Top Five Tuning Tips

About Terracotta Documentation

This documentation is about Terracotta DSO, an advanced distributed-computing technology aimed at meeting special clustering requirements.

Terracotta products without the overhead and complexity of DSO meet the needs of almost all use cases and clustering requirements. To learn how to migrate from Terracotta DSO to standard Terracotta products, see Migrating From Terracotta DSO. To find documentation on non-DSO (standard) Terracotta products, see Terracotta Documentation. Terracotta release information, such as release notes and platform compatibility, is found in Product Information.

Release: 3.6
Publish Date: November, 2011 **Documentation Archive »**

Top 5 Performance Tuning Tips

The following performance tuning tips address common issues and could help boost your clustered application's performance by a significant amount. The issues progress from tuning your code all the way out to tuning network connections.

You can find much more tuning and performance-related information in Tuning Terracotta and in the Terracotta Deployment Guide.

Use Locks Appropriately

In a collection (usually a Map) that is shared across a Terracotta cluster, read and read/write operations take place on value objects. Use read lock-level autolocks on read (get) operations and write lock-level autolocks on write (set) operations. A get from the Map should be a read operation and read locked if you use a HashMap. No Terracotta autolock configuration is necessary if you use a thread-safe collection like CHM or CSM (see DSO Data Structures Guide).

For reads, you can lock on the collection:

```
Read lock on Map m
Bean b = m.get(key)
Read unlock on Map
```

The above happens automatically on thread-safe collections like CHM or CSM without any configuration or synchronization in code. To minimize lock contention, lock on the map value (the object) itself when updating. For example:

```
Write lock on Object obj
obj.setField1(value1)
obj.setField2(value2)
...
Write unlock on Object obj
```

If you see a lot of threads blocking on the same lock or high acquire times for some locks — basically, a few objects are accessed more often on the same JVM — then consider using ReadWriteLock symantics (for example, ReentrantReadWriteLock) instead of plain old synchronized. Note that using lock objects will increase the memory footprint of the application, but this may still provide an advantage in cases where the application is not memory bound yet has high write contention on a small subset of objects, thus causing too many threads to block.

Use an Optimized Collection

A typical Terracotta use case involves a shared Map, often a HashMap. In use cases calling for high concurrency, a more sophisticated map implementation is required to maximize performance by increasing locality of reference and reducing lock contention.

To take advantage of lazy loading, use a striped collection like ConcurrentHashMap (CHM). If all keys can be Strings, use the Terracotta Map implementation called ConcurrentStringMap (CSM), which is optimized to minimize lock contention.

For more information on data structures in Terracotta DSO, see the DSO Data Structures Guide.

Allocate Sufficient Memory

On each application server, allocate enough memory on the heap to hold the data required by your application. Signs that you have not allocated enough heap (Xms, Xmx) in application servers include the following:

- Many full GC cycles
- Long full GC cycles
- Many flushes or an increasing rate of flushes (see the Terracotta Developer Console)
- Many faults or an increasing rate of faults (see the Terracotta Developer console)

Set the Xms and Xmx to the same value.

Be sure to allocate enough heap on the Terracotta Server (default is 512MB). Signs that the server is low on memory are the same as noted above for application servers. In addition, use the Terracotta Developers Console to see the Cache Miss Rate, which when high indicates that the Server is thrashing to disk.

Thrashing to disk is acceptable if the file blocks are fetched from RAM (OS-level file caches). Set the TC Server heap to a max of 2 or 3 GB but make sure to have enough RAM on the Terracotta server machine to hold all the live objects.

Size the heap spaces appropriately. A common formula that works well in many cases is to set Eden space to one-third of the total heap.

Add Locality of Reference

A high rate of change broadcasts or lock recalls indicates that your application may have poor locality of reference and is referencing the same objects on multiple JVMs. One consequence of this is increased network chatter. See the Lock Recalls/Change Broadcasts graph in the Terracotta Developer console to track this rate.

Depending on the access patterns, your requirements, and your use case, poor locality of reference may be acceptable. If it's not, consider partitioning of data across application instances. Partitioning is can improve scale performance in applications with high write contention.

There are a number of ways to partition, including, for example, using a load balancer to consistently steer access to specific data. How you partition is very specific to the application, business use case, and access patterns.

Adjust for Network Issues

Network hiccups, latency, and other network issues can cause problems from orphaned clients to split-brain syndrome. These issues are often solved by adjusting client-to-server and server-server reconnection windows.

Other issues may arise from improperly configured networks, applications that choke bandwidth, or other culprits that can be found using good network analysis tools.

Resources

- See *Configuring Terracotta For High Availability* in
- Tuning Terracotta: Network Performance Tools
- Terracotta Deployment Guide: Network Hardware Provisioning