Использование акторной модели в системах финансовых транзакций

Никита Мельников

x: @nikita_melnikov
Atlantic Money

Обо мне

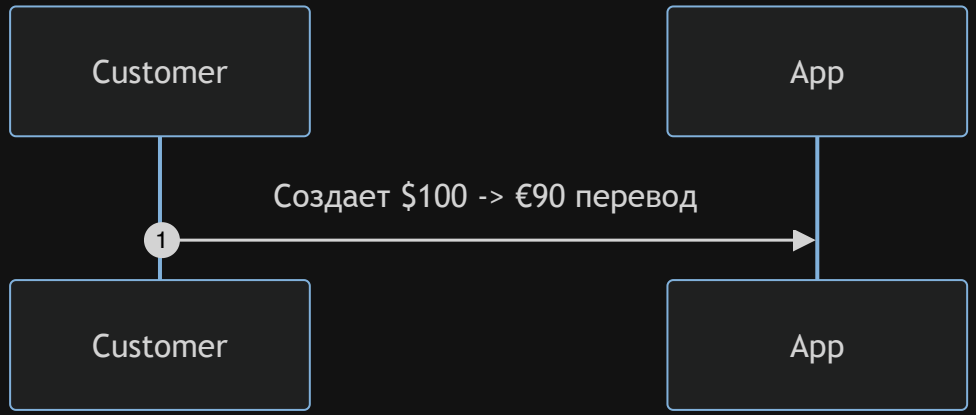
- VP of Engineering@Atlantic Money
- ex-T-Bank, ex-T-Bank Инвестиции
- 10+ лет в финтехе
- High-load
- Scala, Golang, PostgreSQL, and Kafka ❤️‍💔

Агенда

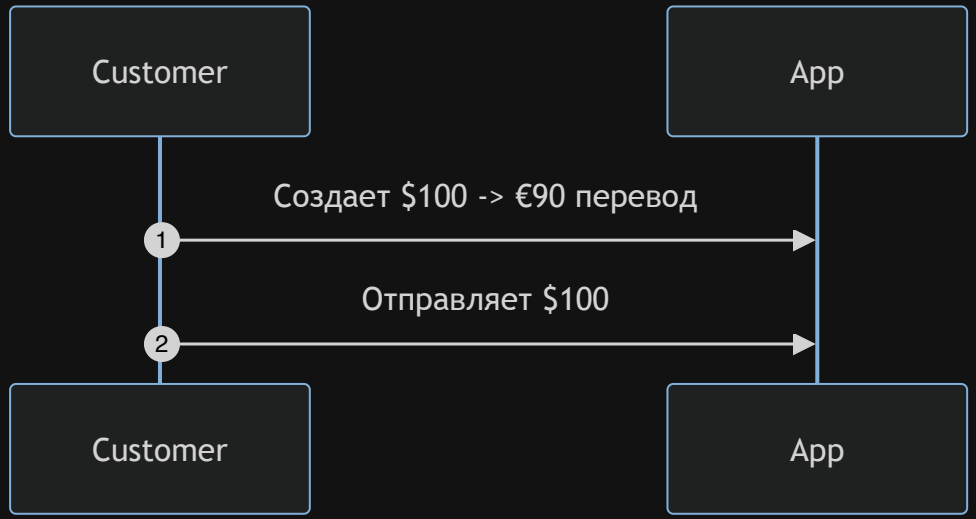
- Финансовые транзакции и типичные проблемы
- Традиционные подходы к решению проблем
- Асинхронная обработка
- Акторная модель
- Kafka
- Собираем все вместе

Что такое финансовая транзакция

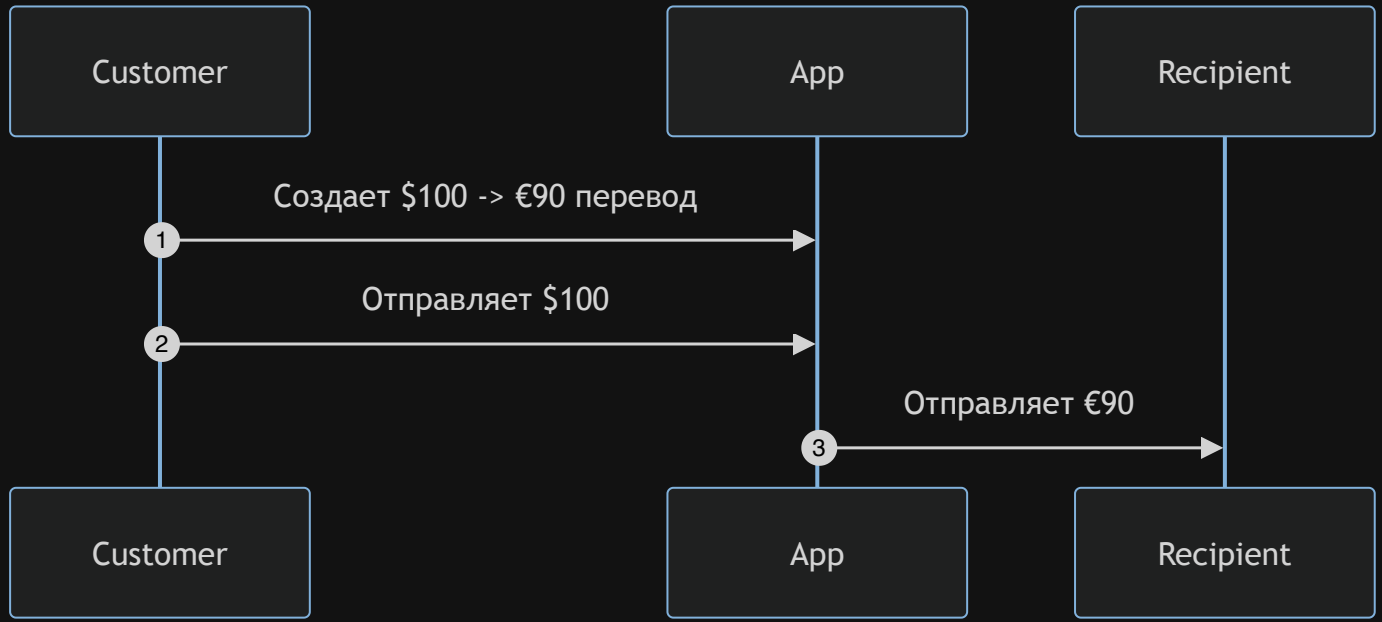
Что такое финансовая транзакция



Что такое финансовая транзакция



Что такое финансовая транзакция



Что происходит на самом деле

Что происходит на самом деле

- Пользователь создает перевод $ → €

Что происходит на самом деле

- Пользователь создает перевод $ → €
- Система ждет $

Что происходит на самом деле

- Пользователь создает перевод $ → €
- Система ждет $
- Система записывает все детали платежа

Что происходит на самом деле

- Пользователь создает перевод $ → €
- Система ждет $
- Система записывает все детали платежа
- Система запускает проверки

Что происходит на самом деле

- Пользователь создает перевод $ → €
- Система ждет $
- Система записывает все детали платежа
- Система запускает проверки
  - Проверка на санкции по различным спискам

Что происходит на самом деле

- Пользователь создает перевод $ → €
- Система ждет $
- Система записывает все детали платежа
- Система запускает проверки
  - Проверка на санкции по различным спискам
  - Анти-фрод

Что происходит на самом деле

- Пользователь создает перевод $ → €
- Система ждет $
- Система записывает все детали платежа
- Система запускает проверки
  - Проверка на санкции по различным спискам
  - Анти-фрод
  - Проверка доступных лимитов и баланса

Что происходит на самом деле

- Пользователь создает перевод $ → €
- Система ждет $
- Система записывает все детали платежа
- Система запускает проверки
  - Проверка на санкции по различным спискам
  - Анти-фрод
  - Проверка доступных лимитов и баланса
  - Подсчет комиссий

Что происходит на самом деле

- Пользователь создает перевод $ → €
- Система ждет $
- Система записывает все детали платежа
- Система запускает проверки
  - Проверка на санкции по различным спискам
  - Анти-фрод
  - Проверка доступных лимитов и баланса
  - Подсчет комиссий
  - Многое другое

Что происходит на самом деле

- Пользователь создает перевод $ → €
- Система ждет $
- Система записывает все детали платежа
- Система запускает проверки
  - Проверка на санкции по различным спискам
  - Анти-фрод
  - Проверка доступных лимитов и баланса
  - Подсчет комиссий
  - Многое другое
- Система обменивает валюту

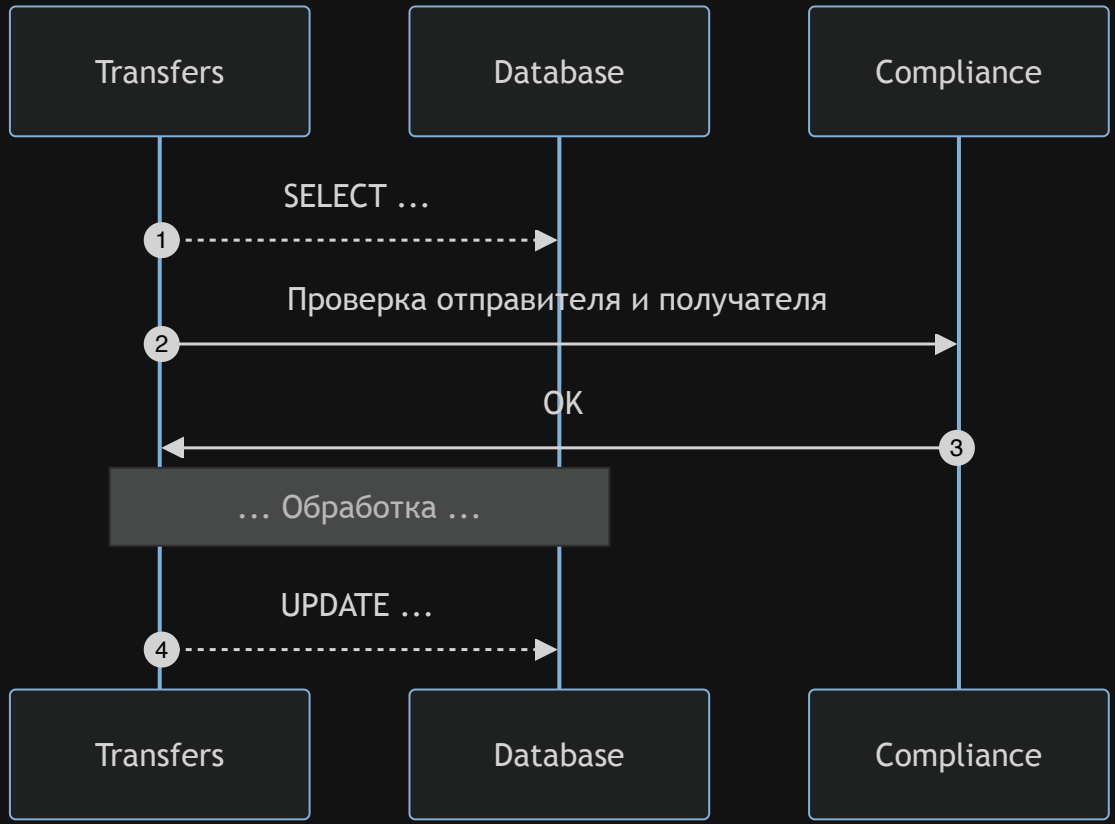
Что происходит на самом деле

- Пользователь создает перевод $ → €
- Система ждет $
- Система записывает все детали платежа
- Система запускает проверки
  - Проверка на санкции по различным спискам
  - Анти-фрод
  - Проверка доступных лимитов и баланса
  - Подсчет комиссий
  - Многое другое
- Система обменивает валюту
- Система отправляет € получателю

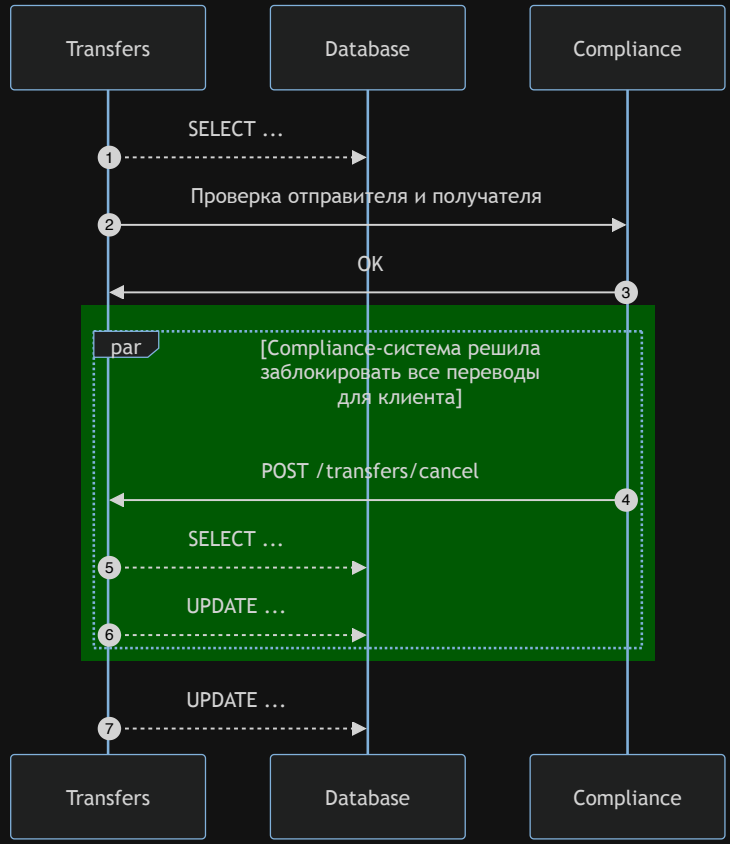
Потенциальные проблемы

Lost update

Lost update


Transfers
Database
Compliance
SELECT ...
1
Проверка отправителя и получателя
2
OK
3
... Обработка ...
UPDATE ...
4
Transfers
Database
Compliance

Lost update


Transfers
Database
Compliance
SELECT ...
1
Проверка отправителя и получателя
2
OK
3
par
[Compliance-система решила заблокировать все переводы для клиента]
POST /transfers/cancel
4
SELECT ...
5
UPDATE ...
6
UPDATE ...
7
Transfers
Database
Compliance

Lost update

```sql
BEGIN TRANSACTION;

SELECT * FROM transfers WHERE id = 1;
-- [id: 1, status: 'CREATED']

UPDATE transfers
SET status = 'CANCELLED'
WHERE id = 1;

COMMIT;
```

```sql
BEGIN TRANSACTION;

SELECT * FROM transfers WHERE id = 1;
-- [id: 1, status: 'CREATED']

UPDATE transfers
SET status = 'PAYMENT_RECEIVED'
WHERE id = 1;

COMMIT;
```

Lost update

```sql
BEGIN TRANSACTION;

SELECT * FROM transfers WHERE id = 1;
-- [id: 1, status: 'CREATED']

UPDATE transfers
SET status = 'CANCELLED'
WHERE id = 1;

COMMIT;
```

```sql
BEGIN TRANSACTION;

SELECT * FROM transfers WHERE id = 1;
-- [id: 1, status: 'CREATED']

UPDATE transfers
SET status = 'PAYMENT_RECEIVED'
WHERE id = 1;

COMMIT;
```

Lost update

```sql
BEGIN TRANSACTION;

SELECT * FROM transfers WHERE id = 1;
-- [id: 1, status: 'CREATED']

UPDATE transfers
SET status = 'CANCELLED'
WHERE id = 1;

COMMIT;
```

```sql
BEGIN TRANSACTION;

SELECT * FROM transfers WHERE id = 1;
-- [id: 1, status: 'CREATED']

UPDATE transfers
SET status = 'PAYMENT_RECEIVED'
WHERE id = 1;

COMMIT;
```

Lost update

```sql
BEGIN TRANSACTION;

SELECT * FROM transfers WHERE id = 1;
-- [id: 1, status: 'CREATED']

UPDATE transfers
SET status = 'CANCELLED'
WHERE id = 1;

COMMIT;
```

```sql
BEGIN TRANSACTION;

SELECT * FROM transfers WHERE id = 1;
-- [id: 1, status: 'CREATED']

UPDATE transfers
SET status = 'PAYMENT_RECEIVED'
WHERE id = 1;

COMMIT;
```

Lost update

```sql
BEGIN TRANSACTION;

SELECT * FROM transfers WHERE id = 1;
-- [id: 1, status: 'CREATED']

UPDATE transfers
SET status = 'CANCELLED'
WHERE id = 1;

COMMIT;
```

```sql
BEGIN TRANSACTION;

SELECT * FROM transfers WHERE id = 1;
-- [id: 1, status: 'CREATED']

UPDATE transfers
SET status = 'PAYMENT_RECEIVED'
WHERE id = 1;

COMMIT;
```

Lost update

```sql
SELECT * FROM transfers WHERE id = 1;
-- [id: 1, status: ???]
```

Lost update

```
SELECT * FROM transfers WHERE id = 1;
-- [id: 1, status: ???]
```

Традиционные подходы

Вариант #1

Транзакция на базе

Вариант #1: Транзакция на базе

```
1  BEGIN TRANSACTION;
```

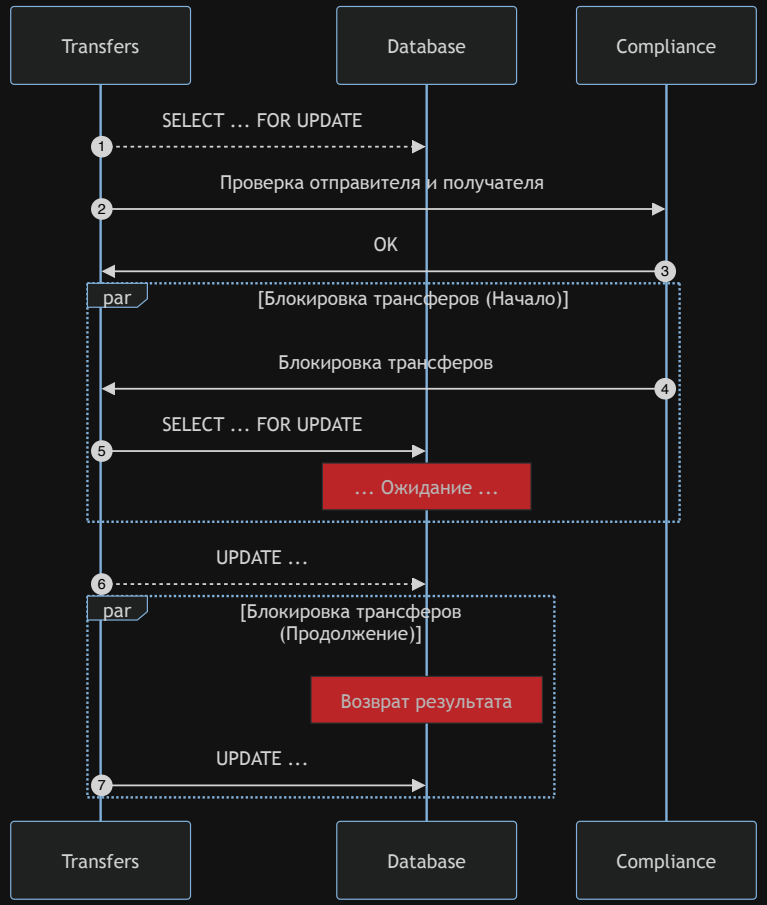
Вариант #1: Транзакция на базе

```
1  BEGIN TRANSACTION;
```

Вариант #1: Транзакция на базе

```
1  BEGIN TRANSACTION;
```

Вариант #1: Транзакция на базе



Ограничения транзакций на базе

Время обработки *5* секунд

x

100 платежей в секунду

=

500 активных транзаций

Вариант #2: Блокировки

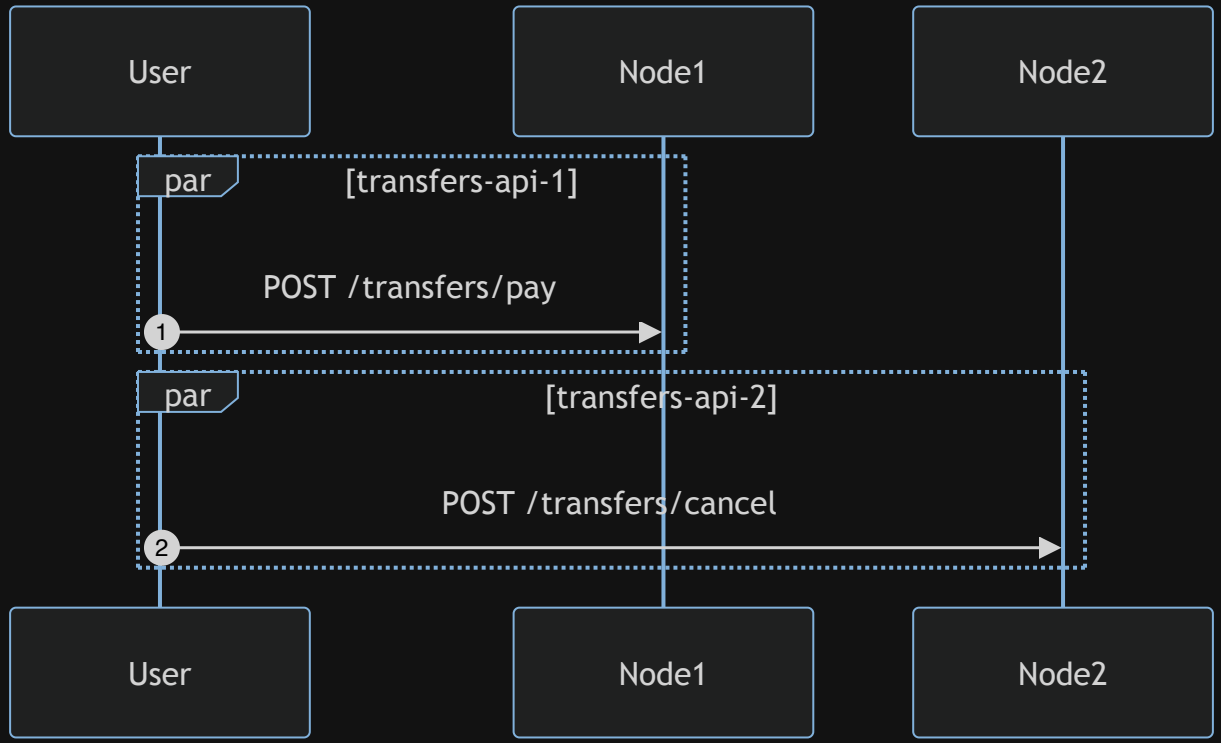
Вариант #2.1: Локальные блокировки

```
1  class Transfer {
2    private final ReentrantLock lock = new ReentrantLock();
3  }
```

Вариант #2.1: Локальные блокировки

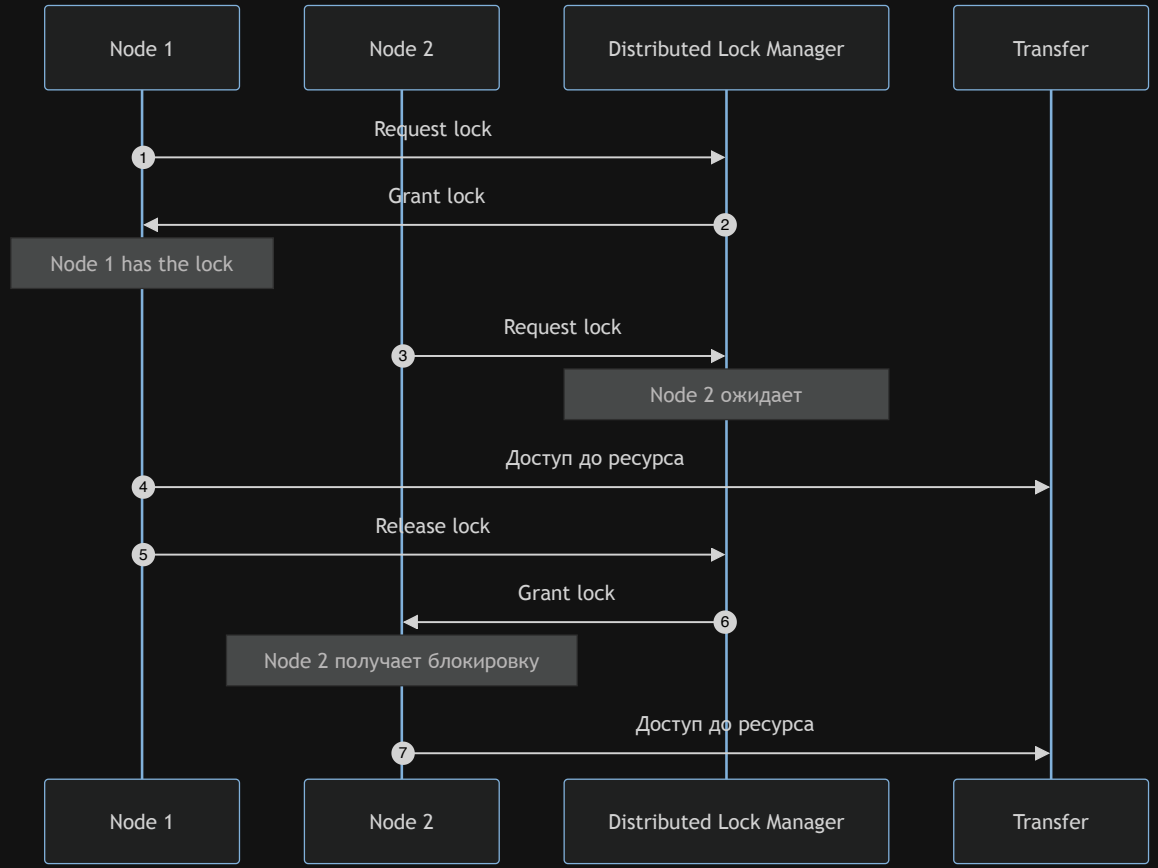
```
1  class Transfer {
2    private final ReentrantLock lock = new ReentrantLock();
3  }
```

Ограничение локальных блокировок



Вариант #2.2: Распределенные блокировки

Распределенные блокировки



Хранилище распределенных блокировок

Хранилище распределенных блокировок

- Hazelcast
- Zookeeper
- Etcd
- Consul
- Redis

Ограничение распределенных блокировок

Ограничение распределенных блокировок

- Проблема порядка блокировок

Ограничение распределенных блокировок

- Проблема порядка блокировок
- Таймауты

Ограничение распределенных блокировок

- Проблема порядка блокировок
- Таймауты
  - Таймаут на ожидание блокировки

Ограничение распределенных блокировок

- Проблема порядка блокировок
- Таймауты
  - Таймаут на ожидание блокировки
  - Таймаут на удержание блокировки

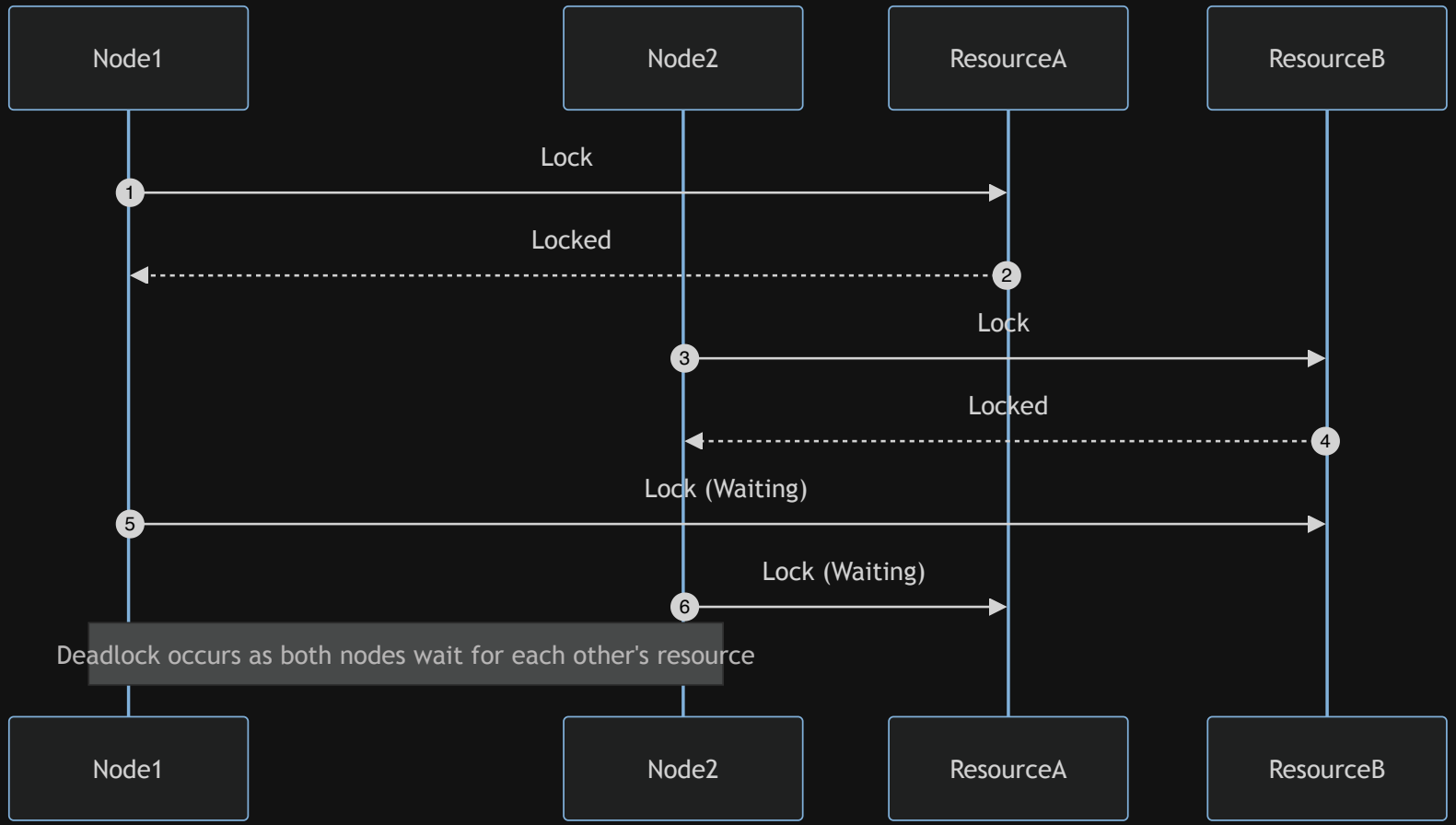
Ограничение распределенных блокировок

- Проблема порядка блокировок
- Таймауты
  - Таймаут на ожидание блокировки
  - Таймаут на удержание блокировки
- Обработка таймаутов

Ограничение распределенных блокировок

- Проблема порядка блокировок
- Таймауты
  - Таймаут на ожидание блокировки
  - Таймаут на удержание блокировки
- Обработка таймаутов
  - Что мы делаем в случае таймаута?

Дедлоки


Node1
Node2
ResourceA
ResourceB
Lock
1
Locked
2
Lock
3
Locked
4
Lock (Waiting)
5
Lock (Waiting)
6
Deadlock occurs as both nodes wait for each other's resource
Node1
Node2
ResourceA
ResourceB

Асинхронная обработка

Модель трансфера: Finite State Machine

Модель трансфера: Finite State Machine

- У трансфера есть несколько состояний

Модель трансфера: Finite State Machine

- У трансфера есть несколько состояний
- Каждое состояние определяет возможные команды

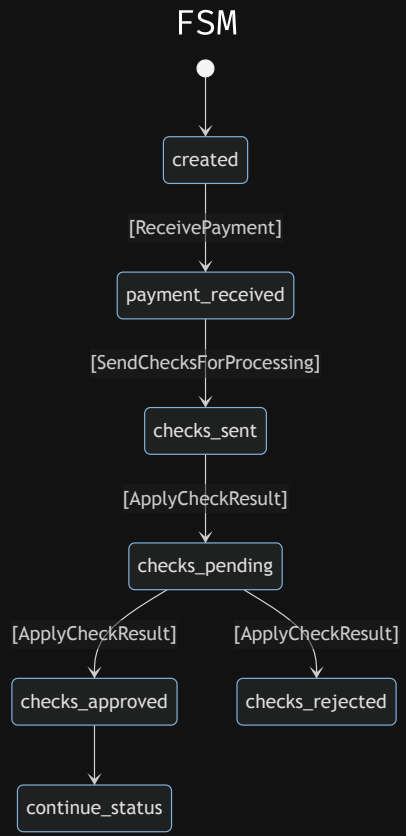
Модель трансфера: Finite State Machine

- У трансфера есть несколько состояний
- Каждое состояние определяет возможные команды
- Переход между состояния происходит через команды

Асинхронная обработка

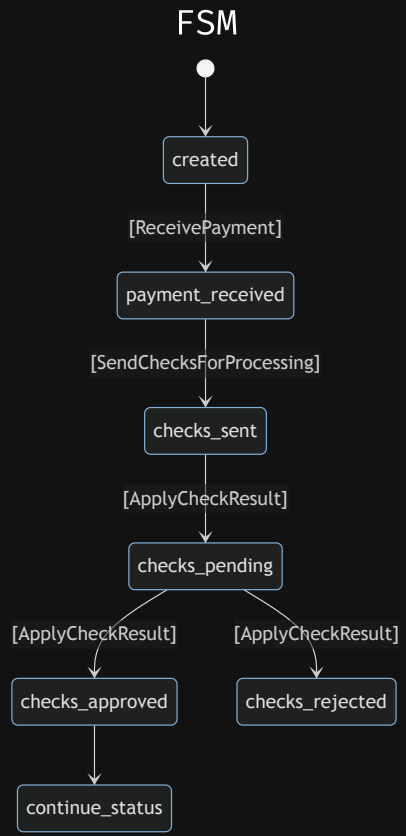
```
1  public record Transfer(UUID id, Status status) {}
```

Асинхронная обработка



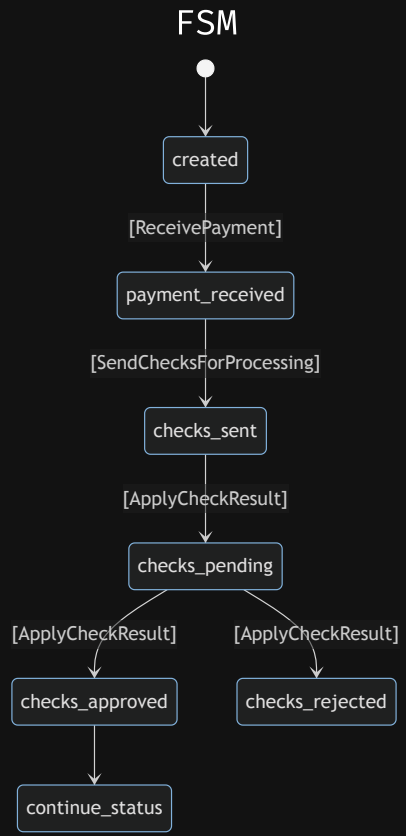

```
1  public record Transfer(UUID id, Status status) {}
```

Асинхронная обработка




```
1  public record Transfer(UUID id, Status status) {}
```

Асинхронная обработка




```
1  public record Transfer(UUID id, Status status) {}
```

Command Processing

```
1  public void process(Command command) {
2    switch (command) {
3    case ReceivePaymentCommand x → {
4        checkStatus(this.status, Status.CREATED);
5        this.paymentDetails = x.paymentDetails;
6        this.status = Status.PAYMENT_RECEIVED;
7        sender.sendChecksCommand();
8        save();
9      }
10     case SendChecksCommand x → {
11       // ...
12     }
13     case ApplyCheckResultCommand x → {
14       // ...
15     }
16   }
17 }
```

Command Processing

```
1  public void process(Command command) {
2    switch (command) {
3    case ReceivePaymentCommand x → {
4        checkStatus(this.status, Status.CREATED);
5        this.paymentDetails = x.paymentDetails;
6        this.status = Status.PAYMENT_RECEIVED;
7        sender.sendChecksCommand();
8        save();
9      }
10     case SendChecksCommand x → {
11       // ...
12     }
13     case ApplyCheckResultCommand x → {
14       // ...
15     }
16   }
17 }
```

Command Processing

```
1  public void process(Command command) {
2    switch (command) {
3    case ReceivePaymentCommand x → {
4        checkStatus(this.status, Status.CREATED);
5        this.paymentDetails = x.paymentDetails;
6        this.status = Status.PAYMENT_RECEIVED;
7        sender.sendChecksCommand();
8        save();
9      }
10     case SendChecksCommand x → {
11       // ...
12     }
13     case ApplyCheckResultCommand x → {
14       // ...
15     }
16   }
17 }
```

Command Processing

```
1  public void process(Command command) {
2    switch (command) {
3    case ReceivePaymentCommand x → {
4        checkStatus(this.status, Status.CREATED);
5        this.paymentDetails = x.paymentDetails;
6        this.status = Status.PAYMENT_RECEIVED;
7        sender.sendChecksCommand();
8        save();
9      }
10     case SendChecksCommand x → {
11       // ...
12     }
13     case ApplyCheckResultCommand x → {
14       // ...
15     }
16   }
17 }
```

Command Processing

```
1  public void process(Command command) {
2    switch (command) {
3    case ReceivePaymentCommand x → {
4        checkStatus(this.status, Status.CREATED);
5        this.paymentDetails = x.paymentDetails;
6        this.status = Status.PAYMENT_RECEIVED;
7        sender.sendChecksCommand();
8        save();
9      }
10     case SendChecksCommand x → {
11       // ...
12     }
13     case ApplyCheckResultCommand x → {
14       // ...
15     }
16   }
17 }
```

Command Processing

```
1  public void process(Command command) {
2    switch (command) {
3    case ReceivePaymentCommand x → {
4        checkStatus(this.status, Status.CREATED);
5        this.paymentDetails = x.paymentDetails;
6        this.status = Status.PAYMENT_RECEIVED;
7        sender.sendChecksCommand();
8        save();
9      }
10     case SendChecksCommand x → {
11       // ...
12     }
13     case ApplyCheckResultCommand x → {
14       // ...
15     }
16   }
17 }
```

Command Processing

```
1  public void process(Command command) {
2    switch (command) {
3    case ReceivePaymentCommand x → {
4        checkStatus(this.status, Status.CREATED);
5        this.paymentDetails = x.paymentDetails;
6        this.status = Status.PAYMENT_RECEIVED;
7        sender.sendChecksCommand();
8        save();
9      }
10     case SendChecksCommand x → {
11       // ...
12     }
13     case ApplyCheckResultCommand x → {
14       // ...
15     }
16   }
17 }
```

Требования для асинхронной обработки

Требования для асинхронной обработки

- Взаимодействие через сообщения

Требования для асинхронной обработки

- Взаимодействие через сообщения
- One-at-a-time обработка сообщений

Акторная модель

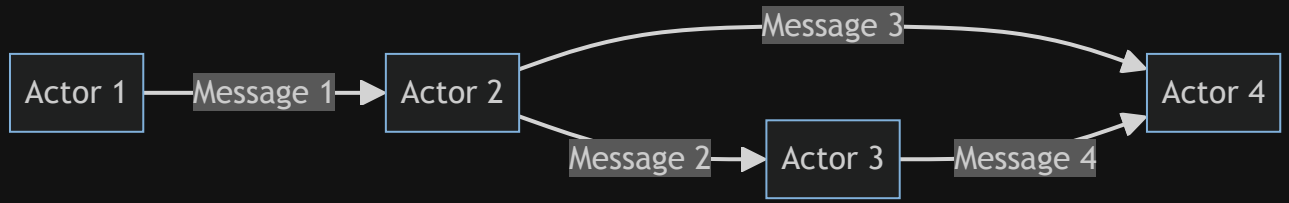
Акторная модель

> Модель а́кторов — математическая модель параллельных вычислений, строящаяся вокруг понятия актора, считающегося универсальным примитивом параллельного исполнения.

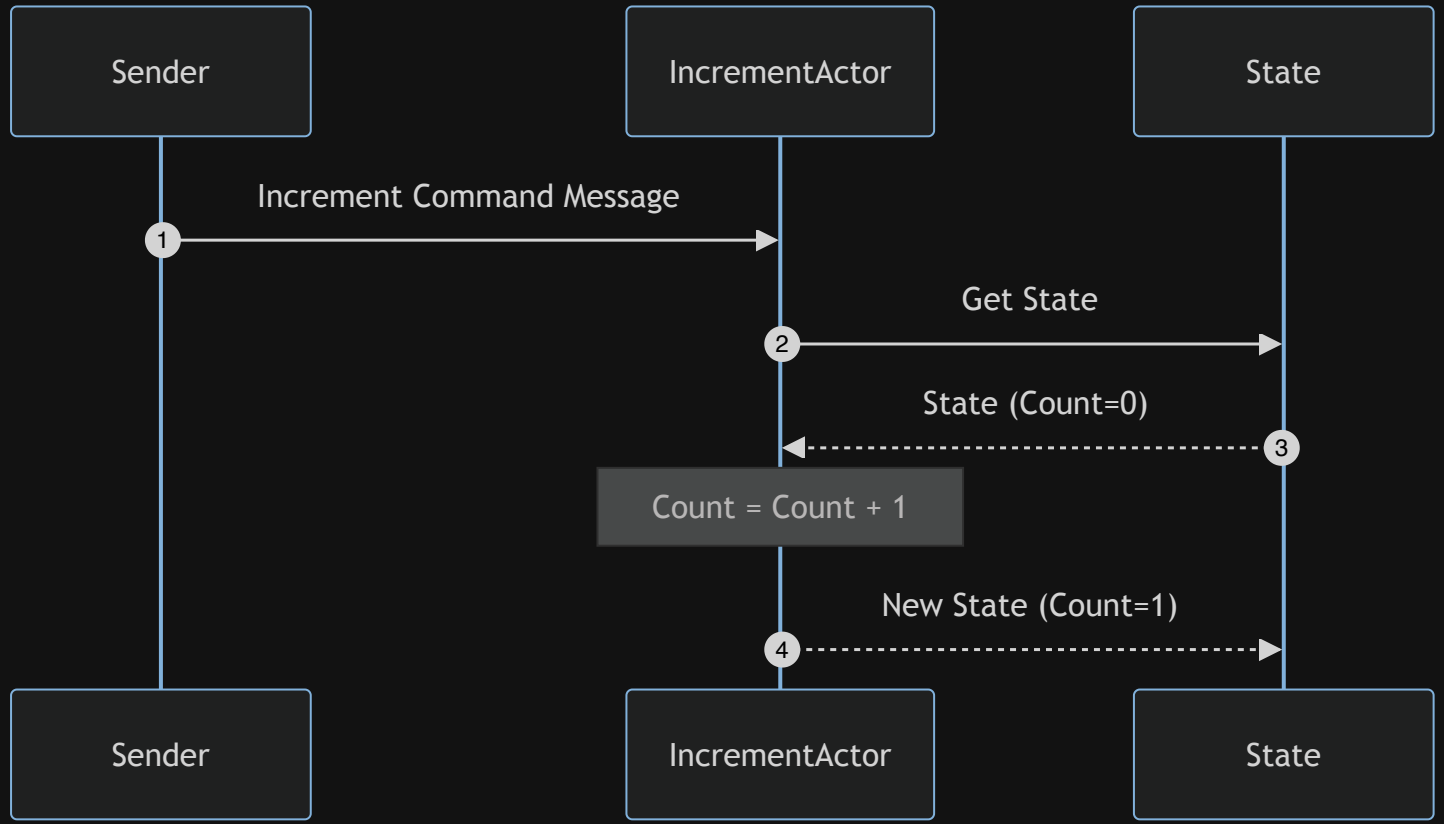
> Актор в данной модели взаимодействует путём обмена сообщениями с другими акторами, и каждый в ответ на получаемые сообщения может принимать локальные решения, создавать новые акторы, посылать свои сообщения, устанавливать, как следует реагировать на последующие сообщения.

Wiki

Акторная модель



Акторная модель с состоянием



Принципы

Принципы

- Актор – фундаментальная единица

Принципы

- Актор – фундаментальная единица
- Асинхронный обмен сообщениями

Принципы

- Актор – фундаментальная единица
- Асинхронный обмен сообщениями
- Изоляция состояния

Принципы

- Актор – фундаментальная единица
- Асинхронный обмен сообщениями
- Изоляция состояния
- Последовательная обработка сообщений

Принципы

- Актор – фундаментальная единица
- Асинхронный обмен сообщениями
- Изоляция состояния
- Последовательная обработка сообщений
- Устойчивость к сбоям

Принципы

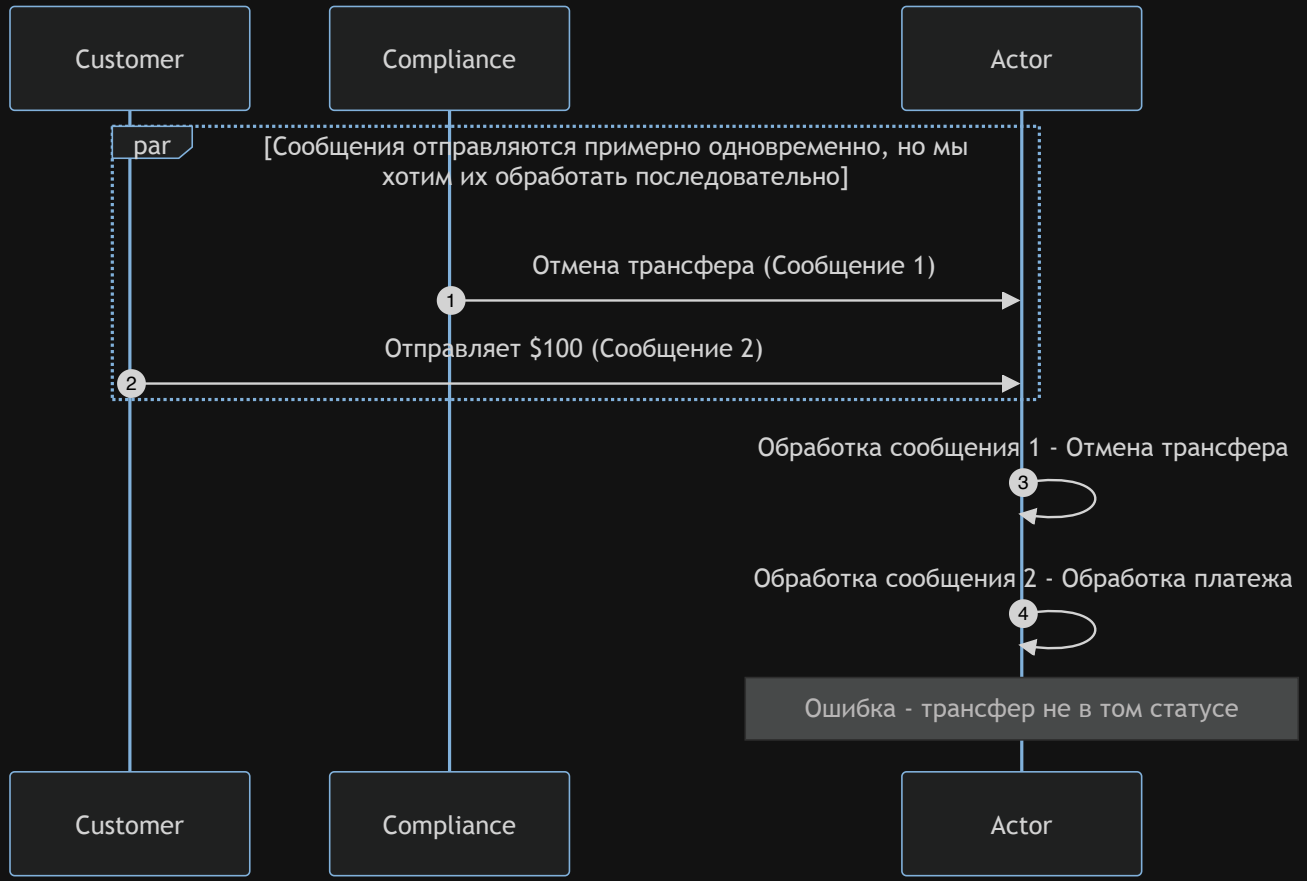
- Актор – фундаментальная единица
- Асинхронный обмен сообщениями
- Изоляция состояния
- Последовательная обработка сообщений
- Устойчивость к сбоям
- Масштабируемость

Утиный тест 

~~Если это выглядит как утка, плавает как утка и крякает как утка. Вероятно, это утка~~

Если это выглядит как актор, обрабатывает сообщения как актор и хранит состояние как актор. Вероятно, это актор

Акторная модель в системе трансферов



Интерфейсы

Интерфейсы

Storage 📦 - хранение состояния

```
1  interface Storage<K, S> {
2    Optional<S> get(K key);
3    void put(K key, S value);
4  }
```

Интерфейсы

Storage 📦 - хранение состояния

```
interface Storage<K, S> {
  Optional<S> get(K key);
  void put(K key, S value);
}
```

Actor 💻 - тут бизнес-логика

```
interface Actor<S, C> {
    S receive(S state, C command);
}
```

Интерфейсы

Storage 📦 - хранение состояния

```
interface Storage<K, S> {
  Optional<S> get(K key);
  void put(K key, S value);
}
```

Actor 💻 - тут бизнес-логика

```
interface Actor<S, C> {
    S receive(S state, C command);
}
```

Actor Mailbox 💌 - по сути очередь сообщений

```
interface ActorMailbox<M> {
    enqueue(M message);

    Optional<M> dequeue();
}
```

Интерфейсы - Actor Mailbox

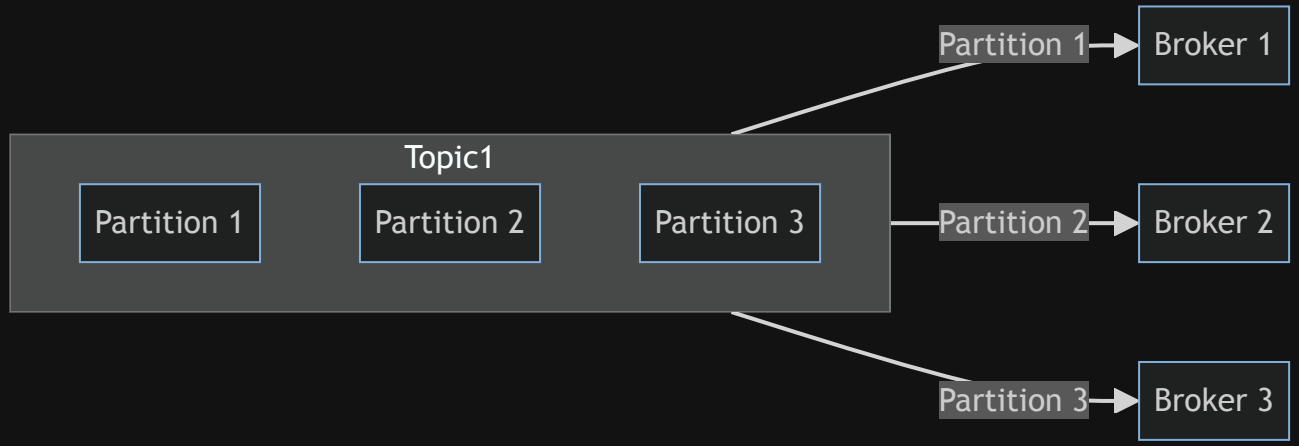
Для mailbox нам нужна какая-нибудь система обмена сообщениями. Что-нибудь надежное, масштабируемое и быстрое. Чтобы еще мы могли гарантировать последовательную обработку сообщений для каждого актора.

Kafka

Kafka ❤️

Kafka основы

Kafka основы

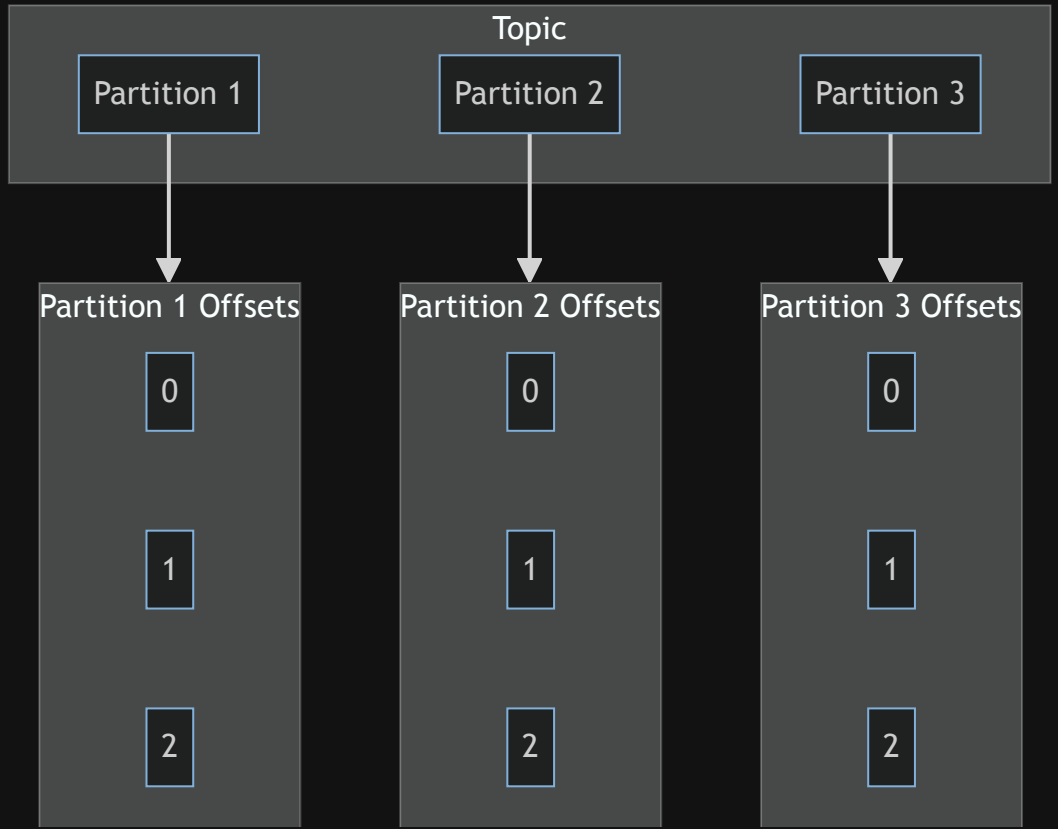


Kafka основы

Partition Offsets

Kafka основы

Partition Offsets

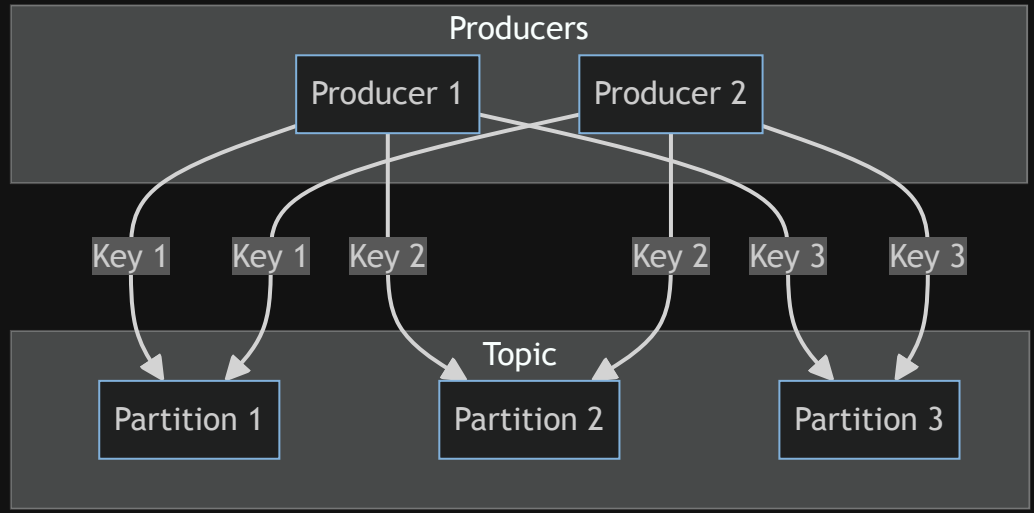


Kafka основы

Роутинг сообщений

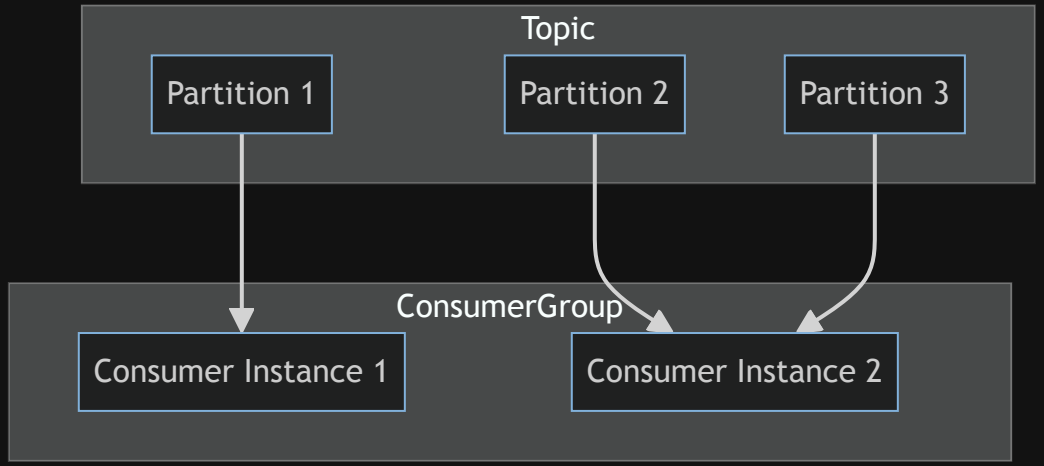
Kafka основы

Роутинг сообщений



Consumer Groups

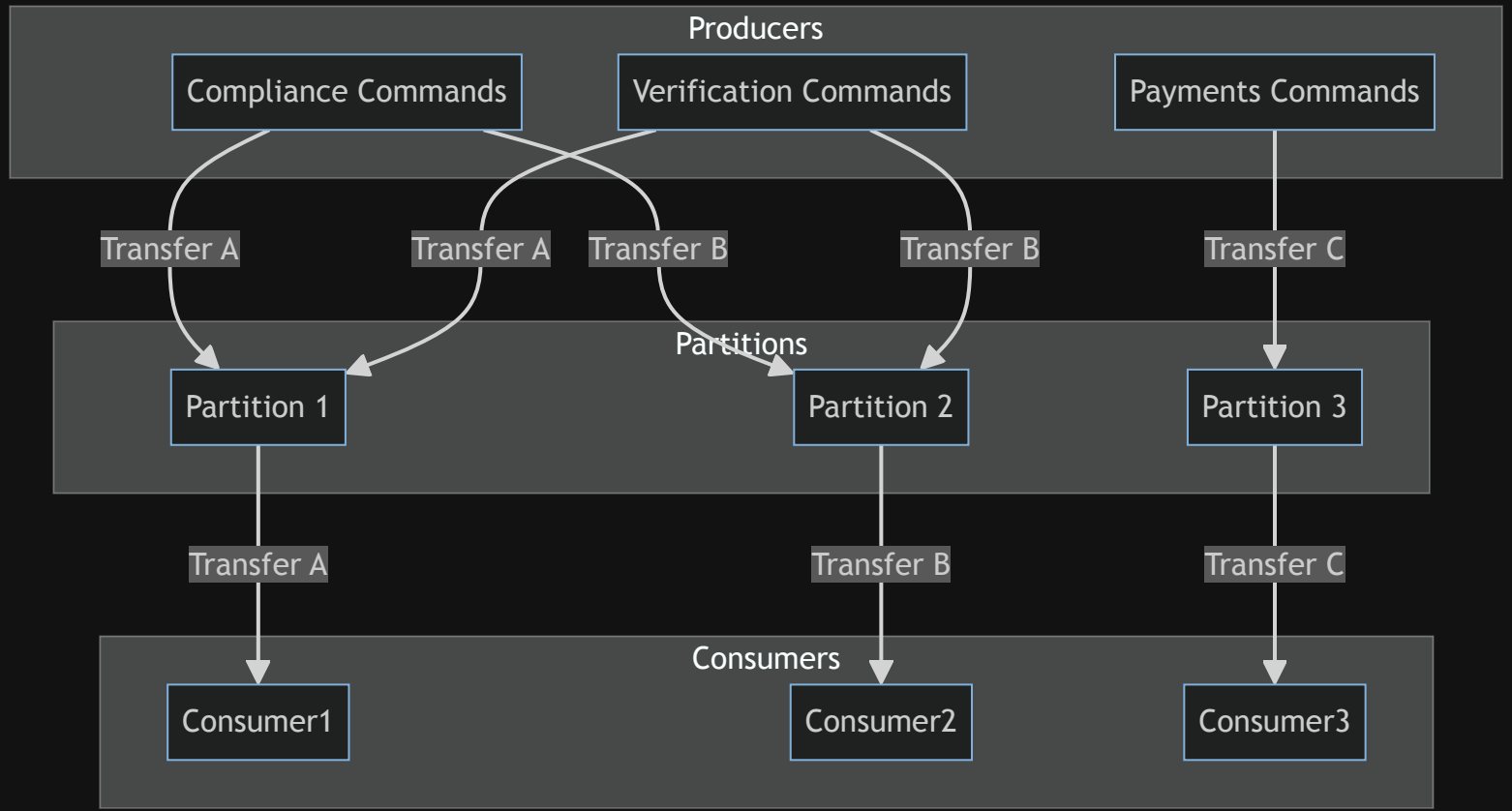
Consumer Groups



Подходит. Берем Kafka 🤝

Система на основе Kafka

Соберем все вместе и построим систему на основе Kafka



Важное замечание

Это не полная реализация модели акторов, как в Erlang или Akka.

Мы будем использовать модель акторов как концепцию для построения системы.

Kafka Consumer

```
1   class Consumer<K, S, C> {
2     void consume(Record<K, C> record) {
3         K key = record.getKey();
4         C command = record.getValue();
5
6         S state = storage.Get(key).orElseGet(() → storage.New(key));
7
8         S newState = actor.receive(state, command).getState();
9         storage.Put(key, newState);
10    }
11  }
```

Падажи!

Тут может быть запись в PostgreSQL и Kafka! 😱

Проблема двойной записи

Если сервис упадет после записи в PostgreSQL, но до отправки сообщения в Kafka, данные будут несогласованными.

Проблема двойной записи: Outbox Pattern

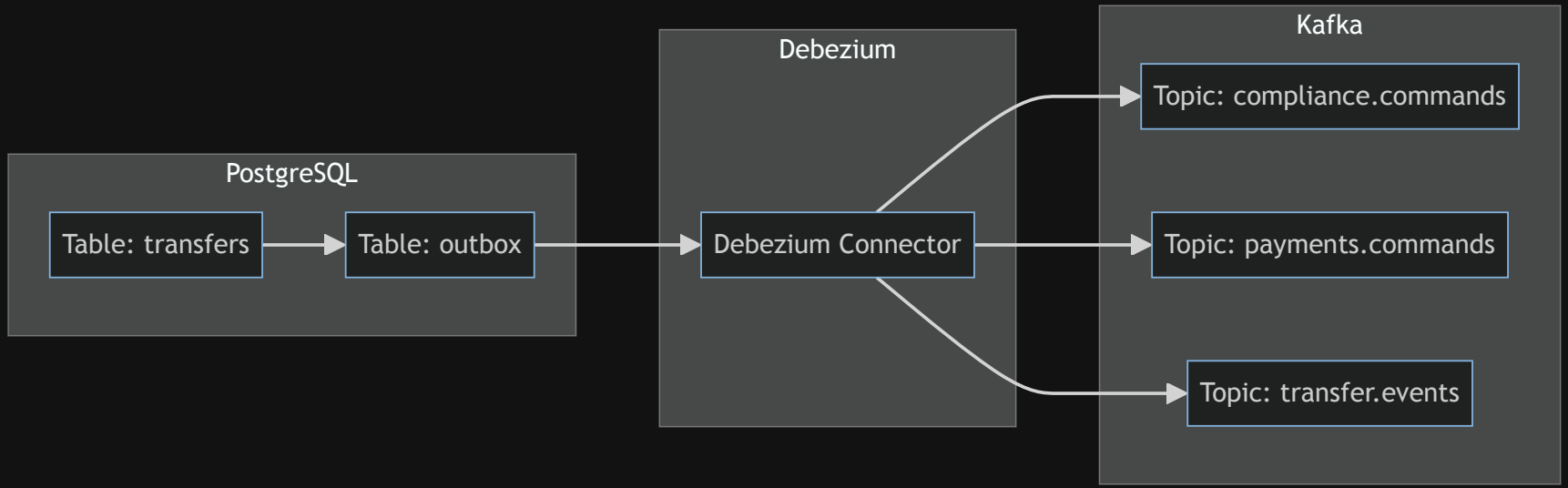
```
void consume(Record<K, C> record) {
  K key = record.getKey();
  C command = record.getValue();

  Optional<S> optionalState = storage.Get(key);
  S state = optionalState.orElseGet(() → storage.New(key));

  Pair<S, List<E>> result = actor.receive(state, command);
  S newState = result.getState();
  List<E> effects = result.getEffects();

  Transaction tx = db.begin();
  try {
    storage.put(tx, key, newState);
    for (E effect : effects) {
      outbox.put(tx, effect);
    }
    tx.commit();
  } catch (Exception e) {
    tx.rollback();
    throw e;
  }
}
```

Outbox Pattern: Запись сообщений из Outbox в Kafka



Toxic-сообщения

Toxic-сообщения

Toxic-сообщения

- Toxic-сообщения необходимо обрабатывать

Toxic-сообщения

- Toxic-сообщения необходимо обрабатывать
- Toxic-сообщения могут быть из-за:

Toxic-сообщения

- Toxic-сообщения необходимо обрабатывать
- Toxic-сообщения могут быть из-за:
  - Некорректного формата сообщения

Toxic-сообщения

- Toxic-сообщения необходимо обрабатывать
- Toxic-сообщения могут быть из-за:
  - Некорректного формата сообщения
  - Некорректной версии сообщения

Toxic-сообщения

- Toxic-сообщения необходимо обрабатывать
- Toxic-сообщения могут быть из-за:
  - Некорректного формата сообщения
  - Некорректной версии сообщения
  - Некорректных данных сообщения

Toxic-сообщения

- Toxic-сообщения необходимо обрабатывать
- Toxic-сообщения могут быть из-за:
  - Некорректного формата сообщения
  - Некорректной версии сообщения
  - Некорректных данных сообщения
- Ошибки логики

Toxic-сообщения

- Toxic-сообщения необходимо обрабатывать
- Toxic-сообщения могут быть из-за:
  - Некорректного формата сообщения
  - Некорректной версии сообщения
  - Некорректных данных сообщения
- Ошибки логики
  - Некорректное состояние актора

Toxic-сообщения

- Toxic-сообщения необходимо обрабатывать
- Toxic-сообщения могут быть из-за:
  - Некорректного формата сообщения
  - Некорректной версии сообщения
  - Некорректных данных сообщения
- Ошибки логики
  - Некорректное состояние актора
  - Некорректная обработка сообщения

Toxic-сообщения и Dead-Letters

```java
public void consume(Record<K, C> record) {
  K key = record.getKey();
  C command = record.getValue();

  Optional<S> optionalState = storage.get(key);
  S state = optionalState.orElseGet(() → storage.new(key));

  try {
    S newState = actor.receive(state, command);
    storage.Put(key, newState);
  } catch (Exception e) {
    if (e instanceof KafkaToxicException) {
      deadLetters.produce(record);
    }

    ...
  }
}
```

Реализация Kafka Consumer

```
1   void consume(Record<K, C> record) {
2     K key = record.getKey();
3     C command = record.getValue();
4
5     S state = storage.Get(key).orElseGet(() → storage.New(key));
6
7     try {
8       Pair<S, List<E>> result = actor.receive(state, command);
9       S newState = result.getState();
10      List<E> effects = result.getEffects();
11
12      Transaction tx = db.begin();
13      storage.Put(tx, key, newState);
14      for (E effect : effects) {
15        outbox.Put(tx, effect);
16      }
17      tx.commit();
18    } catch (Exception e) {
19      if (e instanceof KafkaToxicException) {
20        deadLetters.produce(record);
21      } else {
22        throw e;
23      }
24    }
25  }
```

Реализация Kafka Consumer

```
void consume(Record<K, C> record) {
  K key = record.getKey();
  C command = record.getValue();

  S state = storage.Get(key).orElseGet(() → storage.New(key));

  try {
    Pair<S, List<E>> result = actor.receive(state, command);
    S newState = result.getState();
    List<E> effects = result.getEffects();

    Transaction tx = db.begin();
    storage.Put(tx, key, newState);
    for (E effect : effects) {
      outbox.Put(tx, effect);
    }
    tx.commit();
  } catch (Exception e) {
    if (e instanceof KafkaToxicException) {
      deadLetters.produce(record);
    } else {
      throw e;
    }
  }
}
```

Реализация Kafka Consumer

```
void consume(Record<K, C> record) {
  K key = record.getKey();
  C command = record.getValue();

  S state = storage.Get(key).orElseGet(() → storage.New(key));

  try {
    Pair<S, List<E>> result = actor.receive(state, command);
    S newState = result.getState();
    List<E> effects = result.getEffects();

    Transaction tx = db.begin();
    storage.Put(tx, key, newState);
    for (E effect : effects) {
      outbox.Put(tx, effect);
    }
    tx.commit();
  } catch (Exception e) {
    if (e instanceof KafkaToxicException) {
      deadLetters.produce(record);
    } else {
      throw e;
    }
  }
}
```

Реализация Kafka Consumer

```
1   void consume(Record<K, C> record) {
2     K key = record.getKey();
3     C command = record.getValue();
4
5     S state = storage.Get(key).orElseGet(() → storage.New(key));
6
7     try {
8       Pair<S, List<E>> result = actor.receive(state, command);
9       S newState = result.getState();
10      List<E> effects = result.getEffects();
11
12      Transaction tx = db.begin();
13      storage.Put(tx, key, newState);
14      for (E effect : effects) {
15        outbox.Put(tx, effect);
16      }
17      tx.commit();
18    } catch (Exception e) {
19      if (e instanceof KafkaToxicException) {
20        deadLetters.produce(record);
21      } else {
22        throw e;
23      }
24    }
25  }
```

Реализация Kafka Consumer

```
1   void consume(Record<K, C> record) {
2     K key = record.getKey();
3     C command = record.getValue();
4
5     S state = storage.Get(key).orElseGet(() → storage.New(key));
6
7     try {
8       Pair<S, List<E>> result = actor.receive(state, command);
9       S newState = result.getState();
10      List<E> effects = result.getEffects();
11
12      Transaction tx = db.begin();
13      storage.Put(tx, key, newState);
14      for (E effect : effects) {
15        outbox.Put(tx, effect);
16      }
17      tx.commit();
18    } catch (Exception e) {
19      if (e instanceof KafkaToxicException) {
20        deadLetters.produce(record);
21      } else {
22        throw e;
23      }
24    }
25  }
```

Реализация Kafka Consumer

```
1   void consume(Record<K, C> record) {
2     K key = record.getKey();
3     C command = record.getValue();
4
5     S state = storage.Get(key).orElseGet(() → storage.New(key));
6
7     try {
8       Pair<S, List<E>> result = actor.receive(state, command);
9       S newState = result.getState();
10      List<E> effects = result.getEffects();
11
12      Transaction tx = db.begin();
13      storage.Put(tx, key, newState);
14      for (E effect : effects) {
15        outbox.Put(tx, effect);
16      }
17      tx.commit();
18    } catch (Exception e) {
19      if (e instanceof KafkaToxicException) {
20        deadLetters.produce(record);
21      } else {
22        throw e;
23      }
24    }
25  }
```

Реализация актора

```
1   Pair<TransferState, List<Effect>> receive(TransferState state, TransferCommand command) {
2     if (command instanceof ReceivePaymentCommand) {
3       ReceivePaymentCommand x = (ReceivePaymentCommand) command;
4       TransferState newState = state.receivePayment(x.getPaymentDetails());
5
6       List<Effect> effects = Arrays.asList(
7         new SendChecksCommandEffect(state.getTransferID()),
8         new TransferEvent(new PaymentReceivedEvent(state.getTransferID()))
9       );
10
11      return new Pair<>(newState, effects);
12    } else {
13      throw new KafkaToxicException("unknown command");
14    }
15  }
```

Реализация актора

```
1   Pair<TransferState, List<Effect>> receive(TransferState state, TransferCommand command) {
2     if (command instanceof ReceivePaymentCommand) {
3       ReceivePaymentCommand x = (ReceivePaymentCommand) command;
4       TransferState newState = state.receivePayment(x.getPaymentDetails());
5
6       List<Effect> effects = Arrays.asList(
7         new SendChecksCommandEffect(state.getTransferID()),
8         new TransferEvent(new PaymentReceivedEvent(state.getTransferID()))
9       );
10
11      return new Pair<>(newState, effects);
12    } else {
13      throw new KafkaToxicException("unknown command");
14    }
15  }
```

Реализация актора

```
Pair<TransferState, List<Effect>> receive(TransferState state, TransferCommand command) {
  if (command instanceof ReceivePaymentCommand) {
    ReceivePaymentCommand x = (ReceivePaymentCommand) command;
    TransferState newState = state.receivePayment(x.getPaymentDetails());

    List<Effect> effects = Arrays.asList(
      new SendChecksCommandEffect(state.getTransferID()),
      new TransferEvent(new PaymentReceivedEvent(state.getTransferID()))
    );

    return new Pair<>(newState, effects);
  } else {
    throw new KafkaToxicException("unknown command");
  }
}
```

Реализация актора

```
1   Pair<TransferState, List<Effect>> receive(TransferState state, TransferCommand command) {
2     if (command instanceof ReceivePaymentCommand) {
3       ReceivePaymentCommand x = (ReceivePaymentCommand) command;
4       TransferState newState = state.receivePayment(x.getPaymentDetails());
5
6       List<Effect> effects = Arrays.asList(
7         new SendChecksCommandEffect(state.getTransferID()),
8         new TransferEvent(new PaymentReceivedEvent(state.getTransferID()))
9       );
10
11      return new Pair<>(newState, effects);
12    } else {
13      throw new KafkaToxicException("unknown command");
14    }
15  }
```

Реализация актора

```
1   Pair<TransferState, List<Effect>> receive(TransferState state, TransferCommand command) {
2     if (command instanceof ReceivePaymentCommand) {
3       ReceivePaymentCommand x = (ReceivePaymentCommand) command;
4       TransferState newState = state.receivePayment(x.getPaymentDetails());
5
6       List<Effect> effects = Arrays.asList(
7         new SendChecksCommandEffect(state.getTransferID()),
8         new TransferEvent(new PaymentReceivedEvent(state.getTransferID()))
9       );
10
11      return new Pair<>(newState, effects);
12    } else {
13      throw new KafkaToxicException("unknown command");
14    }
15  }
```

Реализация актора

```
1   Pair<TransferState, List<Effect>> receive(TransferState state, TransferCommand command) {
2     if (command instanceof ReceivePaymentCommand) {
3       ReceivePaymentCommand x = (ReceivePaymentCommand) command;
4       TransferState newState = state.receivePayment(x.getPaymentDetails());
5
6       List<Effect> effects = Arrays.asList(
7         new SendChecksCommandEffect(state.getTransferID()),
8         new TransferEvent(new PaymentReceivedEvent(state.getTransferID()))
9       );
10
11      return new Pair<>(newState, effects);
12    } else {
13      throw new KafkaToxicException("unknown command");
14    }
15  }
```

Заключение

Самый простой способ решить проблемы с конкурентностью - избегать конкурентность.

Заключение

Самый простой способ решить проблемы с конкурентностью - избегать конкурентность.

- Акторная модель - это простой способ решить проблемы конкурентности. Мы работаем с одним сообщением за раз - это упрощает жизнь

Заключение

Самый простой способ решить проблемы с конкурентностью - избегать конкурентность.

- Акторная модель - это простой способ решить проблемы конкурентности. Мы работаем с одним сообщением за раз - это упрощает жизнь
- Для реализации акторной модели даже в сложной системе не обязательно использовать сложные инструменты.

Заключение

Самый простой способ решить проблемы с конкурентностью - избегать конкурентность.

- Акторная модель - это простой способ решить проблемы конкурентности. Мы работаем с одним сообщением за раз - это упрощает жизнь
- Для реализации акторной модели даже в сложной системе не обязательно использовать сложные инструменты.
  - Kafka поможет доставить сообщения в нужном порядке для каждого актора. Это надежная и масштабируемая система

Заключение

Самый простой способ решить проблемы с конкурентностью - избегать конкурентность.

- Акторная модель - это простой способ решить проблемы конкурентности. Мы работаем с одним сообщением за раз - это упрощает жизнь
- Для реализации акторной модели даже в сложной системе не обязательно использовать сложные инструменты.
  - Kafka поможет доставить сообщения в нужном порядке для каждого актора. Это надежная и масштабируемая система
  - Для хранилища берите что вам нравится и подходит под вашу задачу

Q&A