Opportunities & Pitfalls of Event-Driven Utopia

@berndruecker
Why this talk

Software Development Architecture and Design 2018 Q4 Graph
http://infoq.link/architecture-trends-2019

- Blockchain
- HTTP2
- Service Meshes (Linkerd, Istio)
- Chaos Engineering
- Event-Driven Architecture

Software Development Architecture and Design 2019 Q1 Graph
http://infoq.link/architecture-trends-2019

- Blockchain and Distributed Ledgers
- Service Meshes (Envoy, Linkerd, Istio)
- HTTP3
- Event-Driven Architecture (and Event Sourcing)
- Eventual Consistency
- REST
Towards the end of last year I attended a workshop with my colleagues in ThoughtWorks to discuss the nature of “event-driven” applications.

The biggest outcome of the summit was recognizing that when people talk about “events”, they actually mean some quite different things. So we spent a lot of time trying to tease out what some useful patterns might be.
Agenda

1. Events on the inside

Service 1

2. Events on the outside

Service 2
Agenda

1. Events on the inside

Service 1

2. Events inside out

3. Events on the outside

Service 2
Agenda

1. Events on the inside

Service 1

2. Events inside out

3. Events on the outside

Service 2
Once upon a time...

BBC architecture
(box - arrow - box - arrow - cylinder)
Every architecture diagram you'll ever need
The great thing about this architecture
The problem

Application

RDMS

not webscale
resiliency is expensive

Does not fit in Kubernetes:

RDMS
Immutability Changes Everything!
Append-only Log

- bank account created
- +2,500 $ transferred
- -14.99$ paid by credit card

Current Balance = 2,485.01 $

Persistent state

Account # | Balance
---|---
12345 | 2,500$
Persistent change

Append-only Log

+2,500 $ transferred

bank account created

-14.99$ paid by credit card

Event

Bank Account Created
2019/04/16 11:00
# 12345

Event

Money Transfer Received
2,500$
2019/04/16 11:00
# 12345
Event Sourcing in a nutshell

1. Create Customer
2. Read events
3. Build internal state
4. Validation and Invariant Checks
5. Add Customer Created Event
6. Async publish Customer Created Domain Event

Customer Event Store

- e.g. Customer Created, Customer Credit Limit Approved, ...

Replay

Business Logic

Save event

Internal State
Working without distributed transactions

1. Create Customer

2. Read events

3. Build internal state

4. Validation and Invariant Checks

5. Add Customer Created Event

6. Async publish Customer Created Domain Event

This is the only atomic operation required

Customer Event Store

e.g. Customer Created, Customer Credit Limit Approved, ...
Traditional Architecture

1. Create Customer Account
2. Persist state changes
3. Remote Communication

RDMS

Customer

Business Logic
Open Account

Account
Life beyond Distributed Transactions: an Apostate’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.

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 façade 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 instead of 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 repartitioned over time as the application grows. We also discuss the design patterns used in sending messages between these repartitionable pieces of data.
"Grown-Ups Don’t Use Distributed Transactions"
Outbox pattern in traditional architectures

1. Create Customer

2. Persist state changes

3. Remote Communication
   Async after first transaction!

- TX1: Business Logic, Job: "Open Account"
- TX2: Execute: "Open Account"

RDMS
Outbox pattern – Implementation Approaches

1. Create Customer
2. Persist state changes
3. Remote Communication

Customer

TX1

Business Logic
Job: Open Account

TX2

Execute: Open Account

Account

Scheduler
Database
Transaction Log
Workflow
Automation

RDMS
1. Create Customer

2. Persist state changes

3. Remote Communication: Async after transaction!

Capture Request | Business Logic | Job: "Open Account" | Execute: "Open Account"

RDMS

*Idempotency*
Idempotency

1. Create Customer Account
2. Persist state changes
3. Remote Communication

Customer
- TX1: Capture Request
- TX2: Business Logic, Job: "Open Account"
- TX3: Execute: "Open Account"

Account

RDMS
Working without distributed transactions

1. Create Customer
2. Read events
3. Build internal state
4. Validation and Invariant Checks
5. Add Customer Created Event
6. Async publish Customer Created Domain Event

This is the only atomic operation required

Customer Event Store

- e.g. Customer Created, Customer Credit Limit Approved, ...
Events on the inside.
An example from my world
Bernd Ruecker
Co-founder and Chief Technologist of Camunda

mail@berndruecker.io
@berndruecker

Warning: Contains opinion!
We offer two different workflow engines. Why?

- **Camunda**: Persistent State
- **Zeebe**: Persistent change
1. INSERT

2. UPDATE

3. UPDATE

<table>
<thead>
<tr>
<th>Workflow Instance Id</th>
<th>Current Activity</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>RetrievePayment</td>
<td>running</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Workflow Instance Id</th>
<th>Current Activity</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>ShipGoods</td>
<td>running</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Workflow Instance Id</th>
<th>Current Activity</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>OrderDelivered</td>
<td>ended</td>
</tr>
</tbody>
</table>
Append-only Log

1. create workflow instance
   - start event occurred
   - activity activated
   - lock created
   - complete task
   - activity completed

   workflow instance created
   sequence flow taken
   task created
   task locked
   task completed
   sequence flow taken

2. Workflow Engine

Retrieve payment
Ship goods
Event Handling, Replication & Single Writer

1. send
2. append command
3. store & replicate command
4. process
5. respond
6. append event
7. store & replicate event

Single Writer (single thread)

Stream Processor

RocksDB

Broker (Leader)

Follower
What we do different

1. Send
2. Append command
3. Store & replicate command
4. Process
5. Respond
6. Append event
7. Store & replicate event

- Store and replay commands
- Persist & replicate internal state
- Single Writer (single thread)
- Stream Processor
- Delete records that are fully processed
Consistency
Availability
Partition
Zeebe is CP

1. **send** command
2. **append command**
3. **store & replicate command**
4. **process**
5. **respond**
6. **append event**
7. **store & replicate event**

**Single Writer** (single thread)

**Stream Processor**
Horizontal scalability by partitioning

Every workflow instance is exactly handled by one partition
Queries and read models

Zeebe Broker

Streaming Exporter

elasticsearch
Recap 1 – Events on the inside

# Natural mechanism to build scalable services in distributed systems (with Outbox & co included)

But

# You have to think about reads, queries & eventual consistency
# Much less industry experience available
Agenda

1. Events on the inside
   Service 1

2. Events inside out

3. Events on the outside
Event Store and Messaging

1. Create Customer

Customer

Customer Event Store
Merge Messaging and Event Store

1. Create Customer

Customer Event Store
Merge messaging and event store

1. Create Customer

Customer

Shared Event Store
Enter the world of Kafka...
Merge messaging and event store

1. Create Customer

Shared Event Store
Kafka as transport

1. Create Customer

Used as queue (but persistent!)

Customer
Agenda

1. Events on the inside
2. Events inside out
3. Events on the outside
once upon a time

Customer

Change Address

Billing
Event Notification

Account

Customer

Address changed

Billing
Event Notification

Reverse direction of dependency

Customer → Change Address → Billing

Billing → Address changed → Customer

Who changed the address? The customer.
What's the general direction of dependency? From billing to customer.
Event Notification

AdressChanged
{
    customerId: 42
}

Ask for details
Event-carried State Transfer

```json
AddressChanged
{
  customerId: 42,
  oldAddress: ...,
  newAddress: ...
}

CustomerMoved
{
  ...,
}

CustomerChanged
{
  customerId: 42,
  status: A,
  address: ...
}
```
This decision is complex

Reverse direction of dependency

Customer → Change Address

Billing ← Customer

Billing → Address changed

Billing ← Customer
Example

Change Address

Address

Submit

Incoming Email

From: bla@company.com
Date: 2019-04-23 09.05

To confirm your address change please click on this link:

http://company.com/confirm?id=82e97d49-166c-4862-9973-4db58e6225d
Example

**Change Address**

Address

Submit

**Direction of dependency**

**Notification**

http://company.com/confirm?id=8c93
d49-166c-4352-9933-46b3f1ed25d

**Customer**

Address change requested

Address change confirmed
Example

Change Address

Address
Submit

Customer

Address changed

Change Adress

Send mail

Confirmation

Notification

http://company.com/confirm?id=82e97d49-166c-4b56-9973-4db348e6225d

'Confirmation'

approved

direction of dependency

Send mail

'Confirmation'

Address changed

Customer

Change Address
Challenge:
Command vs. Event
It is **NOT** about communication protocols

- **Customer**
  - Change Address
- **Billing**

- **Customer**
  - **Address changed**
- **Billing**

**It can be messaging, REST, whatever, ....**
Manifold ways of transport

- Apache Kafka
- RSS
- Webhooks
- RabbitMQ
Manifold ways of transport

Azure Service Bus
Azure Event Grid
Azure Event Hub
- Event: Fact, happened in the past, immutable
- Command: Intend, Want s.th. to happen, The intention itself is a fact
- Query
The Customer Needs To Be Sent A Message To Confirm Address Change Event

Send Message

Wording of recipient

Wording of Sender
Examples

More general, does not need to know who is retrieving payments

More general, does not need to care about who is interested in address changes

order → Payment

Retrieve Payment

Subscription

Billing

Address Changed

Customer

Order

Send Mail

Notification

Global service

Goods Shipped

Payment Received

Order Placed

Order Notification

Service that can handle notifications for orders autonomously
Distributed Monoliths

Authorization Service

- Document moved
- Document attached
- Page created
- Page moved

Document Context

Page Context

...
Define stable contract/API instead

Authorization Service

Add auth

Document Context

Page Context

...
Next challenge: Event chains
Event Chains

Registration requested → Event Bus

Credit checked → Address checked

Address checked → Customer registered

Credit Check

Adress Check

Customer
How does customer registration work?
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
Monitoring Workflows Across Microservices

Monitor and Managing Workflows Across Collaborating Microservices

Key Takeaways

- Peer-to-peer communication between components can lead to emergent behavior, which is challenging for developers, operators, and business analysts to understand.

- You need to make sure to have the overview of all the backwards-and-forwards communication that is going on in order to fulfill a business capability.

- Solutions that provide an overview range from distributed tracing, which typically misses the business perspective; data lakes, which require some effort to tune to what you need to know; process tracking, where you have to model a workflow for the tracking; process mining, which can discover the workflow, all the way through to orchestration, which comes with visibility built in.

https://www.infoq.com/articles/monitor-workflow-collaborating-microservices
Typical approaches

Distributed Tracing

Data Lake / Event Monitoring

Process Mining

Process Tracking
What we currently build with customers...

Camunda Optimize

Registration requested → Credit checked → Address checked → Customer registered → Registration completed

Elastic
All great – until you have to move...
Changes required for an additional check
Changes required for an additional check
Alternative flow

Registration requested

Address checked

Credit checked

Criminal checked

Customer registered

Credit Check

Adress Check

Kafka

@berndruecker
“Credit checks got more expensive, do that only if all other checks succeed”
Keep it stable, just move sticks with yellow color to the top.

How hard can it be?
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
of course these two services could be merged
Changes

Registration requested → Check credit
Check credit → Credit Check
Credit Check → Customer on-boarding
Customer on-boarding → Address checked
Address checked → Adress Check
Adress Check → Criminal Check
Criminal Check → Customer checked
Customer checked → Customer registered
Customer registered
Comparison

See also https://www.infoworld.com/article/3391592/how-to-tame-event-driven-microservices.html

2 changes, criminal check can be deployed first
In my world...

Customer on-boarding

Leverage Workflow Engine & BPMN within Service
Local Orchestration

Customer on-boarding

Central Orchestration Service
Software Development

Architecture and Design 2019 Q1 Graph

http://infoq.link/architecture-trends-2019

- Reactive Programming
- Functional Programming
- CQRS
- Actor Model
- "Serverless" (FaaS/BaaS/DBaaS/PaaS)
- gRPC and HTTP/2
- GraphQL
- Evolutionary Architecture
- (Lightweight) workflow and decision automation platforms
- Correctly built distributed systems
- "Architect as technical leader"
- Event-Driven Architecture (and Event Sourcing)
- Eventual Consistency
- Microservices
- Domain-Driven Design
- Behaviour-Driven Design
- Test-Driven Design
- REST
Recap 2
# Commands vs. Events: Decide about the direction of dependencies
# Beware of event-chains and avoid losing sight
# Balance choreography and orchestration
Recap

1. Events on the inside
   - Persistent state vs persistent change
   - Event sourcing & Event Store
   - Consistency & CAP
   - Read Models & CQRS

2. Events inside out

3. Events on the outside
   - Events as API
   - Event vs Command
   - Event chains & visibility
   - Orchestration vs Choreography

Shared Event Store
Want to see code?
Events on the inside

Events on the outside

Nothing for the faint of heart...
Nothing for the faint of heart...

...but doable...

...and worth it
Thank you!
Contact: mail@berndruecker.io
@berndruecker

Slides: https://berndruecker.io

Blog: https://medium.com/berndruecker

Code: https://github.com/berndruecker


https://www.infoq.com/articles/events-workflow-automation

https://thenewstack.io/5-workflow-automation-use-cases-you-might-not-have-considered/