Distributed Computing in the Modern Data Center: Matching the Right Technology to Your Task

Ari Zilka
CTO and Founder
Terracotta

http://www.openterracotta.org

Session TS-88040

Goal

Examine some real-world Enterprise Java use cases

From those cases, **learn** how to lighten your application stack burden

Agenda

Define Lightweight

Four Use Cases I Saw This Year

The Gaps in the Stack That Added Weight

The Pattern: CAP and 2PC/Java™ 2 Platform, Enterprise Edition (J2EE™ Platform) Misuse

Heap-Level Replication

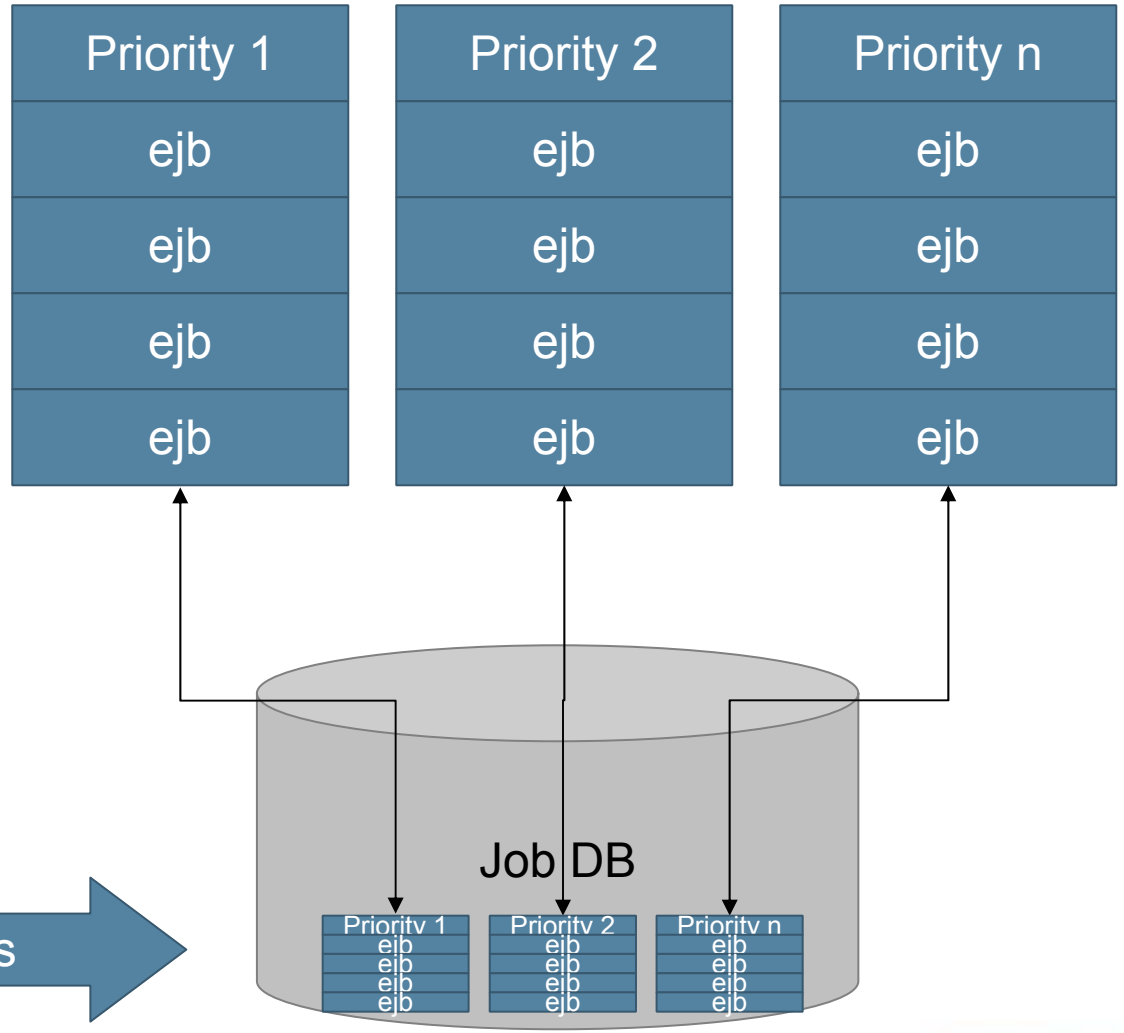
Agility Returns

Two Cases I Saw Where Lightweight Worked

Lightweight

- Starts with POJO
  - Object identity (no **implements serializable**)
  - **No Manager pattern (get() and putback())**
  - **Free domain modeling**
- Then, UWYN—When you need it
  - **Continuations, Cometd**, and **DWR, Spring**
  - Use the Container I choose
- **Leads to abstracting what I see fit without leaks**

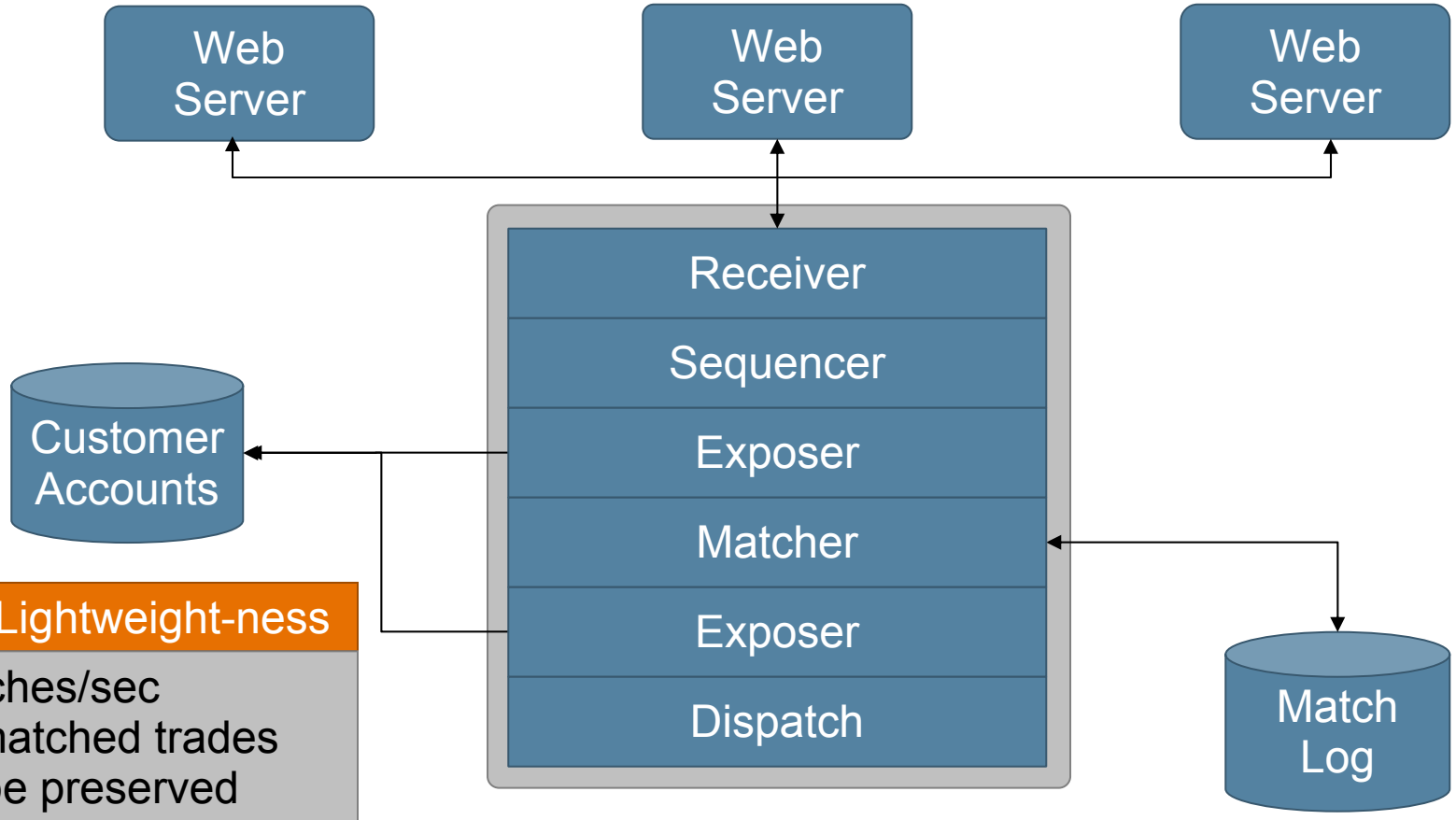
Case #1: Retailer— EJB™ Architecture Abuse



Issues With Lightweight-ness

1. Unit of Work was a queue, not a bean...ugh!
2. Cannot scale work
3. EJB architecture gave transactional queue update, but not what was needed
4. Problem: Durable queues

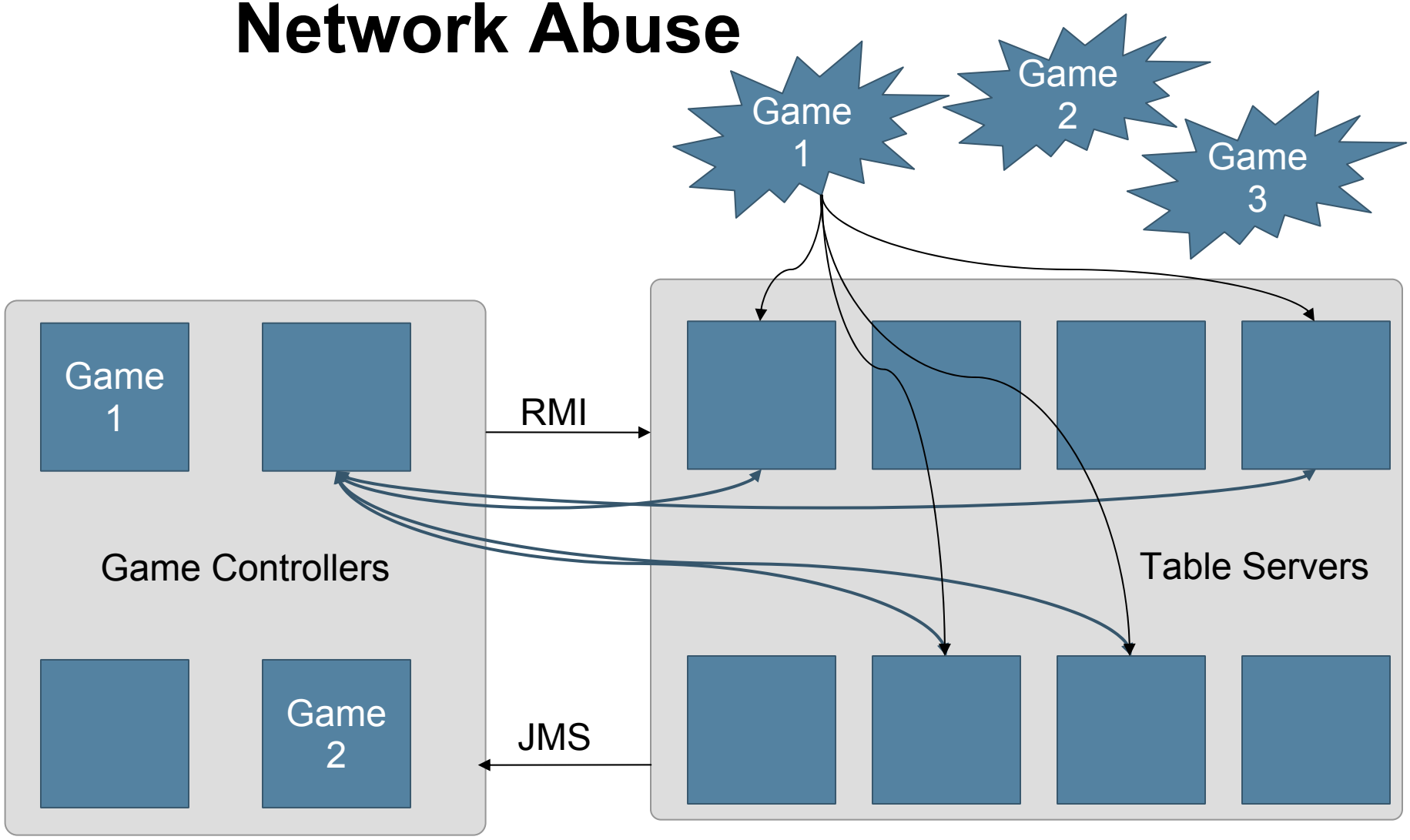
Case #2: Financial—Trading Pipeline



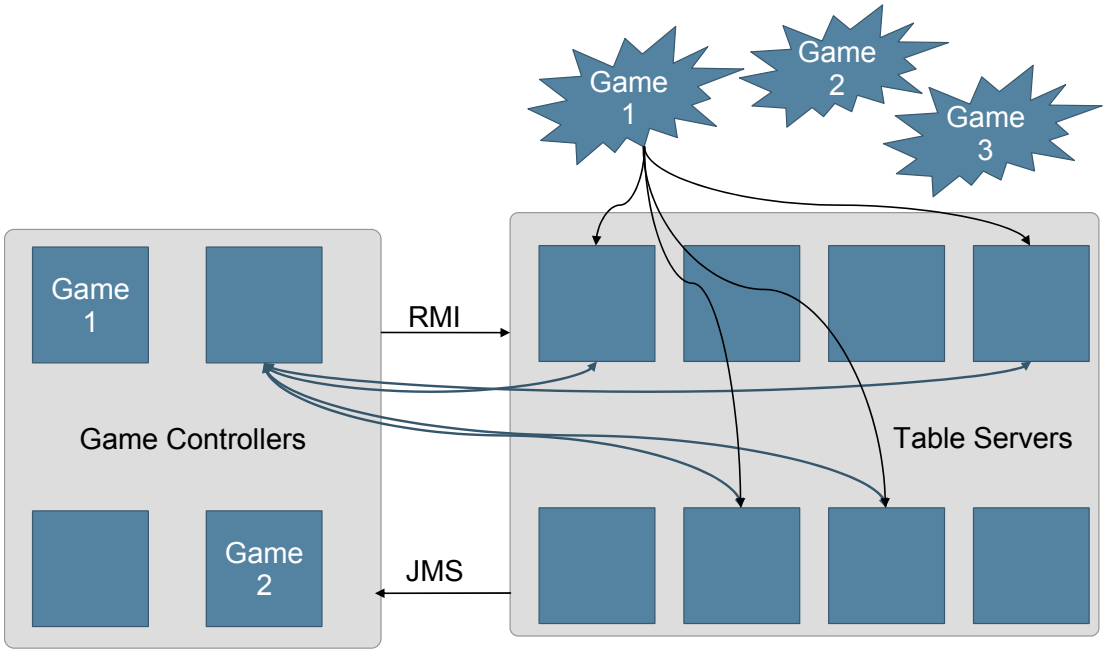
Issues with Lightweight-ness

1. 20K matches/sec
2. Only unmatched trades need to be preserved
3. DB bottleneck and code smell

Case #3: Online Gaming—Network Abuse



Case #3: Online Gaming—Network Abuse



Issues With Lightweightness

1. JMS API + RMI with serialization
2. Domain is gone in favor of infrastructure
3. Results 1 GBit insufficient

Gaps in the Application Stack

- Stack is OS, Virtual Machine for the Java platform (JVM™ machine), (optional) App Server, Frameworks, your code
- Case #1 (EJB Architecture Abuse)—Needs a seamless clustered and durable queue
- Case #2 (Trading Pipeline)—Needs a durable queue
- Case #3 (Network Abuse)—Needs to share data structure (this time not a queue but a map) across processes for socket scale out
- The gap is clustered POJO

The terms “Java Virtual Machine” and “JVM” mean a Virtual Machine for the Java™ platform.

The Pattern: Why All the Confusion

- CAP Theorem (Amazon.com) says you can have only two of the following:
  - Consistency
  - Availability
  - Partitionability

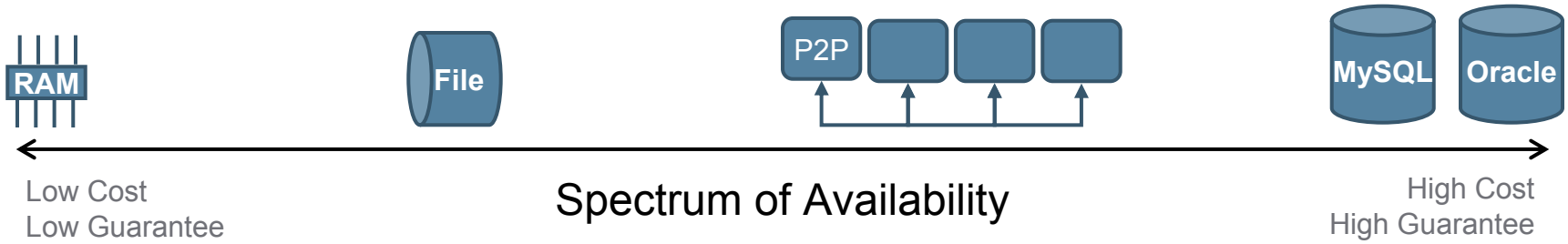
Application of CAP

Architecture Concern or Fear	Trade Off (Cap)	Result
Lost job	CA	No partionability, no scale
Restartability	CA	Bottleneck on DB
Connected end users	CP	Tables or controllers crash; lost game!

Let's Think About It

- **Consistency** means all nodes are the same
- **Availability** means no single point of failure
- **Partitionability** means nodes lose interconnectedness and do not care
  - So Consistent + Available::
    - Copy all data everywhere (consistent) but orphan any node that won't ACK (available)
    - ⇒ No partitioning support
  - Available + Partitionable::
    - Allow autonomous changes (available) with catch up and conflict resolution (partition)
    - ⇒ No consistency

What It All Means



Trade Off the Risk of the Loss of Data Versus the Cost of Storage

A Recipe for Managing CAP

- Make the trade-offs at a fine grained level
- Try to keep them out of code to freely move up and down as lessons are learned
- Fine-grained == RAM or Heap
- Out of code == POJO/annotations
- Monolithic app servers and architectures didn't help because the need is lower level

Heap-Level Replication

- Cluster what you need (for consistency across JVM machines)
- Store objects to disk (for availability across restarts)
- Partition problems organically (each JVM machine consumes only objects it needs)
- Central Traffic Cop that itself must scale—think Network Attached Memory

Agility Returns

- Punt on the clustering work until you cluster
  - Cluster without a wholesale rewrite
  - Cluster the app with approximately the same architecture as you started
- Avoid getting married to one paradigm so that you can switch
- Balance the trade-offs by keeping assumptions out of code where possible; C+A+P = happy line of business owner
- Yes, you are scaling, but first, you are delivering a business app, not a scalability architecture, no?

Positive Case #1

- A European Credit Card Reconciliation
  - SEDA on one server
  - Unit testing complete
  - C+A were most important but at high scale
  - Scale out—Java Message Service (JMS) API scalability failing with durable queues (price of C too high)
  - “C” brought in check with clustered POJO queue (no 2PC, no RDBMS)

Positive Case #2

- Japanese Bank's Trading Engine
  - Single Server design
  - Operational for 2+ years
  - C+A were most important but at high scale
  - Scale out—JavaSpaces™ technology required rewrite, too risky
  - Again, "C" delivered scaled out

Q&A

Ari Zilka
Terracotta

http//www.terracotta.org

Distributed Computing in the Modern Data Center: Matching the Right Technology to Your Task

Ari Zilka
CTO and Founder
Terracotta

http://www.openterracotta.org

Session TS-88040