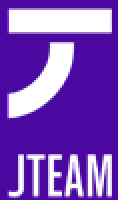
Infinispan

Open source data grids

Tim van Baarsen



Agenda

- Grid computing, datagrids & clouds
- Infinispan
- API
- Cache Modes + Demos
- CAP theorem
- Storage
- Roadmap
- Integration with other frameworks
- Competitors
- Q & A

Grid computing

- Example: SETI@Home
- Number of computers in a network to work on a single problem at the same time
- business problem requires a
  - great number of computer processing
  - process large amounts of data

What are data grids?

- grid for
- sharing and management
- large amounts of data
- in a distributed environment
- data is stored in memory
- achieve very high performance

Clouds vs. Datagrids

- Clouds:
  - on-demand of computing resources
  - current fashion is to use virtualization
- Datagrids:
  - More of a service
  - Sea of memory, spanning several servers
  - Data grids deployed on top of a cloud

When to use a datagrid?

- Database is becoming an unbearable bottleneck
- process tasks in parallel

When don't use a datagrid?

- Distributed messaging

What is Infinispan?

- In memory data grid
- Written in Java
- Open source
- Works on Java 6 compatible JVM
- 4.0.0 final
- LGPL licensed
- Compliant snapshot of JSR-107 specification (JCache)

Infinispan 4.0.0 ?

- Based on code JBoss Cache 3.x
- JBoss Cache not a dependency
- Future effort Infinispan, not JBoss Cache

Infinispan vs. JBoss Cache

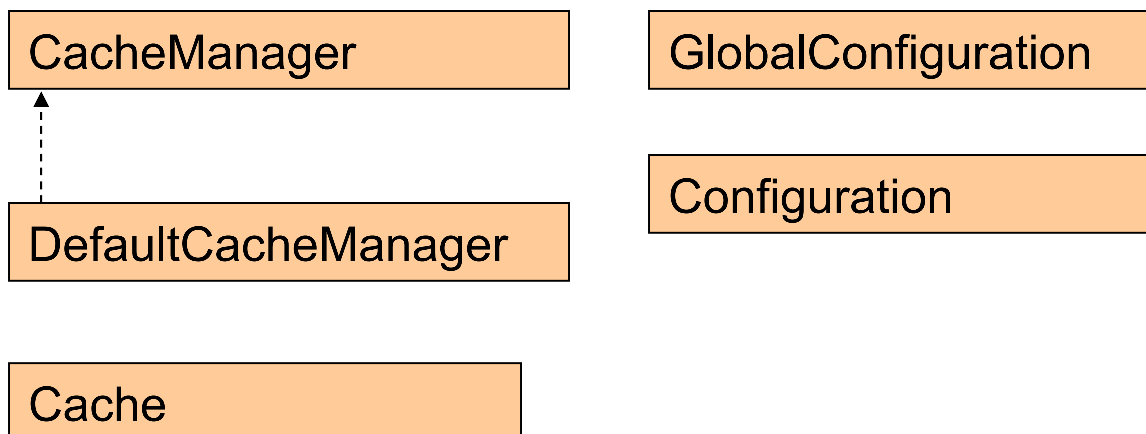
	Infinispan	JBoss Cache
Distributed cache	V	V
RESTful API for remote access	V	X
Management tooling	V	X
Scale to thousands of nodes	V	X

Infinispan provides

- Distributed cache capabilities
- Transaction support (JTA)
- Eviction algorithms to control memory usage
- Persisting state to configurable cache stores

Primary API

- org.infinispan.Cache
- org.infinispan.manager.CacheManager
- org.infinispan.config.Configuration
- org.infinispan.config.GlobalConfiguration



GlobalConfiguration

- Name of the custer
- JMX statistics settings
- Transport used for network communication
- Eviction schedule

```
GlobalConfiguration gc = new GlobalConfiguration();
gc.setClusterName("JteamCluster");
```

```
<infinispan>
  <global>
    <transport clusterName="JTeamDemoCluster"/>
  </global>
</infinispan>
```

GlobalConfiguration (2)

Preconfigured, cluster-aware configuration

```
GlobalConfiguration.getClusteredDefault();
```

Network transport

- JGroups as a network transport
- network transport peer-to-peer
- JGroups handles discovery

Configuration

- Cache uses a clustered mode
- Default cache mode is LOCAL
- Eviction strategy (NONE, FIFO, LRU)
- Number of owners

```
Configuration c = new Configuration();
c.setCacheMode(Configuration.CacheMode.REPL_ASYNC);
```

```
<infinispan>
  <default>
    <clustering mode="replication“>
      <async/>
    </clustering>
  </default >
</infinispan>
```

Create your cache

- Set up your CacheManager
- Create cache
- No name for cache = default cache

```
GlobalConfiguration gc = new GlobalConfiguration();
gc.setClusterName("JteamCluster");
Configuration c = new Configuration();
c.setCacheMode(Configuration.CacheMode.REPL_SYNC);

CacheManager cm = new DefaultCacheManager(gc, c);
Cache defaultCache = cm.getCache();
```

Create named cache

```
CacheManager cm = new DefaultCacheManager(gc, c);
Cache defaultCache = cm.getCache(“myFirstCache");
```

```
<infinispan>
   <namedCache name=“myFirstCache">
      <clustering mode="replication“>
         <async/>
      </clustering>
   </namedCache >
</infinispan>
```

Declaratively

cfg.xml

```
<infinispan>
  <global>
    <!-- not specifying details here will force default transport -->
    <transport />
  </global>
  <default>
    <clustering mode="replication" />
 </default>
</infinispan>
```

```
CacheManager cm = new DefaultCacheManager("cfg.xml");
Cache defaultCache = cm.getCache();
```

Use the cache

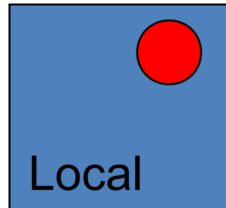
- Infinispan is ConcurrentHashMap
- Basic action on the cache:
  - start and stop
  - put
    - localCache.put(“key”, “value”);
  - get
    - localCache.get(“key”);
  - remove
    - localCache.remove(“key”);
  - size
    - localCache.size();

Modes

- **Non clustering**
  - **Local** mode
- **Clusting**
  - **Invalidated** mode
  - **Replicated** mode
  - **Distributed** mode

Local mode

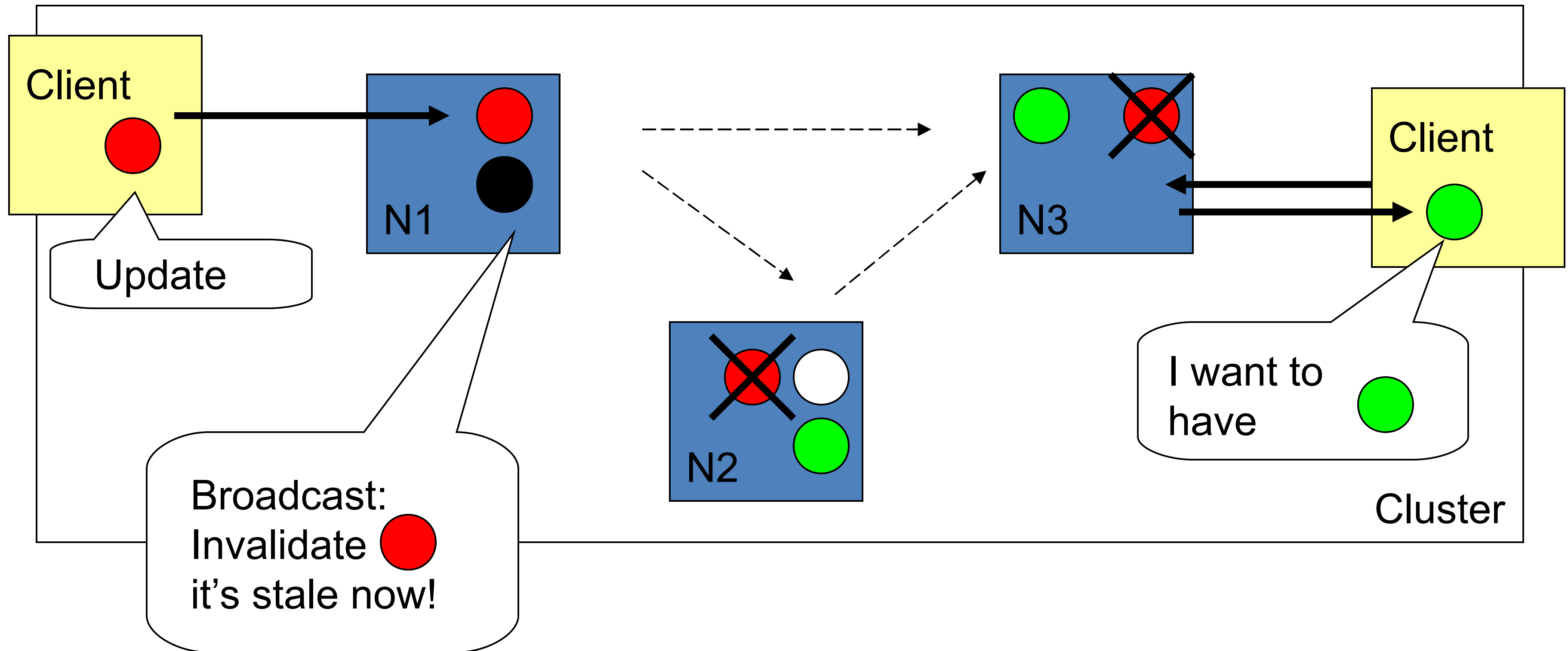
- “Traditional” caching
- Non-clustered caching
- No nodes



Invalidated cache mode

- set of local, standalone caches
- are aware of each other
- entry is changed entire grid is made aware
- nodes invalidate cache entry

Invalidated cache mode



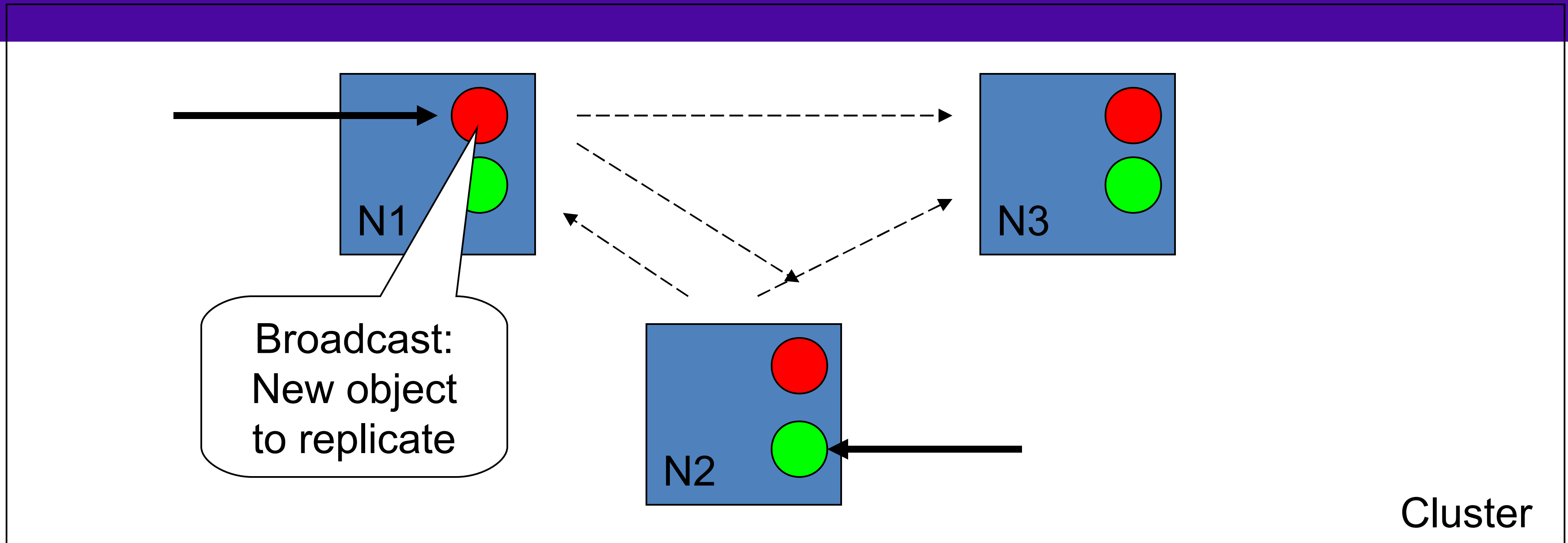
Pros & Cons

- Pros
  - traffic is minimized (invalidation messages)
  - very small compared to replicating updated data
  - look up modified data in a lazy manner
- Cons
  - not all data is local

Demo

- Invalidated mode

Replicated cache mode



- nodes are aware of each other
- able to interact maintain coherence of state
- state of all caches in the cluster identical

Replicated cache mode

Pros

- useful if the cluster size is small
- all of the state local and in-memory

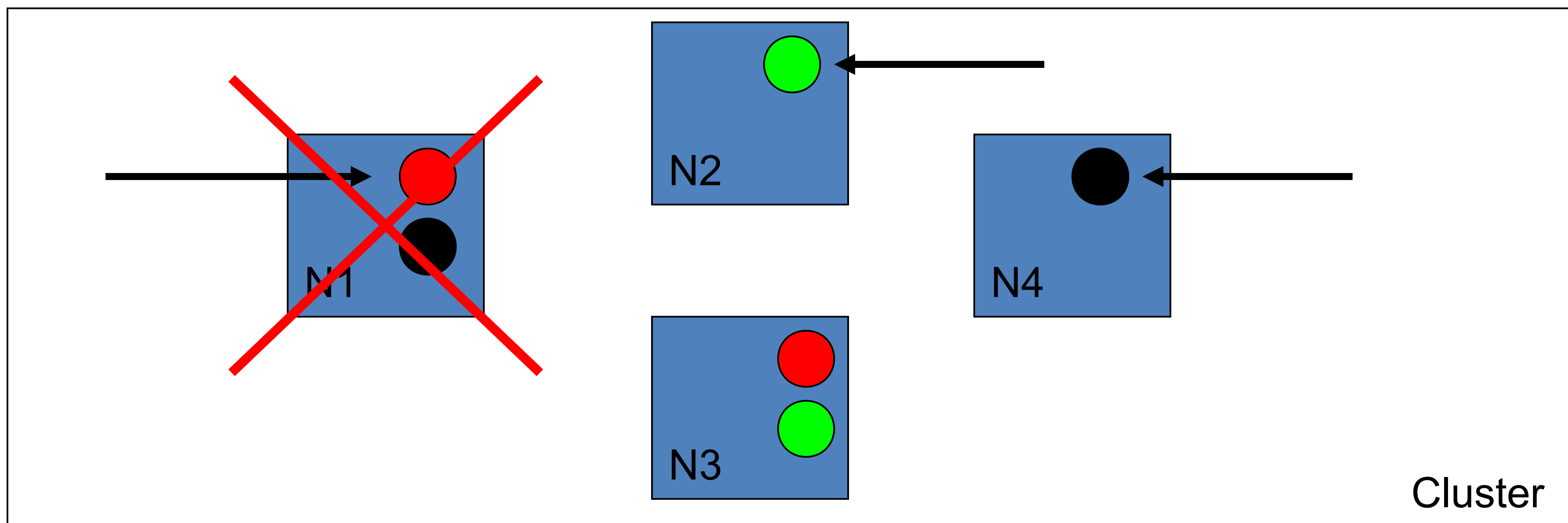
Cons

- does not scale in terms of memory
- limited to the heap of a single JVM

Demo

- Replicated mode

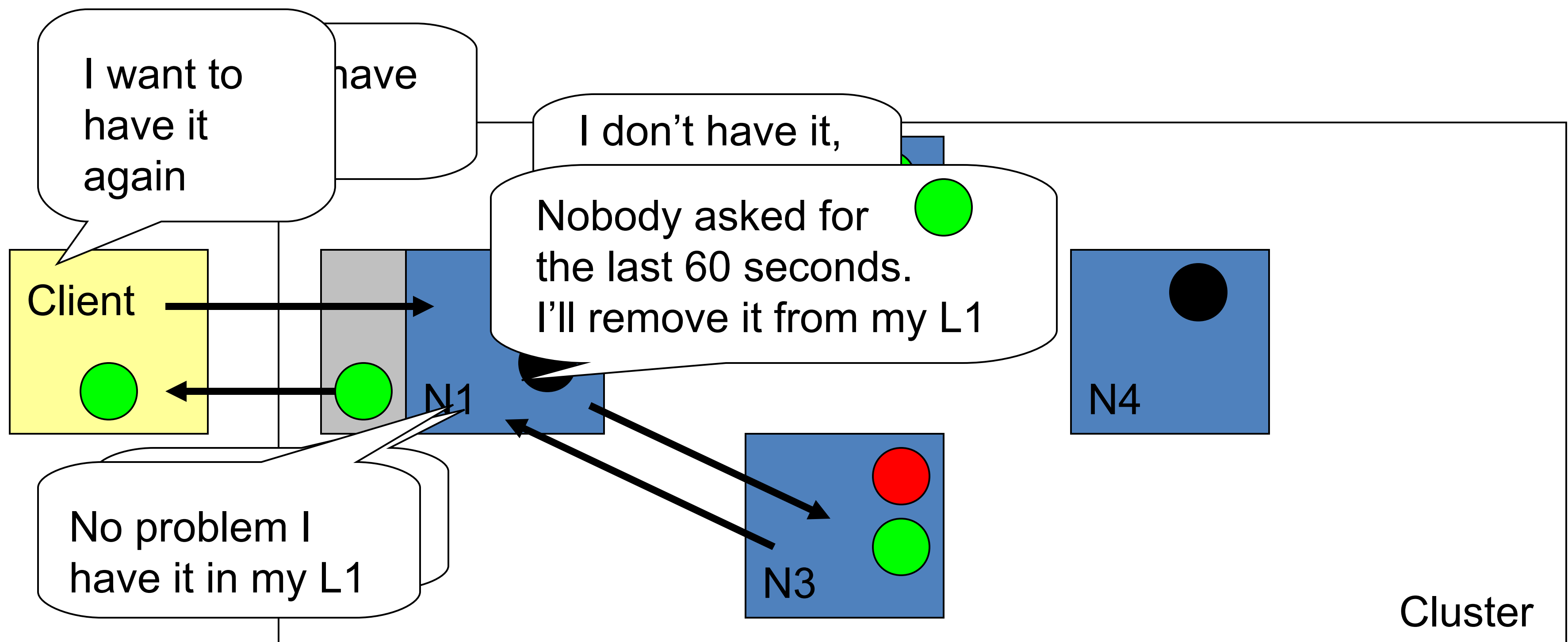
Distributed cache mode



- caches cluster together
- expose a large memory heap
- copies maintained for redundancy and fault tolerance
- dynamic rebalancing on the fly

L1 caching ("near caching")

- optional for distributed caching
- Minimize repeated lookups

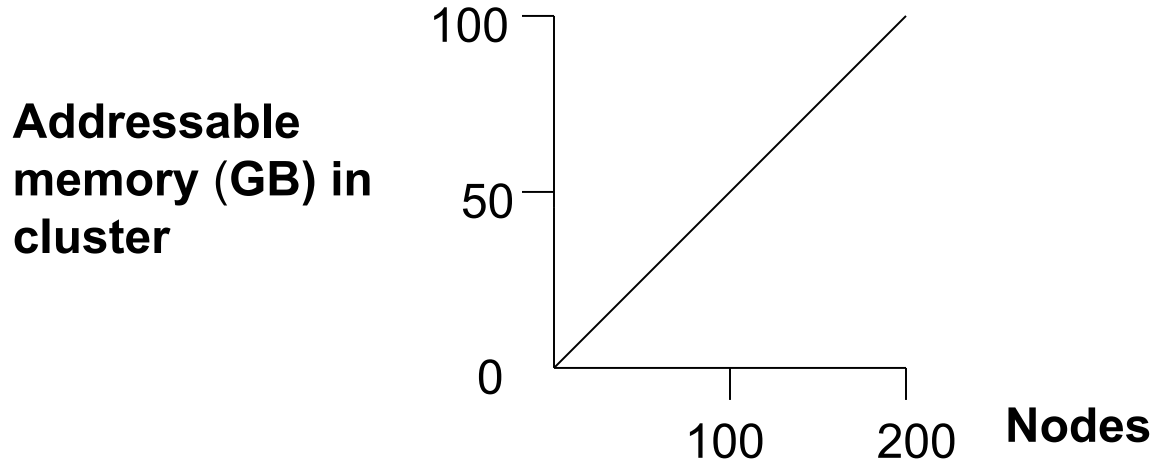


Pros & Cons

- Pros
  - no major limit to the size of the grid
  - scales linearly
  - resilience to server failure
  - fixed number of replicas of each entry
- Cons
  - not all data is local

Linear scaling example

- 100 nodes in the cluster
- heap size of 1GB each
- 1 copy (total 2 of each entry) (configured)
- (100 nodes x 1 GB) /2 = 50 GB
- (200 nodes x 1 GB) /2 = 100 GB



Demo

- Distributed mode

Sync vs. Async

- **Synchronous**
  - blocks the caller (e.g. on a put())
  - until modifications have been replicated successfully to (all) nodes in a cluster
  - Sure all modifications applied to other nodes

Sync vs. Async (2)

- **Asynchronous**
  - performs replication in the background
  - the put() returns immediately
  - faster (no acknowledgments other nodes)
  - Not sure about modification applied to other nodes
  - errors are written to a log
  - fire-and-forget

CAP Theorem

- **C**onsistency
- **A**vailability
- **P**artition-Tolerance

CAP Theorem

- mathematical proven
- simultaneously CAP at same time = Impossible
- only two of these three desirable properties can be achieved at the same time
- design tradeoffs must be made
- see: http://www.cs.berkeley.edu/~brewer/

Cache Stores

- Infinispan exposes a CacheStore interface
  - including JDBC CacheStores
  - filesystem-based CacheStores
  - Amazon S3 CacheStores

Eviction

- not run out of memory
- used together with a cache store
- eviction occurs on a *local* basis, not cluster-wide
- Offloads in-memory state to the CacheStore
  - NONE (default)
  - LRU (least-recently-used)
  - FIFO (first-in-first-out)

Expiration

- Expiration of items based on lifespan
- and (or) idle time
- expired entries are removed *globally*
- from memory, cache stores
- cluster-wide

```
cache.put((“key”, “value”, 1000, TimeUnit.MILLISECONDS);
```

Integrating with frameworks

- **Lucene** (on the roadmap)
  - Lucene Directory Provider
  - Distributed, in-memory store for Lucene indexes
- **Hibernate**
  - 3.5
  - Infinispan 2nd Level Cache Provider

Competitors

- Open source:
  - Hazlecast
- Closed source:
  - GigaSpaces
  - Oracle Coherence
  - Gemstone
  - Terracotta *
  - Cacheonix

Why Infinispan is cool

- **State-of-the-art core**
  - deal with great degree of concurrency
  - Most of the internals are essentially lock- and synchronization-free
- **Massive heap**
  - 100 nodes, 2 GB memory per node = 100 GB memory with 1 copy per data item
  - efficiently accessible from *anywhere* in the grid

Why Infinispan is cool(2)

- **Extreme scalability**
  - data is evenly distributed
  - no major limit to the size of the grid
  - peer-to-peer communication between nodes
- **Not Just for Java**
  - language-independent server module
  - memcached protocol
- **Support for Compute Grids**
- **Distributed Lucene Directory**

Roadmap

- **Server module**
  - memcached-compliant RESTful one
  - so non JMV languages can use Infinispan

Roadmap (2)

- **Support for Compute Grids**
  - pass a Runnable around the grid
  - push complex processing towards the server where data is local, and pull back results using a Future
  - map/reduce style paradigm

- **JPA like API**
  - to store entities in Inifinispan
  - easy migration from "traditional" data stores to Infinispan

Roadmap (3)

- **Query API**
  - Index cache state
  - Search entire grid
  - happen in parallel
  - map/reduce
  - receives and performs query on its locally cached state
  - and returns results

Conclusion

- Under heavy development
- Lack of documentation
- Small community
- JBoss
- Goal to compete with Coherence
- Promising technology
- www.infinispan.org

Q & A

CAP Theorem

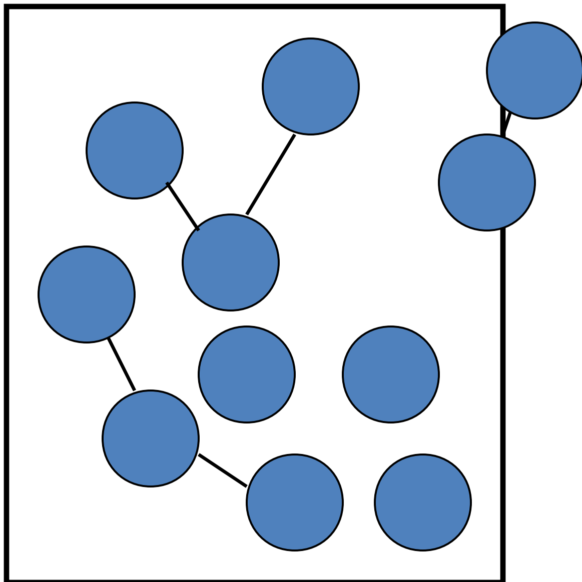
- **Real life example: Buying a last copy of a book online**
- Consistent
  - Add last available book in my basket
  - Another shopper will add the same book to his basket
  - If both customers can continue there is a lack of consistency
  - Maybe not a big issue in this case
  - What about trades on financial exchange?
  - **Changes are immediately visible or eventually**

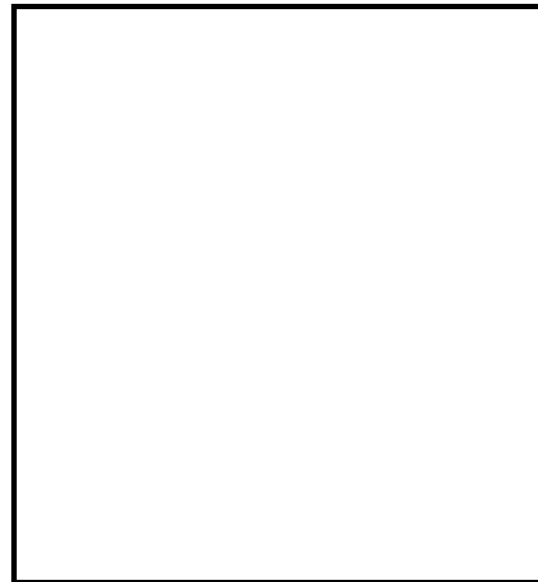
CAP Theorem

- **A**vailability
  - I want to buy the book and not a browser error
  - You will face availability when you need it most
  - Site down because they are thousand of customers
  - **How much is your system relying on other systems?**

CAP Theorem

- **P**artition-Tolerant
  - Data is spread around different machines, Risk of data partition
  - Happens when nodes cannot communicate?
  - Can is still order the book?





- Invalidated cache mode
  - Sync = C OK, A NOT, P OK
  - Async = C NOT, A NOT, P OK
- Replicated cache mode
  - Sync = C OK, A OK, P NOT
  - Async = C NOT, A OK, P NOT
- Distributed
  - Sync = C OK, A NOT, P OK
  - Async = C NOT, A NOT, P OK