ThreadWorks™: User Guide

Documentation Version 1.0.0a

Copyright © 2001 Delta Vortex Technologies, Inc. All Rights Reserved.

Java™ and all Java-based trademarks are registered trademarks by Sun Microsystems, Inc.
Windows® and all Windows-based trademarks are registered trademarks by Microsoft Corporation.

Table of Contents

Installation

This section provides detailed instructions for a Windows installation and a UNIX installation. Even though we differentiate between UNIX and Windows installation procedures, ThreadWorks™ software is 100% Java™ and is identical for all platforms.

Requirements

A Jave runtime (JRE) that is version 1.2.2 or above must be installed and functional. You can verify the installation and version by executing the following command from a command prompt (in either Windows® or UNIX):

```
java –version
```

You will see a display much like the following:

```
java version "1.3.1"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1-b24)
Java HotSpot(TM) Client VM (build 1.3.1-b24, mixed mode)
```

An installation directory must exist and the user doing the install must have authority to write to that directory.

Installation Procedure

Windows Instructions:

After downloading, double-click **`TWInstall.exe`**

You may need to install a Java 1.2.2 (or later) virtual machine. You can download one from Sun's Java web site.

Unix Instructions:

After downloading open a shell and, **`cd`** to the directory where you downloaded the installer.
At the prompt type: **`sh ./TWInstall.bin`**

You need to install a Java 1.2.2 (or later) virtual machine. You can download one from Sun's Java web site or contact your OS manufacturer.
If you do not have a Java virtual machine installed, be sure to download the package above which includes one. Otherwise you may need to download one from Sun's Java web site or contact your OS manufacturer.

All Other Platforms Instructions:

Instructions (Unix or Unix-like operating systems)
For Java 2, after downloading, type

```
  java -classpath TWInstall.jar install
```

If that does not work either, on sh-like shells, try

```
  cd [to directory where TWInstall.jar is located]
  CLASSPATH=TWInstall.jar
  export CLASSPATH
  java install
```

Or for csh-like shells, try

```
  cd [to directory where TWInstall.jar is located]
  setenv CLASSPATH TWInstall.jar
  java install
```

Notes
Be sure you have Java 1.2.2 or later installed. You can download Java from Sun's site
In a console window, change to the directory where you downloaded **`TWInstall.jar`** to before running the installer.

Your operating system may invoke Java in a different way. To start the installer, add **`TWInstall.jar`** to your **`CLASSPATH`**, then start the main class of the installer named **`install`**.

Follow the instructions on the screen or at the command prompt.

Classpath Setting

In order for your applications to use ThreadWorks, place the following two items in your CLASSPATH:

```
[Install Directory]/[ThreadWorks Directory]/lib
[Install Directory]/[ThreadWorks Directory]/lib/ThreadWorks.jar
```

IDE Configuration

To use ThreadWorks within your IDE, you need to create a ThreadWorks library. In most IDEs, the three items of information you need is the classpath information above, a documentation directory, and a source directory.

Please set the documentation directory to the following:

```
[Install Directory]/[ThreadWorks Directory]/docs/api
```

Please set the source directory to the following:

```
[Install Directory]/[ThreadWorks Directory]/lib/ThreadWorks-src.jar
```

ThreadWorks is open source. ThreadWorks-src.jar contains full source code. The benefit to specifying the source is that the descriptive argument names make ThreadWorks easier to use from a development standpoint.

Package Organization

ThreadWorks comes with javadoc. It's located in the documentation directory given above. To help you find the javadoc to the classes referred to in this manual, it's helpful to understand our package structure. ThreadWorks uses the following packages:

Package Name	Description
com.dvt.app.threadworks	Contains classes TaskManager, Scheduler, and all other ThreadWorks related classes.
com.dvt.app.common	Contains common utilities and interfaces used in all Delta Vortex Technologies products.
com.dvt.app.common.log	Contains common Logger interface as well as several Logger implementations

ThreadWorks uses many more classes and packages behind the scenes. But only those classes under the com.dvt.app package are meant for public consumption. Many commercial API's don't take the trouble to separate public consumables from internally needed classes. We think that this makes an API harder to learn.

Samples

ThreadWorks comes with several samples. They are located in directory [Install directory]/doc/samples. The samples to start with are SamplesUsingTaskManager.java and SamplesUsingScheduler.java.

TaskManager

TaskManager is a utility that manages a configurable number of threads that will run tasks for you. The services that TaskManager provides are as follows:

1. TaskManager will asynchronously run tasks for you.
2. If all threads are busy, TaskManager will place your task in the work queue and it will be executed in the order it was received.
3. Optionally, TaskManager will send a CompletionEvent with status information when your task (or group of tasks) has been finished.
4. Optionally, TaskManager will submit tasks on your behalf when all dependent tasks have completed successfully.
5. Tasks which take longer than a configurable time are terminated and the termination is logged in the *Logger* provided.
6. TaskManager logs all processing errors to the *Logger* provided.
7. TaskManager will periodically (and upon request) log tuning diagnostics.

We see TaskManager as a complementary product to application servers, CORBA Object Request Brokers (ORB's), and other server-side application paradigms. One of the many issues that server-side applications have is preventing the machine from thrashing during workload spikes. As a benefit of the work queue concept, TaskManager will manage CPU bandwidth allocated to the tasks it runs (particularly useful for server-side portions of web-enabled applications). During a workload spike, work a TaskManager isn't configured to handle waits in the work queue and isn't allowed to consume all available CPU bandwidth.

TaskManager Setup

Typically, TaskManagers are declared as *static* and shared between multiple users/threads within a Java Virtual Machine (JVM). TaskManager is thread-safe. You do not need to guard access to TaskManager objects via the *synchronized* keyword.

The simplest way to instantiate a TaskManager is to provide a maximum number of Tasks and let all other properties default. An example follows:

```
import com.dvt.app.threadworks.*;
……………
private static TaskManager _manager = null;
……………
_manager = new TaskManager(5);
```

One way to customize your TaskManager is to establish a properties file *and put it in a subdirectory listed in your classpath.* A complete list of TaskManager properties can be found in appendix 1. Example code using a properties file in the classpath called "mytaskmanager.properties" follows:

```
import com.dvt.app.threadworks.*;
……………
private static TaskManager _manager = null;
……………
_manager = new TaskManager("mytaskmanager.properties");
```

Another way to customize an environment for a TaskManager is to instantiate a DefaultTaskManagerEnvironment object and customize it before instantiating the TaskManager. An example follows:

```
import com.dvt.app.threadworks.*;
……………
private static TaskManager _manager = null;
……………
DefaultTaskManagerEnvironment env = new DefaultTaskManagerEnvironment();
env.setMaxNbrConcurrentTasks(10);
env.setTaskTimeAllowedInMillis(4000);
env.setLogger(new FileLogger("foo.txt"));

_manager = new TaskManager(env);
```

TaskManager has several configurable properties. See appendix 1 for a complete list.

There is no benefit to artificially inflating the maximum number of concurrent tasks. In fact, as many server-side EJB containers make the number of Threads configurable, artificially inflating this number can take resources away from other parts of your application or other applications that might need them.

TaskManager Usage

Any task submitted to TaskManager has to implement the java.lang.Runnable interface. This is the same interface that you would need to implement if you were to manage your own threading.

While the run() method (the only method in the Runnable interface) doesn't allow you to throw exceptions, it is possible that low-level JVM errors and exceptions happen when tasks are run. TaskManager will trap and log execution errors when they occur along with the name of the class that generated the error or exception. To facilitate support, you might want more information than jus the class name if you wanted to try to replicate the problem. If a task also implements our Describable (from package com.dvt.common) interface, which means that it can produce a textual description of itself, TaskManager will log the task description as well.

An excerpt of a runnable task as an example follows:

```
public class TestTask implements Runnable {

  public TestTask() {_timeAtCreation = System.currentTimeMillis();}

  public void run() {
    _timeRunStarted = System.currentTimeMillis();
    long idleTime = _timeRunStarted - _timeAtCreation;
……………………………………………… // Omitted code here
    _timeRunEnded = System.currentTimeMillis();
  }
………………………………………………  // Omitted code here
}
```

Once you have a runnable task created and instantiated, you can submit it to a TaskManager to have it run asynchronously. A simple example follows:

```
manager.run(task);     // "manager" is of type TaskManager
                       // "task" is a Runnable
```

Optionally, TaskManager can send you a completion notification. To receive notification when your task completes, you provide a CompletionEventListener as well as a task. The CompletionEventListener is an interface with one method, notify(CompletionEvent event). When your listener is notified of task completion, you'll receive a CompletionEvent which will describe if the execution was successful and provide that task that completed in case the same listener is receiving completion events for multiple tasks.

An excerpt of a CompletionEventListener follows:

```
import com.dvt.app.threadworks.CompletionEvent;
import com.dvt.app.threadworks.CompletionEventListener;

public class TestCompletionListener implements CompletionEventListener {

  public TestCompletionListener() {_startTime = System.currentTimeMillis();}

  public void notify(CompletionEvent event) {
    long time = System.currentTimeMillis() - _startTime;
    System.out.println("CompletionTime: " + time + " ms.");
    System.out.println("Completion Success Ind: " + event.isSuccessful());
    if (! event.isSuccessful())
event.getObjectThrown().printStackTrace(System.out);
  }

  private long               _startTime;
}
```

Once you have a completion event listener, it's easy to get completion notifications. An example follows:

```
manager.run(task, listener);    // "manager" is of type TaskManager
                                // "task" is a Runnable
                                // "listener" is a CompletionEventListener
```

TaskManager can also give you completion notifications for groups of tasks. An example of this follows:

```
manager.run(taskArray, listener);    // "manager" is of type TaskManager
                                     // "taskArray" is a Runnable[]
                                     // "listener" is a CompletionEventListener
```

In this example, the listener will receive a completion notification when *all* tasks have been completed or one of the tasks generated a processing error. If one of the tasks generated a processing error, the CompletionEvent your listener receives will contain information about the error. If all tasks completed successfully, the CompletionEvent your listener receives will describe only the last task to finish.

One of the difficulties with multi-threaded programming is coordinating dependent tasks. For example, suppose upon successful completion of one group of tasks we want to execute additional tasks. This is a complicated problem if you're programming the thread coordination. TaskManager automates this coordination for you so you don't have to.

We call the first group of tasks "dependent" tasks and the addition tasks that execute when the dependent tasks finish "successor" tasks. A short example follows:

```
manager.run(dependentTaskArray, successorTaskArray, listener);
                                     // "manager" is of type TaskManager
                                     // "dependentTaskArray" is a Runnable[]
                                     // "successorTaskArray" is a Runnable[]
                                     // "listener" is a CompletionEventListener
```

In this example, tasks in the "dependentTaskArray" run first. If all of them complete successfully, all tasks in the "successorTaskArray" are run. If one of the tasks generated a processing error, the CompletionEvent your listener receives will contain information about the error. If all tasks completed successfully, the CompletionEvent your listener receives will describe only the last successor task to finish.

Receiving Completion Notifications

CompletionEventListeners can take action when your task(s) complete (e.g. such as activating a menu option or marking results as available for a user).

TaskManager considers task execution successful if it runs without generating an error or exception. The CompletionEvent that your listener is sent when task(s) complete will contain the Throwable (as in java.lang.Throwable) that was generated on a failed execution. The next example excerpt of a completion listener illustrates how to determine if a task run was successful and what information you have about the failure.

```
public void notify(CompletionEvent event) {
  Object task = event.getSource();          // “task” is your runnable.
  if (! event.isSuccessful()){              // Task run unsuccessfull!
     Throwable t = event.getObjectThrown(); // Get Exception or Error thrown
     //  Do your other error processing here.
  }
  else  {
     //  Do any processing for successful completions here.
  }
}
```

Scheduler

Scheduler runs tasks at a scheduled date/times. The services Scheduler provides are as follows:

1. Scheduler will run tasks at a specific date/time.
2. Scheduler will optionally rerun tasks at a given time interval.
3. Scheduler will optionally check to see if custom time-based dependencies are meet before running the task.
4. Scheduler logs all processing errors to the *Logger* provided.
5. Scheduler will periodically (or upon request) log tuning diagnostics.

A valid question at this point might be "Are there advantages to using Scheduler instead of java.util.Timer". For those that aren't familiar, java.util.Timer was added to Java in V1.3.0. Timer has a subset of the ThreadWorks' Scheduler capabilities. Timer will perform one-time executions of tasks at a specific time. As Timer manages one background Thread, any run-away task scheduled through it can keep other scheduled tasks from running. Also, if you need to schedule a recurring task, you need to write that logic yourself. Also, Timer doesn't have non-time based dependency support as ThreadWorks' Scheduler does.

As ThreadWorks' Scheduler manages a TaskManager, you benefit from all the infrastructure and tuning support that TaskManager provides. Also, ThreadWorks' Scheduler, unlike Timer, can scale with your application.

Scheduler Setup

To use the default Scheduler configuration, simply schedule a task. If Scheduler hasn't been instantiated, it will be on the first task scheduling with a default configuration.

To use a custom Scheduler configuration, instantiate Scheduler *before scheduling any tasks.* If you're using Scheduler within an Application Server, such as Weblogic or WebSphere, instantiate the Scheduler in a startup class. An example follows:

```
import com.dvt.app.threadworks.*;
……………
DefaultTaskManagerEnvironment env = new DefaultTaskManagerEnvironment();
env.setMaxNbrConcurrentTasks(3);
env.setTaskTimeAllowedInMillis(4000);
env.setLogger(new FileLogger("foo.txt"));

Scheduler s = new Scheduler(env, 50);
```

One way to customize your Scheduler is to establish a properties file *and put it in a subdirectory listed in your classpath.* A complete list of Scheduler properties can be found in appendix 2. Example code using a properties file in the classpath called "myscheduler.properties" follows:

```
import com.dvt.app.threadworks.*;
……………
Scheduler s = new Scheduler("myscheduler.properties");
```

Only one instance of a Scheduler per JVM is allowed. There is no need to declare a Scheduler in *static* space – Scheduler does that natively.

Usage Examples

Most of Scheduler's methods are *static*. This means that you don't have to have an instance of a Scheduler to use it.

An example of a one-time execution of a task with a 1 minute (60,000 ms) delay follows:

```
Scheduler.schedule(task, 60000);
```

An example of a one-time execution of a task on January 1, 2002 follows:

```
Scheduler.schedule(task, new GregorianCalendar(2002,0,1));
```

An example of a task first executed on January 1, 2001 at 1 am and recurring every hour () after that follows:

```
GregorianCalendar gc = new GregorianCalendar(2002,0,1,1,0,0);
Scheduler.schedule(task, gc, Scheduler.HOURLY_INTERVAL);
```

Custom Dependencies

Custom dependencies are coded in a class that implements the DependencyEventListener interface. To schedule a task with a custom dependency, provide a dependency listener when you schedule it. This dependency listener is responsible for determining if your custom dependency criteria has been meet. Note that the DependencyEventListener is also a CompletionEventListener, so the listener can be notified of task completions.

The execution sequence for implementing custom dependencies will look something like this:

1. Task is scheduled with a DependencyEventListener.
2. After the scheduled execution time/date has passed, Scheduler will invoke the dependenciesAllowExecution() method of the DependencyEventListener.
3. If dependenciesAllowExecution() returns a true value, the task will be run and a CompletionEvent will be sent to the DependencyEventListener.
4. If dependenciesAllowExecution() returns a false value, the task will not be run and will be rechecked on the next spin count interval.

An example DependencyEventListener follows:

```
import com.dvt.app.threadworks.CompletionEvent;
import com.dvt.app.threadworks.DependencyEventListener;

/**
 * Test time-based dependency listener for Threadworks.
 */
public class TestDependencyListener implements DependencyEventListener {
  public TestDependencyListener() {}

  public boolean dependenciesAllowExecution(Runnable task) {
    boolean answer = false;

    /**********************************************
     *  Put your logic to check dependencies here.
     */

    return answer;
  }
  public void notify(CompletionEvent event) {
     // If completion events are part of your dependency check, put logic here.
     //  This method may be stumped for some listeners.
  }
}
```

Once you have a DependencyEventListener, you can schedule tasks with custom dependencies as follows:

```
GregorianCalendar gc = new GregorianCalendar(2002,0,1,1,0,0);
Scheduler.schedule(task, gc, new TestDependencyListener() );
```

If you use the custom dependency feature of the Scheduler, it is important to set the spin count higher than the run time of the dependenciesAllowExecution() methods for all DependecyEventListeners. If the runtime for the dependenciesAllowExecution() method is long, it will prevent other scheduled tasks from being run when they are supposed to. For example, if the spin count is 50 milliseconds and it takes two minutes to check task dependencies, unrelated tasks which were supposed to start in those two minutes will be delayed. In this example, I would set the spin count to something over two minutes or performance tune your dependenciesAllowExecution() method.

TaskManager Tuning

TaskManager will log a tuning diagnostic at a configurable interval. Information in this diagnostic can be used to tune TaskManager configuration for your application. This chapter will show you how.

Diagnostic Message Format

The easiest way to begin is with an example, which is in the listing below. This example is from a TaskManager with the maximum number of tasks set to 20.

Listing 7-1: Example TaskManager diagnostic output.

```
---------------------------------------------
TaskManager Diagnostic -- January 03, 2002 at 04:03 AM.
       Thread Statistics:
              Thread 0: NbrExec=2: NbrInt=3: NbrFlood=1: PctBusy=16.564
              Thread 1: NbrExec=2: NbrInt=3: NbrFlood=1: PctBusy=19.902
              Thread 2: NbrExec=2: NbrInt=3: NbrFlood=1: PctBusy=18.068
              Thread 3: NbrExec=0: NbrInt=1: NbrFlood=0: PctBusy=0
              Thread 4: NbrExec=2: NbrInt=3: NbrFlood=1: PctBusy=28.51
              Thread 5: NbrExec=2: NbrInt=3: NbrFlood=1: PctBusy=22.13
              Thread 6: NbrExec=2: NbrInt=3: NbrFlood=1: PctBusy=22.13
              Thread 7: NbrExec=2: NbrInt=3: NbrFlood=1: PctBusy=28.46
              Thread 8: NbrExec=0: NbrInt=1: NbrFlood=0: PctBusy=0
              Thread 9: NbrExec=0: NbrInt=1: NbrFlood=0: PctBusy=0
              Thread 10: NbrExec=0: NbrInt=1: NbrFlood=0: PctBusy=0
              Thread 11: NbrExec=0: NbrInt=1: NbrFlood=0: PctBusy=0
              Thread 12: NbrExec=0: NbrInt=1: NbrFlood=0: PctBusy=0
              Thread 13: NbrExec=0: NbrInt=1: NbrFlood=0: PctBusy=0
              Thread 14: NbrExec=0: NbrInt=1: NbrFlood=0: PctBusy=0
              Thread 15: NbrExec=0: NbrInt=1: NbrFlood=0: PctBusy=0
              Thread 16: NbrExec=0: NbrInt=1: NbrFlood=0: PctBusy=0
              Thread 17: NbrExec=0: NbrInt=1: NbrFlood=0: PctBusy=0
              Thread 18: NbrExec=0: NbrInt=1: NbrFlood=0: PctBusy=0
              Thread 19: NbrExec=0: NbrInt=1: NbrFlood=0: PctBusy=0

       Task Statistics:
              Task com.dvt.test.app.threadworks.support.TestTask: NbrExec=16:
AvgLag=0.0: AvgRun=2126.0

       TaskManager Statistics:
              Weighed Average-->  Lag: 0.0 Run: 2126.0 Total: 2126.0
              Work Queue Size-->  0
```

You can identify TaskManager diagnostic output by the “Task Manager Diagnostic” label at the head of it. The first section, entitled “Thread Statistics”, contains detailed statistics on each thread managed by TaskManager. The second section, entitled “Task Statistics” present aggregate statistics on individual tasks that were run by the task manager. Aggregate statistics on the TaskManager itself is presented in the “TaskManager Statistics” section. You can also find the current work queue size at the time of the diagnostic.

Definitions on thread-level statistics follow:

Statistic	Description
Thread	TaskManager internal ID for thread.
NbrExec	Total number of tasks run by this thread.
NbrInt	Total number of times thread checked the work queue looking for work.
NbrFlood	Total number of times thread checked the work queue looking for work and received additional work to do.
PctBusy	The percentage of time during the life of this TaskManager that this thread was busy.

Definitions on task-level statistics follow:

Statistic	Description
Task	Fully-qualified class name for task
NbrExec	Total number of times this task was run by the TaskManager.
AvgLag	Average time in milliseconds this task waited in the work queue before beginning execution by a TaskManager thread.
AvgRun	Average execution time for this task in milliseconds.

Definitions on TaskManager-level statistics follow:

Statistic	Description
Lag	Weighted average time in milliseconds that tasks waited in the work queue before execution.
Run	Weighted average execution time in milliseconds for all tasks run by this TaskManager.
Total	Lag + Run

Interpreting TaskManager Diagnostic Information

This section details what to look for when reviewing TaskManager diagnostic information and provides some general tuning rules to consider. These tuning guidelines are as follows:

1. Consider increasing the number of threads if the PctBusy numbers are consistently higher than 50% for all threads.
2. Consider increasing the number of threads if the total NbrFlood from all threads exceeds half of total number of tasks run by the TaskManager.
3. Consider increasing the number of threads if the average “lag” time is not tolerable.
4. Consider decreasing the number of threads if the PctBusy numbers are less than 20% for more than 2 threads.
5. Consider synchronously running tasks with average runtimes less than 100ms (instead of using a TaskManager).

Now that we’ve defined the tuning information TaskManager gives us and some guidelines to follow, let’s interpret the results from the example listing in the previous section. If we re-examine the PctBusy measurements from the example listing above, we see that most threads are idle or nearly so most of the time. If this were a normal load for a TaskManager, we should decrease the maximum number of concurrent tasks and see what happens with the measurements.

As an experiment, I reran the same test decreasing the maximum number of concurrent tasks to three so we can see what the measurements from a more heavily loaded TaskManager look like.

Listing 7-2: Example TaskManager diagnostic output.

```
---------------------------------------------
TaskManager Diagnostic -- January 03, 2002 at 04:07 AM.
       Thread Statistics:
              Thread 0: NbrExec=2: NbrInt=2: NbrFlood=1: PctBusy=24.893
              Thread 1: NbrExec=5: NbrInt=6: NbrFlood=4: PctBusy=50
              Thread 2: NbrExec=4: NbrInt=5: NbrFlood=3: PctBusy=36.335

       Task Statistics:
              Task com.dvt.test.app.threadworks.support.TestTask: NbrExec=16:
AvgLag=125.0: AvgRun=2126.0

       TaskManager Statistics:
              Weighed Average-->  Lag: 125.0 Run: 2126.0 Total: 2251.0
              Work Queue Size-->  0
```

From the listing above, we see much higher PctBusy numbers as well as an increased number of flooding incidents. Given the fact the the total number of flooding incidents (1 + 4 + 3 = 8) is close to exceeding half of the total number of tasks we ran through the task manager, we might consider raising the number of threads by one or two and see how the statistics look.

Also notice that the average lag time increased from 0 ms. to 125 ms. This means that tasks waited an average of 125 ms before they ran. A higher lag time is an indicator of a potentially overworked TaskManager.

Logger Usage

All tools from Delta Vortex Technologies will use a *Logger* to record any type of processing message. *Logger* is an interface with methods that will allow logging of debug messages, informational messages, warning messages, or processing error messages.

We provide two standard logger implementations: a *FileLogger* and an *OutputStreamLogger*. As you might guess, *OutputStreamLogger* can be used with any OutputStream (e.g. System.out or System.err). By default, it uses System.out for informational and debug messages and System.err for warning and error messages.

```
import com.dvt.app.common.log.OutputStreamLogger;
……………
OutputStreamLogger log = new OutputStreamLogger(System.err);
```

FileLogger writes log messages to a file. You specify a filename on construction. If you specify a file name that already exists , such as “log.txt”, *FileLogger* will rename the old log by inserting a date/time stamp in the name (i.e. “foo.20020111.03574900524.txt”). We do this so that you won’t accidentally overwrite previous log files.

An example of creating a FileLogger follows:

```
import com.dvt.app.common.log.FileLogger;
……………
FileLogger logger = new FileLogger("foo.txt");
```

Since *Logger* is an interface, it's easy to implement custom logging features such as logging to a database. Also, it's easy to insert additional steps for warnings and errors.

```
public class SampleLogger extends FileLogger {

  public SampleLogger(String fileName) throws FileNotFoundException
  {
    super(fileName);
  }

  public void logError(String message) {
    super.logError( message);

    /**
     * Put your own custom error logic here.  For example:
     *      Email Notification
     *      Operations Console Notification
     */
  }
}
```

Appendix 1: TaskManager Property Settings

This section describes supported TaskManager Property Settings. Default TaskManager property settings can also be found in ${Install Directory}/lib/taskmaster.default.properties.

Property	Default Value	Description
taskmgr.max.tasks	5	Maximum number of tasks
taskmgr.max.tasktime	3600000	Maximum allowed task time in Millis
taskmgr.diagnostic.interval	3600000	TaskManager Diagnostic report interval in nillis
taskmgr.log.file	Not set	Log File -- Omitting this property will cause standard output to be used.
taskmgr.thread.priority	5	Thread Priority used for individual tasks. Defined by java.lang.Thread. Omitting this property will cause Thread.NORM_PRIORITY to be used. It is recommended that you let this property default.

A sample properties file follows:

```
######################################################
#
#   TaskManager Default Properties
#
#   This properties file contains default properties for Threadworks.
#
#   (c) Delta Vortex Technologies, Inc.   2001  All Rights Reserved.
#
######################################################

#  Default number of tasks
taskmgr.max.tasks=5

#  Default Maximum alowed task time in Millis
taskmgr.max.tasktime=3600000

#  Default TaskManager Diagnostic report interval in nillis
taskmgr.diagnostic.interval=3600000

#  Default Log File -- Omitting this property will cause standard output to be
used.
#taskmgr.log.file=taskmanager.log

#  Default thread Priority -- Defined by java.lang.Thread
#       *Omitting this property will cause Thread.NORM_PRIORITY to be used.
#       *It is recommended that you let this property default.
#taskmgr.thread.priority=5
```

Appendix 2: Scheduler Property Settings

This section describes supported Scheduler Property Settings. As Scheduler uses a TaskManager, all properties for the Scheduler's task manager are configurable as well. Default Scheduler property settings can also be found in ${Install Directory}/lib/scheduler.default.properties.

Property	Default Value	Description
scheduler.spin.count	50	Scheduler spin count (how often Scheduler looks for tasks to execute) in millis
taskmgr.max.tasks	5	Maximum number of tasks
taskmgr.max.tasktime	3600000	Maximum allowed task time in Millis
taskmgr.diagnostic.interval	3600000	TaskManager Diagnostic report interval in nillis
taskmgr.log.file	Not set	Log File -- Omitting this property will cause standard output to be used.
taskmgr.thread.priority	5	Thread Priority used for individual tasks. Defined by java.lang.Thread. Omitting this property will cause Thread.NORM_PRIORITY to be used. It is recommended that you let this property default.

A sample properties file follows:

```
######################################################
#
#   Scheduler Default Properties
#
#   This properties file contains default properties for the Scheduler.
#
#   (c) Delta Vortex Technologies, Inc.   2001  All Rights Reserved.
#
######################################################

#  Default scheduler spin count in millis
scheduler.spin.count=50

#  Default number of tasks
taskmgr.max.tasks=2

#  Default Maximum alowed task time in Millis
taskmgr.max.tasktime=3600000

#  Default TaskManager Diagnostic report interval in nillis
taskmgr.diagnostic.interval=3600000

#  Default Log File -- Omitting this property will cause standard output to be
used.
#taskmgr.log.file=taskmanager.log

#  Default thread Priority -- Defined by java.lang.Thread
#       *Omitting this property will cause Thread.NORM_PRIORITY to be used.
#       *It is recommended that you let this property default.
#taskmgr.thread.priority=5
```