</ Distributed transaction using Saga pattern

} /> [

/>

Speaker: Trung Tran

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</ Table of contents

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Prerequisites

Must have

- Basic knowledge in software architecture
- Understanding of remote communication between servers/software components
- Understanding of concurrency concepts

Nice to have

- C#, .NET knowledge
- Understanding of async communication
- Experience of Docker

/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Resources

Repository URL:

https://github.com/trannamtrung1st/SagaSeminar

Requirements:

- .NET SDK 6.0
- Docker Desktop

/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Saigon Technology

+300 software engineers

+10 years in business

3 development centers (2 offices in Singapore and USA)

Website: https://saigontechnology.com/

} /> [

/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Saigon Technology

1 011 01 110110 110111 1101

</ Contact

trannamtrung1st@gmail.com

(+84)76 544 1077

https://www.linkedin.com/in/trung-tran99/

} /> [

/>

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</>

Context & problems

01

}/>[

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</ Relational database & transactions

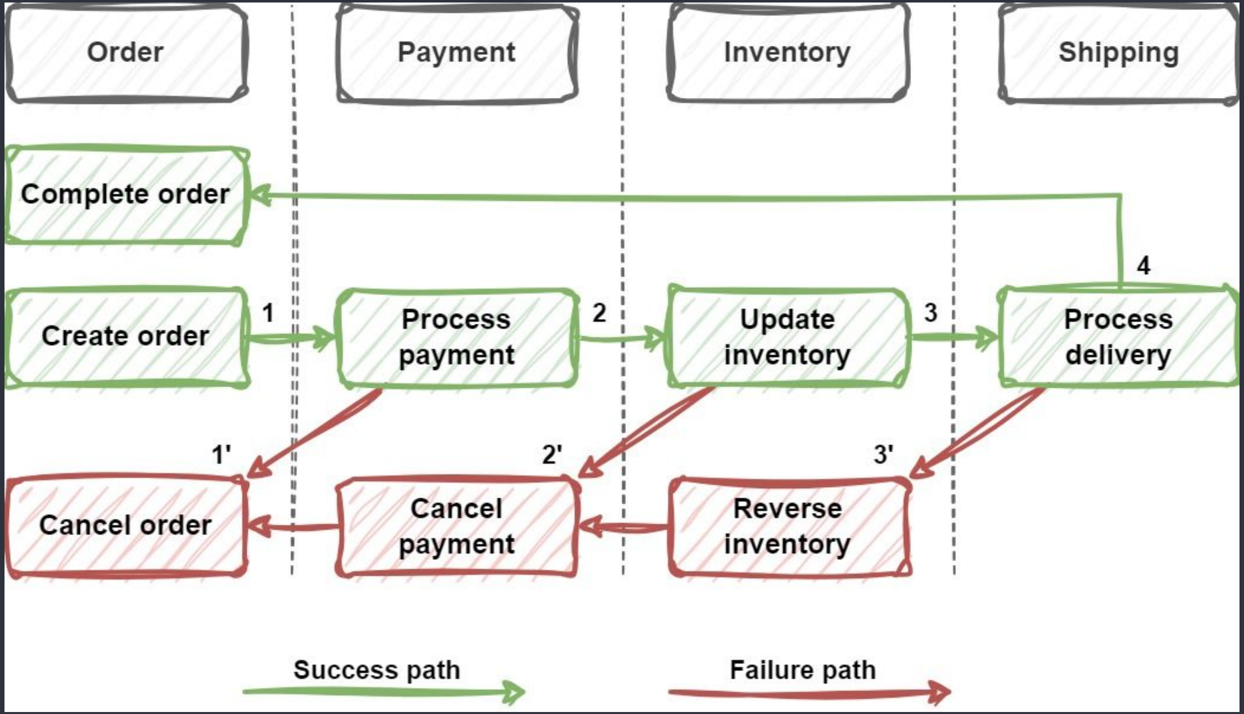
- Transaction is a single unit of logic or work
- ACID properties:
  - Atomic
  - Consistent
  - Isolated
  - Durable

/>

</ Transaction in a distributed system

- Atomicity: a single unit with set of operations that must all occur or none occur
- Consistency: data should only move from one state to another expected valid state in all participants
- Isolation: ensures that concurrent transactions result in the same outcome when running sequentially
- Durability: makes the commit status of transactions persistent and be able to bear with failure related to system or power outage

</ Sample order processing system



/>

</ Problems with a distributed transaction

Problems with transaction in a distributed environment

- Due to usage of database-per-microservice, NoSQL databases, message brokers, file storage

  -> They do not support Two-phase commit (2PC) protocol

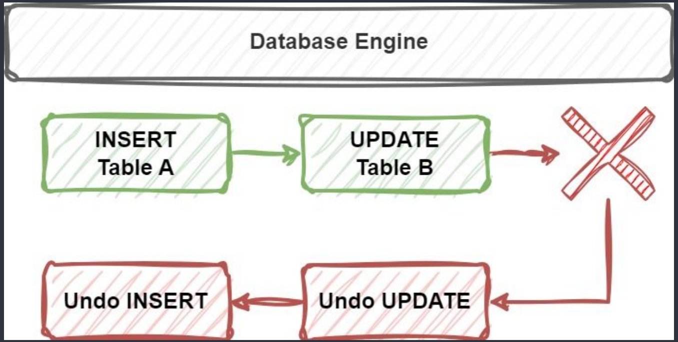
/>

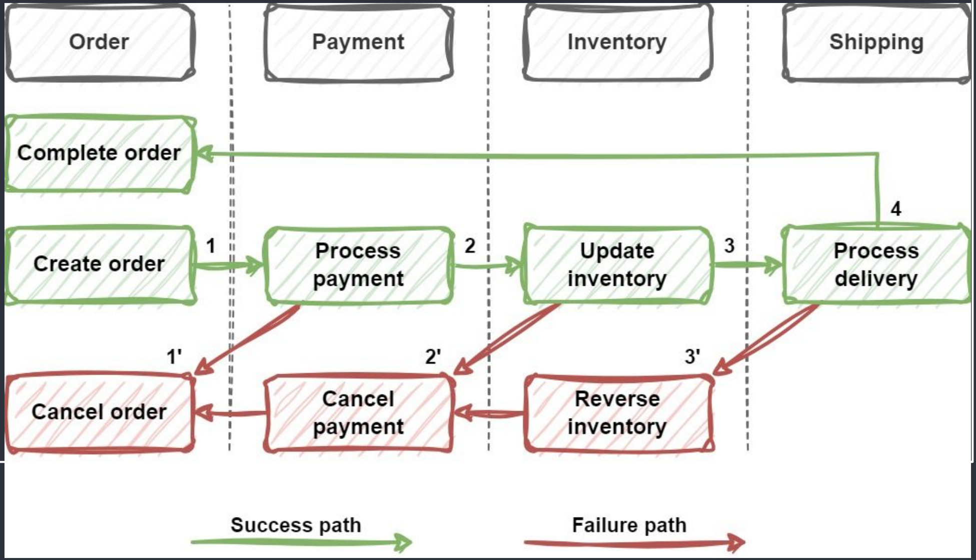
1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Problems with a distributed transaction

Problems with transaction in a distributed environment

- Complicated rollback operations



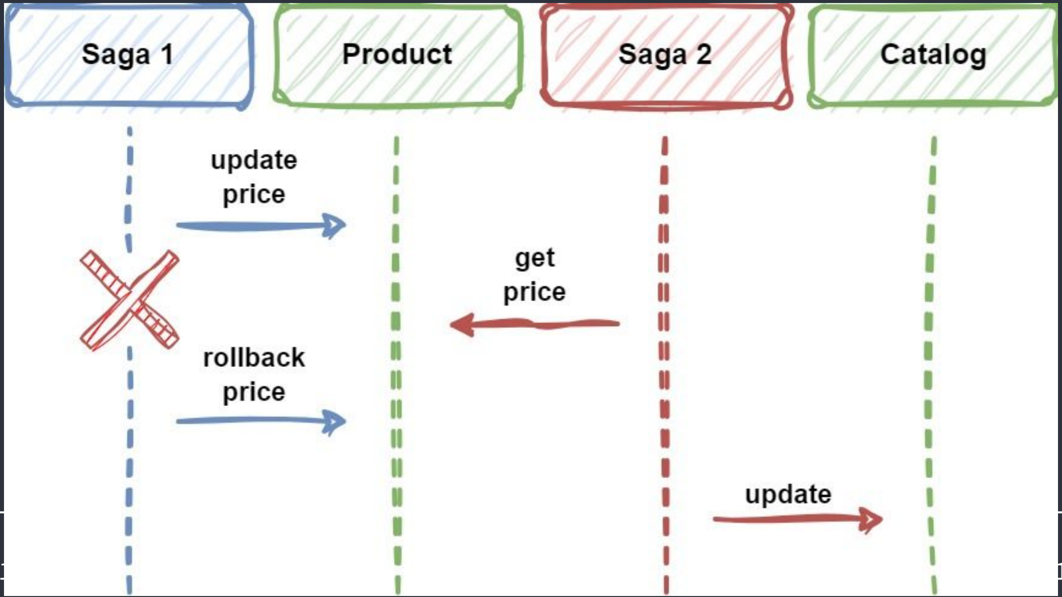


1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1

</ Problems with a distributed transaction

Problems with transaction in a distributed environment

- Atomicity, consistency and isolation is a challenge

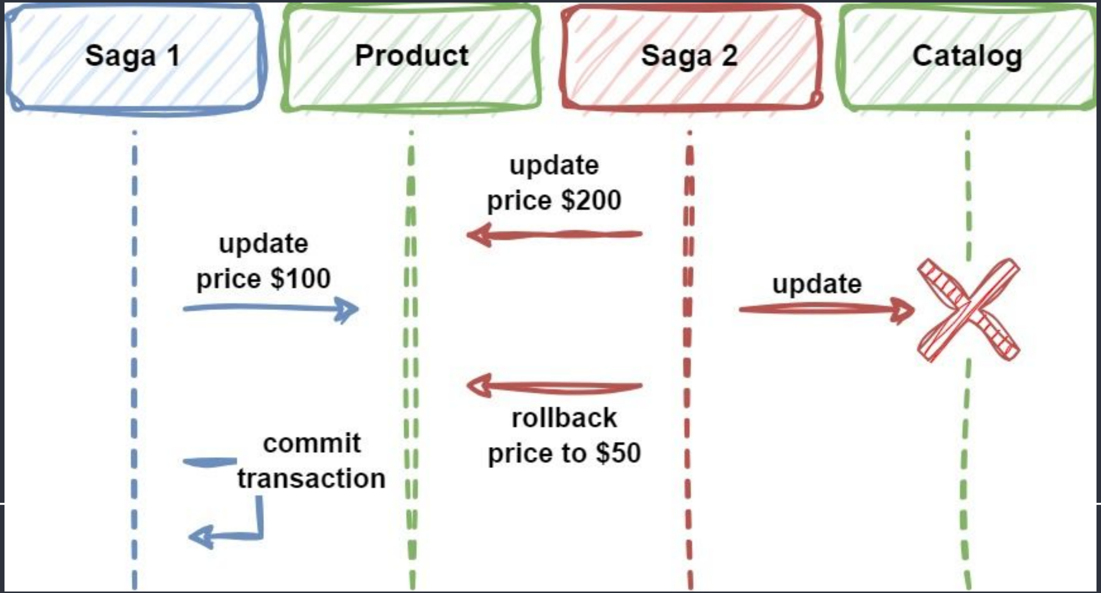


/>

</ Problems with a distributed transaction

Problems with transaction in a distributed environment

- Limitations with synchronicity and availability



/>

</>

Saga - The solution

02

} /> [

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

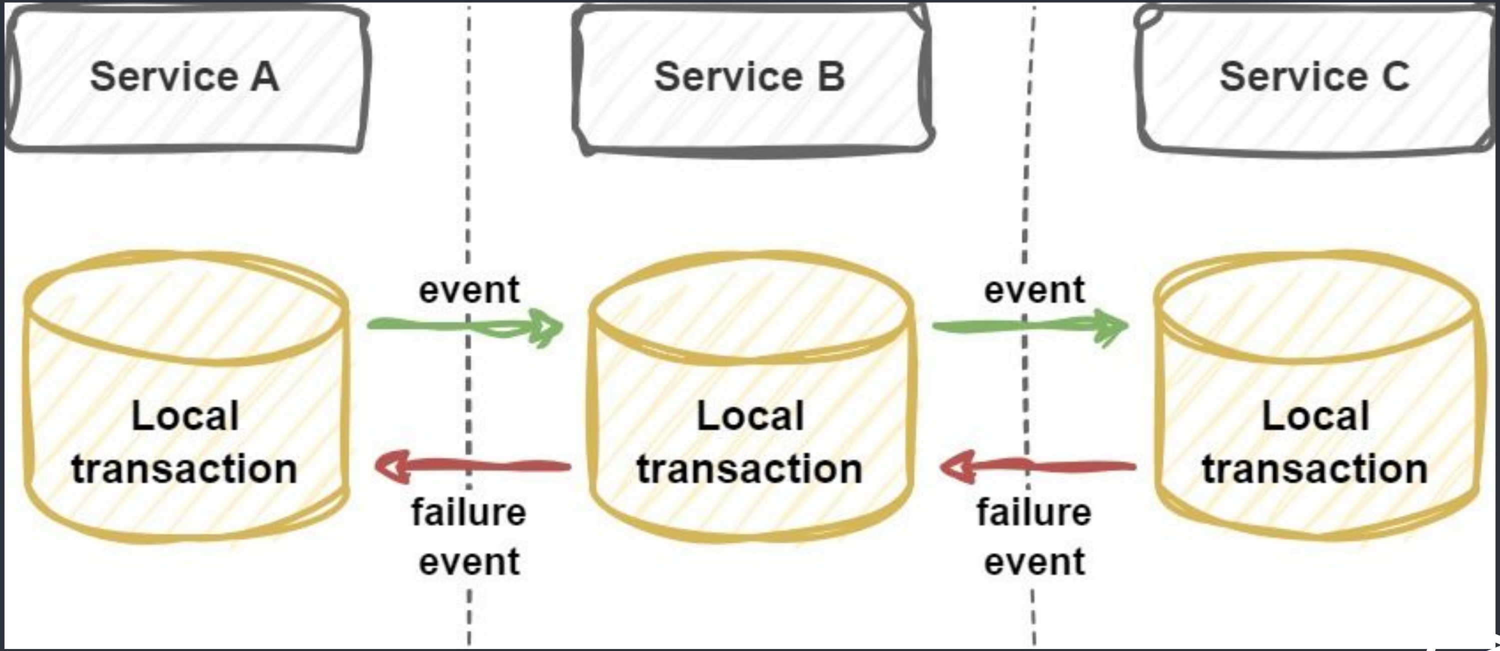
</ What is Saga pattern?

- A transaction management strategy involving with multiple local transactions
- A local transaction is performed by a saga participant
- Each local transaction performs its transaction, then publishes events to trigger the next transaction
- If local transaction fails, the saga will execute a series of compensating transactions that reverse/compensate the changes made before

/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ What is Saga pattern?



/>

</ Saga pattern - Concepts

- Saga participants
- Compensable & compensating transactions
- Pivot transaction
- Retryable transactions
- Transaction key

/>

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</ Saga participants

Services/software components that are responsible for one or more local transactions in the whole saga transaction

/>

</ Compensable, compensating transactions

- Compensable transactions can potentially be undo by executing another transaction with the opposite effect
- Compensating transactions reverse/compensate the changes made before by the compensable transactions

/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Pivot transaction

- The go/no-go (all or nothing) point in a saga
- If the pivot transaction commits, the saga runs the remaining transactions until done
- We can tell the saga is successful if a pivot transaction commits
- A pivot transaction can be neither compensable nor retryable, or it can be the last compensable transaction or the first retryable transaction in the saga

/>

</ Retryable transactions

- Follow pivot transaction and are guaranteed to succeed
- For example, in ordering application
  - Create order
    -> Compensating transaction will be Cancel order
  - Process payment
  - Update inventory
  - Delivery to customer → pivot transaction
  - Complete order → is retryable since all important steps are already successful

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</ Transaction key

- Each transaction should have a unique and consistent key across services
- Is used to retrieve the transaction's information
- Marks entities or operations as associated with a specific transaction
- Can be randomly generated and linked or must be able to construct using consistent hashing

/>

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</ Implementation approaches

There are two common implementations:

- Choreography
- Orchestration

Each has its own set of challenges and techniques to manage the workflow

/>

</>

Techniques in Saga

03

}/>[

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

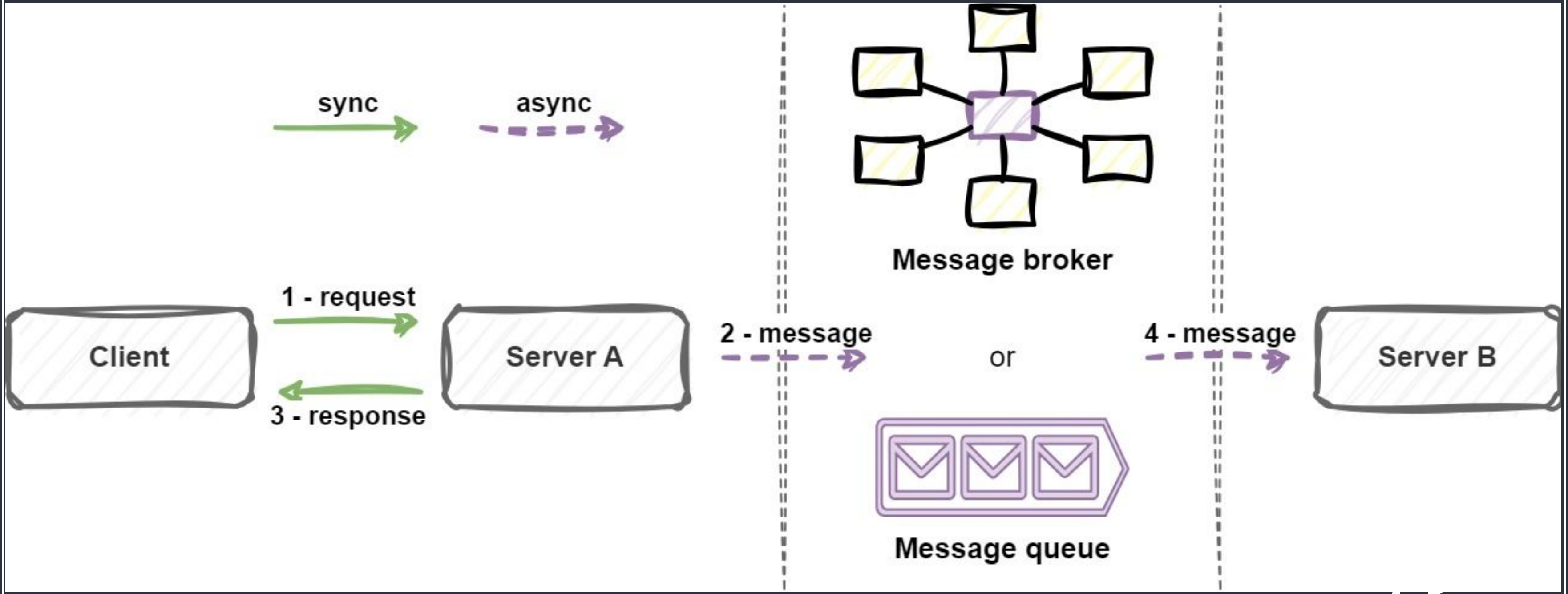
</ Common techniques & design patterns

- Asynchronous communication & Message broker
- Asynchronous request-reply
- "Replay" technique
- Other techniques to ensure resiliency and target isolation, consistency issues

/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Asynchronous communication & Message broker



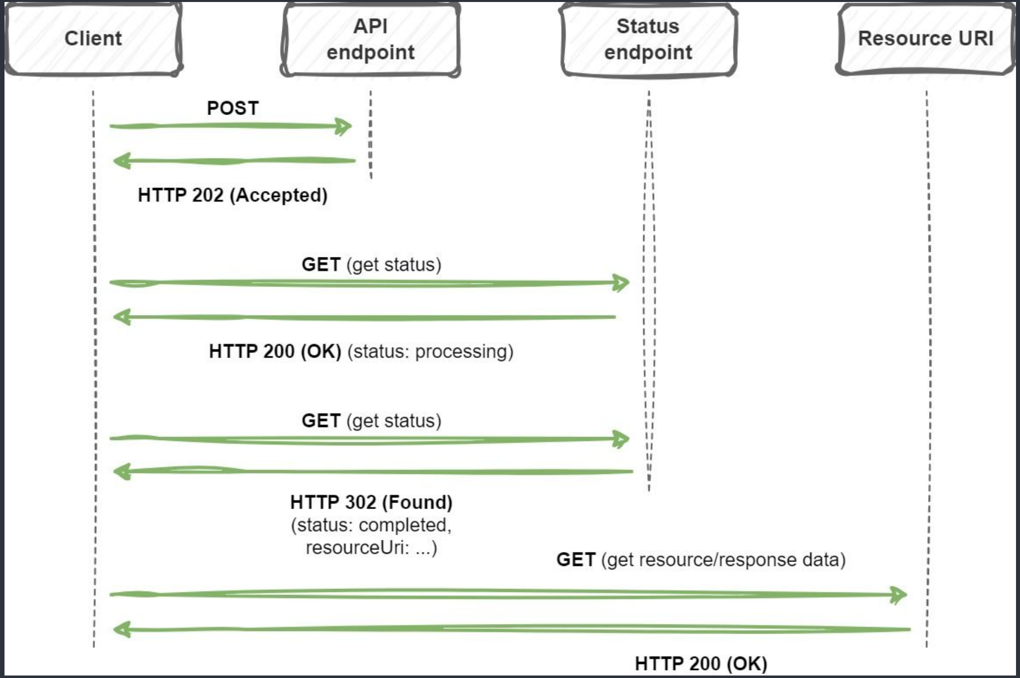
</ Synchronous vs Asynchronous communication

- Asynchronous fits Saga better

	Synchronous	Asynchronous
Technique	HTTP request	Message queue/broker
Response time	Client waits for response	Quick response
Retry	Client waits for retry	Retry in background
Throughput	Lower	Higher
Timeout	Has timeout	Almost no timeout

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Asynchronous request-reply



/>

</ "Replay" technique

- A group of logical operations can be replayed multiple times
- Each operation is coordinated by the orchestrator
- Operation status and result will be persisted
- In the next replay, persisted result will be used if available

-> Is usually used with Event-sourcing architecture

-> More clear and readable but complicated

/>

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</ "Replay" technique

```
try
{
    ThrowIfFailed(transaction);

    OrderModel order = await GetCreatedOrder(transaction);

    // 1st play
    PaymentModel payment = await ProcessPayment(order, transaction, transactionService, publisher);

    // 2nd play
    InventoryNoteModel inventoryNote = await InventoryDelivery(payment, transaction, transactionService, publisher);

    // 3rd play
    DeliveryModel delivery = await ProcessDelivery(inventoryNote, transaction, transactionService, publisher);

    // 4th play
    await CompleteOrder(delivery, transaction, transactionService, publisher);
}
catch (TransactionFailedException ex)
{
    // x? play (with failure)
    await Rollback(ex, transaction, transactionService, publisher);
}
catch (AsyncTransactionException ex)
{
    Console.WriteLine(ex.Message);
}
```

</ Other techniques

To ensure resiliency and target isolation, consistency issues

- Heartbeat pattern
- Retry
- Semantic lock
- Optimistic locking
- Different concurrent read techniques

/>

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</>

Saga - Choreography

04

} /> [

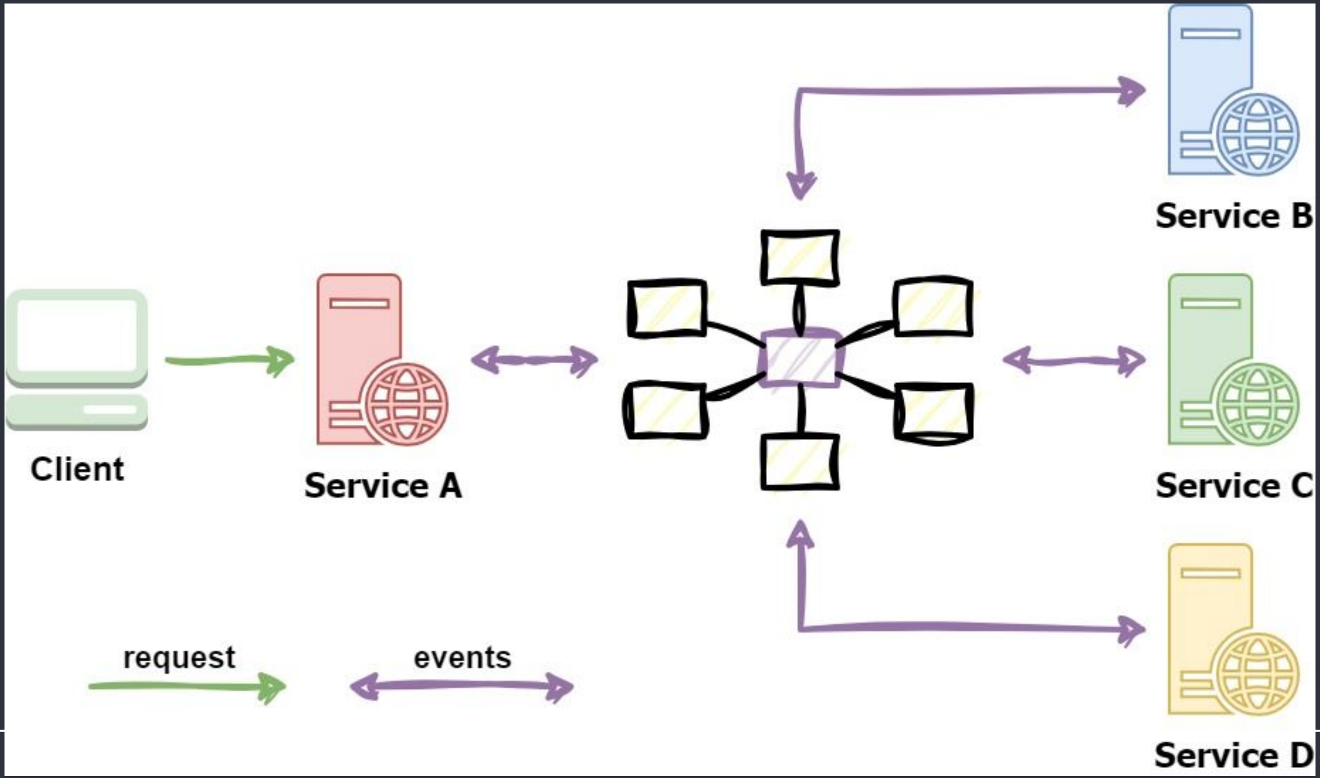
1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</ What is choreography-based saga?

- A way to coordinate sagas where events are exchanged using a message broker or any mechanism without a centralized controller
- Each local transaction publishes events that trigger local transactions in other participants
- Participants must know each other's events to be able to handle them

/>

</ What is choreography-based saga?



/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Apply to our problem [Success path]

1. Create order -> ?

/>

</ Apply to our problem [Success path]

1. Create order -> Order created ->
   Process payment -> Payment created ->
   Inventory delivery -> Delivery note created ->
   Process delivery -> Delivery created ->
   Complete order -> Order completed

/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Apply to our problem [Failure paths]

1. Create order -> Create order failed
2. ... Process payment -> Payment failed -> Cancel order
3. ... Inventory delivery -> Inventory delivery failed -> Cancel order, Cancel payment
4. ... Process delivery -> Delivery failed -> Cancel order, Cancel payment, Reverse inventory delivery

</ Apply to our demo [Publishing events]

```
public async Task PublishOrderCreated(OrderModel model)
{
    string message = JsonConvert.SerializeObject(new OrderCreatedEvent
    {
        Model = model
    });

    await _producer.ProduceAsync(nameof(OrderCreatedEvent),
        new Message<string, string>
        {
            Key = model.Id.ToString(),
            Value = message
        });
}
```

OrderService >

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Apply to our demo [Handling events]

```
2 references
public async Task HandleCreatePaymentWhenOrderCreated(CancellationToken cancellationToken)
{
    await StartConsumerThread(nameof(OrderCreatedEvent), async (message) =>
    {
        using IServiceScope scope = serviceProvider.CreateScope();

        OrderCreatedEvent @event = JsonConvert.DeserializeObject<OrderCreatedEvent>(message.Message.Value);

        IPaymentService paymentService = scope.ServiceProvider.GetRequiredService<IPaymentService>();

        await paymentService.CreatePaymentFromOrder(@event.Model);

    }, cancellationToken: cancellationToken);
}
```

PaymentService

/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Choreography - Pros & Cons

Pros

- Good for simple, few participants, no need coordination logic workflows
- No need to maintain orchestrator service
- No single point of failure
- Responsibilities are shared and distributed

Cons

- Cumbersome when adding new participants and steps
- Complicated dependencies (cyclic, tight coupling) between participants
- All services must be running to simulate a transaction -> difficult to do integration testing

Saga - Orchestrator

05

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

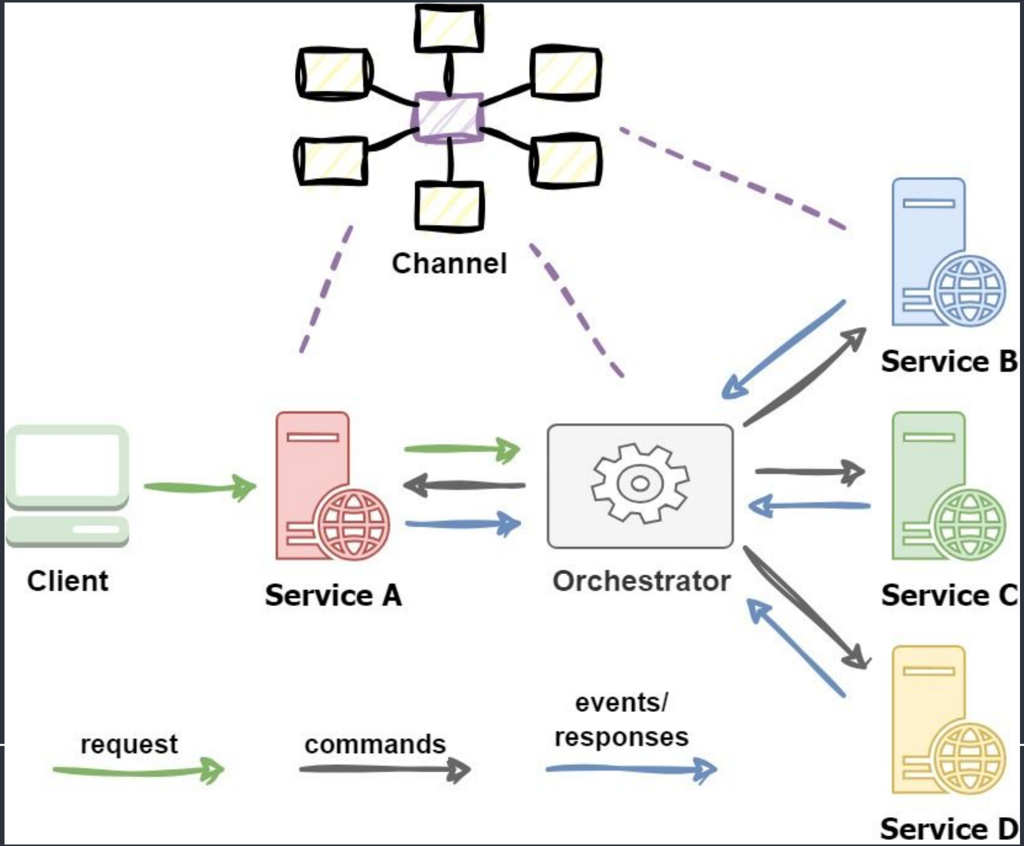
</ What is orchestrator-based saga?

- Orchestrator coordinates sagas using a centralized controller
- Trigger participants local transactions to perform in case of events
- Handle failures by running compensating transactions/actions
- Status can be managed and persisted for monitoring and management purposes

/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ What is orchestrator-based saga?



/>

</ Apply to our problem [Success path]

1. Create order -> Order created -> ?

/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Apply to our problem [Success path]

1. Create order -> Order created -> Orc ->
   C:Process payment -> Payment created -> Orc ->
   C:Inventory delivery -> Delivery note created -> Orc ->
   C:Process delivery -> Delivery created -> Orc ->
   C:Complete order -> Order completed -> Orc

Annotation

Orc: Orchestrator		*C:{action}: command that triggers action*

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Apply to our problem [Failure paths]

1. ... Process payment -> Payment failed -> Orc -> C:Cancel order
2. ... Inventory delivery -> Inventory ... failed -> Orc -> C:Cancel order, C:Cancel payment
3. ...

Annotation

Orc: Orchestrator *C:{action}: command that triggers action*

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Apply to our demo [Orchestrator handles events]

```
1 reference
protected async Task HandleOrderCreated(CancellationToken cancellationToken)
{
    await StartConsumerThread(nameof(OrderCreatedEvent), async (message) =>
    {
        using IServiceScope scope = serviceProvider.CreateScope();

        IOrderProcessingPublisher publisher = scope.ServiceProvider.GetRequiredService<IOrderProcessingPublisher>();

        OrderCreatedEvent @event = JsonConvert.DeserializeObject<OrderCreatedEvent>(message.Message.Value);

        await publisher.ProcessPayment(@event.Model);

    }, cancellationToken: cancellationToken);
}
```

Orchestrator

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</ Apply to our demo [Orchestrator publishes commands]

```
3 references
public async Task ProcessPayment(OrderModel order)
{
    string message = JsonConvert.SerializeObject(new ProcessPaymentCommand
    {
        FromOrder = order,
    });

    await _producer.ProduceAsync(nameof(ProcessPaymentCommand),
        new Message<string, string>
        {
            Key = order.Id.ToString(),
            Value = message
        });
}
```

Orchestrator

</ Apply to our demo (Services handle commands)

```
2 references
public async Task HandleProcessPayment(CancellationToken cancellationToken)
{
    await StartConsumerThread(nameof(ProcessPaymentCommand), async (message) =>
    {
        using IServiceScope scope = serviceProvider.CreateScope();

        ProcessPaymentCommand command = JsonConvert.DeserializeObject<ProcessPaymentCommand>(message.Message.Value);

        IPaymentService paymentService = scope.ServiceProvider.GetRequiredService<IPaymentService>();

        await paymentService.CreatePaymentFromOrder(command.FromOrder);

    }, cancellationToken: cancellationToken);
}
```

PaymentService

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Orchestrator - Pros & Cons

Pros

- Suitable for complex, many participants or new participants added over time workflows
- Good when flow of tasks and activities are under control
- No cyclic dependencies
- Separation of concerns

Cons

- Additional effort to maintain, implement coordination logic
- Single point of failure

/>

</>

Issues & considerations

06

} /> [

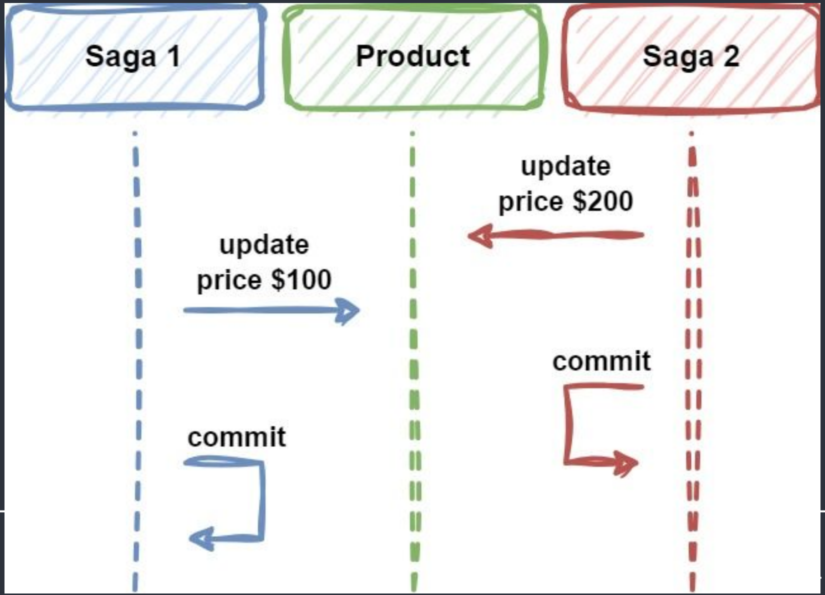
1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</ Some considerations

- Can be challenging in the beginning
- Hard to debug
- Complexity grows when there are more participants
- Consistency is relative only, because changes are committed to local databases
- Improper transient failures handling can cause side-effects
- Difficult to manage without proper monitoring and observability implemented

</ Common issues - Lost updates

- Lost updates, a participant updates data without reading latest changes

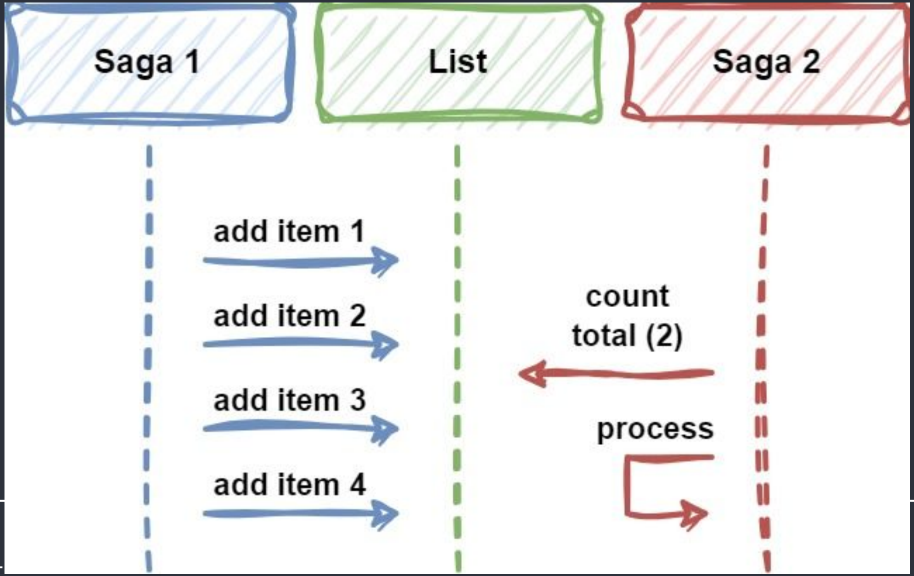


/>

1 0 1 1 0 1 1 0 1 1 0 1 0 1 1 0 1 1 1 1 1 0 1

</ Common issues - Dirty reads

- Dirty reads, when a participant reads data that is just partially updated by another saga

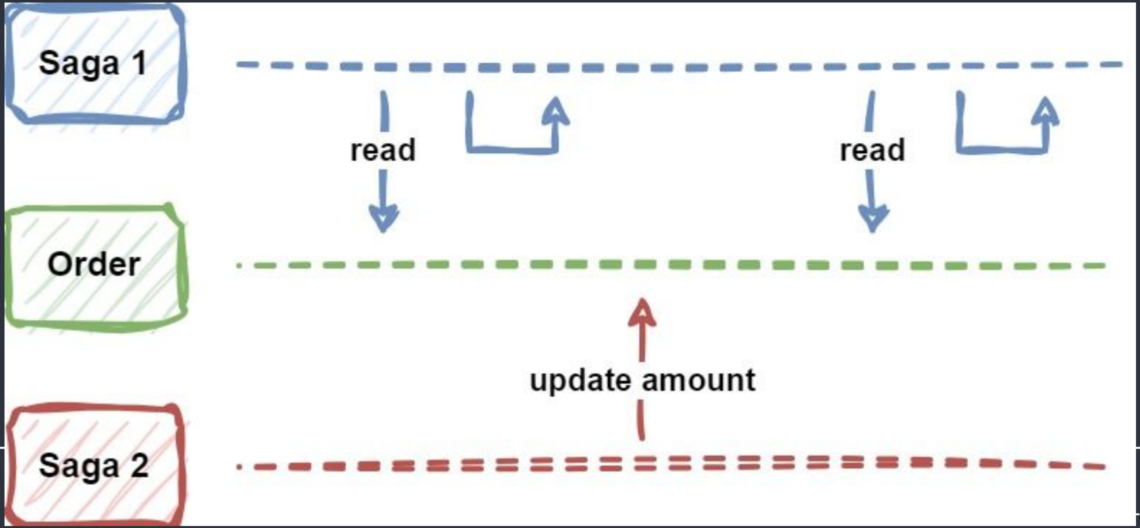


/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Common issues - Fuzzy/non-repeatable reads

- Fuzzy/non-repeatable reads, data is not consistent across participants since data can be updated at some points throughout the entire saga workflow



/>

1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Countermeasures against isolation issues

- Semantic lock
- Commutative updates
- Pessimistic view
- Reread value and restart if data changed
- Record operations in arriving order to execute sequentially
- Low-risk requests use sagas, high-risk requests favor distributed transactions (2PC, 3PC distributed transaction commit/rollback protocol)

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Solutions for resiliency

- Rate limiting
- Delay & retry
- Idempotency

/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</>

Practical usage

07

}/>[

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</ When to use

- Ensure data consistency in a distributed system without tight coupling
- Efficiently execute compensating actions if one of the operations fails in the distributed action sequence

/>

</ When to use

- Microservices
- Cloud-based applications
  - Saga pattern - AWS Prescriptive Guidance (amazon.com)
  - Saga pattern - Azure Design Patterns | Microsoft Learn
  - Saga pattern - IBM Cloud Architecture Center

/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ When to use

- Banking applications: Patterns and implementations for a banking cloud transformation - Azure Architecture Center | Microsoft Learn
- Media processing system: Gridwich saga orchestration - Azure Reference Architectures | Microsoft Learn

/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ When to NOT use

- Monolithic architecture
- Tightly coupled operations and transactions
- Cyclic dependencies

/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</ Summary

- So far, we have learned that:
  - The use of Saga pattern in distributed transaction
  - Some common techniques used in Saga
  - Two Saga implementations: Choreography & Orchestrator
  - Some common issues and challenging when apply Saga

/>

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 0 1 1 1 1 1 0 1

</>

Discussion & QnA

08

} /> [

1011 011 01 1011001 10 11011 011 01 110110 110111 1101

</ References

- Saga pattern - Azure Design Patterns | Microsoft Learn
- Patterns and implementations for a banking cloud transformation - Azure Architecture Center | Microsoft Learn
- Gridwich saga orchestration - Azure Reference Architectures | Microsoft Learn
- Saga Pattern in Microservices | Baeldung on Computer Science
- Saga Pattern for Microservices Distributed Transactions | by Mehmet Ozkaya | Design Microservices Architecture with Patterns & Principles | Medium
- Distributed transaction - SAGA pattern (viblo.asia)
- Implement a distributed transaction in microservices software system using Saga pattern | Careers Saigon Technology

/>

1 0 1 1   0 1 1   0 1   1 0 1 1 0 0 1   1 0   1 1 0 1 1   0 1 1   0 1   1 1 0 1 1 0   1 1 0 1 1 1   1 1 0 1

</ Thanks!

Do you have any questions?

trannamtrung1st@gmail.com

(+84)76 544 1077

https://www.linkedin.com/in/trung-tran99/

} /> [

/>

CREDITS: This presentation template was created by **Slidesgo**, and includes icons by **Flaticon**, and infographics & images by **Freepik**

Please keep this slide for attribution

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 0 1 1 1 1 0 1