Developing Scalable Java Applications with Cacheonix

Introduction

Presenter: Slava Imeshev

- Founder and main committer for open source distributed Java cache Cacheonix
- Frequent speaker on scalability
  - simeshev@cacheonix.org
  - www.cacheonix.org/blog/

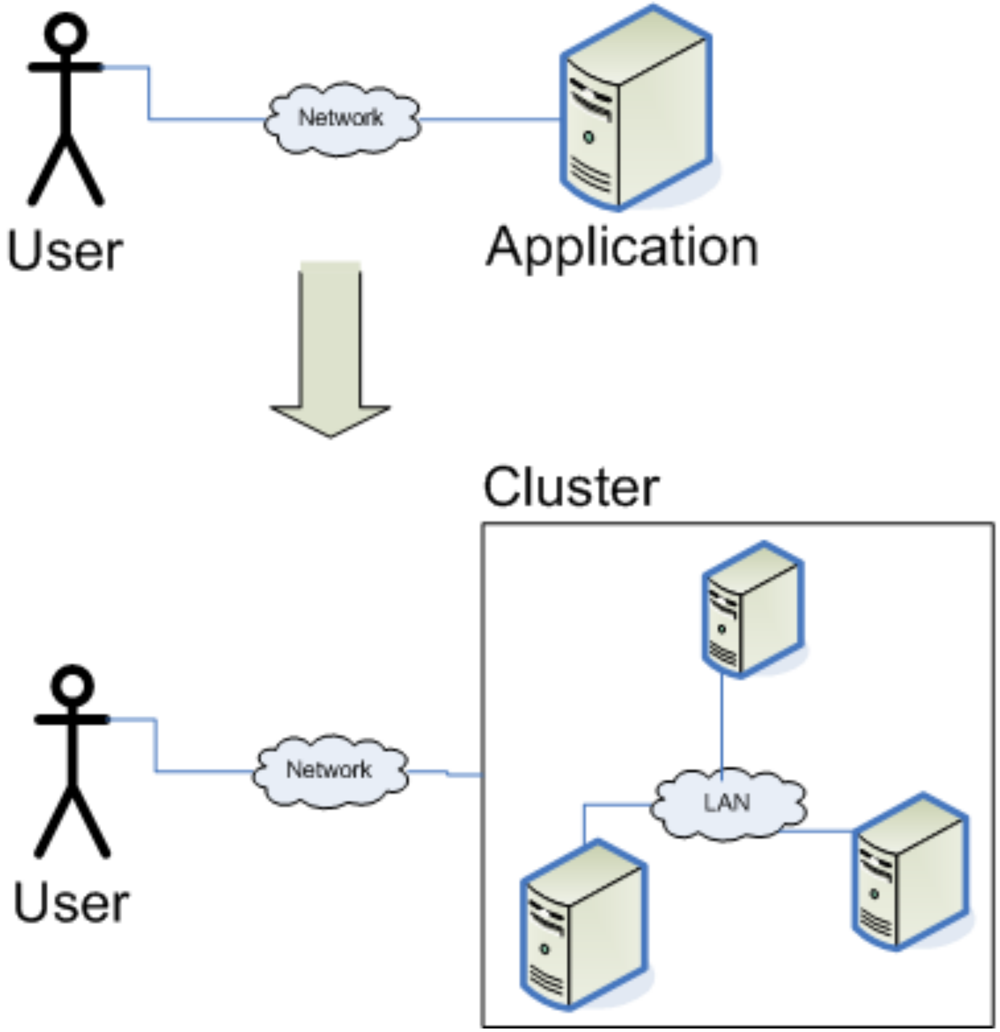
Cacheonix

- An open source distributed Java cache
- Program your distributed applications as easy as if they were singe-JVM applications, with APIs for:
  - Distributed cache
  - Strict data consistency
  - Distributed HashMap
  - In-memory data grid
  - Distributed locks
  - Distributed ConcurrentHashMap
  - Distributed data processing
  - Cluster management
- Open source (LGPL)

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 distribute your application to multiple servers (LAN, AWS, etc)
- A.K.A. horizontal scalability

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

Problems to be Solved by Distributed Applications

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

1. Scalability bottlenecks
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

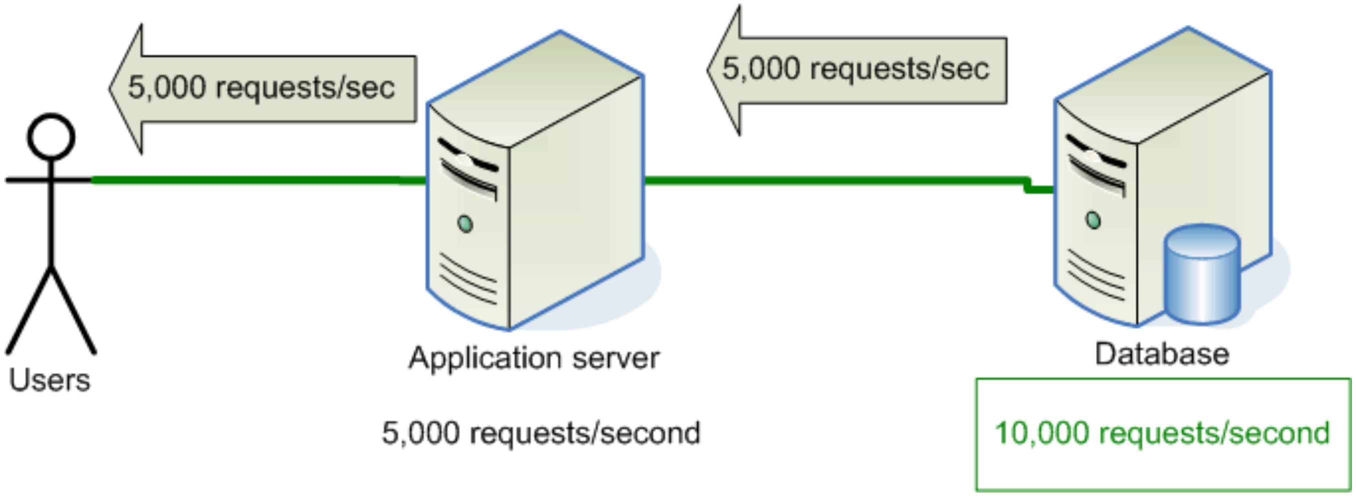
- Horizontal scalability is an ability to handle additional load by adding more servers
- Horizontal scalability offers a much greater benefit as compared to vertical scalability (2-1000 times improvement in capacity)

Bottleneck Problem

- Horizontal scalability is hard to achieve because of ever-present bottlenecks
- A bottleneck 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

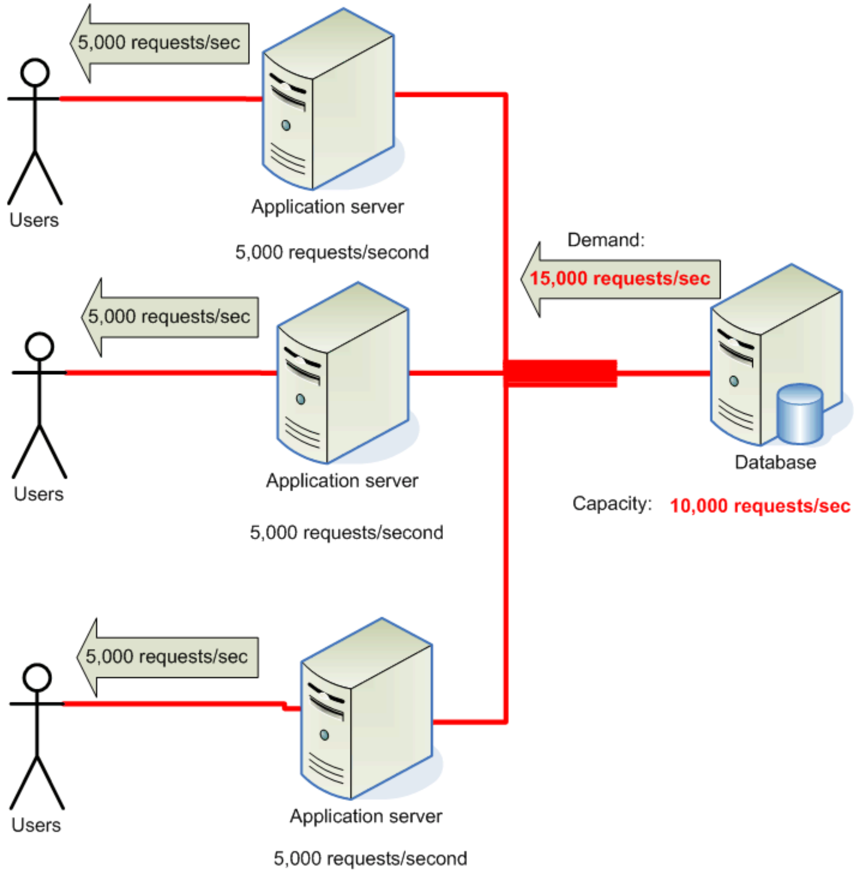
OK – Throughput 5,000 requests/sec



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



Solution To Bottleneck Problem: Distributed Cache

- Cacheonix implements a distributed cache that provides a large clustered in-memory data store for hard-to-get, frequently-read data
- The application is reading from the cache instead of being stuck in reading from the slow database

Distributed Cache

Cacheonix provides:

- Strict data consistency - the result of an update is *immediately* observed on *all* members of the cluster
- Load balancing – cached data is distributed evenly among servers as members join and leave the cluster
- High availability - Cacheonix provides uninterrupted, consistent data access in presence of server failures and cluster reconfiguration

Distributed Cache

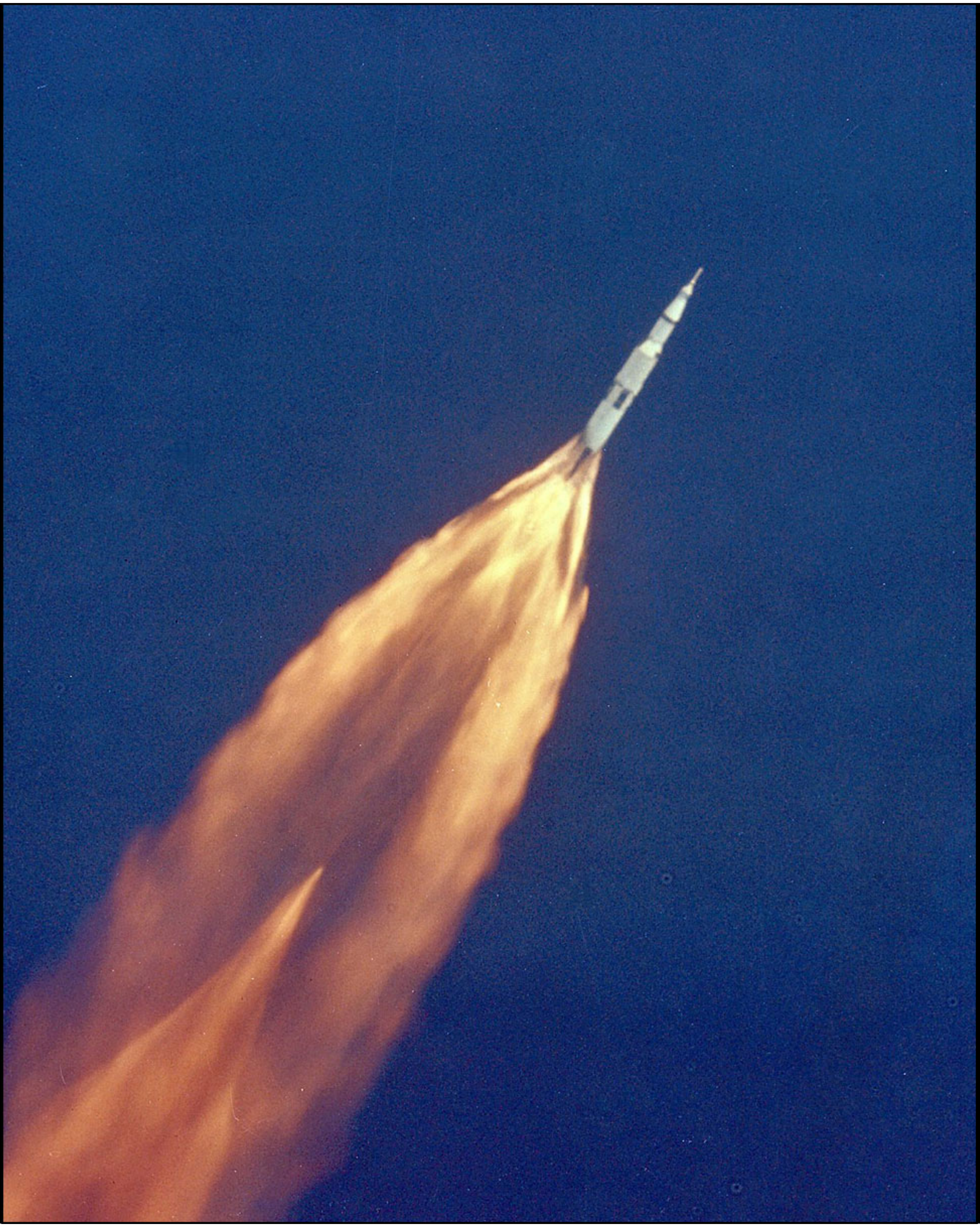
Cacheonix offers:

- Cache coherence for strict data consistency
- Partitioning for load balancing
- Replication for high availability
- Ease of use: Standard java.util.Map interface

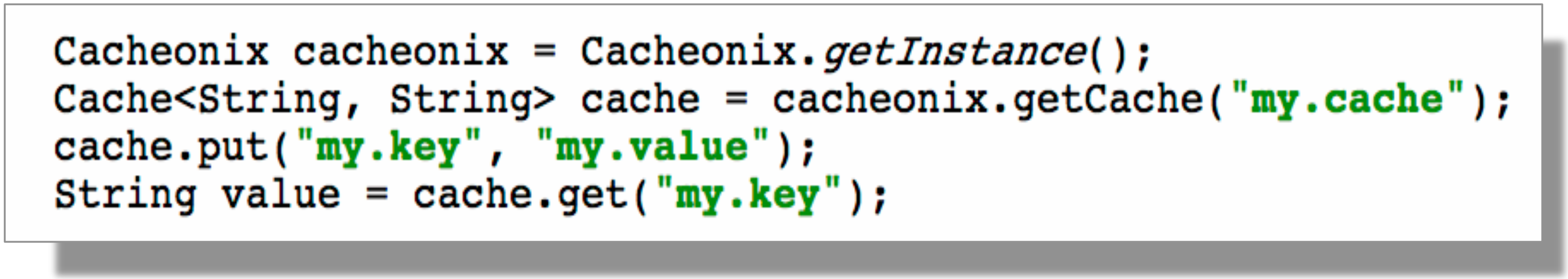
Distributed Cache

Cacheonix cache plugins for ORM frameworks:

- Hibernate
- MyBatis
- DataNucleus



Distributed Cache



```java
Cacheonix cacheonix = Cacheonix.getInstance();
Cache<String, String> cache = cacheonix.getCache("my.cache");
cache.put("my.key", "my.value");
String value = cache.get("my.key");
```

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

Cacheonix provides:

- Data replication
- Even replica storage
- Unique replication protocol
- Instant recovery from failures

Distributed Concurrency Problem

- Threads must prevent reading partially updated shared objects
- Threads need to coordinate (synchronize) access to shared objects
- Distributed concurrency is hard:
  - Servers communicate using a network
  - Servers no longer share memory space
  - Servers may fail while holding locks

Distributed Concurrency Solution

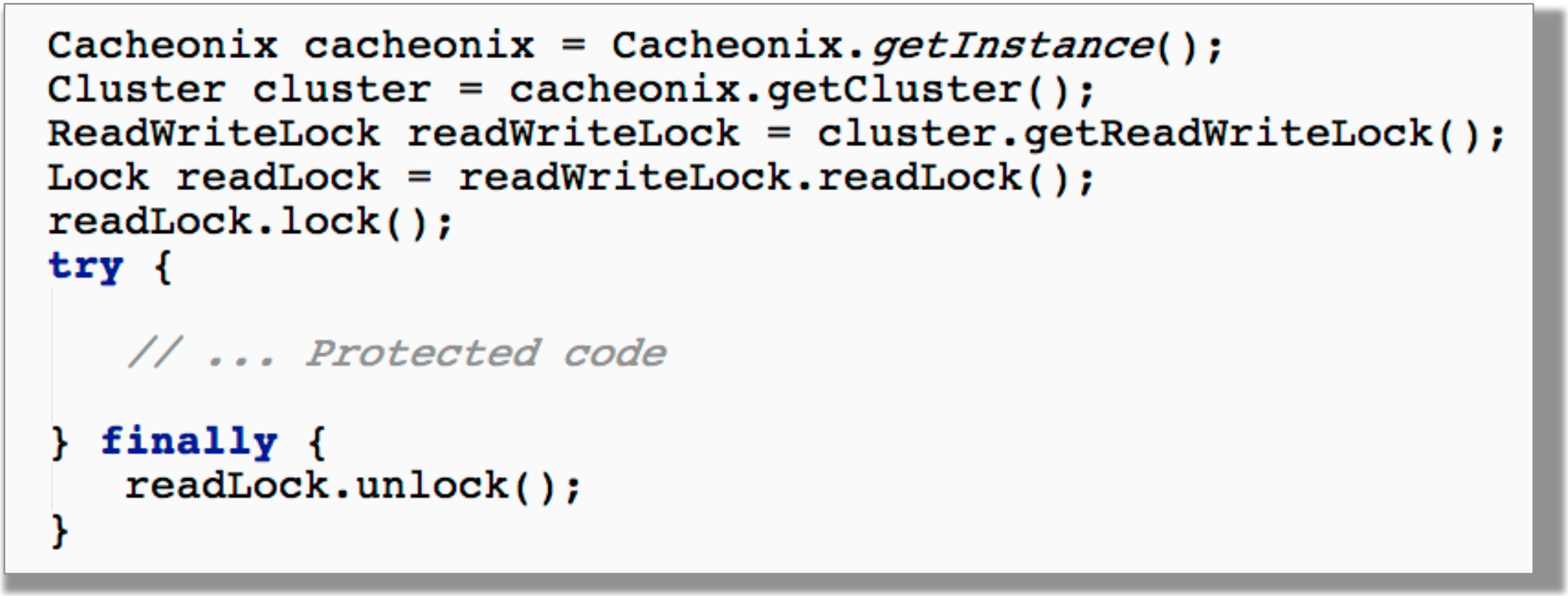
Cacheonix provides:

- Distributed ReadWriteLocks
- Distributed ConcurrentHashMap

Distributed ReadWriteLocks

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

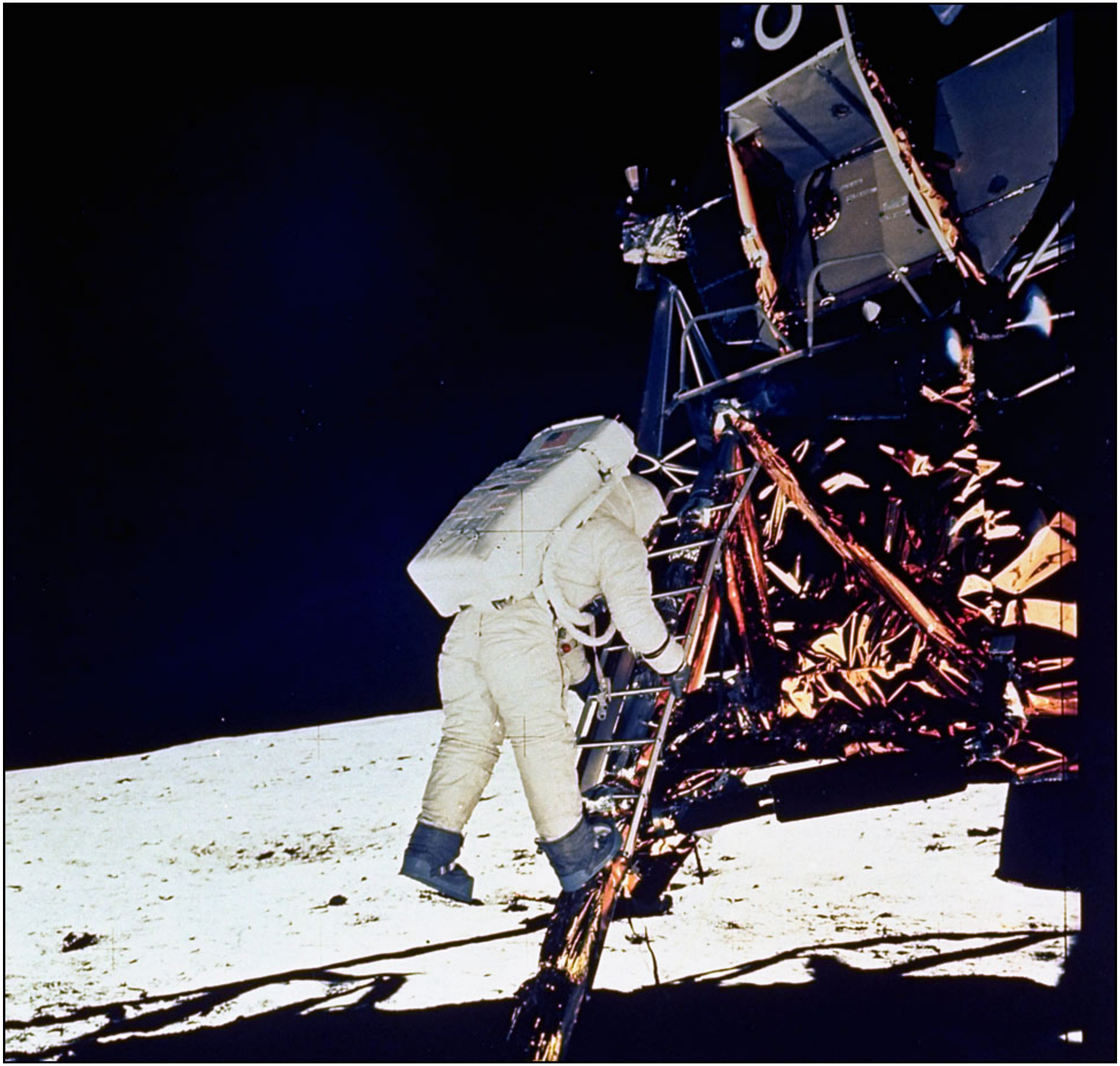
Distributed ReadWriteLocks



```java
Cacheonix cacheonix = Cacheonix.getInstance();
Cluster cluster = cacheonix.getCluster();
ReadWriteLock readWriteLock = cluster.getReadWriteLock();
Lock readLock = readWriteLock.readLock();
readLock.lock();
try {

   // ... Protected code

} finally {
   readLock.unlock();
}
```



Problem of Distributed State Sharing

- Threads need to access shared state in order to do useful work
- State sharing in a single JVM is trivial because of the local memory space
- Distributed state sharing is hard:
  - Servers communicate using the network
  - Distributed applications no longer share the memory space

Solution to Distributed State Sharing Problem

Cacheonix provides:

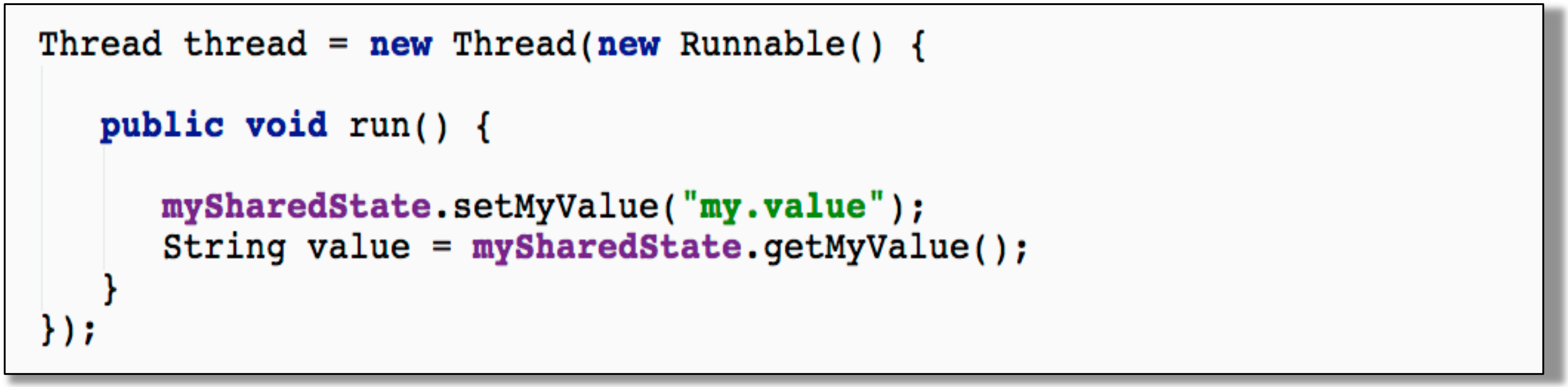
- Distributed HashMap

Distributed HashMap

- Strictly consistent
  - Guarantees that all servers immediately see the updates to the data
- Easy to use
  - java.util.Map interface
- Reliable
  - Retains the data as servers fail or join the cluster

Designing for Running in Cluster

- Store state shared between threads in Maps. Convert the code below:

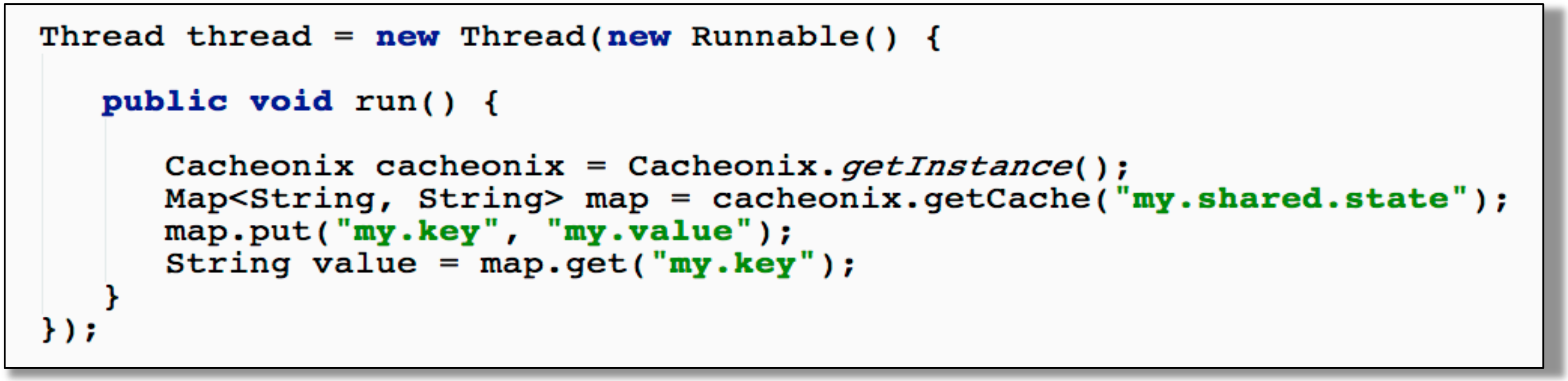


```
Thread thread = new Thread(new Runnable() {

   public void run() {

      mySharedState.setMyValue("my.value");
      String value = mySharedState.getMyValue();
   }
});
```

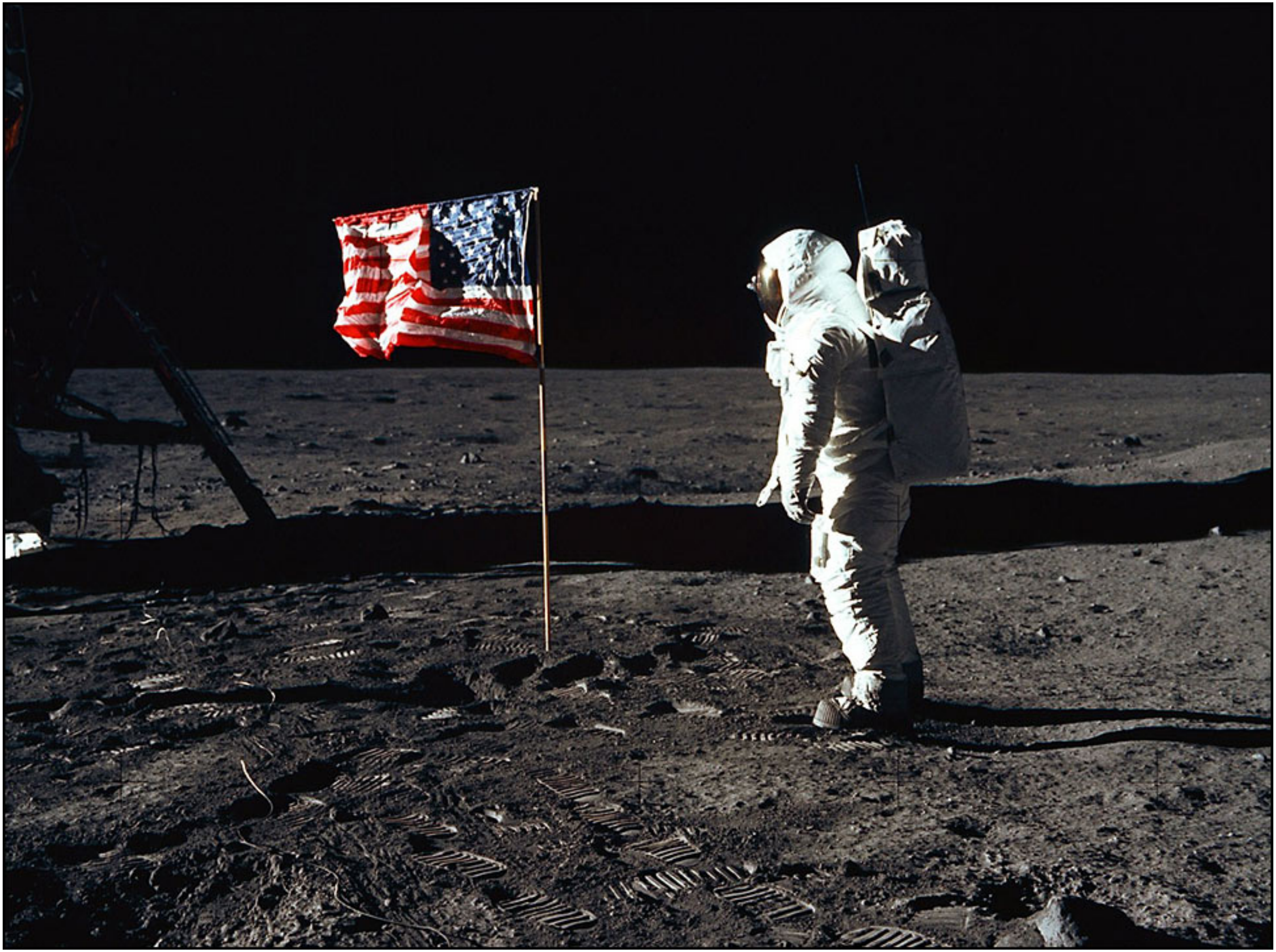
to:



```
Thread thread = new Thread(new Runnable() {

   public void run() {

      Cacheonix cacheonix = Cacheonix.getInstance();
      Map<String, String> map = cacheonix.getCache("my.shared.state");
      map.put("my.key", "my.value");
      String value = map.get("my.key");
   }
});
```



Failure Management

Distributed applications experience failures not seen by single-JVM applications 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

Cacheonix:

- Provides an ability to report a blocked cluster state for communicating it to the end user

- Detects change in cluster configuration (joining other cluster) and cancel consistent operations by throwing exceptions (lock()/unlock() and put()/get())

- Helps to prepare the application for dealing with such conditions, minimally gracefully reporting a error to the user.

Cluster Management and Data Distribution Protocol

Cacheonix protocol:

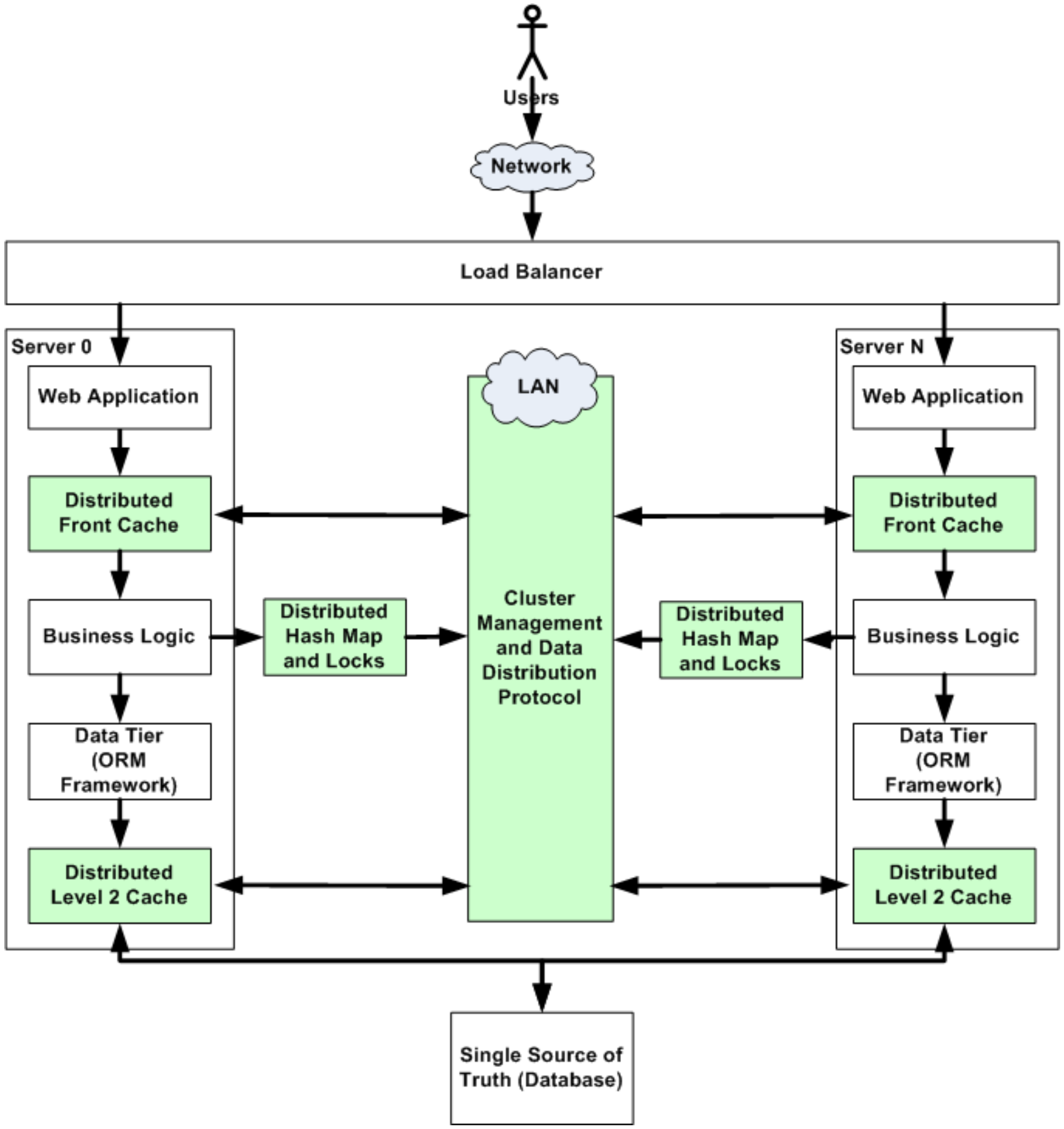
- Symmetric clustering
  - No single point of failure
- Wire-level
  - Highest possible speed
- Data distribution
  - Reliable
  - Strictly consistent

Cluster Management and Data Distribution Protocol

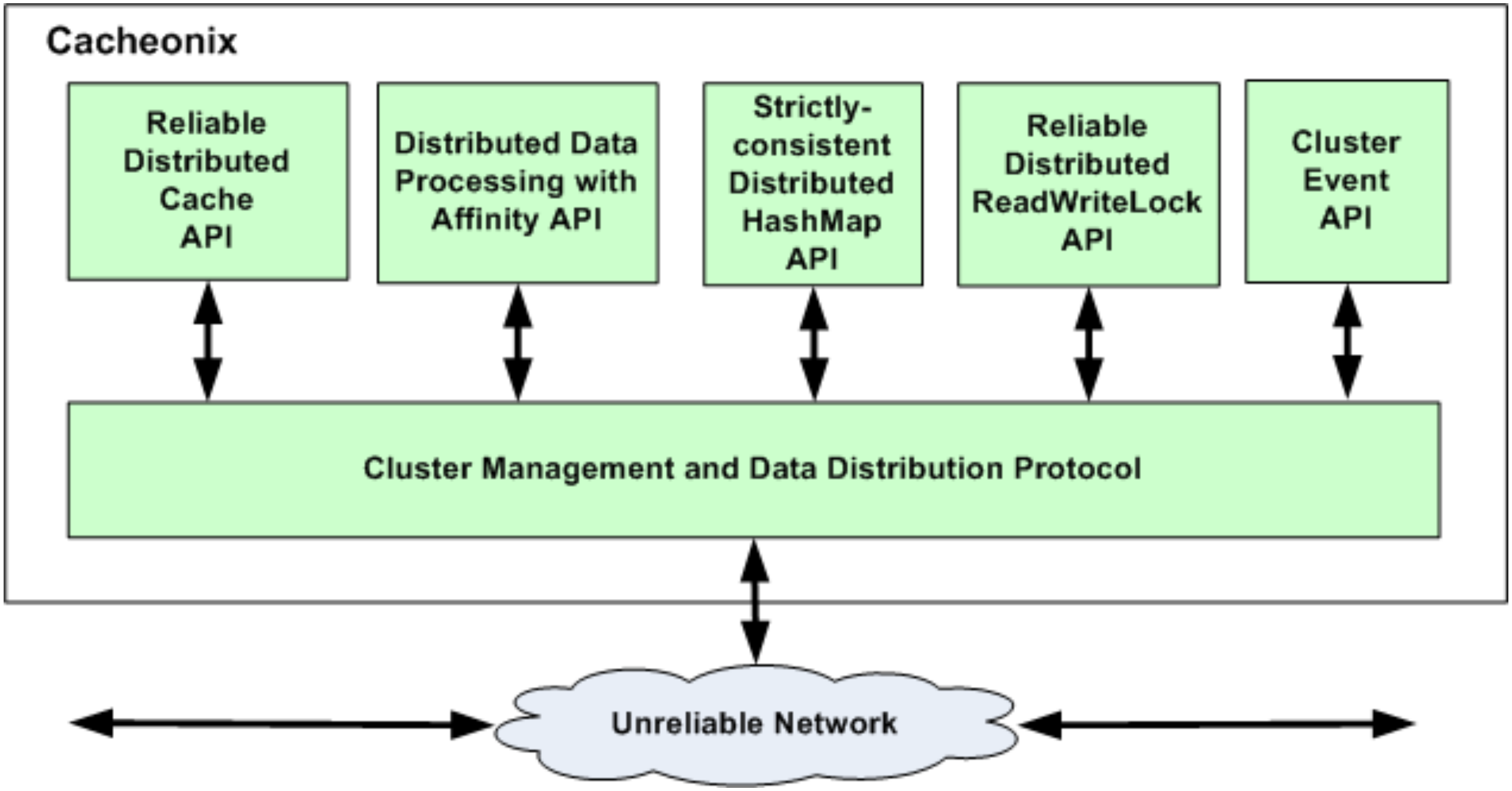
Cacheonix protocol enables:

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

Distributed Architecture

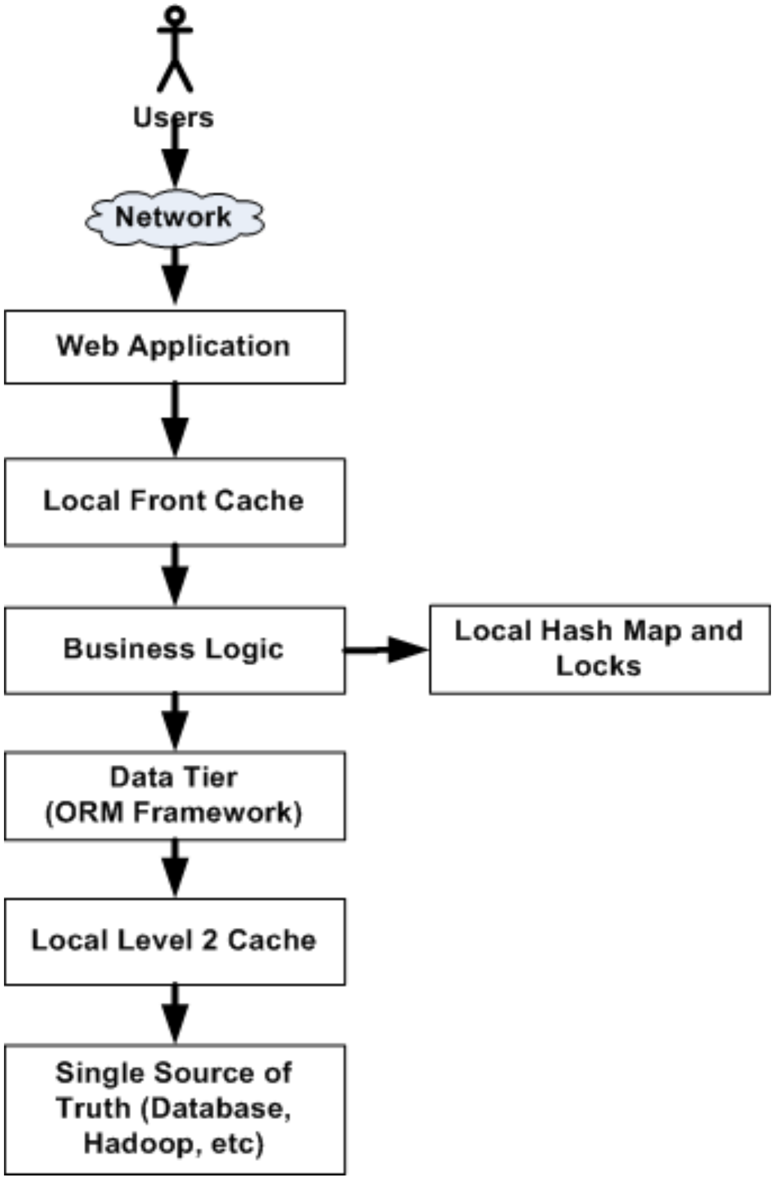

Users
Network
Load Balancer
Server 0
Web Application
Distributed Front Cache
Business Logic
Data Tier (ORM Framework)
Distributed Level 2 Cache
Distributed Hash Map and Locks
LAN
Cluster Management and Data Distribution Protocol
Distributed Hash Map and Locks
Server N
Web Application
Distributed Front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

Tying It All Together: Distributed Data Management Framework Cacheonix


Cacheonix
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

How about single-server applications?

Single-Server Architecture



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 can be limited by bottlenecks:
  - Databases
  - Expensive calculations

NASA
ARMSTRONG
NASA
COLLINS
NASA
ALDRIN

Scaling Vertically with Cacheonix

- Cacheonix provides a fast local cache
  - Eliminates database bottlenecks
  - Improves performance
  - Prepares for scaling in a cluster
- Use cases
  - Local front cache
  - Local query cache
  - Local L2 cache for Hibernate, MyBatis and DataNucleus

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.org

Cacheonix wiki:
wiki.cacheonix.org