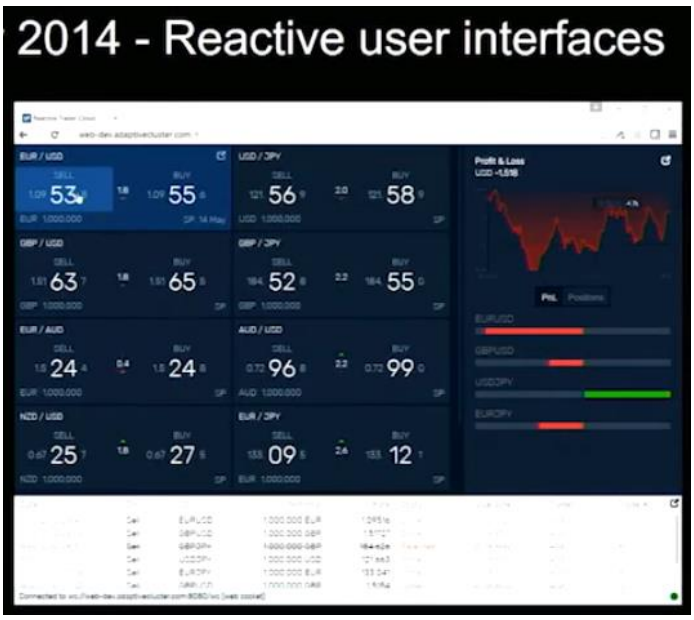
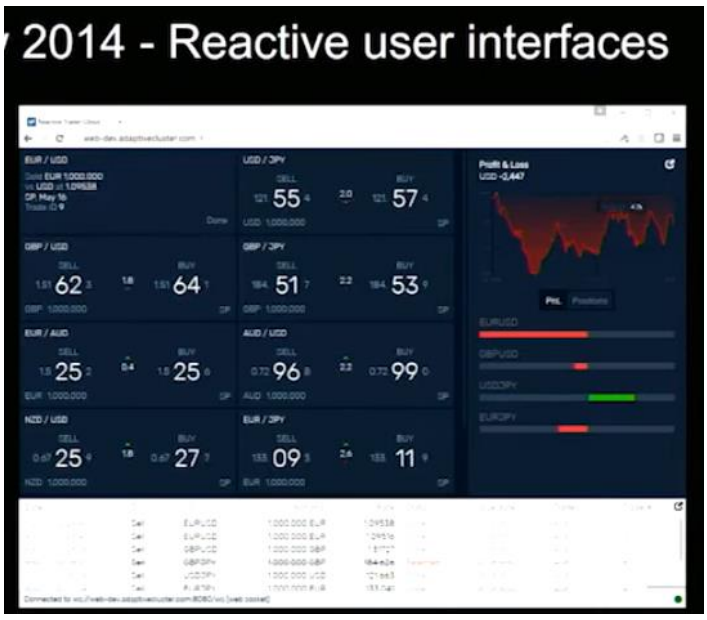
# OLIVIER DEHEURLES / JAMES WATSON

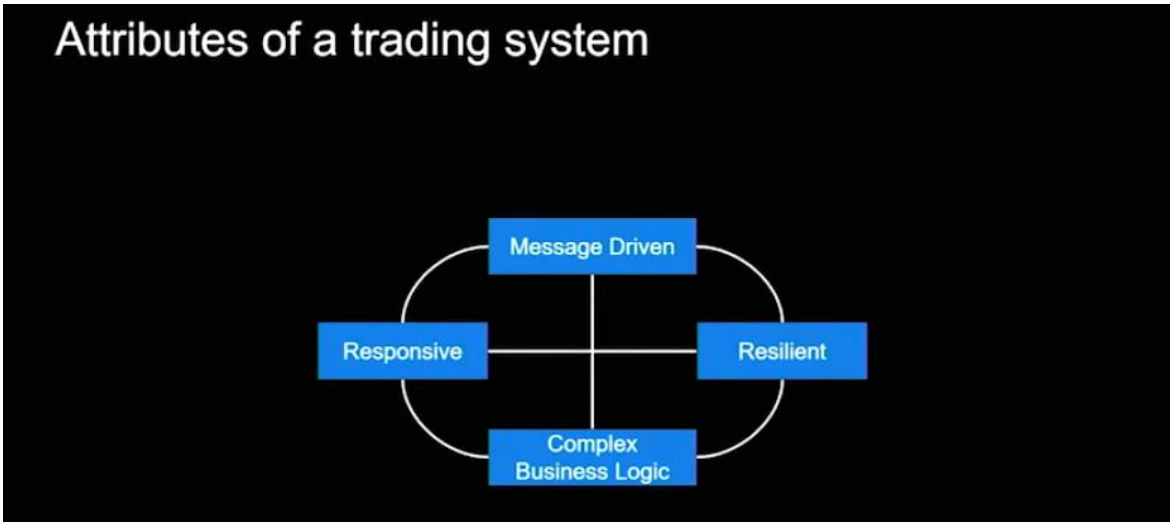
## CLUSTERED EVENT-DRIVEN SERVICES ARCHITECTURE + DESIGN

In 2010 I came across a strange, new (to me!) architecture that the LMAX team used for their Foreign Exchange system. You might have heard about the Disruptor; it came out of this project. The core of our system is a clustered service which uses the Raft consensus algorithm to reliably replicate state between the different nodes and hosts our application logic. We will take a quick look at Raft and then at the benefits of this design compared to more "mainstream" architectures.




This architecture offers a clean separation of concerns between the infrastructure – which takes care of the concurrency, I/O and high availability aspects – and the application logic. The clean architecture is a great fit for domain-driven design. If you fancy building fast, resilient services without a database you should come to this talk. Our UIs and the backend systems are constantly sending messages between each other back and forth repeatedly in milliseconds.



## The problem

**Complex** multi-threaded programs

⇒ Hard to **reason about**

⇒ Hard to **debug**

## The Solution - LMAX Architecture

**Blueprint** for reactive services

Clean **separation of concerns**
- technical requirements
- Business logic

Keep the business logic as **simple** as possible

Very easy to debug

Dave Farley

The business logic runs on a single thread in this architecture.

## Our journey to the LMAX Architecture

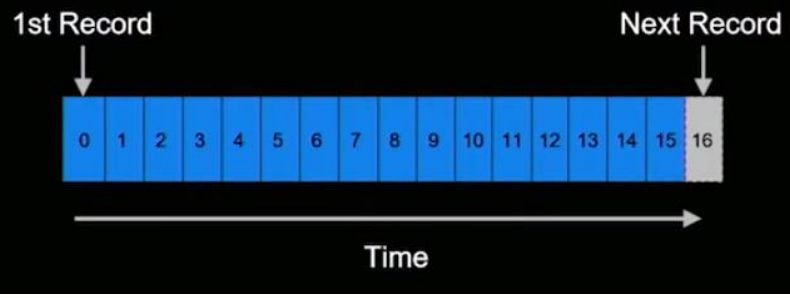
- Stage 1
  - Simple Model
  - Easy To Debug
- Stage 2
  - Transparent fault tolerance
  - Clustering with consistent replication
- Stage 3
  - Durability without a database

# Deterministic execution in a distributed event system

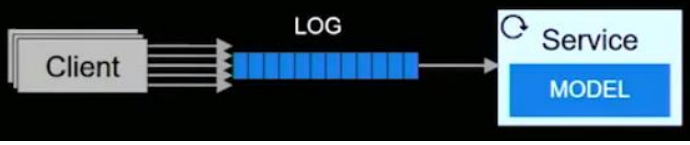
## Determinism

Given an **initial state** and a **command**, a deterministic model will always produce the same **output state** and **side-effect(s)**


## Log as Sequencing Mechanism



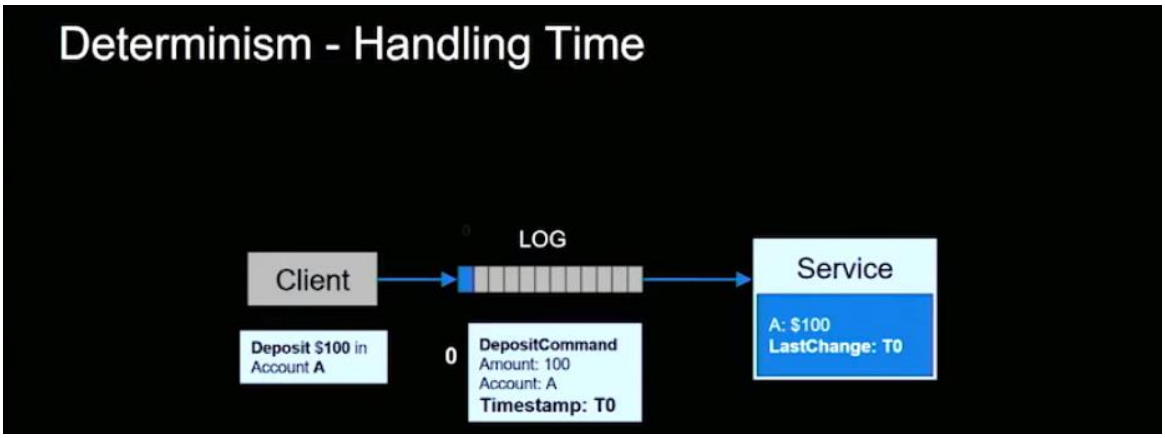

## Single Threaded Log Consumer



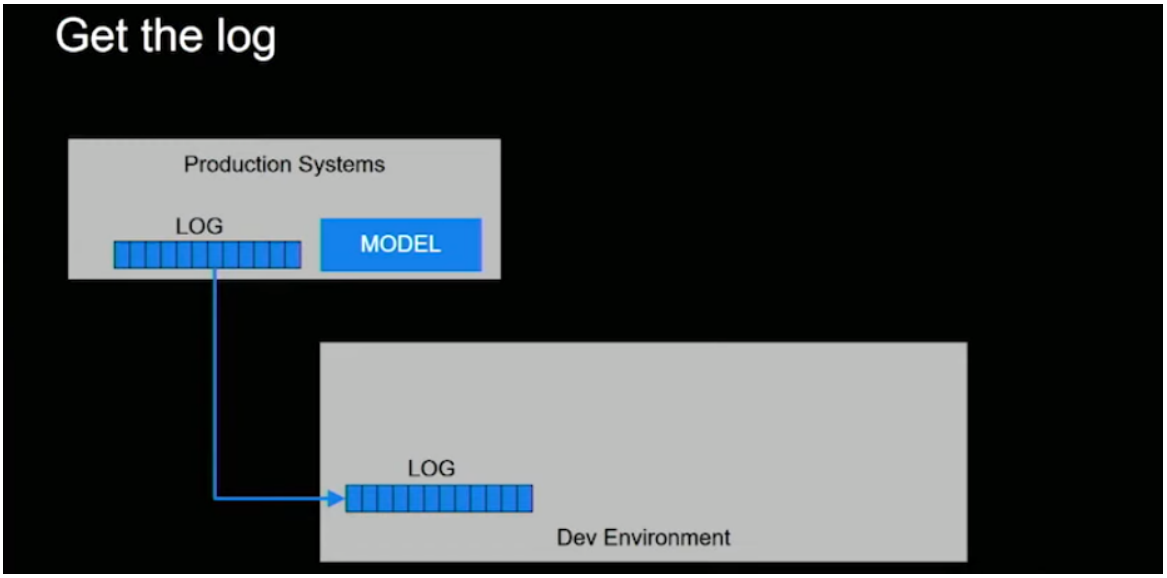

## Single threaded, isn't that slow? No!

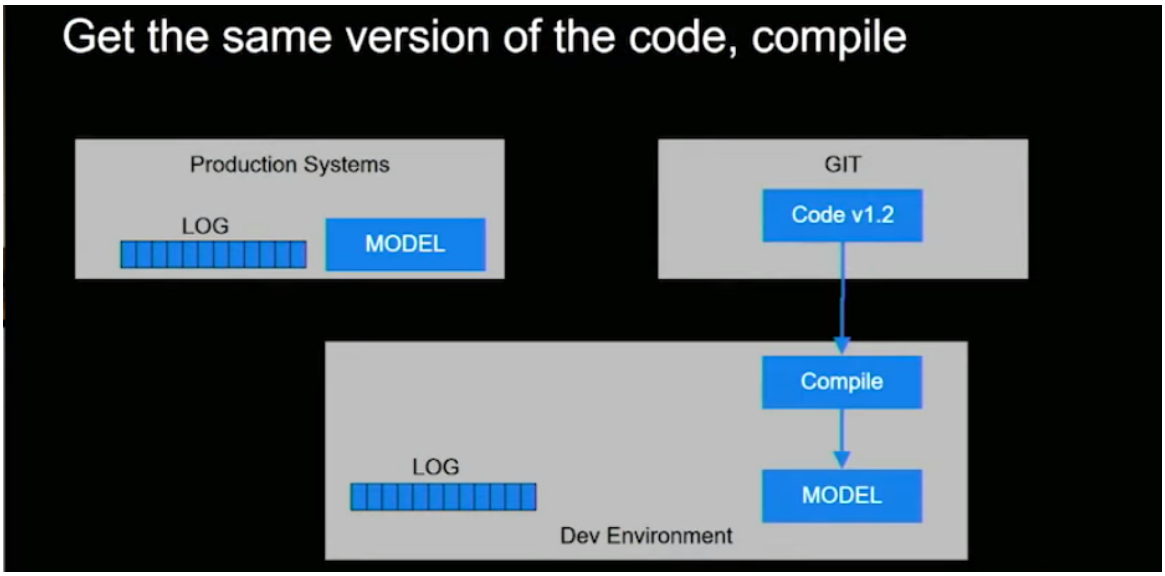
10K ops/sec: quite easy to achieve with sensible code
100K ops/sec: some profiling required
1M+ ops/sec:

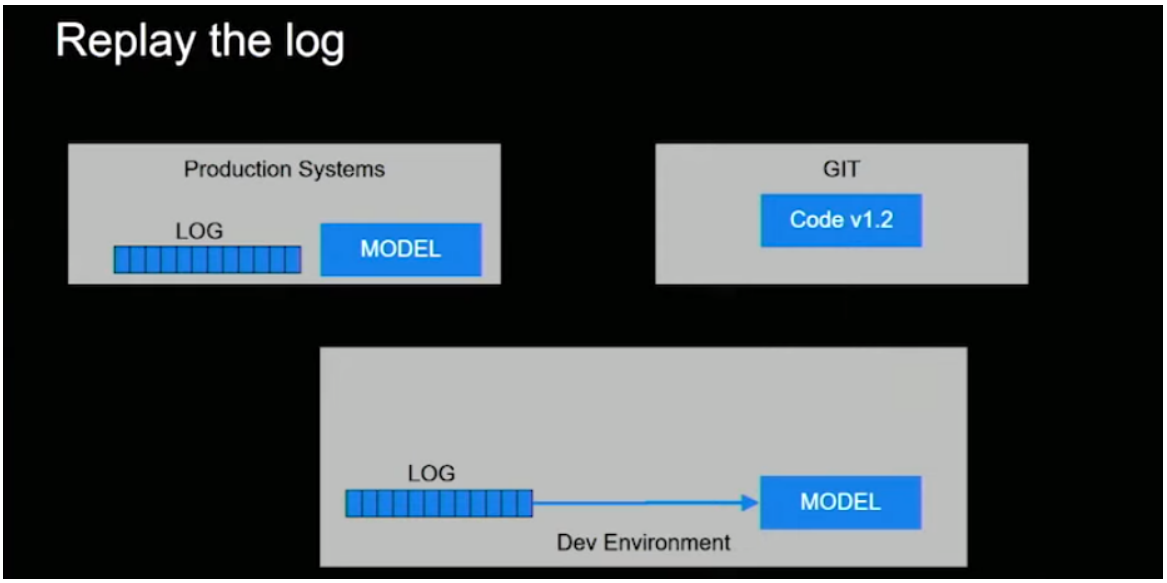
- Optimised data structures
- Low complexity algorithms - O(1), O(log(N))
- Minimise allocation

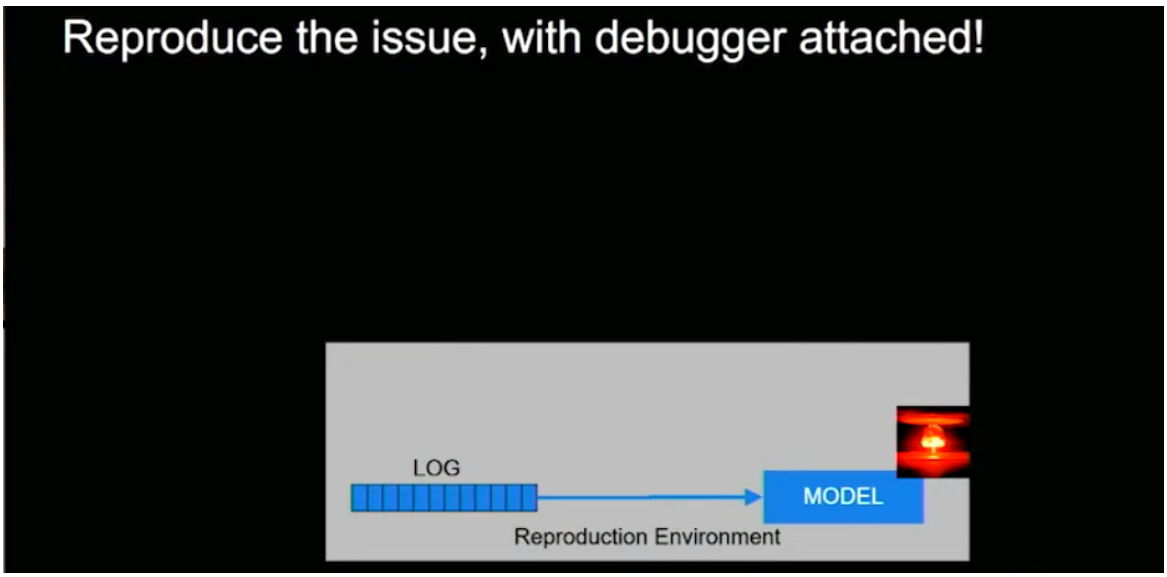



We add the timestamp of the pickup time to the message

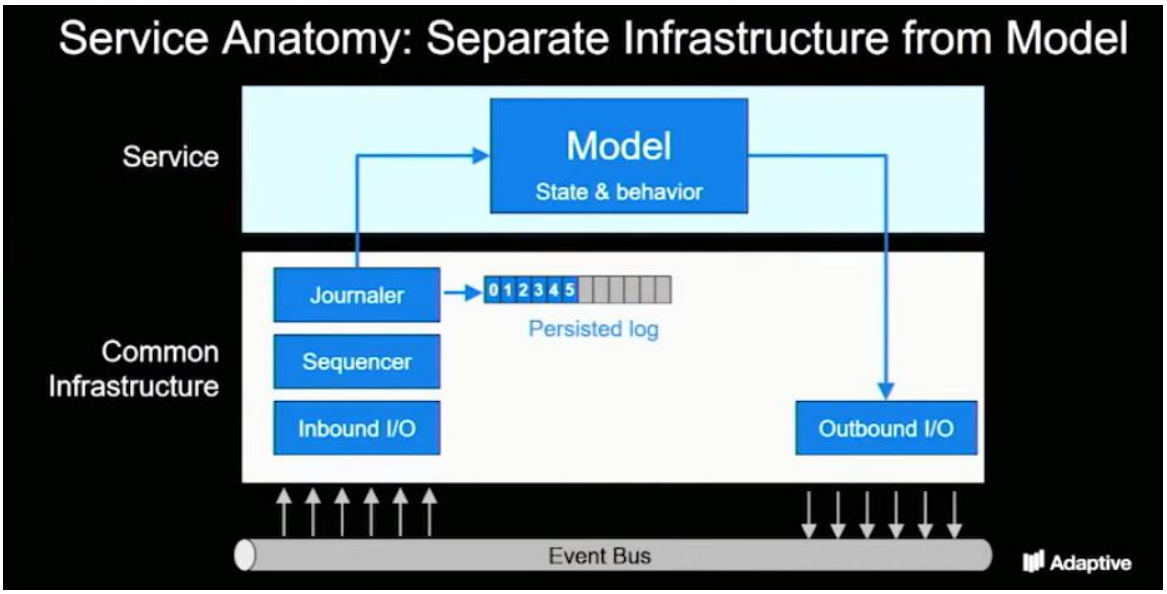



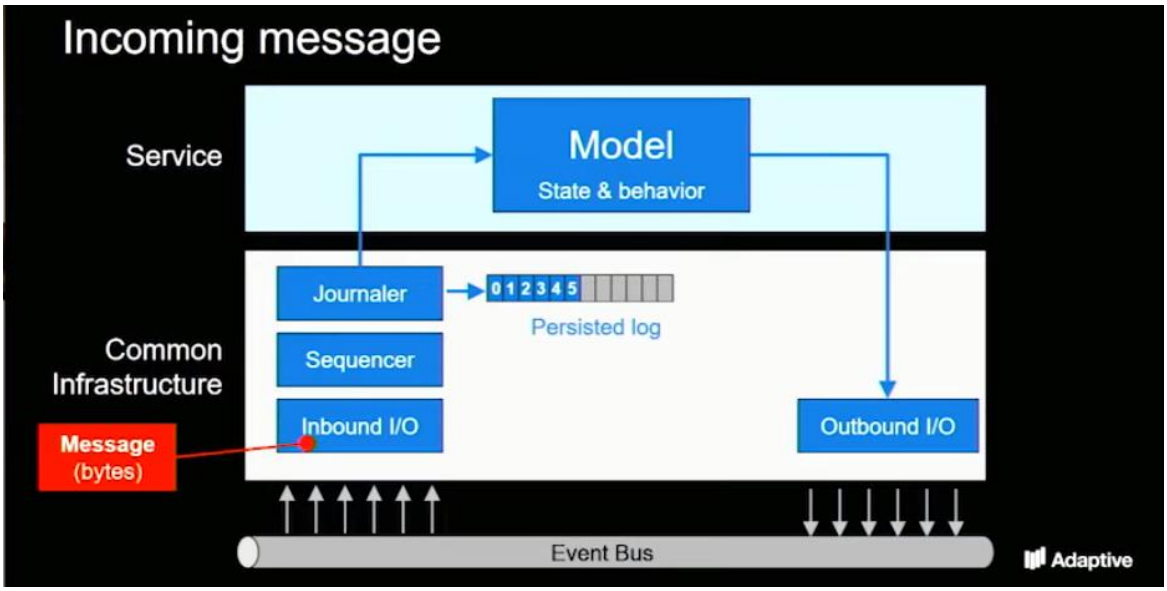


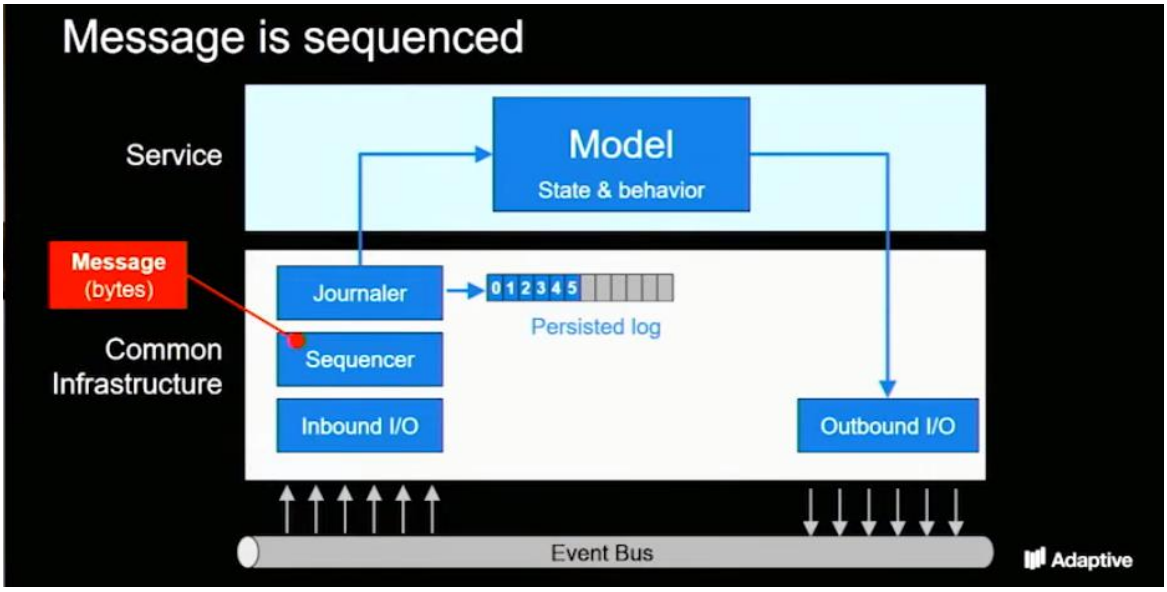



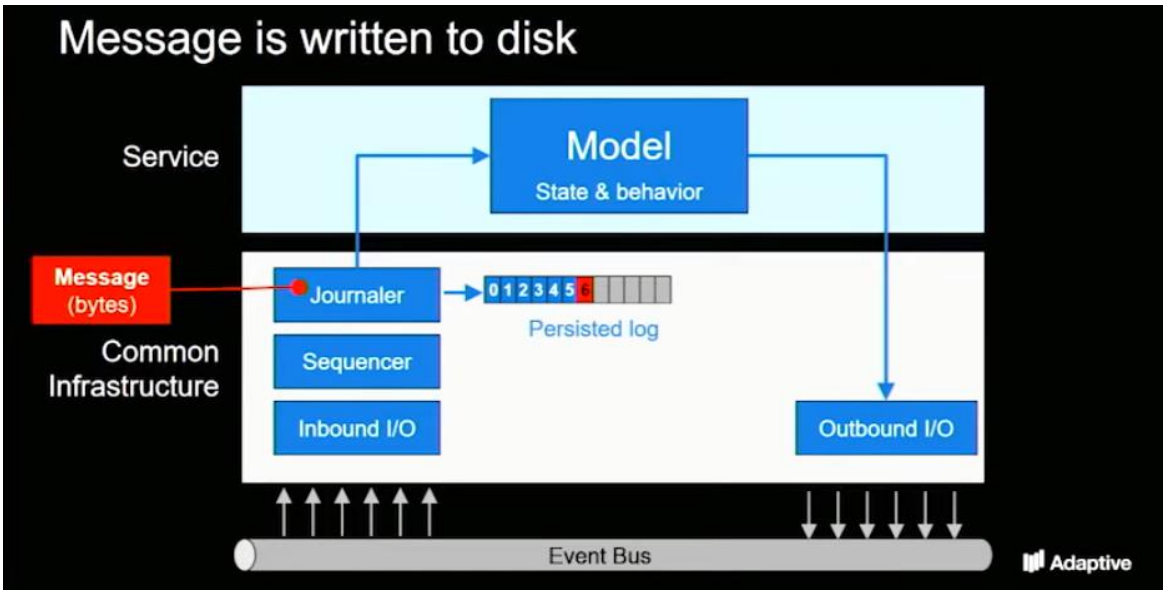



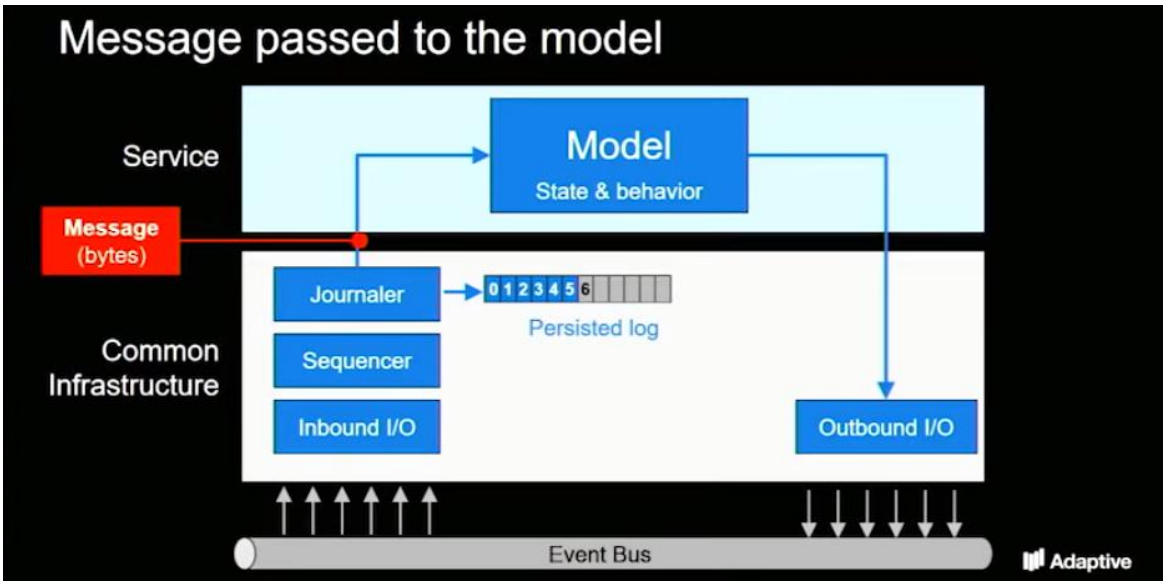

# Service Anatomy

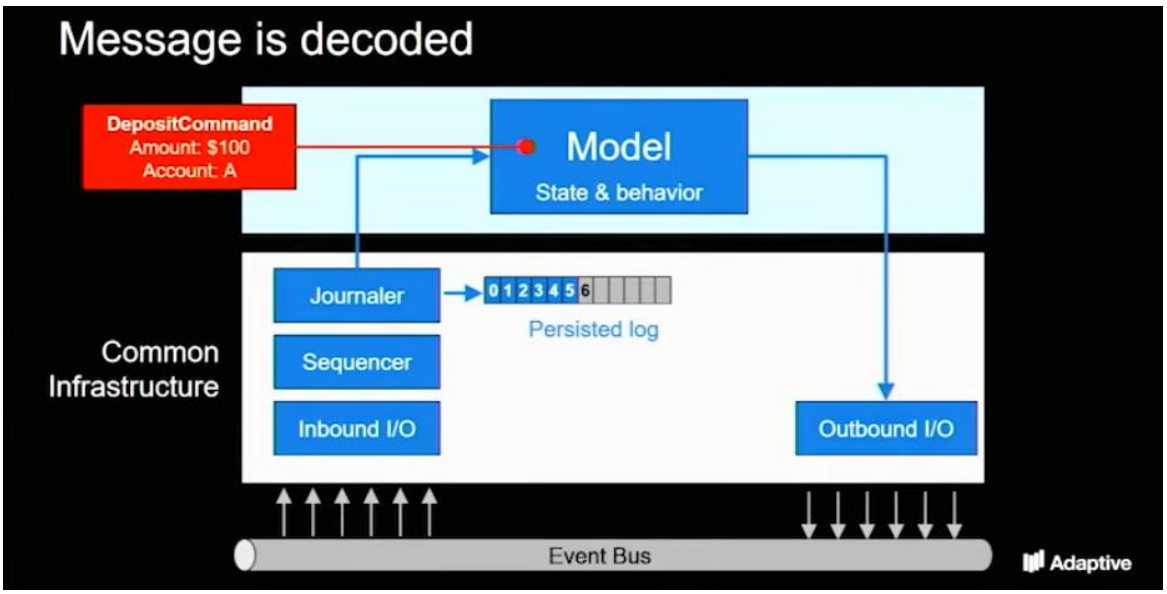

Service Anatomy: Separate Infrastructure from Model
Service
Model
State & behavior
Journaler
0 1 2 3 4 5
Persisted log
Common Infrastructure
Sequencer
Inbound I/O
Outbound I/O
Event Bus
Adaptive

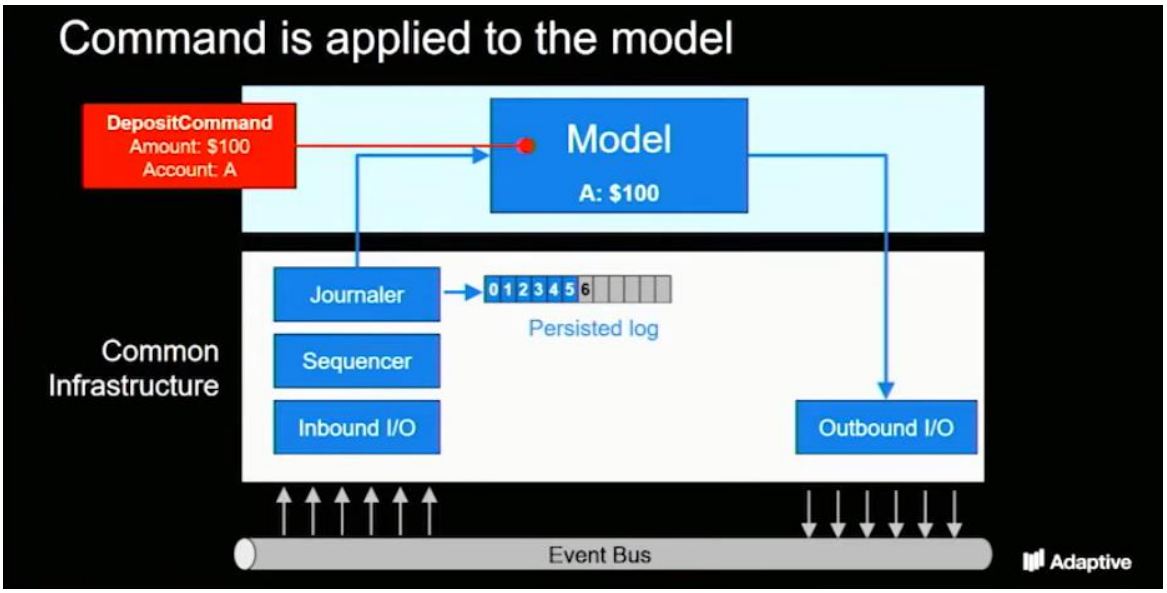

Incoming message
Service
Model
State & behavior
Journaler
0 1 2 3 4 5
Persisted log
Common Infrastructure
Sequencer
Inbound I/O
Outbound I/O
Message (bytes)
Event Bus
Adaptive

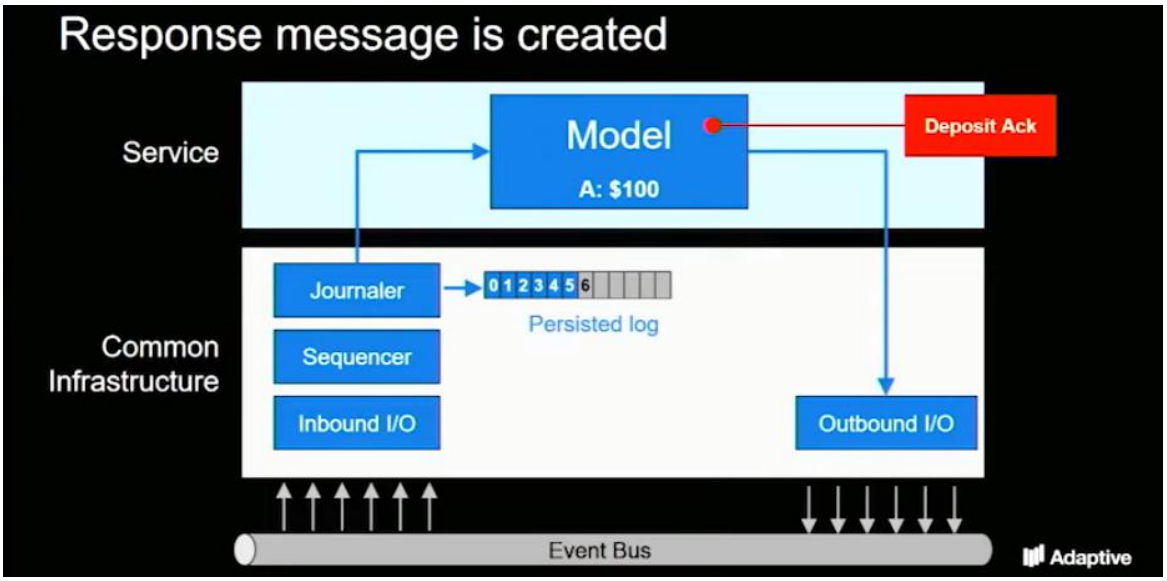

Message is sequenced
Service
Model
State & behavior
Message (bytes)
Journaler
0 1 2 3 4 5
Persisted log
Common Infrastructure
Sequencer
Inbound I/O
Outbound I/O
Event Bus
Adaptive

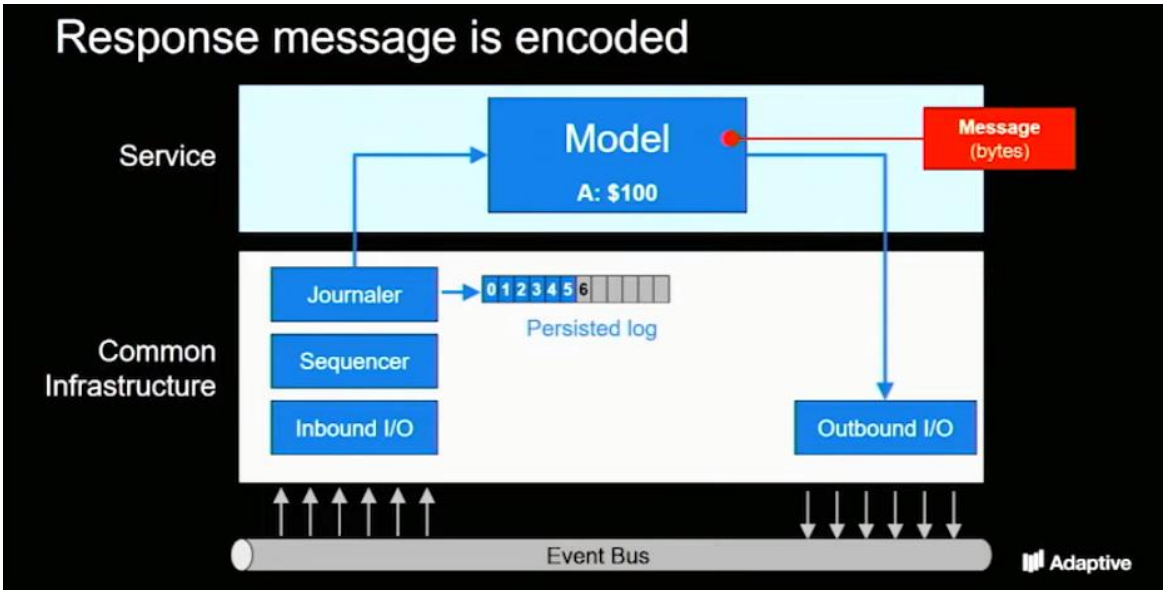

Message is written to disk
Service
Model
State & behavior
Message
(bytes)
Journaler
0 1 2 3 4 5 6
Persisted log
Common
Infrastructure
Sequencer
Inbound I/O
Outbound I/O
Event Bus
Adaptive

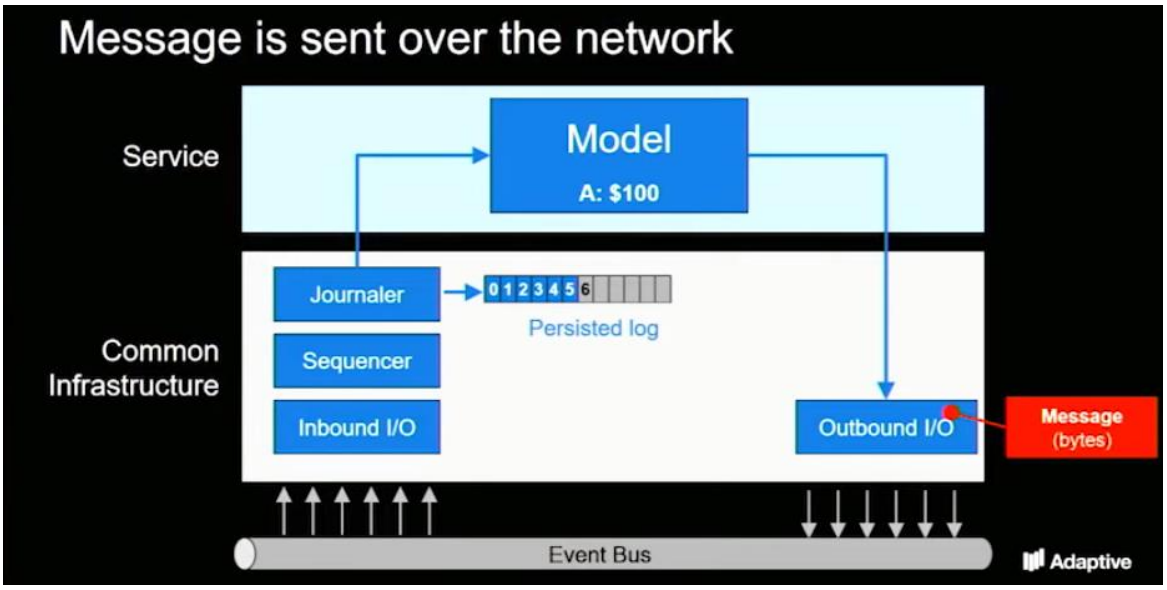

Message passed to the model
Service
Model
State & behavior
Message
(bytes)
Journaler
0 1 2 3 4 5 6
Persisted log
Common
Infrastructure
Sequencer
Inbound I/O
Outbound I/O
Event Bus
Adaptive

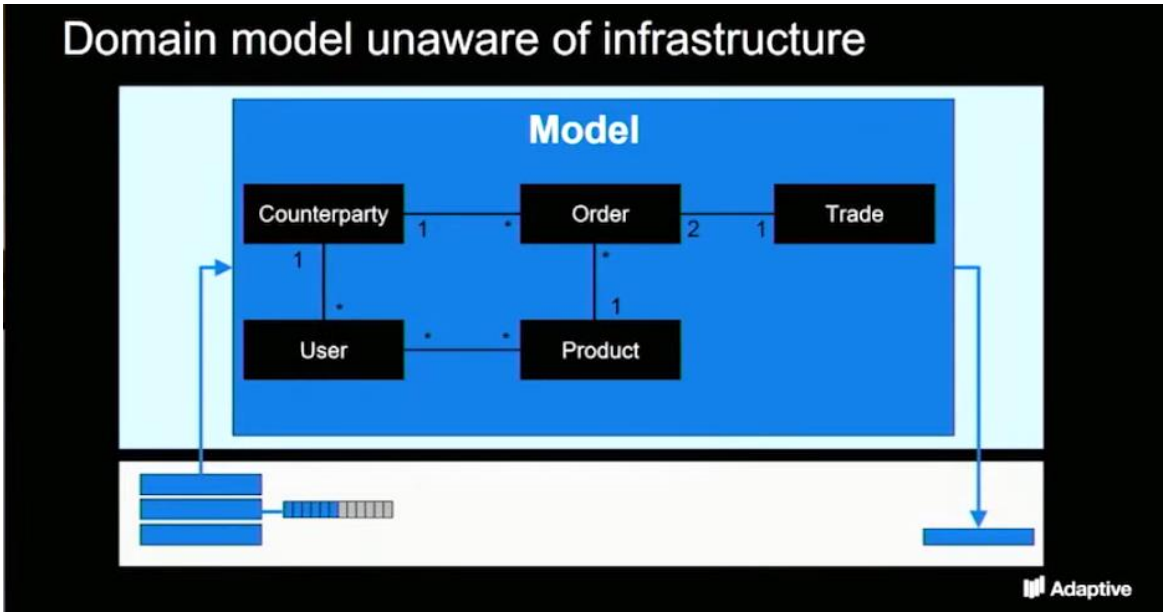

Message is decoded
DepositCommand
Amount: $100
Account: A
Model
State & behavior
Journaler
0 1 2 3 4 5 6
Persisted log
Common
Infrastructure
Sequencer
Inbound I/O
Outbound I/O
Event Bus
Adaptive

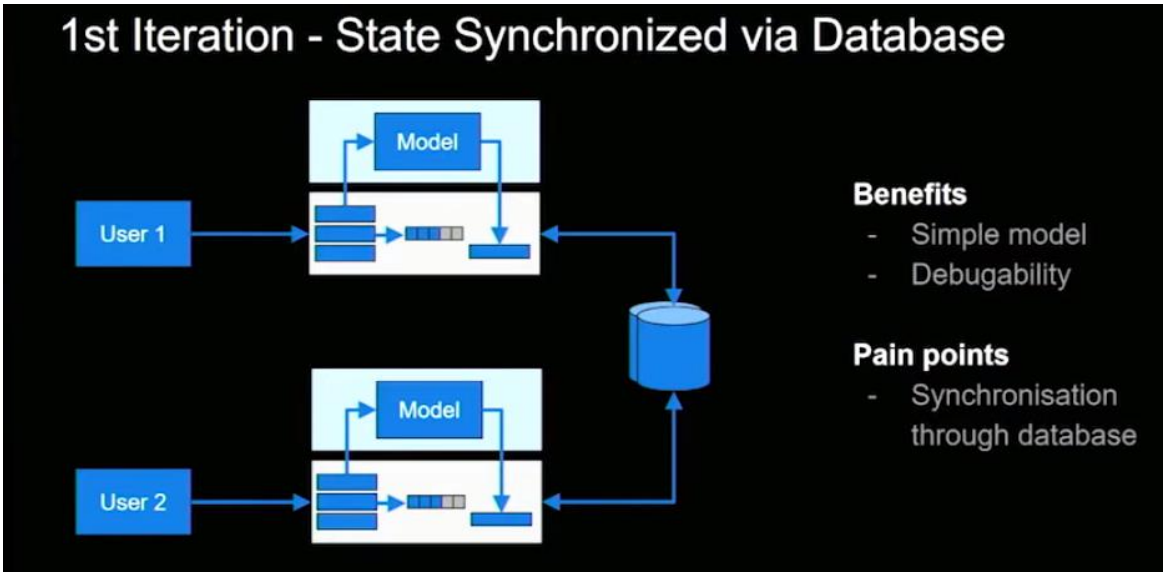

Command is applied to the model
DepositCommand
Amount: $100
Account: A
Model
A: $100
Journaler
0 1 2 3 4 5 6
Persisted log
Sequencer
Inbound I/O
Outbound I/O
Common Infrastructure
Event Bus
Adaptive


Response message is created
Service
Model
A: $100
Deposit Ack
Journaler
0 1 2 3 4 5 6
Persisted log
Sequencer
Inbound I/O
Outbound I/O
Common Infrastructure
Event Bus
Adaptive


Response message is encoded
Service
Model
A: $100
Message
(bytes)
Journaler
0 1 2 3 4 5 6
Persisted log
Sequencer
Inbound I/O
Outbound I/O
Common Infrastructure
Event Bus
Adaptive

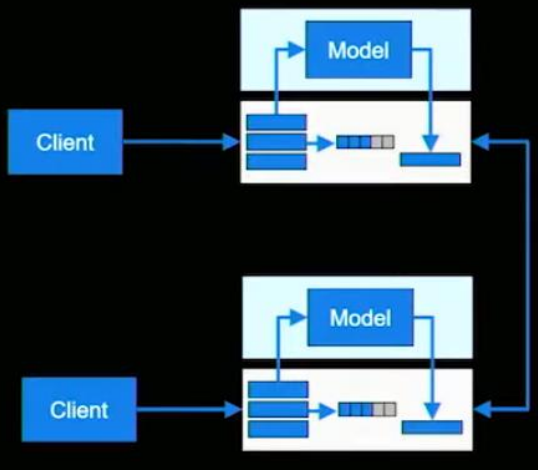

Message is sent over the network
Service
Model
A: $100
Journaler
0 1 2 3 4 5 6
Persisted log
Common Infrastructure
Sequencer
Inbound I/O
Outbound I/O
Message (bytes)
Event Bus
Adaptive


Domain model unaware of infrastructure
Model
Counterparty
Order
Trade
User
Product
1
*
2
Adaptive


1st Iteration - State Synchronized via Database
Model
User 1
Model
User 2
Benefits
- Simple model
- Debugability
Pain points
- Synchronisation through database

# Fault Tolerance & State Replication

## 2nd Iteration - Clustering



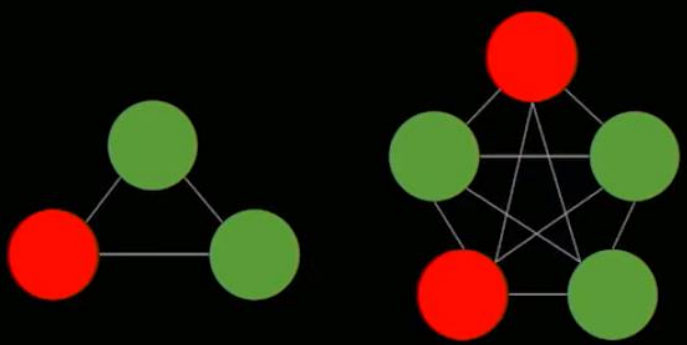

## Fault tolerant data replication

Difficult problem:

- The log must stay **consistent** between servers
- Even if a **server fails** or the **network fails**

## Distributed consensus problem

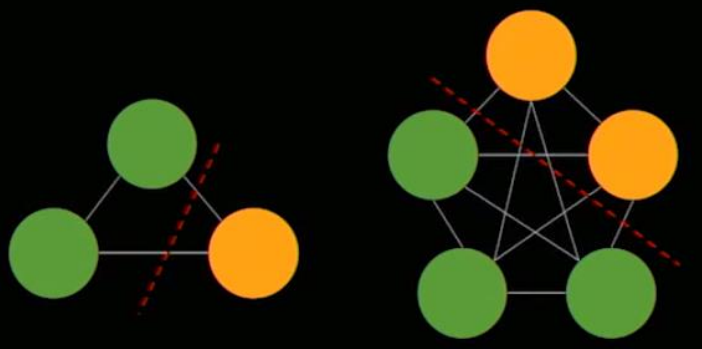
## RAFT



Tolerates fault of **1 node**

Tolerates fault of **2 nodes**

## RAFT - Network failure



**1 node** partitioned from the other nodes

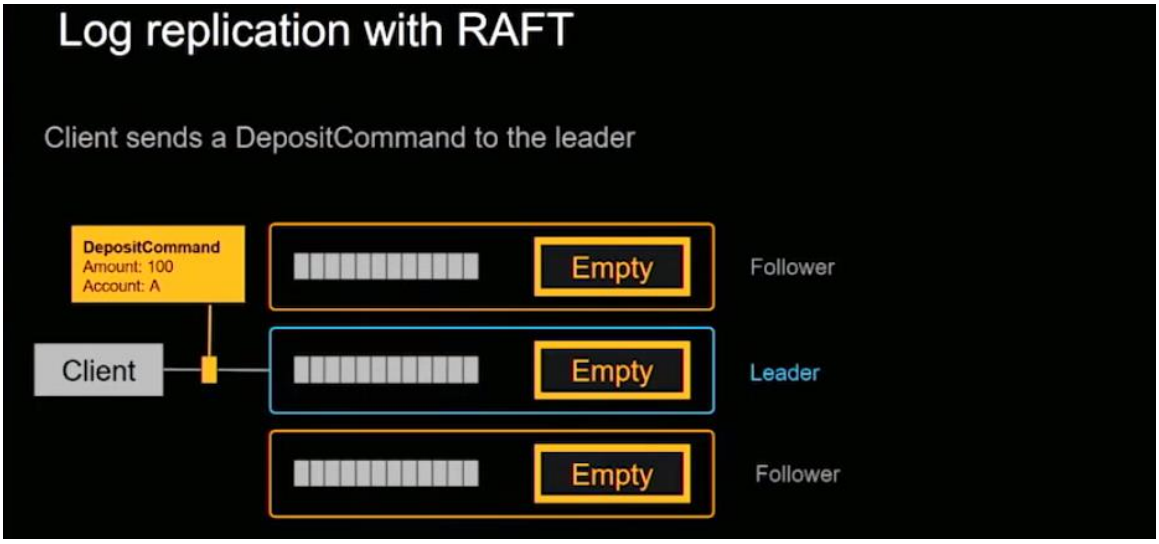
**2 nodes** partitioned from the other nodes

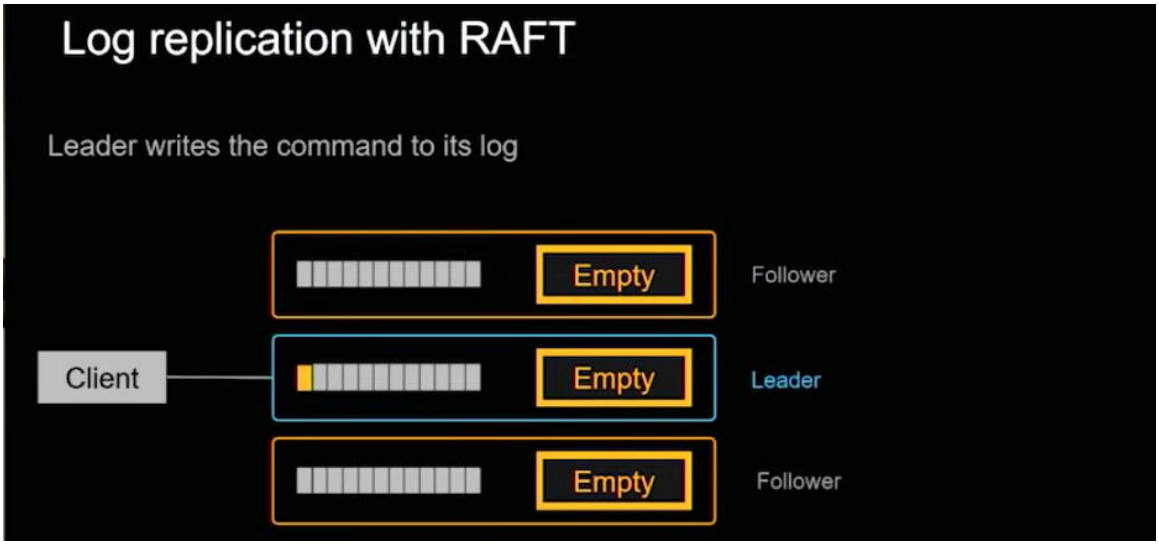
## Log replication with RAFT

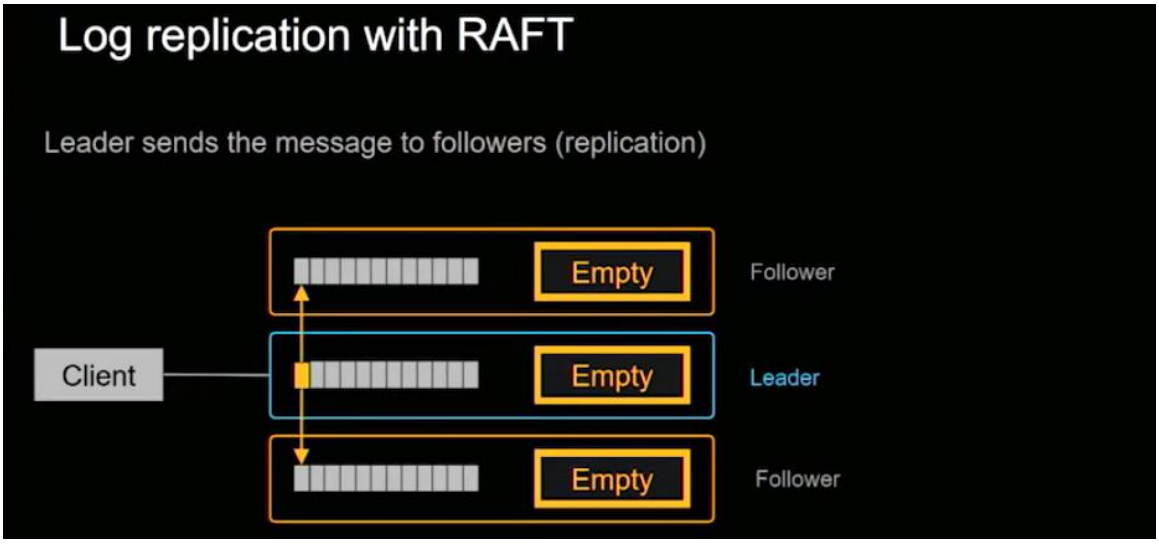
Leader election


## Log replication with RAFT

Clients connect to the leader


Log replication with RAFT
Client sends a DepositCommand to the leader
DepositCommand
Amount: 100
Account: A
Client
Empty
Empty
Empty
Follower
Leader
Follower


Log replication with RAFT
Leader writes the command to its log
Client
Empty
Empty
Empty
Follower
Leader
Follower


Log replication with RAFT
Leader sends the message to followers (replication)
Client
Empty
Empty
Empty
Follower
Leader
Follower

## Log replication with RAFT

Followers store the message in their log and acknowledge back to the leader


## Log replication with RAFT

Leader marks the message as committed, process the message and notifies followers to commit the message


## Log replication with RAFT

Followers commit the message in their log and process it

## Log replication with RAFT

Leader sends a deposit acknowledgement to the client


## Log replication with RAFT

What if the leader dies?


## Log replication with RAFT

Followers hold an election, one of them becomes leader

## Log replication with RAFT

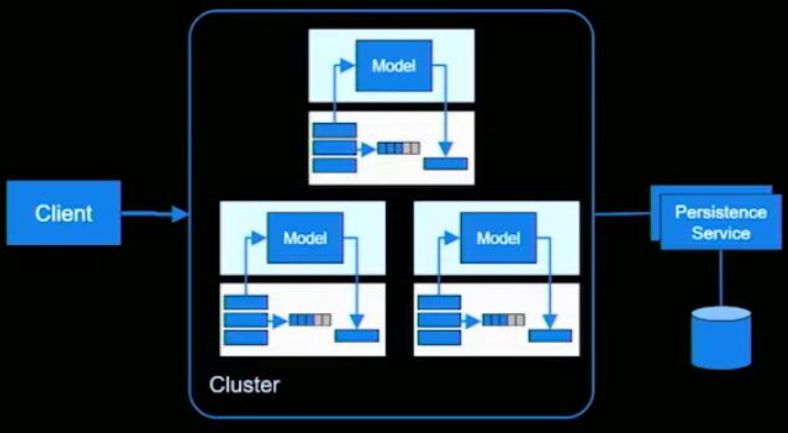
Client detects disconnection from leader and connects to new leader


## Log replication with RAFT

Client is connected to another instance of the service, in the same state

⇒ System is tolerant to server and network failures


## 2nd Iteration - Clustering




**Benefits**

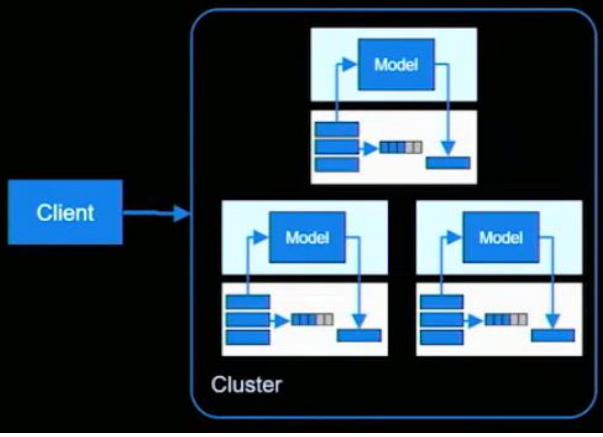
- Fault tolerant
- Consistent model & it doesn't change

**Pain points**

- Database as golden source

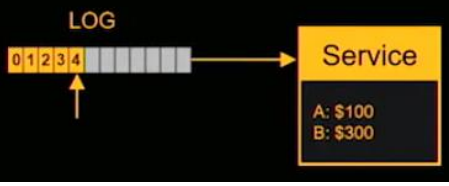
# Durability

## 3rd Iteration - Durable Services




## System restart

Given the current state...

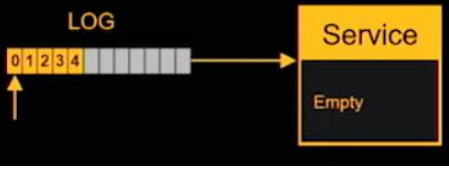



## System restart

Stop the service


## System restart

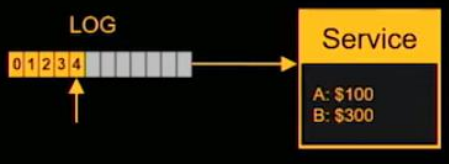
Service restarts, reads from beginning of the log



## System restart

Replay all commands up to latest

Services is guaranteed to get back in the same state due to **deterministic** logic




## Managing the log

**Problem**: the size of the **log** affects **recovery time**

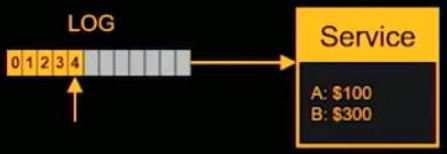
Not a problem if

- Log grows slowly
- Your model has short lifetime

Otherwise consider **snapshotting** or other forms of **compaction**

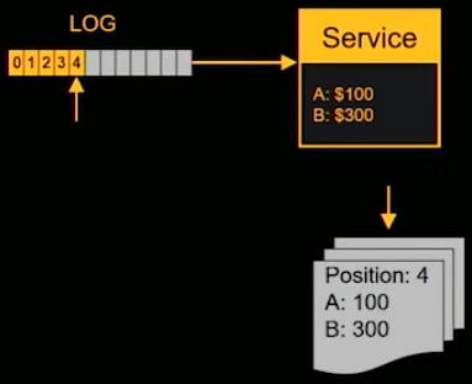
## Snapshotting

Considering the system in this state...




## Snapshotting

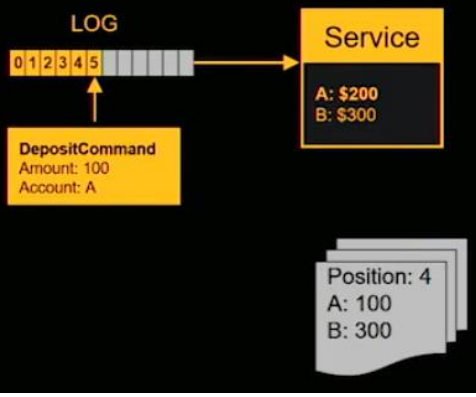
Service writes current state to disk with position in the log




We take the model in memory and serialize it to save it

## Snapshotting

Message 5 gets written to the log and processed

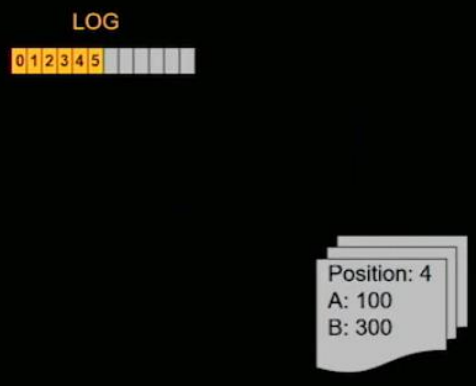



## Snapshotting

Stop the service and restart

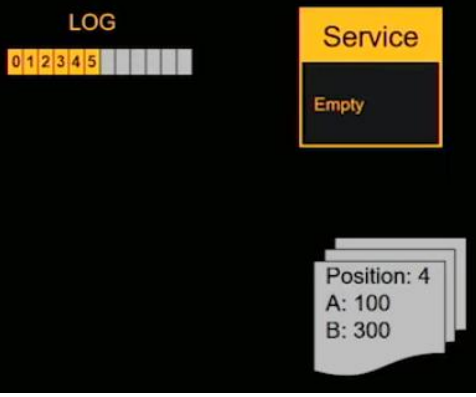
We still have

- The **log**
- The **snapshot**




## Snapshotting

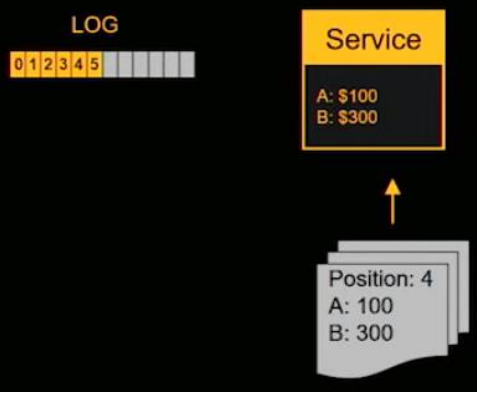
Service starts, initially with an **empty** model



# Snapshotting

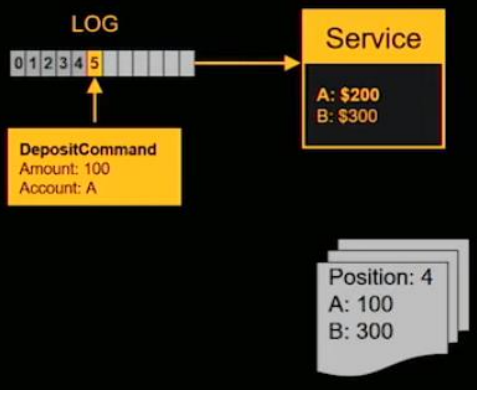
Service loads most recent snapshot

- Re-hydrate model




# Snapshotting

Replay all messages after current position

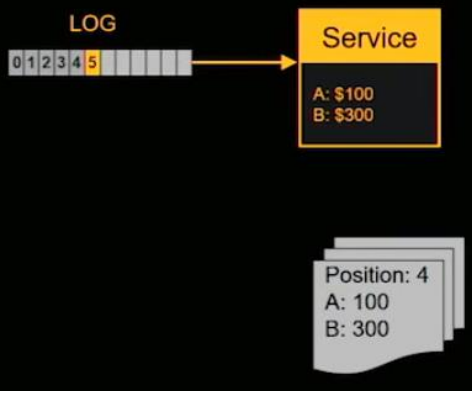



# Snapshotting

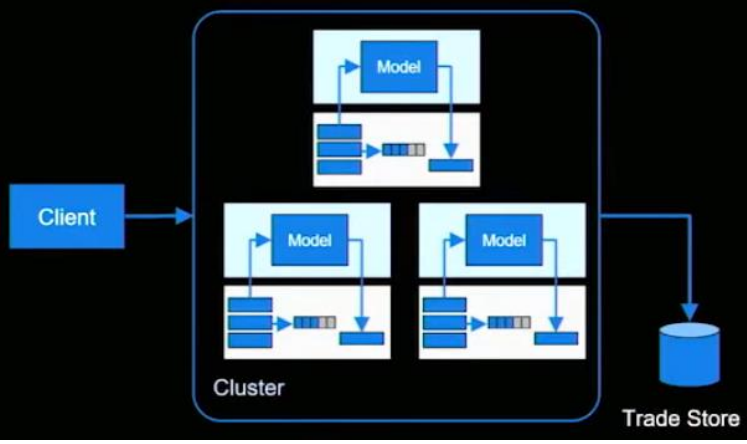
System is back in the same state

Snapshotting makes it possible to **trim** the log

With a snapshot at position N, we can **archive** the log up to the same position



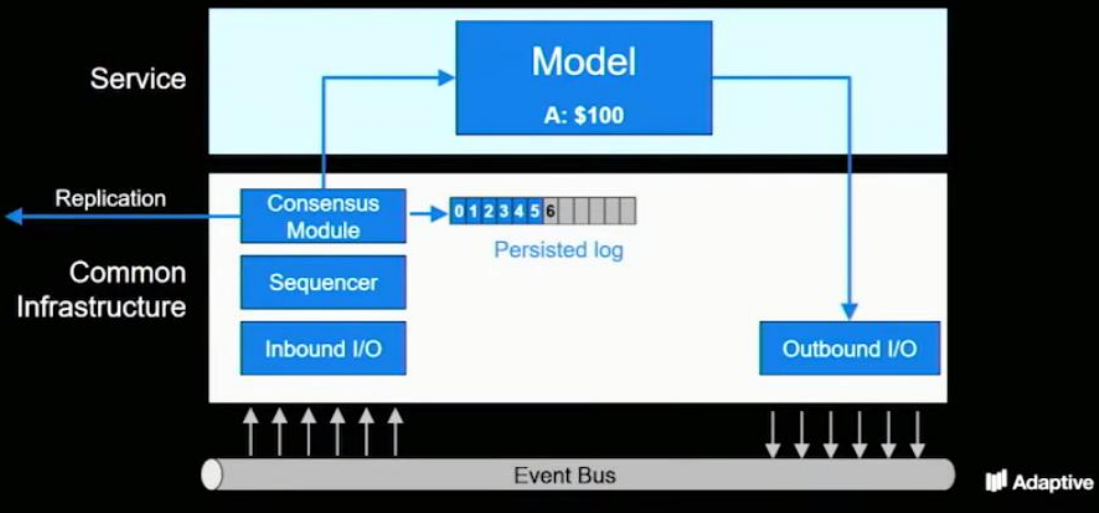
## Iteration 3 - Durable Services




**Benefits**

- Model unaffected
- Fault tolerant
- Consistent model
- Durable without database

## Reactive Application Server




## Don't build the infrastructure yourself!

Open source

- Copycat - http://atomix.io/
- Aeron - https://github.com/real-logic/aeron

Todd Montgomery

Martin Thompson

## Other use-cases

- Online games
  - MMO
  - Gambling
- Ticketing system
- Consistent Databases / Caches
- Many more..

## Wrapping up

- Demonstrated the simplicity of this approach
- Widely applicable
- Deserves more attention!
- Learn more
  - LMAX Architecture https://goo.gl/q1iSCB
  - Raft paper and website - https://raft.github.io/
  - The Log - https://goo.gl/m4iWqn
  - White paper - http://weareadaptive.com/blog