HACEP: Highly Available and Horizontally Scalable Complex Event Processing

HANDS on LAB

Ugo Landini, Sanne Grinovero, Andrea Tarocchi, Andrea Leoncini

04/05/2017

Agenda

- Rule Engine 101
- The case for HACEP
- Patterns at work
- High Level Architecture
- Deep Dive
- **LAB 1: INSTALLATION**
- **LAB 2: MODIFY THE RULES**
- **LAB 3: UPDATE THE RULES, THE RIGHT WAY**
- HACEP Internals
- HACEP Roadmap
- Conclusions

RULE ENGINE 101

What is a rule engine?

A business **rules engine** is a software system that executes one or more business rules in a runtime **production environment**. The rules might come from legal regulation ("*An employee can be fired for any reason or no reason but not for an illegal reason*"), company policy ("*All customers that spend more than $100 at one time will receive a 10% discount*"), or other sources.

A business rule system enables these company policies and other operational decisions to be defined, tested, executed and maintained **separately** from application code.

Source: wikipedia

DROOLS/Red Hat JBoss BRMS

Advantages of a Rule Engine

- Rule engines allow you to say "**What** to do", not "**How** to do it".
- Logic and Data **Separation**
- **Centralization** of Knowledge
- Tool Integration
- **Understandable** Rules
- **Speed** and Scalability

When you **should** use Red Hat JBoss BRMS

- Very complex scenarios that are difficult to fully define even for **business experts**
- don't have a known or **well-defined algorithmic** solution
- volatile **requirements** that need to be updated **very often**
- need to make **decisions** fast, usually based on partial amounts of data

A simple rule

This is a plain Drools rule, with **no concept of time**

```
rule "Sound the alarm"
when
    $f : FireDetected( )
    not( SprinklerActivated() )
then
    // sound the alarm
end
```

A simple rule

This is a plain Drools rule, with **no concept of time**

```
rule "Sound the alarm"
when
    $f : FireDetected( )
    not( SprinklerActivated() )
then
    // sound the alarm
end
```

Rule name

A simple rule

This is a plain Drools rule, with **no concept of time**

```
rule "Sound the alarm"
when
    $f : FireDetected( )
    not( SprinklerActivated() )
then
    // sound the alarm
end
```

LHS

A simple rule

This is a plain Drools rule, with **no concept of time**

```
rule "Sound the alarm"
when
    $f : FireDetected( )
    not( SprinklerActivated() )
then
    // sound the alarm
end
```

RHS

CEP with DROOLS

Complex **E**vent **P**rocessing: detect events of significance to a business by recognizing **time-based patterns** in one or more real-time data feeds...

- Rules + Time: adding the concept of **time** to basic rules
  - Sliding windows
  - Entry points
  - Time operations

A simple CEP rule

This is a Drools CEP rule, with **time**

```
rule "Sound the alarm"
when
    $f : FireDetected( )
    not( SprinklerActivated( this  after[0s,10s] $f ) )
then
    // sound the alarm
end
```

THE CASE FOR HACEP

HA use cases

Out of the box, Drools/BRMS **doesn't provide** neither an HA architecture nor an horizontal scalability solution

- Everything must be in a **single** jvm
  - You have to provide your **own** HA/Scalability solution
- Many use cases needs a solution for that:
  - CEP with **big sliding windows** usually consumes a lot of RAM
  - whenever a single session is **not enough**

Sample use case

Functional requisites

- User does something **T** times
- User does something **T** times for **D** consecutive days
- User places **X** actions in **D** days
- User wins/loses more than **X**
- User wins/loses a cumulative **X** amount

User gets “something” back, in near real time

(levels/ranking assignment, rewards, achievements, fraud detection, etc.)

Sample use case

Functional requisites

- User does something **T** times
- User does something **T** times for **D** consecutive days
- User places **X** actions in **D** days
- User wins/loses more than **X**
- User wins/loses a cumulative **X** amount

LHS

User gets "something" back, in near real time

(levels/ranking assignment, rewards, achievements, fraud detection, etc.)

Sample use case

Functional requisites

- User does something **T** times
- User does something **T** times for **D** consecutive days
- User places **X** actions in **D** days
- User wins/loses more than **X**
- User wins/loses a cumulative **X** amount

User gets "something" back, in near real time

(levels/ranking assignment, rewards, achievements, fraud detection, etc.)

RHS

A simple Player reward rule

```
rule "User gains a point for each gameplay and every 10 points increases player level"
when
    $gamePlay : Gameplay($playerId : playerId) over window:length(1)
    $numberOfTimes : Number()
                    from accumulate ($gamePlayCount :
Gameplay($gamePlay.playerId == playerId) over window:time(30d),
                    count($gamePlayCount))
then
    channels["playerPointsLevel"].send(new PlayerPointLevel(
        $playerId,
        $numberOfTimes.intValue() % 10,
        $numberOfTimes.intValue() / 10)
    );
end
```

LHS

RHS

Sample use case

Non functional requisites

- **10M** events per day
- **1M** registered users
- CEP sliding window in the **~60** days
- **8k** concurrent users per day
- **90k** unique users in 30 days
- **~200** bytes per event
- **1 second** available (end to end) to run all user rules and process rewards

Doing the Math

- **~200** bytes *** 10M** events *** 60** days is **~120GB** just for the "raw" facts
- BRMS Sessions contains **much more** than just the events
- **2** VMs of **128Gb** heaps each at a minimum would be needed to store everything
  - in 2 single sessions
    - **no** HA
    - **no** Scaling out

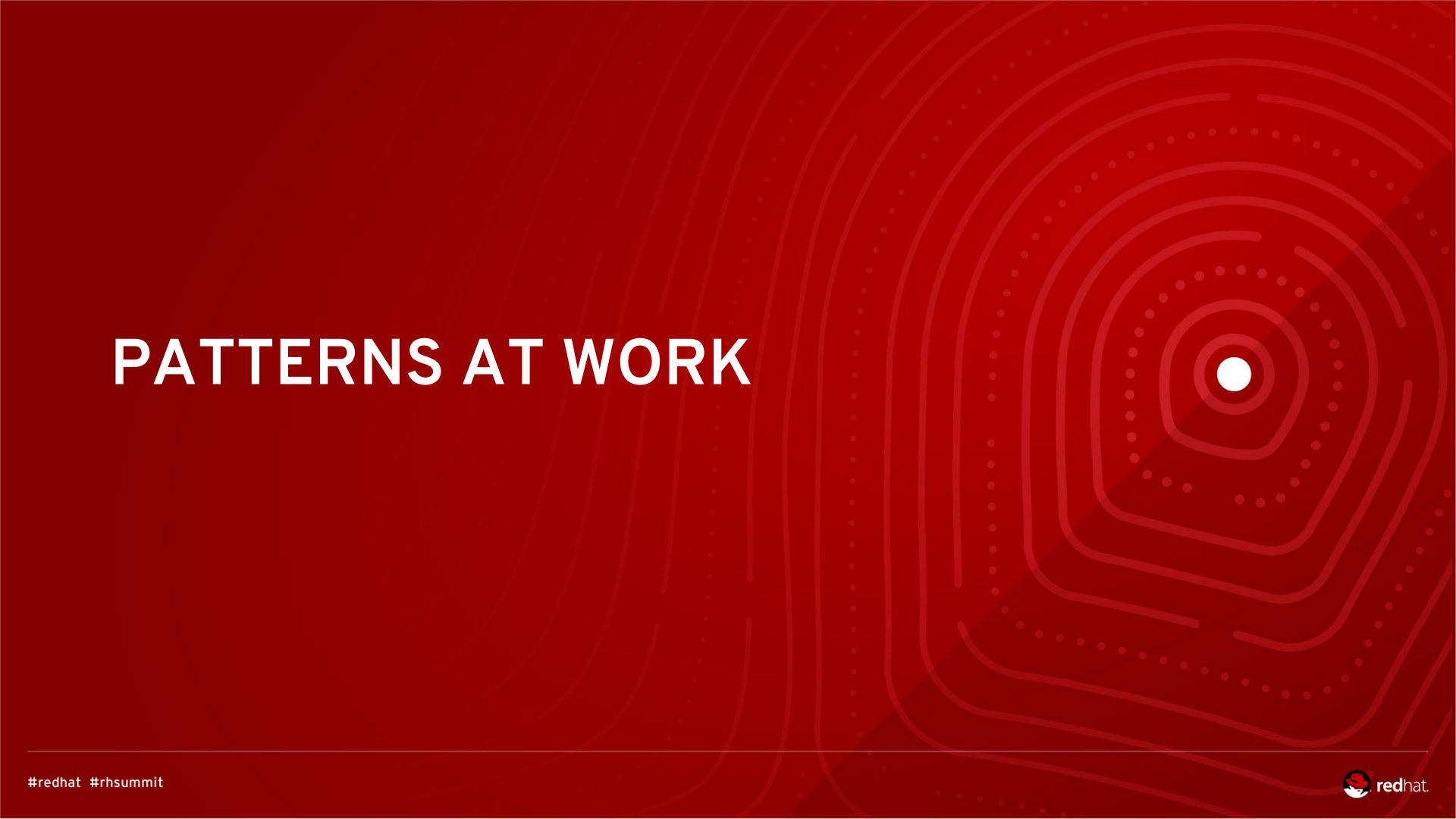
The case for HACEP

- **HACEP** uses **Infinispan**, **Camel** and **ActiveMq** to make Drools session scalable and highly available
- **HACEP** is a generic solution and impose just a partitioning criteria
  - no other constraint/limitations
- **HACEP** is useful in **any** CEP use case, in particular:
  - Financial
  - Gaming
  - IoT

HACEP 1.0 Features

- **Linearly** scalable from **2** to **100s** of nodes
- **Dynamically** scaling up and scaling down
- Survives to multiple node **failures**
- **In-memory** read/write performance for **extreme** throughput
- **Dynamic** CEP rules **update** on a live cluster
- Several **disk** storage options
- **Minimal** footprint
- **Rolling upgrades** support
- Plain **JVM** or **EAP** support

PATTERNS AT WORK



HACEP Patterns

- **HACEP** is **designed** on three **fundamental** patterns
  - **Sharding**/horizontal partitioning
  - Data **affinity**
  - Event **sourcing**

Sharding

- AKA **horizontal partitioning**
- **Splitting** data in separate nodes in order to improve **performance** and **scalability**

Data Affinity

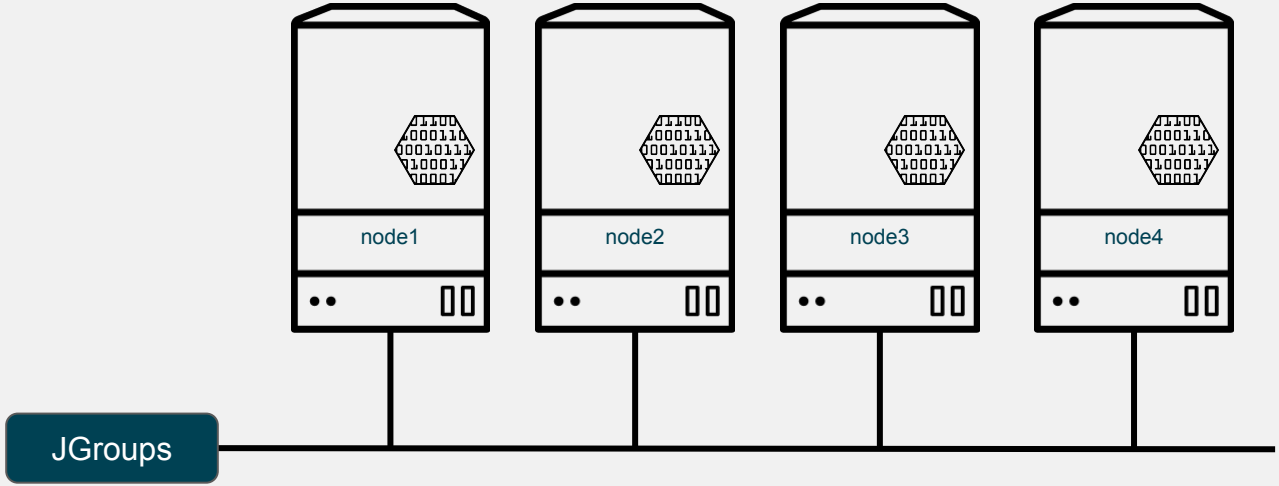
- Data affinity means **co-locating data** together to improve performance and scalability
- Data affinity means **co-locating computing code** with data **too**

Event Sourcing

- http://martinfowler.com/eaaDev/EventSourcing.html
- The fundamental idea of event sourcing is that of ensuring **every change to the state** of an application is captured in an **event object**, and that these event objects are themselves **stored in the sequence** they were applied for the same lifetime as the application state itself

JBoss Data Grid 101

JBoss Data Grid 101



JBoss Data Grid

- OSS: Apache License, active **upstream** community: *Infinispan*
- High Availability & Horizontal Scalability
- Transactions: **JTA** & full **XA** support
- **Distributed task** executions, **server side events** & beautiful **Stream API**
- **Queries** and **Vector Model** with Apache **Lucene**, **Analytics** with Apache **Spark**
- High performance custom Hot Rod clients, REST API
- Remote **X-Site** high availability
- Fully customizable network stack: get the best out of **any cloud** / metal platform
- **Offloading** to slower / larger / traditional **storage**: Cassandra, LevelDB, JDBC, itself...
- **Openshift** aware! And many more integrations...

HIGH LEVEL ARCHITECTURE

10 thousand foot

- Store everything in the **data grid** to overcome the single JVM limitation
- Treat Drools CEP as a **stateless** module
- Leverage **sharding** and **data affinity** to optimise network hops and make Drools scalable
- Basically, use Infinispan as Drools **distributed working memory**

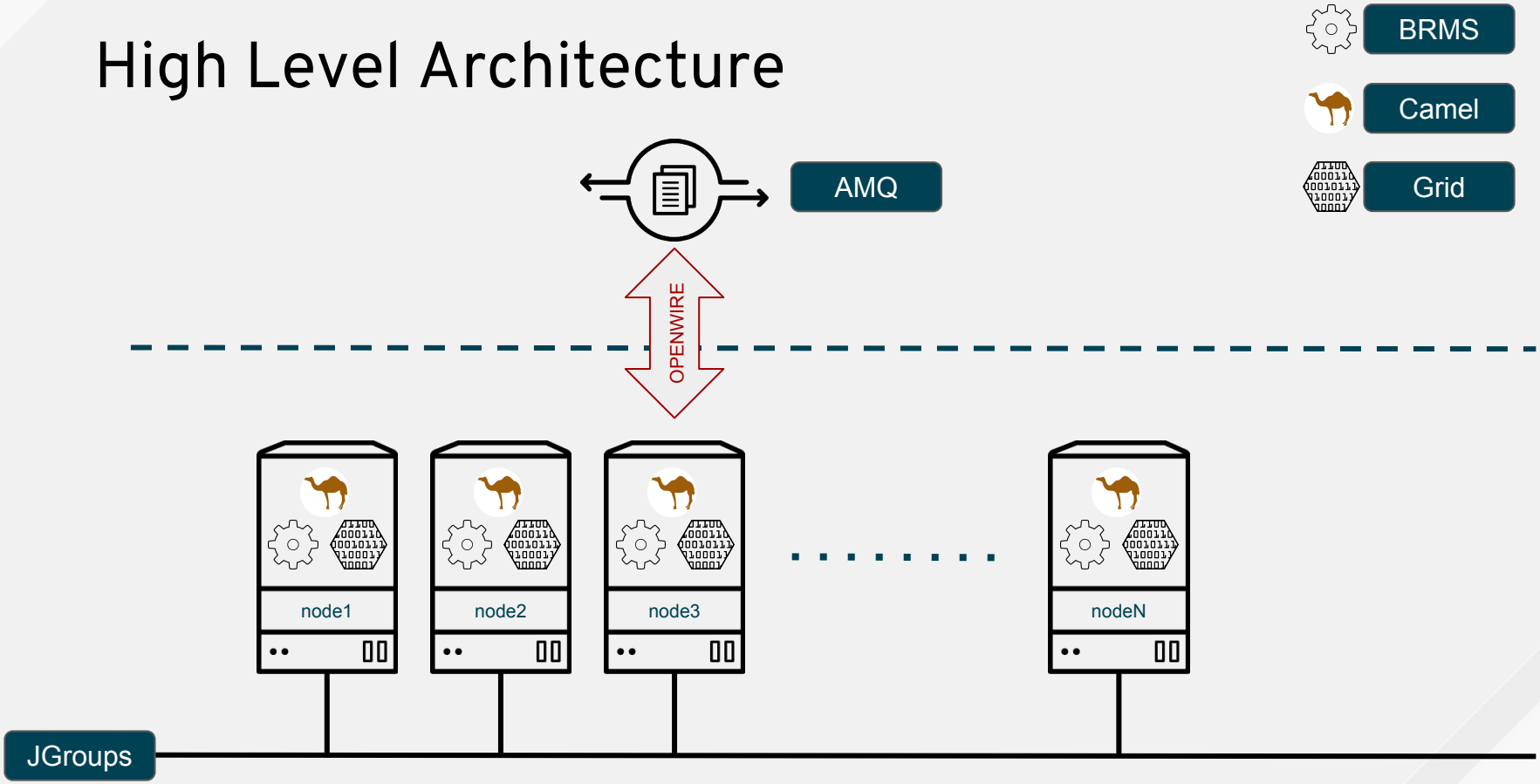
Sharding

- Drools sessions partitioned in different nodes on a **particular user defined criteria**
  - Gaming: sharding **per player**
  - FSI: sharding **per customer/cc**

Data Affinity

- Events and sessions both sharded on **same nodes**
  - i.e. using same **user defined partitioning criteria**
- Drools code **running on** the node containing needed data
- Events must be related to a **group**, so we can partition them and data affinity will be our friend
- a **group** is whatever business criteria we can use to partition (player, cc, location, etc.)
  - *you can't have generic cross-group rules!*

High Level Architecture



DEEP DIVE

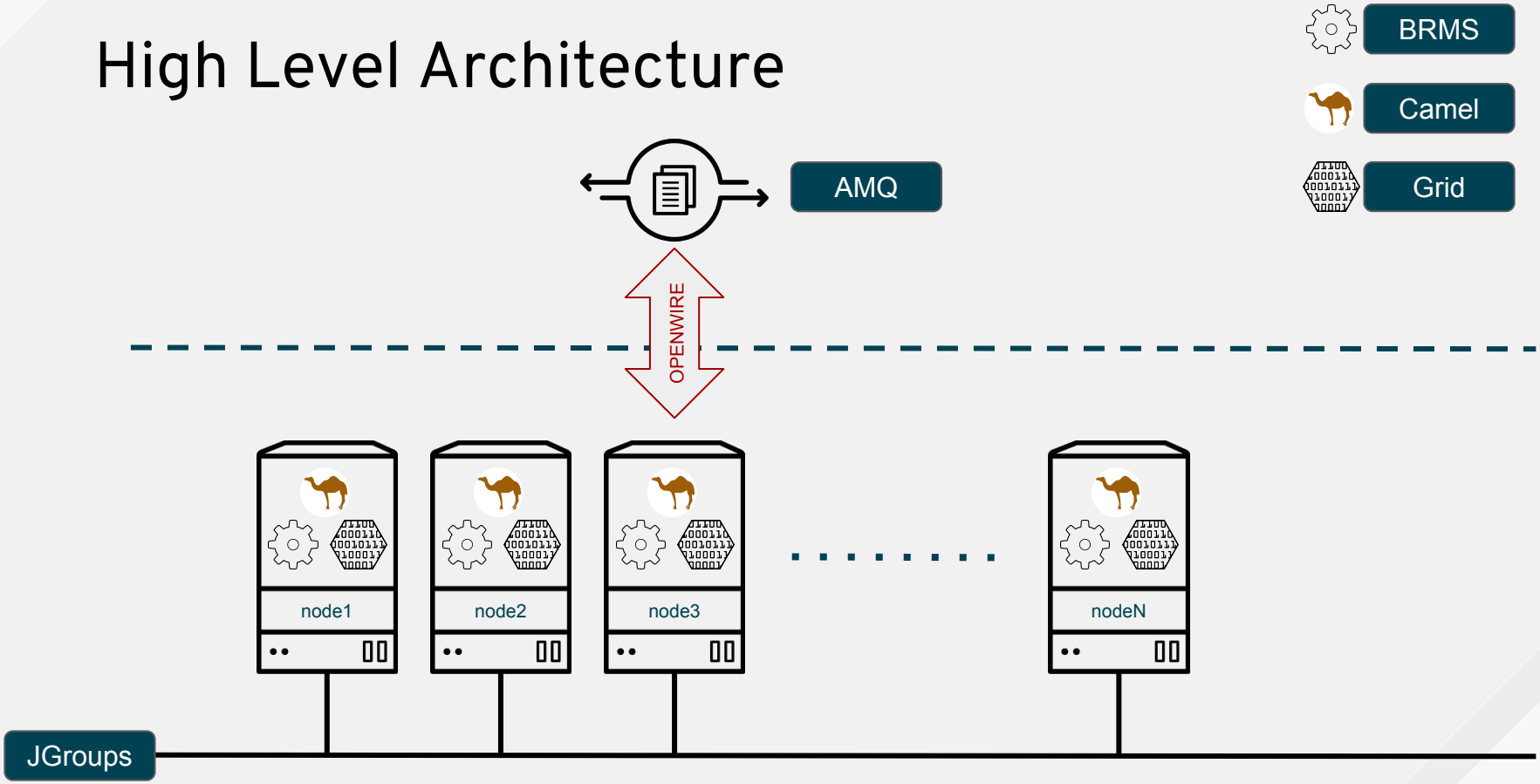
HACEP Nodes

- Each HACEP node is **identical**
- Each node contains:
  - a **Camel** route
  - a portion of the data, in 2 different **Infinispan** caches
  - **Drools** code

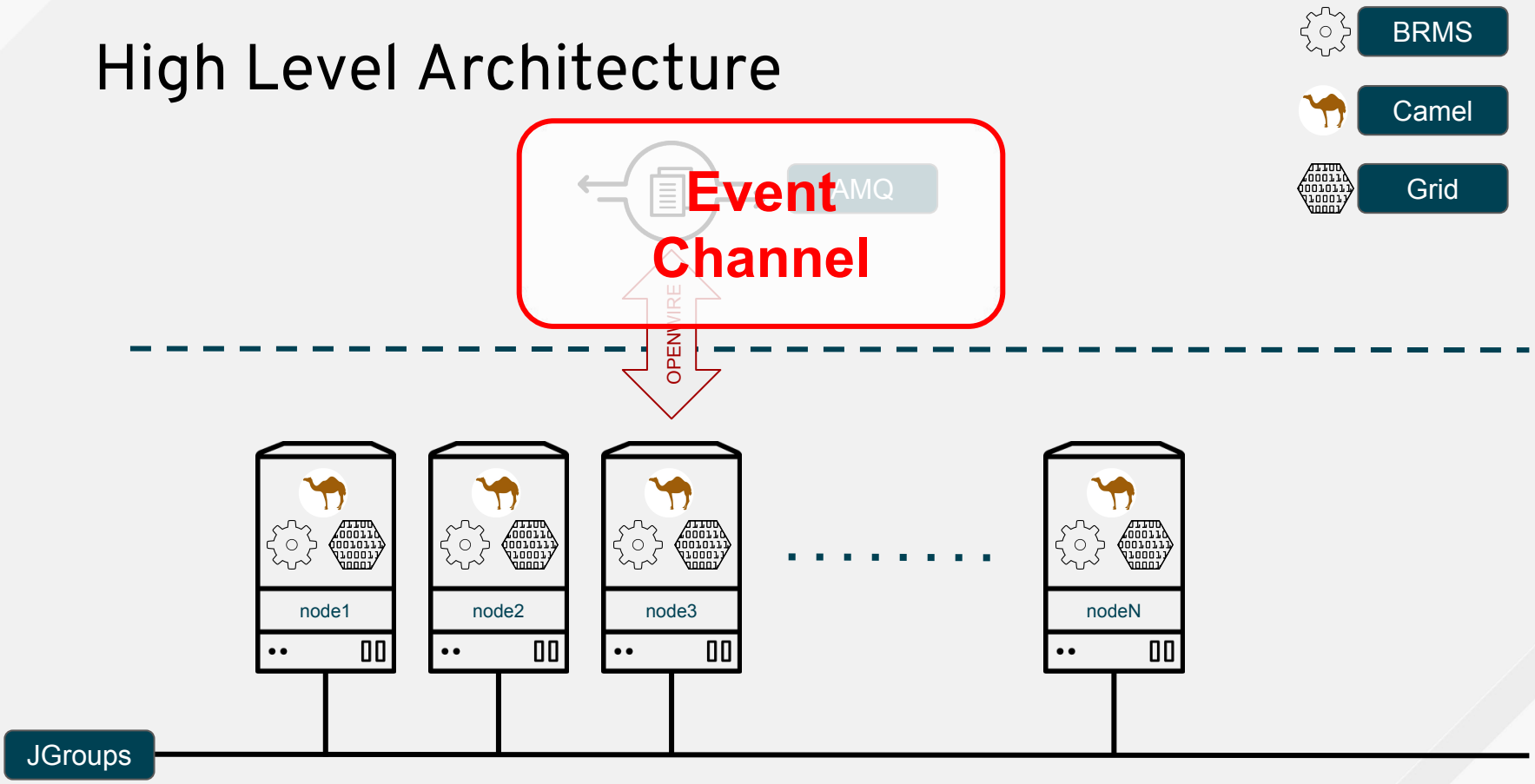
HACEP Nodes

- **Event channel** is external to HACEP nodes
- Could be anything:
  - Typically some kind of **MoM (Message Oriented Middleware)** software (AMQ, AMQP, generic JMS, Kafka…)

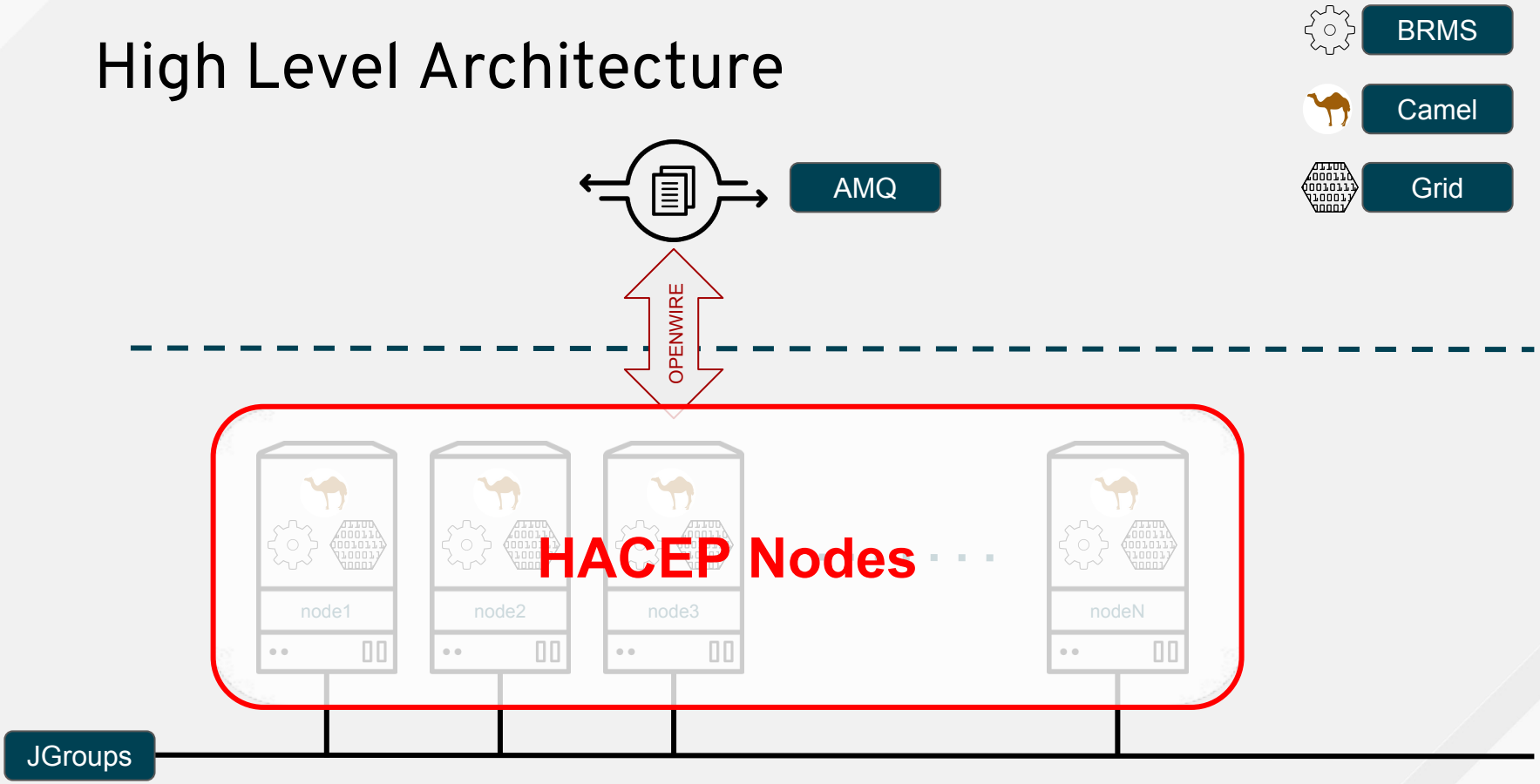
High Level Architecture



High Level Architecture



High Level Architecture



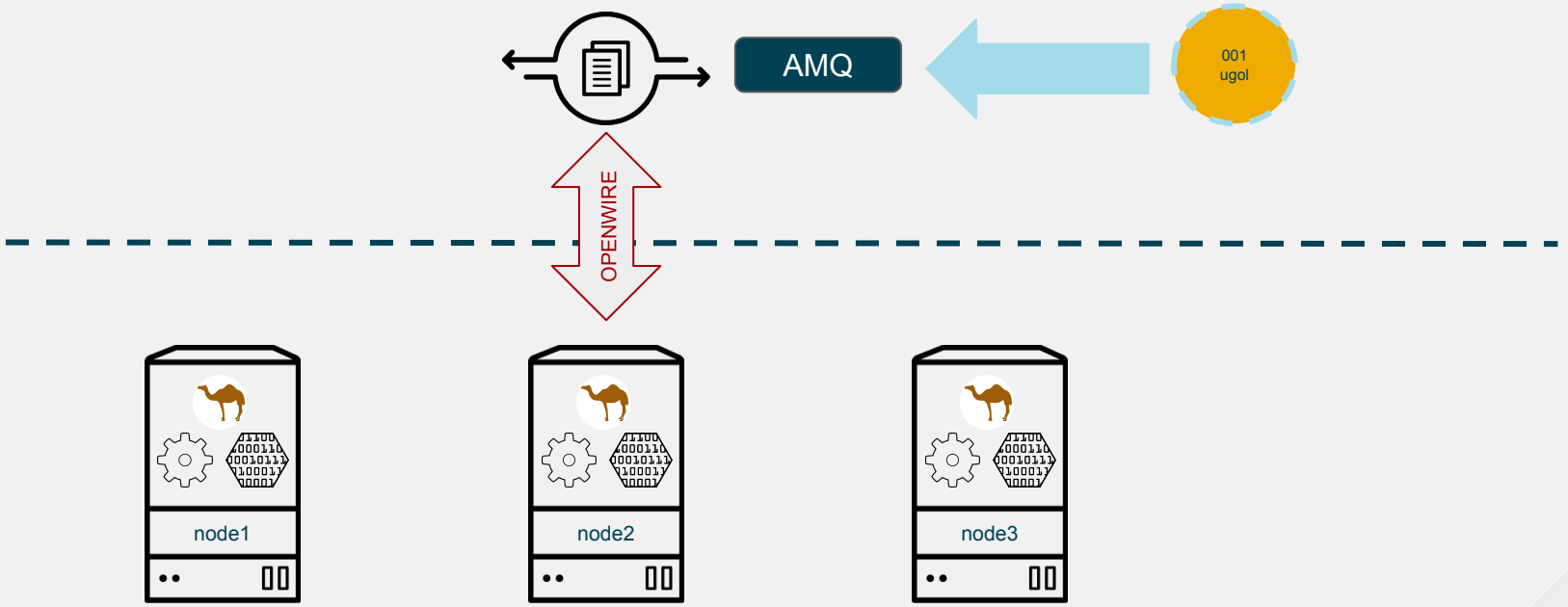
Camel Route

- The Camel route:
  - Gets events from an event channel
  - Puts the events in the **events** cache in Infinispan
- **Events** cache is configured with a **distributed** topology
  - **Grouping** is enabled
  - Events expires after a few milliseconds idle time

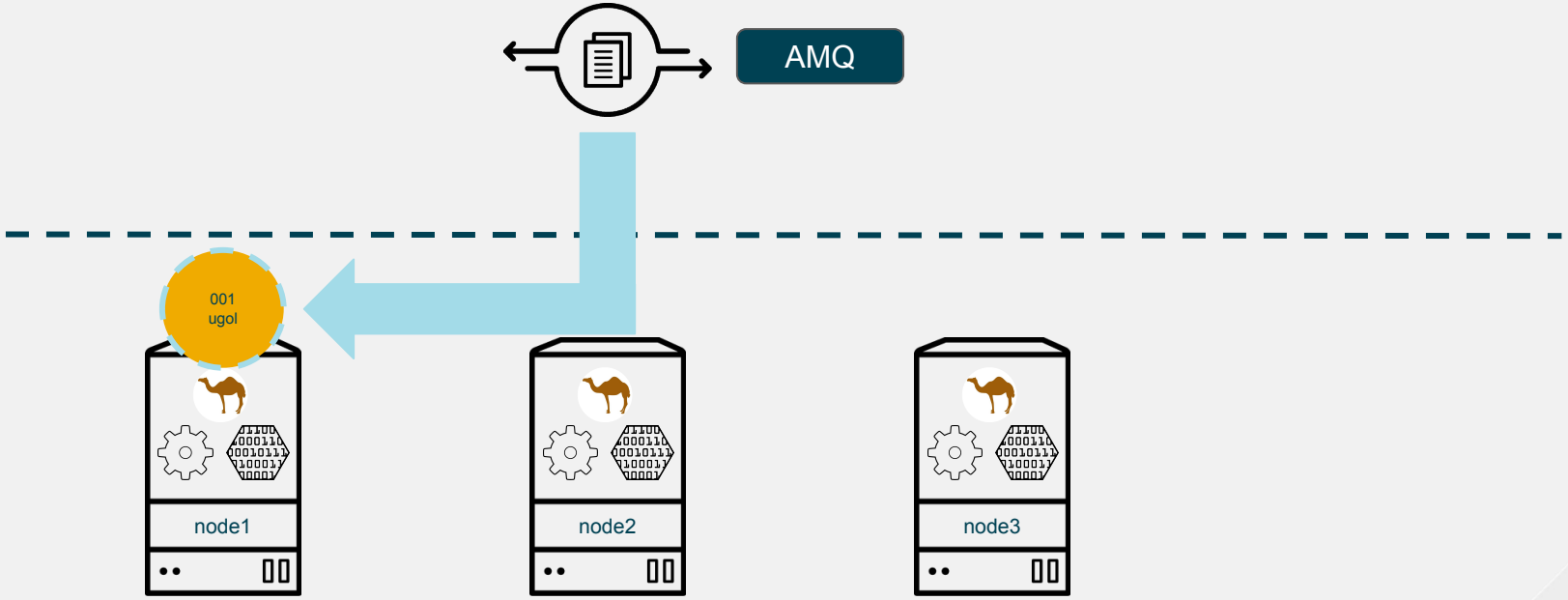
Why Expiration?

- We don't care about immediately **storing** the facts in the grid:
  - We put the facts in the Grid just to fire a notification
  - The notification is **synchronous** and happens only on the **primary** node
  - That means that the notification will be fired only on the node which will be (or already is) the primary node for that particular **group**
- So the first step is about **finding the right compute node**
  - **Grouping** guarantees that each key with same group will be **always hashed** on same nodes, regardless of topology changes

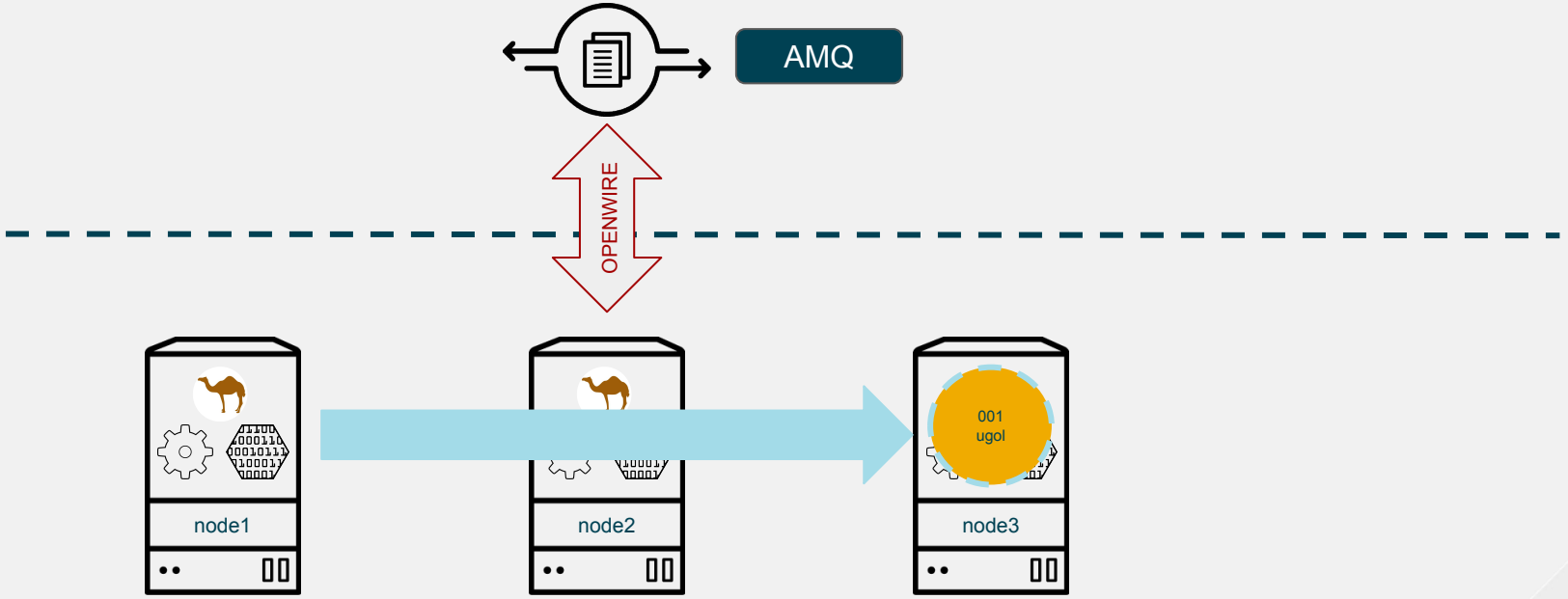
High Level Architecture



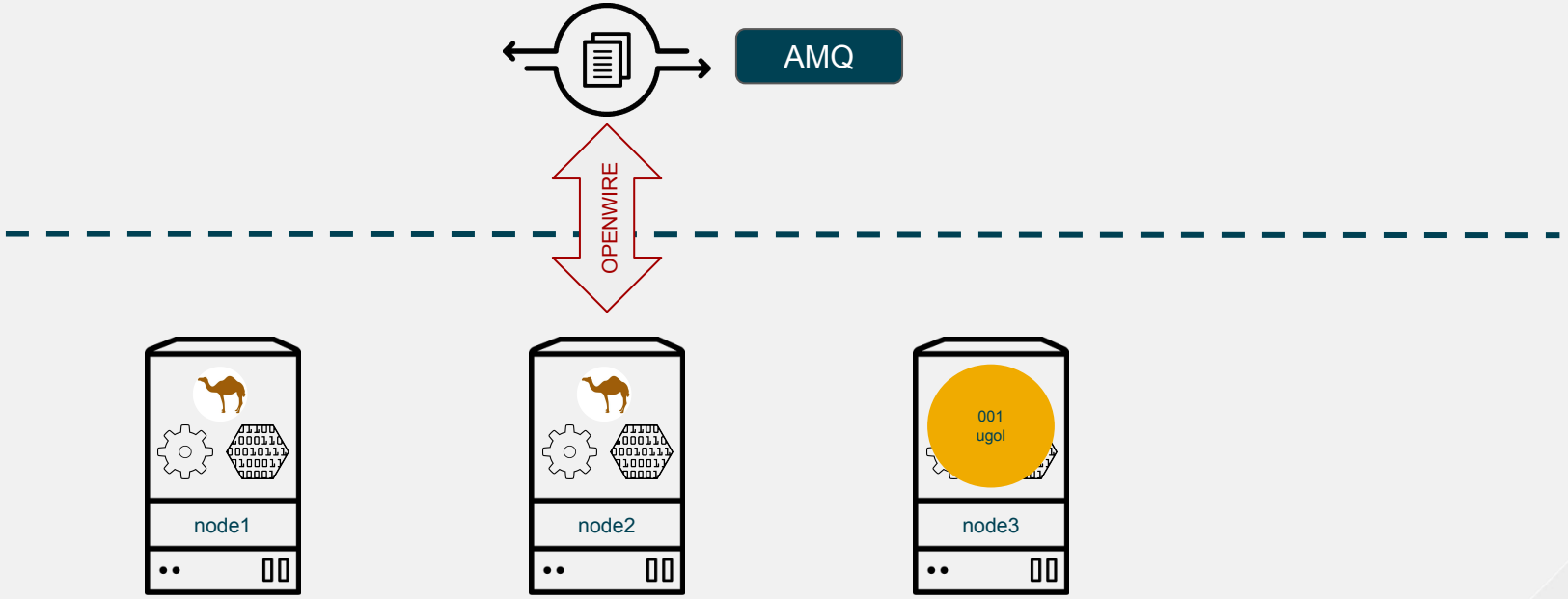
High Level Architecture



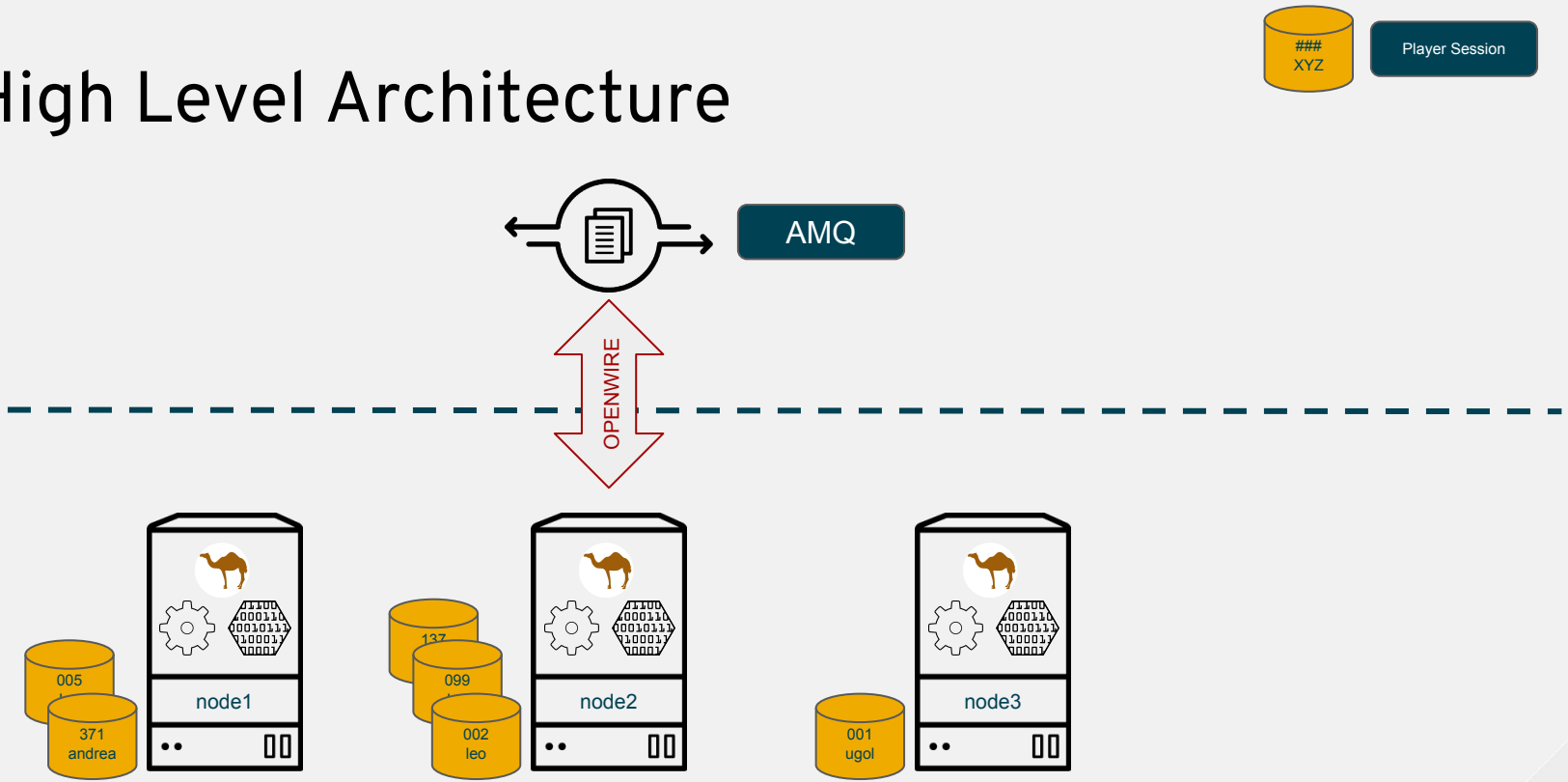
High Level Architecture



High Level Architecture



High Level Architecture



Camel Route

- So the **Camel route** just gets the event and puts it in a special **Red Hat JBoss Data Grid** cache
  - this cache **doesn't** store the events…
  - … but the notification system gives us the opportunity to find the **pertaining node** (remember data affinity?)

Finding the Node

- The **synchronous** and **primary** only listener will receive the event
  - Every event with the same **group** (player id, cc number, etc.) will be **always** notified on the **same node**
  - **Consistent hashing** and **grouping** guarantees that with a given topology, objects will **always** be hashed on the same nodes

Getting the Session

- The listener will **get** the specific session from the grid
  - Or will create an **empty** one if it's not already in the grid
- Sessions are **grouped** with same events criteria too, so everything is happening **locally**
  - Data affinity again!

… and finally, Drools!

- The **event** will be added to the session
- The **pseudoclock** will be advanced accordingly
- The rules will be fired
- The updated session will be **saved** again in the Infinispan **session** cache.
    - remember, everything is **local!**

“Stateless Drools”

- The **Drools state** lives only on JBoss Data Grid
- The Drools session of a single group (player, customer, ecc.) should never be more than **a few megabytes**

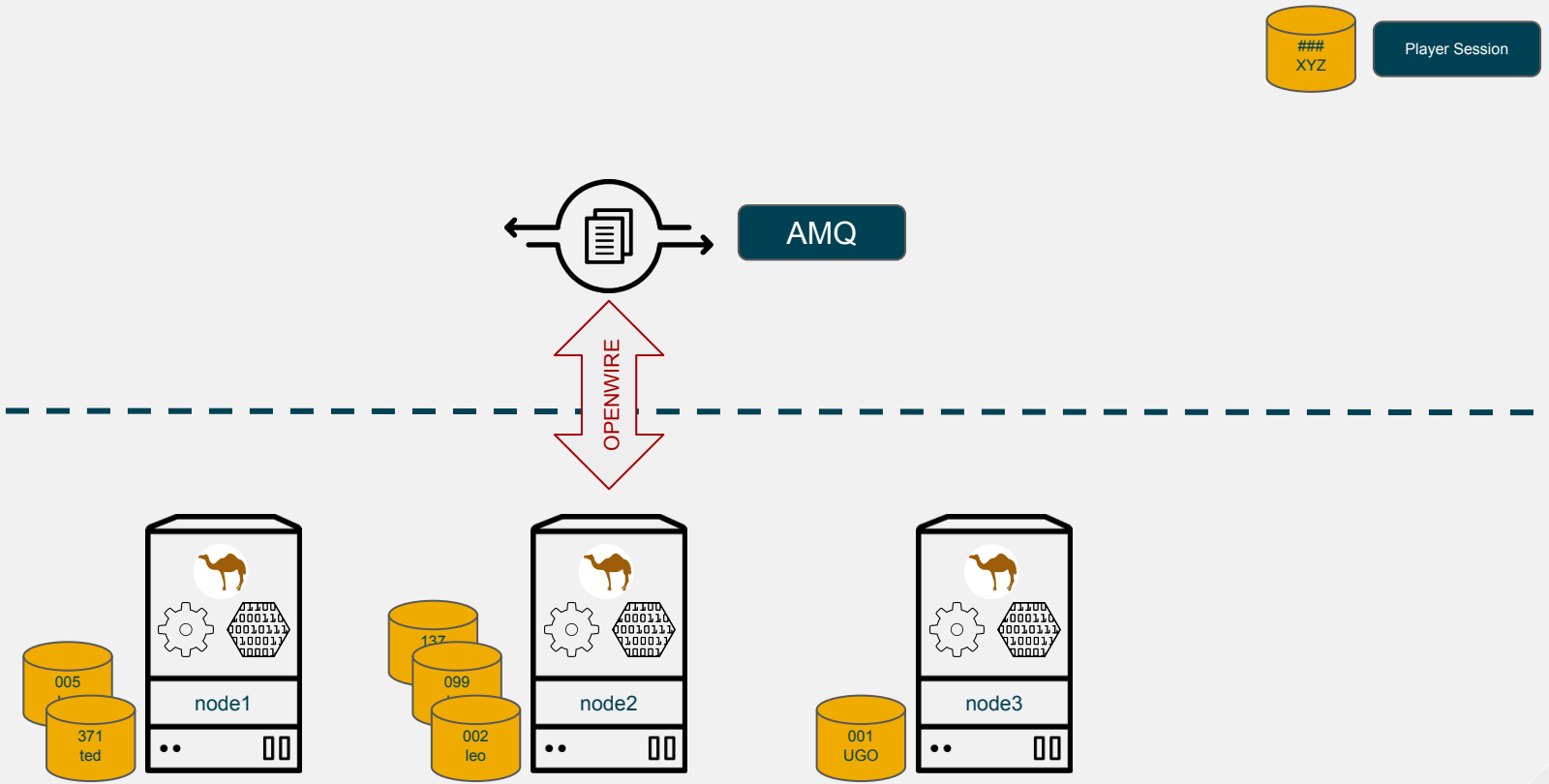
Note: Events are buffered with DeltaAware. No extra bit will move on the network if not really needed!

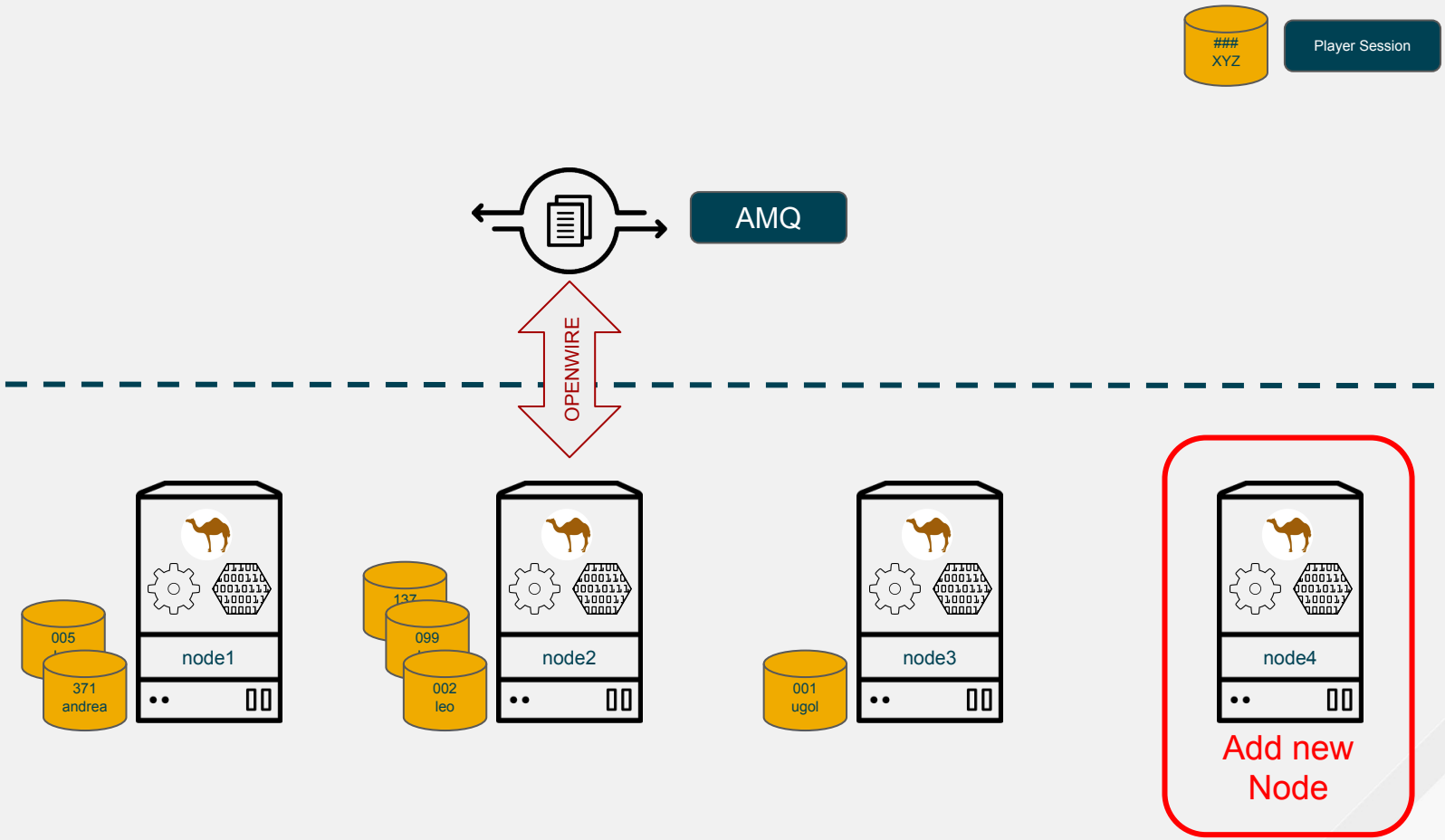
JBoss Data Grid Nirvana

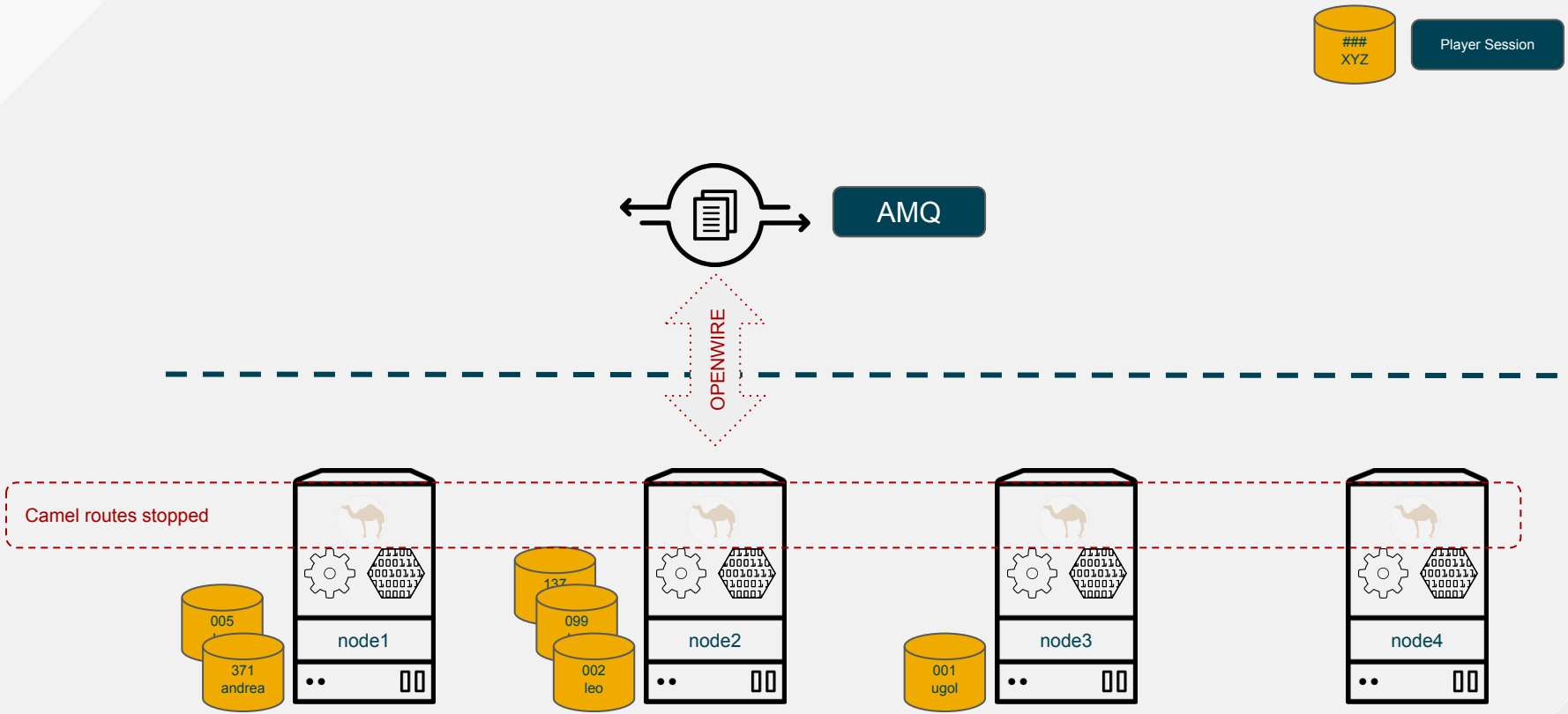
- *“All data are at the distance of a local Java call”*
- **HACEP** is **100% nirvana** compliant!

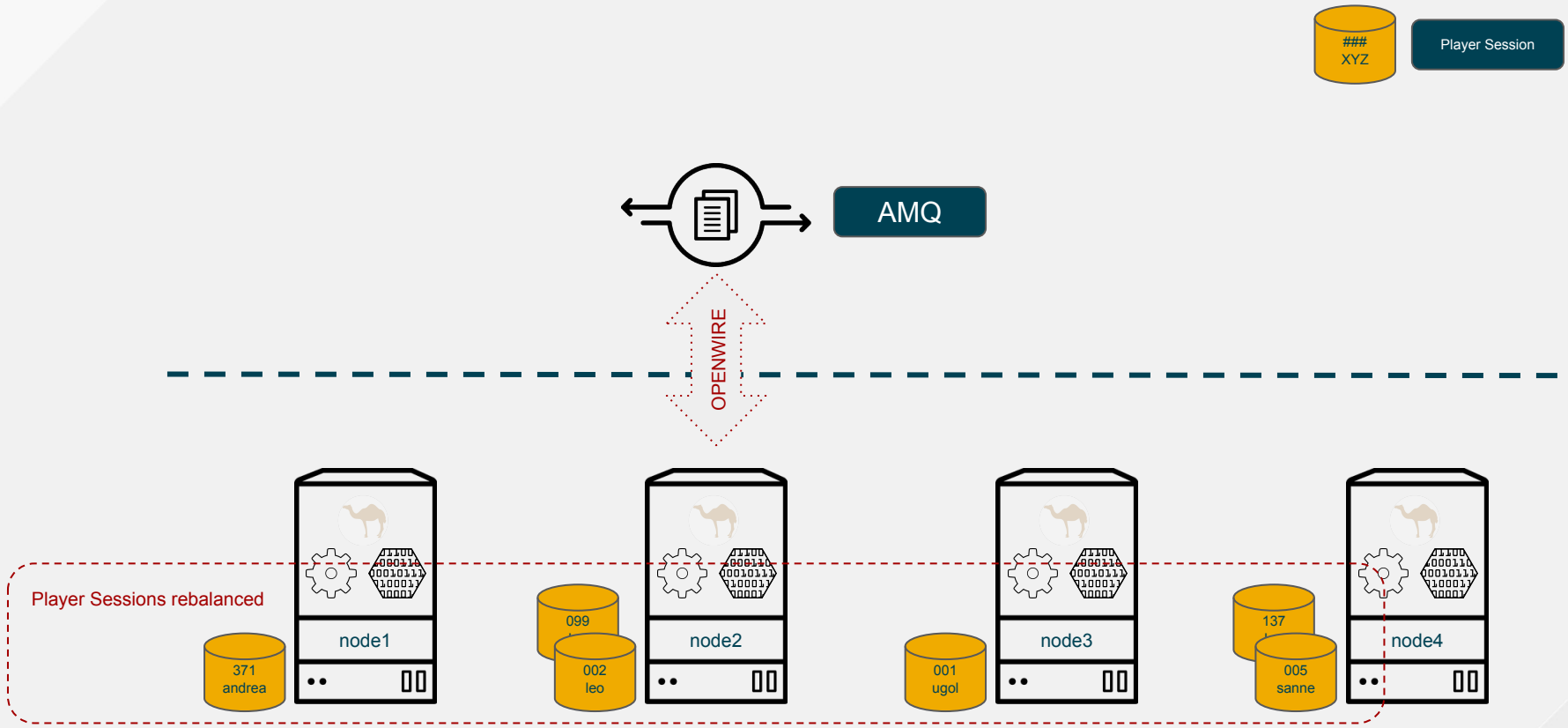
HACEP Topology

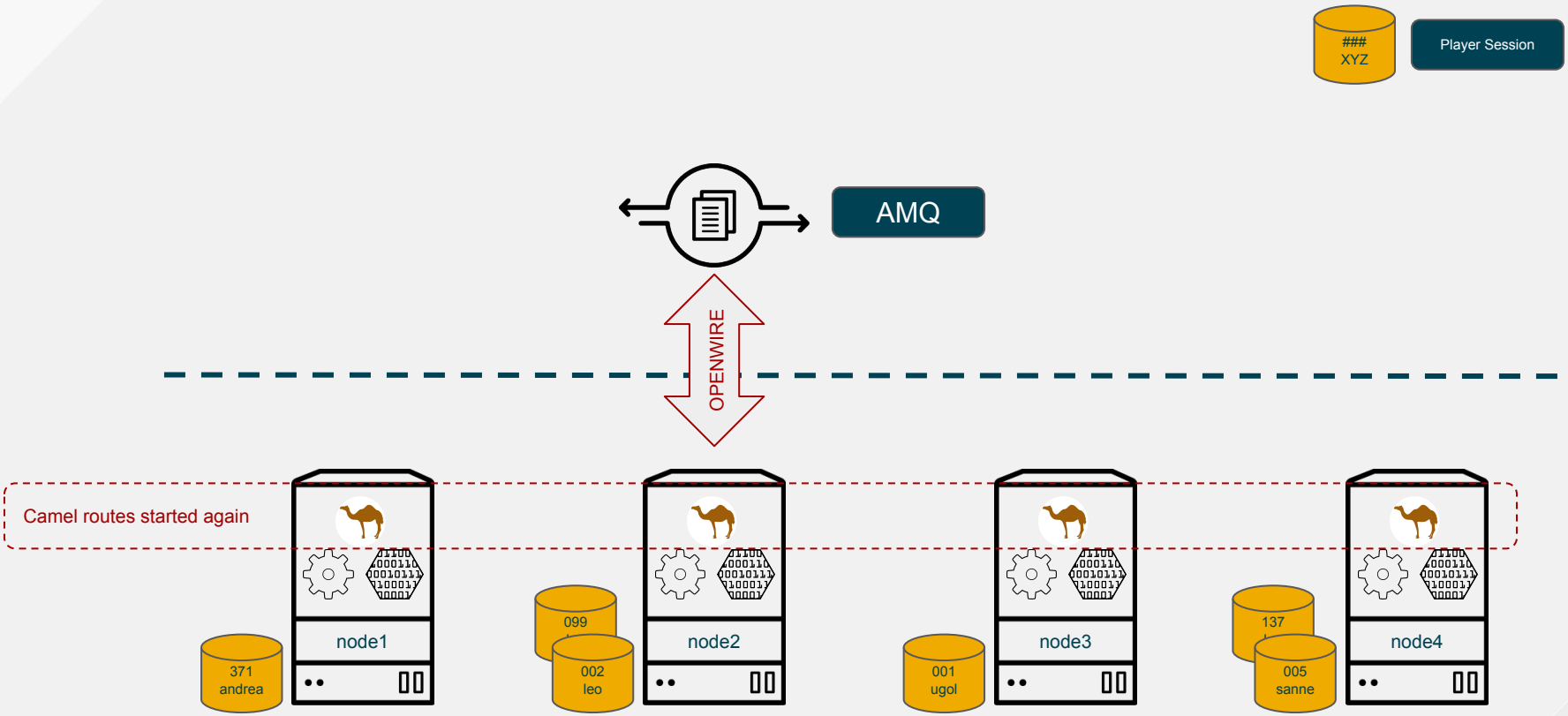
- If a node is added/removed to the cluster
  - Camel routes are automatically **stopped** when a rehashing event begins in the cluster
  - And **started** again when rehashing finishes


###
XYZ
Player Session
AMQ
OPENWIRE
005
371
ted
node1
137
099
002
leo
node2
001
UGO
node3


###
XYZ
Player Session
AMQ
OPENWIRE
005
371
andrea
node1
137
099
002
leo
node2
001
ugol
node3
node4
Add new
Node


###
XYZ
Player Session
AMQ
OPENWIRE
Camel routes stopped
005
371
andrea
node1
137
099
002
leo
node2
001
ugol
node3
node4


###
XYZ
Player Session
AMQ
OPENWIRE
Player Sessions rebalanced
node1
371
andrea
node2
099
002
leo
node3
001
ugol
node4
137
005
sanne


###
XYZ
Player Session
AMQ
OPENWIRE
Camel routes started again
node1
371
andrea
node2
099
002
leo
node3
001
ugol
node4
137
005
sanne

HACEP Topology

- Sessions will be redistributed following their **consistent hashing**, but always with the **right** group
- From now on, **events** will just flow in the **right nodes**

Events Ordering

- Events Ordering may or may not be important for the customer use case
- Events channel is **external** to HACEP
- **HACEP** proposes two different designs
  - JMS **grouping**
  - **Reordering** component

JMS Grouping

- **JMS grouping** could be used on event source server and is the preferred solution
- JMS grouping is conceptually similar to Infinispan one and gives us the guarantee that "same group" events are **consumed by the same thread**, thus guaranteeing message ordering (per group)
- Event source **must be** a JMS server like ActiveMQ and events **must contain** JMSXGroupId metadata

JMS Grouping

- Wouldn't it be nice if ActiveMQ could use the **same** Infinispan grouping algorithm so to consume messages directly on the "**right**" node?
  - Planned feature for HACEP 1.1
- Extending the **data affinity** concept
  - 100% nirvana compliant architecture!

Optional Reordering

- If **JMS grouping** isn't an option
- HACEP can internally reorder events on the nodes
  - Ordering based on a **configurable** event property
  - Could introduce **some latencies** due to buffering and gaps in events

Cool, how do I use HACEP?

- Decide for **plain JVM** or **EAP** version
- Make your business events implement **Fact** interface
  - Decide your **grouping** criteria (is a simple String)
- Configure Message ordering
  - With **JMSXGroupId** metadata or with optional Reordering
- Plan how many nodes you need and start the cluster

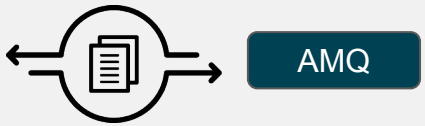
LAB 1: INSTALLATION

What you will do...

- Install and configure a complex distributed application with:
  - **4 HACEP** nodes, deployed on **EAP 7** in domain mode
  - **1 AMQ** node
  - **1** Simple Java application injecting player events in **AMQ** (from terminal)

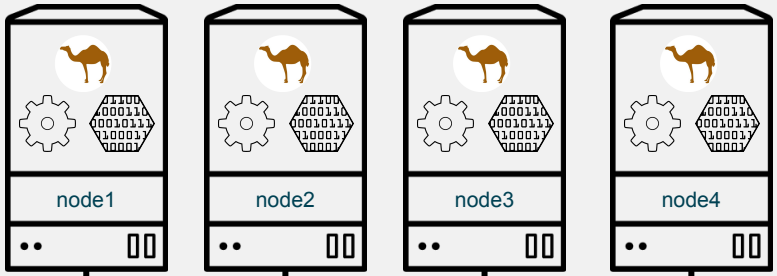
- What you will see:
  - Simple rule outcomes published on **AMQ** (in a different queue)
  - A custom Javascript (D3) application consuming outcomes via **STOMP, graphically** showing **nodes**, their BRMS **sessions** and simple **player rewards**

HACEP LAB Architecture



AMQ

OPENWIRE



JGroups

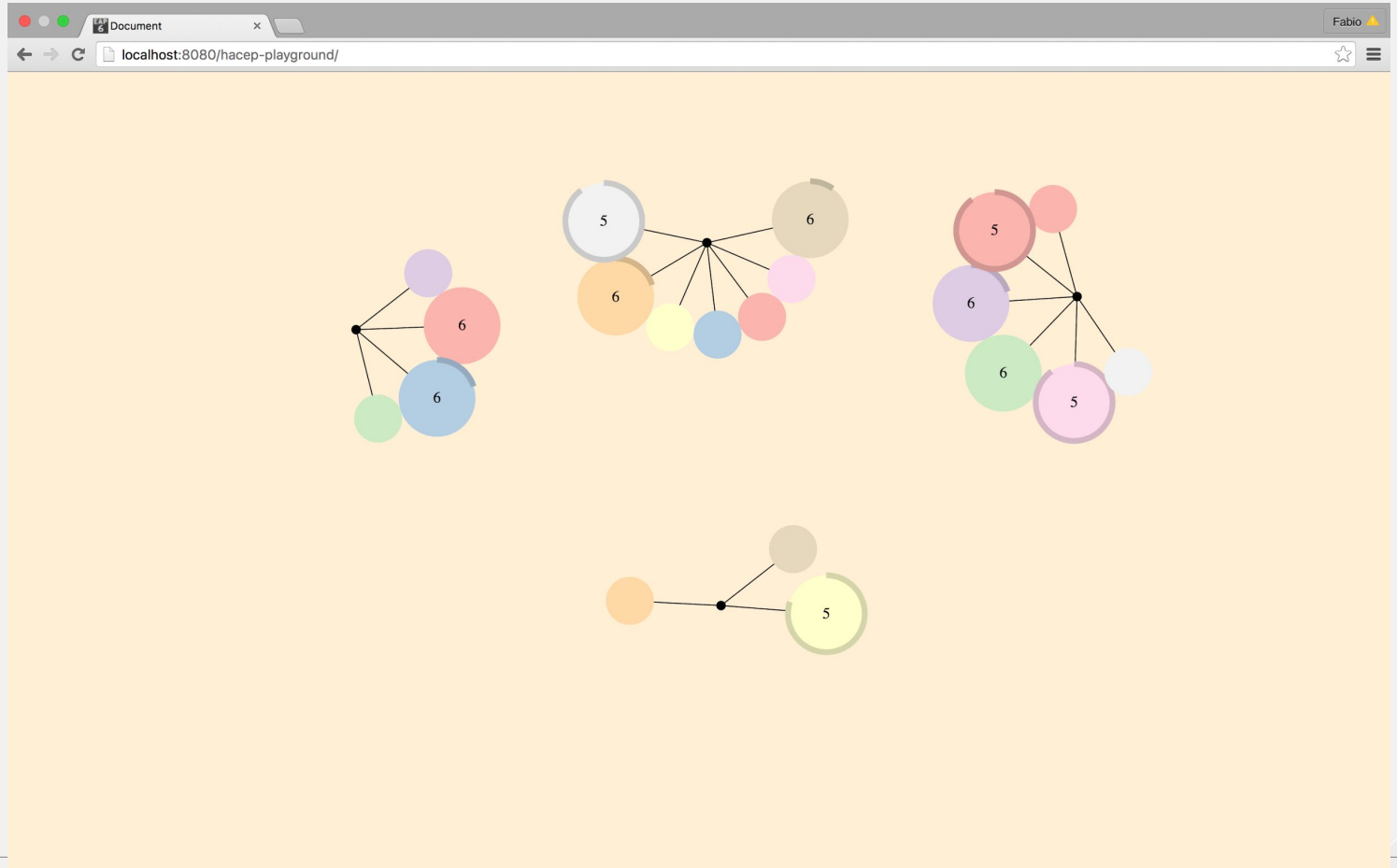
BRMS

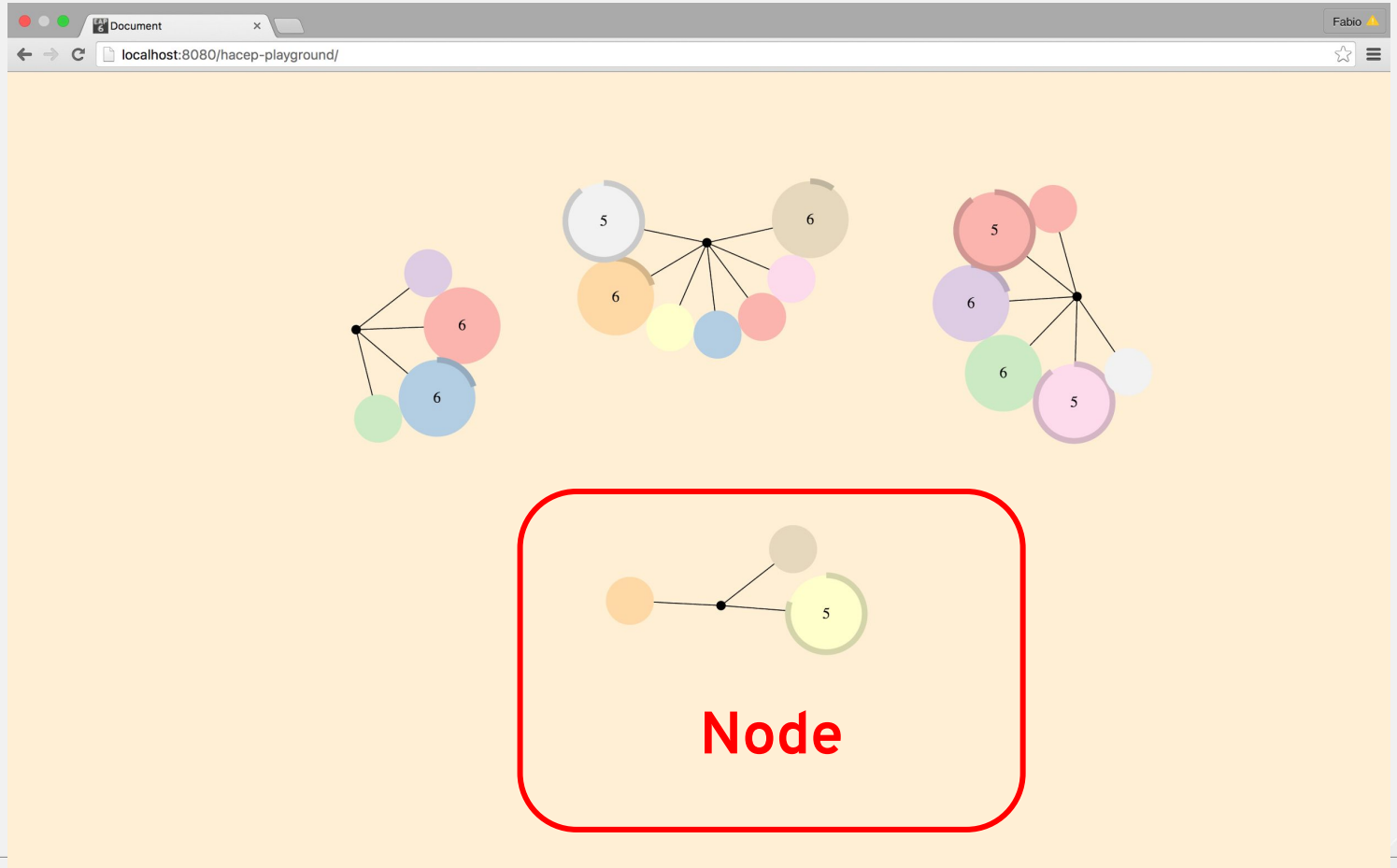
Camel

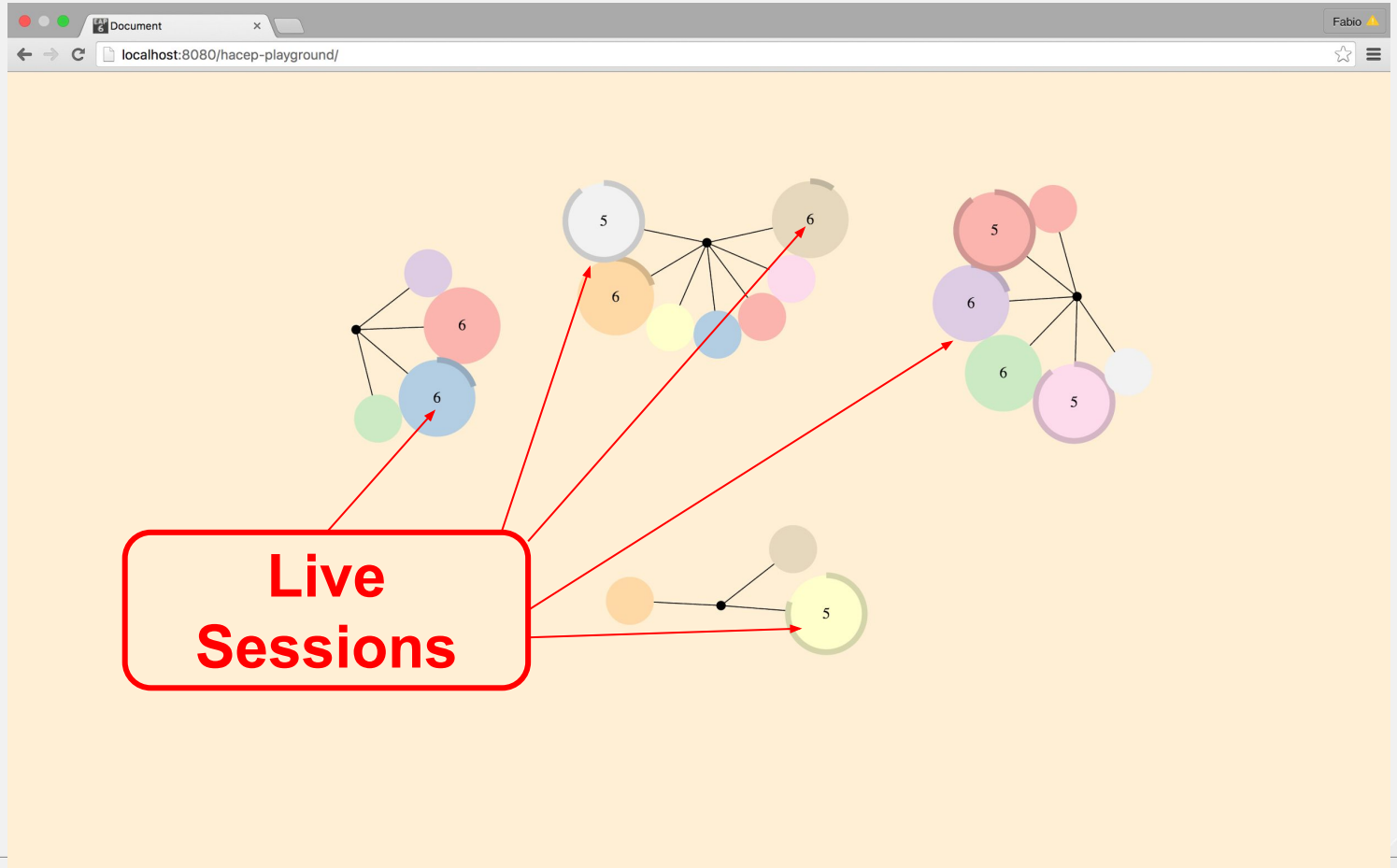
Grid

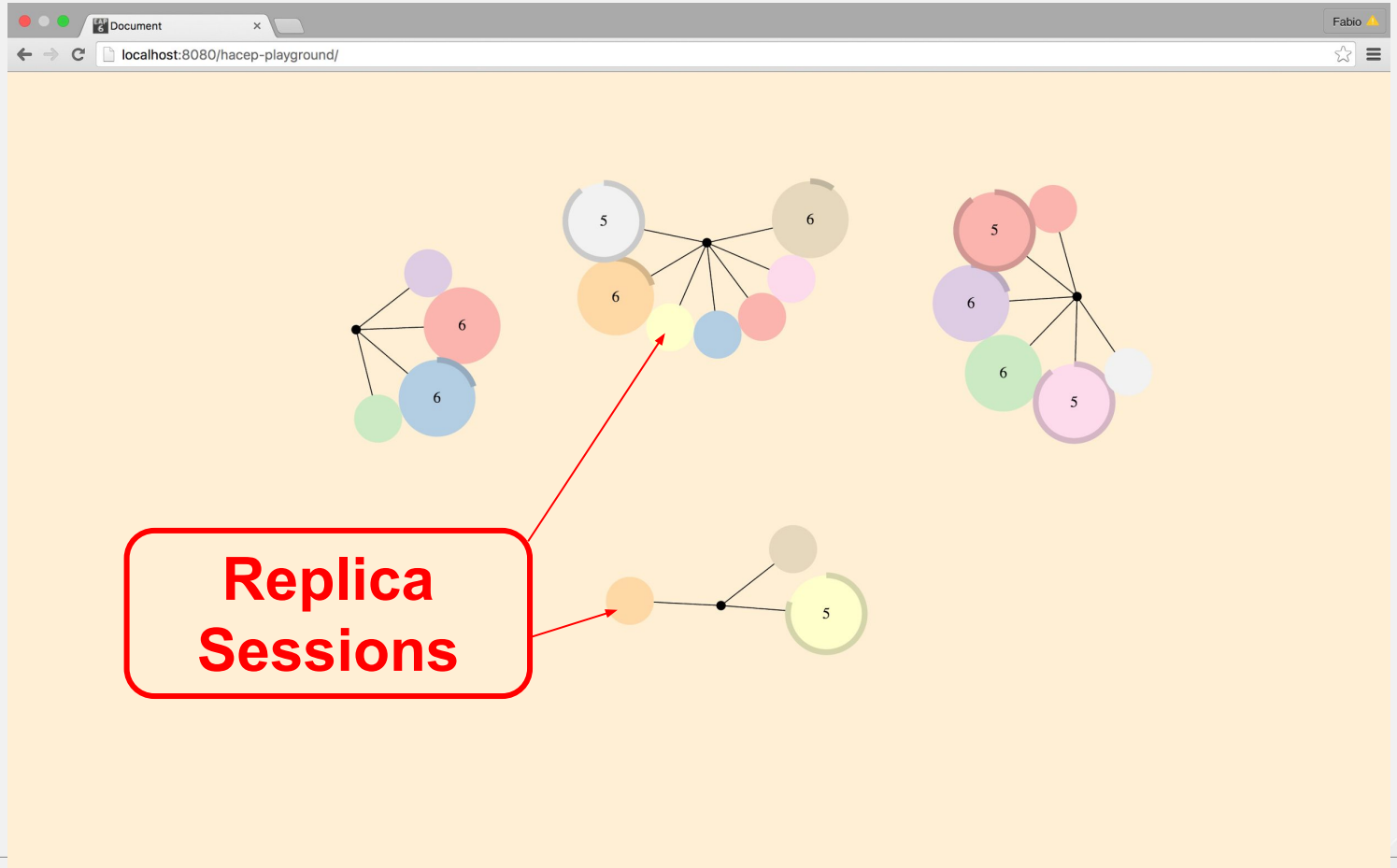
A simple Player reward rule

```
rule "User gains a point for each gameplay and every 10 points increases player level"
when
    $gamePlay : Gameplay($playerId : playerId) over window:length(1)
    $numberOfTimes : Number()
                  from accumulate ($gamePlayCount :
Gameplay($gamePlay.playerId == playerId) over window:time(30d),
                  count($gamePlayCount))
then
    channels["playerPointsLevel"].send(new PlayerPointLevel(
        $playerId,
        $numberOfTimes.intValue() % 10,
        $numberOfTimes.intValue() / 10)
    );
end
```


Document
Fabio
localhost:8080/hacep-playground/
6
6
5
6
6
5
6
6
5
5


Document
Fabio
localhost:8080/hacep-playground/
5
6
6
5
6
6
6
5
5
Node


Document
Fabio
localhost:8080/hacep-playground/
5
6
6
6
5
6
6
5
5
Live
Sessions


Document
Fabio
localhost:8080/hacep-playground/
5
6
6
6
5
6
6
5
5
Replica
Sessions

To begin..

- Start the lab vm
- For your convenience update the vm screen resolution and go fullscreen
- You should find a PDF with all the instructions on your desktop: open it and have fun!
- If you get stuck or have any kind of question, just ask

LAB 1: INSTALLATION

LAB 2: MODIFY THE RULES

Simple modification of the rewards

```
rule "User gains a point for each gameplay and every 10 points increases player level"
when
    $gamePlay : Gameplay($playerId : playerId) over window:length(1)
    $numberOfTimes : Number()
                    from accumulate ($gamePlayCount :
Gameplay($gamePlay.playerId == playerId) over window:time(30d),
                    count($gamePlayCount))
then
    channels["playerPointsLevel"].send(new PlayerPointLevel(
        $playerId,
        $numberOfTimes.intValue() % 10,
        $numberOfTimes.intValue() / 10)
    );
end
```

LAB 2: MODIFY THE RULES

LAB 3: UPDATE THE RULES, THE RIGHT WAY

Reverting the simple modification of the rewards

```
rule "User gains a point for each gameplay and every 10 points increases player level"
when
    $gamePlay : Gameplay($playerId : playerId) over window:length(1)
    $numberOfTimes : Number()
                    from accumulate ($gamePlayCount :
Gameplay($gamePlay.playerId == playerId) over window:time(30d),
                    count($gamePlayCount))
then
    channels["playerPointsLevel"].send(new PlayerPointLevel(
        $playerId,
        $numberOfTimes.intValue() % 10,
        $numberOfTimes.intValue() / 10)
    );
end
```

LAB 3: UPDATE THE RULES, THE RIGHT WAY

HACEP INTERNALS

HAKie*Session

- **HAKie*Session**s are special objects that we save in the grid
  - instead of plain Drools **KieSession**s
- We differentiate between live sessions and replica sessions
  - Live Sessions contains a **KieSession**
  - Replica sessions contains a **buffer** of events and a **snapshot** of a KieSession

Replica Sessions Buffers

- The **buffer** is used to maintain a list of events without having a live **KieSession**
  - Event sourcing pattern at work
- HACEP uses Infinispan **DeltaAware** apis to minimize network traffic: only the event itself is transmitted from the live session to the replica
- **Snapshots + buffer** is all we need to recreate a live session in case of failures

Snapshots

- Sessions **snapshots** are very useful to avoid replaying the whole ever growing buffer in case of failures
- Every node applies its buffer of events to its previously stored session, asynchronously
  - At the moment there is only a configurable *size-based* policy
  - More policies are planned for HACEP 1.1
- Again, no big serialized sessions **ever travel on the network**, not even when doing snapshots

Idempotency

- When events in the **buffer** are replayed they must be discarded to avoid duplication
- A **ReplayChannel** is dynamically injected in the session, substituting real channels
  - all the events that we know for sure have already been fired will be **discarded**
  - Null Object Pattern + Command Pattern

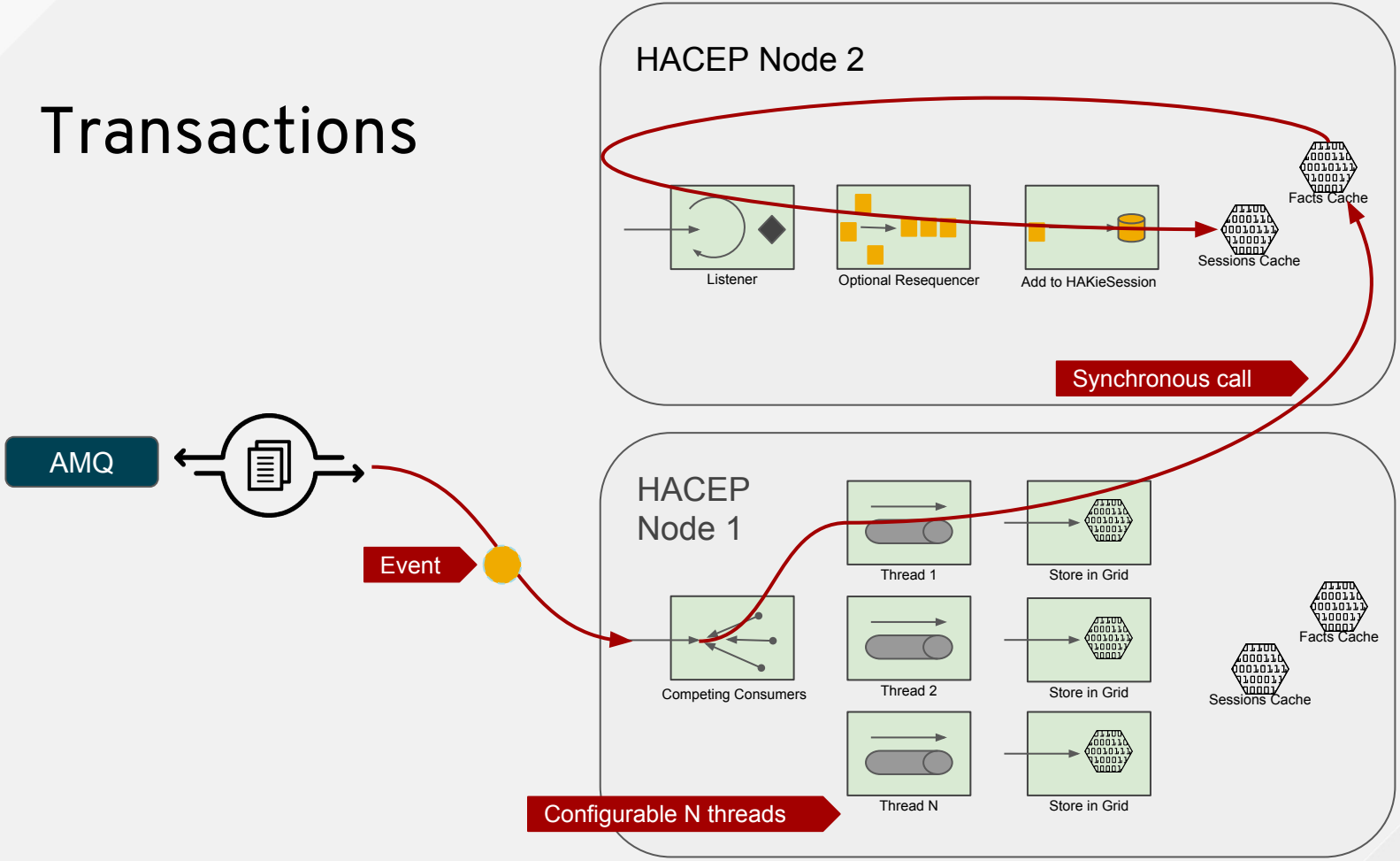
Idempotent channels

- **Idempotent channels**, due in HACEP 1.1, are useful in cases in which a rule could fire **many** actions on different external systems
- If you need to replay only some of them (let's say a node crashes in the middle of a multi-action) you need to know which actions have been already executed and which haven't
- **Idempotent channels** will be implemented using the grid as an **idempotent repository** for command actions

Transactions

- Every Hacep node has configurable **ActiveMQ consumers** (threads) serving all its sessions
- Consumers use ActiveMQ **grouping**
- Every consumer is completely **synchronous**
  - Infinispan **notifications** are **synchronous** too
- Messages are consumed **in order** (per group) and won't leave their queue if something goes wrong

Transactions



HACEP ROADMAP

HACEP ROADMAP

Planned Features

HACEP 1.1

- Spring Boot version
- Make ActiveMQ grouping **aware of Infinispan topology** and therefore "*plugin*" Infinispan consistent hashing in ActiveMQ
  - With this design even Infinispan notifications will be **local** in the vast majority of cases
- **Idempotent** replay channels
- **Pluggable policies** for snapshots
  - Cron policies
  - Only when the rules fire, or never
  - Scripting

Planned Features

HACEP 1.2

- Scaling out of **unpartitionable Drools sessions** using "**map/reduce**" like Infinispan collections API
- Migrate from DeltaAware to new **command-based functional** approach when we'll support it (at the moment it's just in Infinispan community bits)

CONCLUSIONS

CONCLUSIONS

- **HACEP** can easily scale **horizontally**, from 2 nodes to 100s of nodes if needed, even dynamically at runtime
- **HACEP** is **inherently** HA: the minimal HA deployment needs just 2 nodes

Links and Contacts

- **HACEP** is open to contributions:
  - https://github.com/redhat-italy/hacep
  - http://redhat-italy.github.io/hacep
- **HACEP** reference architecture
  - https://access.redhat.com/articles/2542881
- Free code downloads for development use
  - JBoss Data Grid
    http://developers.redhat.com/products/datagrid/download/
  - JBoss BRMS
    http://developers.redhat.com/products/brms/download/
  - JBoss Fuse
    http://developers.redhat.com/products/fuse/download/
  - JBoss A-MQ
    http://developers.redhat.com/products/amq/download/
- Contact us: hacep@redhat.com

THANK YOU


plus.google.com/+RedHat


facebook.com/redhatinc


linkedin.com/company/red-hat


twitter.com/RedHatNews

youtube.com/user/RedHatVideos

LEARN. NETWORK.
EXPERIENCE
OPEN SOURCE.