Terracotta versus JBossCache

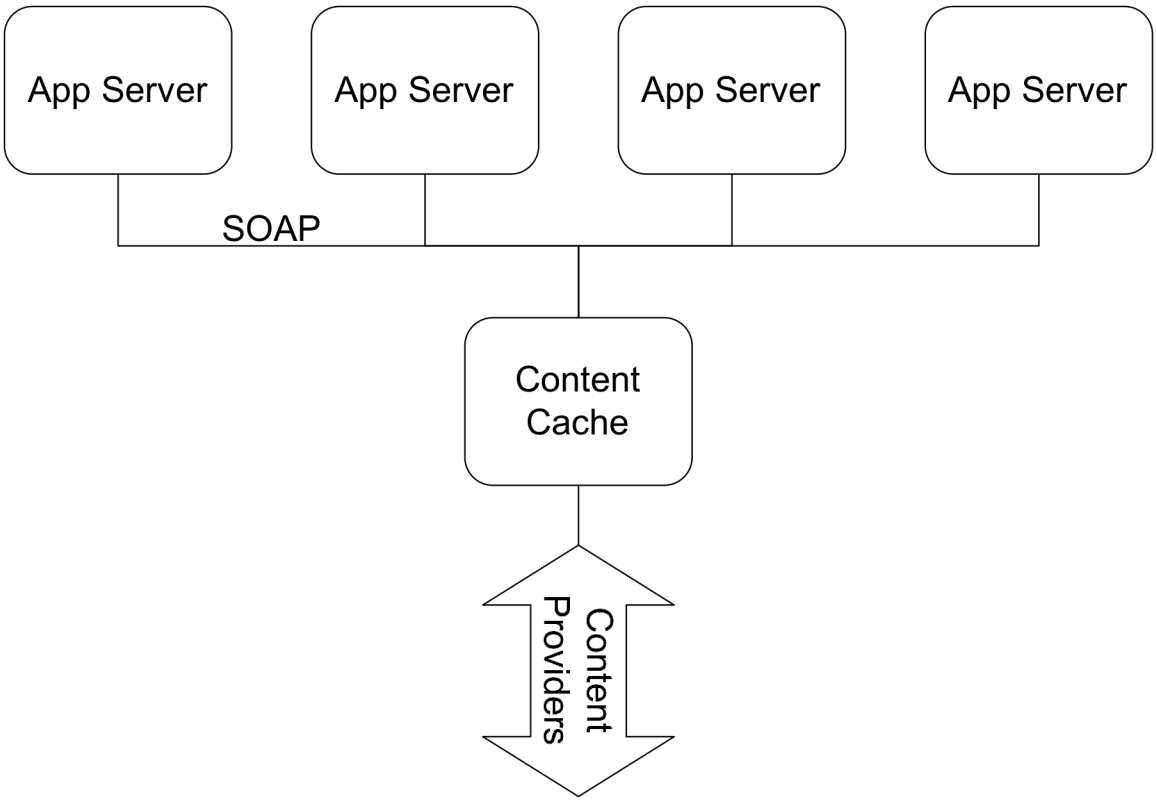
Introduction

Application scalability is sometimes considered a black art. Terracotta seeks to help our community by moving clustering logic out of source code and into configuration. We believe this declarative approach will help, in many cases, to reduce the degree of "black art" in clustering, give more developers and operators the control they need, and make applications more resilient and more scalable.

Many discussions have been offered regarding the value of configuration-driven clustering. With Terracotta, this promise of clustering simplicity comes with assurances of the highest levels of performance. In this article, we document an experience in migrating from JBossCache and the associated programming model to Terracotta's runtime approach. We will see how Terracotta can make the migration path simple by adapting itself to JBoss's API, and we will see how the core DSO technology provides for two orders of magnitude greater performance than JBoss, with near linear scalability.

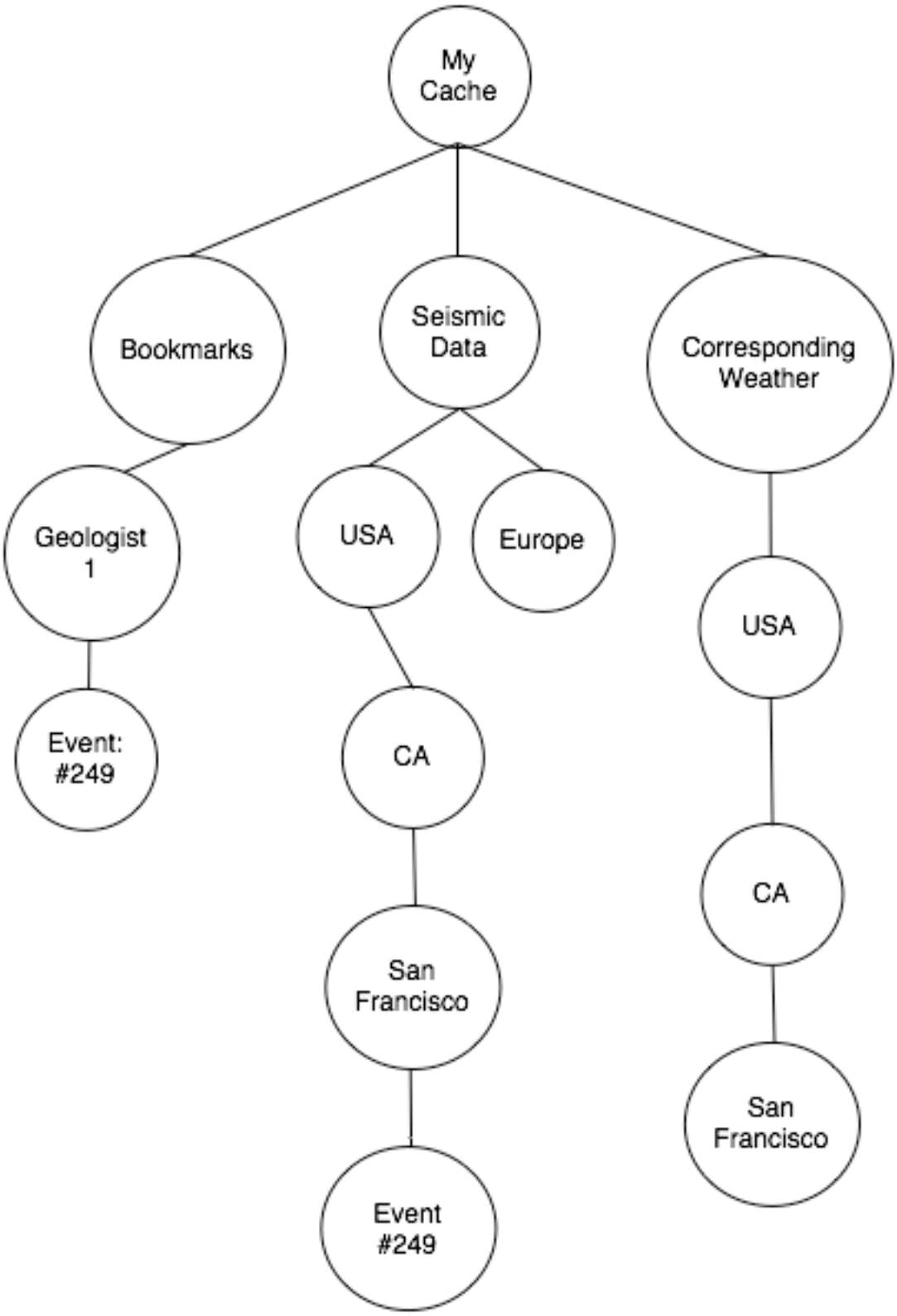
The Use Case

The use case has occurred multiple times where third party content gets served via a web portal. A read-only content aggregator is used to ensure quality of service for end users. The approach in several cases, Terracotta has seen, is to create a web service hosted on one app server instance that would query the Internet or Extranet-based content providers asynchronously with end user traffic, thus limiting end users to only being able to access the aggregator itself.



The benefits of this approach are in the ability to scale the aggregator separately of the application tier, as well as the ability to serve Internet-based or Extranet-based content at the same speed as the application's local content; end users are indifferent and unaware that content is outsourced.

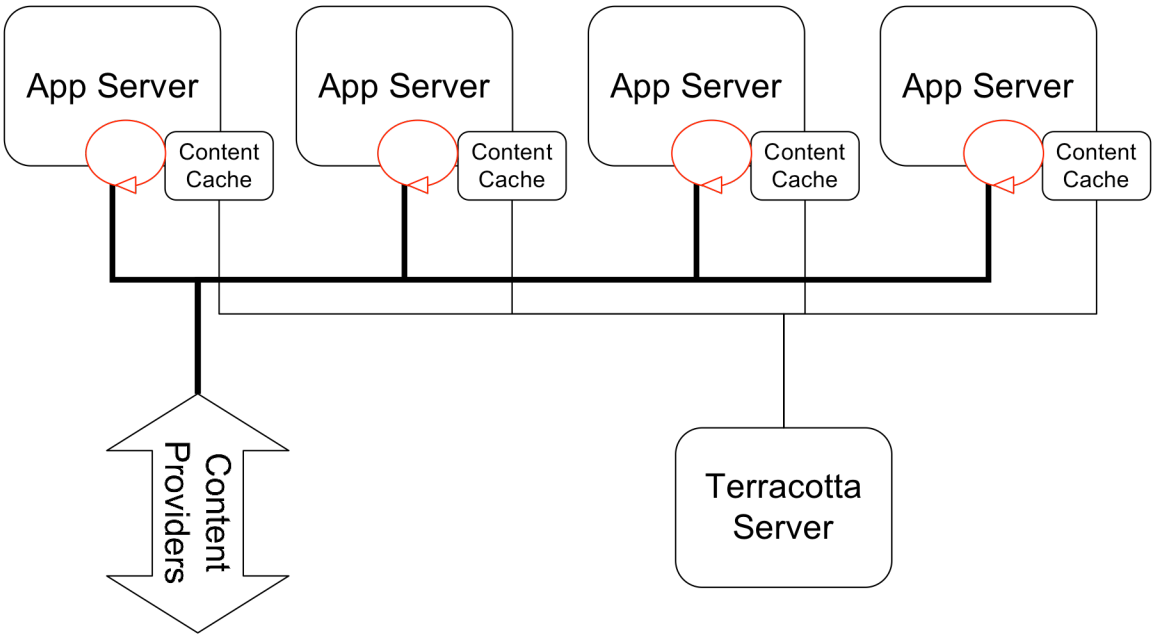
In three recent use cases, JBoss was the container of preference because it offered a web service API, the app server itself, and built-in caching API's with configurable clustering that can be turned on as scale demands; so it appeared to be a one-stop shop for this use case. JBoss's app server and API's worked well for building the application. More importantly, the caching API was helpful in that JBoss Treecache helped developers express the content problem in a very natural way. For example, if a content aggregator contains seismic information per region and custom annotations on that region per geologist observing this data, surely each type of cache data has different scope from per-user to per-geography and each will expire on its own schedule. Expressing each type of cache under an expiration policy and per source makes cache management simple. The following diagram depicts an example of the power of the Treecache:



Under such a design, one could easily expire all seismic data in the USA every 15 minutes by deleting the USA node from the tree. And, theoretically, scaling the architecture is just a matter of sharing or partitioning the tree across many caching servers.

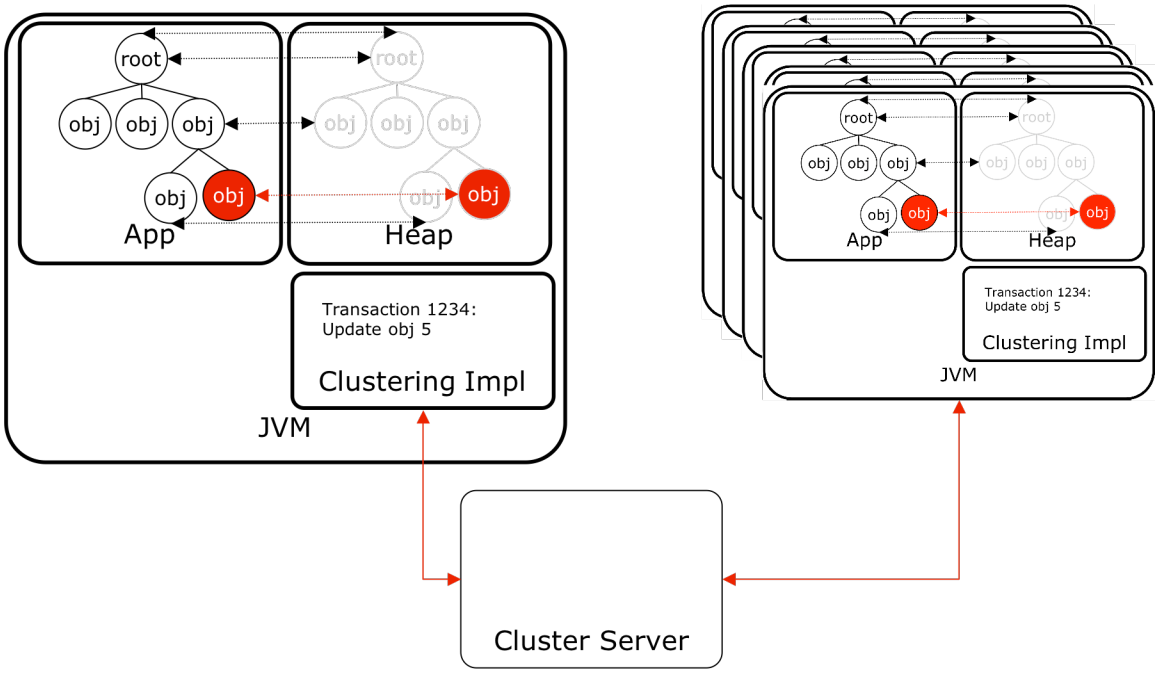
The Pain

In practice, however, JBosscache's Treecache project worked on one node but when it was time to scale, the customers found the aggregation server degraded from more than one hundred requests per second to about ten requests per second per aggregation server. This 90% penalty for clustering was unacceptable. The cost of making a SOAP call to the server(s) responsible for aggregation was so high that the value of the cache came into question. Interestingly, no matter how slow the cache, it is always better than risking the Internet / Extranet call to get the content straight from the source while servicing the end user's HTTP request. So, at the end of the experiments, it became clear that availability was reason enough to keep the cache. The requirements were to run no more than 4 app-servers at the beginning with 1 or more app-servers also doubling up as an "aggregation" server and then gradually add more nodes as application usage increases. So, why not stay with JBosscache, go to production, and tune for performance throughout the year? The answer was, Terracotta provides availability without compromising scale, and without having to delay the solution until the problem hits a production application. In fact, all three of these customers started by conflating small scale with the cost of building efficient systems, believing that one must trade of scalability for time to market when in fact Terracotta helped them decouple system scale from architecture complexity. While scalability and availability is a concern at 100 nodes, Terracotta's runtime clustering lowered the cost of tuning such that it could be started early in the application's lifecycle.



For this content caching use case, Terracotta scaled linearly, presented itself via a POJO Treemap interface as a bolt-on replacement for JBosscache, and provided higher availability than JBoss could. The availability advantage comes from the fact that Terracotta, as shown in the diagram below, runs both in process and outside your Java process.

The app servers can be stopped. The caching SOAP service disappears from the architecture altogether, and Terracotta's out-of-process Server can be stopped and the cache will not be lost without the need to implement a CacheLoader on top of a proprietary API. This afforded the operations team a 100% available cache without the need to maintain special application startup procedures, re-warm the cache, or suffer a recovery period when the application goes down and back up.



Let us now dive into the key features of Terracotta's DSO clustering technology that provided these customers with scale and improved availability. Note that a detailed discussion of Terracotta's design is outside the scope of this article, but is available on our website at http://docs.terracotta.org/confluence/display/docs1/Terracotta+Scalability. Terracotta DSO works to keep objects consistent across JVM boundaries as depicted in the following diagram:

Note that the image depicts the system's ability to keep the JVM on the left in sync with the others. This implies a level of integration to the JVM unlike any other in that Terracotta does not require objects to implement Java's serialization interface, nor does Terracotta invoke Java serialization at any time. Such integration preserves object identity and allows applications and development frameworks to remain true POJO. This proves vital to our use case, as we will explain later.

The following key features each contribute to the availability and performance gains the customers saw:

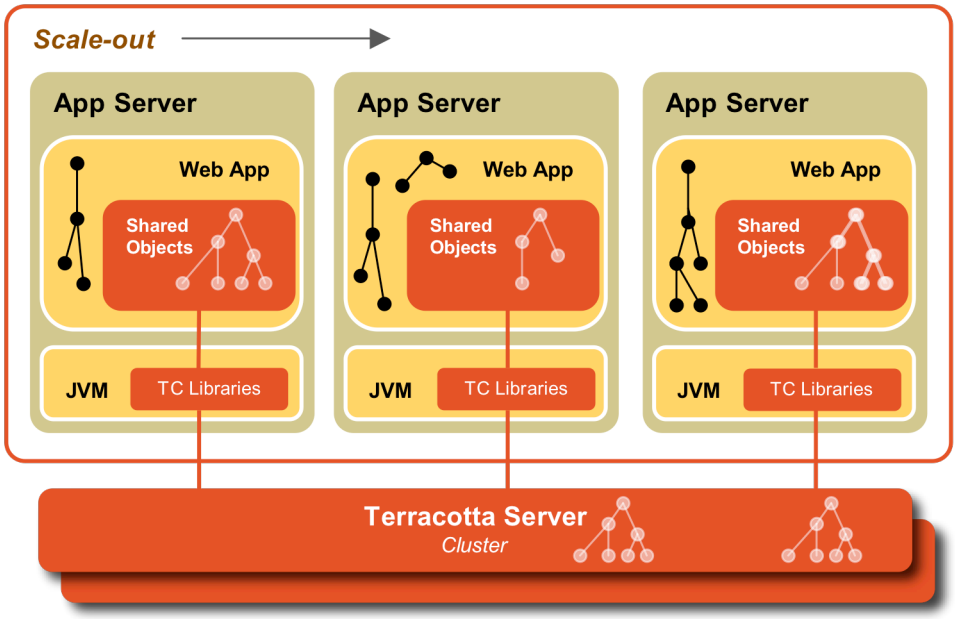
- Two-tier caching with LRU / LFU optimizations
- Terracotta's fine-grained updates allowing clustering to push only the object data that changes
- Page in and out objects on demand
- Linear scale in the server

Two tier caching

Terracotta DSO is designed under the same concepts as symmetric multiprocessing. If Application Servers are our CPU's and the caching server is a slow I/O subsystem, we need to put read-thru / write-thru L1 caches into the App server (CPU, by analogy), and

the caching server needs to be a consistent, cross-app server L2 cache. Terracotta injects itself into your JVM, and introduces L1 caches automatically.

The following diagram depicts Terracotta's architecture. The boxes labeled "TC Libraries" represent our JVM plug-in. The plug-in contains LRU / LFU caching logic inside. When depicted as per the diagram, one can visualize the app server boxes as CPUs with the TC Libraries acting as L1 caches. The Terracotta Server cluster is then the highly-available L2 keeping shared access to memory in synch. With two tier caching as opposed to a single JBoss caching service, Terracotta helps the application avoid unnecessary calls to your caching server by remembering the last answer the caching server provided. An important note here is that under Terracotta, as opposed to JBoss, this two tier caching technique was used in place of SOAP so as to avoid the marshalling / un-marshalling cost of calling the caching service. The cache became a pure POJO cache, with the behavior of a centralized caching server but with the performance of a local-only cache.



Fine Grained Updates

Terracotta plugs in to your existing JVM and weaves itself into the calls to read and write to heap (heap being the Java process's memory in the host operating system). This means that regardless of object graph size or complexity, Terracotta will see the details of a memory update. When memory is read by a Java thread (such as looking up a cache entry in the data tree) that memory can come from another JVM over the network just in time to satisfy the thread's need to access a particular object. When memory is written by a Java thread (such as flushing the cache or updating a cache entry in the aggregation server) that write can be pushed to other JVMs in the application server cluster as needed.

Performance comes from the fact that small changes to cache represent small updates to the network. For this caching use case there are 2 types of updates to cache and both are always small. The first is a cache flush where a whole sub-graph of the treemap is deleted once stale. This is a simple reference update (this.child = null in pseudo code), and represents four bytes of data (the reference ID) changing on heap (assuming we can ignore the garbage that eventually gets collected). The second type of update is a cache miss, leading to the addition of a new cache node to the tree. The size of this update is dependent on the type of cache data being populated but typically ranges from 250 – 1KBytes.

While JBoss's Treecache pushes only the nodes in the tree that change, Terracotta can push fine-grained tree updates at a field level versus a node level. This can add significant performance in data models where a cache node is much larger than any one field in that node and individual fields can be updated without deleting or overwriting the entire tree node. Terracotta does this with is core JVM plug-in and without a purpose built treecache technology.

Page In and Out

The ability to connect into the JVM's very access to its own heap allows Terracotta to page data into memory just in time to satisfy a cache lookup. This adds another performance benefit in that the cache can be sparsely populated and data will be pulled in as needed but not before. Unlike JBosscache's peer-to-peer approach, Terracotta's server (or server cluster as the need may arise) is the only server that must keep track of all data in the cache. Meanwhile, application server instances can remember the pieces of cache needed to satisfy end user traffic at any one period in time. Memory footprint is smaller so more heap remains for end user session and request processing. More importantly, the application servers do not need to hear about all cache updates since they do not store the entire cache in memory; there is no data to keep consistent. This approach minimizes network chatter and assures close-to-linear scalability.

Linear Scale in the Server

As a result of app server-internal caching, fine-grained updates, and the ability to page in only the subset of cache needed to service requests, Terracotta will outperform JBosscache, whether using Treecache or Pojocache. Both technologies are clustered using JGroups and, while functionally viable, the scalability of keeping all servers consistent via copying all data to all caching nodes will eventually bottleneck on the network. Terracotta does not suffer from this bottleneck because the caching servers can, in a just-in-time fashion, update only the parts of cache that become stale in each application server's local copy over time. No cluster-wide conversations need occur on cache update. This all but eliminates the network bottleneck except for cache data that is resident in all nodes. So, the server adds another level of scalability by forcing the app

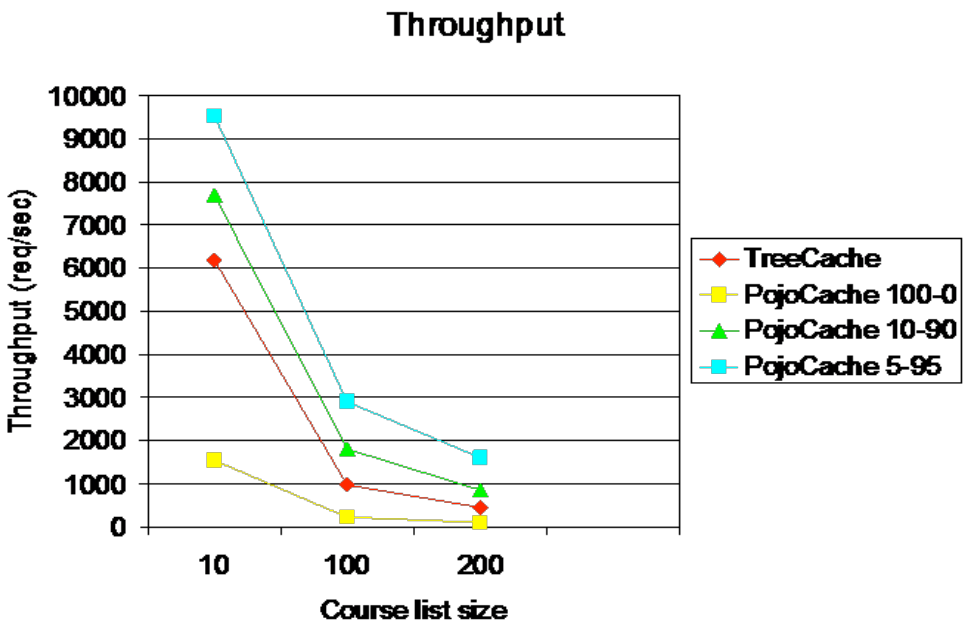
servers to pull changes on-read instead of pushing the changes on write. This means the server lazily updates clients as they read stale data and thus was able to avoid the network bottleneck in **all** cases where a Terracotta customer has migrated from JBoss Cache.

Results, Graphs, and Analysis

Terracotta has prepared a harness with which you can reproduce the test results that our customers used to convince themselves of the performance benefits of our approach. (Keep in mind the development benefit of being able to eliminate the SOAP calls and remaining pure POJO while simultaneously introducing multi-tier caching strategies.) There are two approaches we have undertaken. You can either use JBoss's own performance test or use our kit to insert Terracotta into your application. We have bundled both to make testing easier. This analysis focuses on JBoss's own test harness. The harness is based on JBoss's internal performance test harness made available on their website at http://wiki.jboss.org/wiki/Wiki.jsp?page=WhatShouldWeExpectOfThePojoCachePerformance

JBoss's site does not document the size or configuration of the computer cluster for this test, so Terracotta chose to first reproduce the test using JBosscache to form a local hardware baseline for our own computer cluster. The first thing we found is that we could not reproduce the performance JBoss had. But this is not too important. Because Terracotta DSO was able to be introduced to JBoss's harness without rewriting it. So we could test the same code base clustered by JBosscache and DSO side by side. The conclusion was documented in a webinar we hosted recently.

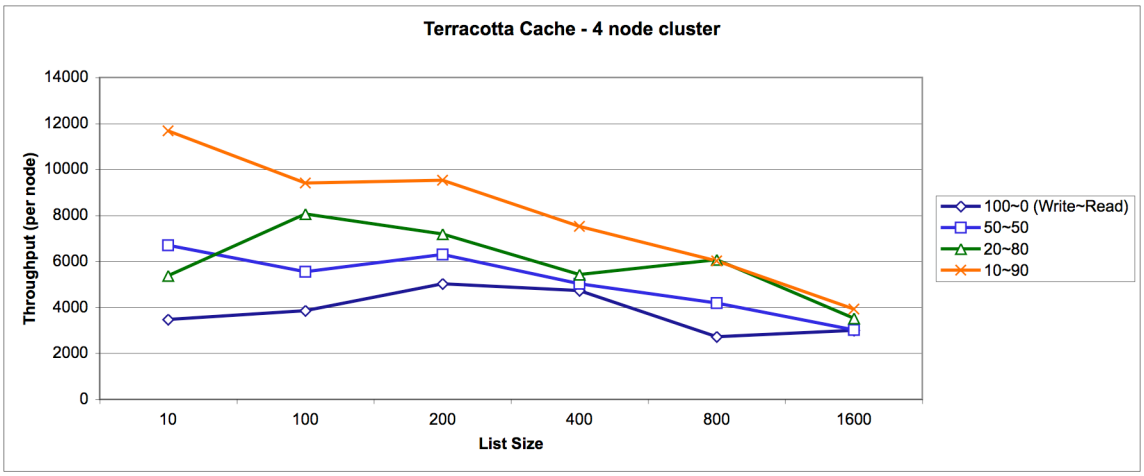
According to JBoss's own results in the graph directly below, Treecache produced on the order of 500 requests per second from a four-node cluster.



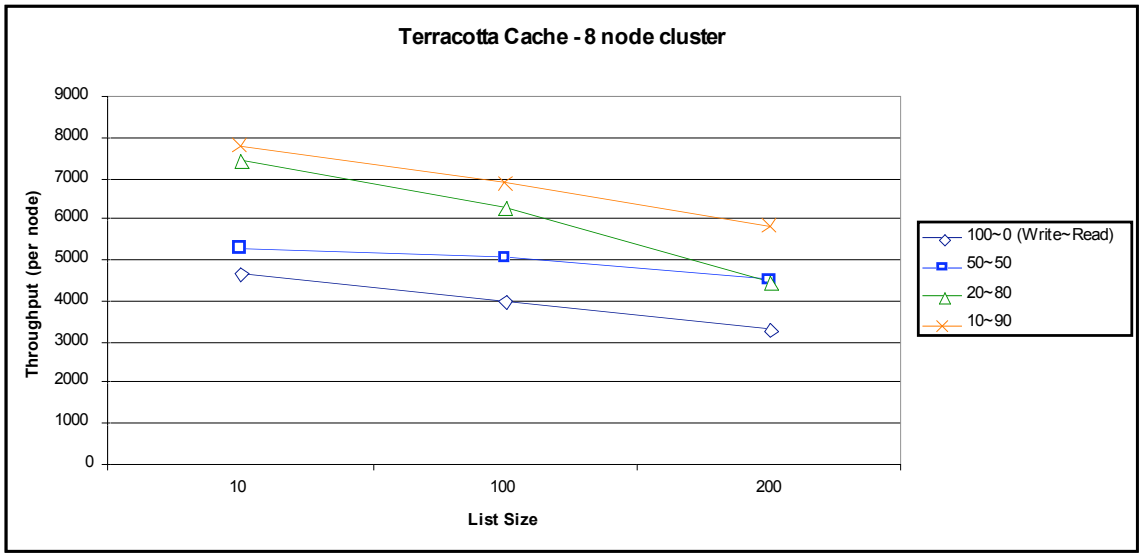
(All data for four nodes. Source of graph is JBoss.com)

Treecache and Pojocache both decreased significantly in throughput on 2 or more nodes when caches had more than 100 entries and/or had a significant write mix. The graph above shows a 4-node JBoss cluster reading and writing a cache of 10 to 200 objects in scale. The green and blue graphs labeled 10-90 and 5-95 represent a 10% and 5% write mix, respectively. 100-0 represents a write-only scenario that is irrelevant to the caching use case. Terracotta produced more than 2000 requests per second totaling 8000 per second for JBoss's 4-node 200 list test.

Terracotta engineers gathered similar data for our clustering implementation but we tested larger datasets, higher read / write mix, and more nodes. The 4-node cluster produces about five times the performance of JBoss's own performance numbers from the graph above at low scale like 10 cache objects. And the cluster never drops below 4000 transactions per second, even at 1600 tree nodes in cache.



At eight caching nodes, the cache still scales well, whereas customer testing reports that JBoss's JGroups internals no longer function and the JBoss version of the test crashes:



As one can see, the throughput of a single app server using the standard Java Treemap class and backed by Terracotta far outpaces the same architecture built on JBosscache. When adding SOAP service calls to the JBoss architecture as opposed to Terracotta's pure POJO approach JBoss's system dropped from hundreds or thousands of request per second to tens. This observation in this use case led customers to conclude that Terracotta is 100 **times** faster than JBoss. Terracotta's pure POJO approach demonstrated throughput 100 times that of the alternative approach where SOAP calls are added to the JBoss architecture and the Terracotta approach scales to 16 nodes without significant slow down.

Also, recall that with Terracotta, the application servers, the caching servers, and the Terracotta server can all be restarted at will, since Terracotta is persisting the cache to disk at this level of throughput. So availability is significantly higher than JBoss's in-memory clustering, whose caches are lost on restart or crash.

Reproducing the Tests

Several tests were run. You can download the tests from wiki.jboss.org and try to hook up the system yourself. You can also download a pre-built kit from Terracotta to test and / or migrate your application from JBoss to Terracotta DSO:

http://www.terracotta.org/confluence/display/labs/Terracotta+Cache

The basic approach to migration is to remove JBoss cache initialization and replace it with org.tc.ITerracottaTreeCache instead. The interface supports most of the methods in the original JBoss Cache implementation:

```
public interface ITerracottaTreeCache {


Set getKeys(String fqn) throws CacheException;
Set getKeys(Fqn fqn) throws CacheException;

 Object get(String fqn, Object key) throws CacheException;

 Object get(Fqn fqn, Object key) throws CacheException;

 ITerracottaTreeCache get(String fqn) throws CacheException;


 ITerracottaTreeCache get(Fqn fqn) throws CacheException;

 Fqn getFqn() throws CacheException;

 public String getStringFqn() throws CacheException;

 public Map getData(String fqn) throws CacheException;

 public Map getData(Fqn fqn) throws CacheException;

 boolean exists(String fqn) throws CacheException;

 boolean exists(Fqn fqn) throws CacheException;

 boolean exists(String fqn, Object key) throws CacheException;

 boolean exists(Fqn fqn, Object key) throws CacheException;

 void put(String fqn, Map data) throws CacheException;

 void put(Fqn fqn, Map data) throws CacheException;
```

```
Object put(String fqn, Object key, Object value) throws
CacheException;

Object put(Fqn fqn, Object key, Object value) throws
CacheException;

void remove(String fqn) throws CacheException;

void remove(Fqn fqn) throws CacheException;

Object remove(String fqn, Object key) throws CacheException;

Object remove(Fqn fqn, Object key) throws CacheException;

void removeData(String fqn) throws CacheException;

void removeData(Fqn fqn) throws CacheException;

String print(String fqn);

String print(Fqn fqn);

Set getChildrenNames(String fqn) throws CacheException;

Set getChildrenNames(Fqn fqn) throws CacheException;

String toString();

String printDetails();

int getNumberOfNodes();
}
```

The Terracotta version of the cache is implemented in the class org.tc.TerracottaTreeCache.java. The implementation supports many backing stores such as hashmap, synchronizedHashMap, and ConcurrentHashMap. The cache keys implement a comparator (so as to allow applications to reference a subsection of the TreeMap), while maintaining Jboss TreeCache APIs – the user then replaces TreeCache transparently by simply instantiating the Terracotta cache (instead of JbossTreeCache) , since most of the APIs are preserved.

Conclusion

Terracotta's architecture helped deliver higher application availability than JBosscache by keeping cache data available across JVM crashes and restarts. The caching service is also now capable of scaling well past 16 nodes at linear scales far beyond all three customers' initial requirements (60,000 requests per second from 16 app servers as opposed to a planned 10,000 requests per second projected from 80 app servers at 100 requests each per second). Thus the resulting cluster will be smaller over time and save the customers money. There is a second savings because the application architecture is

now POJO as opposed to SOAP-based, and scales with less complexity, thus freeing developers from what was a planned six month development and scalability project, to in this case, an effort measured in weeks.

The architecture is more elegant in that it incorporates a two-tier caching system. It also scales through on-demand partial cache loading that avoids unnecessary network chatter.

Most important, the customer's developers helped Terracotta develop a migration kit that is now freely downloadable under the same open source license as Terracotta at http://www.terracotta.org/confluence/display/labs/Terracotta+Cache.

This migration kit will help you get on your way with Terracotta DSO in place of JBoss Cache without rewriting your application or causing any alteration of your caching service.