**MICROSERVICE & DISTRIBUTED TRANSACTION**

1. **Terms**

X-axis scaling: horizontal duplication, scaling up by cloning 🡪 common in monolithic. Run multiple instances of the application on a load balancer

Z-axis scaling: data partitioning, scale by splitting. Same as X, but each instance runs on a particular data set

* Improve capacity and availability but not the problem of increasing development and application complexity.

Y-axis scaling: functional decomposition, scale by splitting things (functions) that are different

1. **Monolithic**

Toàn bộ codebase, business logic, database layer đều nằm ở trong một hệ thống

Advantage:

* Development flow đơn giản, logically straightforward
* Dễ test dễ deploy

Disadvantage:

* Scalability: Khó để scale hệ thống lên một cách nhanh chóng nếu đột ngột cần đáp ứng một lượng người dung tăng cao.
* Theo thời gian thì khối project sẽ ngày càng to 🡺 ide start up lâu hơn, khó quản lý hiệu quả
* Các module liên quan đến nhau nên khi muốn update hệ thống hoặc áp dụng các công nghệ mới có thể sẽ phức tạp
* Fault tolerance vì nếu có lỗi thì có thể ảnh hưởng cả hệ thống 🡪 unreliable

1. **Intro to microservice**

Microservice architecture là loại structure mà hệ thống được chia ra làm các service nhỏ độc lập với nhau 🡪 Loose coupling 🡪 distributed system.

Service là một mini application mà chỉ tập trung vào 1 tính năng duy nhất, mỗi service sẽ có một database riêng cho nó (polygot persistence architecture) và liên lạc giữa các services được thực hiện thông qua api

Properties:

* Single responsibility: mỗi service chỉ chịu trách nhiệm cho duy nhất một tính năng
* Built around business capabilities: có thể sử dụng nhiều kiểu công nghệ để phục vụ tính năng mình muốn có, hướng tới
* Design for failure: cẩn trọng với các failure cases
* Thay vì tất cả service đều chia sẻ một database chung như monolithic thì trong microservice architecture, mỗi microservice sẽ có một database riêng để sử dụng. Tuy sẽ có sự trùng lặp record nhưng đây là tradeoff đáng để có được các tính năng khác của kiến trúc này. 🡪

Advantage:

* Continuous delivery and deployment of large, complex applicaiton
* Mỗi microservice nhỏ nên dễ quản lý, dễ develop, được develop và scale lên độc lập với nhau (theo x-axis hoặc z-axis), dễ hiểu nên có thể được pick up dễ hơn nếu có thành viên hoặc nhóm mới tham gia vào. 🡪 hiệu quả, không quá nặng nên ide không bị slow down đáng kể như monolithic
* (Vì các service quan hệ rời rạc lỏng lẻo với nhau nên là) nếu cần update hay modify một module thì sẽ không làm ảnh hưởng đến các services còn lại
* Mỗi microservice có thể sử dụng một framework hoặc công nghệ khác nhau. Có thể thay đổi công nghệ sử dụng liên tục mà không làm ảnh hưởng đến các microservice khác.

Disadvantage:

* Cần tìm đươc thuật toán phù hợp với yêu cầu để có thể decompose được hệ thống thành các services. Nếu làm sai thì mình sẽ tạo distributed monolith mà các services vẫn cần phải được deployed cùng nhau.
* Development process phức tạp hơn so với monolithic
* Deploy phức tạp, cần có kế hoạch hợp lý
* (Tính bảo mật không quá cao)
* Debug khó hơn vì control flow sẽ liên quan đến nhiều microservices, khó để chỉ ra được là lỗi ở chỗ nào
* Khó implement các thay đổi mà liên quan đến nhiều microservices khác nhau.

1. **Monolithic architecture vs. Microservice**

|  |  |  |
| --- | --- | --- |
| Tính năng | Monolithic | Microservice |
| Development process | Đơn giản | Phức tạp hơn |
| Deployment process | Once | Mỗi microservice sẽ được deployed độc lập với nhau, nếu có vấn đề gì xảy ra ở một microservice thì những cái khác không bị ảnh hưởng |
| Test | Dễ | Khó |
| Bảo trì | Cần một đội chuyên gia để bảo trì. Tìm lỗi mất thời gian và tốn kém | Nhanh hơn monolithic, qua thời gian thì sẽ hiệu quả và đỡ tốn kém hơn |
| Reliability | Nếu có bug thi ảnh hưởng đến hệ thống | Nếu có lỗi ở một hay một vài microservice thì sẽ không ảnh hưởng đến phần còn lại |
| Phù hợp với hệ thống | Nhỏ, vừa phải | Medium -> large |
| Scalability | Khó | Tốt |
| Ứng dụng công nghệ mới (Flexibility) | Khó, difficult to migrate to a new technology or adopt new framework. Old technology that the system has adopted will become obsolete sooner or later | Dễ |
| Database | 1 database to | Nhiều databases, mỗi service một database, Data consistency is a problem |
| Fault | Fault tolerance | Fault isolation |
| Cost | Bình thường | Cao |

1. **Distributed transaction**

Transaction (giao dịch) là quá trình xử lý từ lúc bắt đầu đến khi kết thúc sao cho thỏa mãn 4 nguyên tắc ACID.

Distributed transaction là một giao dịch mà trải dài qua nhiều database   
2 possible outcomes:

* All operations successfully complete
* None is performed due to failure at some point in the system. Trong trường hợp này, nếu có operation nào đã được hoàn thành trước khi có failure thì nó sẽ được reverse để đảm bảo rằng không có sự thay đổi nào xảy ra
* Đảm bảo nguyên tắc ACID

Distributed transaction pattern

* 2 phase commit
* Saga
* TCC (try commit cancel)

1. **Two phase commit**

Có 2 phase trong pattern này:

* Prepare phase: các service được thông báo là phải chuẩn bị cho các thay đổi sắp tới trong database (và đảm bảo atomicity). Các microservice thực hiện commit hoặc rollback trong phase này. Khi một service đã chuẩn bị xong thì nó sẽ issue ra một prepared message. Tin nhắn này nghĩa là service hoặc sẽ commit hoặc rollback và đảm bảo là sẽ có lock để mà nếu transaction thất bại thì sẽ không bị vấn đề gì.
* Commit phase: sau khi các microservice đã chuẩn bị xong thì trong phase này tất cả thực hiện các thay đổi đã thông báo. Các microservice thực hiện commit trong phase này
* Ngoài ra còn có forget phase: global coordinator sẽ quên cái giao dịch vừa thực hiện.

Advantage:

* Đảm bảo được tính acid

Disadvantage:

* Synchronous (blocking). Protocol sẽ lock object đang được thay đổi cho đến khi transaction hoàn thành. 🡪 mất thời gian, có thể sẽ tạo ra deadlock
* Không có rollback cho commit phase

1. **Saga**

A mechanism to maintain data consistency without using distributed transaction. Well known, được sử dụng nhiều.

Asynchronous and reactive.

Chia transaction ra thành các local transaction, mỗi cái sẽ có transaction module và compensation module riêng, nếu một local transaction mà fail thì các compensation module kia sẽ được sử dụng để đảm bảo tính nhất quán cho transaction

Một saga là một dãy các local transaction. Khi một local transaction hoàn thành thì message sẽ được gửi đi, message này sẽ trigger bước tiếp theo của saga (bắt đầu một local transaction mới)

2 hình thức compensating transaction:

* Forward recovery: retry lại local transaction mà đang có lỗi
* Backward recovery: rollback lại các local transaction phía trước mà đã thành công (dùng cái này là chính)

Advantage của saga:

* Đảm bảo được data consistency cho transaction dài (long running)

Disadvantage:

* Phức tạp
* Không có read isolation (thiếu isolation – chỉ có tính acd), có thể tạo ra các anomalies: dirty reads, lost updates

2 cách để coordinate saga:

* Choreography: distribute the decision making and sequencing among the saga participants. They communicate using messages

+ Benefit:

* Đơn giản: services sẽ publish event mỗi khi có sự kiện nào xảy ra
* Loose coupling: các thành viên đăng ký nhận event mà không biết thông tin của nhau

+ Drawbacks:

* Khó hiểu
* Phụ thuôc vòng giữa các dependencies: không hẳn là vấn đề
* Orchestration: centralized. Dùng một orchestrator để điều phối các thành viên phải làm gì

+ Benefit:

* Simpler dependencies: không có phụ thuộc vòng giống như choreography
* Less coupling: mỗi thành viên thì có api invoked bởi orchestrator nên không biết gì về các thành viên còn lại
* Đơn giản hóa business logic

+ Drawback: may be too centralized

Giải quyết vấn đề isolation:

+ semantic lock

+ reread value

+ pessimistic view

+ commutative update

+ version file

+ by value

1. **TCC**

Chia nhỏ business logic

TCC is a compensating transaction pattern for business model that is two-phased.

Các bước:

* Try: key component.check nghiệp vụ, reserve các resources cần thiết cho nghiệp vụ (business)
* Confirm: thực hiện business logic sử dụng các resources trên mà không check lại business
* Cancel: release các resource được lưu lại trong try

1. **Ngoài ra còn các pattern sau:**

XA protocol

Soft transactio

1. **Reference**

<https://medium.com/@Alibaba_Cloud/breaking-the-limits-of-relational-databases-an-analysis-of-cloud-native-database-middleware-2-d3e790de0673>

<https://docs.oracle.com/cd/B19306_01/server.102/b14231/ds_txns.htm>

<https://medium.com/swlh/handling-transactions-in-the-microservice-world-c77b275813e0>

<https://blog.couchbase.com/saga-pattern-implement-business-transactions-using-microservices-part/>

<https://servicecomb.apache.org/docs/distributed_saga_3/> (check)

1. **Info thêm**

Cách truyền thống để có được data consistency cho hệ thống đa nghiệp vụ, database hoặc message brokers là sử dụng distributed transaction.

Một hệ thống XA thì gồm có database XA, message broker, database drivers, messaging api và interprocess communication mechanism

Hầu hết các database SQL đều XA-compliant và nhiều message broker cũng vậy

1. **2PC vs. Saga vs TCC**

|  |  |  |  |
| --- | --- | --- | --- |
|  | 2 phase commit | Saga | TCC |
| ACID | Đảm bảo ACID principle | Thiếu isolation, cần phải tự implement để có thể tackle được vấn đề này |  |
| Transaction | Ngắn, immediate | Long running, long lived |  |
| Speed | Lâu vì synchronous, transaction sẽ khóa các object nó đang sử dụng nên các transaction khác không dùng được | Nhanh hơn vì asynchronous (không đồng bộ) |  |
| Nếu có lỗi xảy ra | Không có rollback cho commit phase | Compensating transaction |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |