Gotchas

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 »**

Gotchas

Introduction

We strive to make Terracotta as transparent as possible and we are constantly improving our transparency and usability, but there are still some things which may seem unintuitive at first glance.

This document seeks to highlight some of these issues and solutions for them.

Debuggers and Breakpoints

Debuggers and breakpoints may stop the execution of Terracotta byte code before it has a chance to execute. Terracotta often resolves data in arrays and Object references "just in time".

This means if you happen to inspect an array or object reference before Terracotta has resolve the data, you may find that the data structure appears to be null.

This behavior is a result of the way Terracotta works. It changes your byte code instructions to resolve data "just in time". To observe the clustered data, you must "use" the data you are inspecting in some way, e.g. traverse the Object reference, or de-reference the array index.

More on Debuggers

Using the debugger, one can observe field values of shared objects to have phantom null values (or null element values in a array). This is a side effect of the just-in-time lazy loading and dynamic memory management features in DSO. Although a field can appear null in the debugger view, your application will see the correct values when/if the fields are read. The main issue here is that the debugger uses non-bytecode means of viewing object state. Stepping over some code that reads the phantom null field will cause the value to be resolved. Although it might be obvious, using the debugger to mutate a field of shared object will bypass Terracotta leaving your locals in an inconsistent state with the cluster.

Lock-Then-Share

Summary
Terracotta requires that locks are acquired on an object that is already shared. In some instances, your code may inadvertently lock a local object, then share it. The unlock will then throw an exception.

Example

```
public class LockThenShare
{
    private static Map myMap = new HashMap();  // tc root

    public void lockThenShare()
    {
        Object foo = new Object();
        synchronized (foo) { // object is not shared yet so only a local lock is acquired
            synchronized (myMap) {  // ok, myMap is a root
                myMap.put("key", foo);  // foo is now shared
            }
        } // oops, tried to unlock the foo object which is now shared, but a tc lock was never acquired
    }
}
```

Solution

Share then lock. For this example:

```
public class ShareThenLock
{
    private static Map myMap = new HashMap();  // tc root

    public void lockThenShare()
    {
        Object foo = new Object();
         synchronized (myMap) {  // ok, myMap is a root
                myMap.put("key", foo);  // foo is now shared
        }
        synchronized (foo) { // tc lock on foo is acquired
             ... foo operations
        } // ok, tc lock on foo is released
    }
}
```

Not Autolocking the Enclosing Synchronization

The following exception can seem anomalous if a class is using local synchronization but wrapping a wait() or notify():

```
java.lang.IllegalMonitorStateException
       at

com.tc.object.tx.ClientTransactionManagerImpl.wait(ClientTransactionManage
rImpl.java:212)
...
```

wait()/notify() calls are always instrumented, but forgetting to autolock the enclosing synchronization will cause this exception.

Sharing Locks in a Session

If code in class Foo contains:

```
public void synchronized m()
```

then the code is synchronizing on the instance of Foo. If m() process a request, and a session object is created, that session object is locked withing the thread that processes the request. But if another thread mutates that object and Foo is not shared, then an unlocked shared exception will be thrown. Locks do not hop along with the session object.

Uninstrumented Access

Summary

Reading and writing to objects shared in Terracotta DSO from uninstrumented methods can be problematic under certain conditions.

Condition 1

The first problem involves reading/writing accessible fields of a shared object from uninstrumented code. For example, consider two classes (A and B) in the same package. Class A has an accessible field (obj), and is instrumented. Class B is not instrumented.

```
public class A  // instrumented
{
   public Object obj = new Object();
}

public class B // not instrumented
{
  public void foo(A a)
  {
    a.obj = this;  //access to obj is not instrumented - trouble looming
  }
}
```

If class B is not instrumented, the reads and writes of the accessible "obj" field of class A will not function correctly for shared A instances from class B. The negative effects of an uninstrumented read usually provoke a NullPointerException, but could also manifest in stale values (and not just nulls).

Condition 2

A second, similar problem can occur with arrays when uninstrumented code reads/writes a shared array instance.

Condition 3

A third problem can occur when uninstrumented code interns an instrumented string. Terracotta DSO compresses instrumented strings whose size exceeds a certain configured value. Interning compressed strings from instrumented code is safe because that code is monitored and the strings are decompressed when called. However, if uninstrumented code interns a compressed string by calling java.lang.String.intern(), that string is *not* decompressed.

You can configure the threshold for compressing strings, or disable compression altogether. **However, disabling string compression may reduce the efficiency of your clustered application.**

To enable or disable string compression, set the `l1.transactionmanager.strings.compress.enabled` property to true (default) or false. You can set the string compression threshold size by adjusting the `l1.transactionmanager.strings.compress.minSize` property to the number of bytes a string must reach before compression is applied (default is 512).

See the Configuration Guide and Reference on how to configure these properties in your Terracotta configuration file.

Solution 1

Take an OOP best-practices approach and encapsulate all access to external fields.

```
public class A  // instrumented
{
   private Object obj = new Object();

   public void setObj(Object o) { this.obj = o; }
}

public class B // not instrumented
{
  public void foo(A a)
  {
    a.setObj(this); //  ok
  }
}
```

Solution 2

Instrument all involved classes.

```
public class A  // instrumented
{
   public Object obj= new Object();
}

public class B // instrumented
{
  public void foo(A a)
  {
    a.obj = this;  // ok
  }
}
```

Improper Locking

Summary

Locking in Terracotta honors the Java 1.5 Memory Model. It is important to understand this model, because race conditions and/or improper locking in your application can affect its operation in a clustered context using Terracotta.

You must, however, keep in mind that these problems **are not Terracotta specific**. If your application has race conditions and/or improper locking, it is susceptible to data corruption or unexpected behavior **at any time**. Moving your application to a multi-cpu machine, or integrating it with Terracotta will likely expose these flaws more quickly than if left to execute in a single-cpu single-core box, however these environments only serve to expose these existing flaws more quickly.

An application that works well in a clustered environment is a better written application, even on a single-cpu single-core environment.

Example

A classic data-race can be exposed by omitting synchronization. Here is a classic data race condition:

```
public class RaceCondition
{
    private boolean on = false;

    public void doSomething()
    {
        if (on) {
            // do an on something
        } else {
           // do an off something
        }
    }

    public void setOn(boolean on)
    {
        this.on = on;
    }
}
```

If two threads execute doSomething() at the same time, while another thread is calling setOn(), then the state of the on field will be non-deterministic. Worse, in a multi-cpu machine, updates to on **may never even be seen by another thread** and could cause serious problems.

The common refrain that is repeated too often is that "primitive updates are atomic". While this is certainly true, the compiler (and Terracotta) do not have any "happens-before" semantic that they are required to honor, and therefore updates to the on field do **not** have to be propagated across threads (or JVMs in the case of Terracotta.

The second problem that is quite common is the solve only part of the problem, by locking one half of the execution, but leaving the other half unlocked. This may or may not be intentional, but has the same problem as the previous example. Without explicit locking that gives strict "happens-before" semantics, your program is at risk of having unintended side-effects.

```
public class WriteLockedButNotReadLocked
{
    private static Map myMap = new HashMap(); // tc root

    public void readAndUpdate(String key, String value)
    {
        MyObject myObject = myMap.get(key);  // read is not protected!!!
        myObject.update(value);
    }

    public void put(String key, MyObject object)
    {
        synchronized (myMap) {
            myMap.put(key, object);
        }
    }
}
```

As in the previous example, the compiler, there is no happens-before relationship established between the write to the map, and the read from the map. Other threads may **never** see the write, so the readAndUpdate method could very easily throw a NullPointerException.

Solution

Synchronization is cheap so **do not pre-optimize**. Omitting synchronization is a classic case of early optimization, and can lead to subtle and often difficult to resolve bugs. Write your program correctly, profile it, and optimize the parts that are actually running slow.

The correct way to write the "WriteLockedButNotReadLocked" class is the following:

```
public class WriteLocked
{
    private static Map myMap = new HashMap(); // tc root

    public void readAndUpdate(String key, String value)
    {
         synchronized (myMap) {
             MyObject myObject = myMap.get(key);  // read is now protected!!
             myObject.update(value);
         }
    }

    public void put(String key, MyObject object)
    {
        synchronized (myMap) {
            myMap.put(key, object);
        }
    }
}
```

If you find that the `readAndUpdate` method has contention then you can convert this example into a striped-lock:

```
public class WriteLockedAndReadLocked
{
    private static Map myMap = new HashMap(); // tc root

    public void readAndUpdate(String key, String value)
    {
        synchronized (myMap) {  // read lock alleviates contention on myMap
            MyObject myObject = myMap.get(key);  // read is now protected!!!
            update(myObject)
        }
    }

    public void update(MyObject myObject, String value)
    {
         // lock is striped, it would be rare for multiple updates to happen to the
         // same myObject
          synchronized (myObject) {
               myObject.update(value);
          }
    }
    public void put(String key, MyObject object)
    {
        synchronized (myMap) {
            myMap.put(key, object);
        }
    }
}
```

Class Evolution

Summary

Changing the class structure of shared data may introduce errors such as ClassCastExceptions into cluster operations.

Class evolution is supported only in some simple cases, such as adding or removing fields. Changing the type of a field is not supported.

Existing objects (previously instantiated) are not affected by the changes made to a class.

Example

The class Example has the following structure:

```
Class Example {

   String name;
   String address;
   int age;
}
```

Some time after Example has been used, it is changed to the following:

```
Class Example {

   Person name;
   String address;
   double weight;
   int age;
}
```

The addition of the reference `weight` probably will not cause any errors, but the change in the type of `name` is likely to cause errors.

Solution

Export your data to a form that can be re-imported in way that worked with the changed data and schema.

Static Fields Not Shared

Summary

The static fields of classes instrumented by Terracotta are not shared along with instances of those classes. Shared instances and non-static fields of the instrumented class are clustered, which means they exist in shared object graphs on all JVMs in the cluster. Static fields belonging to the class are not clustered and do not exist in shared object graphs, but still exist in local object graphs on any JVMs where their containing class is instantiated.

Example

```
public class Test
{
   public static AnotherType anotherType = new AnotherType(argument);

//  ...

}
```

Even if instances of Test are shared, anotherType is *not* shared.

Solution

If AnotherType can be be made non-static, then it will be shared if instances of Test are shared. If AnotherType must remain static, then it must be declared a Terracotta root, which allows it to be shared while remaining a static member of Test.

Multiple Initialization

Summary

The behavior of root initialization can be a bit unexpected, and can cause more than one root object to be constructed. Terracotta will ensure the root field is initialized once-and-only once across the cluster, however it cannot prevent a constructor from being called more than once.

Example

```
public class Root
{
    public static Root root = new Root();  // configured as tc root

    public Root()
    {
        System.out.println("Root constructor called.  I am:" + this);
    }

    public static void main(String[] args)
    {
        System.out.println("Cluster root is: " + root);
    }
}
```

If you run this program once, you will see:

```
$ dso-java Root
Root constructor called.  I am @12345
Cluster root is: @12345
```

Run it again, you will now see:

```
$ dso-java Root
Root constructor called.  I am @56789
Cluster root is: @88123
```

Confusing? It's not that bad. Here's what happened.

The second time you ran this program, in a separate VM, the constructor was called per standard VM rules. And it's object id on the heap is @56789. However, this second time it runs, there is already a cluster wide instance for the root. When the assignment happened to the root field, Terracotta discarded the new instance and instead gave the field the instance that corresponds to the cluster instance.

So why is that instance id not the same as the first time through? Well even though Terracotta preserves Object Identity across the cluster, it does so by proxy. There is no way to make object ids the same, but it is possible to make the *behavior* identical. Terracotta does the latter, so the answer is that @88123 and @12345 are both the same logical instance.

Solution

This will always happen. You just should be aware that it will happen. Your constructors (for root objects) should not have side-effects.

Transients

Summary

Transient mostly works exactly like it does in Serialization. When an object is transported to a second VM that does not have the object in memory, any transient fields will not be replicated. In the target VM, the transient field will be null.

However, in Terracotta, this process can happen at any time, due to Virtual Memory, and can be a bit unexpected.

Example

In a single VM, you wouldn't expect the following to happen:

```
public class TransientGotcha
{
    private transient Object object = new Object();

    public static TransientGotcha create()
    {
        TransientGotcha gotcha = new TransientGotcha();
         SharedMap.put("theObject", gotcha);  // assume SharedMap shares the object
         return gotcha;
    }

    public static TransientGotcha get()
    {
        TransientGotcha gotcha = SharedMap.get("theObject");
         gotcha.check();
         return gotcha;
    }

    public void check()
    {
        if (object == null) {
            System.out.println("Object ref is null!!");
        }
    }

    public static void main(String[] args)
    {
        create();
         .... // more code here
        get();  // this might in fact detect a null reference
    }
}
```

This can happen because Terracotta's Virtual Memory subsystem may have decided to evict the transientObject object. Then, after some time, the same VM requests the object again, and therefore it is faulted in. Since this happened at the discretion of Terracotta, and not the programmer, it can be expected.

Solution

This will always happen. Just be aware that it may happen even if you only have one VM. Make sure you test your shared objects being faulted in to a new VM. If the new VM throws a null pointer exception because a transient was not initialized, you can fix that using the "onLoad" configuration setting explained in the Concept and Architecture Guide:

- Concept and Architecture Guide - Transients in Terracotta

String Equality in Distributed Applications

Summary

In applications distributed with Terracotta, reference equality for string literals may not behave as expected. In applications that are not distributed, the following is expected to be true since all existing String literals are interned at compile time:

```
"abc" == "abc"
```

However, when the application is distributed with Terracotta, the statement above could be false because the references being compared may have been generated on different JVMs.

Cause

Terracotta ignores string internment at compile-time and treats these automatically interned strings as if they were not interned. If the first String object "abc" and the second String object "abc" were both interned at compile time, then later faulted into Terracotta clients, the equality shown in the Summary above would be false.

Example

If an application manages a string literal with AtomicReference, which uses the == operator internally to find reference equality, errors could occur because references that are expected to match may not.

This kind of error could also occur with other types of literals.

Solution: Explicitly intern String literals to store a single canonical value

Terracotta tracks String.intern() calls and guarantees reference equality for these explicitly interned strings. Since all references to an interned String object point to the canonical value, reference equality checks will work as expected even for distributed applications.

The equality shown in the Summary above is true when the first String object "abc" and the second String object "abc" are explicitly interned with String. intern() before their equality is tested.

Conclusion

Avoid relying on compile-time string internment if your application needs to test for string equality, using String.intern() instead.

More Information

For related issues and information, see:

- #Uninstrumented Access
- Literals in Terracotta

Method.invoke() Cannot Expand Container of Arguments

Passing an array of arguments that came from a Distributed Shared Object (DSO) that contains other DSOs doesn't get expanded inside `java.lang.reflect.Method.invoke()`.

Manually resolving the array in place fixes the problem (traverse all contents before handing off the DSO to the reflection-invoked method).

When a DSO enters or exits a clustered LinkedBlockingQueue, for example, its contents are not always paged in (references are null).
This happens when its contents are passed as an argument to `Method.invoke()`.

Instrumented Method Too Long

Summary
An application that runs without error gets the following error when running with Terracotta:

```
ava.lang.ClassFormatError: Invalid method Code length
```

Cause
The method named by the ClassFormatError is just under the maximum length allowed by the JVM. After being instrumented with Terracotta, the method is over the length limit.

Solution 1
If the method named by the ClassFormatError does not need to be instrumented by Terracotta, exclude it from instrumentation. For example, if the method is called `myMethod`, add the following <exclude> element to the Terracotta configuration file:

```
<instrumented-classes>
...
  <exclude>myPackage.myClass.myMethod</exclude>
...
</instrumented-classes>
```

Solution 2
If the method named by the ClassFormatError must be instrumented by Terracotta, split the method so that after instrumentation no portion of it is over the length limit allowed by the JVM.

Mutable Enums

TODO: Check - this may not be a gotcha as of 2.5 or 2.6. Describe that changes to enumerations aren't clustered (DSO behavior mimics their out-of-the-box serialization/deserialization behavior)

java.lang.ref.Reference

If user code has a java.lang.ref.Reference to a shared object Foo, Foo can be paged out of memory by the DSO reaper. The reference will be nulled out.

This behavior may be surprising to some user code. The expectation is that the reference value won't become null until the object has entered some phase of the garbage collection cycle.

See http://jira.terracotta.org/jira/browse/CDV-330 for more information.

Generics and Runtime Class-Cast Exceptions

Under certain circumstances, code using generics can cause a runtime class-cast exception when clustered with Terracotta.

The following code example shows a collection using generics.

```
public class Main<T>
 {
    // tc root
    private final HashMap<String, T> root = new HashMap<String, T>();

    public void put(T t)
    {
        root.put("foo", t);
    }

    public T get()
    {
        return root.get("foo");
    }
 }

// execute this code in one node:

      Main<Main> main = new Main<Date>();
      main.put(new Date());

// execute this code in another node:
      Main<String> main = new Main<String>();
      main.get();
```

A `put()` operation on one node can put objects of type "Main<Date>" into the data structure held by the root. But then calling `get()` on another node results in a runtime class-cast exception (essentially, the second node is attempting to cast a `Date` to a `String`).

The exception can be prevented by avoiding the use of different type parameters on different nodes.

Shared Iterator

Summary
Explicit sharing of iterator instances is not supported. Iterators created from shared collections are not problematic, but again, these iterator instances themselves cannot be shared objects (only the underlying collection).

Solution
Don't share iterators. Share the collection, then iterate it from within a local context.

TODO: Check. Iterators may be shareable in Terracotta 2.5 or 2.6.

Collections

Standard Java provides a number of collections (HashTable, SynchronizedMap etc.) that are thread-safe by default. For performance reasons, these data structures are not auto-locked by default in Terracotta, meaning they will not work as expected if they are part of a clustered graph.

The following Collections are not auto-locked in Terracotta:

- HashTable
- SynchronizedCollection
- SynchronizedList
- SynchronizedMap
- SynchronizedSet
- SynchronizedSortedMap
- SynchronizedSortedSet
- Vector

`java.util.concurrent.ConcurrentHashMap` is auto-locked by default.

Solution

The Terracotta Forge project `tim-collections` provides pre-configured modules to enable auto-locking for these Collections. More details can be found on the Terracotta Forge:

- Forge TIM Collections

`tim-collections` does not provide auto locking for `java.util.concurrent.SynchronizedList`. Check the following Jira issue for the latest information:

Unable to locate Jira server for this macro. It may be due to Application Link configuration.

Hidden References

Summary

The java compiler generates a synthetic reference back to the enclosing instance for anonymous and non-static inner classes. In anonymous and non-static inner classes, these "hidden" references can unexpectedly lead to sharing of instances that were not intended for sharing.

Example

Consider this code:

```
class A
{
    public void bar()
    {
        // does something here
    }

    public Runnable foo()
    {

          return new Runnable()  {
              public void run() {  }
          };
   }

   private class Inner
   {
         //  ...

         public void doSomething()
         {
             bar();
         }
   }
}
```

In both the non-static inner class named "Inner" and the anonymous inner class in the "foo" method, the compiler introduced as synthetic field named "this$0" containing a reference to the enclosing A instance.

This hidden reference becomes significant in Terracotta DSO when trying to share inner class instances since the enclosing type must be portable, and the enclosing instance automatically becomes shared.

Solution
Consider using static inner classes to prevent unintended side-effects. Using an explicit pointer for inner classes allows the pointer to be marked transient, in effect preventing the enclosing instance from being shared.

```
class A
{
    public void bar()
    {
        // does something here
    }

   private static class Inner
   {
         private transient final A a;

         public Inner(A a)
         {
              this.a = a;
         }

         public void doSomething()
         {
             a.bar();
         }
   }
}
```

Different Hashcode on Separate JVMs

Summary

For primitive types, if an Object is present on two separate JVMs, then it should return the same hashcode on both the JVMs. But the hashcode of a custom object depends on the Object.hashCode() method and could have different hashcodes on separate JVMs.

The inconsistency occurs when two key-value pairs with the same key are added in a Map by two different JVMs. This leads to different hashcodes for the same key, and possibly results in a clustered Map with two entries for the same key-value pair.

Cause

Object.hashCode() is a native call and uses the physical memory location of the object to generate a hashcode. On two separate JVMs, it may return different values and cause inconsistencies.

Solution

Override the Object's hashCode() method in custom objects to ensure that the hashcode generation for the custom object is through the overridden hashcode method, not through Object.hashcode(). The overridden method should always return a consistent and unique hashcode for the same Object across different JVMs and between JVM restarts.

Terracotta DSO does not manipulate the Object.hashCode() method in any way.

In the following example, overriding the hashCode() method ensures that no inconsistency exists in the hashcode generated for the same Object of type A in two different JVMs. The Class A uses String.hashCode() to compute the hashcode for Objects of A and not the Object.hashCode() method. This ensures that the hashcode for objects of class A is dependent on the int arithmetic done by String.hashCode() method and not based on the physical memory location of the object.

```
Class A {
    String str = "HashCodeTest";
    public int hashCode() {
        return (str != null ? str.hashCode() : 0);
    }
}
```

IBM JDK

There are some known issues with the IBM JDK.

key	summary	status

⚠ Can't show details. Ask your admin to add this Jira URL to the allowlist.

View these issues in Jira