Architecture for Scaling Java Applications to Multiple Servers

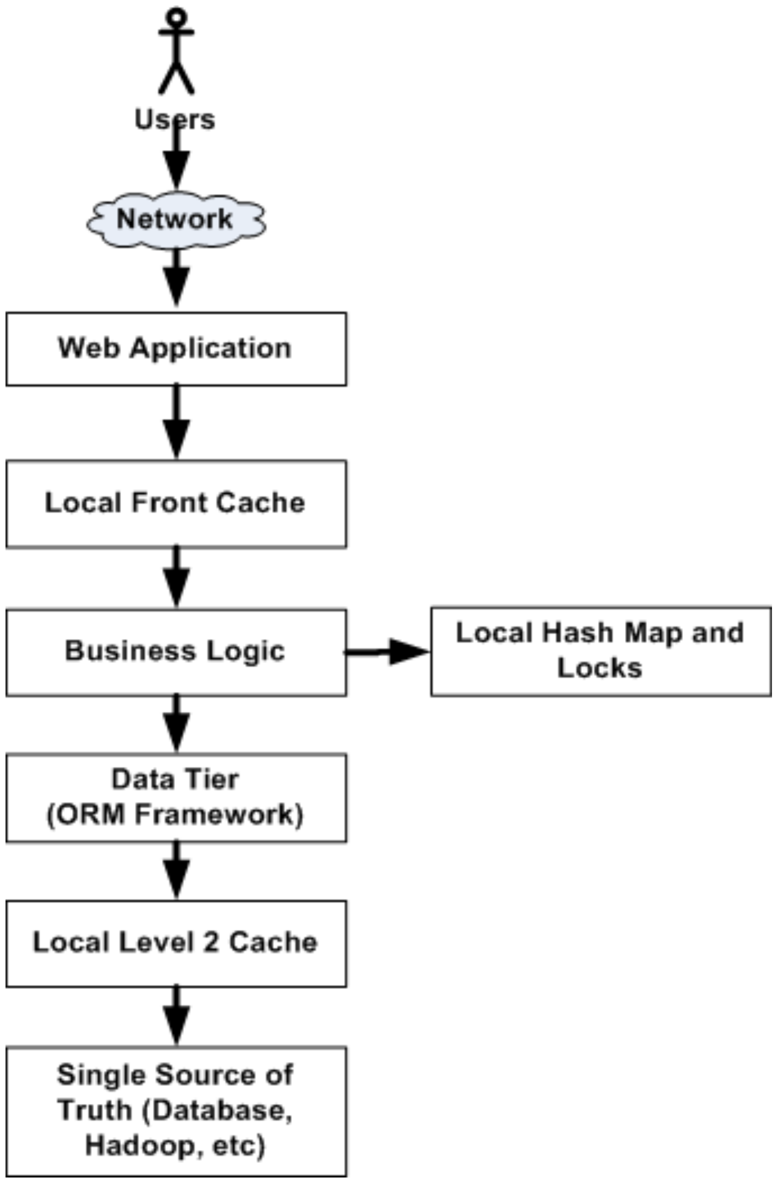
Introduction

Presenter: Slava Imeshev

- Founder, main committer and project lead for open source distributed data management framework Cacheonix
- Frequent speaker on scalability
  - simeshev@cacheonix.com
  - www.cacheonix.com/blog/

БАЙКОНУР

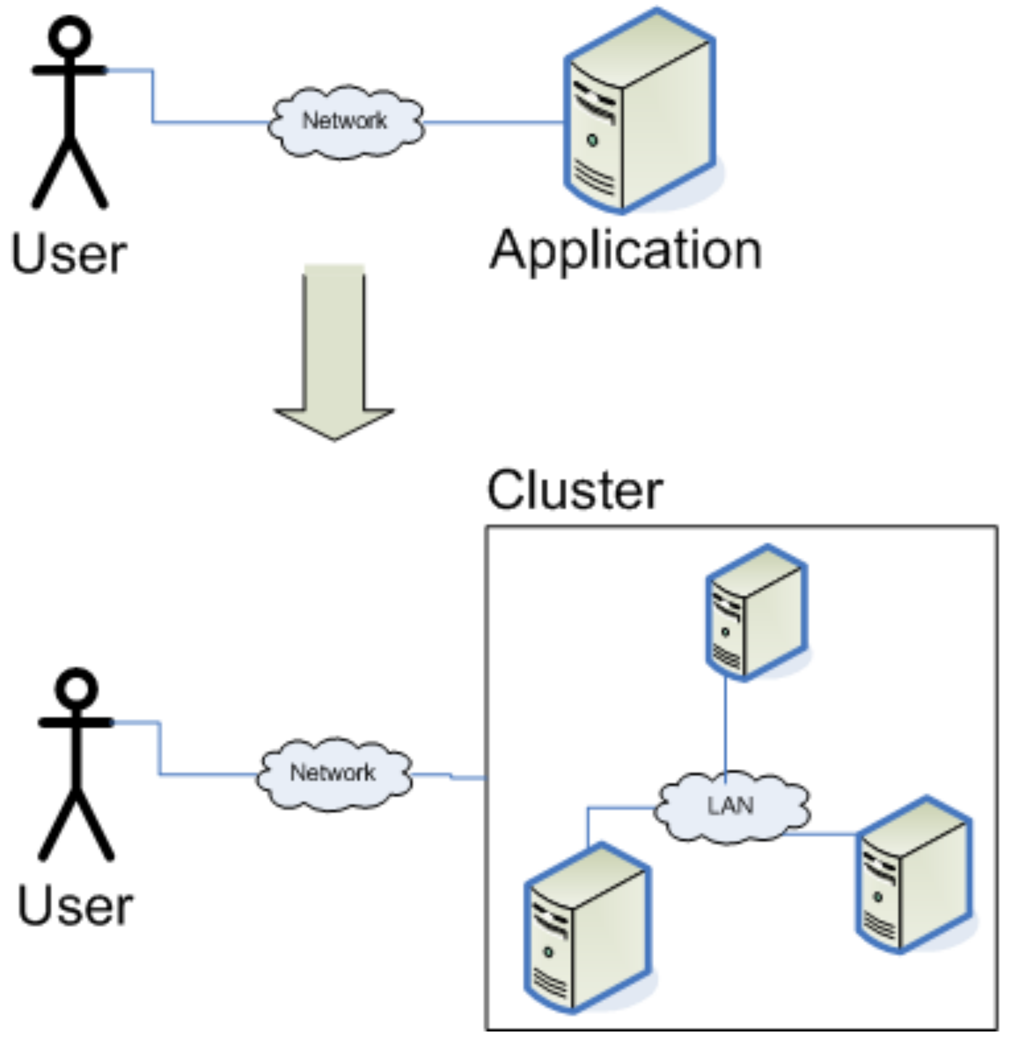
Single-Server Architecture



When Single Server Is Not Enough

- Sooner or later your application will have to process more requests than a single server can handle
- You need to scale your application to multiple servers A.K.A. to scale the application horizontally (LAN, AWS, etc)

Scaling Horizontally



Distributed Systems

- Processes communicate over the network instead of local memory
- Distributed programming is easy to do poorly and surprisingly tricky to do well:
  - The network in unreliable
  - The latency varies wildly
  - The bandwidth is limited
  - Topology changes
  - The network is nonuniform
  - Network costs money

Problems to be Solved by Distributed Applications

Distributed applications must address a lot of concerns that don’t exist in single-JVM applications

1. Horizontal scalability
2. Reliability
3. Concurrency
4. State sharing
5. Data consistency
6. Load balancing
7. Failure management
8. Make sure it is easy to develop!

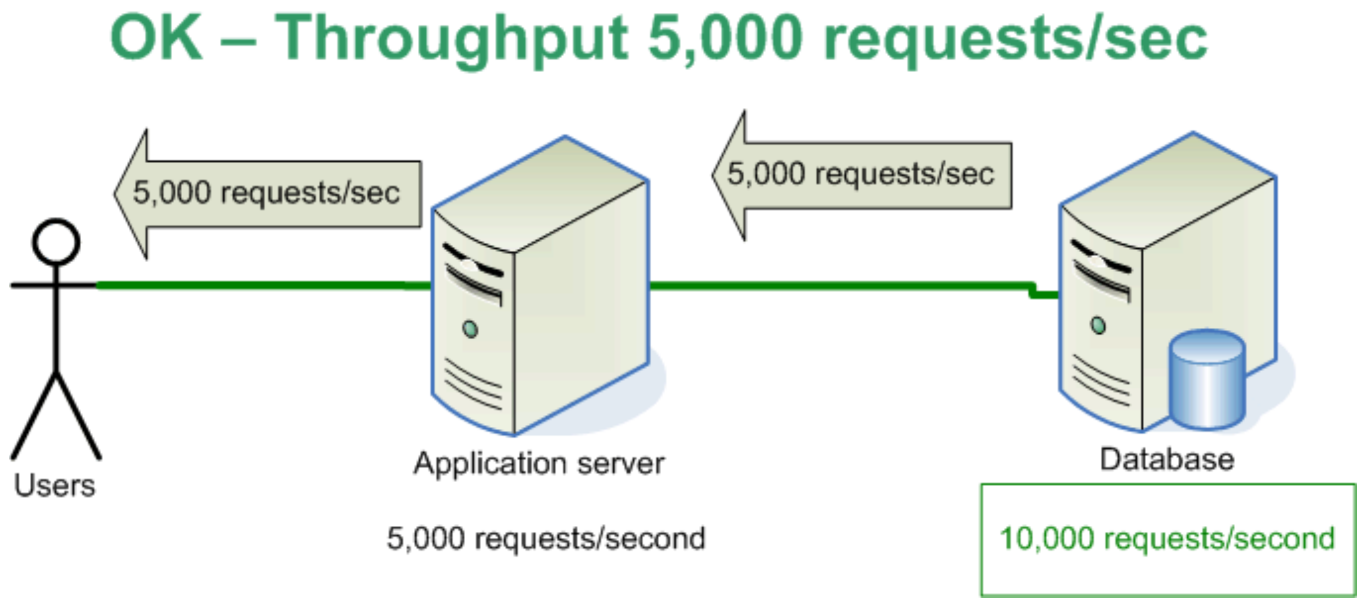
Horizontal Scalability

- Horizontal scalability is an <u>ability to handle additional load by adding more servers</u>
- Horizontal scalability offers a much greater benefit as compared to vertical scalability (2-1000 times improvement in capacity)

Problem of Horizontal Scalability

- Horizontal scalability is hard to achieve because of ever-present bottlenecks
- A <u>bottleneck</u> is a shared server or a service that:
  - All or most requests must go through
  - Request latency is proportional number of requests (100 requests 1 ms/req., 1000 requests 5 ms/req.)
  - Examples: Databases, Hadoop clusters, file systems, mainframes

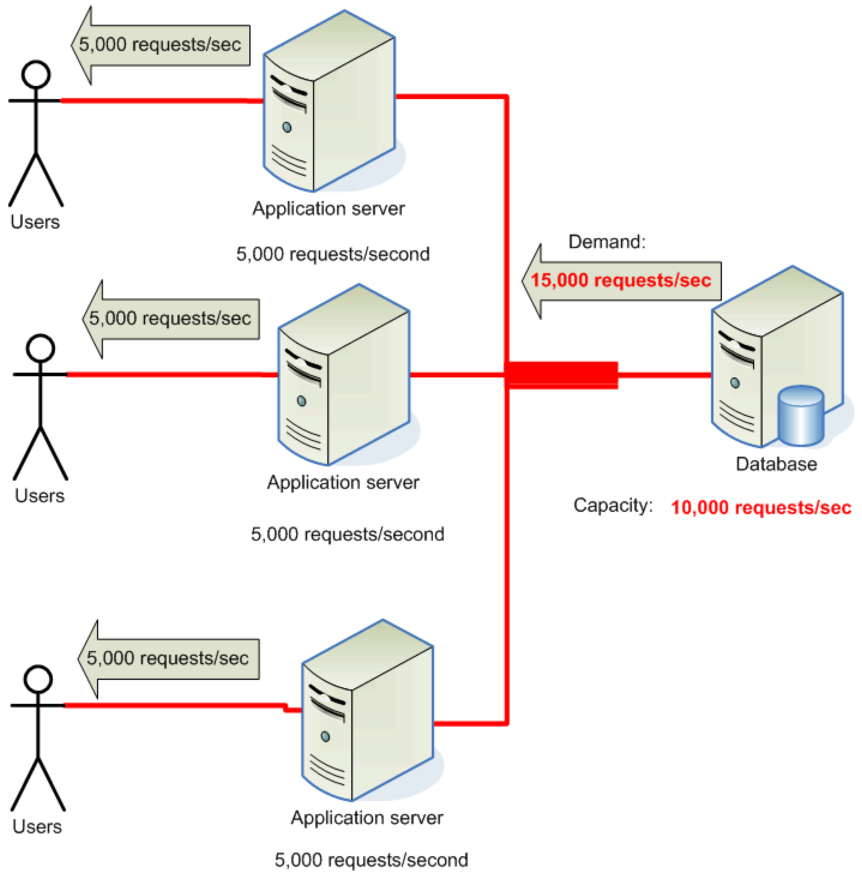
Bottleneck-Free System



Systems That Cannot Scale

- Added 2 more app servers
- Expected x3 increase in capacity
- Got only x2
- System hit scalability limit
- Capacity of the database or other data source is a bottleneck

BAD– Throughput is 10,000 requests/sec, not 15,000



Horizontal Scalability Solution: Distributed Cache

- Distributed cache provides large and fast in-memory data store for frequently-read data
- Application is reading from the cache instead of reading from the slow database

Distributed Cache Requirements

Three key requirements:

- Strict data consistency - an ability to *immediately* observe the result of an update on *all* members of the cluster
- Load balancing – an ability to evenly distribute cached data among servers as members join and leave the cluster
- High availability - an ability to provide uninterrupted, consistent data access in presence of server failures and cluster reconfiguration

Distributed Cache Capabilities

Required capabilities:

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

Reliability Problem

Reliability is an ability of the system to continue to function normally in presence of failures of cluster members

- Processing of user requests must be automatically picked up by operational servers
- Reliability is hard:
  - Cluster members leave and join
  - Networks fail
  - Servers die

Solution to Reliability Problem

- Data replication
- Automatic recovery from failures

Distributed Concurrency Problem

- Threads need to coordinate (synchronize) access to shared objects in order prevent reading partially updated shared objects

Distributed Concurrency Problem

- Distributed concurrency is hard:
  - Servers communicate using a network
  - Servers no longer share memory space
  - Servers may fail while holding locks

Concurrency Solution

- Distributed ReadWriteLocks

Distributed ReadWriteLocks

Required capabilities:

- Fault-tolerant for liveness
  - Locks must be released when a lock-holding server fails or leaves the cluster
- Reliable for high availability
  - Locks must be maintained as long as there is at least a single live server in the cluster
- Strictly consistent
  - All servers must immediately observe mutual exclusions
  - New members of the cluster must observe existing locks

Problem of Distributed State Sharing

- Threads need to access shared state in order to do useful work

Problem of Distributed State Sharing

- Distributed state sharing is hard:
  - Servers communicate using the network
  - Distributed applications no longer share the memory space

Solution to Distributed State Sharing Problem

- Distributed HashMap

Distributed HashMap

Required capabilities:

- Reliable
  - Must retain the data as servers fail or join the cluster
- Strictly consistent
  - Must guarantee that all servers immediately see the updates to the data

Failure Management

Distributed applications experience <u>failures not seen by single-JVM applications</u> because networks are unreliable and servers die

- Event: Cluster partitioning causes a minority cluster to block
- Result: distributed operations may block for extended periods of time to avoid consistency errors

- Event: Cluster reconfiguration leads to leaving the minority cluster and joining the majority cluster
- Result: Locks and other consistent operations in progress are no longer valid and must be cancelled

Failure Management

Required capabilities:

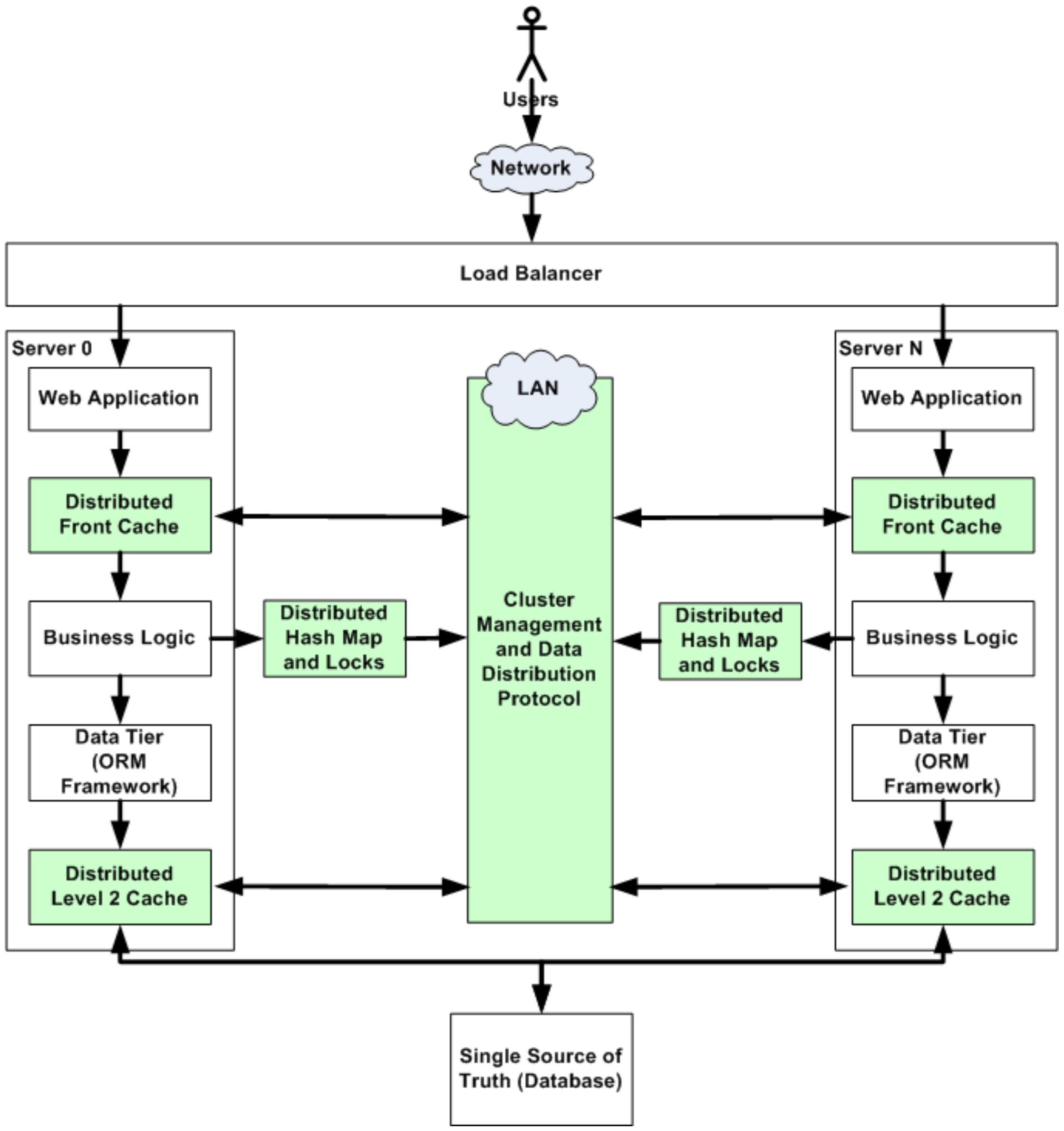
- An ability to report a blocked cluster state for communicating it to the end user
- Detect change in cluster configuration (joining other cluster) and cancel consistent operations by throwing exceptions (lock()/unlock() and put ()/get())
- Prepare the application for dealing with such condition, minimally gracefully reporting a error to the user.

Cluster Management and Data Distribution Protocol

Wire-level protocol that enables

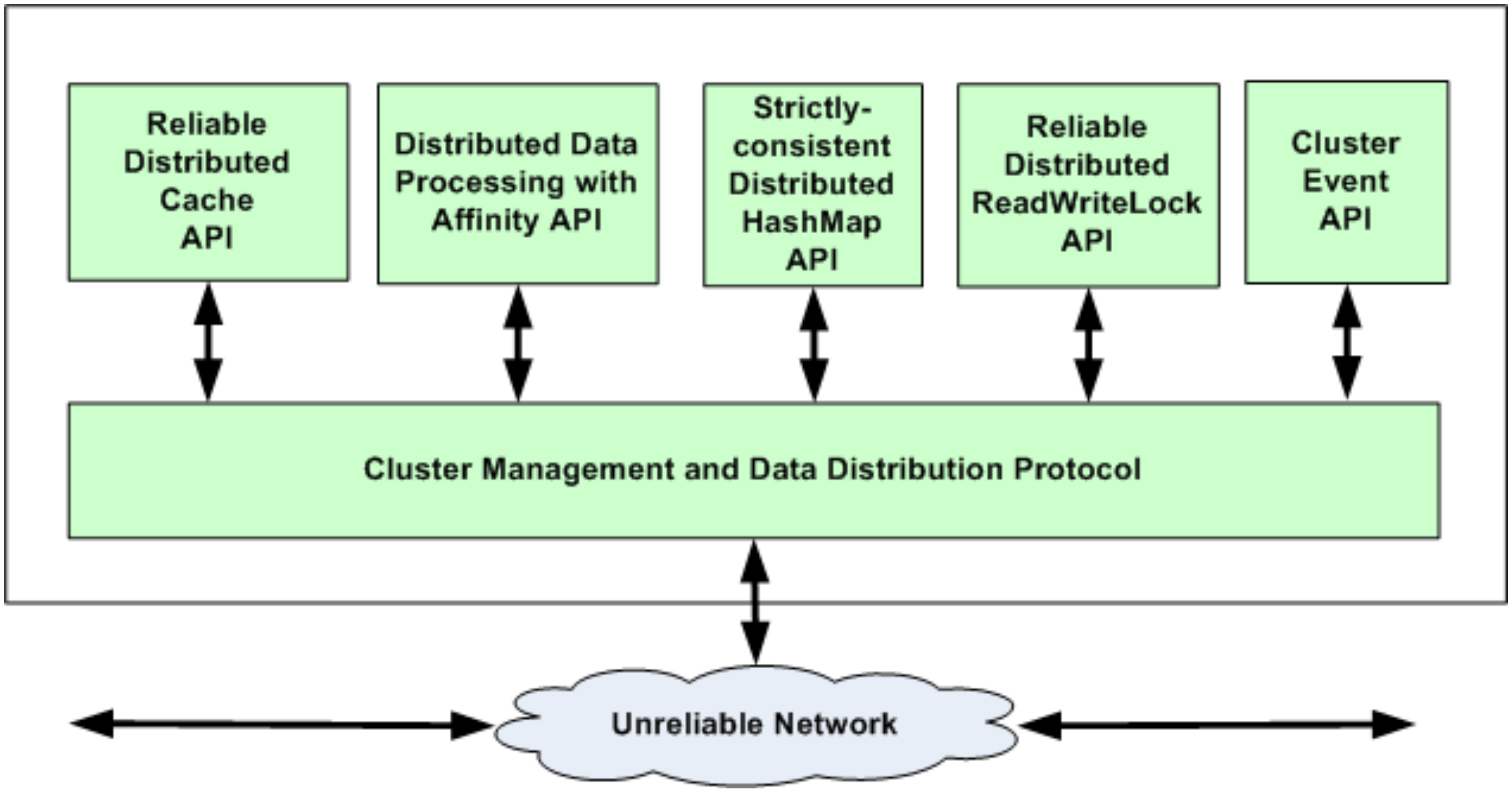
- Distributed caching,
- Data replication,
- Reliable distributed locks,
- State sharing and
- Cluster management

Distributed Architecture



Users
Network
Load Balancer
Server 0
Web Application
Distributed Front Cache
Business Logic
Distributed Hash Map and Locks
Data Tier (ORM Framework)
Distributed Level 2 Cache
LAN
Cluster Management and Data Distribution Protocol
Server N
Web Application
Distributed Front Cache
Distributed Hash Map and Locks
Business Logic
Data Tier (ORM Framework)
Distributed Level 2 Cache
Single Source of Truth (Database)

Tying It All Together: Distributed Data Management Framework


Reliable Distributed Cache API
Distributed Data Processing with Affinity API
Strictly-consistent Distributed HashMap API
Reliable Distributed ReadWriteLock API
Cluster Event API
Cluster Management and Data Distribution Protocol
Unreliable Network

Ease of Development

The set of APIs provided by the distributed data management framework should allow to program distributed applications as easy as if they were singe-JVM applications

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

Best Practice: Design for Extreme Loads Upfront

- Use the architecture for scaling the application to multiple servers.
- Design for scalability won't emerge on its own
- Design for loads the worst case x1000
- Accommodate going distributed
- Good designs are easy to optimize

Best Practice: Stay Local before Going Distributed

- Distributed systems are slower than local ones because they 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 APIs that work

Best Practice: Stay Local before Going Distributed

Scale vertically first:

- Better CPU, more RAM, faster network, SSDs

Optimize:

- Avoid premature optimization
- Profile using a decent profiler (JProfiler is great)
- Use synthetic point load tests
- Run realistic end-to-end load tests

Antipattern: “Cache Them All”

Don’t cache objects that are easy to get:

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

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

Never cache memory allocations

Best Practice: Cache Right Objects

Cache objects that are <u>expensive</u> to get

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

Best Practice: Cache Right Objects

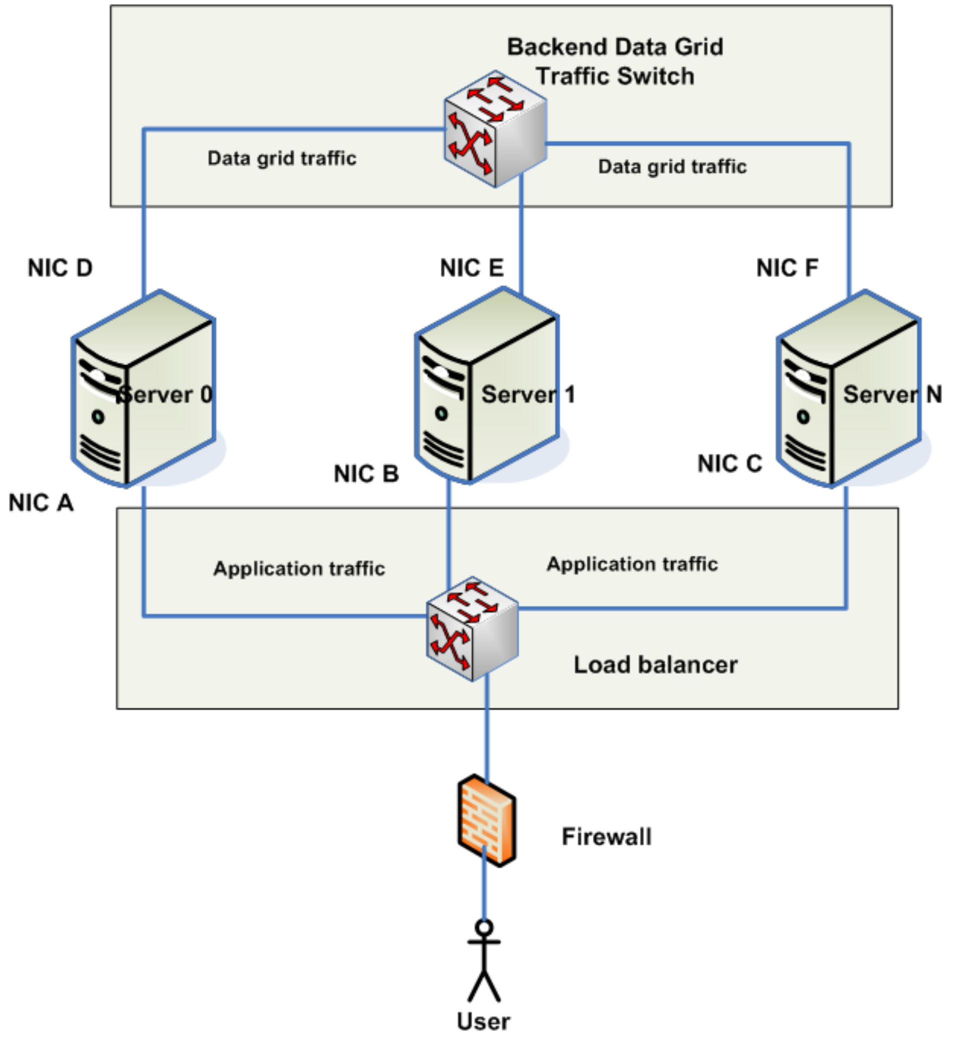
Cache objects that are expensive to get

- 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

Best Practice: Dedicate Separate Network for Backend Traffic



Best Practice: Use Multicast

- Modern caching solutions can use multicast
- Significant reduction in network traffic (~100s of times)

Problem:

- Ops usually disable multicast in production without explanation

Solution:

- Allow them to disable multicast only on edge routers and switches

Best Practice: Use Existing Solutions

- Don't reinvent the wheel (AKA infrastructure software)
- Developing a distributed framework is a lot fun, but:
- Data load balancing is trivial…

  … and the rest is extremely hard:
- Strictly consistent data access in presence of server failures; reliable clustering; replication for high availability; queue theory, state machines, NIO, sockets, messaging…
- It takes about 3-4 years to get it right. What is your plan for the next 3 years?

Q & A

Cacheonix
Open Source Distributed Data Management Framework

- Ease of development,
- Reliable distributed cache,
- Strict data consistency,
- Replicated distributed locks,
- State sharing in a cluster,
- Distributed ConcurrentHashMap,
- Cluster management,
- Fast local cache,

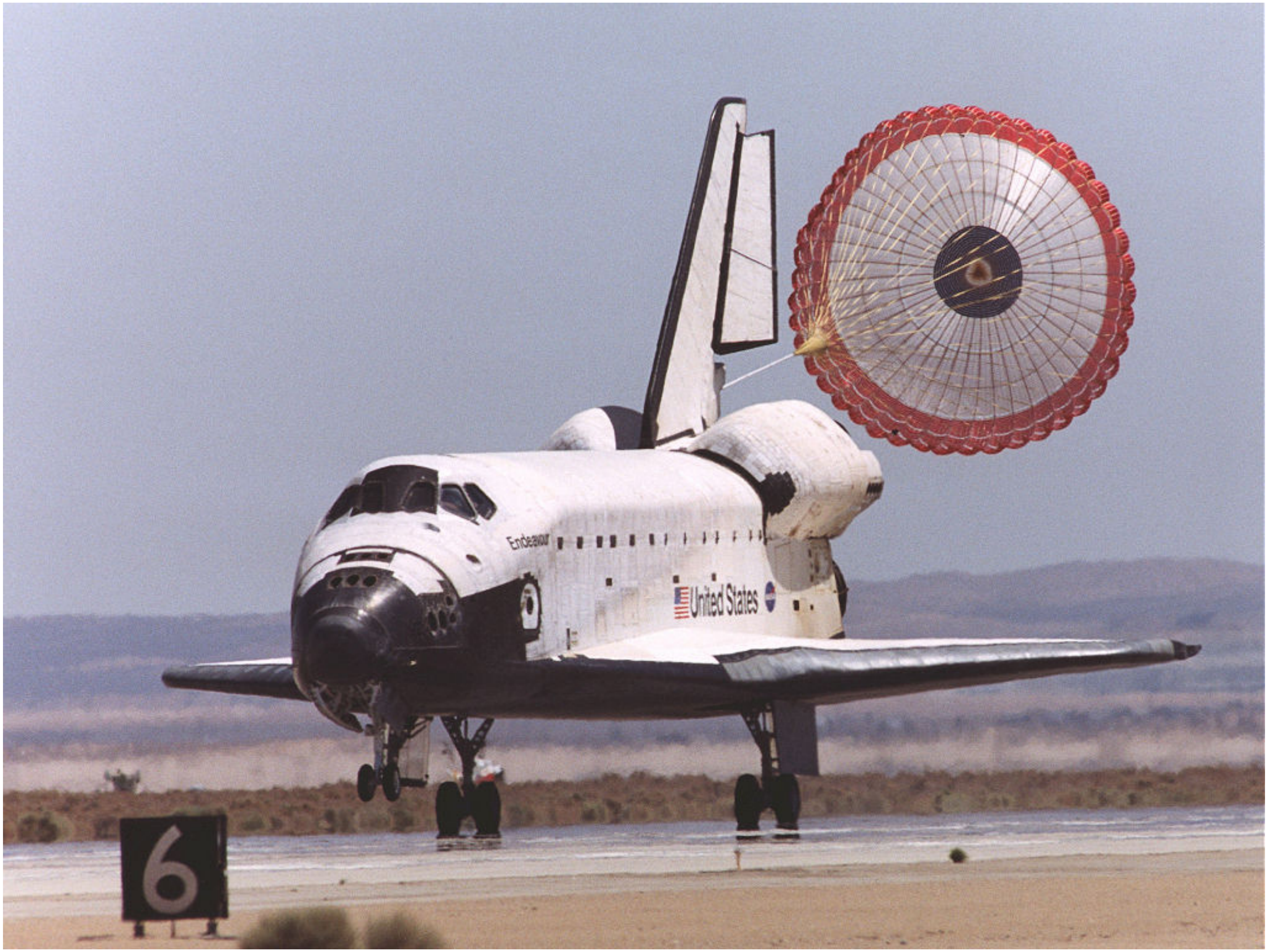
And more!

Download Cacheonix at

downloads.cacheonix.com

Cacheonix wiki:

wiki.cacheonix.com


Endeavour
United States
6