Is it possible to consistently recover a microservice architecture?

Cesare Pautasso
Software Institute, USI, Lugano, Switzerland

http://www.pautasso.info/talks/2019/lamsade

@pautasso@scholar.social
Software Institute

- Opened July 2017
- 6 Professors
- 30+ PhD/Postdoc researchers
- New Master Software and Data Engineering

http://www.si.usi.ch/
Architecture, Design and Web Information Systems Engineering

- RESTalk - API Conversation Modeling
- Liquid Software Architecture
- BenchFlow - a benchmark for workflow engines
- ASQ - Interactive Web Lectures, Classroom Analytics
- Parallel JavaScript/Multicore Node.JS
- SAW - Collaborative Architectural Decision Making
- NaturalMash - API Composition with Natural Language

http://design.inf.usi.ch
Decomposition

Monolith

microservices
Independent DevOps Lifecycle

Monolith

microservices
Autonomous Microservices
Autonomous Microservices

**Rapid Evolution**: If you have to hold a release until some other team is ready you do not have two separate microservices

**Avoid Cascading Failures**: A failed microservice should not bring down the whole system
Isolated Microservices

Customer  Order

Monolith
Isolated Microservices

Monolith

Customer

Order

Microservices

Customer

Order
Isolated Microservices

Monolith

Microservices
Isolated 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
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
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?

MySQL

MongoDB

Redis

Neo4J
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

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

Order

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

Customer

Order

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

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

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

Customer

Order

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>new C/1</td>
<td>new C/1</td>
<td>new 0/1</td>
</tr>
<tr>
<td>2</td>
<td>C/1/name</td>
<td>C/1/name</td>
<td>O/1 → C/1</td>
</tr>
<tr>
<td>3</td>
<td>new C/2</td>
<td>new O/2</td>
<td>new 0/2</td>
</tr>
<tr>
<td>4</td>
<td>C/2/name</td>
<td>O/2 → C/2</td>
<td>new 0/3</td>
</tr>
<tr>
<td>5</td>
<td>new C/3</td>
<td>new O/3</td>
<td>O/3 → C/3</td>
</tr>
<tr>
<td>6</td>
<td>C/3/name</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Independent Backup

**Customer**
- new C/1
- C/1/name
- new C/2
- C/2/name
- new C/3
- C/3/name

**Order**
- new O/1
- O/1 → C/1
- new O/2
- O/2 → C/2
- new O/3
- O/3 → C/3

Diagram showing relationships between customer and order with new entries.
Independent Backup

Backups taken independently at different times
Disaster Strikes

Customer

Order

new C/1
C/1/name
new C/2
C/2/name
new C/3
C/3/name

new 0/1
0/1 → C/1
new 0/2
O/2 → C/2
new 0/3
O/3 → C/3

new 0/1
0/1 → C/1
new 0/2
O/2 → C/2
new 0/3
O/3 → C/3
Disaster Strikes

One microservice is lost
Recovery from Backup

Customer

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

Order

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

1. new C/1
2. C/1/name
3. new C/2
4. C/2/name
5. new O/1
6. O/1 → C/1
7. new O/2
8. O/2 → C/2
9. new O/3
10. O/3 → C/3
Recovery from Backup

Customer

Order

1. new C/1
   C/1/name
2. new C/2
   C/2/name
3. new C/1
   C/1/name
4. new C/2
   C/2/name
5. new O/1
   O/1 → C/1
6. new O/2
   O/2 → C/2
7. new O/3
   O/3 → C/3
8. new O/1
   O/1 → C/1
9. new O/2
10. O/2 → C/2
11. new O/3
12. O/3 → C/3
Recovery from Backup

Broken link after recovery
Recovery from Backup

Customer

Order

Broken link after recovery

Eventual Inconsistency
Synchronized Backups

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)
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 **B**acking up a microservice architecture, it is not possible to have both **C**onsistency and **A**utonomy
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

Is this still a microservice architecture?
- Yes
- No

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

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

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

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

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

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 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

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
Orphan State

- **Customer**
  - new C/1
  - C/1/name
  - new C/2
  - C/2/name
  - new C/3
  - C/3/name

- **Order**
  - new O/1
  - O/1 → C/1
  - new O/2
  - O/2 → C/2
  - new O/1
  - O/1 → C/1
  - new O/2
  - O/2 → C/2
Orphan State

Customer

Order

1
new C/1
C/1/name
new C/2
C/2/name
new C/3
C/3/name

1
new C/1
C/1/name
new C/2
C/2/name
new C/3
C/3/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
Orphan State

Orphan state is no longer referenced after recovery
Synchronous Replication

An expensive, replicated database with high-availability for every microservice
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 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
2. new C/2
   C/2/name
3. new O/1
   O/1 → C/1
4. new O/2
   O/2 → C/2
Distributed Transactions
## Distributed Transactions

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

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

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

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

Take snapshots only when all microservices are consistent.
Distributed Transactions

Take snapshots only when all microservices are consistent

Backups taken as part of the distributed transaction
Distributed Transactions

Take snapshots only when all microservices are consistent
Backups taken as part of the distributed transaction
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

Consistency

Not Autonomous

Autonomy

Backup

Not Consistent
The BAC Theorem

Consistency

Not Autonomous

Autonomy

Backup

Not Consistent
The BAC Theorem

Consistency

Not Backed Up

Autonomy

Backup

Not Consistent

Not Autonomous
No 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  new O/3
6  O/3 → C/3
### No Backup

The diagram illustrates a system where there is no backup mechanism in place. The interactions between the `Customer` and `Order` databases are shown through transactions.

#### Customer Database:
- **Line 1:** `new C/1`  
- **Line 2:** `C/1/name`  
- **Line 3:** `new C/2`  
- **Line 4:** `C/2/name`  
- **Line 5:**  
- **Line 6:**  

#### Order Database:
- **Line 1:** `new O/1`  
- **Line 2:** `O/1 -> C/1`  
- **Line 3:** `new O/2`  
- **Line 4:** `O/2 -> C/2`  
- **Line 5:** `new O/3`  
- **Line 6:** `O/3 -> C/3`  

The transitions indicate the flow of data and operations between the two databases.
### No Backup

**Diagram: No Backup**

- **Customer**
  - 1: new C/1
  - 2: C/1/ name
  - 3: new C/2
  - 4: 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

---

**Table: No 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>
</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></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></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></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>4</td>
<td></td>
<td>4</td>
<td>new O/3</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>5</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>6</td>
<td></td>
<td>6</td>
<td>O/3 → C/3</td>
<td></td>
</tr>
</tbody>
</table>
No Backup

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
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
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
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 have to 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 polyglot 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 implemented 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


