How do you back up and consistently recover your microservice architecture?

Cesare Pautasso
Software Institute, USI, Lugano, Switzerland

http://www.pautasso.info/talks/2018/apidays

@pautasso@scholar.social
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
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](https://www.martinfowler.com/articles/microservices.html)
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](https://www.martinfowler.com/articles/microservices.html)
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.
Devops meets Disaster Recovery
Devops meets Disaster Recovery
How do you back up a monolith?
How do you back up a monolith?
How do you back up one microservice?
How do you back up one microservice?
How do you back up an entire microservice architecture?
How do you back up an entire microservice architecture?
How do you back up an entire microservice architecture?
How do you back up an entire microservice architecture?

Are you sure?
Example

Customer

Order

Product

Shipment
Example

Data relationships across microservices = Hypermedia
Independent Backup

Customer

Order

1  new C/1
2    C/1/name
3  new C/2
4    C/2/name
5  
6  

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

1  
2  
3  
4  
5  
6  
Independent Backup

- **Customer**
  - new C/1
  - C/1/name
  - new C/2
  - C/2/name

- **Order**
  - new O/1
  - O/1 → C/1
  - new O/2
  - O/2 → C/2
Independent Backup

<table>
<thead>
<tr>
<th></th>
<th>new C/1</th>
<th></th>
<th>new O/1</th>
<th></th>
<th>new O/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C/1/name</td>
<td>2</td>
<td>O/1 → C/1</td>
<td>3</td>
<td>new O/2</td>
</tr>
<tr>
<td>2</td>
<td>new C/2</td>
<td>3</td>
<td>new O/2</td>
<td>4</td>
<td>O/2 → C/2</td>
</tr>
<tr>
<td>3</td>
<td>new C/3</td>
<td>4</td>
<td>new O/3</td>
<td>5</td>
<td>O/3 → C/3</td>
</tr>
<tr>
<td>4</td>
<td>C/2/name</td>
<td>5</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Independent Backup

Customer

Order

1. new C/1
   2. C/1/name
   3. new C/2
   4. C/2/name
   5. new C/3
   6. C/3/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
Independent Backup

Backups taken independently at different times
Disaster Strikes
Disaster Strikes

One microservice is lost
Recovery from Backup

Customer

Order

new C/1
C/1/name
new C/2
C/2/name

new O/1
O/1 → C/1
new O/2
O/2 → C/2
new O/3
O/3 → C/3

new O/1
O/1 → C/1
new O/2
O/2 → C/2
new O/3
O/3 → C/3
Recovery from Backup

<table>
<thead>
<tr>
<th></th>
<th>new C/1</th>
<th></th>
<th>new C/1</th>
<th></th>
<th>new O/1</th>
<th></th>
<th>new O/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C/1/name</td>
<td>1</td>
<td>C/1/name</td>
<td>1</td>
<td>O/1 → C/1</td>
<td>1</td>
<td>O/1 → C/1</td>
</tr>
<tr>
<td>2</td>
<td>new C/2</td>
<td>2</td>
<td>new C/2</td>
<td>2</td>
<td>new O/2</td>
<td>2</td>
<td>new O/2</td>
</tr>
<tr>
<td>3</td>
<td>C/2/name</td>
<td>3</td>
<td>C/2/name</td>
<td>3</td>
<td>O/2 → C/2</td>
<td>3</td>
<td>O/2 → C/2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>4</td>
<td></td>
<td>4</td>
<td>new O/3</td>
<td>4</td>
<td>new O/3</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>5</td>
<td></td>
<td>5</td>
<td>O/3 → C/3</td>
<td>5</td>
<td>O/3 → C/3</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>6</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recovery from Backup

Customer

Order

Broken link after recovery
Recovery from Backup

Broken link after recovery

Eventual Inconsistency
Synchronized Backups

Customer

Order

new C/1
C/1/name
new C/2
C/2/name

new C/1
C/1/name
new C/2
C/2/name

new O/1
O/1 → C/1
new O/2
O/2 → C/2
Synchronized Backups

Backups of all microservices taken at the same time.
Synchronized Backups

Backups of all microservices taken at the same time.

Limited Autonomy
The BAC theorem
The BAC theorem

When Backing up a microservice architecture, it is not possible to have both Consistency and Autonomy
The BAC theorem

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

Each microservices has an independent DevOps lifecycle

Backup autonomy: snapshots taken at different times without any coordination across multiple microservices
Backup

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

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

No.
Backup + Autonomy

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:

• agree among all microservices on when to perform the backup (limited autonomy)
Backup + Consistency

Taking a consistent backup requires to:

- agree among all microservices on when to perform the backup (limited autonomy)
- disallow updates anywhere during the backup (limited availability)
Backup + Consistency

Taking a consistent backup requires to:

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

A centralized, shared database would require only one backup
Shared Database

Customer  Order  Is this still a microservice architecture?
- Yes
- No

Product  Shipment

A centralized, shared database would require only one backup
Shared Database, Split Schema

A centralized, shared database would require only one backup

Each microservice must use a logically separate schema
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
Links can break

No guarantees for references crossing microservice boundaries

Microservices inherit a fundamental property of the Web
Orphan State

Customer

Order

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

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

Customer

Order

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

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

Orphan state is no longer referenced after recovery
Synchronous Replication

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

How do you restart an unstoppable system?
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.
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.
Distributed Transactions

Customer

Order

new C/1
C/1/name

new O/1
O/1 → C/1
Distributed Transactions

Customer

Order

1
new C/1
C/1/name
new C/2
C/2/name

1
new O/1
O/1 → C/1
new O/2
O/2 → C/2

1
Distributed Transactions

Customer

Order

1. new C/1
2. C/1/name
3. new C/2
4. C/2/name
5. 
6. 

1. new C/1
2. C/1/name
3. new C/2
4. C/2/name
5. 
6. 

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

1. 
2. 
3. 
4. 
5. 
6. 
Distributed Transactions
Distributed Transactions

Take snapshots only when all microservices are consistent
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
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
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
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 autonomy)

⇒ **Consistent Disaster Recovery**
The BAC Theorem

- Not Backed Up
- Not Autonomous
- Not Consistent
No Backup

- New C/1
- C/1/name
- New C/2
- C/2/name

- New O/1
  - O/1 \rightarrow C/1
- New O/2
  - O/2 \rightarrow C/2
- New O/3
  - O/3 \rightarrow C/3
**No Backup**

### Customer
- new C/1
- C/1/name
- new C/2
- C/2/name

### Order
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

---

**Note:** The diagram illustrates the flow of data from the Customer to the Order, with new entries being added to both tables and the relationship between orders and customers being shown.
No Backup

Customer

Order

<table>
<thead>
<tr>
<th>1</th>
<th>new C/1</th>
<th>2</th>
<th>new C/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>C/1/name</td>
<td>3</td>
<td>O/1 → C/1</td>
</tr>
<tr>
<td>3</td>
<td>new C/2</td>
<td>4</td>
<td>new O/2</td>
</tr>
<tr>
<td>4</td>
<td>C/2/name</td>
<td>5</td>
<td>O/2 → C/2</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>6</td>
<td>new O/3</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>O/3 → C/3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>new O/1</th>
<th>2</th>
<th>new O/1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>O/1 → C/1</td>
<td>3</td>
<td>new O/2</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>4</td>
<td>O/2 → C/2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>5</td>
<td>new O/3</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>6</td>
<td>O/3 → C/3</td>
</tr>
</tbody>
</table>
No Backup

Customer

Order

1
new C/1
C/1/name
new C/2
C/2/name

1
new C/1
C/1/name
new C/2
C/2/name

1
new O/1
O/1 → C/1
new O/2
O/2 → C/2

1
new O/1
O/1 → C/1
new O/2
O/2 → C/2
new O/3
O/3 → C/3
No Backup

Trim to the oldest backup
No Backup

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
Guy Pardon, Cesare Pautasso, Olaf Zimmermann, **Consistent Disaster Recovery for Microservices: the BAC Theorem**, IEEE Cloud Computing, 5(1):49-59, January/February 2018

http://design.inf.usi.ch/bac
References


