J-Fall

11 oktober 2007 Spant! - Bussum

Lightweight Grids with Terracotta

Cesario Ramos

Xebia

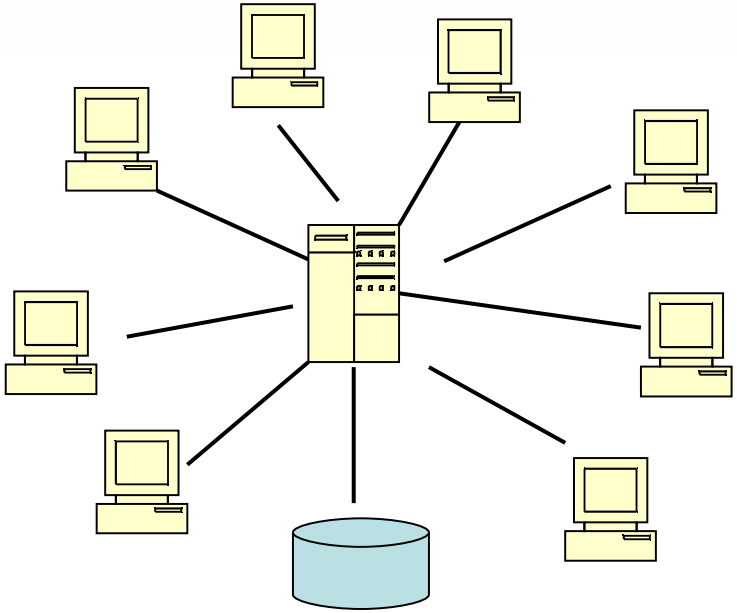
Overview

- **Enterprise application challenges.**
- How can a grid approach help?
- Terracotta Overview
- Terracotta Use Cases.
- A simple grid example using Terracotta.
- Q&A.

Application challenges

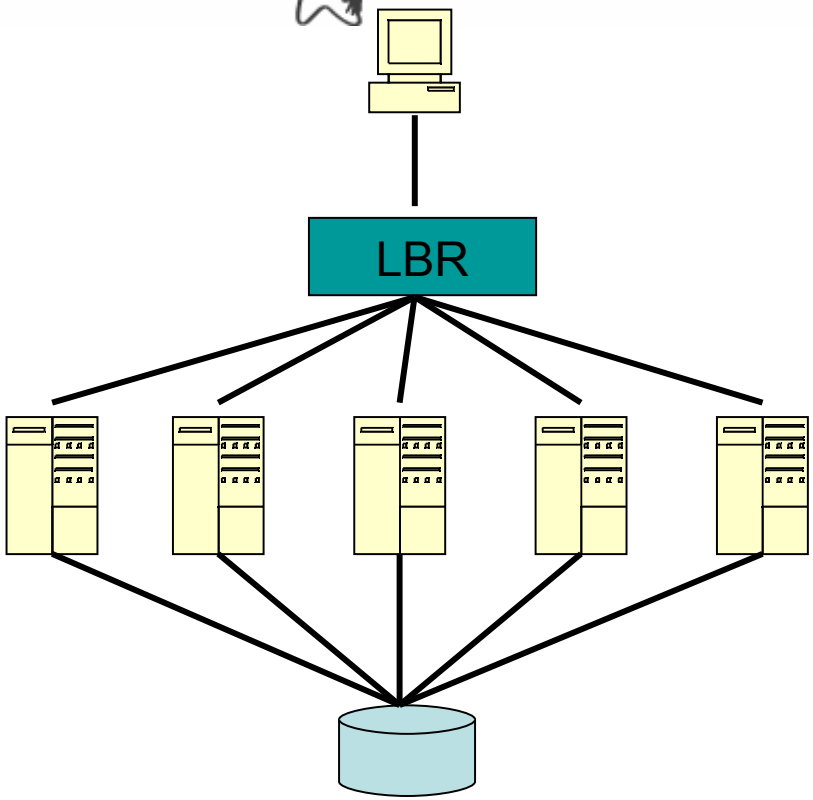
- Availability
  - Measured as the probability of being down.
  - How do you keep your systems up?
    - Important to avoid a Single Point of Failure.
- Scalability
  - Increase in computing power proportionally to the capacity added.
  - How do you make your system scalable?
    - Important to avoid Single Point of Bottleneck.

The client/server architecture



- Single Point Of Failure.
- Single Point Of Bottleneck.

Clustering your application



- Cluster the servers.
  - Use Serialization, RMI, JMS, ...
- Keep session state small, max 10kb.
- Store state in db.
  - Transient state.

- Unload responsibility to the database
  - Complicated and expensive to cluster.

High Availability

- Take into account that:
  - Redundancy in machinery means that:
    - 1 node: p(down) = .01 (99%)
    - 2 node: p(down) = .01 * .01 = .0001 (99.99%)
  - Adding tiers can decrease availability:
    - Three tiers with 99% uptime each gives .99 * .99 * .99 = 97% uptime.
  - Often better to have more ‘small’ then few ‘big’ machines.
    - Eight 1-cpu machines often better then two 4-cpu machines.
    - Node failure increases load on others.

High Scalability

- Focus on:
  - Eliminate Single Points of Bottleneck.
    - E.g. services that requests have to go to through.
  - Avoid concurrency control bottlenecks.
    - Isolation levels, application lock contention, …
  - Favor bottleneck in the application tier.
    - CPU, memory bound.
  - Use caching for
    - Fast data access
    - Unloading of expensive tiers

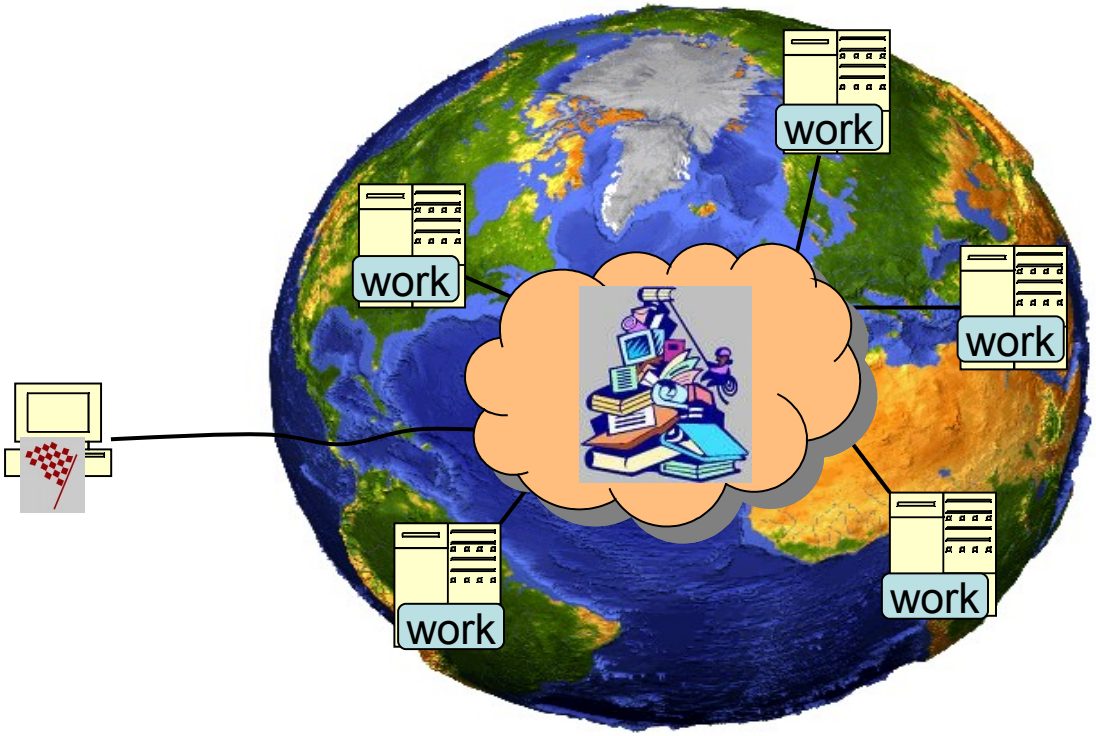
Summary

- Try to avoid SPOF and SPOB.
- Don’t misuse you DB for transient data.
  - Eventually it will become a SPOB
- Often better to have more ‘small’ then few ‘big’ machines.
- Use caching to unload expensive tiers.

Overview

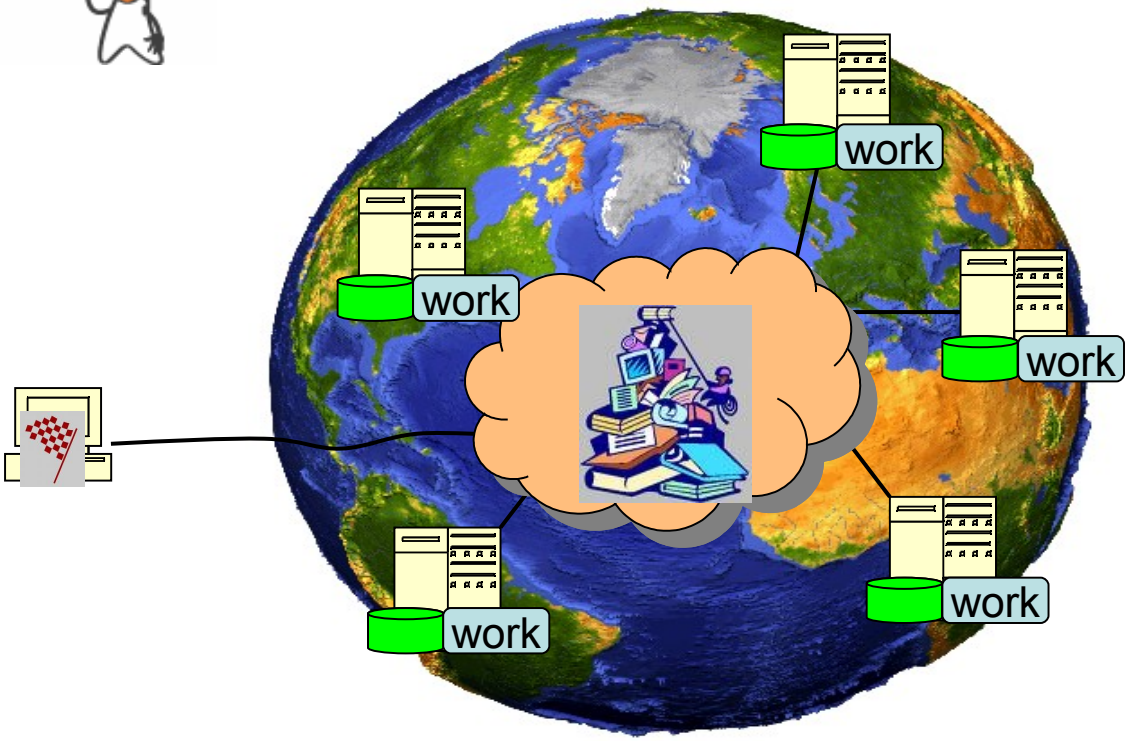
- Enterprise application challenges.
- **How can a grid approach help?**
- Terracotta Overview
- Terracotta Use Cases.
- A simple grid example using Terracotta.
- Q&A.

What is a computational grid?



Increased computing power is only as good as the data infrastructure that supports it

What is a data grid?



- Locality of reference.

What do grids offer?

- Availability and scalability.
- Dynamic allocation of resources.
- Transparent distribution of
  - logic, data or events.
- Transparent access of data.
- Persistence.
- Transactions.

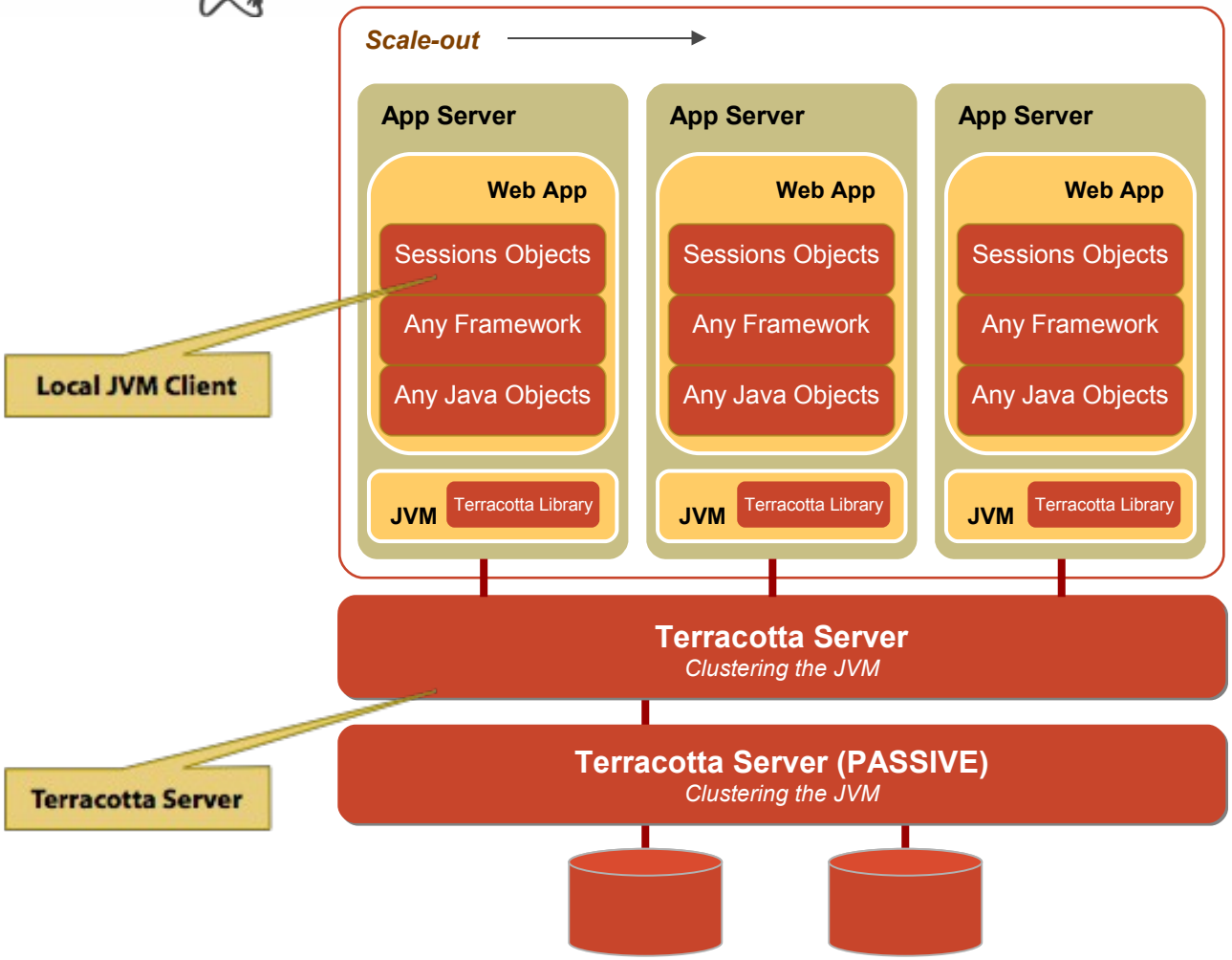
Overview

- Enterprise application challenges.
- How can a grid approach help?
- **Terracotta Overview.**
- Terracotta Use Cases.
- A grid example using Terracotta.
- Q&A.

What is Terracotta?

- Solution for clustering Java at the JVM level.
- Provides extended local JVM heap.
  - external heap is realized by a TC server.
  - acts like paging/swapping in your OS.
- Advantages.
  - Availability and scalability at the JVM level.
  - keeps JVM semantics across the cluster.
  - reduces complexity of distributed computing.
  - provides a basis for realizing grid solutions.

Terracotta Architecture



How to use the extended heap?

- Distributed Shared Object.
  - Terracotta instrumented object.
  - Cluster wide unique object.
- Shared root
    - starting point of a distributed object graph
    - class data member that is transparently mirrored
    - are persisted across JVM lifecycle
- Terracotta transaction
  - MONITOR ENTRY and MONITOR EXIT
  - Changed data is synchronized using synchronous commit.
- Locks
  - Cluster wide synchronized, wait/notify
  - Synchronous-Write, Write, Read, Concurrent
  - Named, Auto

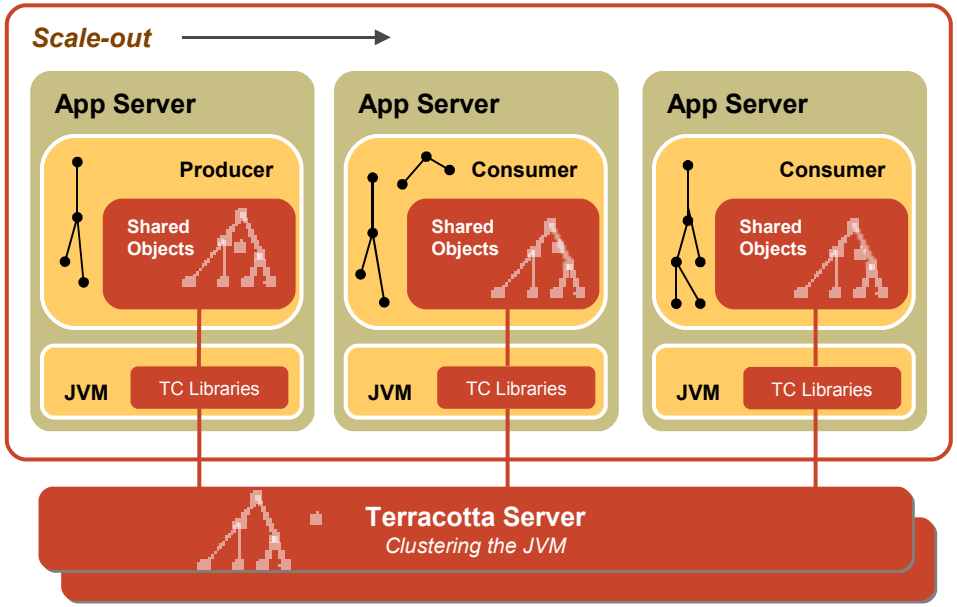
Simple Example

```
public class Consumer {
private LinkedBlockingQueue<Something>
    queue;
public void consume() {
    Something msg = null;
    while(true) {
    msg = queue.take()
…
public class Producer {
private LinkedBlockingQueue<Something>
    queue;
public void produce() {
    Something msg = null;
    while(true) {
    msg = queue.put();
…
```

tc-config.xml

```
…
<instrumented-classes>
 <include>
  <class-expression>
    example.Something
  </class-expression>
...
<root><field-name>
      example.Consumer.queue
 </field-name>
 <root-name>rootQueue</root-name>
</root>
<root>
 <field-name>Producer.queue</field-name>
 <root-name>rootQueue</root-name>
</root>
…
```

Virtual Heap



What does Terracotta offer?

- Availability and Scalability
  - No SPOF, SPOB.
  - Fine grained changes.
  - Automatic partitioning.
  - Redundant TC servers.
  - Copy only where resident.
- Dynamic allocation of resources.
- Transparent distribution and partitioning of
  - logic, data or events.
- Transparent access of data.
- Persistence.
  - Data written to TC is persistent until explicitly removed.
- Transactions.
  - Java memory model semantics
  - Lock acquire and lock release for flushing data to TC.

Summary

Terracotta does not offer you a grid solution, it just offers you a easy way to do some griddy stuff easily.

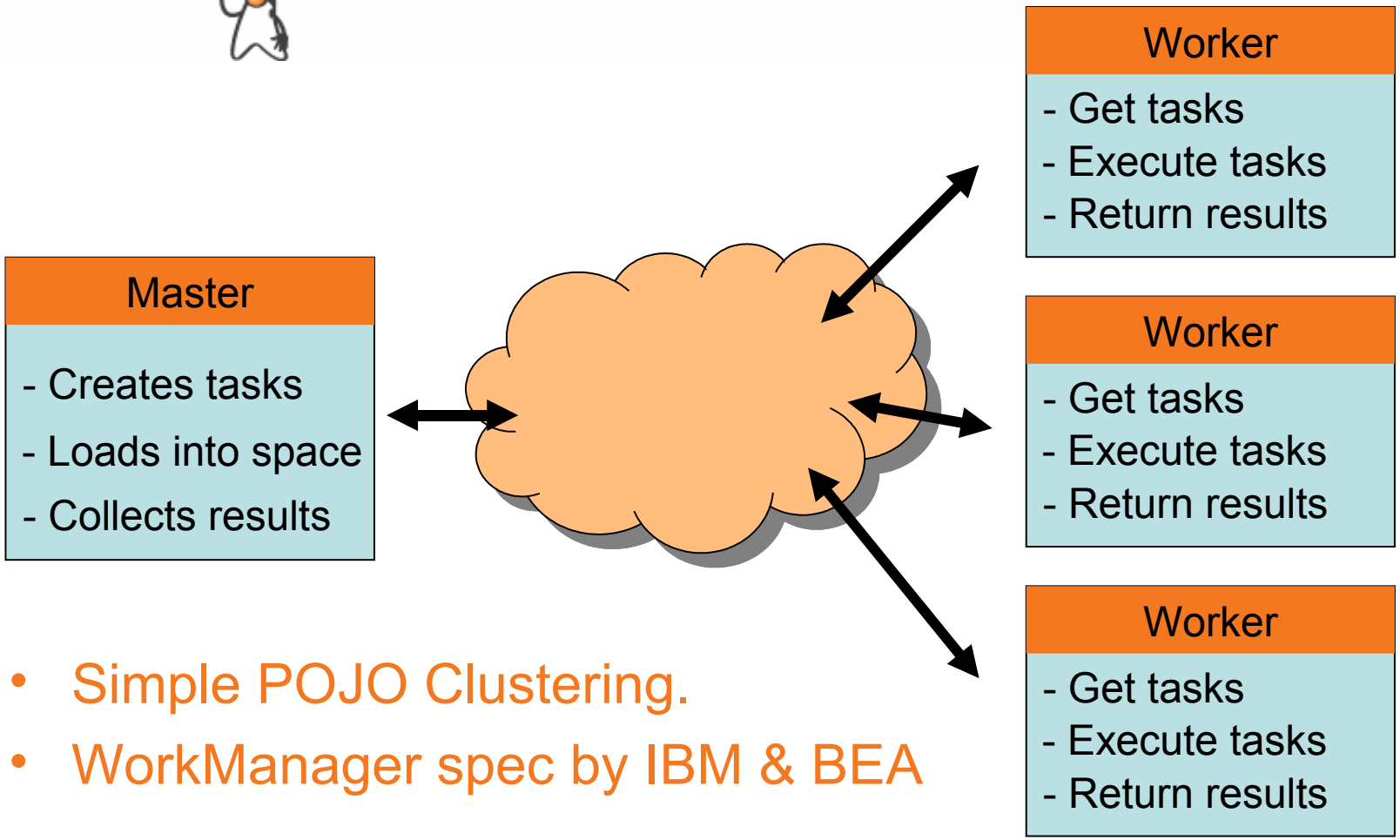
Overview

- Enterprise application challenges.
- How can a grid approach help?
- Terracotta Overview.
- **Terracotta Use Cases.**
- A simple grid example using Terracotta.
- Q&A.

Terracotta Sessions

- Clusters the HttpSession data
  - Keeps a shared root holding SharedData objects for all HttpSessions.
  - Gives high throughput even with a large session.
  - No serialization.
- Transparent
  - No setArrtibute(..) needed.
  - You need to declare your instrumented classes!
- Supported platforms
  - Jetty
  - Tomcat 5.0 & 5.5
  - Websphere 6.1, CE
  - JBoss 4
  - WebLogig 8.1, 9.2

Grids



- Simple POJO Clustering.
- WorkManager spec by IBM & BEA

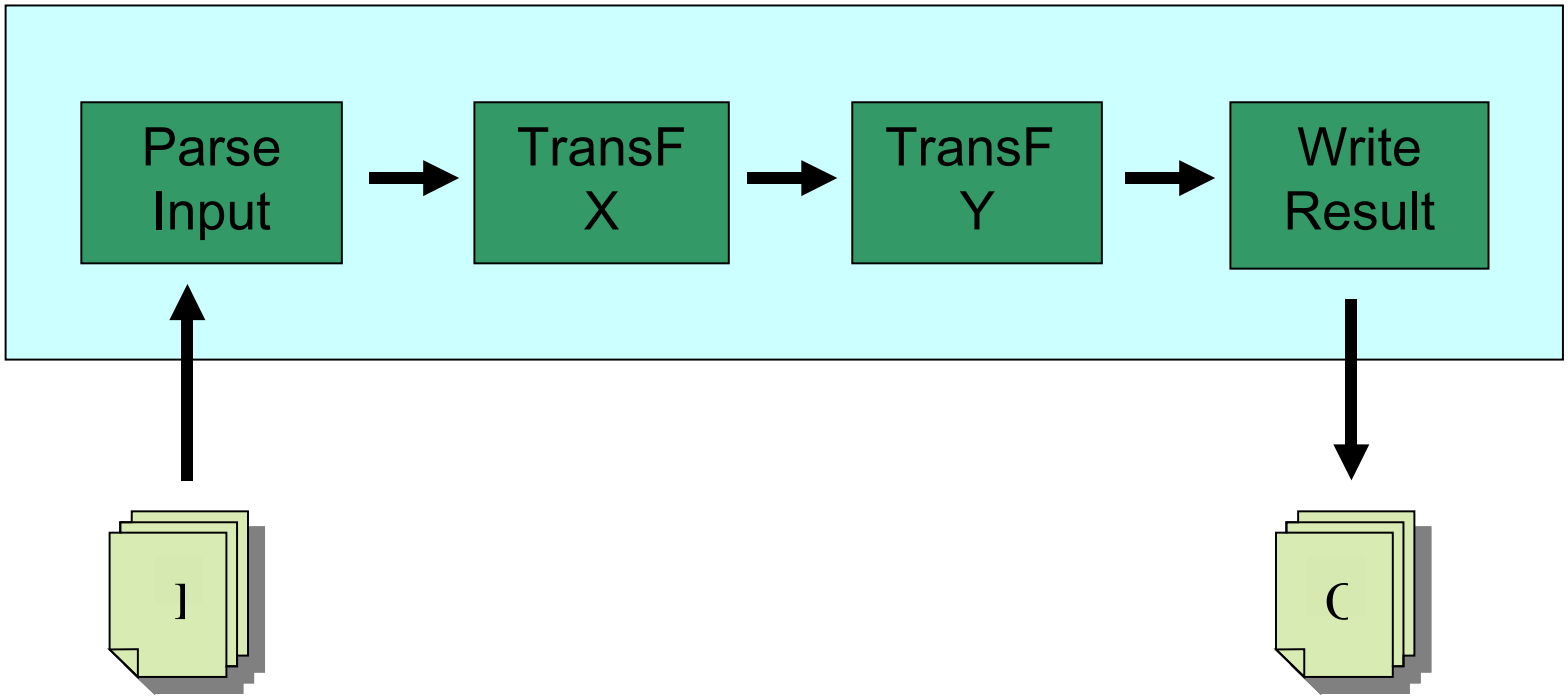
Distributed Caching

- Stand alone caches.
  - EHCache, OSCache
- Hibernate L2 cache (e.g. EHCache)
  - Reduces db load by eliminating intermediate loads.
  - Still need intermediate commits for updates.
  - Detach POJO's
    - Eliminate intermediate commits.
    - Need worry about availability, scalability and graph size.
  - Cluster it with Terracotta
    - Detached POJO's are clustered.
    - Reattach Session in any JVM.
    - Terracotta scaling and availability.
    - Can use large graphs.

Overview

- Enterprise application challenges.
- How can a grid approach help?
- Terracotta Overview.
- Terracotta UseCases.
- **A simple grid example using Terracotta.**
- Q&A.

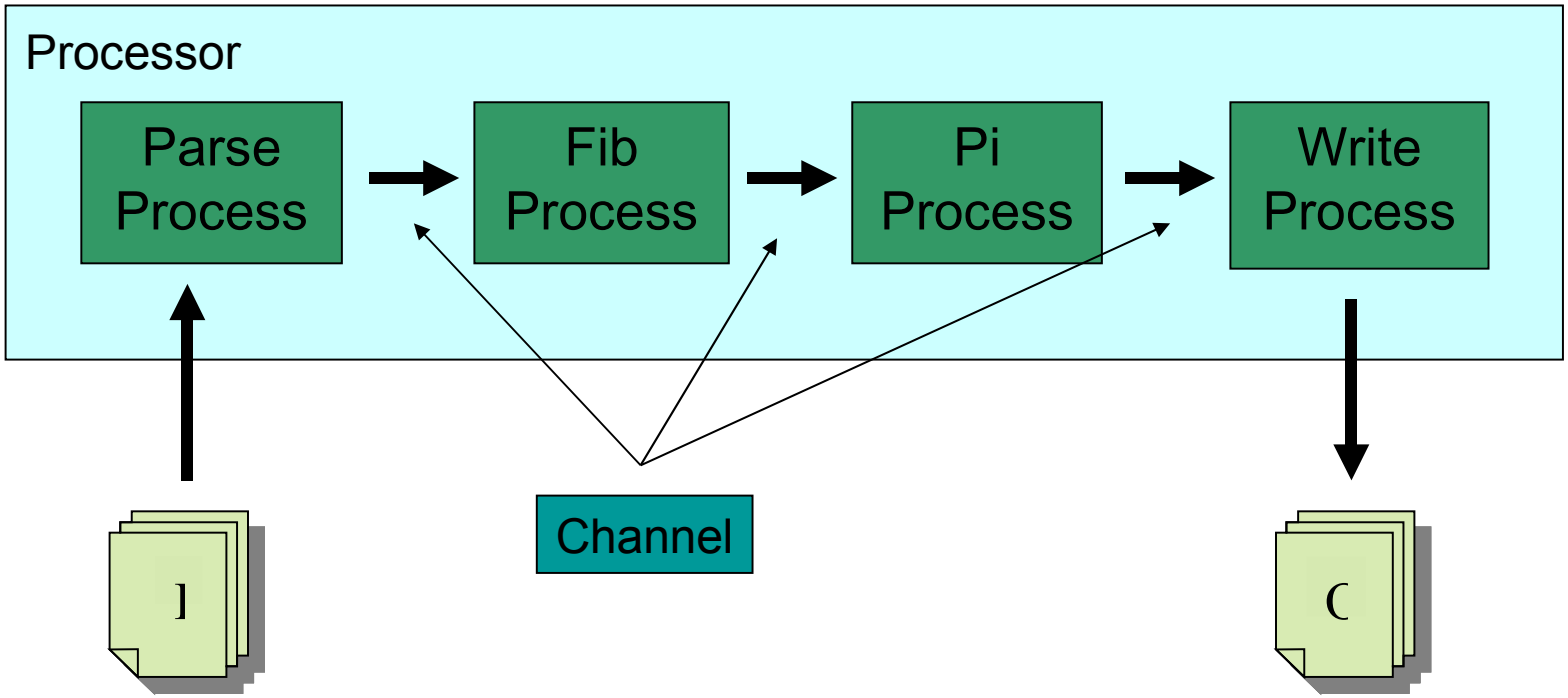
Use Case



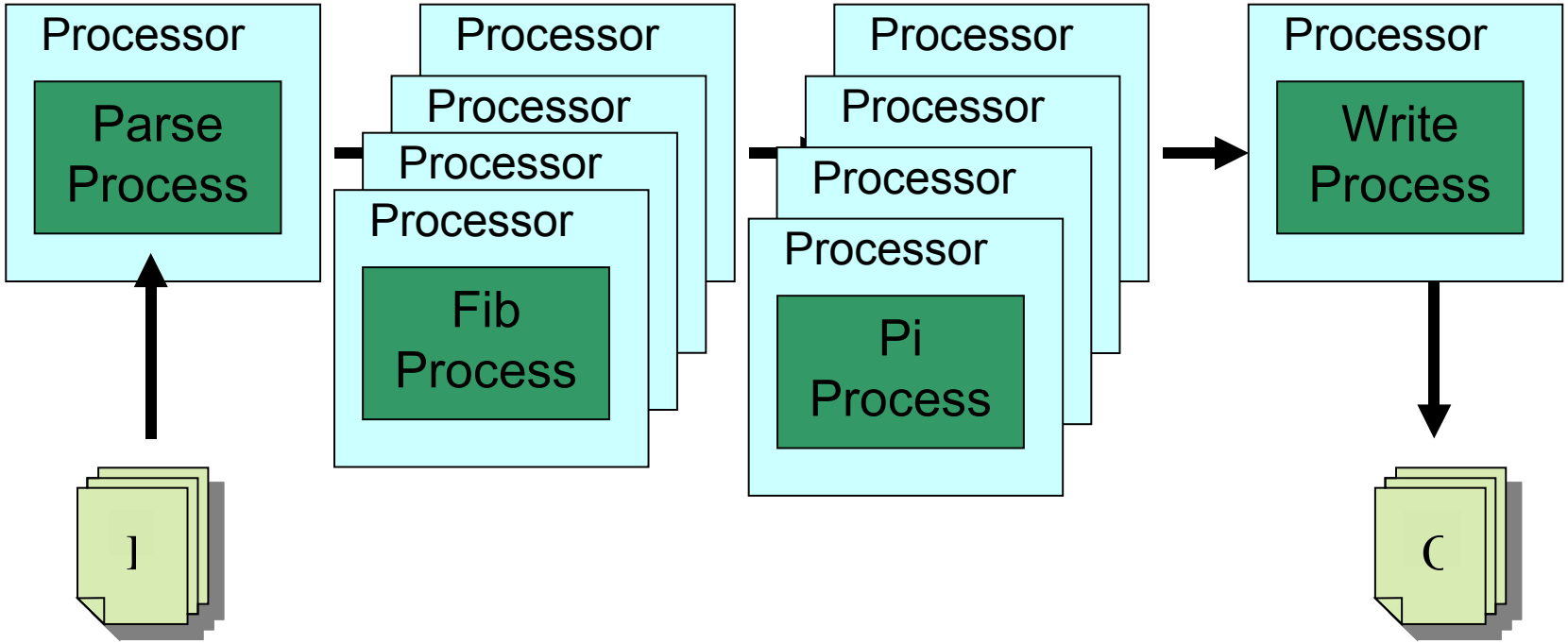
Prometheus

- Library that extends java.util.concurrent.
- Makes it easy to separate business logic from concurrent infrastructure logic.
- Offers coarse grained building blocks i.e.
  - Processor
  - Process
  - Repeater
  - Channel

Fits nicely in Prometheus



Distributed parallelized



Fibonacci Process

```
public class FibonacciProcess {
    …
    public void receive(Task task) {
        task.setOutFibonacci(fibonacci(task.getInFibonacci()));
        log.info(task);
    }
    public static long fibonacci(long n) {
        if (n <= 1)
            return n;
        else
            return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Spring Fibonacci Process config

```
<bean id="fibonacciProcess"
        class="com.xebia.clustering.processes.FibonacciProcess"/>
   <bean id="fibonacciProcessor"

    class="org.codehaus.prometheus.processors.standardprocess
    or.StandardProcessor"
        parent="abstractProcessor">
      <constructor-arg index="0"
                    ref="fibonacciProcessor-input"/>
      <constructor-arg index="1"
                    ref="fibonacciProcess"/>
      <constructor-arg index="2"
                    ref="fibonacciProcessor-output"/>
</bean>
```

Spring Pipeline config

```
<bean id="pipe1"
        name="piProcessor-input,parserProcessor-output"
        class="org.codehaus.prometheus.channels.BufferedChannel">
      <constructor-arg index="0" value="100"/>
    </bean>
    <bean id="pipe2"
        name="piProcessor-output,fibonacciProcessor-input"
        class="org.codehaus.prometheus.channels.BufferedChannel">
      <constructor-arg index="0" value="100"/>
    </bean>
    <bean id="pipe3"
        name="fibonacciProcessor-output,fileWritingProcessor-input"
        class="org.codehaus.prometheus.channels.BufferedChannel">
      <constructor-arg index="0" value="100"/>
    </bean>
```

tc-config

```
…
        <application-contexts>
            <application-context>
                <paths>
                    <path>applicationcontext-channels.xml</path>
                </paths>
                <beans>
                    <bean name="pipe1"/>
                    <bean name="pipe2"/>
                    <bean name="pipe3"/>
                </beans>
            </application-context>
        </application-contexts>
…
```

Wrapping up

- Terracotta turns scalability and high-availability into a **deployment artifact**
- Keep the simplicity of POJO-based development – get **Scale-Out with Simplicity**
- Makes mission-critical applications **simpler to**:
  - Write
  - Understand
  - Maintain
  - Test
- Endless possibilities for clustering and distributed programming - these were just a few
- Be creative, use your imagination and have fun…
- But beware, TC introduces subtle changes:
  - Semantics: tc-roots
  - API's and behaviour of modules like EHCache

Final Note

- Want to hear more?
  - podcast.xebia.com
    - Interview with Jonas Boner from Terracotta
- Want to learn more about performance?
  - Xebia course 6-9 november:
    - Speeding up Java applications of Kirk Pepperdine
- Prometheus
  - http://prometheus.codehaus.org/

J-Fall

11 oktober 2007 Spant! - Bussum

Lightweight Grids with Terracotta

Cesario Ramos

Xebia