Scalable Web Systems

Splitting the Monolith Part 2

Overview

1. Transactional Boundaries
2. Reporting
3. Data Retrieval
4. Data Pumps
5. Cost of Change

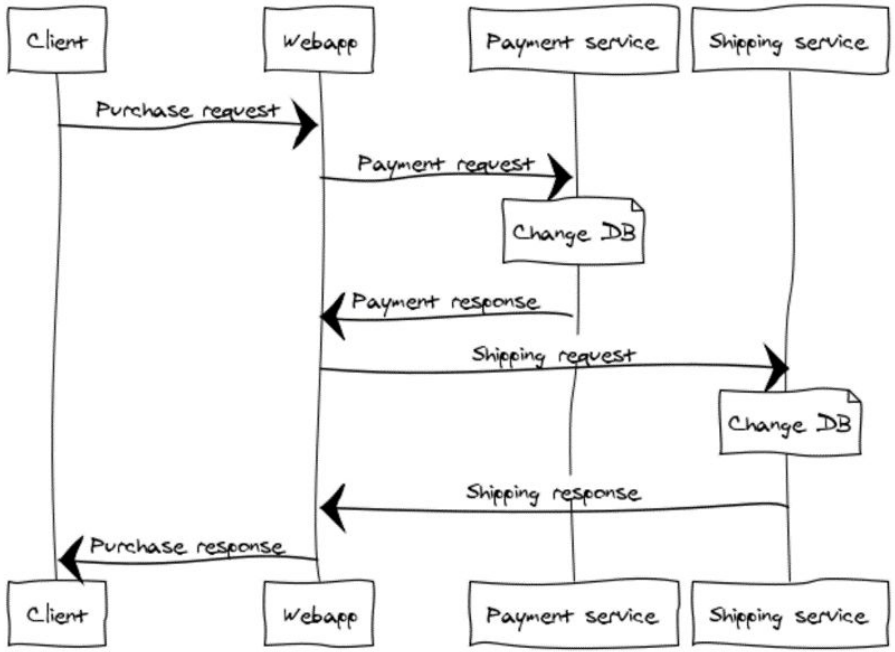
Transactional Boundaries

What is a transaction?

A transaction is an event or set of events that either all complete or none complete.

The concept is often used in the database domain, however, a transaction can occur in other areas as well.

In this discussion, it is about events that must all complete or none complete across microservice boundaries.

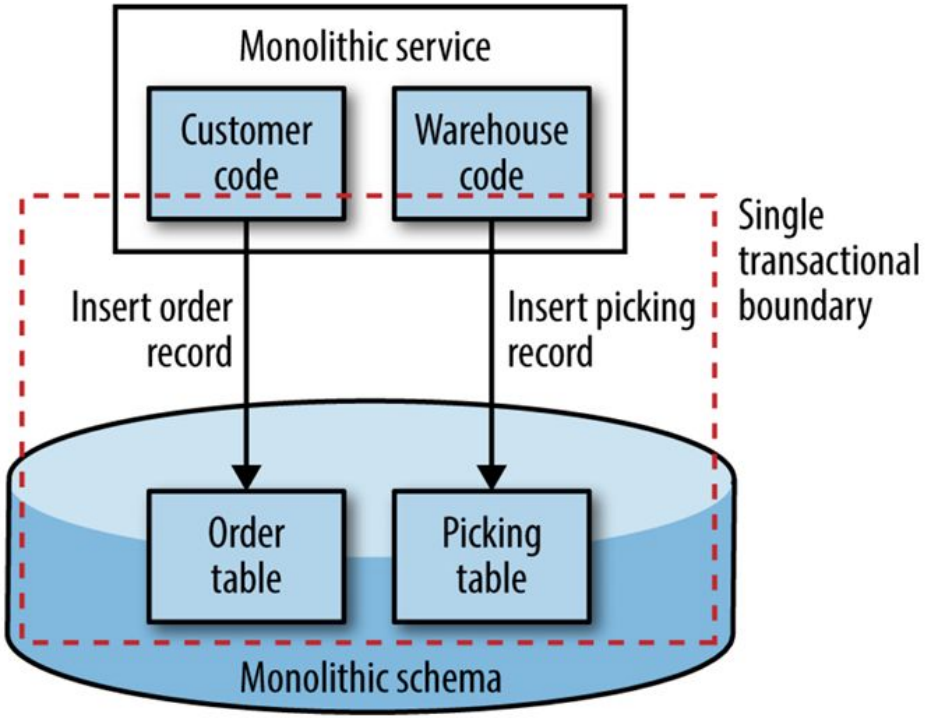


Single Transactional Boundary

Within a *monolithic service* transactions are relatively easy to handle in the case where we have a single database with a shared schema.

Although the code may be split across packages we arrange for calls to the database to occur in a single transaction.

In this example, we insert an order record and a picking record within a *single transactional boundary*.



Separate Transactional Boundaries

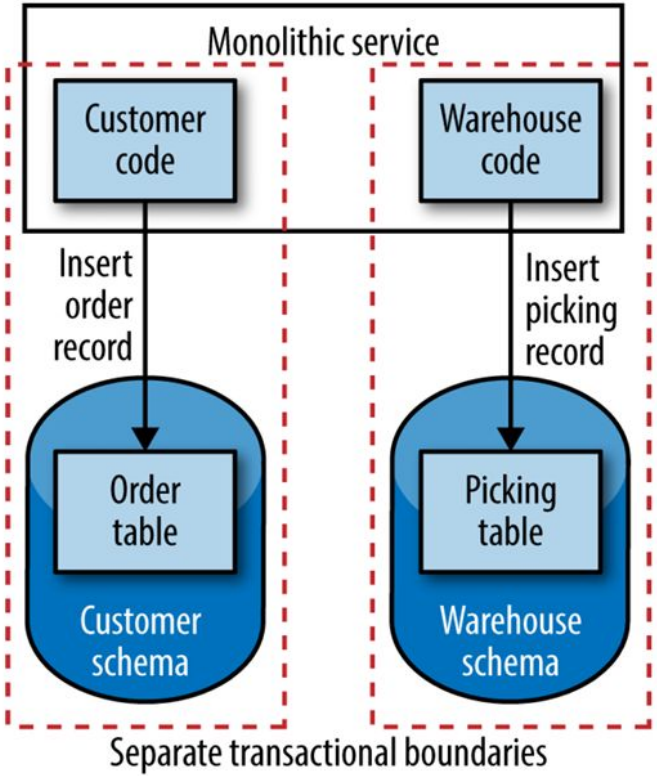
If we have migrated the monolith to use separate databases where the **Order** table and the **Picking** table reside in different databases, we now have *separate transactional boundaries*.

In this example, the ordering process now spans two transactional boundaries.

If our insert to the **Order** table fails we can clearly stop everything.

If our insert to the **Order** table succeeds and then an insert to the **Picking** table fails

What do we do?



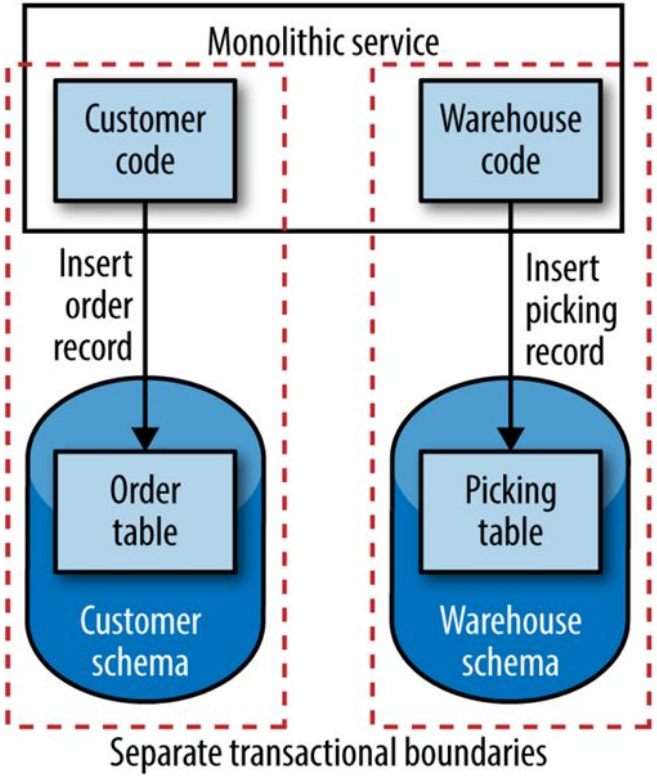
Separate Transactional Boundaries

What do we do?

Try Again Later: we could try the operation again later hoping it will succeed. Hoping that the state of the system will *eventually* be consistent.

Abort The Entire Operation: we could reject the entire operation. To do this we need a *compensating transaction*.

Compensating Transaction: an operation to undo any prior operations to ensure the system and its data are in a consistent state.

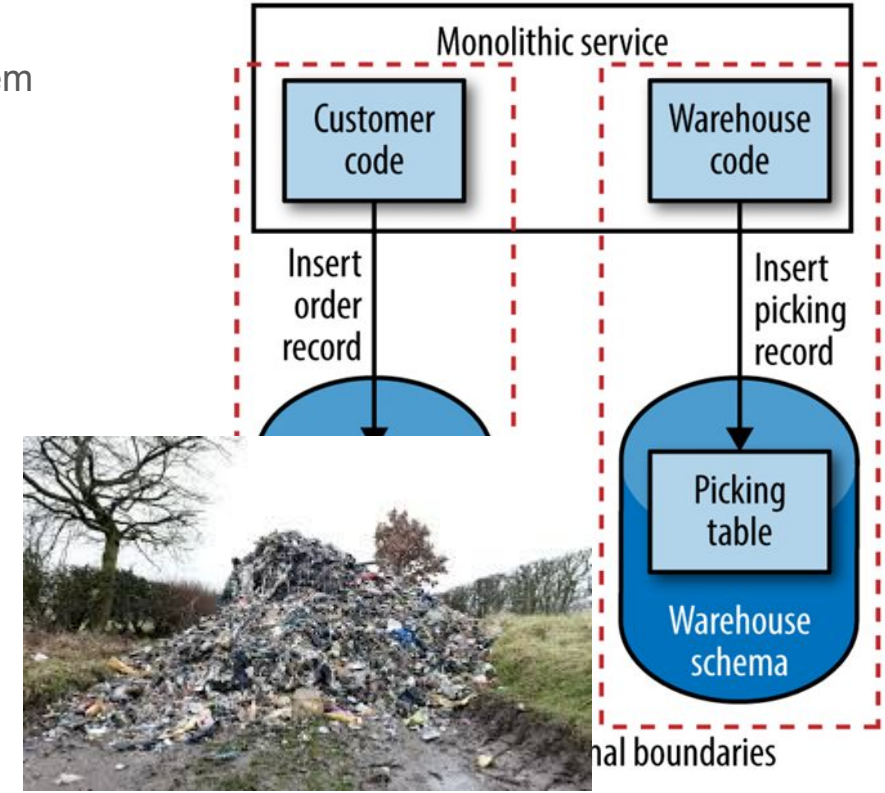


Separate Transactional Boundaries

Compensating Transaction: an operation to undo any prior operations to ensure the system and its data are in a consistent state.

What if the compensating transaction fails?

We need to have another process that runs periodically to clean up the mess.



Distributed Transactions

Distributed Transaction: distributed transactions try to span multiple transactions within them, using an overall governing process called a **transaction manager** to orchestrate the various transactions being done by underlying systems.

A common algorithm used in practice is referred to as the 2-phase commit.

The transaction manager coordinates amongst several services using a voting mechanism. Each service votes (Yes/No) if they think it is safe to proceed.

Any No vote causes a roll back of all actions.


```
Transaction Manager
  send to all: send ok to commit?
  collect replies
    all ok => send commit
    else   => send abort
Participant
  receive ok to commit =>
    save to temp area, reply ok
  receive commit =>
    make change permanent
  receive abort =>
    delete temp area
```

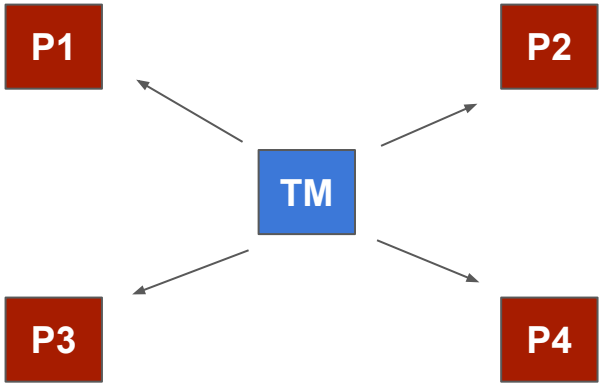
Distributed Transactions

Distributed Transaction: distributed transactions try to span multiple transactions within them, using an overall governing process called a **transaction manager** to orchestrate the various transactions being done by underlying systems.

A common algorithm used in practice is referred to as the 2-phase commit.

The transaction manager coordinates amongst several services using a voting mechanism. Each service votes (Yes/No) if they think it is safe to proceed.

Any No vote causes a roll back of all actions.




```
Transaction Manager
  send to all: send ok to commit?
  collect replies
    all ok => send commit
    else   => send abort
Participant
  receive ok to commit =>
    save to temp area, reply ok
  receive commit =>
    make change permanent
  receive abort =>
    delete temp area
```

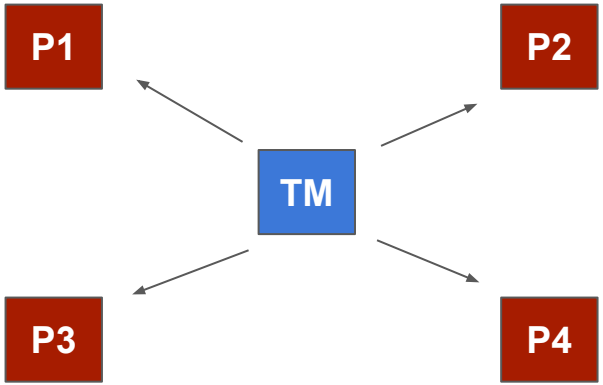
Distributed Transactions

Distributed Transaction: distributed transactions try to span multiple transactions within them, using an overall governing process called a **transaction manager** to orchestrate the various transactions being done by underlying systems.

A common algorithm used in practice is referred to as the 2-phase commit.

The transaction manager coordinates amongst several services using a voting mechanism. Each service votes (Yes/No) if they think it is safe to proceed.

Any No vote causes a roll back of all actions.




```
Transaction Manager
  send to all: send ok to commit?
  collect replies
    all ok => send commit
    else   => send abort
Participant
  receive ok to commit =>
    save to temp area, reply ok
  receive commit =>
    make change permanent
  receive abort =>
    delete temp area
```

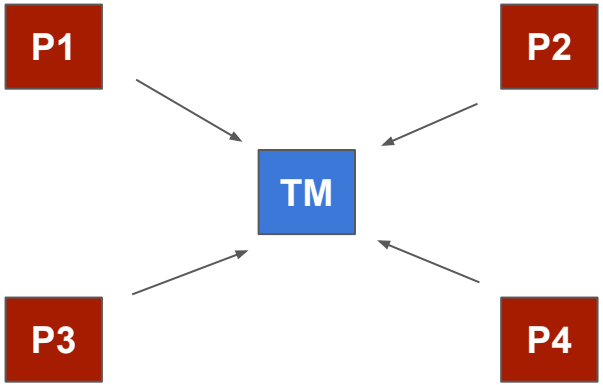
Distributed Transactions

Distributed Transaction: distributed transactions try to span multiple transactions within them, using an overall governing process called a **transaction manager** to orchestrate the various transactions being done by underlying systems.

A common algorithm used in practice is referred to as the 2-phase commit.

The transaction manager coordinates amongst several services using a voting mechanism. Each service votes (Yes/No) if they think it is safe to proceed.

Any No vote causes a roll back of all actions.




```
Transaction Manager
  send to all: send ok to commit?
  collect replies
    all ok => send commit
    else   => send abort
Participant
  receive ok to commit =>
    save to temp area, reply ok
  receive commit =>
    make change permanent
  receive abort =>
    delete temp area
```

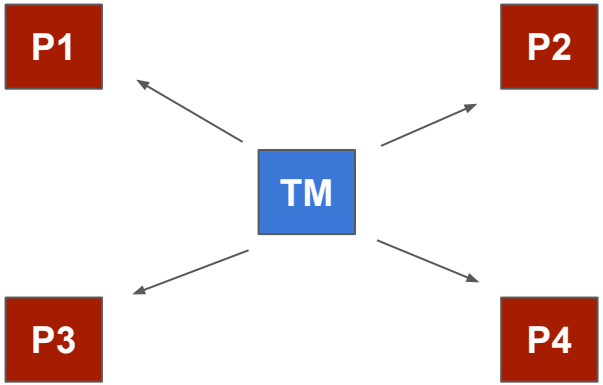
Distributed Transactions

Distributed Transaction: distributed transactions try to span multiple transactions within them, using an overall governing process called a **transaction manager** to orchestrate the various transactions being done by underlying systems.

A common algorithm used in practice is referred to as the 2-phase commit.

The transaction manager coordinates amongst several services using a voting mechanism. Each service votes (Yes/No) if they think it is safe to proceed.

Any No vote causes a roll back of all actions.




```
Transaction Manager
  send to all: send ok to commit?
  collect replies
    all ok => send commit
    else   => send abort
Participant
  receive ok to commit =>
    save to temp area, reply ok
  receive commit =>
    make change permanent
  receive abort =>
    delete temp area
```

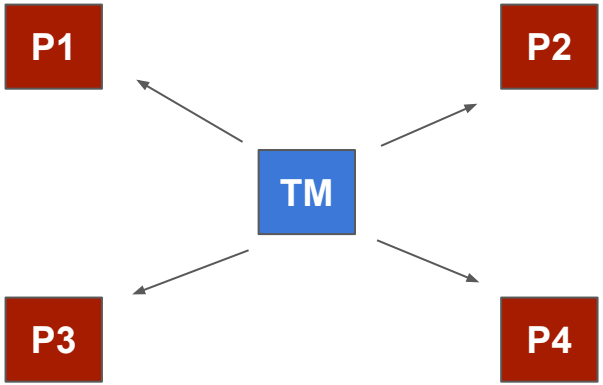
Distributed Transactions

Distributed Transaction: distributed transactions try to span multiple transactions within them, using an overall governing process called a **transaction manager** to orchestrate the various transactions being done by underlying systems.

A common algorithm used in practice is referred to as the 2-phase commit.

The transaction manager coordinates amongst several services using a voting mechanism. Each service votes (Yes/No) if they think it is safe to proceed.

Any No vote causes a roll back of all actions.




```
Transaction Manager
  send to all: send ok to commit?
  collect replies
    all ok => send commit
    else   => send abort
Participant
  receive ok to commit =>
    save to temp area, reply ok
  receive commit =>
    make change permanent
  receive abort =>
    delete temp area
```

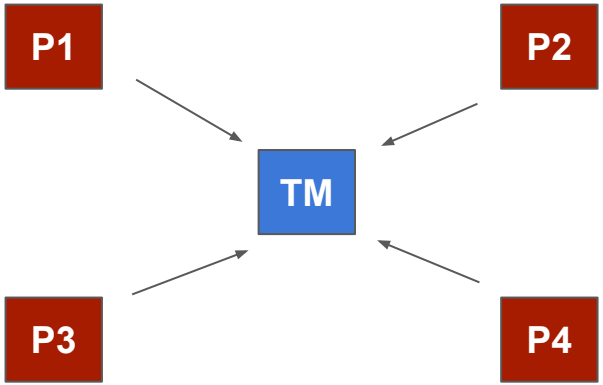
Distributed Transactions

Distributed Transaction: distributed transactions try to span multiple transactions within them, using an overall governing process called a **transaction manager** to orchestrate the various transactions being done by underlying systems.

A common algorithm used in practice is referred to as the 2-phase commit.

The transaction manager coordinates amongst several services using a voting mechanism. Each service votes (Yes/No) if they think it is safe to proceed.

Any No vote causes a roll back of all actions.



What if a participant is not ok?

```
Transaction Manager
  send to all: send ok to commit?
  collect replies
    all ok => send commit
    else   => send abort
Participant
  receive ok to commit =>
    save to temp area, reply !ok
  receive commit =>
    make change permanent
  receive abort =>
    delete temp area
```

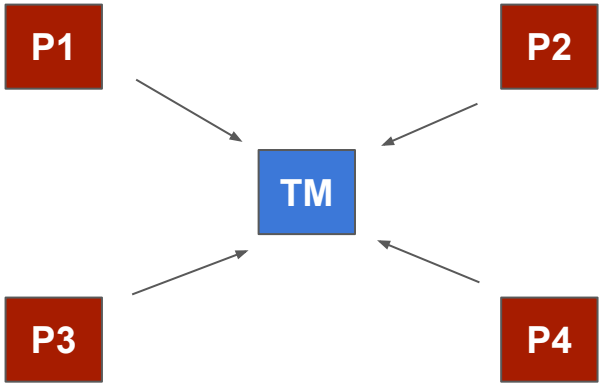
Distributed Transactions

Distributed Transaction: distributed transactions try to span multiple transactions within them, using an overall governing process called a **transaction manager** to orchestrate the various transactions being done by underlying systems.

A common algorithm used in practice is referred to as the 2-phase commit.

The transaction manager coordinates amongst several services using a voting mechanism. Each service votes (Yes/No) if they think it is safe to proceed.

Any No vote causes a roll back of all actions.



What if a participant is not ok?

```
Transaction Manager
  send to all: send ok to commit?
  collect replies
    all ok => send commit
    else   => send abort
Participant
  receive ok to commit =>
    save to temp area, reply !ok
  receive commit =>
    make change permanent
  receive abort =>
    delete temp area
```

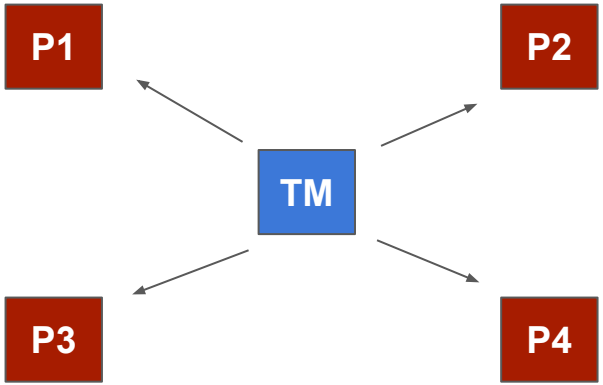
Distributed Transactions

Distributed Transaction: distributed transactions try to span multiple transactions within them, using an overall governing process called a **transaction manager** to orchestrate the various transactions being done by underlying systems.

A common algorithm used in practice is referred to as the 2-phase commit.

The transaction manager coordinates amongst several services using a voting mechanism. Each service votes (Yes/No) if they think it is safe to proceed.

Any No vote causes a roll back of all actions.



What if a participant is not ok?

```
Transaction Manager
  send to all: send ok to commit?
  collect replies
    all ok => send commit
    else   => send abort
Participant
  receive ok to commit =>
    save to temp area, reply !ok
  receive commit =>
    make change permanent
  receive abort =>
    delete temp area
```

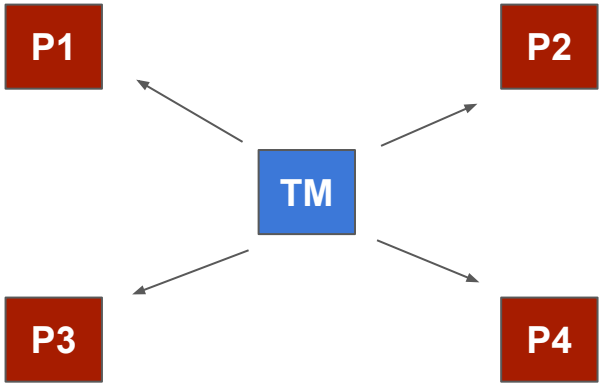
Distributed Transactions

Distributed Transaction: distributed transactions try to span multiple transactions within them, using an overall governing process called a **transaction manager** to orchestrate the various transactions being done by underlying systems.

A common algorithm used in practice is referred to as the 2-phase commit.

The transaction manager coordinates amongst several services using a voting mechanism. Each service votes (Yes/No) if they think it is safe to proceed.

Any No vote causes a roll back of all actions.



What if a participant is not ok?

```
Transaction Manager
  send to all: send ok to commit?
  collect replies
    all ok => send commit
    else   => send abort
Participant
  receive ok to commit =>
    save to temp area, reply !ok
  receive commit =>
    make change permanent
  receive abort =>
    delete temp area
```

Distributed Transactions

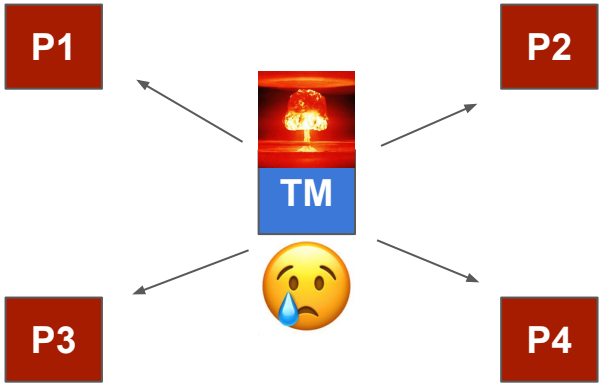
What if the transaction manager goes down?

It can happen. If your system can continue to operate with missing data - perhaps you are ok.

But, if your data must be consistent, you have problems.

Solution 1: try really hard to move data that relies on several transactions into one DB.

Solution 2: create another service that handles the entire transaction.



What if a participant is not ok?

```
Transaction Manager
  send to all: send ok to commit?
  collect replies
    all ok => send commit
    else   => send abort
Participant
  receive ok to commit =>
    save to temp area, reply !ok
  receive commit =>
    make change permanent
  receive abort =>
    delete temp area
```

Reporting

Reporting is a common operation that is performed as part of an application.

Your application may have internal reporting mechanisms. For example, Moodle allows you to generate reports on the activities of the students in a class.

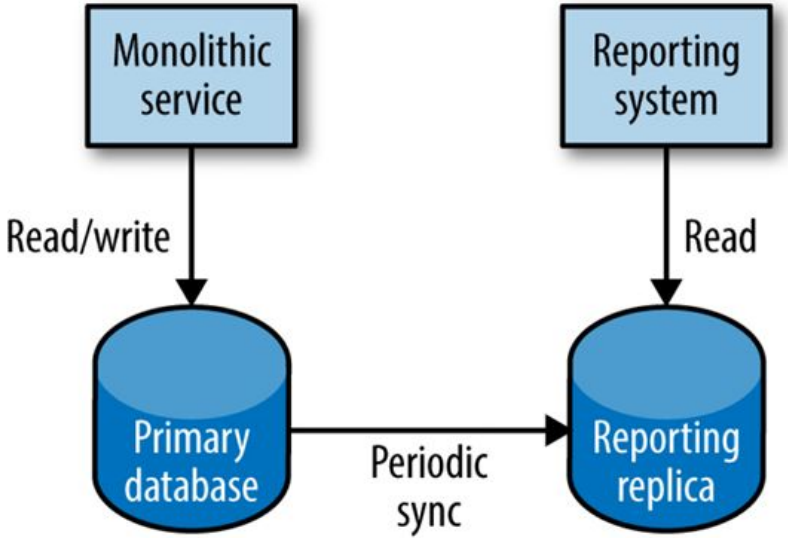
The problem with this particular use case is that the data typically cuts across the entire application and its many services.

So, what can we do about this?

The Reporting Database

In a standard, monolithic service architecture, all our data is stored in one big database. This means all the data is in one place, so reporting across all the information is actually pretty easy, as we can simply join across the data via SQL queries or the like.

Typically we won't run these reports on the main database for fear of the load generated by our queries impacting the performance of the main system, so often these reporting systems hang on a read replica



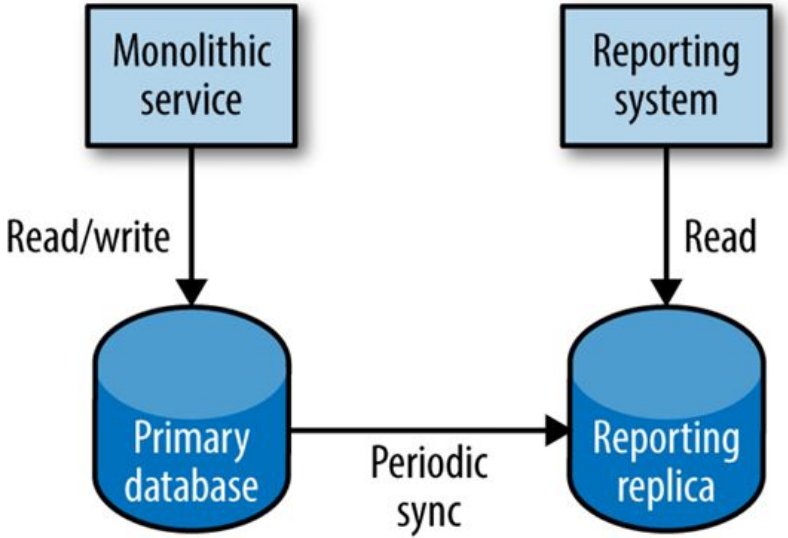
The Reporting Database

Advantages

- All the data is in one place
- Replication tools exist and easy to use

Disadvantages

- The DB is effectively the API
- Reporting system is impacted by changes
- Hard to optimize the DB as the interaction between the monolith service and the database and the reporting system and the database are typically very different
- What if the monolith wants to use a different database technology?



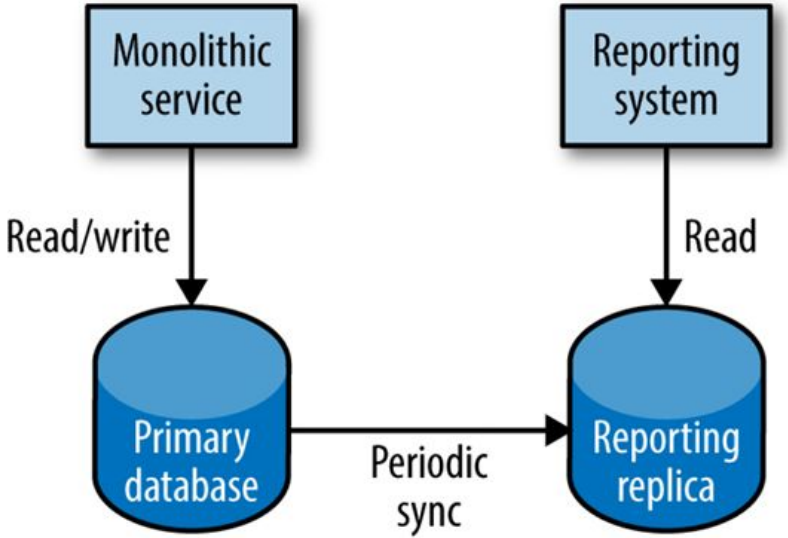
The Reporting Database

Why not simply expose the DB through service API calls?

This has its downsides as most services are not designed for reporting.

Further problems occur when the reporting system requires large amounts of data. This requires all the data to be transferred to the reporting system to do its job.

What to do?



Data Pumps

One alternative is to use **data pumps**.

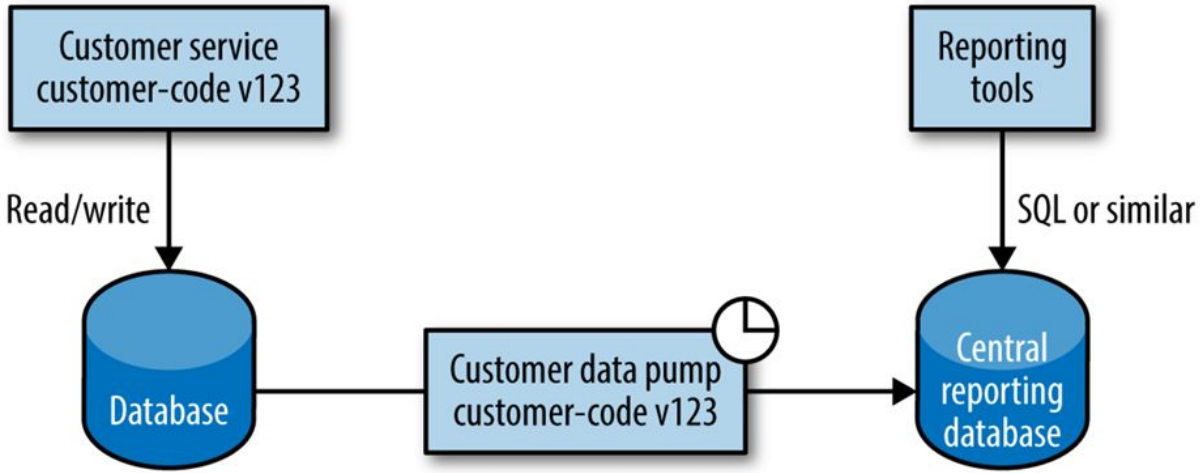
A data pump *pushes* the data to the reporting system rather than have the reporting system *pull* the data from the main database.

The service developers develop a data pump service in conjunction with the actual service.

Thus, a change to the original service or its data would be updated in both and released together.

Data Pumps

In this example, a data pump service executes periodically pushing the changed data to the reporting system.

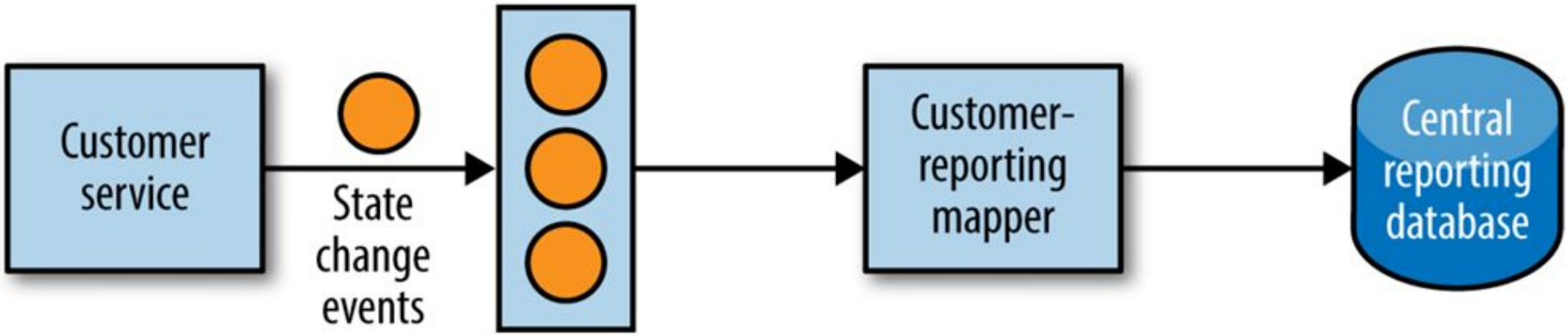


Event Data Pump

We touched on the idea of microservices emitting events based on the state change of entities that they manage.

An *event data pump* emit events when data is created, updated, deleted, etc.

These events are subscribed to by other services and receive events that are published.



Cost of Change and Design

Much of what we have been talking about in terms of scalability has to do with changes to the code and data.

It is often easier to move code around using modern IDEs. The database is harder.

In all of this it is about understanding how the design of the system interacts and how it can handle change.

One technique that is often promoted is the use of Class Responsibility Collaboration (CRC) cards.

This uses an index card for each service listing its capabilities, responsibilities, and who are its collaborators.

You then use these cards to work through use cases.