Testing Terracotta

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 »**

Testing a Clustered Application

Introduction

This guide is designed to introduce you to the major concepts involved in testing and tuning an application with Terracotta. The testing section describes the major kinds of testing that you should perform on an ongoing basis within your software development lifecycle and presents some best practices to follow to ensure successful software testing in the presence of Terracota.

Testing

Software testing already has a universe of different methodologies, each with their own emphases and terminology. Given that Terracotta is transparent to your application code, you should be able to continue using whatever testing methodology you prefer. However, because Terracotta introduces distributed computing to your application, you must, necessarily, augment your testing portfolio to include coverage of the extra functionality Terracotta provides.

To arrive at a common ground, we have defined a number of different kinds of tests (which may be known variously by other names) that you should implement, if you haven't already:

- Functional tests on a single node
- Functional tests on multiple nodes
- Thrash tests
- Long-running tests
- Boundary tests
- Performance tests
- Failure/destructive tests

Following is a description of each of these kinds of testing and what implications Terracotta has on each.

Functional Testing

The goal of functional testing is to make sure that your application business logic works as expected. In addition to your regular functional testing, you must also test your application running on multiple nodes to make sure that it still behaves as expected in the presence of Terracotta.

Functional testing can be done manually or automated. The basic steps for manual functional testing are as follows:

1. Test on one node without Terracotta enabled to make sure everything works as expected.
2. Test on one node with Terracotta enabled to make sure everything still works in the presence of Terracotta
3. Check the Terracotta Developer Console to see if the state in the console corresponds to what you were expecting. Pay particular attention to whether the Terracotta roots that you expect to exist actually do. If they don't, then something is probably misconfigured. Make sure Terracotta is actually enabled and reading the configuration you intend it to.
4. Bring up two nodes and alternate between them to make sure that your application behaves well when work hops between the two nodes. In a web application, this can be efficiently done by fronting your application cluster with a round-robin load balancer—this will make sure that your application behaves properly after hopping back and forth between the two application servers.
5. As a final sanity check, perform the multi-node test without Terracotta enabled and make sure that your application doesn't behave properly. This will ensure that your multi-node test is valid.

Thrash Testing and Long Running Tests

We define *thrash testing* as a set of tests that runs all of your application functions in combination with the high end of realistic load. This is similar to functional testing, but will help you flush out bugs and undesireable performance behaviors that occur under stress conditions.

Running thrash tests in a long-running form is also critical to shake out bugs and undesireable behavior that occurs only occasionally. If possible, you should always have a suite of long-running thrash tests running, 24 hours a day, seven days a week, in your test lab with alerts that notify you when they fail. You should expect these tests to run indefinitely with no failures. Any failures you find in your long running tests will almost certainly (eventually) occur in production, so make sure you run these tests constantly.

As you run your thrash tests, you should watch the operating stats over time. These tests are the closest proxy you have to what your application looks like in production, so use them to get to know what your application looks like through the lens of your monitoring infrastructure. You should be able to tell the difference based on what your monitoring tools tell you between your application in a healthy state and an unhealthy state. In particular, make sure that your JVM GC across the cluster and the Distributed Garbage Collector in each Terracotta server instance is behaving well. This will also give you practice at problem analysis and resolution when things go wrong—and things always go wrong, eventually.

Boundary Testing

Boundary testing is another term we made up to describe a set of tests that pushes the limits of your system along a particular set of vectors. The purpose of this kind of testing is to find out what the limits of your application cluster are and to find out how it behaves under those conditions. This is different than thrash testing in that you are looking for behavior at the limits of your application rather than at the high end of expected capacity.

Again, you should get to know what your application looks like under these conditions through the lens of your monitoring infrastructure so that you can recognize these conditions should they happen in production.

You can also use boundary testing to aid in capacity planning.

Performance Testing

Performance testing (and subsequent tuning and optimization) is a favorite pastime of many a developer. However, there are many pitfalls along the path to obtaining useful performance testing results, especially in a distributed computing environment.

The following is a useful set of guidelines to aid you in developing effective performance tests for distributed applications. (Many of these were taken from Steve Harris's excellent blog series on performance testing anti-patterns.) They are applicable generally to any distributed computing platform, but certainly true for Terracotta. You should also keep them in mind if you are planning to benchmark Terracotta's performance against another technology.

About Throughput And Latency

"Performance" is a measure of the throughput and latency of your system. For scalability, what you are looking to achieve is an increase in the total throughput your system can deliver when you add more CPUs. This allows you to scale your application's capacity by adding new servers to the cluster. This only works if your application can benefit from increased parallelism: adding more CPU to a single-threaded task won't help that task complete any faster.

Of course, you also need to consider latency when measuring performance. Most tasks have an upper bound of acceptable latency beyond which performance is perceived by the user to be poor, even if the total system throughput is very high. For example, if your web application is able to service many thousands of total requests per second, an individual user will still perceive your site as "slow" if the latency of servicing requests is high.

To balance throughput and latency, you should determine the upper bound of acceptable latency for a given operation, then measure the total throughput of your system such that no task exceeds that latency window. This will tell you how well your system performs in terms of both latency and throughput. To do this, you should take the output of your thrash tests and compute averages, medians, and standard deviations to find out how fast you appear to the user.

Always Test on Multiple Machines

It's tempting to run your performance tests on a single machine and forgo the hassle of setting up a multi-machine test infrastructure. There are two flavors of this: the very lazy flavor of testing on a single JVM and the not quite as lazy flavor of testing on multiple JVMs on the same machine. Tempting as it might be to do otherwise, you really *must* do your performance testing on multiple machines.

Testing on a single JVM will tell you either 1) nothing, because the clustering framework recognizes it has no partners so optimizes itself out or 2) very little—it might give you an idea of maximum theoretical read/write speed for that framework, but not the actual performance characteristics of your application running in a clustered manner.

Testing in multiple JVMs, but all on the same physical hardware machine has two problems. First, distributed applications running on the same machine have different latency and networking characteristics than distributed applications on different machines. This can hide various classes of problems around pipeline stalls, batching, and windowing issues.

The second problem is that single-machine "distributed" testing introduces artificial resource contention that will skew your results. By running multiple JVMs on one machine you are contending for CPU, disk, network, and potentially affecting context switch rate, etc.

When trying to evaluate the performance of any kind of clustering or distributed computing software, always use an absolute minimum of 2 application nodes, preferably more. Also, when testing Terracotta in particular, make sure you run your tests with each Terracotta server instance on its own dedicated hardware.

Send Realistic Load To All Nodes

When testing clustering software, make sure you are throwing load at all nodes.

If you are testing with multiple nodes but only sending load/work to one of those nodes while leaving the others just hanging out doing little or nothing, the nodes that are not receiving load may be actually doing a lot of work. If that's the case, only loading one of the nodes will skew your performance results. Also, in some cases, data is lazily loaded into nodes so only putting load on one node could be putting you in the same boat as the single-node tester where no actual clustering is happening.

Test With Realistic Data

Make sure you test with object graphs that vary in size, type, and depth in similar ways to the data you plan to use in your application. Don't assume, for example, that a Map of Strings will behave anything like the way real object data will behave.

Distributed computing solutions use all kinds of strategies to move data between nodes under the hood. Just representing a size of data to be shared ignores those strategies and in many cases misrepresents the performance of a real system with real data under real load, both positively and negatively. You may be testing specially optimized flattening tricks that make the system look faster than it is; likewise, you may be testing a particular case that doesn't perform well, but that isn't representative of the true performance of the system with real data.

Test The Coherency You Actually Require

Some clustering products are coherent, some are not, and some have both modes. Don't ignore whether you are testing the performance using the mode you really need for your application.

While it is quite possible to have a coherent cluster that has the same throughput as an incoherent cluster, it is certainly harder to do. Coherently clustered software frameworks like Terracotta do some fancy locking, batching, windowing, and coherent lazy-loading tricks that aren't for the faint of heart (in the internals of the clustering engine, that is, not for the application developer). You can't assume that performance between a coherent and incoherent clustering approach will be the same.

Make sure that if what you need is coherently clustered data that you are actually testing that way. Also, if it's coherence you're after, it's a good idea to verify the end-state of a performance test to make sure the system actually is coherent.

Test Realistic Parallelism

Make sure your test uses multiple threads for generating load in each JVM. Check to see if you are CPU bound on any node. If you are not CPU bound you might have a concurrency issue or just need to add more threads.

For most clustered software, the name of the game is throughput with acceptable latency. Pretty much all distributed computing software does batching and windowing to improve throughput in a multi-threaded environment. Maxing out a single thread will usually not even approach the max throughput of the JVM or the system as a whole in the same way that a single node will not max out an entire cluster.

Test Realistic Access Patterns: Don't Compare In-Memory Access To Distributed Access

Make sure you aren't comparing the speed of adding objects to a local, in-memory data structure vs. a more realistic pattern of adding objects to a clustered data structure where some or all of that data must be sent out of process.

Adding things to a local, in-memory data structure takes virtually no time at all. In-memory object changes happen so fast, they are hard to even measure. However, when you are making changes to a distributed data structure, no matter what, those state changes have to be shipped off to another location. This takes instructions to be executed to make this happen on top of the ones used for the original task. This isn't just slower, it is way slower. Remeber that clustering is not free. The purpose of clustering is to get more throughput as you add more hardware, not to make a single application node go faster. As long as you can get more capacity by adding CPUs to the cluster, you have the virtue of scalability that clustering is designed to deliver. The comparison between in-memory object changes and distributed object changes is useless.

Figure out how much data you are going to be clustering and what the usage patterns of that data becoming clustered will be. Then simulate and time that. Once again, focus on total throughput with acceptable latency.

Test Realistic Access Patterns: Test Cross-Node Access Patterns If You Have Them

Write your performance tests in a way that allows you to set a percentage for locality of reference. Is an object accessed on the same node 80%, 90%, or 99% of the time? You should usually have some cross-node chatter, but usually not too much—although you should be as realistic to the problem you are trying to solve as possible.

Generally speaking, whether reading or writing, it is more expensive to access the same data concurrently across nodes. Depending on the underlying clustering infrastructure, this can be more or less of a problem. If you are using an "everything everywhere" strategy, the performance hit of random access across all the data on all the nodes is less, but the "everything everywhere" sharing strategy generally does not scale well. Most other strategies perform better when data access is consistently read from and/or written to the same node.

Since Terracotta automatically faults objects in local heap as you access them, if you use the same data everywhere, then Terracotta will keep all data everywhere, as long as local JVM heap permits, after which, you will start to see object churn. In general, you should attempt to partition the workload and the data such that the data for a given workload is partitioned on the same JVM as much as possible.

Test Realistic Access Patterns: Don't Test Just Reads Or Just Writes

In the real world, an application does a certain amount of reading, writing, and updating of shared objects. And those reads, writes, and updates are of certain sizes.

If your app likely changes only a few fields in a large object graph, then that is what your performance test should do. If your app is 90% read from multiple threads and 10% write from multiple threads than that is what your test should do. Make your test be true to what you need when it comes to data and usage.

Don't create performance tests that only read or only write data. This will tell you virtually nothing about how your application will actually behave in production.

Don't Log Yourself To Death

Doing extra work like writing data out to a log chews up CPU. Logging too much in any performance test can render the test results meaningless. The only thing you should always log is JVM verbose GC, since that is often critical to understanding GC-related bottlenecks.

Look For Any CPU-Bound Components

In general, if one or more of your nodes is CPU bound in a cluster performance test, you likely have not maxed-out the performance of your cluster. This is important enough to bear repeating: if you are resource constrained on any node, including your load-generating nodes (but not including Terracotta server instances), then you are probably not maxing out what your cluster as a whole can handle. Investigate further. Try adding more nodes. Make sure that the load generator isn't your bottleneck. If it is, add more load-generating resources.

Always have machine monitoring on all nodes in a performance test. Any time one of the nodes or load generators becomes resource constrained make sure you test with an additional node and see if it adds to the scale. If a node is unexpectedly resource constrained, then take a series of thread dumps and figure out where all the time is going

If a Terracotta server instance is CPU bound (or over-utilizing disk I/O), consider testing multiple Terracotta server instances in a striped configuration on multiple machines. See *Terracotta Server Arrays* in

Error rendering macro 'html'

Notify your Confluence administrator that "Bob Swift Atlassian Add-ons - HTML" requires a valid license. Reason: EXPIRED

for more information.

A less visible but important performance issue involves optimizing the under-utilization of the hardware found in the cluster. You should regularly monitor the CPUs in the cluster to validate that the capacity of the each machine is being utilized. If a CPU is underutilized, consider:

- increasing the number of threads for more throughput;
- deploying multiple JVM instances on stronger machines (to utilize more CPU, RAM, and other resources)

Destructive Testing

Destructive testing is the process of testing various failure modes by causing such failures and making sure that the system behaves as expected under those conditions. This is a crucial step that must not be skipped before you deploy your application to production.

The first step to destructive testing is to understand and define what your failover requirements are. For example, if you lose a disk under a Terracotta server instance, your application service level agreements should determine the maximum amount of time acceptable for another server instance in the Terracotta server array to take over and resume service. Once your failover requirements are defined, create a test plan that exercises all the failure modes, including network failures, hardware failures, and various software subsystem failures. Then tune the failover times to meet your requirements.

Resources:

- See *Configuring Terracotta For High Availability*

  Error rendering macro 'html'

  Notify your Confluence administrator that "Bob Swift Atlassian Add-ons - HTML" requires a valid license. Reason: EXPIRED

  describes how to set up Terracotta server instances for high-availability (HA) and failure resistance, and includes a set of tests you can run to ensure HA is delivered.

Testing Automation

Thorough testing is critical to the long-term success of software applications. A comprehensive suite of automated tests is the only way to scale your testing effort to achieve the kind of test coverage and consistency that an enterprise software application requires.

Terracotta Maven Plugin

Terracotta has developed a Maven Plugin that lets you define Terracotta-specific goals inside your maven POM. For example, using the Maven Plugin, your project can define the main class to launch. The plugin can also easily launch all of the required Java VMs, including the Terracotta server instance, to run and test your clustered project. This is one good way to add testing automation to your Terracotta project.

The maven plugin can be downloaded from the Terracotta Forge.

Terracotta Forge »