MUDI

Multithreaded Dispatcher Framework

Bruno Ranschaert, S.D.I.-Consulting BVBA

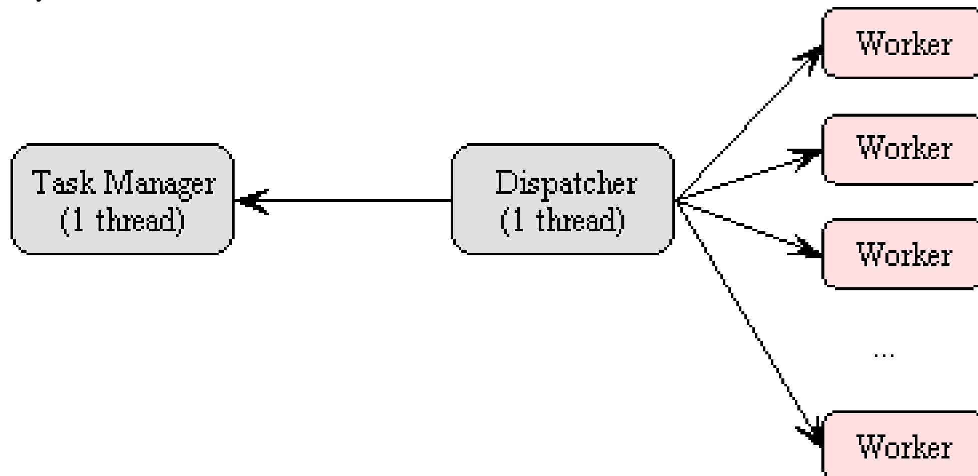
Table of Contents

1. Introduction 1
    1.1. Description 1
    1.2. Why use MUDI? 2
    1.3. Goal 2
2. Quick Start 2
    2.1. A minimal example 2
3. Building the project 4

1. Introduction

1.1. Description

MUDI is a small framework to enable the creation of a multithreaded dispatcher application. The framework has already proven its usability, it is used in a number of very robust applications which run 7/7 days 24/24 hours.



There are three important concepts in the overview. First there is the Task Manager which creates the tasks. Secondly, there is the Dispatcher wich dispatches the work to the worker threads, and third we have the worker threads themselves.

Task Manager

- Generates the tasks, the work that has to be done.
- Tasks could be fetched from file, database, ...

Dispatcher

- Manages the worker threads.
- Dispatches work items to idle threads.
- Asks for new work to the task manager.
- Keep track of which worker is working on which task.
- See that tasks are executed once.

Worker

- Give control to the task.
- Manage a single thread.

1.2. Why use MUDI?

- Create a dispatcher to send commands to a J2EE based command processor.
- Replace a typical batch application with a multithreaded application.

1.3. Goal

It is the goal of the MUDI framework:

- Should be able to handle a large number of threads, thousands.
- Should be configurable at runtime, when using a JMX wrapper.
- Should have enough parameters so that it can be scheduled using a scheduling mechanism (e.g. UNIX cron) on a regular basis.
- Basis to rewrite batch applications in J2EE context.
- Should be configurable using the Spring framework.

2. Quick Start

2.1. A minimal example

First we define a piece of work we want to have accomplished. We create a dummy task for the sake of simplicity and call it "MyWork".

```
private static class MyWork
implements BatchItem
```

```
{
    public void process()
    throws BatchException
    {
        System.out.println("Working...");
    }
}
```

Now we have to create a class that is able to generate the work items. In this example we create a composer that composes 20 times the same task. In a real application, the composer will walk through a file or a database to fetch the data to create tasks.

```
private static class MyComposer
implements BatchComposer
{
    private static int nrWork = 20;

   public List fetchItemsToProcess()
    throws BatchException
    {
        final List lResult = new LinkedList();
        if(nrWork == 0) return lResult;
        else
        {
            lResult.add(new MyWork());
            nrWork--;
            return lResult;
        }
    }
}
```

Finally we are ready to set up our batch processor.

```
package com.sdi.mudi;

import com.sdi.mudi.impl.BatchProcessorImpl;
import junit.framework.TestCase;

import java.util.LinkedList;
import java.util.List;

public class TestExample
extends TestCase
{
    public void testExample()
    throws BatchException
    {
        ❶BatchProcessorImpl lProcessor = new BatchProcessorImpl();
        ❷lProcessor.setBatchComposer(new MyComposer());
        ❸lProcessor.setIdleQuit(true);
        ❹lProcessor.setMinutes(1);
        ❺lProcessor.setThreads(10);
        ❻lProcessor.start();
    }

    // SNIP helper classes - see above
    ///////////////////////////////////
```

```
}
```

1. A processor is created. Most applications will only need one.
2. We provide our composer to the processor. The processor asks the composer for work if it needs work.
3. This optional parameter tells the processor to stop working if there is no more work (if the composer cannot create any work anymore). If you set this to false the processor will keep on running, it will go to sleep for an amout of time (configurable) and it will poll the composer for work after this interval.
4. The number of minutes the processor should run. Handy if the processor is scheduled using a scheduling application. If it is scheduled every 4 hours, the time to run could be set to 3 hours 45 minutes.
5. The number of threads. At least one. The maximum is dependent on the virtual machine implementation and operating system. On some UNIX systems, we were able to run thousands of threads smoothly.
6. Go for it.

3. Building the project

The project is built using Jakarta Maven 1.0.2. You first have to install a working version of Maven on your development machine. You can find it on the Jakarta website. The docbook manual can be rendered to HTML or to PDF using the sdocbook plugin which can be found in the maven-plugins project on SourceForge. I will not describe in detail how this should be done, consult the Maven manuals on how to do this.

I experienced a problem while automatically installing the sdocbook plugin. When running the maven sdocbook command the system complained that the jimi-1.0.jar was missing. This jar used to be available in the Maven repository but due to license problems (it is from Sun) the jar had to be removed. You should download the jimi library yourself from Sun, extract the JimiProClasses.zip from the download, rename it to jimi-1.0.jar and put in your local repository {yourhome}/.maven/repository/jimi/jars/jimi-1.0.jar

A second problem when downloading the plugin on Linux is that you should have the rights to write the plugin into the installation directory of Maven. Consult your system administrator when in doubt.