Effortless Concurrency

Leveraging the Actor Model in Financial Transaction Systems

Nikita Melnikov
x: @nikita_melnikov
Atlantic Money

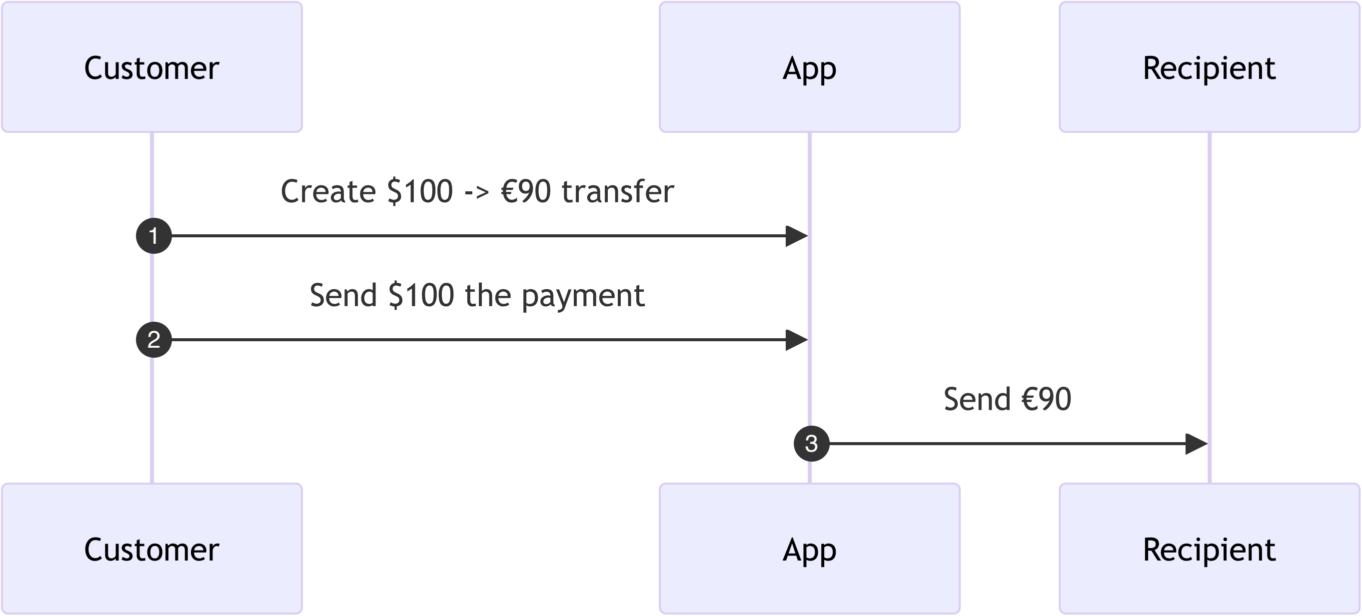
About me

- VP of Engineering at Atlantic Money
- ex-Tinkoff Bank and ex-Tinkoff Investments
- 10+ years in Fintech
- Was working on high-load systems, 300k+ RPS
- Scala, Golang, Postgres, and Kafka ❤️

Agenda

- Financial transaction and typical problems
- Traditional Approaches and Their Limitations
- Shifting to Asynchronous Processing
- Kafka as a Messaging Backbone
- Implenting Asynchronous Processing
- Actor Model
- Conclusions and Q&A

What is financial transaction



Transfer flow

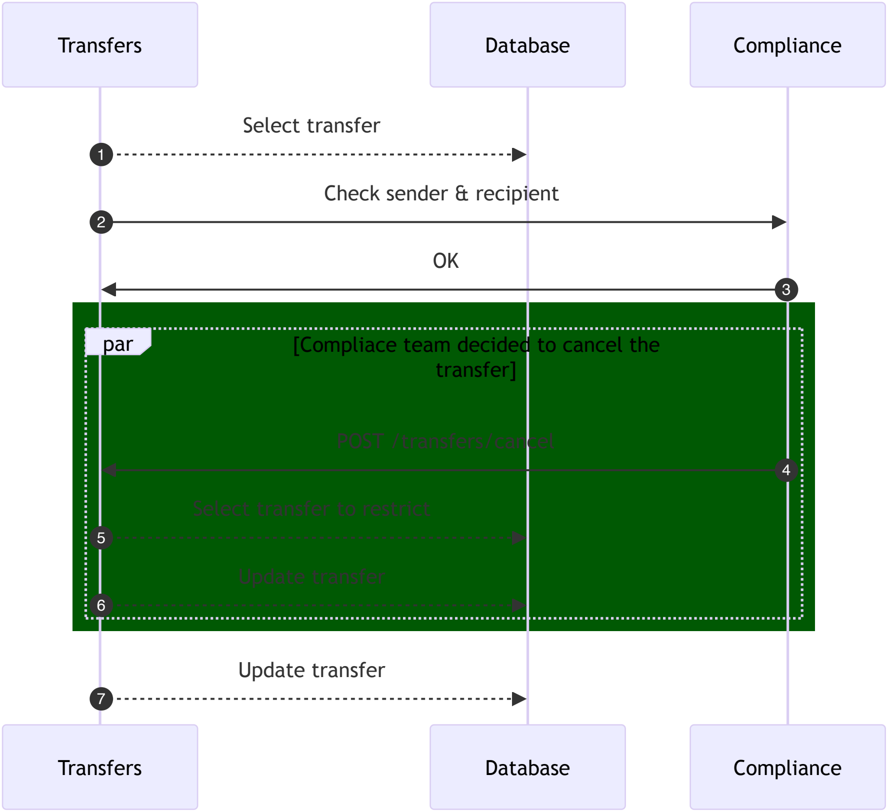
- Wait for USD
- Run checks
- Send EUR

What happens in the real world

- Customer creates USD → EUR transfer
- System waits for USD
- System writes the payment details
- System runs checks
  - Sanction lists
  - Anti-fraud
  - Check payment limits
  - Calculate fees
  - Many more
- Exchange currencies
- Send EUR to the recipient

Typical problem

Lost update


Transfers
Database
Compliance
Select transfer
1
Check sender & recipient
2
OK
3
par
[Compliace team decided to cancel the transfer]
POST /transfers/cancel
4
Select transfer to restrict
5
Update transfer
6
Update transfer
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
SELECT * FROM transfers WHERE id = 1;
-- [id: 1, status: ???]
```

Traditional approaches

Option #1

Database transaction

Option #1: Database transaction

```
BEGIN TRANSACTION;
```

Option #1: Database transaction

Database transaction limitations

Processing time is *5* seconds

x

100 operations / second

=

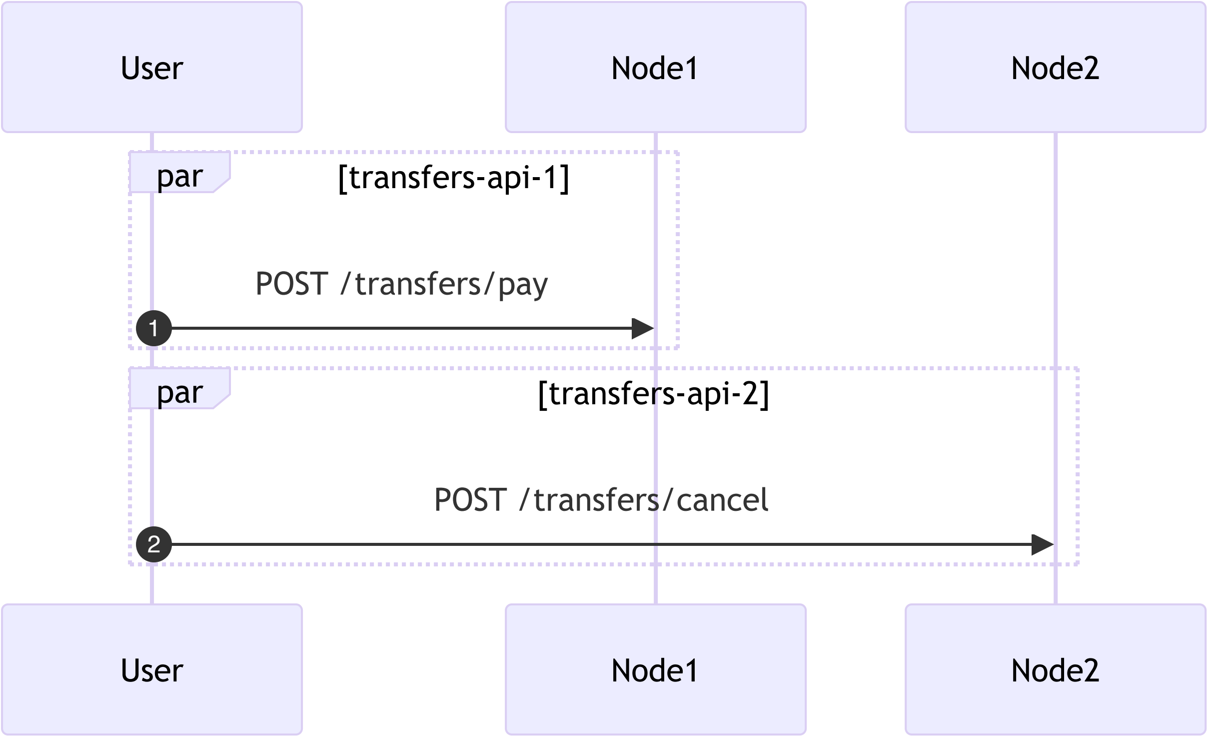
500 active transactions

Option #2: Locks

Option #2.1: Local locks

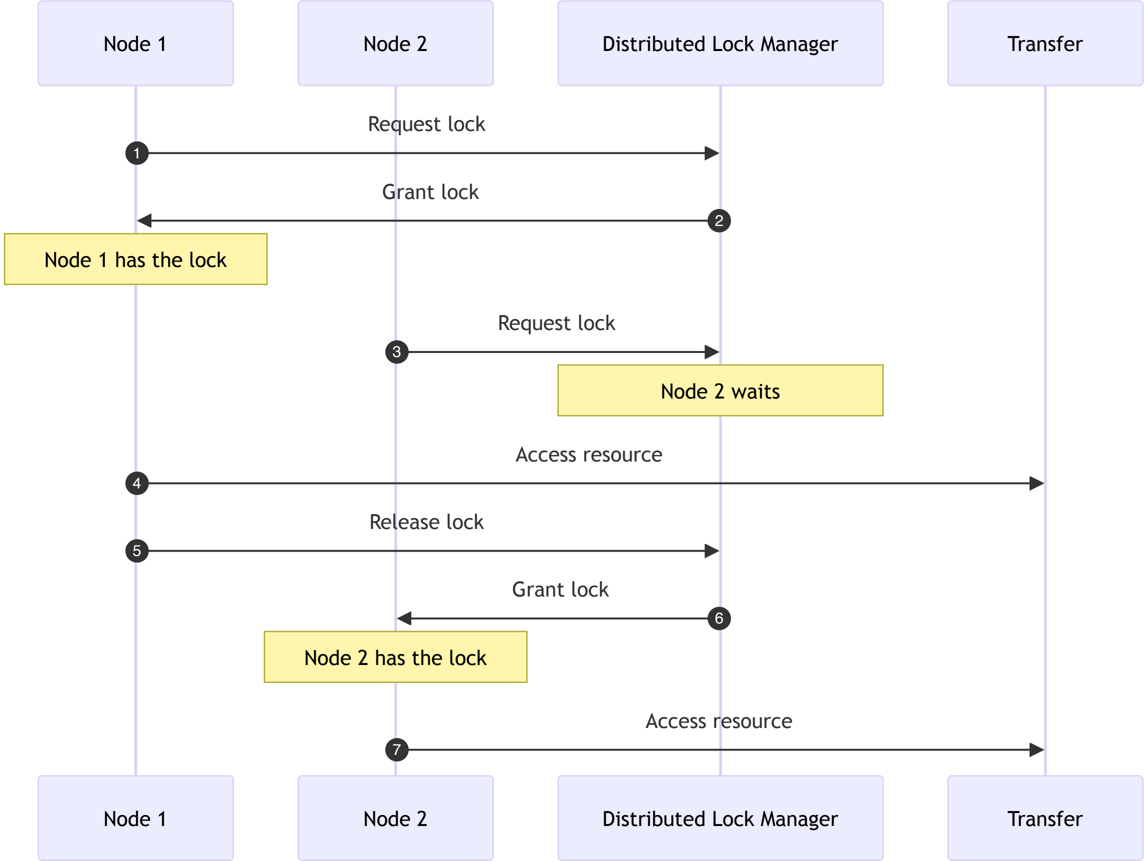
```go
type Transfer struct {
  mu *sync.Mutex
}
```

Local locks limitations



Option #2.2: Distributed locks

Distributed locks



Distributed locks storages

- Hazelcast
- Zookeeper
- Etcd
- Consul
- Redis

Distributed locks limitations

- The Problem of Ordering
- Timeouts
  - Lock Acquisition Timeouts
  - Lock Holding Timeouts
- Timeout Handling
  - What will we do in case of timeouts?
- Potential deadlocks

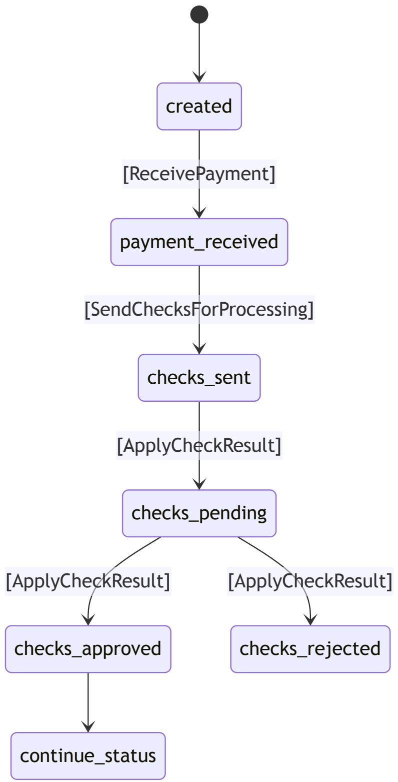
Asynchronous Processing

Transfer Model: Finite State Machine

- Transfer has multiple states
- State transitions occur via commands
- Each state defines allowed commands
- Commands trigger actions and state changes

Asynchronous Processing

FSM



```
struct Transfer {
  ID UUID
  Status Status
}
```

Command Processing

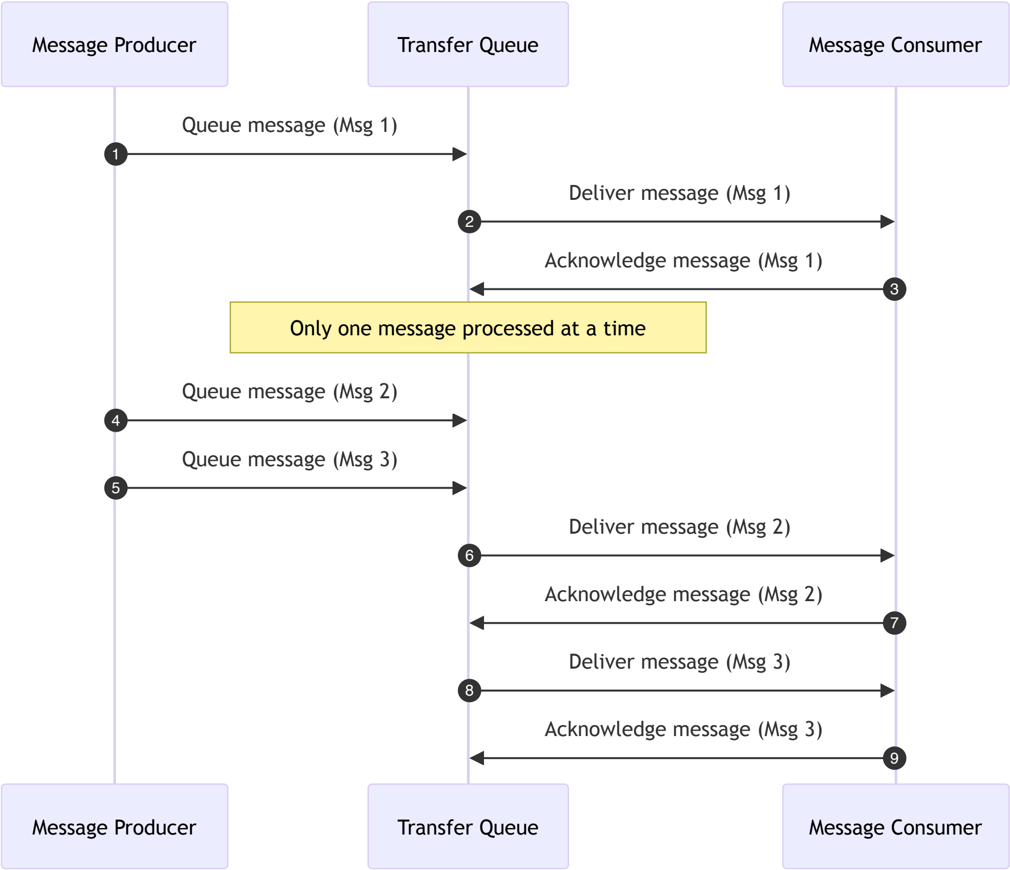
```go
func (t *Transfer) Process(command Command) {
  switch x := command.(type) {
  case ReceivePaymentCommand:
    CheckStatus(t.Status, StatusCreated)

    t.PaymentDetails = x.PaymentDetails
    t.Status = StatusPaymentReceived
    t.Save()

    t.sender.SendChecksCommand()
  case SendChecksCommand:
    CheckStatus(t.Status, StatusPaymentReceived)

    t.Status = StatusChecksSent
    t.Save()

    t.checks.SendRunChecksCommand(t.Checks)
  case ApplyCheckResultCommand:
    CheckStatus(t.Status, StatusChecksPending)
    t.ApplyCheckResult(x.CheckResult)
    t.Status = CalculateNewStatus(x.CheckResult)
    t.Save()
  }
}
```

Requirements for asynchronous processing

- Communication through messages
- One-at-a-time message handling
- Durable message storage

Communication through messages

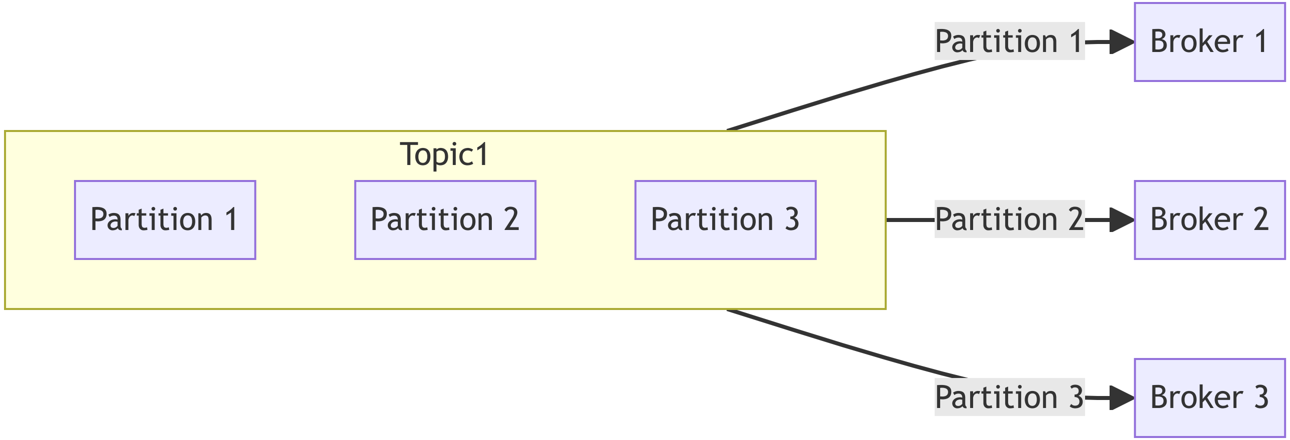
One-at-a-time message handling



Durable message storage

Kafka ❤️

Kafka Basics

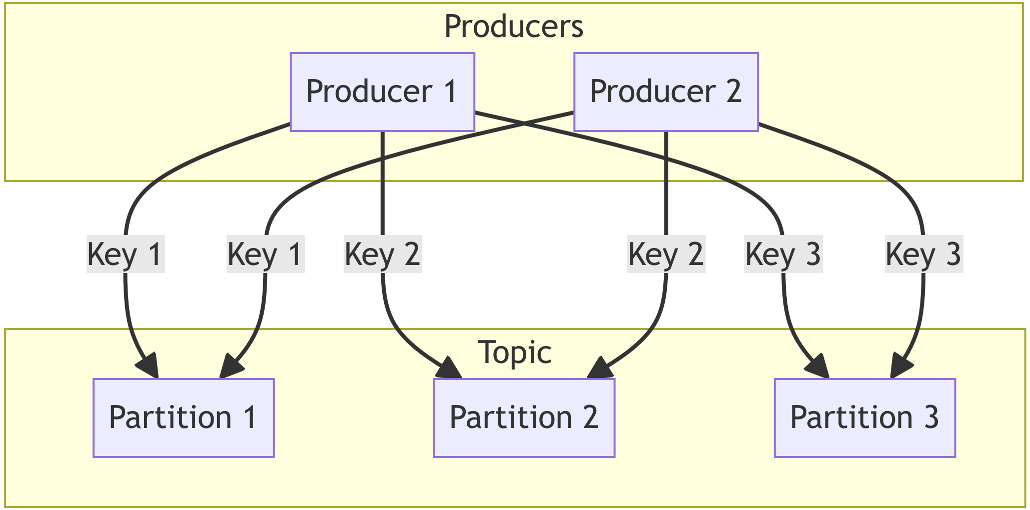


Kafka Basics

Partition Offsets

Kafka Basics

Message Routing



Producer guarantees

- acks=0
  - No acknowledgment is needed
  - Lowest latency
  - No delivery guarantees
- acks=1
  - The leader acknowledges the write.
  - In case of leader failure, data loss is possible.
- acks=all
  - All in-sync replicas.
  - Highest durability
  - Highest latency

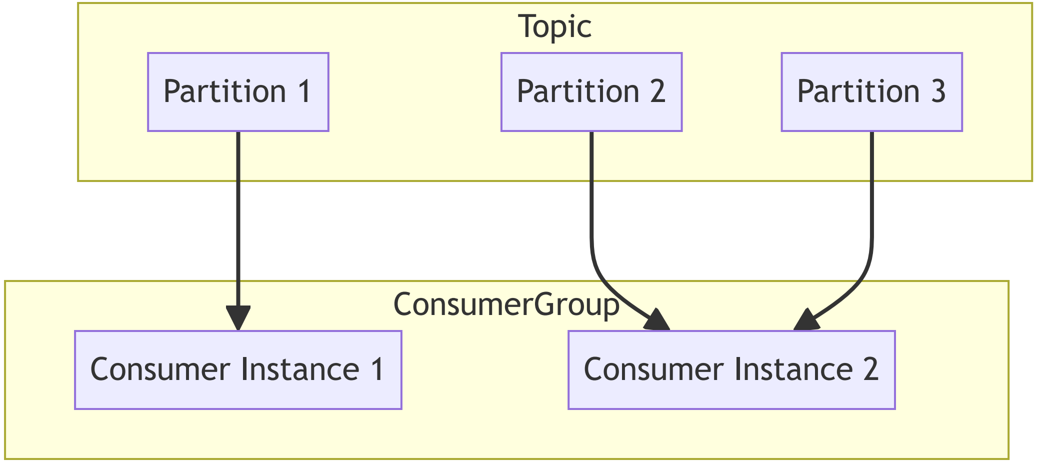
Requirements for asynchronous processing (again)

- Communication through messages - ✅
- Durable message storage - ✅
- One-at-a-time message handling - 🤔

Consumer requirements

- We want to have one-at-a-time message processing
- Each transfer should be processed by a single consumer
- Consumers should be able to scale horizontally

Consumer Groups



We’ve got everything we need!

Messaging system using Kafka

Combine all together to build a messaging system

Actor Model

Core Concepts

- Actors as Fundamental Units
- Asynchronous Message Passing
- State Isolation
- Sequential Message Processing
- Location Transparency
- Fault Tolerance
- Scalability

Important Disclaimer

This is not full implementation of the Actor Model as Erlang or Akka.

We will use the Actor Model as a concept to build a system.

There is no need to implement all the features of the Actor Model such as supervision, location transparency, etc.

Implementing Actor model in the system

Interfaces

Storage

```go
type Storage[K, S any] {
  New(K) S

  Get(K) (S, bool)
  Put(K, S)
}
```

Actor Mailbox

```go
type ActorMailbox[K, S, C any] struct {
  Consume(K, S, C) error
}
```

Actor

```go
type Actor[S, C any] {
  Receive(S, C) (S, error)
}
```

Command Producer

```go
type CommandProducer[K, C any] struct {
  Produce(K, C) error
}
```

- K - key type (transfer id)
- S - value type (transfer state)
- C - command type (transfer command)

Kafka Consumer

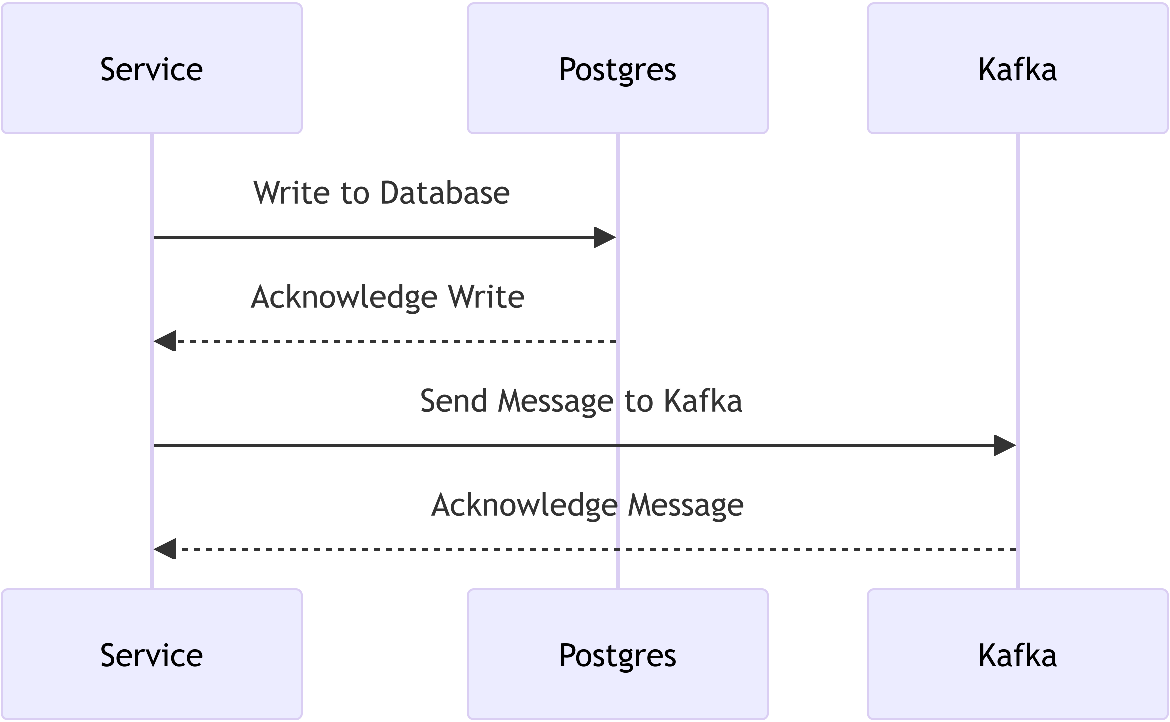
```go
func (c *Consumer) Consume(record *Record) {
  key := record.Key()
  command := record.Value()

  state, found := c.storage.Get(key)
  if !found {
    state = c.storage.New(key)
  }

  newState := c.actor.Receive(state, command)
  c.storage.Put(key, newState)
}
```

Hold on!

There is a problem with double writes!

Double write problem

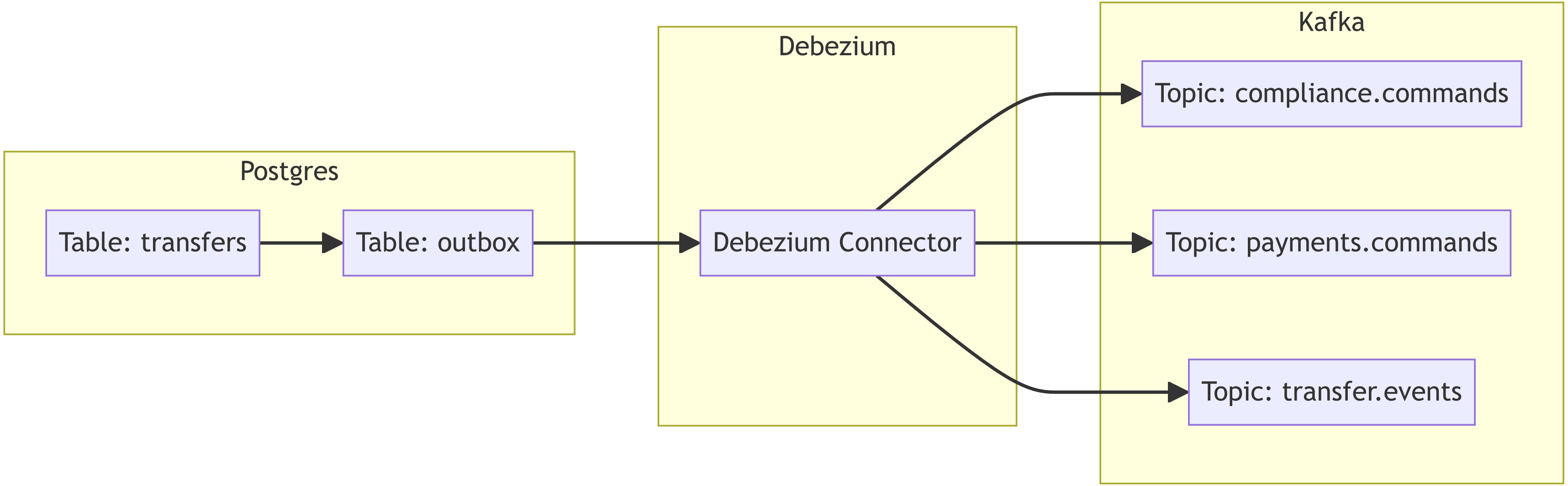
If the service crashes after writing to Postgres but before sending the message to Kafka, the data will be inconsistent.



Double write problem: Outbox Pattern

```go
func (c *Consumer) Consume(record *Record) {
  key := record.Key()
  command := record.Value()

  state, found := c.storage.Get(key)
  if !found {
    state = c.storage.New(key)
  }

  newState, effects := c.actor.Receive(state, command)

  tx := c.db.Begin()
  c.storage.Put(tx, key, newState)
  for _, effect := range effects {
    c.outbox.Put(tx, effect)
  }
  tx.Commit()
}
```

Outbox Pattern: Writing messages from Outbox to Kafka



Toxic Messages

Toxic Messages

- Toxic messages are messages that cannot be processed
- Toxic messages can be caused by:
  - Incorrect message format
  - Incorrect message version
  - Incorrect message data
- Logic errors
  - Incorrect actor state
  - Incorrect message processing

Toxic Messages and Dead-Letters

```go
func (c *Consumer) Consume(record *Record) {
  key := record.Key()
  command := record.Value()

  state, found := c.storage.Get(key)
  if !found {
    state = c.storage.New(key)
  }

  newState, err := c.actor.Receive(state, command)
  if errors.Is(err, kafka.ErrToxic) {
    return c.deadLetters.Produce(record)
  }
  c.storage.Put(key, newState)
}
```

Kafka Consumer implementation

```go
func (c *Consumer) Consume(record *Record) {
  key := record.Key()
  command := record.Value()

  state, found := c.storage.Get(key)
  if !found {
    state = c.storage.New(key)
  }

  newState, effects, err := c.actor.Receive(state, command)
  if errors.Is(err, kafka.ErrToxic) {
    return c.deadLetters.Produce(record)
  }

  tx := c.db.Begin()
  c.storage.Put(tx, key, newState)
  for _, effect := range effects {
    c.outbox.Put(tx, effect)
  }
  tx.Commit()
}
```

Transfer Actor Implementation

```go
func (a *TransferActor) Receive(state TransferState, command TransferCommand) (TransferState, []Effect, error) {
  switch x := command.(type) {
  case ReceivePaymentCommand:
    newState := state.ReceivePayment(x.PaymentDetails)

    effects := []Effect{
      NewSendChecksCommandEffect(state.TransferID),
      NewTransferEvent(NewPaymentReceivedEvent(state.TransferID)),
    }

    return newState, effects, nil
  default:
    return state, nil, kafka.NewErrToxic("unknown command")
  }
}
```

Conclusion

The most simple way to solve concurrency problems is to avoid concurrency.

Q&A