Microservices: An Eventually Inconsistent Architectural Style?

Cesare Pautasso

http://www.pautasso.info
c.pautasso@ieee.org
@pautasso
University of Lugano (USI)
Faculty of Informatics

- Opened 2004
- 30 Professors
- 150+ PhD/Postdoc researchers
- New Software Institute
- New Master in Software and Data Engineering
Architecture, Design and Web Information Systems Engineering

- RESTful Business Process Management
- RESTalk - RESTful Conversation Modeling/Mining
- BenchFlow - a benchmark for workflow engines
- Liquid.js - Cross-Device, Decentralized Web Apps
- ASQ - Interactive Web Lectures
- Parallel JavaScript/Multicore Node.JS
- NaturalMash - Mashups with Natural Language

http://design.inf.usi.ch
Microservices: An Eventually Inconsistent Architectural Style?

Microservices by definition let each service independently manage its own database. In this talk we illustrate the ultimate consequences of the Polyglot persistence principle, which can be summarized using the BAC theorem: When Backing up a microservice architecture, it is not possible to have both Consistency (after recovery) and full Availability (while backing up the system). In other words, loosely coupled Microservice architectures are doomed to become inconsistent after disaster strikes. We will present and compare several coping strategies to deal with this limitation and discuss how they impact the monolith decomposition process and the corresponding service API boundary design.
Microservices

The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own container and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.
Microservices

The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own container and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.
**Monolith**

**Customer**

**Order**

**Microservice**

**Customer**
Monolith

Customer

Order

Microservices

Customer

Order
How to connect two microservices with HTTP?

Please enter your answer and click submit
Remote Procedure Call (Poll)
Remote Procedure Callback (Push)
Message Bus
Representational State Transfer
The Web as a Software Connector
soap+xml over HTTP

call()

POST

200

grpc+proto over HTTP/2
The Web as a Tunnel

POST

200

POST

202 Accepted

POST

s⇒s'
Representational State Transfer

Remote Procedure Call (Poll)

GET

PUT

call()

s'

Message Bus

Remote Procedure Callback (Push)

HTTP-*

receive

send

callback(s')
Representational State Transfer

Remote Procedure Call (Poll)

**Coupling (Time)**

Message Bus

Remote Procedure Callback (Push)
Representational State Transfer

Remote Procedure Call (Poll)

Cardinality

1:N

Message Bus

Remote Procedure Callback (Push)

GET

PUT

call()

s'

callback(s')

send

receive

1:1
Shared Web Resource
Multicast
Representational State Transfer

Message Bus (Multicast)
REST/WWW

Resource

Shared Memory

PUT

GET

GET

MEST/ESB

Queue

Message Passing

send

receive

receive
How **not** to connect two microservices
For us service orientation means encapsulating the data with the business logic that operates on the data, with the only access through a published service interface. No direct database access is allowed from outside the service, and there’s no data sharing among the services.

Werner Vogels, Interviews Web Services: Learning from the Amazon technology platform, ACM Queue, 4(4), June 30, 2006
Do you:

- Operate more than one microservice?
- Use polyglot persistence?
- Avoid storing everything in the same database?
- Assume eventual consistency?
Stateful Microservices

Microservices prefer letting each service manage its own database, either different instances of the same database technology, or entirely different database systems - an approach called Polyglot Persistence.

M. Fowler, J. Lewis  https://www.martinfowler.com/articles/microservices.html
Eventual Inconsistency

Microservice architectures are doomed to become inconsistent after disaster strikes

Stateful Microservices

Microservices prefer letting each service manage its own database, either different instances of the same database technology, or entirely different database systems - an approach called Polyglot Persistence.

M. Fowler, J. Lewis  https://www.martinfowler.com/articles/microservices.html
Devops meets Disaster Recovery
How do you back up a monolith?
How do you back up one microservice?
How do you back up an entire microservice architecture?

Are you sure?
Data relationships across microservices = Hypermedia
Independent Backup

Backups taken independently at different times
Disaster Strikes

One microservice is lost
Recovery from Backup

Broken link after recovery
Eventual Inconsistency

Broken link after recovery
Synchronized Backups

Backups of all microservices taken at the same time.
Limited Availability

No updates allowed anywhere while backing up the microservices
The BAC theorem

When Backing up a microservice architecture, it is not possible to have both Consistency and Availability
Consistency

During normal operations, each microservice will eventually reach a consistent state

**Referential integrity**: links across microservice boundaries are guaranteed eventually not to be broken
Availability

It is possible to both read **and update** the state of any microservice at any time
Backup

While backing up the system, is it possible to take a consistent snapshot of all microservices without affecting their availability?

No.
Backup + Availability

Backing up each microservice independently will eventually lead to inconsistency after recovering from backups taken at different times
Backup + Consistency

Taking a consistent backup requires to:

- disallow updates anywhere during the backup (limited availability)
- wait for the slowest microservice to complete the backup
- agree among all microservices on when to perform the backup (limited autonomy)
Shared Database

A centralized, shared database would require only one backup

Is this still a microservice architecture?
Shared Database, Split Schema

A centralized, shared database would require only one backup

Each microservice must use a logically separate schema

What happened to polyglot persistence?
Links can break

No guarantees for references crossing microservice boundaries

Microservices inherit a fundamental property of the Web
Orphan State

Orphan state is no longer referenced after recovery
Unstoppable System

An expensive, replicated database with high-availability for every microservice
Unstoppable System

How do you restart an unstoppable system?
Disaster Strikes

Eventual Consistency

Consistency

Recovery

Eventual Inconsistency
Eventual Consistency

Retries are **enough** to deal with **temporary** failures of read operations, eventually the missing data will be found.
Eventual Consistency

Retries are **enough** to deal with **temporary** failures of read operations, eventually the missing data will be found.

Eventual Inconsistency

Retries are **useless** to deal with **permanent** failures of read operations, which used to work just fine before disaster recovery.
Backup

Eventual Consistency

Consistency

Disaster Strikes

Recovery

E

Inco
Distributed Transactions

Take snapshots only when all microservices are consistent

Avoid eventual consistency
Microservices

Distributed transactions are notoriously difficult to implement and as a consequence microservice architectures emphasize transactionless coordination between services, with explicit recognition that consistency may only be eventual consistency and problems are dealt with by compensating operations.

M. Fowler, J. Lewis  https://www.martinfowler.com/articles/microservices.html
Splitting the Monolith

Keep data together for microservices that cannot tolerate eventual inconsistency
Does it apply to you?

- More than one stateful microservice
- Polyglot persistence
- Eventual Consistency
- (Cross-microservice references)
- Disaster recovery based on backup/restore

- **Independent** backups

⇒ Eventual inconsistency (after disaster recovery)
Does it apply to you?

- More than one stateful microservice
- Polyglot persistence
- Eventual Consistency
- (Cross-microservice references)
- Disaster recovery based on backup/restore
- **Synchronized** backups (limited availability/autonomy)

⇒ Consistent Disaster Recovery
The BAC Theorem

- Not Backed Up
- Not Available for updates
- Not Consistent
No Backup

Customer

Order

1. new C/1
2. C/1/name
3. new C/2
4. C/2/name

1. new C/1
2. C/1/name
3. new C/2
4. C/2/name

1. new O/1
2. O/1 → C/1
3. new O/2
4. O/2 → C/2
5. new O/3
6. O/3 → C/3

1. new O/1
2. O/1 → C/1
3. new O/2
4. O/2 → C/2
5. new O/3
6. O/3 → C/3
Trim to the oldest backup

Loose even more data!
The BAC Theorem

When Backing up a whole microservice architecture, it is not possible to have both Consistency and Availability

Corollaries

1. Microservice architectures eventually become inconsistent after disaster strikes when recovering from independent backups

2. Achieving consistent backups can be attempted by limiting the full availability/autonomy of the microservices and synchronizing their backups
Dealing with the Consequences of BAC

1. Eventual Consistency breeds Eventual Inconsistency
2. Trade off: Cost of Recovery vs. Prevention
3. Cluster microservices to be backed up together
Consistent Disaster Recovery for Microservices: the BAC Theorem

How do you back up a microservice? You dump its database. But how do you back up an entire application decomposed into microservices? In this article, we discuss the tradeoff between the availability and consistency of a microservice-based architecture when a backup of the entire application is being performed. We demonstrate that service designers must select two out of three qualities: backup, availability, and/or consistency (BAC). Service designers must also consider how to deal with consequences such as broken links, orphan state, and missing state.

Microservices are about the design of fine-grained services, which can be developed and operated by independent teams, ensuring that an architecture can organically grow and rapidly evolve. By definition, each microservice is independently deployable and scalable; each stateful one relies on its own persistent storage mechanism. Integration at the database layer is not recommended, because it introduces coupling between the data representation internally used by multiple microservices. Instead, microservices should interact only through well-defined APIs, which—following the REST architectural style—provide a clear mechanism for managing the state of the resources exposed by each microservice. Relationships between related entities are represented using hypermedia, so that representations retrieved from one microservice API can include links to other entities found on other microservice APIs. While there is no guarantee that a link retrieved from one microservice will point to a valid URL served by another, a basic notion of consistency can be introduced for the microservice-based application, requiring that such references can always be resolved, thus avoiding broken links. As the scale of the system grows, such a guarantee can be gradually weakened, as is currently the case for the World Wide Web.
References


Jim Webber, Savas Parastatidis, Ian Robinson, REST in Practice: Hypermedia and Systems Architecture, O'Reilly, 2010

Cesare Pautasso, Erik Wilde, Push-Enabling RESTful Business Processes, 9th International Conference on Service Oriented Computing (ICSOC 2011), Paphos, Cyprus, December 2011


Made with

ASQ

http://asq.inf.usi.ch
Acknowledgements

Guy Pardon, Olaf Zimmermann, Florian Haupt, Silvia Schreier, Ana Ivanchikj, Mathias Weske, Adriatik Nikaj, Sankalita Mandal, Hagen Overdick, Jesus Bellido, Rosa Alarcón, Alessio Gambi, Daniele Bonetta, Achille Peternier, Erik Wilde, Mike Amundsen, Stefan Tilkov, James Lewis