Autokrator - an event sourced financial platform

Software Engineering Team D
Final Presentation - 21st March 2018

Goals

Goals

1. Create a generic event sourcing platform that enables the storage, replay and distribution of arbitrary events to multiple microservices.
2. Demonstrate this platform with a simple money transfer application, using multiple microservices.

Background

What is event sourcing?

- State of an entity is stored as a sequence of events.
- Each event modifies the state, so by replaying every event in order, you can “rebuild” the state.
- Events are stored in the event store, which acts as a database and also a message broker.
- Events are distributed out to all interested subscribers.

What is event sourcing?

Account	Balance
John	$25
SED	$1000000
Tony Hawk	-$3
Other Team	-$100000

Traditional

Deposit $5 to John’s Account

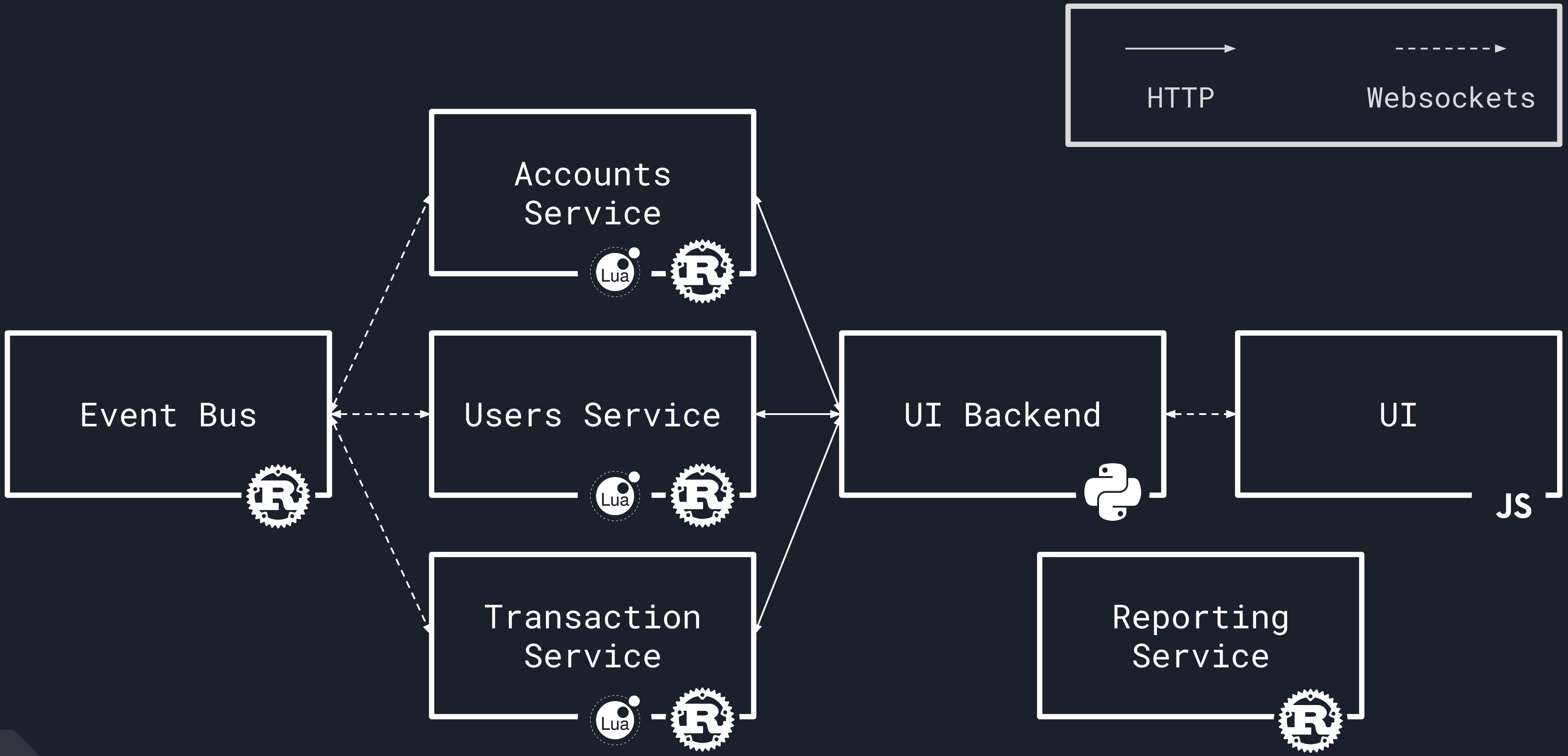
Event Sourced

Software Development Process

Software Development Process

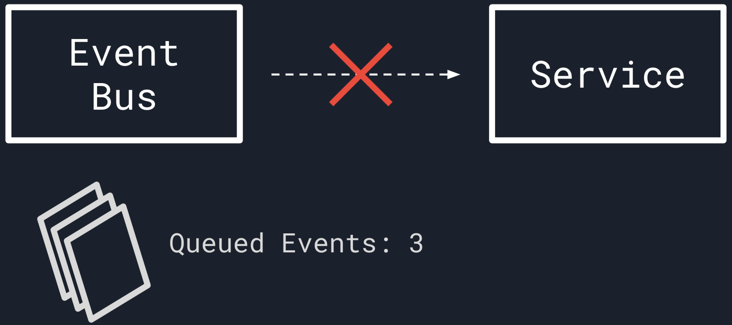
- Agile Methodology
  - Scrum sprints with two week duration.
  - Sprint planning and retrospectives.
  - Team elected a Scrum Master and Product Owner.
- Co-operative techniques:
  - Pair Programming
  - Mentored Issues
    - Issue has an assignee and a mentor who is familiar with that project/part of codebase.
    - Mentor looks at the requirements and writes up initial instructions for the assignee.
    - The assignee can ask the mentor questions if they need help.
- Continuous Integration
  - Build pipelines on every repository, running at least minimal testing on every commit.
  - Use of test coverage checkers.

System Architecture


HTTP
Websockets
Accounts Service
Lua
Event Bus
Users Service
UI Backend
UI
JS
Transaction Service
Lua
Reporting Service

Advantages and Disadvantages of our Platform

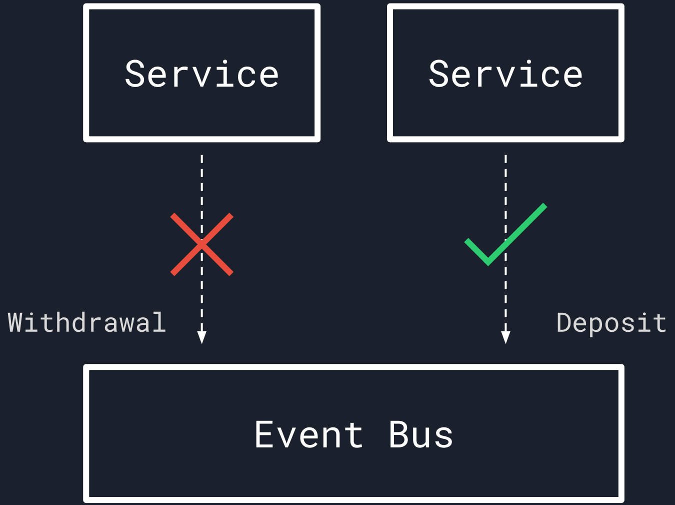
Advantages: Redelivery



Can guarantee that all events get processed at least once by each type of process.

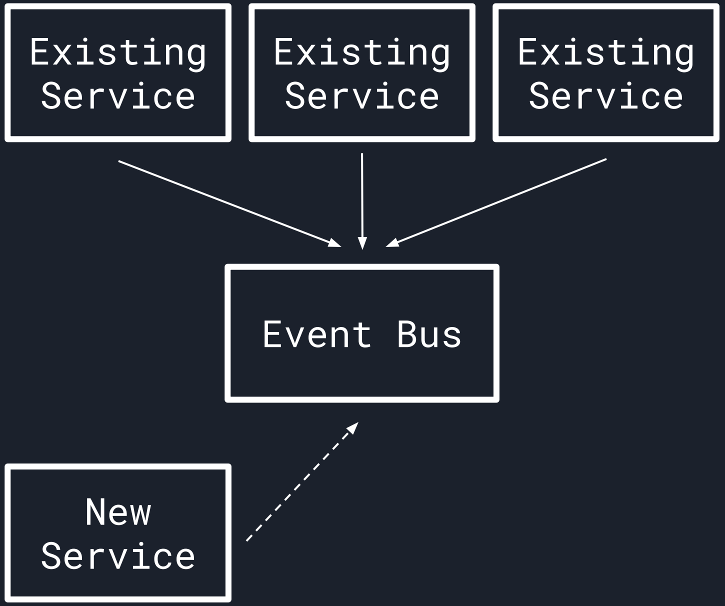
Allows for graceful service failure and recovery without manual intervention.

Advantages: Consistency



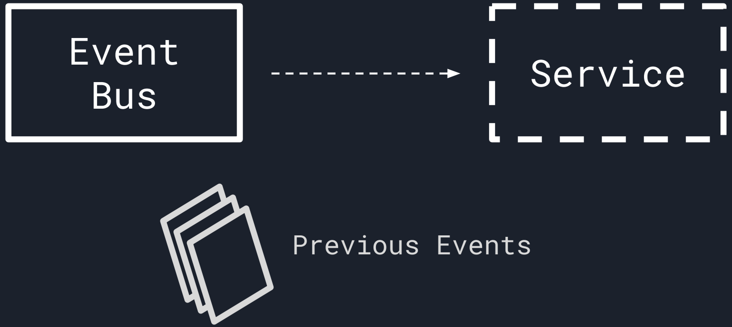
Being able to ensure that two transactions happening concurrently won’t update an account at the same time. A withdrawal being received before a deposit is processed might make the user withdrawn.

Advantage: Extensibility



Ability to add new services that interact with the existing components and their events without any modification to the existing components.

Advantage: Rebuilding State



We can destroy any of the services and their local databases and they will request that all events be resent so that they can rebuild their local state.

Advantages: Auditability

Being able to review older events and find out what happened to debug issues and track down malicious behaviour.

For example, tracing fraudulent transactions through a system.

Disadvantages

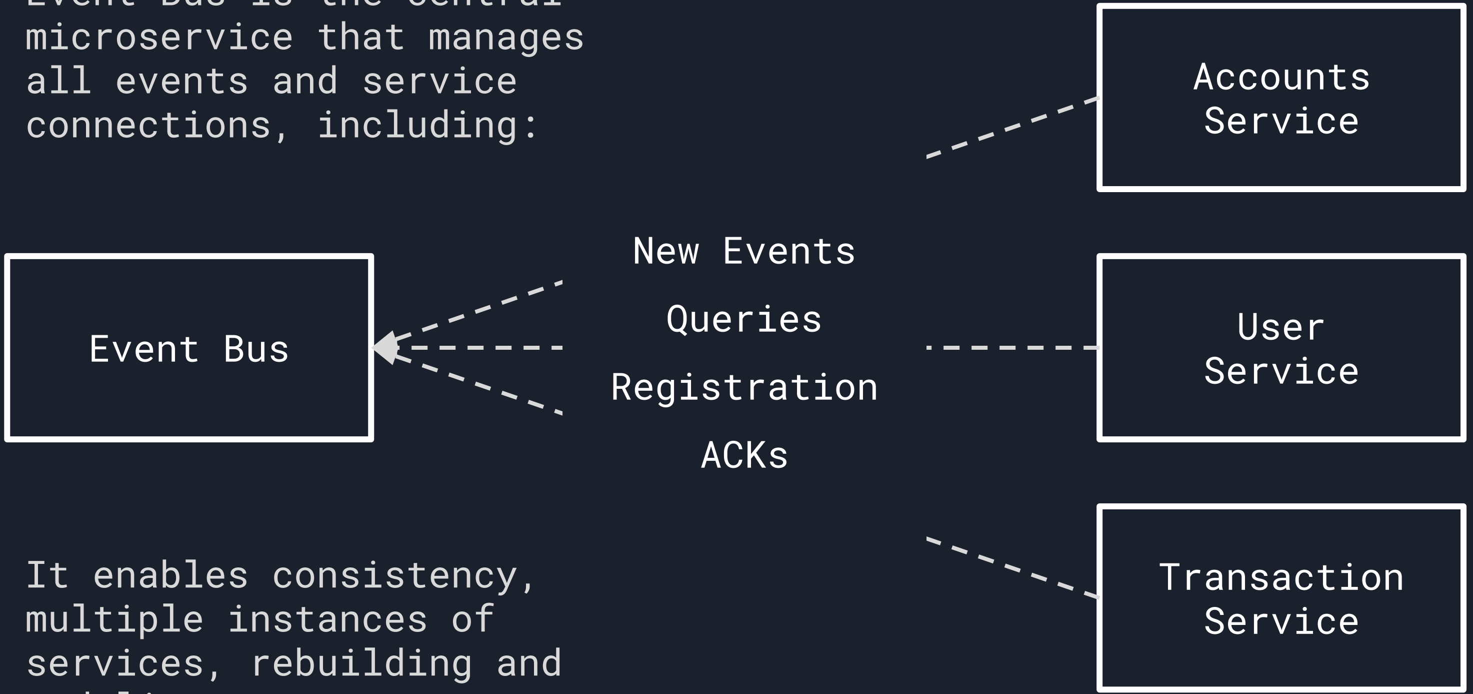
- Increased complexity in core components.
- Visibility of the overall state of the system is difficult and potentially requires looking at every event since the beginning of time.
- High resource utilisation by all components of system combined than traditional approach.
- Event Bus is the single point of failure.
  - However, this can be mitigated by making the event bus horizontally-scalable.

See it in action...

Event Bus

What is the event bus?

Event Bus is the central microservice that manages all events and service connections, including:



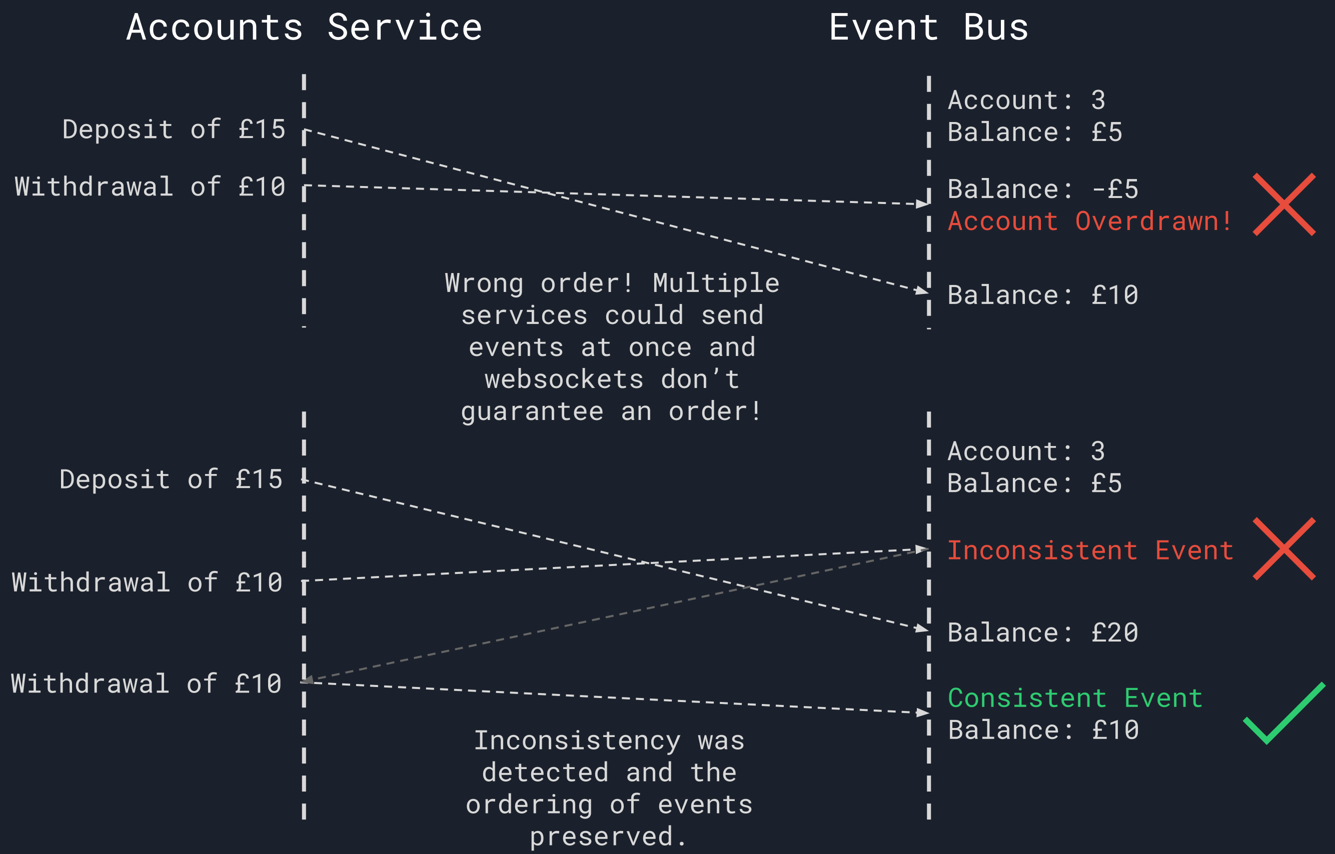
It enables consistency, multiple instances of services, rebuilding and redelivery.

Event Bus

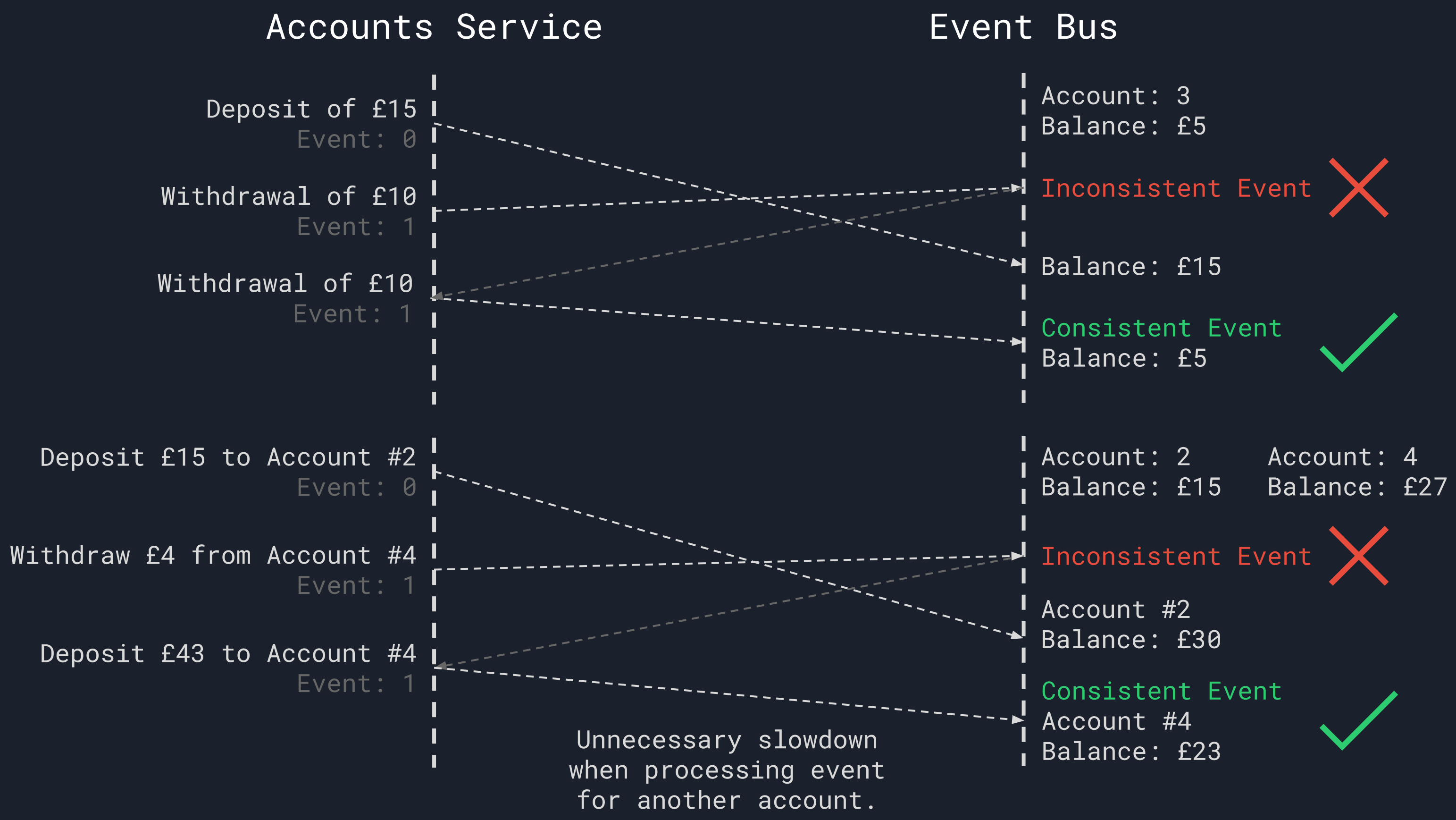
Other noteworthy things:

- Events are persisted to Couchbase (for later analysis and querying) and Kafka (as permanent event storage).
- Implemented with an Actor architecture.
- Fully asynchronous and multithreaded.

Event Bus: Consistency



Why we need consistency



Accounts Service

Event Bus

Deposit of £15
Event: 0

Withdrawal of £10
Event: 1

Withdrawal of £10
Event: 1

Account: 3
Balance: £5

Inconsistent Event ✗

Balance: £15

Consistent Event ✓
Balance: £5

Deposit £15 to Account #2
Event: 0

Withdraw £4 from Account #4
Event: 1

Deposit £43 to Account #4
Event: 1

Account: 2
Balance: £15

Account: 4
Balance: £27

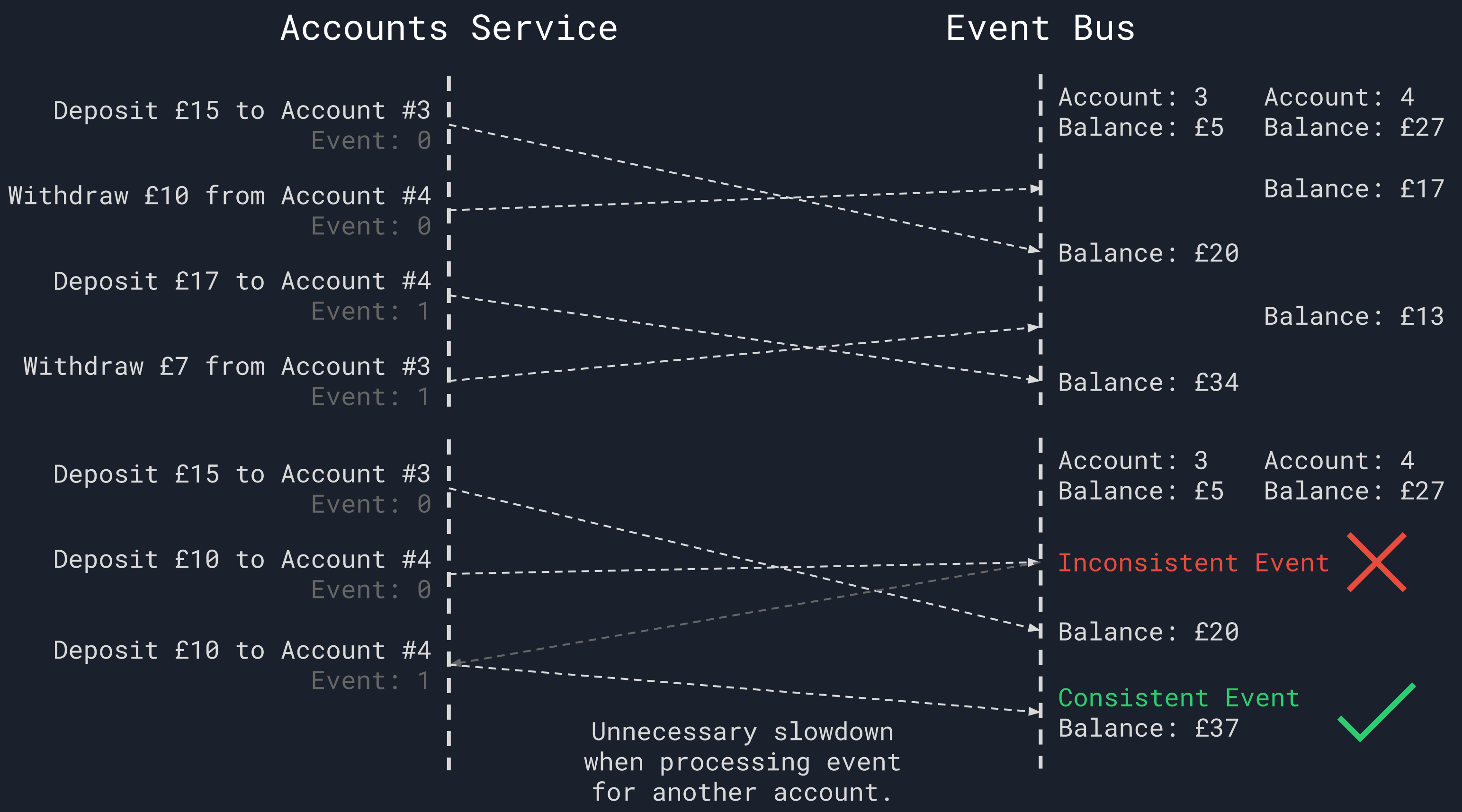
Inconsistent Event ✗

Account #2
Balance: £30

Consistent Event ✓
Account #4
Balance: £23

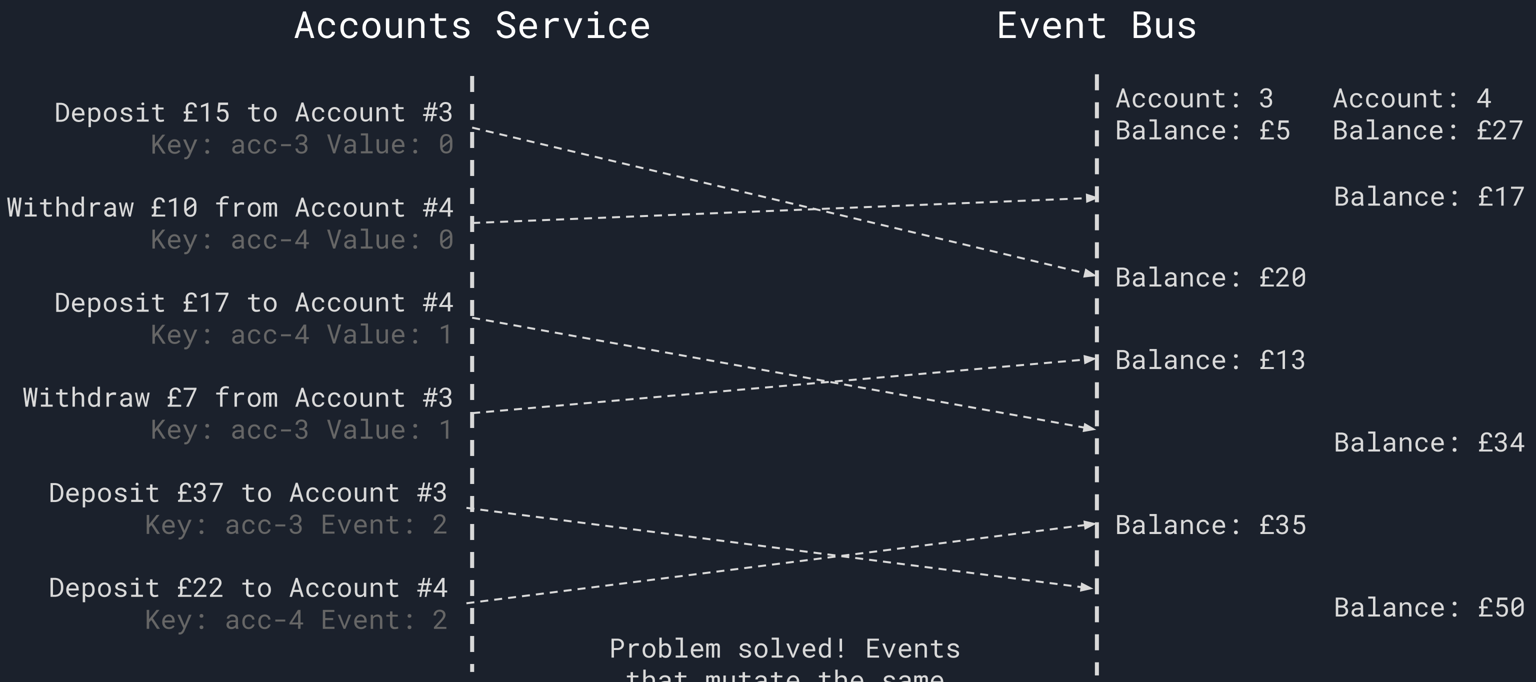
Unnecessary slowdown
when processing event
for another account.

Naive Consistency: Global Ordering with previous event hash/number



Consistency Attempt 2: Ordering per event type with previous event hash/number

Consistency Final Implementation: Sequence Key/Value



Accounts Service

Event Bus

Deposit £15 to Account #3
Key: acc-3 Value: 0

Withdraw £10 from Account #4
Key: acc-4 Value: 0

Deposit £17 to Account #4
Key: acc-4 Value: 1

Withdraw £7 from Account #3
Key: acc-3 Value: 1

Deposit £37 to Account #3
Key: acc-3 Event: 2

Deposit £22 to Account #4
Key: acc-4 Event: 2

Account: 3
Balance: £5

Account: 4
Balance: £27

Balance: £17

Balance: £20

Balance: £13

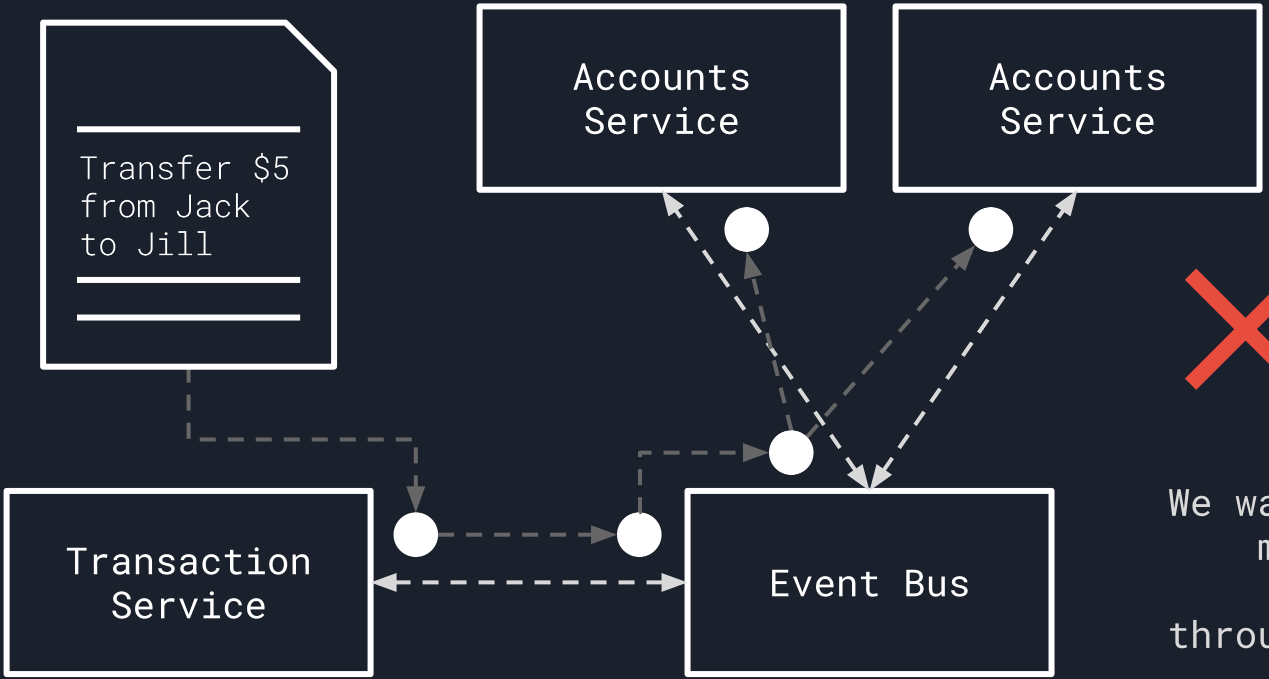
Balance: £34

Balance: £35

Balance: £50

Problem solved! Events that mutate the same state have forced ordering while not slowing down the entire system.

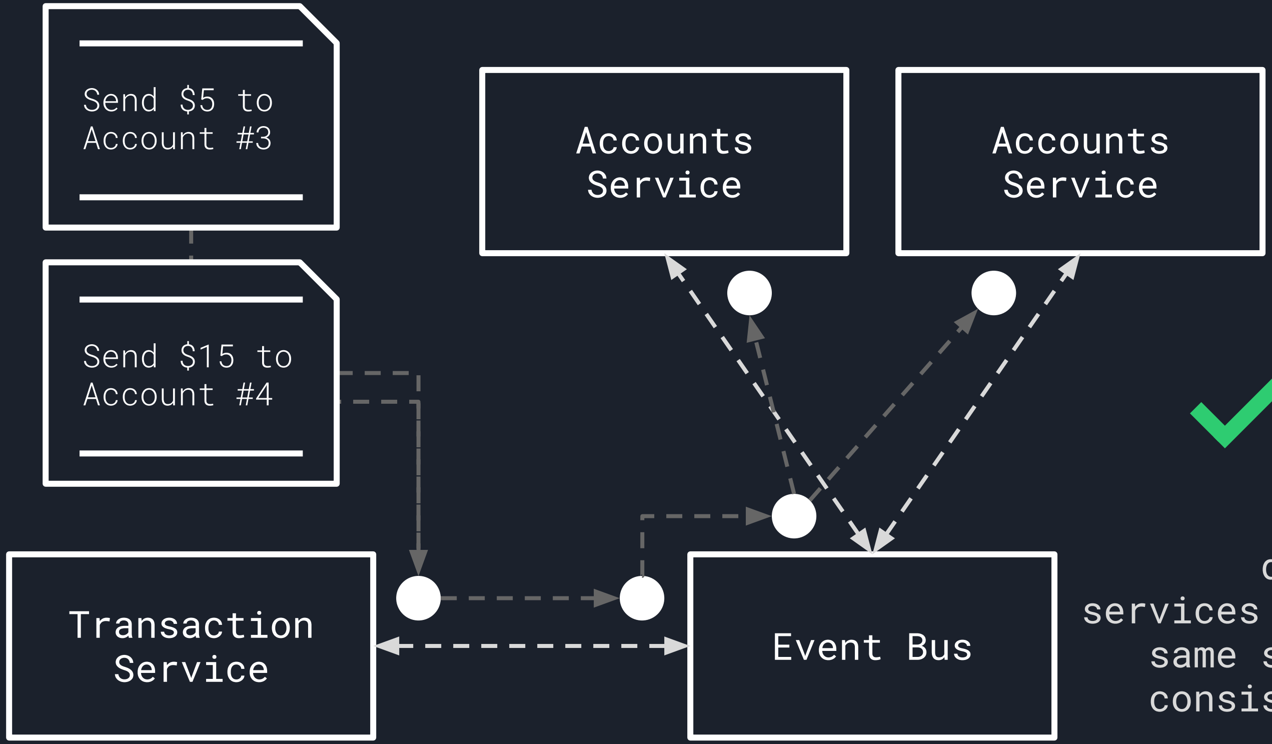
Event Bus:
Sticky Round Robin



Duplicate processing of events!

We want to be able to run multiple instances of services to increase throughput of the system.

Why we need sticky round robin - the multiple instances problem



✓ No more duplicate processing of events!

Sticky round robin distributes events to services but ensures that the same service gets the same consistency key each time.

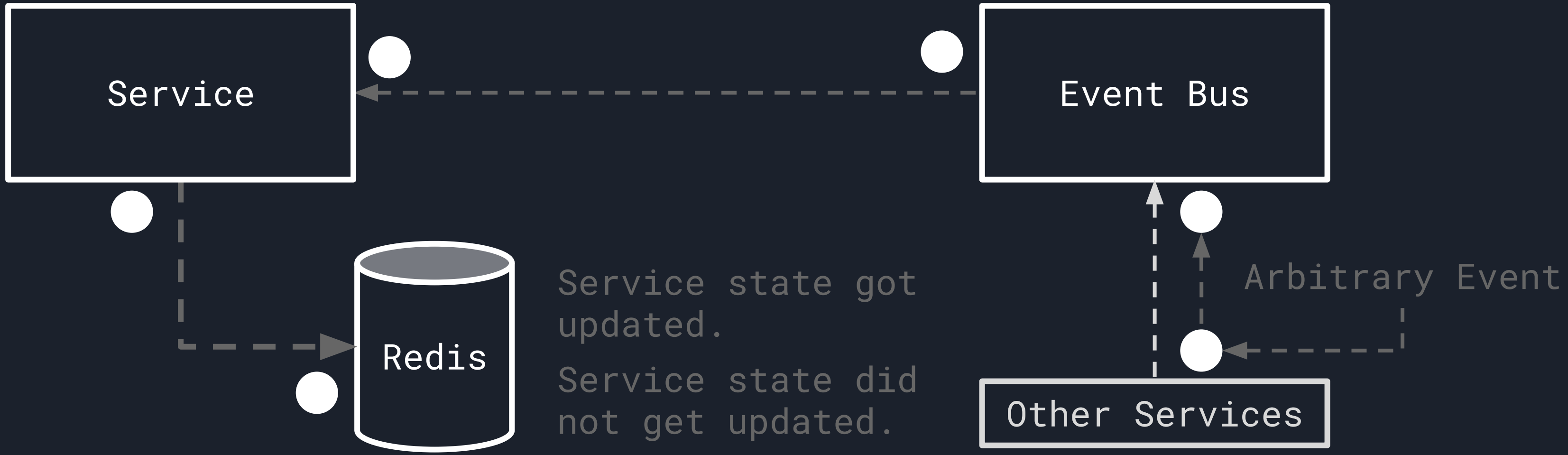
Sticky round robin - our solution to the multiple instances problem

Event Bus:
ACKs and Redelivery

✗ Event processing failed!

? Event processing in progress!

✓ Event processing successful!



Service

Event Bus

Redis

Other Services

Service state got updated.

Service state did not get updated.

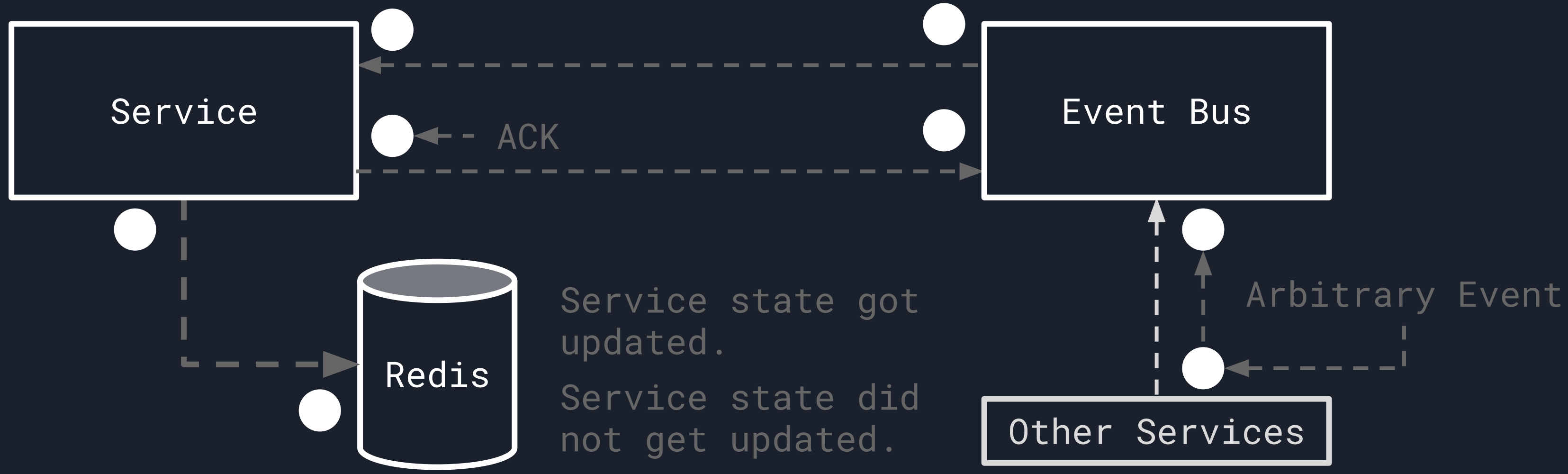
Arbitrary Event

Why do we need redelivery?

✗ Event processing failed!

? Event processing in progress!

✓ Event processing successful!



Service

Event Bus

ACK

Redis

Other Services

Arbitrary Event

Service state got updated.

Service state did not get updated.

ACK sent, event bus knows not to re-send.

No ACK sent, event bus will send to another instance or save for later.

How do we implement redelivery?

Superclient

Superclient

The superclient is a framework for building microservices that communicate with the event bus.

It embeds the Lua programming language and exposes an API for services to create HTTP routes, process incoming events, send events, save/load state to Redis and manage rebuilding of the state.

Services are written in small Lua scripts that only contain the business logic for that service, improving maintainability and speed of iterations and bug fixes.

The superclient made implementation of rebuilding and redelivery simpler than the previous Java versions of the services while being more maintainable - almost ½ as much code.

Superclient

```
-- Return account balance.
bus:add_route("/account/{id}", "GET", function(method, route, args, data)
    log:debug("received " .. route .. " request")

    -- Get the information we have stored about this account.
    local account = redis:get(PREFIX .. args.id)
    if account then
        -- Return some of the data.
        return HTTP_OK, { id = account.id, balance = account.balance }
    else
        -- Return an error if we do not have data.
        return HTTP_NOT_FOUND, { error = "could not find account with id: " ..
args.id }
    end

end)
```

Superclient

```lua
-- Handle request for an account creation.
bus:add_event_listener("AccountCreationRequest", function(event_type, key,
correlation, data)
    log:debug("received " .. event_type .. " event")
    -- Get the next ID.
    local last_id = redis:get(ID_KEY)
    local next_id = last_id.id + 1
    redis:set(ID_KEY, { id = next_id })

    -- Create a new account and send the event out.
    create_account(next_id ,data.request_id, true)
end)
```

Superclient

The superclient replaced a client library and three services written in Java.

Superclient:

- Contains HTTP server, Websocket client, Redis client, Lua interpreter.
- Handles consistency, rebuilding, redelivery.
- Approximately 1,600 lines of superclient and 100-200 lines per service (x3).

Previous Java Version:

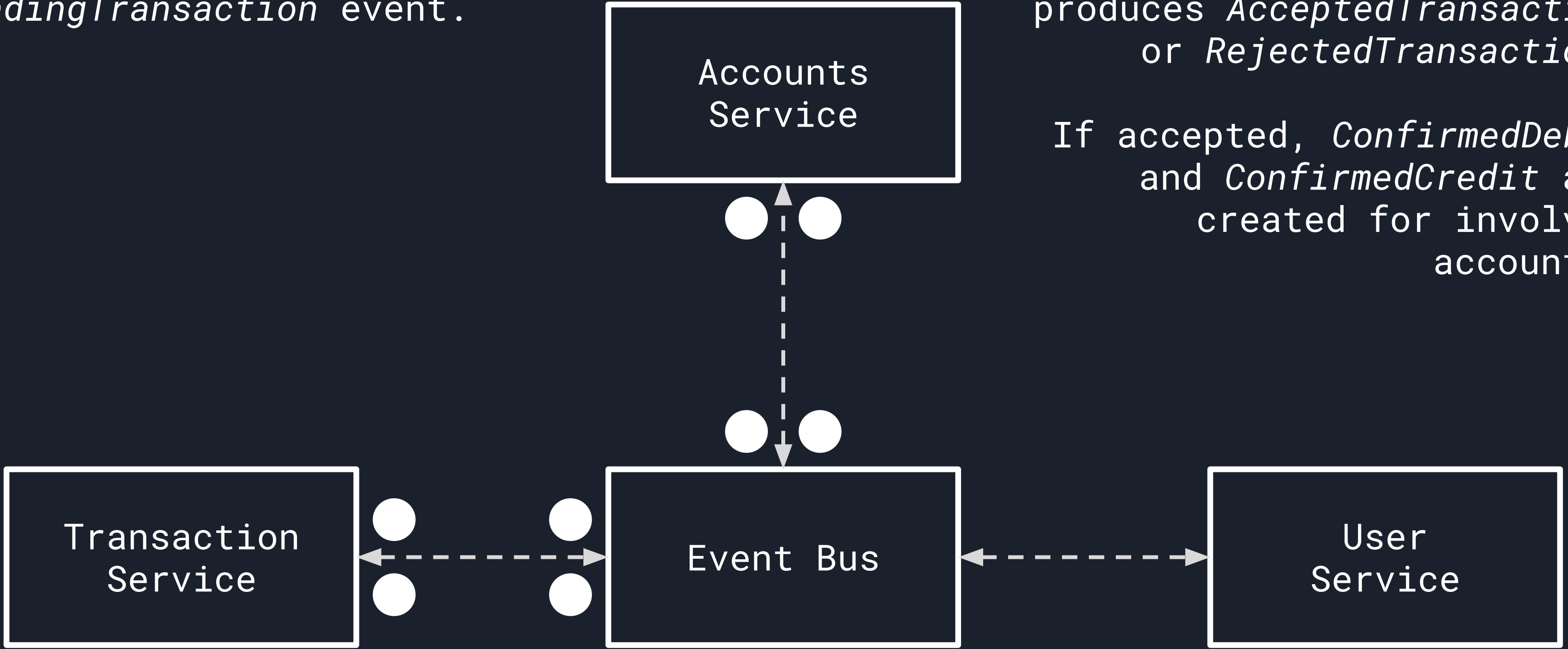
- Contains HTTP server, Websocket client and PostgreSQL client.
- Handles consistency.
- Approximately 1,900 lines of client library and 600-800 lines per service (x3).

Services

1. Transaction Service produces a *PendingTransaction* event.

2. Accounts Service checks for sufficient balance and produces *AcceptedTransaction* or *RejectedTransaction*.

If accepted, *ConfirmedDebit* and *ConfirmedCredit* are created for involved accounts.



Example: Creating a transaction

User Service

- Handles creation/registration of user accounts.
- Allows users to request creation of money accounts, but delegates the actual creation to the Accounts Service.
- Maintains the mapping between user accounts and money accounts.

UI

BfaF - “Backend for a Frontend” and UI

- BfaF acts as a proxy layer for the microservices - a gateway that the UI can call to talk to multiple different backend services.
- Processes and formats data so that the UI code can be simplified.
- Provides real-time updates to the UI via WebSockets.

- UI is written in React.js.
- Single Page Application - only once loaded from the server.
- Redux as the data model, which uses a global state, modified by Reducer functions, to coordinate the application.

Reporting Service

- Small Python/Flask utility for browsing events by correlation and consistency information.
  - Useful for debugging potential bugs or flaws.
  - Useful for tracing fraudulent transactions and malicious events.
- Queries events in couchbase directly - does not communicate with event bus!

Adding a new service to the platform

Introducing the Nectar Service

If you spend over £100, you will receive a percentage of your spend as nectar points.

If you spend over £25, and have enough nectar points, you will receive cashback.

```lua
-- This service is a "nectar" service - used within sticky round
-- robin to distribute events.
bus:register("nectar")

-- Declare that will be used as part of the consistency key
-- for nectar-related events.
local PREFIX = "nectar-"

-- When £100 or more is spent, we debit the nectar balance by
-- 10% of this amount.
local GET_THRESHOLD = 100.0
local GET_AMOUNT = 0.1

-- If greater than £25.0, if the nectar account has sufficient balance for the
-- amount of the transaction, then 10% of the amount is refunded as a debit.
local USE_THRESHOLD = 25.0
local USE_AMOUNT = 0.1
```

Nectar Service (1 / 10)

```lua
-- Register for events of a type.
bus:add_event_listener("AcceptedTransaction", function(event_type, key, correlation, data)
    -- Calculate the nectar consistency key the account involved in this transaction.
    local nectar_key = PREFIX .. data.from_account_id
    -- Fetch the account details
    local account = redis:get(nectar_key)









    -- ...
end)
```

Nectar Service (2 / 10)

```lua
-- Register for events of a type.
bus:add_event_listener("AcceptedTransaction", function(event_type, key, correlation, data)
    -- Calculate the nectar consistency key the account involved in this transaction.
    local nectar_key = PREFIX .. data.from_account_id
    -- Fetch the account details
    local account = redis:get(nectar_key)

    if data.amount > GET_THRESHOLD then
        log:info("received " .. event_type .. " and debiting nectar account")
        -- Credit nectar equal to GET_AMOUNT% of the transaction value.
        bus:send("NectarCredit", nectar_key, false, correlation, {
            amount = data.amount * GET_AMOUNT
        })
    end



    -- ...
end)
```

Nectar Service (3 / 10)

```lua
  -- ...

  -- Only give cashback if this account has enough points.
  if data.amount > USE_AMOUNT and account and account.balance > data.amount then
    log:info("received " .. event_type .. " and crediting nectar account")

    -- Take the points away equal to the value of the transaction.
    bus:send("NectarDebit", nectar_key, false, correlation, { amount = data.amount })

    -- Credit GBP equal to USE_AMOUNT% of the transaction value.
    local spent = "Spent " .. data.amount .. " points"
    local remaining = account.balance - data.amount .. " remaining."
    bus:send("ConfirmedCredit", "acc-" .. data.from_account_id, true, correlation, {
      id = data.from_account_id,
      amount = data.amount * USE_AMOUNT,
      note = "Nectar cashback! " .. spent .. ", " .. remaining,
    })
  end

end)
```

Nectar Service (4 / 10)

```lua
bus:add_event_listener("NectarDebit", function(event_type, key, correlation, data)
    local details = redis:get(key)
    if details then
        log:info("received " .. event_type .. " and updating nectar account")
        -- If the account already exists, then remove to the balance.
        details.balance = details.balance - data.amount
        redis:set(key, details)
    else
        -- If the account doesn't exist, then we can't take balance away.
        log:warn("received " .. event_type .. " without nectar account")
    end
end)
```

Nectar Service (5 / 10)

```lua
bus:add_event_listener("NectarCredit", function(event_type, key, correlation, data)
    local details = redis:get(key)
    if details then
        log:info("received " .. event_type .. " and updating nectar account")
        -- If the account already exists, then add to the balance.
        details.balance = details.balance + data.amount
        redis:set(key, details)
    else
        log:info("received " .. event_type .. " and creating nectar account")
        -- If the account doesn't exist, create it with the new balance.
        redis:set(key, { balance = data.amount })
    end
end)
```

Nectar Service (6 / 10)

```lua
function handle_receipt(status, event_type, key, correlation, data)
    log:debug("received " .. event_type .. " receipt")
    -- Resend the event.
    if status == "inconsistent" then
        bus:send(event_type, key, event_type == "ConfirmedCredit", correlation, data)
    end
end

-- Only handle the receipts for event types that this service sends out.
bus:add_receipt_listener("NectarDebit", handle_receipt)
bus:add_receipt_listener("NectarCredit", handle_receipt)
bus:add_receipt_listener("ConfirmedCredit", handle_receipt)
```

Nectar Service (7 / 10)

```
function handle_balance_change(event_type, key, correlation, data)
    log:debug("received " .. event_type .. " rebuild")
    local account = redis:get(key)
    if account then
        -- Update the balance to add the amount credited. data.amount should be negative if this
        -- is a debit.
        account.balance = account.balance + data.amount

        -- Save these changes.
        redis:set(key, account)
    else
        -- Create an account with this amount.
        redis:set(key, { balance = data.amount })
    end
end

bus:add_rebuild_handler("NectarDebit", handle_balance_change)
```

Nectar Service (8 / 10)

```lua
function handle_balance_change(event_type, key, correlation, data)
    log:debug("received " .. event_type .. " rebuild")
    -- ...
end

bus:add_rebuild_handler("NectarCredit", function(event_type, key, correlation, data)
    -- The NectarDebit event has a positive value so negate this so that the same function
    -- can handle both credit and debit balance changes.
    data.amount = -data.amount
    handle_balance_change(event_type, key, correlation, data)
end)
```

Nectar Service (9 / 10)

```
bus:add_route("/balance/{id}", "GET", function(method, route, args, data)
    log:debug("received " .. route .. " request")
    -- Get the account details.
    local key = PREFIX .. args.id
    local account = redis:get(key)
    if account then
        -- Return the balance.
        return HTTP_OK, { balance = account.balance }
    else
        -- Return an error if we don't have data.
        return HTTP_NOT_FOUND, { error = "could not find nectar account with id: " .. args.id }
    end
end)
```

Nectar Service (10 / 10)

See it in action…

Summary

We built...

- an Event Bus for consistent distribution of messages to multiple services;
- a Superclient framework for easy building of new microservices in Lua;
- and a Demo application with multiple microservices to prove the concept of Event Sourcing as a viable solution.

Find it on GitHub at https://github.com/autokrator-uog and GitLab at https://gitlab.com/autokrator-uog.

Event sourcing is a viable architecture for building applications.

There is an overhead in complexity and the requirement to build an event bus and derive solutions to consistency, redelivery and rebuilding.

However, in larger systems with more moving parts that overhead is small compared to the various advantages such as auditability and extensibility.

Any questions?