

**RED HAT
SUMMIT**

10 YEARS *and counting*
SAN FRANCISCO | APRIL 14-17, 2014

High Available Complex Event Processing

with Red Hat JBoss BRMS

Duncan Doyle
Senior Architect, Red Hat

Who am I?

- Duncan Doyle
- Senior Architect @ Red Hat Consulting
- EMEA Architect Team
- JBoss Middleware
 - EAP
 - SOA-P
 - Data Grid
 - BRMS/BPMSuite
- Rotterdam, The Netherlands

Agenda

- What is Complex Event Processing (CEP)?
- What is High Availability (HA)?
- Challenges when combining HA and CEP
- Architectural Quest
 - Persistence Architecture
 - Replay Architecture
 - Active-Active Architecture
- Keeping things in-sync
- Conclusion

What is Complex Event Processing?

- What is an Event?

A significant change of state at a particular point in time.

- What is Complex Event Processing?

The ability to detect, correlate, abstract, aggregate or compose and react to events.

Red Hat JBoss BRMS Enables:

Event Detection

From an event cloud or set of streams, select all the meaningful events and only then:

(Temporal) Event Correlation

Ability to correlate events and facts declaring both temporal and non-temporal constraints between them. Ability to reason over event aggregation.

Event abstraction

Ability to compose complex events from atomic events AND reason over them.

A small CEP Example

```
rule "Baggage lost at sorting"  
when  
    $b1:BagScannedEvent( location == Location.CHECK_IN )  
    not BagScannedEvent( bagTag == $b1.bagTag, location == Location.SORTING, this after[0s,10m] $b1 )  
then  
    System.out.println("Baggage lost at sorting");  
end
```


What is High Availability?

High availability is a system design approach and associated service implementation that ensures a prearranged level of operational performance will be met during a contractual measurement period.

- 0-downtime?
- MTBF, MTTR?
- Requirements?
 - Is limited downtime acceptable?
 - How much downtime is acceptable?
 - Is state-loss acceptable?
 - Costs, Probability, Risks?

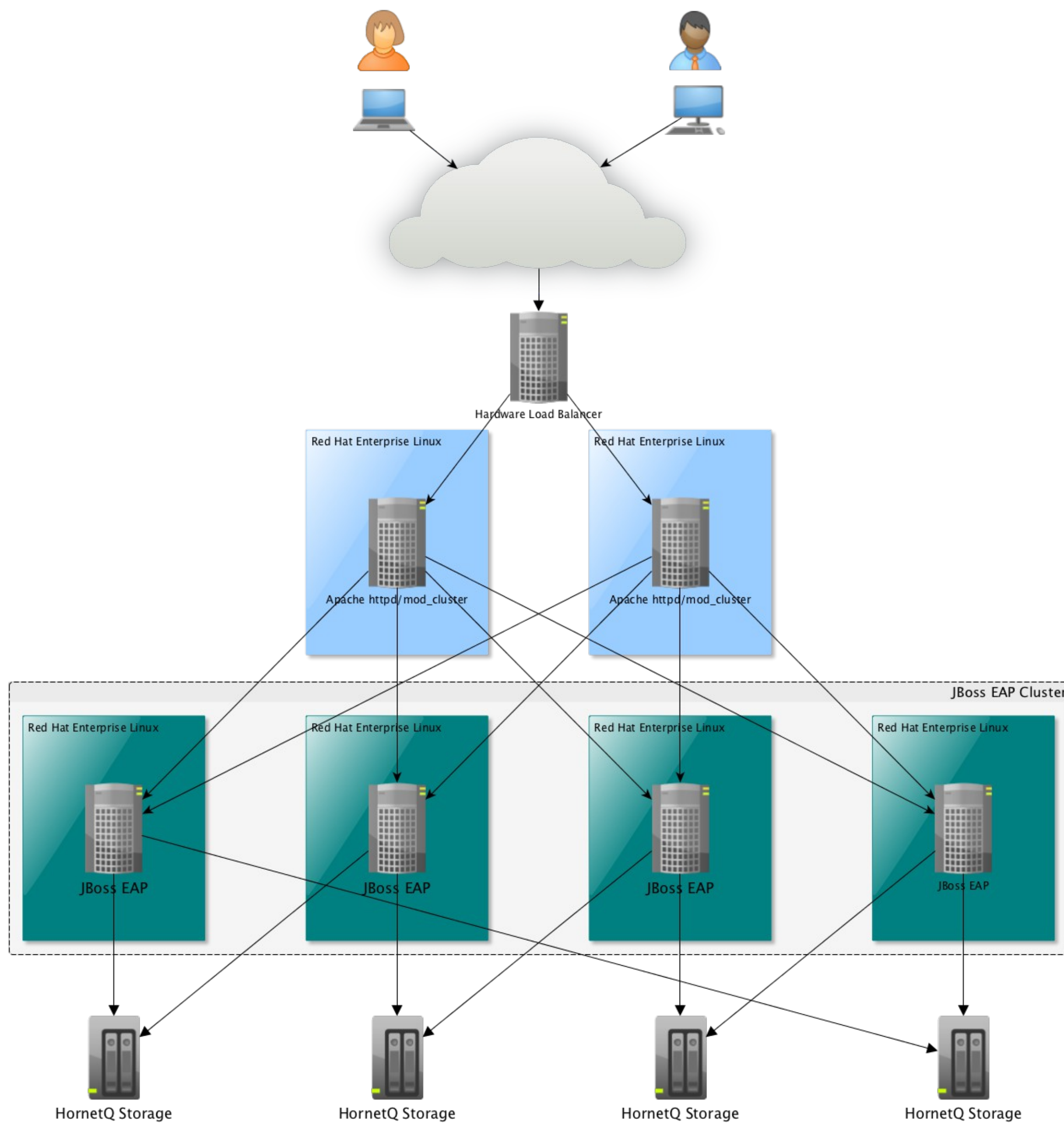
JBoss EAP High Availability

Multi-node (active-active) setup (HA):

- mod_cluster
- sticky-sessions
- application(s) deployed to all nodes.

Clustered setup (HA + Failover):

- JGroups cluster:
 - Web Session replication (Infinispan)
 - SFSB replication (Infinispan)
- HornetQ cluster
 - Cluster (HA)
 - Live/Backup (Failover)



JBoss BRMS HA CEP Challenges

Sessions are:

- Stateful

State needs to be persisted.

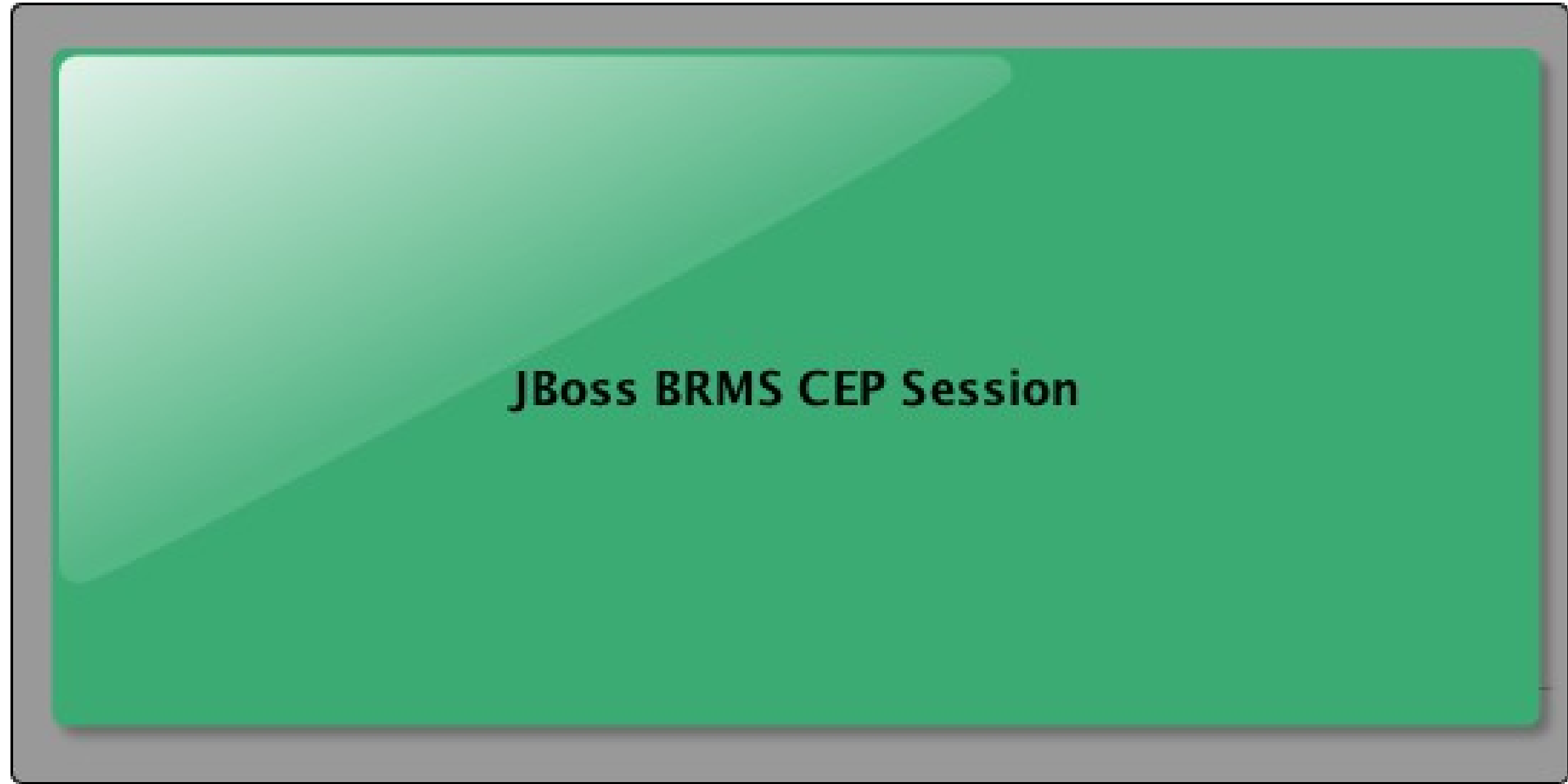
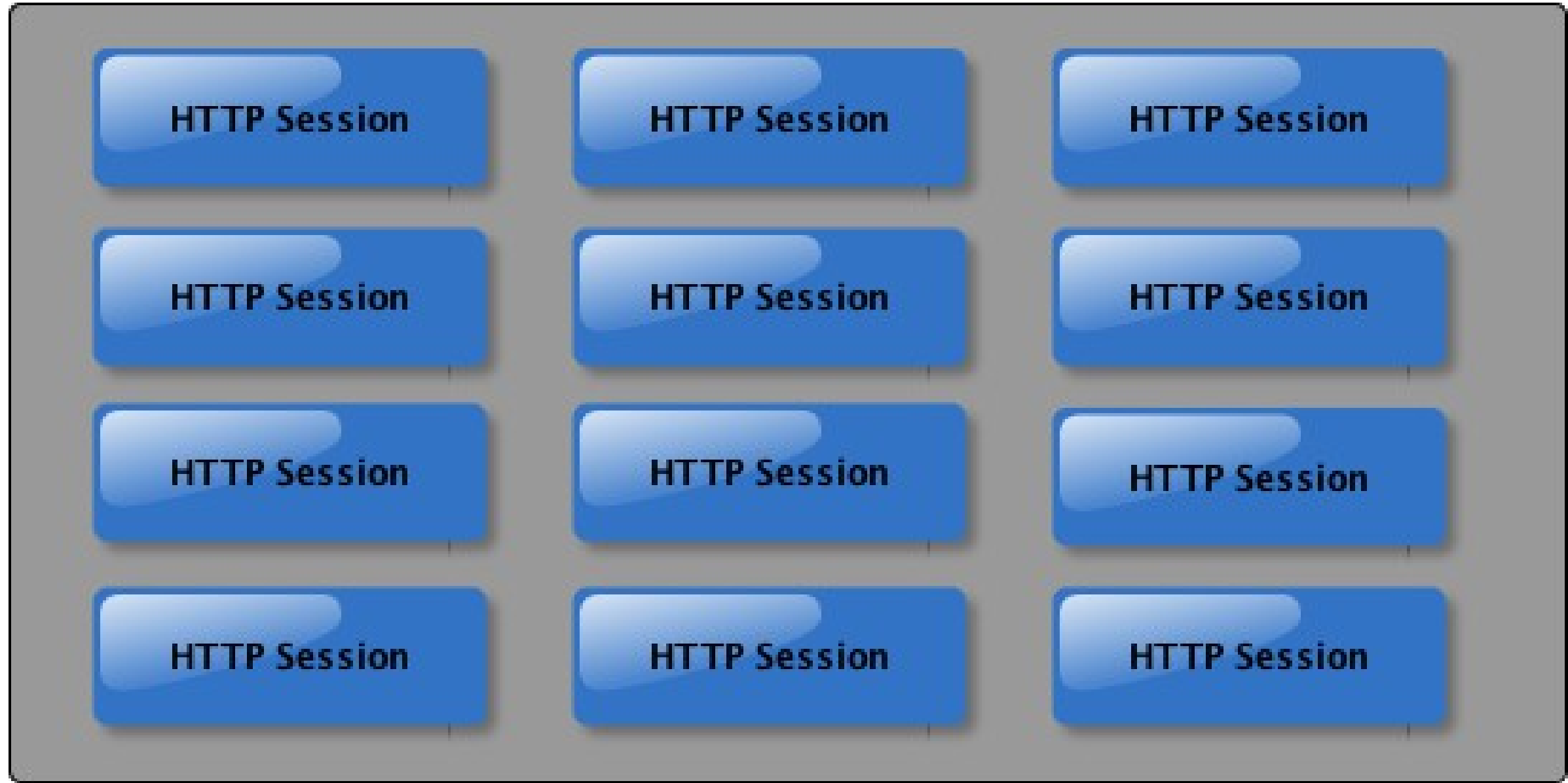
- Used by all Events

Sessions are not partitioned per event type, user, session-id, etc (like, for example, HTTP sessions).

- Long lived sessions (days), which are (usually) very Large and therefore consume a lot of memory

Full persistence of state is expensive.

Unless we can partition our events, all events are stored in the same session!

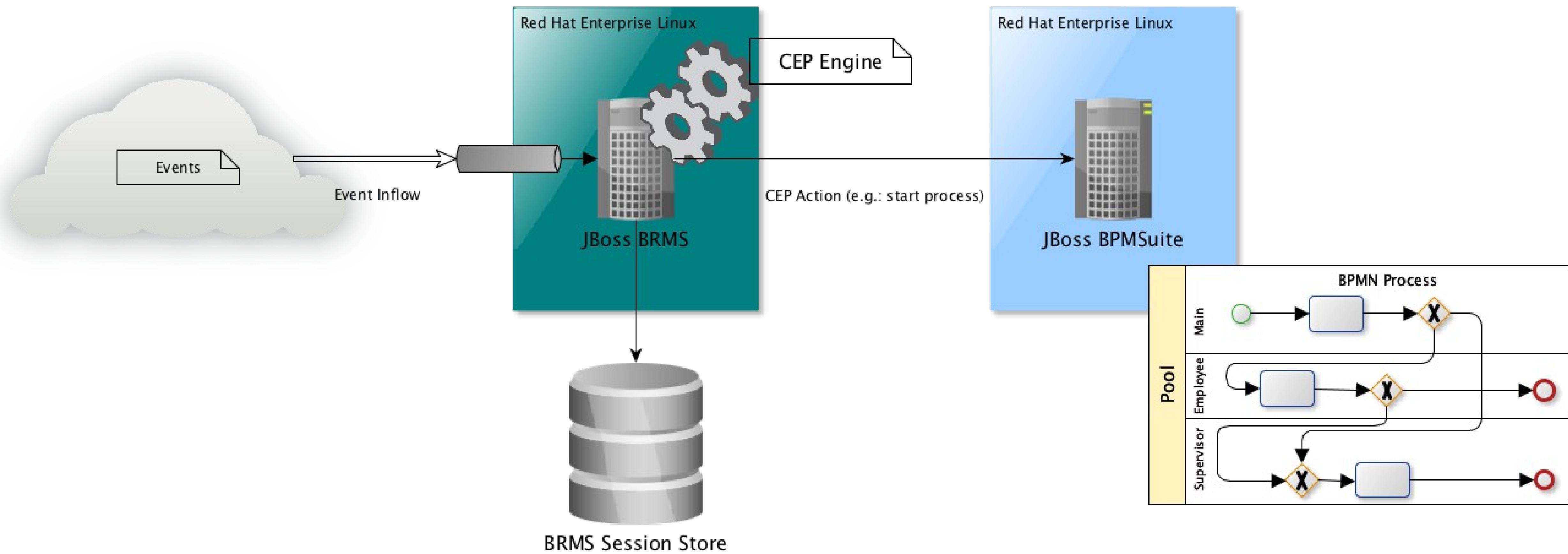


Persistence Architecture

The session is persisted after every event insert + rule firing.

On system failure, the system is restarted and the session is reloaded from the persistence store.

Session can be stored on HA storage (Gluster, SAN, etc.).



Persistence Architecture

Pro:

- Easy to implement.
- Easy to maintain.
- Requires only a single processing node.

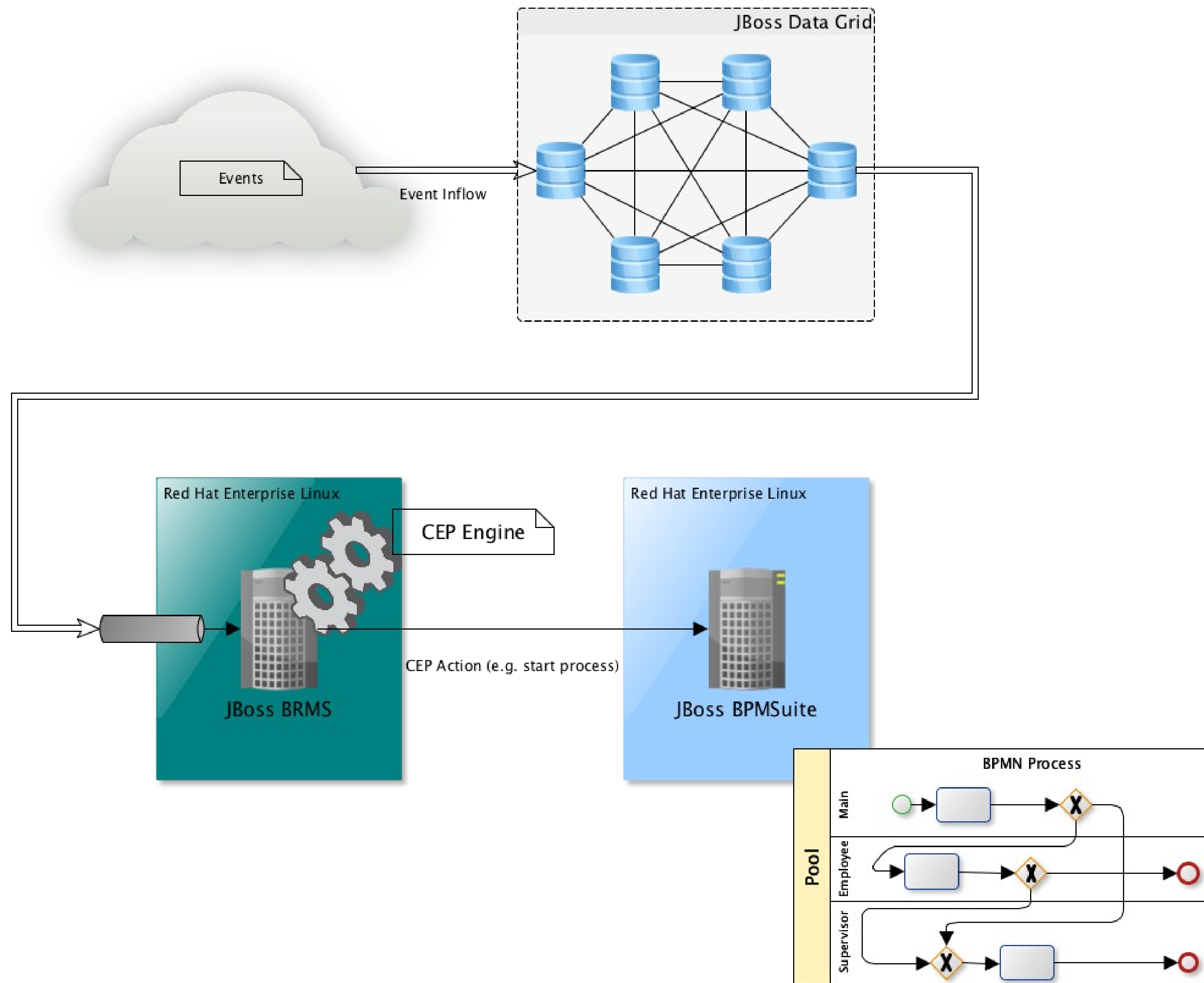
Con:

- System is down during restart and session restore.
- Very naïve implementation: Performance
- and Performance
- and Performance

Replay Architecture

Events are stored in big-data storage (e.g. JBoss Data Grid).

On system failure, a new JBoss BRMS CEP Session is created and events are replayed to restore the state of the session.



Replay Architecture

Pro:

- Requires only a single processing node.
- No continuous session persistence.

Con:

- System is down during restart, restore and replay.
- Event persistence maintenance (store/remove message)
 - How to re-create the exact same state of the system?
 - When can we discard events?
- Duplicate output (i.e. rule RHS executed twice for replayed events).

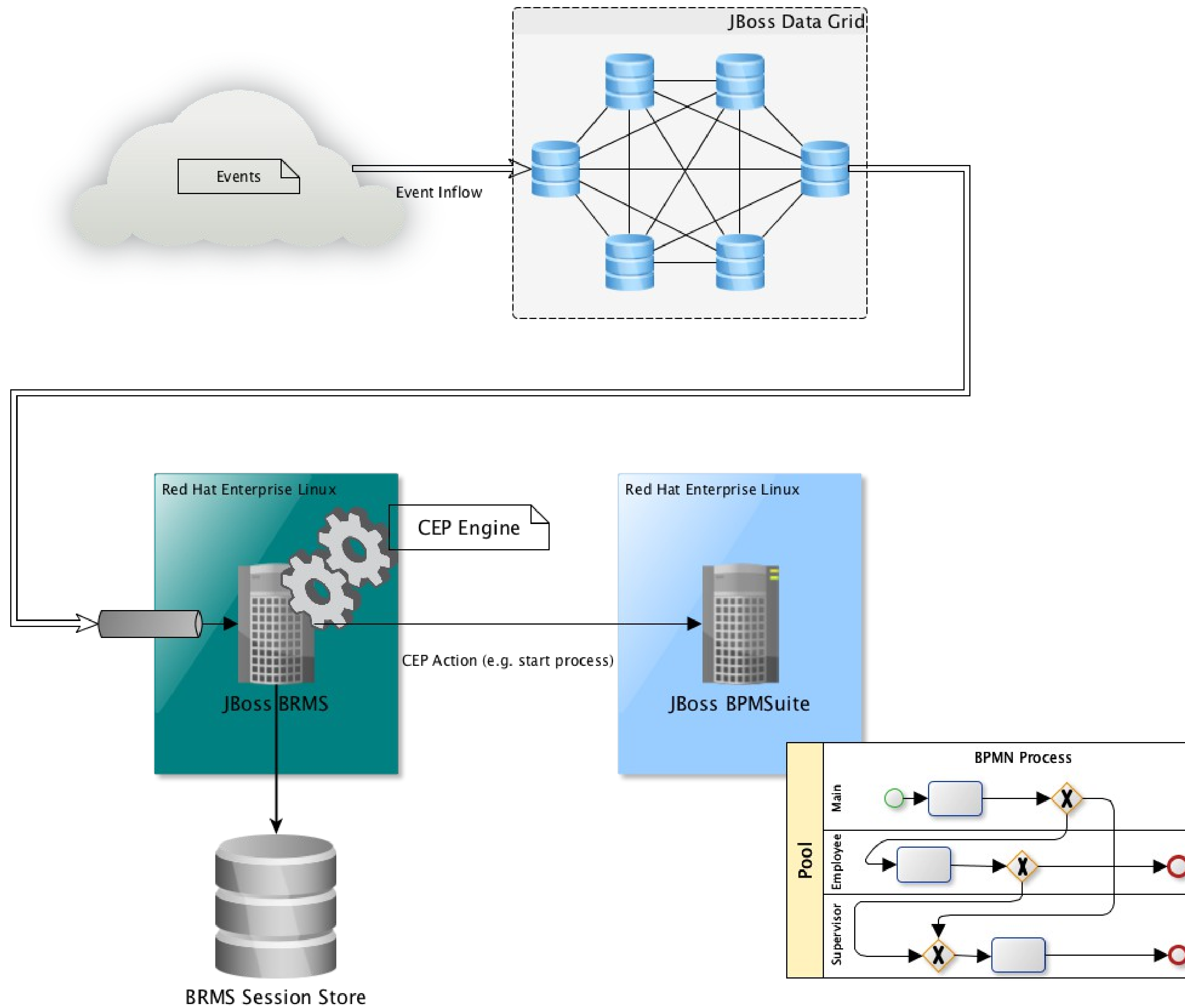
Replay & Persistence Architecture

Events are stored in big-data storage (e.g. JBoss Data Grid).

Sessions are stored periodically creating save-points.

On system failure, restore the session from the save-point, and replay the events to restore the full state of the session (Event Sourcing).

Resolves (part of) the event persistence maintenance problem. Events can be discarded when the session is saved.



Replay & Persistence Architecture

Pro:

- Requires only a single processing node.
- Easier maintenance. Events can be discarded when the save-point has been created.

Con:

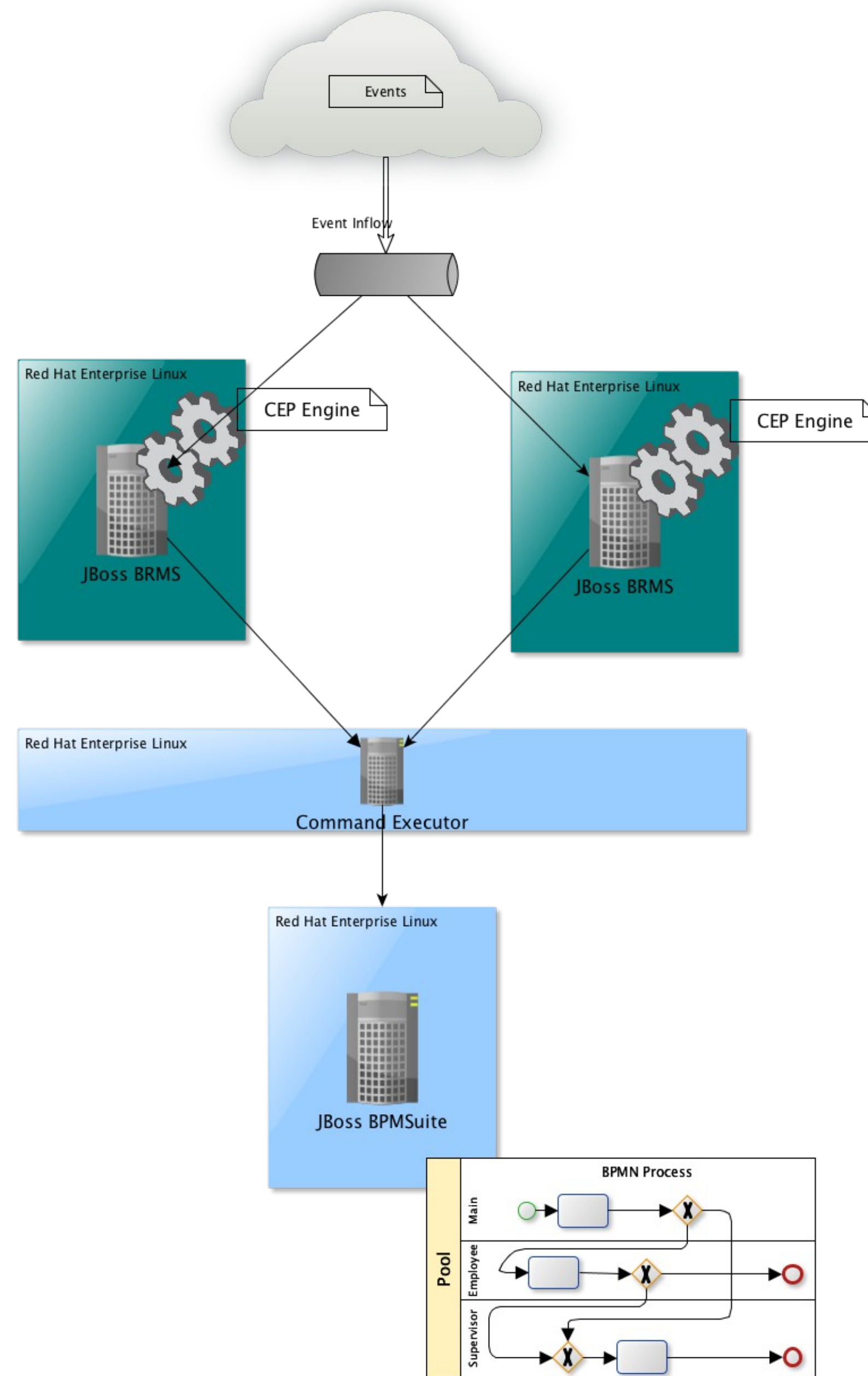
- System is down during restart, restore and replay.
- Duplicate output (i.e. rule RHS executed twice for replayed events).

Active-Active Architecture

2 + n nodes processing the same events in parallel.

No persistence: on failure, the other node will take over.

Only one node executes the rule consequence.



Active-Active Architecture

Pro:

- Seamless failover (no event replay required).
- No need to store events.

Con:

- Multiple nodes required.
- Sessions need to be kept in-sync.
- As there are multiple CEP engines firing rules, the of the engines needs to be controlled. I.e. only one of the nodes is allowed to execute the rule consequence (or action).
- How do we revive a failed instance?

Active-Active + Persistence + Replay Architecture

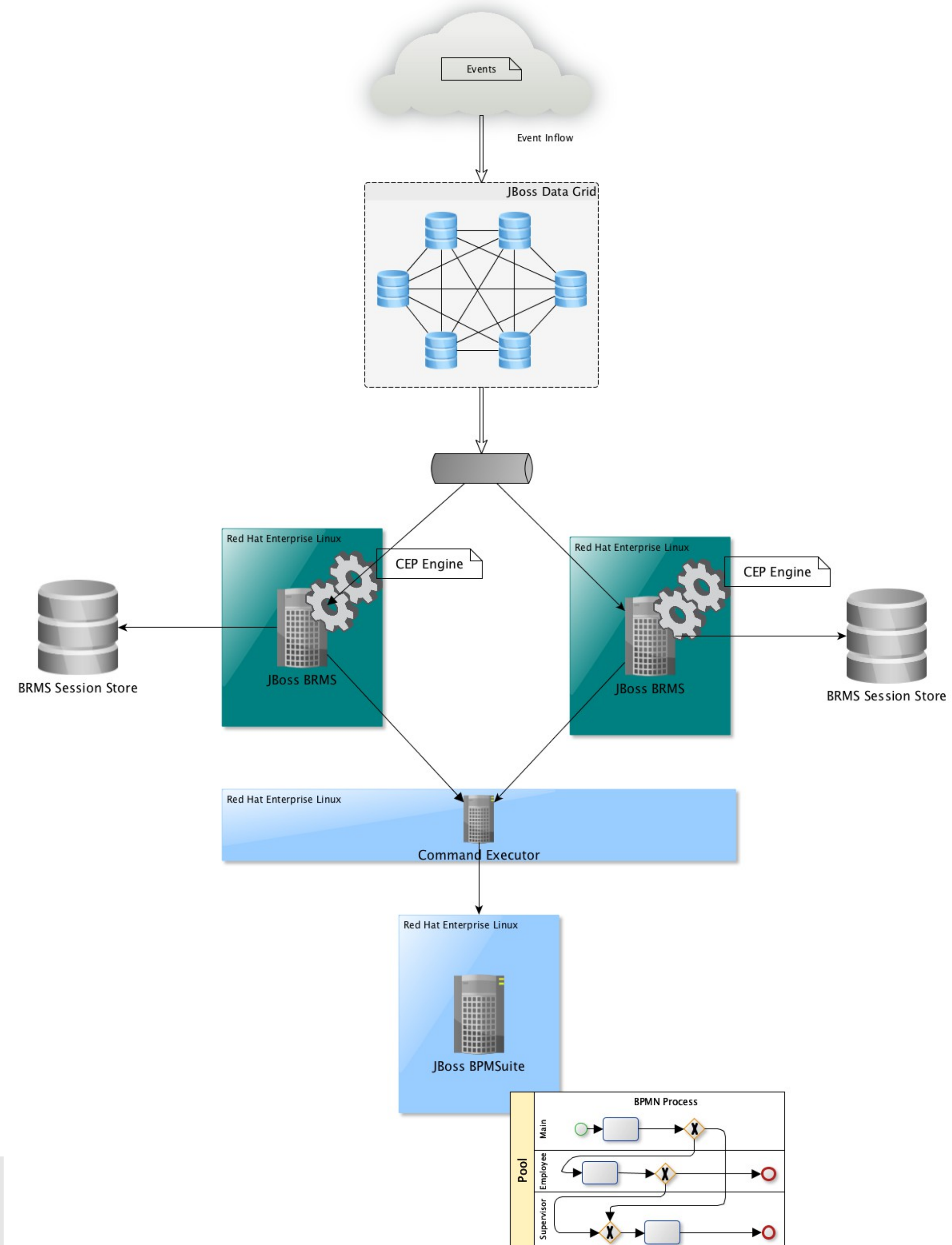
2 + n nodes processing the same events in parallel.

Sessions are stored periodically creating save-points.

On system failure, restore the session from the save-point, and replay the events to restore the full state of the session.

Only one node executes the rule consequence.

Event Sourcing: the state the systems are kept in-sync by the 'Event Sourcing' principal:
The state of the system can be recreated by replaying the events on a base-line.



Active-Active + Persistence + Replay Architecture

Pro:

- Seamless failover (no event replay required).
- Easier maintenance. Events can be discarded when the save-point has been created.
- Instances can be restored by loading the save-point and replaying events.

Con:

- Multiple nodes required.
- Sessions need to be kept in-sync.
- As there are multiple CEP engines firing rules, the of the engines needs to be controlled. I.e. only one of the nodes is allowed to execute the rule consequence (or action).

Keeping Stuff In-Sync

Agenda

- Agenda contains the matches/activations of rules that will be fired. These need to be in-sync so on failover, the backup node will continue firing rules where the master stopped.

Events

- Each node needs to receive the same events in the same order.

Clock

- Due to temporal reasoning, the clock of the nodes needs to be kept in-sync to keep the WM and Agenda in-sync.

Working Memory

- The WorkingMemory (session state) needs to be kept in exact sync so the Agenda will be kept in-sync.

Keeping the Agenda in-sync

Why can't we just discard rule-firing on the slave? I.e using an AgendaFilter:

- On failover, the backup node needs to continue firing rules where the master stopped.
- RHS of a rule can influence the state of the WM, i.e. insert, update, discard Events/Facts, or push agenda groups.

Therefore, rules need to be put on the agenda and fired on both the master and the slaves. Output of the slaves needs to be discarded.



Keeping Events in-sync

- All Events have a UUID and a timestamp.
- Direct, non-deterministic manipulation of the session is not allowed (e.g. insert/update/delete fact).
- Events need to be processed in the exact same order on both engines.
 - For example by using strict-message ordering at the receiver.

Synchronizing the Clock

Due to the temporal reasoning in the engine, the clock needs to be kept in-sync.

Easiest solution: Use the pseudo-clock and advance the clock based on the timestamp of the events:

```
long advanceTime = fact.getTimestamp() - pseudoClock.getCurrentTime();
if (advanceTime > 0){
    pseudoClock.advanceTime(advanceTime, TimeUnit.MILLISECONDS);
}
```

This keeps the clocks in-sync without communication between the nodes, as long as the order of events entering the systems is the same on all nodes.

Working Memory Synchronization

By keeping the clock in-sync, keeping the order of events in-sync and by adhering to these constraints, the working memory of the engines can be kept in-sync:

- All Events have a UUID and a timestamp.
- Use the pseudo-clock and advance clock based on the timestamp of the incoming fact.
- Direct, non-deterministic manipulation of the session is not allowed (e.g. insert/update/delete fact).
- CEP Engine actions are wrapped in Command objects. Engine is not allowed to directly interact with other systems.
- Command Objects have an ID that can be deterministically computed in the individual nodes.

Working Memory Synchronization

AND DON'T USE
`ksession.fireAllRules()!!!!`

Rules need to be fired after the same event 'insert' on both engines for the engines to behave deterministically. This can for example be done by:

- Firing after a defined 'batch' size.
- Using 'poison-pills'.

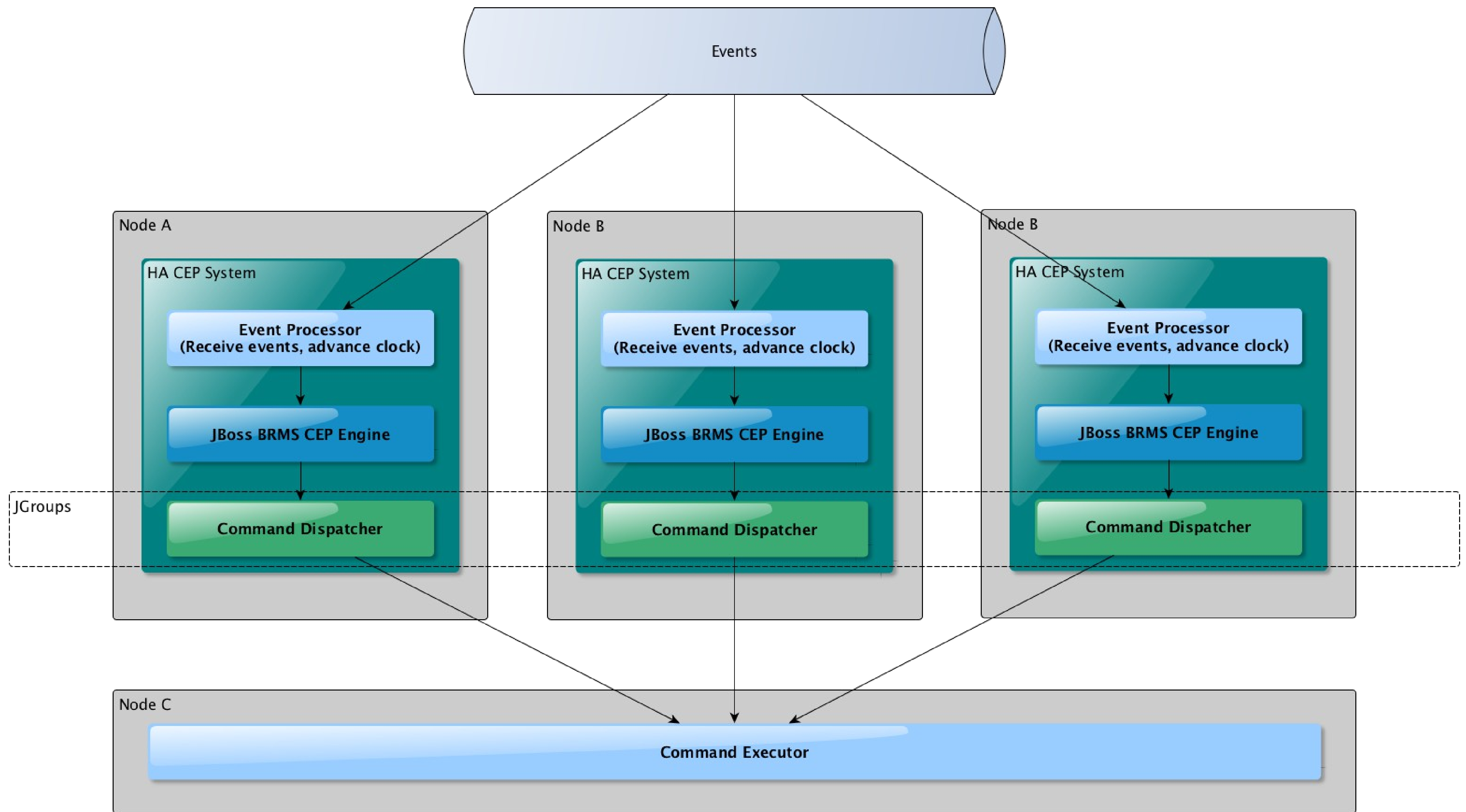
Working Memory Synchronization

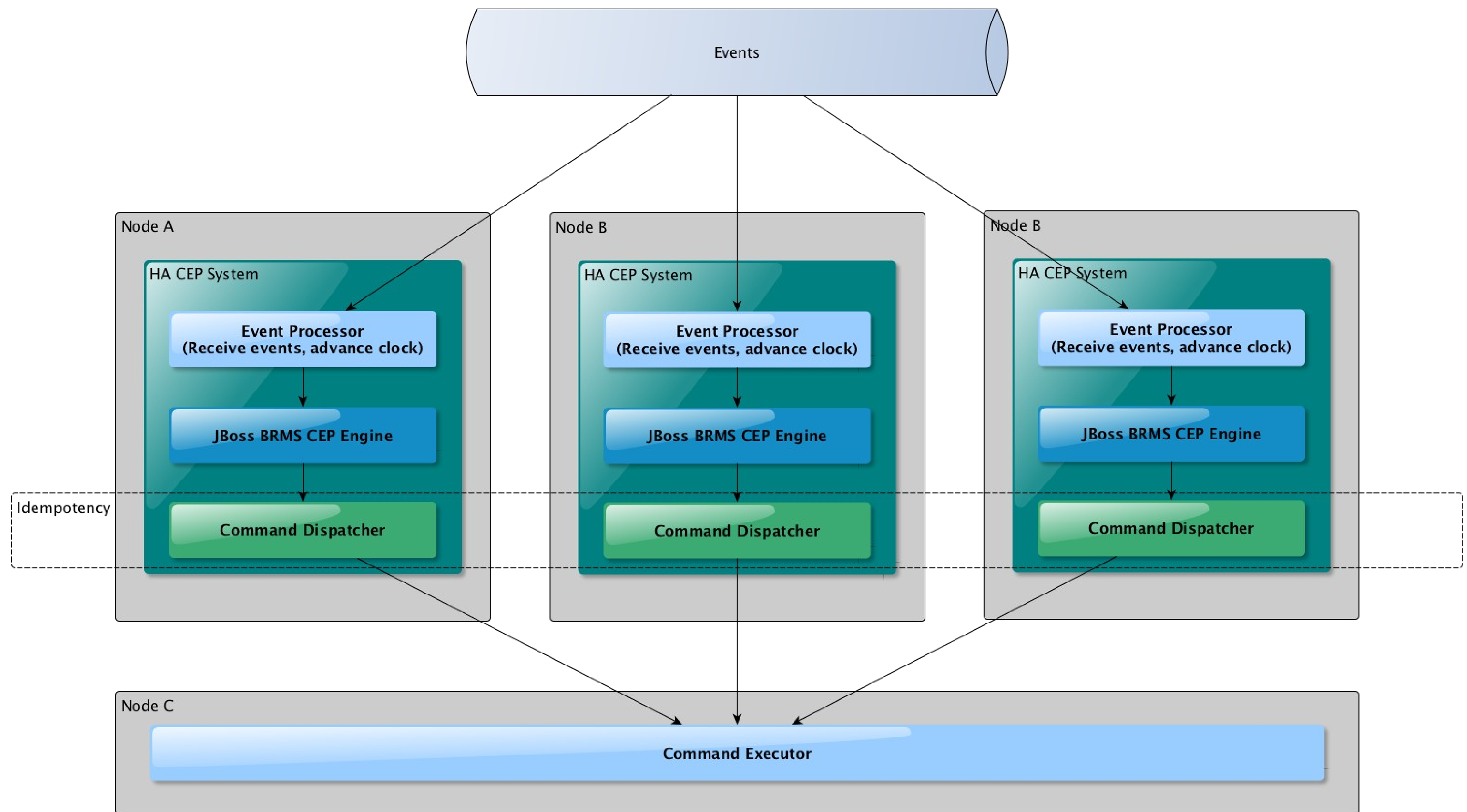
The Command Object ID can be used to detect duplicate commands between engines and can be calculated from the rule Match:

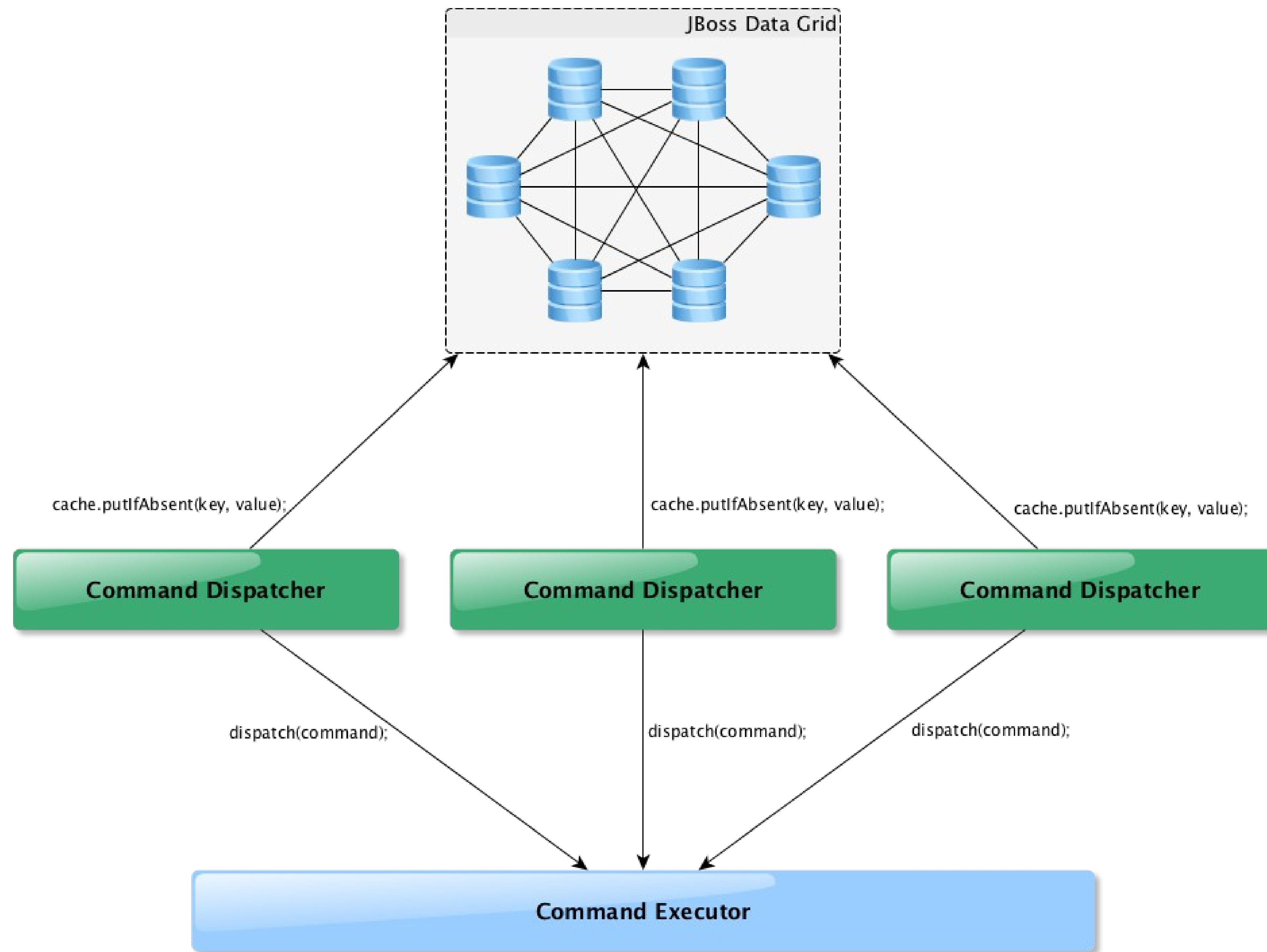
- Rule Package
- Rule Name
- Event UUIDs

This concept allows to both:

- Run 2 engines in parallel and executing the RHS only once by using duplicate detection (i.e. idempotency principal).
- Replaying events and filtering already fired RHSs.







JBoss BRMS HA CEP

Architectural Decisions recap

- All Events have a UUID and a timestamp.
- Direct, non-deterministic manipulation of the session is not allowed (e.g. insert/update/delete fact).
- Use the pseudo-clock and advance clock based on the timestamp of the incoming fact.
- CEP Engine actions are wrapped in Command objects. Engine is not allowed to directly interact with other systems.
- Command Objects have an ID that can be deterministically computed in the individual nodes.

JBoss BRMS HA CEP

Architectural Decisions recap

- Commands are not directly executed by the engines, instead, CommandDispatchers are responsible for sending the Commands to be executed to a CommandExecutor.
- CommandDispatchers will execute a Command 'once-and-only-once'.
- Commands have a unique id per engine. This ID can be deterministically computed.
- Idempotency principle is implemented by discarding commands that have already been executed.
 - This allows for both parallel engine processing and session restoring via replay architecture without executing actions twice.

Conclusion

- JBoss BRMS CEP adds powerful semantics to existing architectures.
- JBoss BRMS can be setup as a highly available system.
- There are multiple architectures possible to setup a HA CEP system. Which architecture to use highly depends on the system requirements (i.e. acceptable downtime, state-loss, etc.).
- JBoss BRMS HA CEP does not come out-of-the-box.

RED HAT
SUMMIT

10 YEARS *and counting*
SAN FRANCISCO | APRIL 14-17, 2014

Questions?