them to scale horizontally by adding more servers as needed, increasing the system's capacity and resilience.

In simpler terms: As Facebook got more and more users, their original database couldn't handle all the information and activity. To fix this, they built a special database system called TAO that is really good at handling social connections. They also split their data into smaller chunks and spread them across many computers (sharding). They used caching to quickly access frequently used information, and they designed their whole system to run on many different computers working together (distributed architecture). These changes allowed Facebook to handle its massive amount of data and users efficiently.