Lost in transaction?

Strategies to deal with (in)consistency in distributed systems
Do A
Do B

All or nothing

once upon a time:

try {
    tx.begin();
    doA();
    doB();
    tx.commit();
} catch (Exception e) {
    tx.rollback();
}

or simply:

@Transactional
public void createCustomer(Customer cust) {
    // ...
}
Atomicity
Consistency
Isolation
Durability
Distributed systems
Distributed systems

Fallacies of distributed computing

The fallacies of distributed computing are a set of assertions made by L. Peter Deutsch and others at Sun Microsystems.

The fallacies

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
Distributed systems
But there is two-phase commit (XA)!!
Life beyond Distributed Transactions: an Apostle's Opinion

Position Paper

Pat Helland
Amazon.Com
705 Fifth Ave South
Seattle, WA 98104
USA
PHelland@Amazon.com

The positions expressed in this paper are personal opinions and do not in any way reflect the positions of my employer Amazon.com.

ABSTRACT

Many decades of work have been invested in the area of distributed transactions including protocols such as 2PC, Paxos, and various approaches to quorum. These protocols provide the application programmer a facade of global serializability. Personally, I have invested a non-trivial portion of my career as a strong advocate for the implementation and use of platforms that instead, applications are built using different techniques which do not provide the same transactional guarantees but still meet the needs of their businesses.

This paper explores and names some of the practical approaches used in the implementations of large-scale mission-critical applications in a world which rejects distributed transactions. We discuss the management of fine-grained pieces of application data which may be re-partitioned over time as the application grows. We also discuss the design patterns used in sending messages between these re-partitionable pieces of data.
“Grown-Ups Don’t Use Distributed Transactions

Pat Helland
Distributed Systems Guru
Worked at Amazon,
Microsoft & Salesforce
Starbucks does not use two phase commit

https://www.enterpriseintegrationpatterns.com/ramblings/18_starbucks.html

Photo by John Ingle
But we forfeit “C” and “I” for availability, graceful degradation, and performance

This tradeoff is fundamental.

BASE:

- Basically Available
- Soft-state
- Eventual consistency

PODC Keynote, July 19, 2000

That means

Do A

Do B

Temporarily inconsistent

Eventually consistent again

Violates "I" of ACID

Consistent

Local ACID

Local ACID

1 (micro-)service
1 aggregate
1 program
1 resource

t
You might know this from:

- Do A
- Do B

Consistent

Temporarily inconsistent

Eventually consistent again

Photo by Gerhard51, available under Creative Commons CC0 1.0 license.
Associative
Commutative
Idempotent
Distributed

2.0

(a + b) + c = a + (b + c)

a + b = b + a

f(x) = f( f(x) )

"Building on Quicksand" Paper

Pat Helland
Photo by pixabay, available under Creative Commons CC0 1.0 license.
Requirement: Idempotency of services!

Photo by pixabay, available under Creative Commons CC0 1.0 license.
Requirement: Idempotency of services!

Photo by Chr.Späth, available under Public Domain.
Distributed
It is impossible to differentiate certain failure scenarios:

Independant of communication style!
Network problems
Strategy: retry

Payment → charge → Credit Card

Charge Credit Card
- cardNumber
- amount

Charge Credit Card
- cardNumber
- amount
- transactionId

Not idempotent

Idempotent

has to be idempotent
Strategy: Cleanup

- Payment
- Credit Card
- Raise payment failed
- Make sure it is not charged!
- Cancel charge
- cardNumber
- amount
- transactionId
Some communication challenges require state.
Strategy: Stateful retry

Payment \[\xrightarrow{\text{charge}}\] Credit Card

Charge retry
Strategy: Stateful retry

- Make sure it is not charged!
Warning: Contains Opinion
Bernd Ruecker
Co-founder and Chief Technologist of Camunda

Bernd.ruecker@camunda.com
@berndruecker

Berlin, Germany
Let's use a lightweight OSS workflow engine for this:
Stateful retry

Stateful retry: e.g. 10 times, delay 15 min.

Charge credit card

Payment

REST

Credit Card
Stateful retry & cleanup

Charge credit card → Payment received
- no retries left → Cancel charge → Payment failed
- Payment failed

Payment → REST → Credit Card

Do it reliably
Live hacking

https://github.com/flowing/flowing-retail/tree/master/rest
Embedded Engine Example (Java)

https://blog.bernd-ruecker.com/architecture-options-to-run-a-workflow-engine-6c2419902d91
Remote Engine Example (Polyglot)

https://blog.bernd-ruecker.com/architecture-options-to-run-a-workflow-engine-6c2419902d91
A relatively common pattern

1. Receive
2. Business Logic
3. Send response
4. Send additional events

Service (e.g. Go)

Kafka / Rabbit

RDMS
Without cross-service transactions:

A. Compensating transactions
   - abort/rollback at app level
   (Garcia-Molina & Salem, 1987)

B. Apologies
   - detect & fix constraint violations
   (Helland & Campbell, 2009)
   (after the fact, rather than preventing them)
That means

In case of failure: Compensate or apologise
Memories, Guesses, and Apologies

Well, here I am blogging on the bus with my newly installed Windows Live Writer!!!

This blog is a text version of a five minute "Gong Show" presentation I did at CIDR (Conference on Innovative Database Research) on Jan 8,2007.

All computing can be considered "work". Furthermore, it often active "work".

Microsoft
Compensation – the classical example

1. book hotel
2. book car
3. book flight

4. cancel hotel
5. cancel car
6. In case of failure trigger compensations

Saga

Note: The diagram illustrates the sequence of booking and canceling hotel, car, and flight, with a trigger for compensations in case of failure.
2 alternative approaches: choreography & orchestration
Event-driven choreography

- Trip
- Flight
- Hotel
- Car

Request trip → Trip
Trip booked → Trip
Trip requested → Hotel
Hotel booked → Hotel
Flight booked → Flight
Car booked → Car
Event-driven choreography

- Trip requested
- Hotel booked
- Car booked
- Flight failed
- Hotel canceled
- Car canceled
- Trip canceled
- Perform undo (cancel hotel)
- Perform undo (cancel car booking)
- Request trip
- Trip failed
- Flight failed
- Trip failed
The danger is that it’s very easy to make nicely decoupled systems with event notification, without realizing that you’re losing sight of that larger-scale flow, and thus set yourself up for trouble in future years.

https://martinfowler.com/articles/201701-event-driven.html
The danger is that it’s very easy to make nicely decoupled systems with event notification, without realizing that you’re losing sight of that larger-scale flow, and thus set yourself up for trouble in future years.

https://martinfowler.com/articles/201701-event-driven.html
The danger is that it’s very easy to make nicely decoupled systems with event notification, without realizing that you’re losing sight of that larger-scale flow, and thus set yourself up for trouble in future years.

https://martinfowler.com/articles/201701-event-driven.html
Classical example

1. book hotel
2. book car
3. book flight

In case of failure trigger compensations

6. cancel hotel
5. cancel car

book trip
If your transaction involves 2 to 4 steps, choreography might be a very good fit.

However, this approach can rapidly become confusing if you keep adding extra steps in your transaction as it is difficult to track which services listen to which events. Moreover, it also might add a cyclic dependency between services as they have to subscribe to one another’s events.
Microservice pioneers have become aware

Traditionally, some of these processes had been orchestrated in an ad-hoc manner using a combination of pub/sub, making direct REST calls, and using a database to manage the state. However, as the number of microservices grow and the complexity of the processes increases, getting visibility into these distributed workflows becomes difficult without a central orchestrator.
Implementing changes in the process

We have a new basic agreement with the car rental agency and can cancel for free within 1 hour – do that first!
Implementing changes in the process

You have to adjust all services and redeploy at the same time!

We have a new basic agreement with the car rental agency and can cancel for free within 1 hour – do that first!
What we wanted vs. what we got

Photo by Lijian Zhang, available under Creative Commons SA 2.0 License and Pedobear19 / CC BY-SA 4.0
Orchestration
Orchestration

We have a new basic agreement with the car rental agency and can cancel for free within 1 hour — do that first!

You have to adjust one service and redeploy only this one!
Describe orchestration with BPMN

Saga Pattern (implemented by BPMN compensation)
The workflow is part of the service
The workflow is part of the service
The workflow is part of the service

Workflow engine itself can run decentralized or centralized
Modular Services
with Distributed Sagas

Only Flight Booking

Flight Booking & Feral Concurrency Control Mechanisms
“Can this really handle 15k requests per second?”
Microservice Orchestration Engine

Zeebe scales orchestration of workers and microservices using visual workflows. Zeebe is horizontally scalable and fault tolerant so that you can reliably process all your transactions as they happen.

Zeebe is currently a tech preview and under heavy development - See Roadmap.

Featuring:
- Visual Workflows
- Audit Logs and History
- Horizontal Scalability
- Durability & Fault Tolerance
- Fully Message-Driven
- Easy to operate
- Language agnostic

Learn more
Graphical models?
Clemens Vasters
Architect at Microsoft

http://vasters.com/archive/Sagas.html
Clemens Vasters
Architect at Microsoft

http://vasters.com/archive/Sagas.html
Living documentation for long-running behaviour
Visual HTML reports for test cases
BizDevOps
Saga with AWS Step Functions

Thoughts on the state machine | workflow engine market
Thoughts on the state machine / workflow engine market

- **Stack Vendors**, **Pure Play BPMS**, **Low Code Platforms**
  - PEGA, IBM, SAG, ...

- **OSS Workflow or Orchestration Engines**
  - Camunda, Zeebe, jBPM, Activiti, Mistral, ...

- **Integration Frameworks**
  - Apache Camel, Balerina, ...

- **Cloud Offerings**
  - AWS Step Functions, Azure Durable Functions, ...

- **Data Pipelines**
  - Apache Airflow, Spring Data Flow, ...

- **Homegrown frameworks to scratch an itch**
  - Uber, Netflix, AirBnb, ING, ...

Does it support stateful operations?

Does it support the necessary flow logic?

Does it support BizDevOps?

Does it scale?
My personal pro-tip for a shortlist ;-)
Recap

• Grown ups don’t use distributed transactions but eventual consistency
• Idempotency is super important in distributed systems
• Some consistency challenges require state
• Know some strategies
  • Stateful retry & cleanup
  • Saga / Compensation
  • Apologies
Thank you!