Planning for a Clustered Application

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 »**

Planning for a Clustered Application

Introduction

As with most aspects of enterprise software, success with Terracotta will require some planning on your part. While Terracotta is transparent, you should not wait until your application is code complete before trying to get Terracotta working with it. The following sections describe some of the considerations you should make as you embark on your project.

If you want assistance planning for application clustering, Terracotta can help. For more information, see our Enterprise offerings or email info@terracottatech.com.

Architecture Considerations

What Application State Should You Cluster?

The most important thing to know is what application state you want to cluster. The answer depends on why you are clustering in the first place. When clustering for:

- High availability – Determine what data should be available to one application server should another application server instance (Terracotta client) fail.
- Scalability – Determine how to partition data and workload such that different application server instances can work in parallel.
- Cross-JVM communication and coordination – Determine what data will be shared between JVMs, and how it will be shared.

Below are high-level rules of thumb to keep in mind when deciding what data to cluster with Terracotta.

Good Locality of Reference

Try to keep as much of the data that a given application server will work on in memory on that application server. This implies a sensible data and workload partitioning strategy.

Less Garbage, Please

Try to make as little clustered garbage data (short-lived objects that last only seconds or minutes) as possible. Don't use Terracotta as a garbage can—remember that there is always overhead to clustering, so cluster only what is necessary.

Inter-JVM Communication, Not Data Streaming

Use Terracotta to pass small, relatively infrequent messages between JVMs. Clustering a chat application or configuration data is a smart use of Terracotta.

Terracotta is very efficient at keeping object data coherent and available, but it is not a point-to-point protocol. Don't use Terracotta to stream large amounts of data from one application server to another. Video streaming is unlikely to be an efficient use of Terracotta.

When Do I Use It?

If you're still wondering where to use Terracotta, we have put together a section of the website devoted to helping you figure out what clustered architecture patterns work best with Terracotta. If you haven't already, read the material in this introductory section.

Workload Partitioning

For effective scale out, you will need some sort of workload partitioning whereby you send controlled quantities of workload to each application server in your cluster. The most well-known example of workload partitioning is a layer 7 load balancer where user web traffic is bound to specific application server instances based on a user session identifier. This strategy works very well for partitioning workload along user boundaries. However, not all workload is user-based. If you have a non-user based workload (for example, you are running batch processes), you will want to determine a workload partitioning strategy specific to your application.

Data Partitioning

In addition to workload partitioning, you should also plan for data partitioning. As you scale out you want to avoid having all clustered data resident in heap on all JVMs at the same time. Each application server should only have the data that its workload requires access to in heap.

For example, if you are using a layer 7 load balancer which supports server affinity ("sticky sessions") to bind sets of user sessions to particular application servers, the user session data will automatically be partitioned across the cluster. If your load balancer sends web requests for sessions 1-500 to application server A and web requests for sessions 501-1000 to application server B, the session data for sessions 1-500 will be resident only on the heap of application server A; likewise, the session data for sessions 501-1000 will be resident only on the heap of application server B. As your capacity requirements grow, you can keep adding application servers, partitioning the session data according to how much heap you have available on each application server.

Assuming the session data for sessions 1-500 fits in heap in application server A and, likewise, the session data for sessions 501-1000 fits in heap in application server B, your cluster will exhibit a number of virtuous operating characteristics:

- All session access will happen at memory speed; your cluster will be very, very fast.
- You will only need as much RAM on each application server as required to accomodate its allocation of user sessions.
- If an application server fails, its session data will be automatically repartitioned across the remaining application servers, ensuring business continuity for your users.

If you do not use a layer 7 affinity based load balancer, but instead use a "round-robin" load balancer approach, your cluster will exhibit a number of vicious operating characteristics:

- Since you have a random distribution of session access across all of your application servers, all session data will tend to be required to be resident on all application servers.
- If you don't have enough memory in each application server to accomodate **all** of the session data for the entire cluster, some, perhaps most, session access will happen slower than memory speed as session data that doesn't fit in memory is swapped back and forth between application servers and Terracotta server instances; this kind of data "churn" is similar to what happens when the filesystem cache on your computer is blown out by one ore more very large files: file access that used to happen at memory speed degrades to disk speed and you experience the painful latency characterized by "swapping."
- To counteract the swapping effect, you will have to add very large amounts of RAM to each application server to accomodate all of the session data.

Obviously, using a data partitioning strategy that will allow the data for a given workload partition to fit entirely in heap is highly advantageous and strongly encouraged.

What Delivers Partitioning?

As discussed, using a layer 7 load balancer to distribute user workload and data across your cluster is an effective partitioning mechanism. For non-web applications or applications that have non-user data, a different partitioning mechanism will be required. If you are unsure how to determine your partitioning strategy and mechanism, Terracotta offers a number of services and a lot of clustered architecture experience that can help. See our Enterprise offerings or contact info@terracottatech.com for more information.

Caching Strategy

As you scale out, your database will come under increased load as the number of active connections and queries to it multiply. You will almost certainly need to cache data in your application servers to relieve database load.

Caching can alleviate database load, but it does so with some trade-offs:

- Your caches must be kept up to date, so you need to implement an expiration or update policy.
- As you add application servers, their caches will get out of sync with each other, especially if you use a time-based expiration policy.
- The more application servers you have, the less effective your caching will be, since each application server will be required to load data from the database independently. We call this the "1/n-effect," where the effectiveness of caching is inversely proportional to the number of application servers you have.
- If the data set you are caching is larger than what will fit in memory, you will be forced to purge even valid cache entries to reclaim heap space, further reducing the effectiveness of your cache.

Terracotta can help with all of these problems by providing a coherent and durable distributed cache:

- All cache expirations and updates are simultaneously and coherently available across the cluster, so your caches will remain in sync.
- Since there is a single view of the cache kept in the Terracotta server, your caches will not suffer the 1/n-effect. Once the cache is loaded, the database need not be consulted as new cache instances come online.
- Since the Terracotta virtual heap can be as large as the disk underneath a Terracotta server instance, you don't have to purge valid cache entries to reclaim heap space.

Do You Really Need A Database For That?

If you keep application data in your database solely to make it highly available—for example, because you have implemented a "stateless" architecture—you might consider pulling that data out of the database and leaving it in Terracotta instead. Because Terracotta stores your object data to disk, it has the same availability characteristics as the database, so you can treat object data in memory as safe. Plus, since your application has access to it at memory speed, it will be much more efficient than marshalling and unmarshalling it back and forth between the database and your application server without the data coherency and freshness problems engendered by database caching.

As a rule of thumb, for business data that you will report on later or need to run queries against, a relational database is the best choice. However, for application state data that you don't need to query or report on, using a relational database to store it is probably highly inefficient and expensive. A good example is user session data. You will almost never need to run a report on the contents of your user session data. Therefore, it makes no sense to store it in a relational database. Terracotta's durable heap is a much better place to keep that kind of data.

Code Considerations

What TIMs To Use

When it comes time to configure your application for Terracotta, your task will be much easier if you can use some of the many pre-built and tested Terracotta Integration Modules (TIMs). TIMs are purpose-built, tested, and tuned to implement a particular clustered architecture pattern or integrate with a particular third-party technology. There are integrations for most of the major application servers as well as TIMs for many popular development frameworks and libraries.

For a complete list of Terracotta third-party technology integrations, see Integrating Terracotta DSO or the Terracotta product documentation.

What If There Is No TIM?

If you do not find a TIM to match a technology you want to use, you can try to configure Terracotta to work with it yourself. Likewise, if you want to cluster your own data structures with Terracotta, there is a simple configuration process to follow. This configuration process is described in detail in the next section, Configuring Terracotta.

Locking Strategy

Just like the JVM uses synchronization to batch and order changes to objects between different threads, Terracotta uses Java synchronization to batch and order changes to objects into transactions between threads in different JVMs. In this way, your Terracotta clustered application will look just like a single-JVM multi-threaded application. This means that you will have to consider how you are going to lock (or synchronize) access to shared data.

There are some Terracotta-specific considerations vis a vis locking that you should consider that will have an effect on application performance. For more information about how locking works in Terracotta, see the locks section of the Concept and Architecture Guide. For locking-related performance considerations, see the lock tuning section of the DSO Tuning Guide guide.

Are There Any Non-Clusterable Objects?

There are some objects that are not clusterable. Mostly, these objects represent JVM-specific or system-specific resources, like network sockets, filehandles, graphics contexts, and the like. If a clustered object graph that you create has a reference to one of these non-clusterable objects, you will need to make that reference transient. This tells Terracotta to ignore that reference for the purposes of clustering. You must, however, remember to initialize those transient fields when the object is materialized in another JVM or if the the object is purged from the local heap due to memory pressure and re-materialized in the local JVM. Terracotta provides a few different ways to perform this task.

Resources:

- For more information on what is not clusterable, see the "Portability" section of the Terracotta Concept and Architecture Guide.
- For more information on how to make fields transient, see the next section on Configuring Terracotta.

Development Plan

Your development plan should reflect the fact that you are using Terracotta. To ensure success, you should develop with Terracotta as part of your process. Terracotta uniquely provides you the ability to code at a unit level without having to set up a complex clustering environment. This is a significant benefit to your development efficiency, but you should regularly checkpoint that the code you are writing still works as expected when Terracotta is clustering your application.

Test Plan

You **must** have a plan for testing your software in the presence of Terracotta. We strongly recommend that you implement an automated continuous integration and testing system to be warned early and often of any regressions or new bugs in your application. We, ourselves, take this advice to heart and have a sophisticated automated distributed testing framework—and, it's built using Terracotta! We are currently in the process of making this system consumable by the rest of the world, so stay tuned.

You should be prepared to develop and perform three kinds of tests:

1. Functional tests—these will ensure that your application works as expected
2. Performance tests—these will help you perform capacity planning and to uncover any scalability bottlenecks that will prevent your application from reaching the capacity you require.
3. Destructive tests—these will ensure that your application and cluster work as expected when parts of your system fail (e.g., switches, servers, disks, etc.).

Performance Tuning and Capacity Planning

You should plan on spending some time prior to deployment on performance testing and tuning. We like to say that Terracotta reduces clustering to tuning. Because you don't have to spend weeks or months implementing lots of enterprise infrastructure, you get to the tuning phase of an enterprise application much faster than you may be used to. But, as with any application that sees significant production load (and, after all, that's why you're scaling out, right?), your application with Terracotta will most likely need some tuning to achieve maximum performence. Even a Steinway needs to be tuned.

With proper tuning, we have seen applications increase their capacity by orders of magnitude. Keep in mind that tuning is an ongoing part of the maintenance of your application. You should plan on a performance tuning step before you release new versions of your software.

Systems and Network Provisioning

Part of the purpose of your performance testing will be to do capacity planning so you can provision your network and systems hardware. While there are no hard and fast rules to follow for provisioning your hardware, there are some rules of thumb you can follow. The Terracotta Deployment Guide provides an in-depth discussion of what to consider during capacity planning.

Network Architecture

The Deployment Guide also offers guidelines for your network architecture in the presence of Terracotta. You will want to consider your requirements for high-availability, throughput, and latency when designing the network fabric between Terracotta clients and Terracotta server instances as well as between active and standby server instances in the Terracotta server array.

You will also want to determine what your Terracotta server array will look like. There are a number of options for configuring Terracotta server instances for high-availability and fast fail-over. These are also discussed in the Deployment Guide.

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. See the Testing Guide for information on destructive testing.

Operations

Living successfully with a Terracotta deployment, as with any production deployment, requires developing and rigorously adhering to an operating plan. This plan should include a runbook that describes what steps to take for all routine operations on your cluster as well as how to monitor the health of your cluster and take action against unplanned events, such as hardware failures, load spikes, and the like.

See the Operations Guide for information on how to prepare for operating your Terracotta cluster.

Next Steps

Now that you've read about planning for Terracotta, you're ready to start configuring your application to use Terracotta.

Configuring Terracotta »