DSO Async Processing

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 »**

Terracotta DSO Clustered Asynchronous Processing

Introduction

Terracotta Distributed Shared Objects (DSO) clustered asynchronous processing, implemented with the Terracotta Integration Module **`tim-async`**, removes commit-related database bottlenecks by committing data in an independent and fully asynchronous way.

How DSO Asynchronous Processing Works

Each application node has an asynchronous coordinator managing a number of independent buckets, each of which processes the same type of data. While the data in these buckets is clustered, remote processors do not fault this data in. Instead they work only on local data which has been assigned to them. However, this processing design does allow for "work stealing" whenever a coordinator detects that processing has become slow or died.

Scale

Scaling out is achieved by adding more processing threads and buckets to a coordinator.

Data Persistence

Data is persisted in Terracotta until commitment to the SOR is confirmed. Without Terracotta, data loss becomes a potential issue for some ORM applications such as Hibernate. However, even with persistence, if a node fails before data commitment is verified then another commit attempt is made. See #Improving the Data Commitment Model for a design that avoids recommitting data already in the SOR.

How to Implement DSO Asynchronous Processing

`tim-async` provides a simple API with data structures that create and manage local buckets:

- AsyncCoordinator – The bucket coordinator class. The bucket coordinator manages buckets which have registered with it. All buckets registering with a coordinator must process the same type of data. If possible, maximize the efficiency of work distribution by using one logical instance of AsyncCoordinator to manage the cluster's buckets. This is done by declaring the AsyncCoordinator a Terracotta root.
- ProcessingBucket – The bucket class. Instances of ProcessingBucket register with the instance of AsyncCoordinator to become working buckets. The data held by ProcessingBucket instances is shared and persisted by Terracotta, making it failsafe.
- ItemProcessor – The processor class that works with the data in a bucket. The ProcessingBucket constructor takes an instance of ItemProcessor that is able to work with the type of data intended for that bucket. ItemProcessor instances are not shared, giving them the flexibility to freely to refer to and interact with local resources such as Hibernate Sessions Managers, JDBC connection pools, and filesystems.

If different types of data items must be handled, create a new AsyncCoordinator for that type or build an ItemProcessor that detects and handles the different types your application needs to commit.

The typical cycle of a data item is to be put into a ProcessingBucket that's intended for that data type, then wait to be processed by the appropriate ItemProcessor. Once the data item is successfully processed, it is removed from the ProcessingBucket.

For example:

```
import org.terracotta.modules.async.AsyncCoordinator;
import org.terracotta.modules.async.ItemProcessor;

@Root
private final AsyncCoordinator<MyItem> asyncCommitter = new AsyncCoordinator<MyItem>();

public Test() {
  asyncCommitter.start(new MyItemProcessor()); // MyItemProcessor implements ItemProcessor.
   }

public void doStuff(MyItem) {
      // ...
     asyncCommitter.add(data);
      // ...
   }
```

⚠ For @Root to successfully create a Terracotta root, the Terracotta annotations TIM must be configured in Terracotta.

Adding Error Handling and Callback

If an error occurs while a data item is being processed by ItemProcessor, ItemProcessor.process() should throw a ProcessingException error. Then the data item will remain in the ProcessingBucket until it can be processed again. Note that it is possible the item was actually successfully processed because the failure may have occurred just after data was written to the SOR. In this case, see Improving the Data Commitment Model.

If you require a callback mechanism during an error, such as to connect to a database or log file, implement the callback logic in ItemProcessor. ItemProcessor is not shared and can reference local resources. A typical implementation involves passing in local resources or state to the ItemProcessor constructor. A lookup mechanism is another option.

Configuring and Tuning AsyncCoordinator

As the heart of Terracotta clustered asynchronous processing, AsyncCoordinator makes available a number of controls and settings for tuning its operation to customize its functionality.

Work Stealing

"Work stealing" functionality allows the AsyncCoordinator to even out workload and account for unresponsive ProcessingBucket instances by moving data items from one bucket to another. The following work-stealing policies are available:

- FallBehindStealPolicy – (Default) Work stealing is active and stealing will occur if threshholds are met.
- NeverStealPolicy – Work stealing is active but *no* stealing can occur.

To set one of the standard policies, or to set a custom policy, pass an instance of a work-stealing policy to the AsyncCoordinator constructor. For example:

```
new AsyncCoordinator<MyItem>(DefaultAsyncConfig.getInstance(), new MyCustomPolicy());
```

Delays and Timeouts

The delays and timeouts used by the AsyncCoordinator have default values. You can get the default values by using DefaultAsyncConfig.getInstance(). While the default values cannot be changed, custom values can be configured by passing a configuration class to AsyncCoordinator.

First, create the custom configuration class, which must implement AsyncConfig:

```
import org.terracotta.modules.async.configs.DefaultAsyncConfig;

public final static class TestAsyncConfig implements AsyncConfig {
     public long getAliveWakeUpDelay() { return 10 * 1000; }
     public long getDeadBucketDelay() { return 45*1000; }
     public long getMaxAllowedFallBehind() { return 2000; }
     public long getWorkDelay() { return 1000; }
     public boolean isStealingEnabled() { return false; }
   }
```

To tune an AsyncCoordinator instance, edit the values shown in each TestAsyncConfig.get..(). These values replace the default values set by the following constants:

- ALIVE_WAKEUP_DELAY = 10 * 1000; // 10 seconds; sets the interval for AsyncCoordinator to check if a ProcessingBucket is alive.
- DEAD_BUCKET_DELAY = 20 * 1000; // 20 seconds; sets the maximum time for AsyncCoordinator before deciding a ProcessingBucket is dead.
- WORK_DELAY = 1000; // 1 second; facilitates "workstealing" by setting the time a ProcessingBucket waits between processing its existing batch of pending items and the next. This normally affects nodes that run out of work, forcing them to attempt to steal work after the period specified by WORK_DELAY expires.
- MAX_ALLOWED_FALLBEHIND = 2000; // 2 seconds; the maximum amount of consecutive WORK_DELAY intervals a ProcessingBucket can reach before another ProcessingBucket can steal pending work.

To take effect, an instance of TestAsyncConfig must be passed to AsyncCoordinator constructor:

```
new AsyncCoordinator<MyItem>(new TestAsyncConfig());
```

Concurrency

The AsyncCoordinator concurrency setting determines the number of buckets – per JVM – used by the AsyncCoordinator to simultaneously handle incoming data items.
Use the int processingConcurrency in AsyncCoordinator.start() to set the number of buckets. For example if processingConcurrency is set to 2, and there are 4 client JVMs, 8 buckets are created in total.

Scatter Policy

Scatter policy determines how data items are parceled out to the AsyncCoordinator buckets. The default policy is to randomly select buckets.

Starting AsyncCoordinator

Running AsyncCoordinator instances cannot detect new cluster clients. Therefore, when a new client is started, application code must explicitly execute `AsyncCoordinator.start()` to allot buckets and allow asynchronous processing of data to take place.

In the Examinator reference application, the Spring implementation creates an instance of the ExamSessionServiceImpl service at application startup, which starts an AsyncCoordinator instance in its constructor.

Stopping AsyncCoordinator

You can stop an instance of AsyncCoordinator by calling `AsyncCoordinator.stop()`. All processing stops, and no guarantee is made that unprocessed items will be processed.

Improving the Data Commitment Model

If the process of committing data is interrupted before completion, for example by node failure, shared data is preserved by Terracotta. However, the data will have to be recommitted unless application logic exists to discover and prevent duplicate commits. For example, the Examinator reference application, which uses Hibernate as an ORM, has to frequently commit exam results to a database. If the commit process gets interrupted by some type of failure, Terracotta ensures that the data is not lost, but some data may have to be recommitted because normally the application would have no way of knowing if certain data had been committed successfully before the failure.

Examinator solves the "duplicate commits" issue by attaching an ID to each exam result. This is done using Hibernate annotations in the ExamResults class:

```
import org.terracotta.modules.async.AsyncConfig;

  @Id
  @GeneratedValue(generator="system-uuid")
  @GenericGenerator(name="system-uuid", strategy = "uuid")
  @Column(name = "RESULT_ID")
```

The class that commits exam results, ExamResultsCommitHandler, can then check for the ID to determine if the exam results need to be committed:

```
/*
 * All content copyright (c) 2003-2009 Terracotta, Inc., except as may otherwise be noted in a separate
copyright
 * notice. All rights reserved.
 */
package org.terracotta.reference.exam.async;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.terracotta.modules.async.ItemProcessor;
import org.terracotta.reference.exam.domain.ExamResult;
import org.terracotta.reference.exam.service.ExamService;

@Service
public class ExamResultCommitHandler implements ItemProcessor<ExamResult> {
  private final ExamService examService;

  @Autowired
  public ExamResultCommitHandler(final ExamService examService) {
    this.examService = examService;
  }

  public void process(final ExamResult result) {

    // check if the ID property is set
    if (result.getId() != null) {
      if (examService.examResultExistsById(result.getId())) {
        // this entity was already persisted
        return;
      } else {
        // the entity was not persisted, but the ID was set
        // clear the ID so that it will be persisted below
        result.setId(null);
      }
    }

    // save the new exam result
    examService.saveExamResult(result);
  }
}
```

Data Freshness

While data waits in a bucket to be asynchronously written to the SOR, it is possible for a request aimed at the same data to complete. This request would fetch stale data. Depending on your application's characteristics, this scenario may not be likely as data is committed quickly enough.

If, however, fresh data must always be available, it is possible to design your application to query memory before making a request of the SOR. The data inside the `tim-async` data structures is not visible outside, so it should also be copied to a secondary data structure ("staged results") also shared by Terracotta. A commit handler should clear this secondary data structure of any data which is confirmed committed to the SOR.

Synchronous vs. Asynchronous Data Commitment in Examinator

Synchronously writing data to a system of record (SOR) can create a performance bottleneck for clustered applications, especially with requirements to commit data under spiky or high-volume conditions. As load on the system rises, it is common for applications that persist data to a database to hit this bottleneck. This is because of the inherent scaling limitations found in these types of performance-driven environments.

An example illustrating the advantages of the `tim-async` write-behind design is provided with the Examinator reference application, where exam results are asynchronously written to a database. This prevents application servers from suffering potentially severe performance degradation during high-volume commit operations. Without asynchronous processing, a large amount of data would have to be synchronously committed each time a large number of tests ended simultaneously, not an unlikely occurrence in a test-taking application.

With the synchronous commit pattern, performance is negatively affected all the way up to the user-interaction level. The asynchronous commit pattern introduced by `tim-async` separates application threads from the slower write processes, which are moved into the background. To this, Terracotta adds safety through in-memory data persistence, and scale and efficiency through clustering.