Best Practices for Scaling Java Applications with Distributed Caching

Copyright © 2011 Project Cacheonix

Introduction

Presenter: Slava Imeshev

- Main committer for Cacheonix, open source distributed Java cache
- Core expertise in reliable distributed systems

– simeshev@cacheonix.org

– www.cacheonix.org/blog/

Copyright © 2011 Project Cacheonix

БАЙКОНУР

Definitions

Copyright © 2011 Project Cacheonix

Performance

Number of operations per unit of time

- Requests per second
- Pages per second
- Transactions per second

Performance is not scalability (is 200 pages/s more scalable than 150 pages/s?)

Copyright © 2011 Project Cacheonix

Scalability

Ability to handle additional load by adding more computational resources

- Vertical scalability
- Horizontal scalability

Copyright © 2011 Project Cacheonix

Vertical Scalability

- Vertical scalability is handling additional load by adding more power to a single machine
- Vertical scalability is trivial to achieve. Just switch to a faster CPU, add more RAM or replace an HDD with an SSD
- Vertical scalability has a hard limit (2-5 times improvement in capacity)

Copyright © 2011 Project Cacheonix

Horizontal Scalability

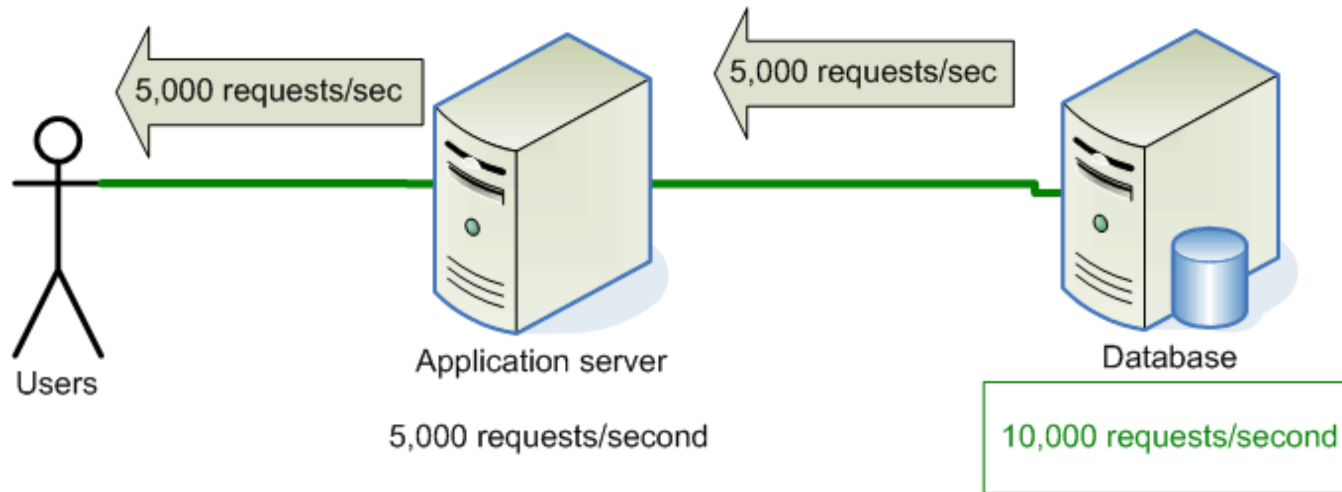
- Horizontal scalability is handling additional load by adding more servers
- Horizontal scalability offers much greater benefit (2-1000 times improvement in capacity)
- Horizontal scalability is much harder to achieve as adding servers requires ensuring data consistency and coherent view of cache updates

Copyright © 2011 Project Cacheonix

Scalability Problem

Copyright © 2011 Project Cacheonix

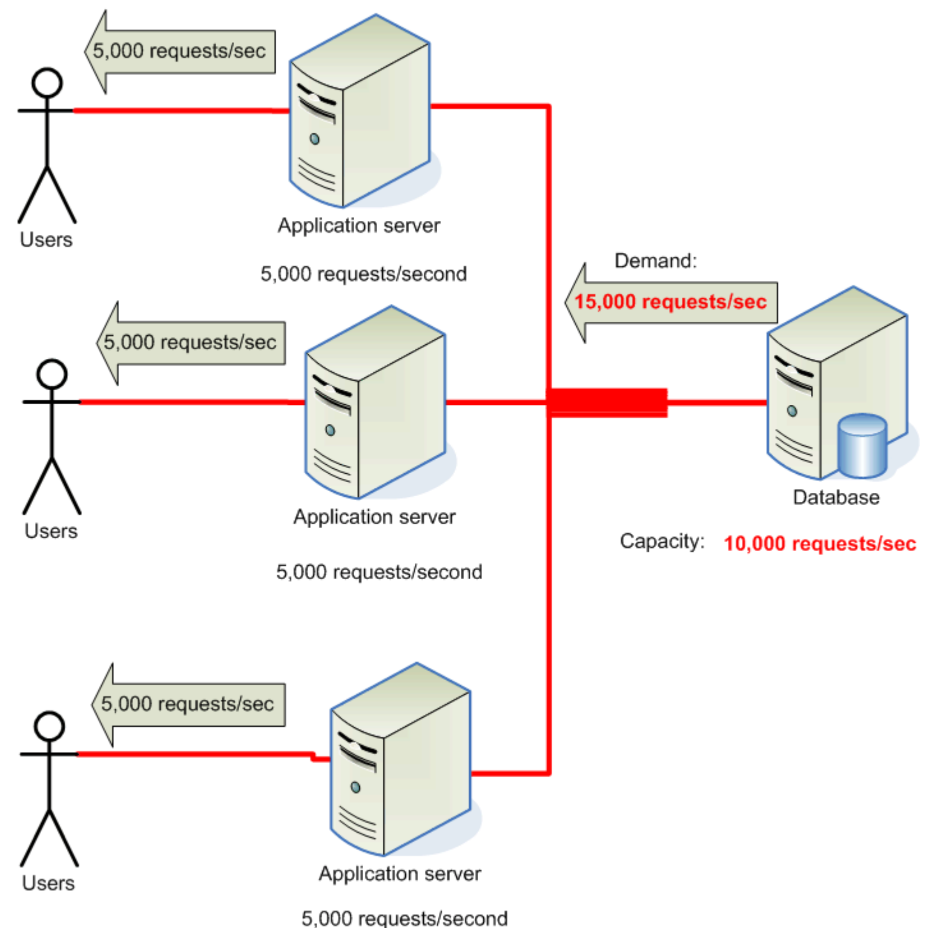
Normal Situation



Copyright © 2011 Project Cacheonix

System Cannot Scale

- Added 2 more app servers
- Expected x3 increase in capacity
- Got only x2
- System hit scalability limit
- Database capacity is a bottleneck



Copyright © 2011 Project Cacheonix

Cache

An area of local memory that holds a copy of frequently accessed data that is otherwise expensive to get or compute

```java
/**
 * A cache.
 */
public interface Cache {

  /**
   * Returns a cached value associated with a key.
   */
  Object get(final Object key);

  /**
   * Puts a value associated with a key into the cache.
   *
   * @return a previous value.
   */
  Object put(final Object key, final Object value);
}
```

Copyright © 2011 Project Cacheonix

Key Cache Parameters

- Cache size defines how many elements a cache can hold

Copyright © 2011 Project Cacheonix

Key Cache Parameters

- Cache size defines how many elements a cache can hold
- Cache eviction algorithm defines what to do when the number of elements in cache exceeds the size

Copyright © 2011 Project Cacheonix

Key Cache Parameters

- Cache size defines how many elements a cache can hold
- Cache eviction algorithm defines what to do when the number of elements in cache exceeds the size
- Time-to-live defines time after that a cache key should be remove from the cache (expired)

Copyright © 2011 Project Cacheonix

Cache Eviction Algorithm

Least Recently Used (LRU) works best

- Catches temporal and spatial locality

Copyright © 2011 Project Cacheonix

Cache Eviction Algorithm

Least Recently Used (LRU) works best

- Catches temporal and spatial locality

Most of other cache algorithms (MRU, LFU, etc)

- Not applicable to most of practical situations
- Subject of cache poisoning
- Expensive from performance point of view

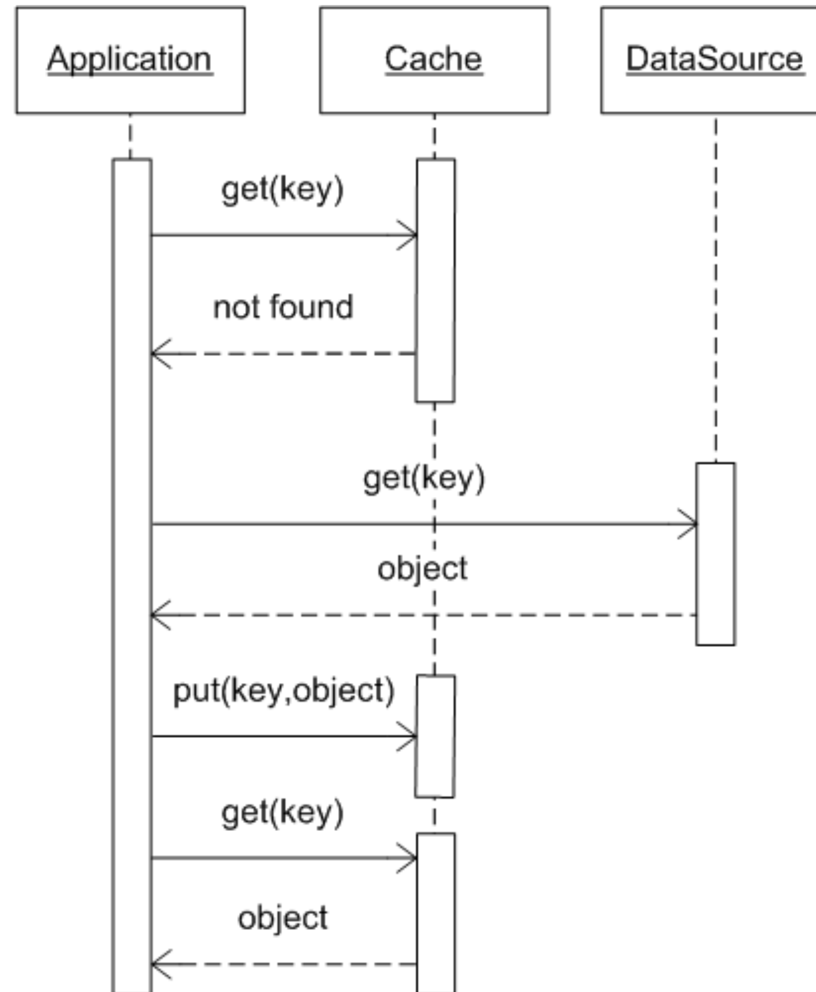
Copyright © 2011 Project Cacheonix

Cache Types

- Application cache
- Second level (L2) cache
- Hybrid cache

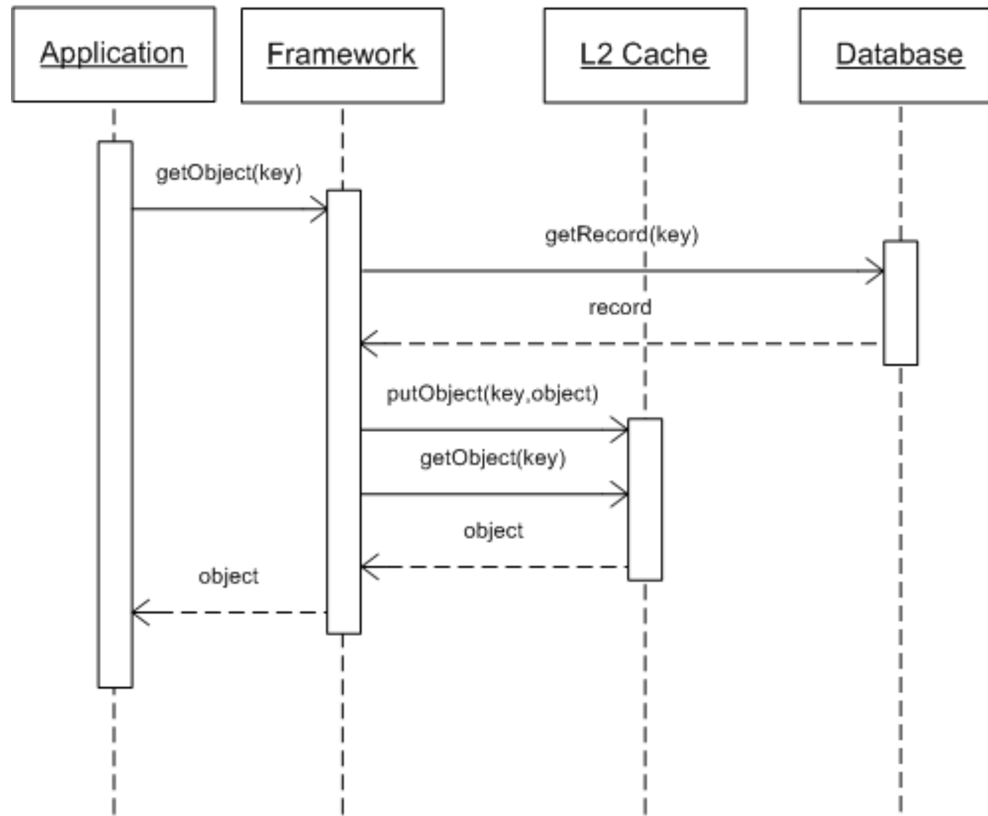
Copyright © 2011 Project Cacheonix

Application Cache



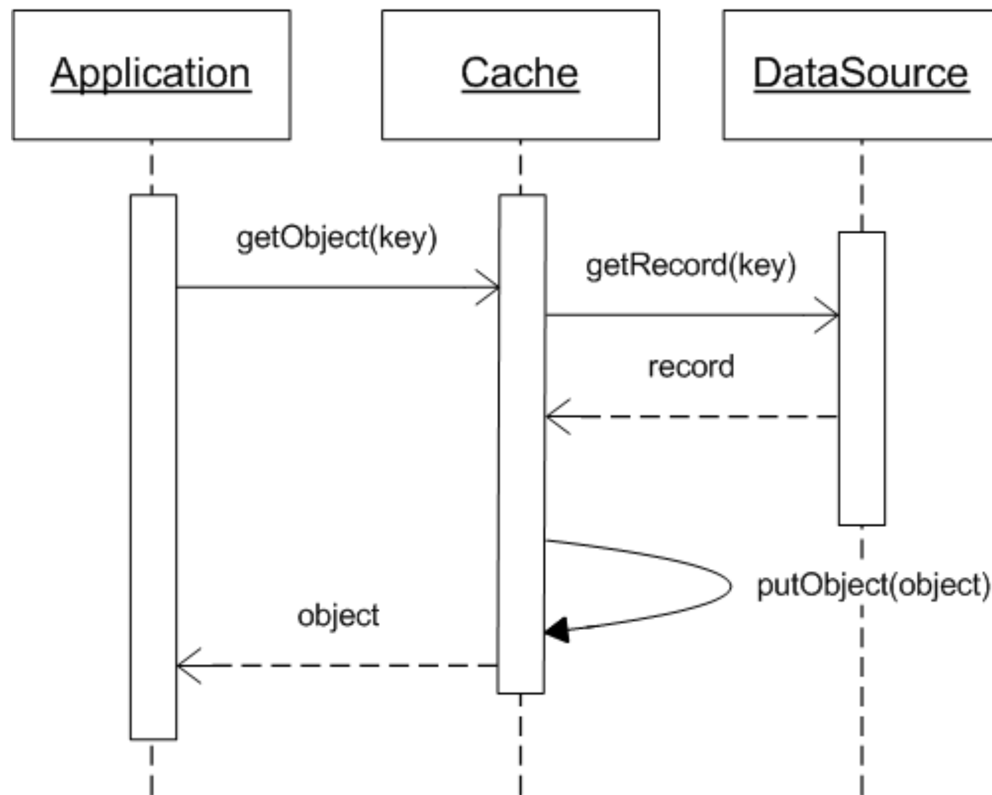
Copyright © 2011 Project Cacheonix

Level-2 Cache



Copyright © 2011 Project Cacheonix

Hybrid Cache



Copyright © 2011 Project Cacheonix

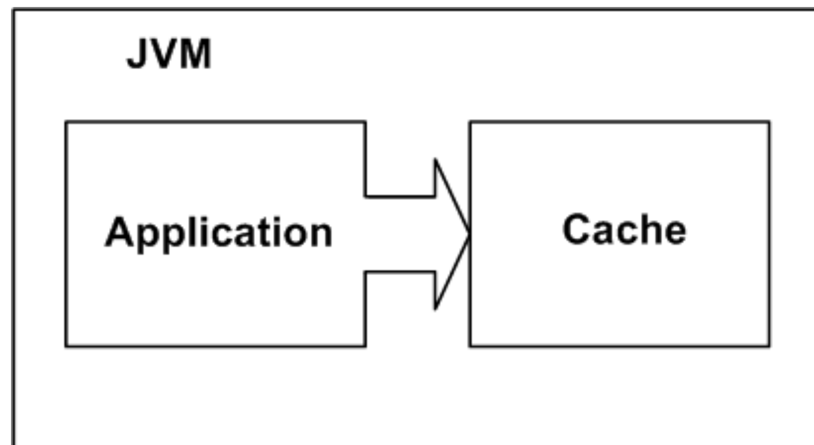
Cache Architectures

- Local
- Distributed

Copyright © 2011 Project Cacheonix

Local Cache

- All elements are stored in local memory
- Size is limited by a single JVM's heap



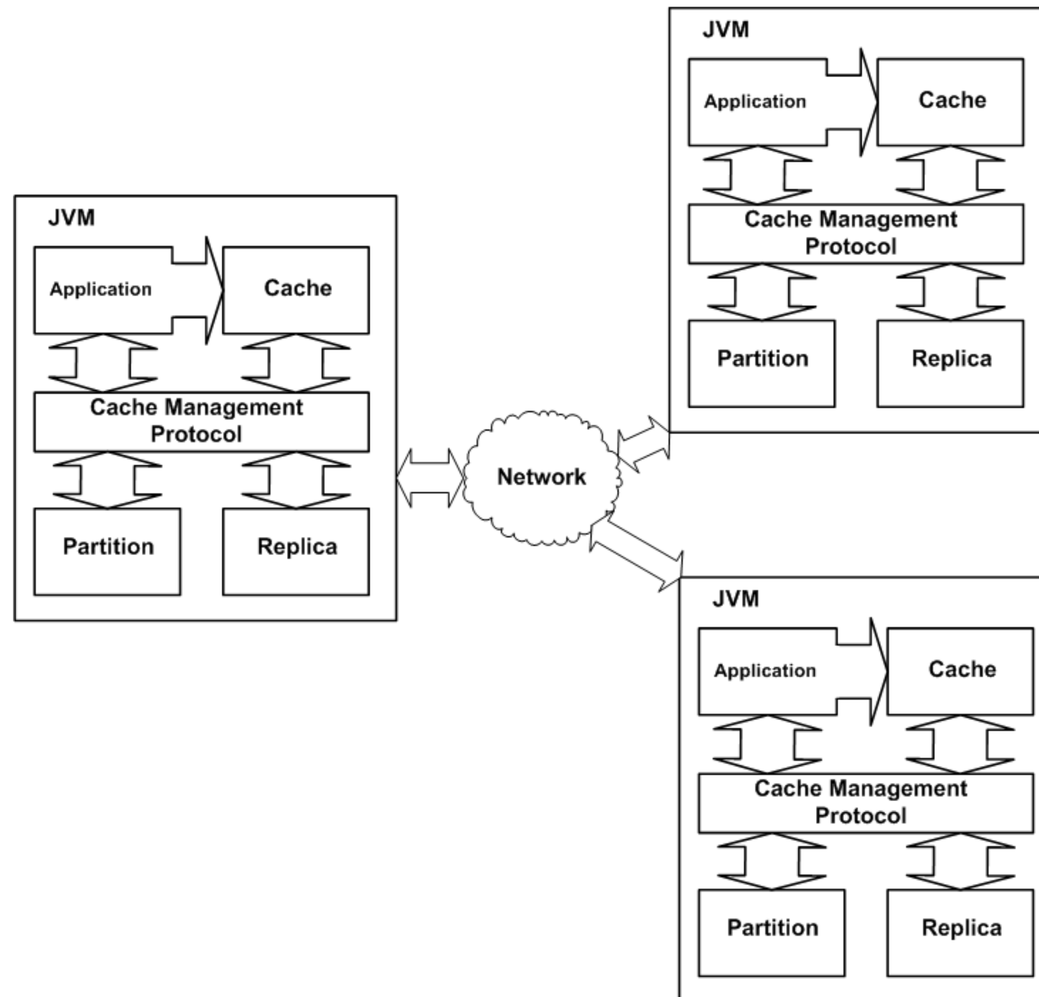
Copyright © 2011 Project Cacheonix

Distributed Cache

- Cache elements are distributed across a set servers (a cluster)
- Cache size is a sum of cache partitions in case of a partitioned cache
- Cache size can be much bigger than a single Java VM
- Distributed cache can scale horizontally by adding more servers

Copyright © 2011 Project Cacheonix

Distributed Cache Example (Cacheonix)



Copyright © 2011 Project Cacheonix

Distributed Cache

Important characteristics:

- Partitioning for load balancing

Copyright © 2011 Project Cacheonix

Distributed Cache

Important characteristics:

- Partitioning for load balancing
- Replication for high availability

Copyright © 2011 Project Cacheonix

Distributed Cache

Important characteristics:

- Partitioning for load balancing
- Replication for high availability
- Cache coherence for data consistency

Copyright © 2011 Project Cacheonix

Distributed Cache

Important capabilities:

- Partitioning for load balancing
- Replication for high availability
- Cache coherence for data consistency
- Fault tolerance for high availability

Not all systems have these capabilities

Copyright © 2011 Project Cacheonix

Availability and Fault Tolerance

Ability to continue to operate despite of failure of members of the cluster

- When applied to distributed caching, HA means an ability to provide uninterrupted, consistent data access

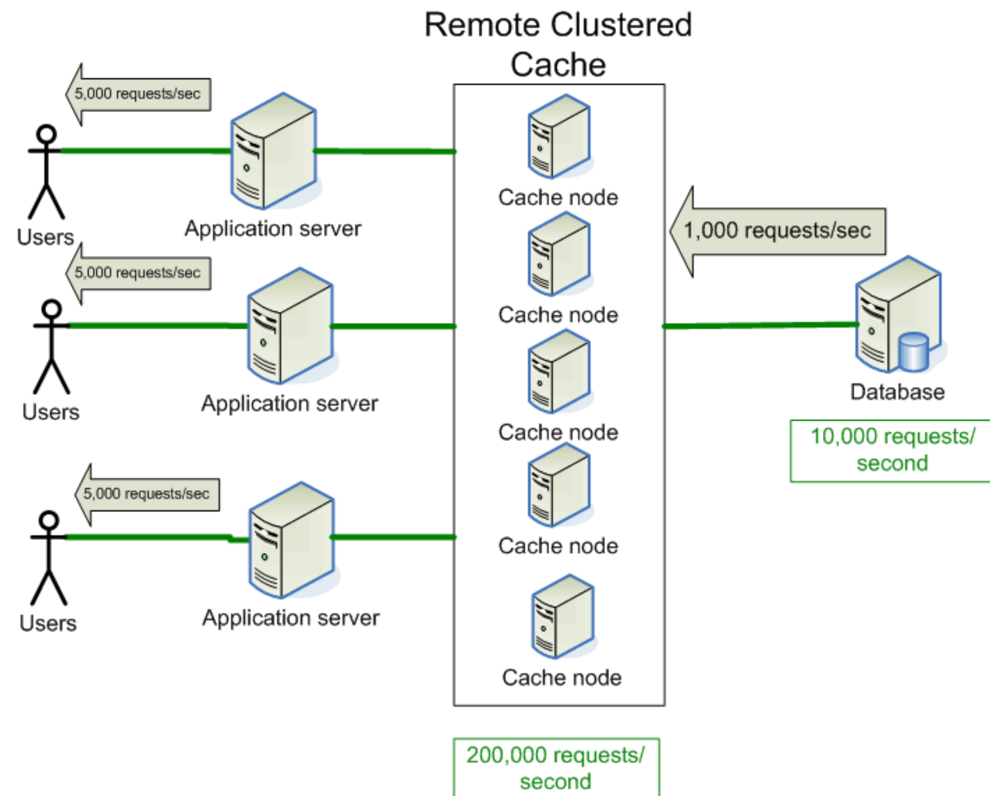
Copyright © 2011 Project Cacheonix

Solution to Scalability Problem

Copyright © 2011 Project Cacheonix

Add Distributed Cache

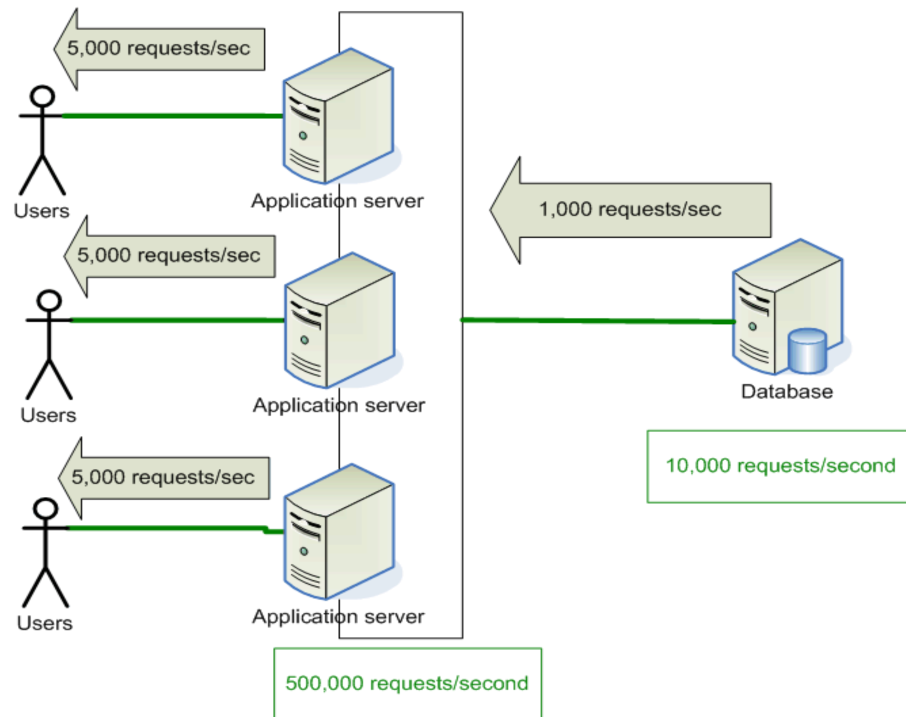
- Bottleneck is removed
- System is reading mostly from the cache
- Distributed cache provides large cache and load balancing



Copyright © 2011 Project Cacheonix

In-Process Distributed Cache

An in-process distributed cache provides memory-like speed and coherent and consistent data access

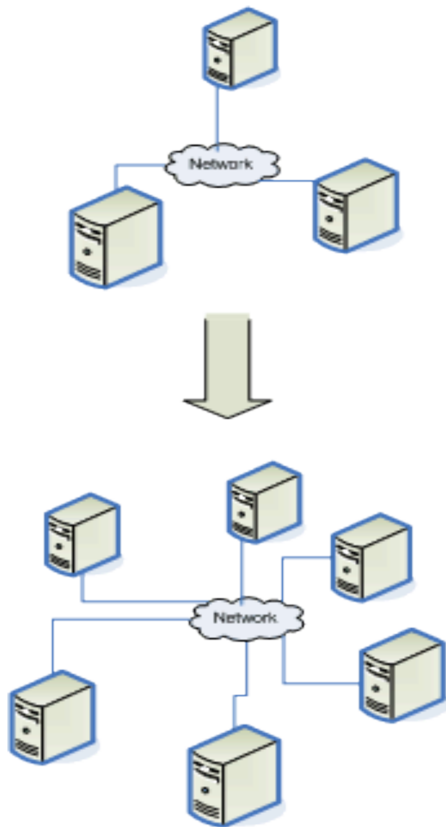


Copyright © 2011 Project Cacheonix

Best Practices

Copyright © 2011 Project Cacheonix

Best Practice: Scale Out by Adding More Servers



More cache nodes means:

- Smaller partition size
- Lesser node traffic
- Reduced load
- Smaller GC delays
- Higher availability

Copyright © 2011 Project Cacheonix

Best Practice: Scale Out by Adding More Servers

More cache nodes means:

- Bigger distributed cache
- Better performance

Copyright © 2011 Project Cacheonix

Best Practice: Design for Scalability Upfront

- Design for scalability won't emerge on its own
- Design for loads the worst case x10
- Accommodate going distributed
- Good designs are easy to optimize

Copyright © 2011 Project Cacheonix

Best Practice: Optimize before Caching

Optimize late:

- Avoid premature optimization
- Profile using a decent profiler. We prefer JProfiler
- Grow a local profiling expert
- Use synthetic point load tests
- Run realistic end-to-end load tests

Copyright © 2011 Project Cacheonix

Best Practice: Automate Problem Detection

Automate detection of performance problems:

- PMD
- FindBugs
- KlocWork

Copyright © 2011 Project Cacheonix

Best Practice: Stay Local before Going Distributed

- Scale vertically first (better CPU, more RAM)

Copyright © 2011 Project Cacheonix

Best Practice: Stay Local before Going Distributed

- Scale vertically first (better CPU, more RAM)
- Go distributed only when opportunities for vertical scalability are completely exhausted

Copyright © 2011 Project Cacheonix

Best Practice: Stay Local before Going Distributed

- Scale vertically first (better CPU, more RAM)
- Go distributed only when opportunities for vertical scalability are completely exhausted
- A distributed cache is slower than a local one because it must use network I/O and more CPU to maintain coherence, partitioning and replication

Copyright © 2011 Project Cacheonix

Best Practice: Stay Local before Going Distributed

- Scale vertically first (better CPU, more RAM)
- Go distributed only when opportunities for vertical scalability are completely exhausted
- A distributed cache is slower than a local one because it must use network I/O and more CPU to maintain coherence, partitioning and replication
- Distributed systems require additional configuration, testing and network infrastructure.

Copyright © 2011 Project Cacheonix

Best Practice: Stay Local before Going Distributed

- Scale vertically first (better CPU, more RAM)
- Go distributed only when opportunities for vertical scalability are completely exhausted
- A distributed cache is slower than a local one because it must use network I/O and more CPU to maintain coherence, partitioning and replication
- Distributed systems require additional configuration, testing and network infrastructure.
- There are some licensing costs associated with distributed caching solutions that work

Copyright © 2011 Project Cacheonix

Best Practice: Cache Right Objects

Cache objects that are expensive to get

- Results of database queries
- I/O
- XML
- XSL

Copyright © 2011 Project Cacheonix

Best Practice: Cache Right Objects

Cache objects that are expensive to get

- Results of database queries
- I/O
- XML
- XSL

Cache objects that are read-mostly

- Guarantees high hit/miss ratio and
- Low cache maintenance and
- Low cache coherence and replication costs

Copyright © 2011 Project Cacheonix

Antipattern: "Cache Them All"

Don't cache objects that are easy to get:

- Caching makes them harder to get
- Caching complicates design and implementation

Copyright © 2011 Project Cacheonix

Antipattern: "Cache Them All"

Don't cache objects that are easy to get:

- Caching makes them harder to get
- Caching complicates design and implementation

Don't cache write-mostly objects:

- Little to no benefit
- Cache maintenance becomes an expense

Copyright © 2011 Project Cacheonix

Best Practice: Implement java.io.Externalizable

Default Java serialization is too slow

- Does a lot of useless things, automatically
- Was developed with networked object transfers in mind
- Is done by simply implementing signature interface java.io.Serializable

Copyright © 2011 Project Cacheonix

Best Practice: Implement java.io.Externalizable

java.io.Externalizable

- Can be significantly faster (2-8 times than default serialization)
- 2-4 times smaller byte footprint – higher network throughput
- Requires additional code

Copyright © 2011 Project Cacheonix

Externalizable Example

```
public final class LineItemKey implements Externalizable {

  private int invoiceID;
  private int lineItemID;

  public LineItemKey() {
  }

  public LineItemKey(final int invoiceID, final int lineItemID) {
    this.invoiceID = invoiceID;
    this.lineItemID = lineItemID;
  }

  public int getInvoiceID() {
    return invoiceID;
  }

  public int getLineItemID() {
    return lineItemID;
  }

  public int hashCode() {
    int result = invoiceID;
    result = 29 * result + lineItemID;
    return result;
  }

  public boolean equals(final Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    final LineItemKey that = (LineItemKey) o;
    if (invoiceID != that.invoiceID) return false;
    if (lineItemID != that.lineItemID) return false;
    return true;
  }

  public void writeExternal(final ObjectOutput oo) throws IOException {
    oo.writeInt(invoiceID);
    oo.writeInt(lineItemID);
  }

  public void readExternal(final ObjectInput oi) throws IOException, ClassNotFoundException {
    invoiceID = oi.readInt();
    lineItemID = oi.readInt();
  }

  public String toString() {
    return "LineItemKey{" +
            "invoiceID=" + invoiceID +
            ", lineItemID=" + lineItemID +
            '}';
  }
}
```

Copyright © 2011 Project Cacheonix

Externalizable Example

```
public final class LineItemKey implements Externalizable {

  private int invoiceID;
  private int lineItemID;

  public LineItemKey() {
  }

  public LineItemKey(final int invoiceID, final int lineItemID) {
    this.invoiceID = invoiceID;
    this.lineItemID = lineItemID;
  }

  public int getInvoiceID() {
    return invoiceID;
  }

  public int getLineItemID() {
    return lineItemID;
  }

  public int hashCode() {
    int result = invoiceID;
    result = 29 * result + lineItemID;
    return result;
  }

  public boolean equals(final Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    final LineItemKey that = (LineItemKey) o;
    if (invoiceID != that.invoiceID) return false;
    if (lineItemID != that.lineItemID) return false;
    return true;
  }

  public void writeExternal(final ObjectOutput oo) throws IOException {
    oo.writeInt(invoiceID);
    oo.writeInt(lineItemID);
  }

  public void readExternal(final ObjectInput oi) throws IOException, ClassNotFoundException {
    invoiceID = oi.readInt();
    lineItemID = oi.readInt();
  }

  public String toString() {
    return "LineItemKey{" +
           "invoiceID=" + invoiceID +
           ", lineItemID=" + lineItemID +
           '}';
  }
}
```

Copyright © 2011 Project Cacheonix

Externalizable Example

```
public void writeExternal(final ObjectOutput oo)
       throws IOException {

  oo.writeInt(invoiceID);
  oo.writeInt(lineItemID);
}

public void readExternal(final ObjectInput oi)
       throws IOException, ClassNotFoundException {

  invoiceID = oi.readInt();
  lineItemID = oi.readInt();
}
```

Copyright © 2011 Project Cacheonix

Best Practice: Test for Serializability

- You must ensure that the object that was received at another end is the object that was sent
- Cache keys AND cached values routinely travel across the network
- It is critical to write proper serialization tests for keys and values

Copyright © 2011 Project Cacheonix

Best Practice: Test for Serializability

- Test pattern: Serialize, deserialize, compare

Copyright © 2011 Project Cacheonix

Best Practice: Test for Serializability

```java
public final class InvoiceKeyTest extends TestCase {

  public InvoiceKeyTest(String name) {
    super(name);
  }


  /**
   * Tests that the key can travel across the network.
   */
  public void testSerializeDeserialize() throws IOException, ClassNotFoundException {

    // Create an object under test
    int invoiceID = 1;
    InvoiceKey originalInvoiceKey = new InvoiceKey(invoiceID);

    // Serialise the object
    ByteArrayOutputStream baos = new ByteArrayOutputStream(100);
    ObjectOutputStream oos = new ObjectOutputStream(baos);
    oos.writeObject(originalInvoiceKey);
    oos.close();

    // Deserialize the object in serialized form
    byte[] serializedInvoiceKey = baos.toByteArray();
    ByteArrayInputStream bais = new ByteArrayInputStream(serializedInvoiceKey);
    ObjectInputStream ois = new ObjectInputStream(bais);
    InvoiceKey deserializedInvoiceKey = (InvoiceKey) ois.readObject();
    ois.close();

    // Assert object went through serialization without any problem
    assertEquals(originalInvoiceKey, deserializedInvoiceKey);

    // Do per-field comparison if necessary
    assertEquals(invoiceID, deserializedInvoiceKey.getInvoiceID());
  }
}
```

Copyright © 2011 Project Cacheonix

Best Practice: Test for Serializability

```java
public final class InvoiceKeyTest extends TestCase {

  public InvoiceKeyTest(String name) {
    super(name);
  }


  /**
   * Tests that the key can travel across the network.
   */
  public void testSerializeDeserialize() throws IOException, ClassNotFoundException {

    // Create an object under test
    int invoiceID = 1;
    InvoiceKey originalInvoiceKey = new InvoiceKey(invoiceID);

    // Serialise the object
    ByteArrayOutputStream baos = new ByteArrayOutputStream(100);
    ObjectOutputStream oos = new ObjectOutputStream(baos);
    oos.writeObject(originalInvoiceKey);
    oos.close();

    // Deserialize the object in serialized form
    byte[] serializedInvoiceKey = baos.toByteArray();
    ByteArrayInputStream bais = new ByteArrayInputStream(serializedInvoiceKey);
    ObjectInputStream ois = new ObjectInputStream(bais);
    InvoiceKey deserializedInvoiceKey = (InvoiceKey) ois.readObject();
    ois.close();

    // Assert object went through serialization without any problem
    assertEquals(originalInvoiceKey, deserializedInvoiceKey);

    // Do per-field comparison if necessary
    assertEquals(invoiceID, deserializedInvoiceKey.getInvoiceID());
  }
}
```

Copyright © 2011 Project Cacheonix

Best Practice: Test for Serializability

```java
// Create an object under test
int invoiceID = 1;
InvoiceKey originalInvoiceKey = new InvoiceKey(invoiceID);

// Serialise the object
ByteArrayOutputStream baos = new ByteArrayOutputStream(100);
ObjectOutputStream oos = new ObjectOutputStream(baos);
oos.writeObject(originalInvoiceKey);
oos.close();

// Deserialize the object in serialized form
byte[] serializedInvoiceKey = baos.toByteArray();
ByteArrayInputStream bais = new ByteArrayInputStream(serializedInvoiceKey);
ObjectInputStream ois = new ObjectInputStream(bais);
InvoiceKey deserializedInvoiceKey = (InvoiceKey) ois.readObject();
ois.close();

// Assert object went through serialization without any problem
assertEquals(originalInvoiceKey, deserializedInvoiceKey);

// Do per-field comparison if necessary
assertEquals(invoiceID, deserializedInvoiceKey.getInvoiceID());
```

Copyright © 2011 Project Cacheonix

Best Practice: Split Large RAM Between Multiple JVMs

Big fat boxes have become common:

- 8 CPU cores 32Gb RAM
- All modern 64 bit JVMs support large heaps

Copyright © 2011 Project Cacheonix

Best Practice: Split Large RAM Between Multiple JVMs

Big fat boxes have become common:

- 8 CPU cores 32Gb RAM
- All modern 64 bit JVMs support large heaps

Problem:

- Large heaps mean long major GCs (10s of seconds)
- Cluster nodes seem to appear gone causing cluster configuration jitter

Copyright © 2011 Project Cacheonix

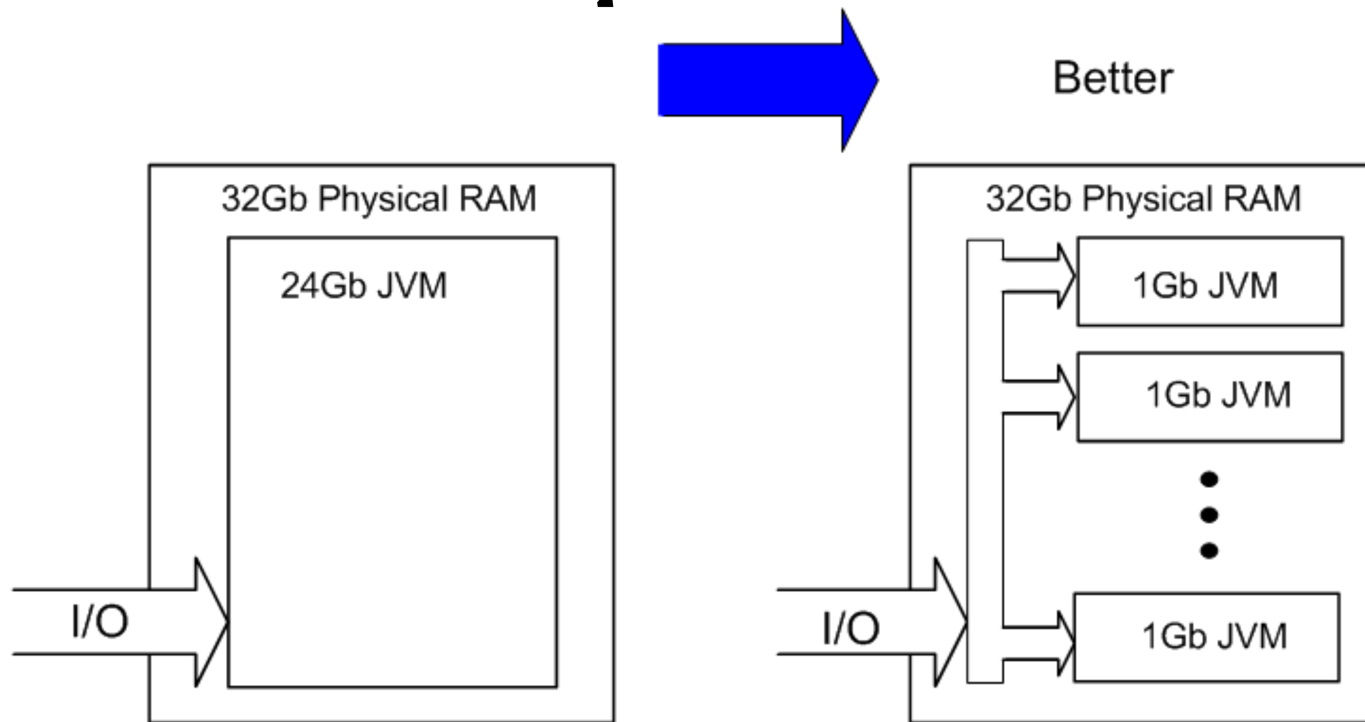
Best Practice: Split Large RAM Between Multiple JVMs

Solution: Split large RAM into multiple 1-2Gb JVMs

- Distributed caching allows to split data processing into many JVMs.
- Shorter major GCs mean lesser latency and more stable cluster
- Nice side effects such as higher availability, better load balancing and improved concurrency

Copyright © 2011 Project Cacheonix

Best Practice: Split Large RAM Between Multiple JVMs



Copyright © 2011 Project Cacheonix

Problem: Distributed Caching Adds Network Traffic

- Remote partition access
- Cache coherency traffic
- Replication traffic

Copyright © 2011 Project Cacheonix

Problem: Distributed Caching Adds Network Traffic

- Remote partition access
- Cache coherency traffic
- Replication traffic

Cache being on the same network with the application leads to:

- Increased cache access latency
- Increased application response time

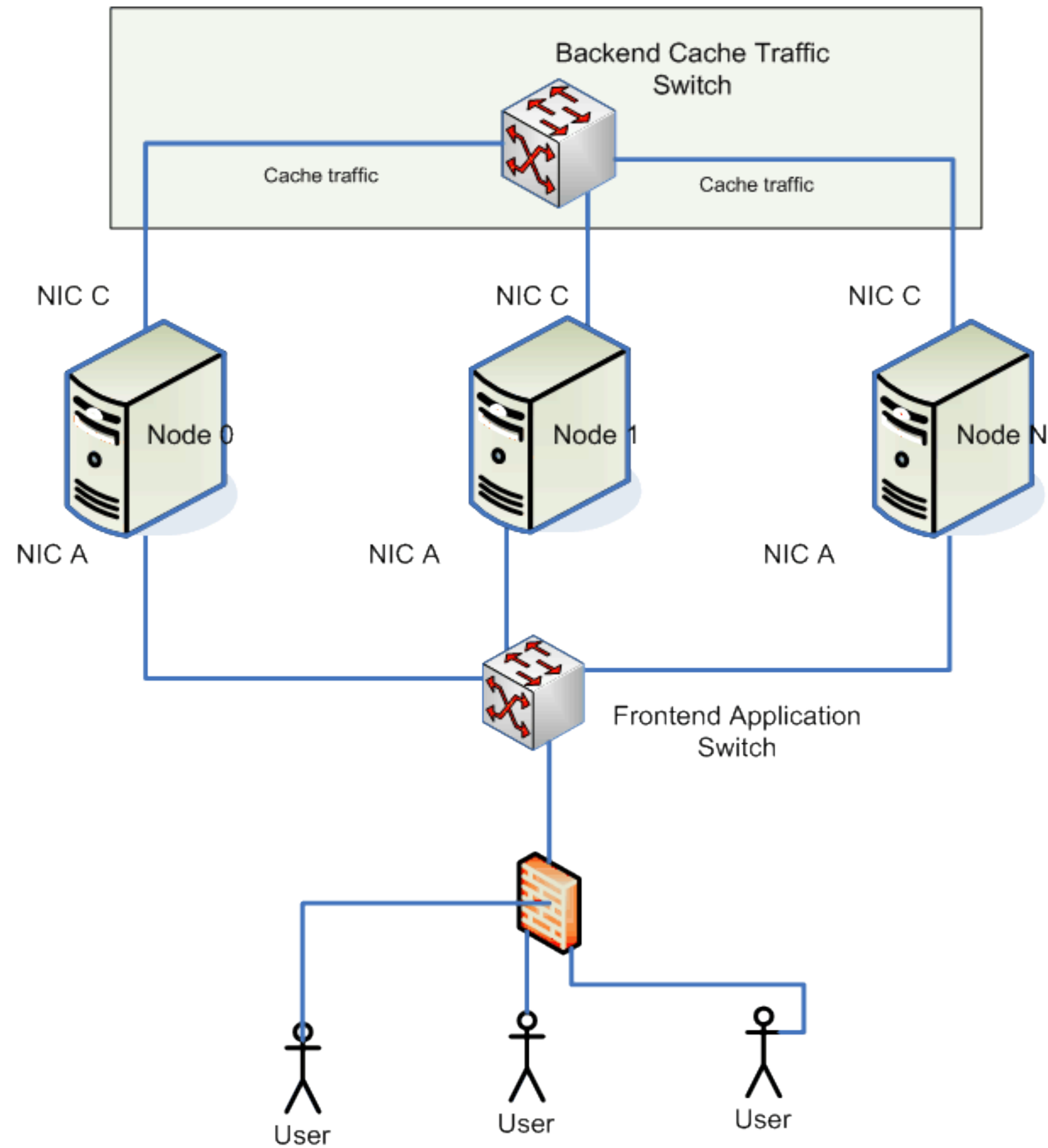
Copyright © 2011 Project Cacheonix

Best Practice: Provide Dedicated Network Infrastructure

Solution: Dedicate separate network to distributed cache traffic:

1. Add a network card. Most of the modern rackmount servers already have two NICs
2. Add a separate switch to serve the distributed cache traffic

Copyright © 2011 Project Cacheonix



Copyright © 2011 Project Cacheonix

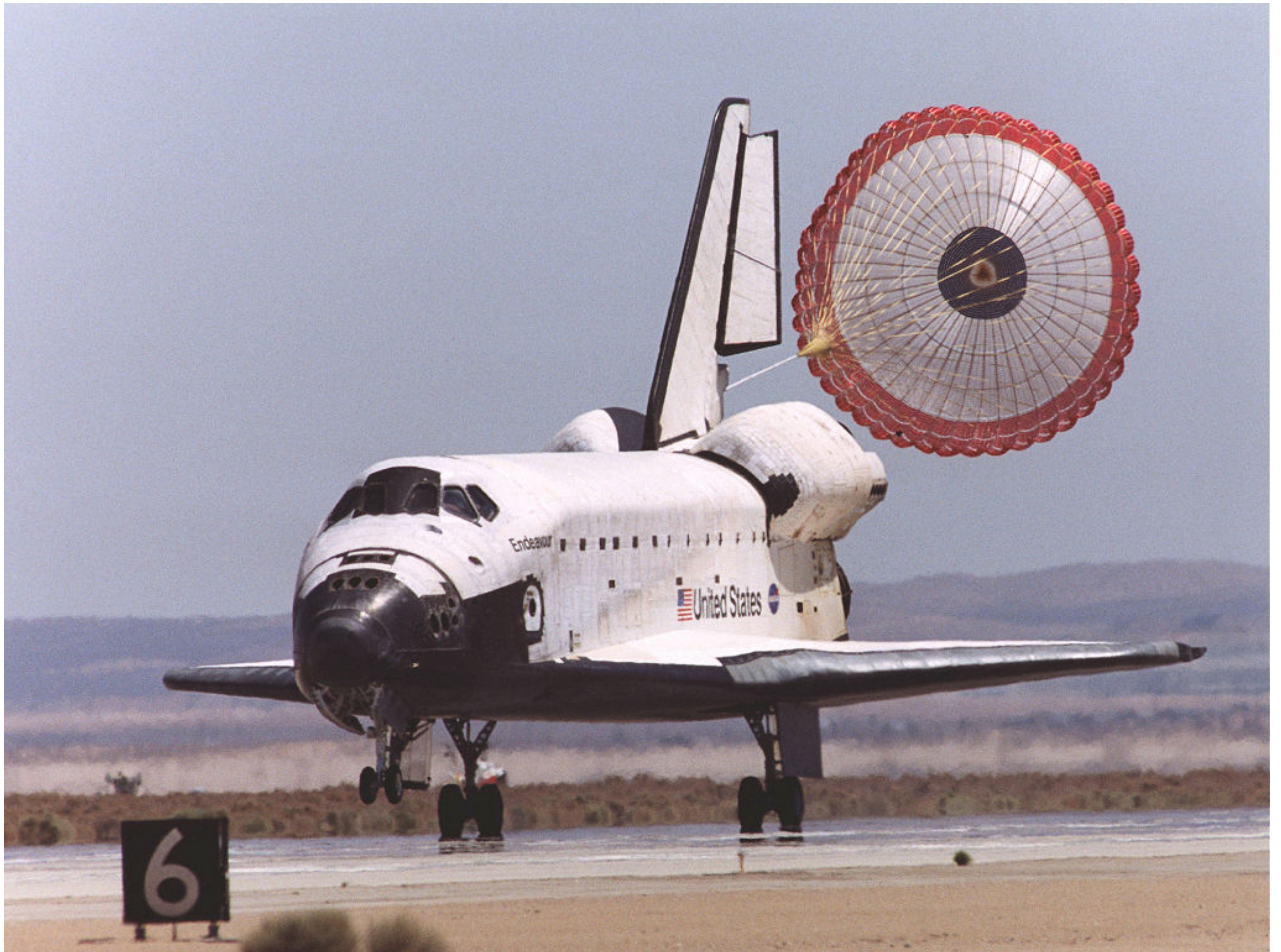
Best Practice: Use Multicast

- Most of modern caching solutions efficiently utilize multicast.
- If done right, multicast provides significant reduction in network traffic (~100 of times)

Copyright © 2011 Project Cacheonix

"Best Practice is
a technique or
methodology that,
through experience
and research, has
proven to reliably lead
to a desired result."

Copyright © 2011 Project Cacheonix



Endeavour
United States
6

Q&A

Q: How does replication work in Cacheonix? Is it master/slave?

A: Cacheonix replication protocol is more advanced then master/slave. In Cacheonix every cache node carries a partition that it owns, and a set of partition replicas. This allows Cacheonix restore operational partition from a replica automatically and instantaneously.

Copyright © 2011 Project Cacheonix

Q&A

Q: Does Cacheonix allow to access cached data so that some clients see updates in progress and some don't

A: Cacheonix supports this scenario by providing distributed reliable read/write locks. If the code wants to be shielded from the transactions in progress it should access the cache inside a lock. Otherwise just read/write the data as usual.

Copyright © 2011 Project Cacheonix

Q&A

Q: So, Cacheonix provides strict data consistency when it comes to updates. How does it work?

A: Cacheonix builds its data access capability on its very sophisticated cluster management protocol that allows it to guarantee consistent data access even when servers fail, leave or join the cluster while keeping latency low. Cacheonix supports disabling strict consistency for situations when speed is more important.

Copyright © 2011 Project Cacheonix

Q&A

Q: Does Cacheonix provide data grid functionality?

A: Cacheonix fully supports operating as a data grid where a cache is the only source of application data. Cacheonix does so by providing DataSource and DataStore APIs that it uses as a backed data source for its read-through and write-through caches.

Copyright © 2011 Project Cacheonix

Q&A

Q: How does Cacheonix compare to other distributed caching products?

A: Unlike other products Cacheonix allows to utilize multi-core machines fully by running each cache in a separate thread. Cacheonix offers least time for recovery from server failures by making all servers equal, by not having a single point of failure. Also, Cacheonix offers many unique features that are great for developing low-latency systems such as coherent local front caches and read-ahead caches.

Copyright © 2011 Project Cacheonix

Q&A

Q: Should I have a single cache or many caches?

A: A best practice is to have multiple caches that names reflect types values stored in them. Usually those are either per-object such as my.app.Invoice or per-query such as my.app.InoiceQueryResult. Hiberhate requres cache names match names of persistent objects. This practice provides best concurrency Cacheonix as it runs each cache in a separate thread.

Copyright © 2011 Project Cacheonix

Q&A

Q: Aren't automatic serialization frameworks more convenient that implementing Externalizable, especially when it comes to versioning?

A: First, Externalizable is the closest to wire speed when it comes to serialization. Second, even if a serialization framework can enforce a cached object being a pure value object, there will be hard-to figure out production failures associated with different versions of the system expecting data and not finding it. On the contrary, implementing Externalizable and following best practices for production change management produces faster and more stable system.

Copyright © 2011 Project Cacheonix

Q&A

Q: I am deploying my application in a cloud. How do I know if my cloud provider follows best practices?

A: The best way to find out is to ask them directly. E-mail, call them, or file a request through their web support.

Copyright © 2011 Project Cacheonix

Need help with scaling your application and improving its performance with distributed caching?

Visit Cacheonix at www.cacheonix.org

Copyright © 2011 Project Cacheonix

Downloading Cacheonix

http://downloads.cacheonix.org

Copyright © 2011 Project Cacheonix

Thank you!

Copyright © 2011 Project Cacheonix