Timer and Work Manager for Application Servers

International Business Machines Corp. and BEA Systems, Inc.
Version 1.1
May, 2004

Authors

John Beatty, BEA Systems, Inc.
Chris D Johnson, IBM Corporation
Revanuru Naresh, BEA Systems, Inc
Billy Newport, IBM Corporation
Andy Piper, BEA Systems, Inc.
Stephan Zachwieja, BEA Systems, Inc

Status of this Document

This specification may change before final release and you are cautioned against relying on the content of this specification. IBM and BEA are currently soliciting your contributions and suggestions. Licenses are available for the purposes of feedback and (optionally) for implementation.

Introduction

The Timer and Work Manager for Application Servers specification provides a concurrent programming API for use within managed environments on the Java™ platform, such as Servlets and EJBs.

This specification is organized as follows:

- **Version Updates** describes the changes since version 1.0 of the specification.
- **Architecture** describes the design of the specification.
- **Deployment** discusses how Timers and Work Managers are configured by deployment descriptors.
- **Examples** provides a series of examples showing common usages of the Timer and Work Manager API
- The Java API is provided as Javadocs in a separate file

Timer API

The Timer API enables applications to schedule future timer notifications and receive timer notification callbacks to an application-specified listener.

When inside these managed environments, this API is a much better alternative to `java.util.Timer`: `java.util.Timer` should never be used within managed environments, as it creates threads outside the purview of the container. Further, there is no clean way of subclassing `java.util.Timer` to avoid thread creation, as all constructors create and start a thread. This API is also a better choice than using the JMX Timer Service because the JMX Timer Service API is tightly coupled with the JMX framework and thus does not provide a sufficiently user-friendly or independent API.

Work Manager API

The Work Manager service provides a high-level programming model that enables applications to logically execute multiple work items concurrently under the control of the container. In essence, the work manager provides a container-managed alternative to using the `java.lang.Thread`, which is inappropriate for use within applications hosted in managed environments.

The Work Manager API enables a number of common use cases:

- A Servlet or JSP needs to aggregate data from various sources and render an HTML page after all the data has been retrieved. In this case, the Work Manager

API could be used to retrieve the data in parallel and allow execution to continue once all the data is ready.
- An EJB needs a result from any one of several network services in order to complete its task. The EJB can use the Work Manager API to initiate concurrent requests to the network services and continue execution once one of the services has completed.

When inside managed environments, this Work Manager API is a much better alternative to `java.lang.Thread`, as `Thread` should never be used by application-level code within managed environments as the container needs full visibility and control over all executing threads. Also, this Work Manager API is a better alternative than the J2EE Connector Architecture 1.5 [1] Work Service, as the JCA Work Service is tightly coupled with the JCA framework and thus does not provide a sufficiently independent API for use outside JCA. In particular, the JCA `javax.resource.spi.work.WorkManager` interface exposes methods taking `javax.resource.spi.work.ExecutionContext`, which is not generally the context mechanism that should be used by J2EE applications.

The Timer and Work Manager for Application Servers specification thus provides a clean, simple, and independent API that is appropriate for use within any J2EE container.

Version Updates

Version 1.1 of the Timer and Work Manager for Application Servers specification merges the former Timer Specification for Application Servers and Work Manager Specification for Application Servers specifications and resolves a few minor API problems and clarifies behaviors.

TimerManager

Each method was changed to explicitly declare the runtime exceptions that can be thrown and the JavaDocs were clarified the difference between fixed-rate and fixed-delay timers.

The `suspend` and `stop` methods were changed to no longer block until the timer listeners are completed. Both `stop` and `suspend` will now return immediately.

New methods were added to allow tracking the state of timer listeners once the `suspend` or `stop` methods are issued.

Timer

The `scheduledExecutionTime` method name was changed to `getScheduledExecutionTime`.

Each method was changed to explicitly declare the runtime exceptions that can be thrown and the JavaDocs were clarified to describe the `getScheduledExecutionTime` and `getPeriod` methods.

WorkEvent

The `getWork` method was removed and replaced with the `getWorkItem` method to allow the `WorkListener` to correlate the event with a specific `WorkItem`.

WorkManager

The `waitForAll` and `waitForAny` methods were updated to throw a `java.lang.InterruptedException`. This is required to signal the caller that the wait has been interrupted. These methods will also now throw a `java.lang.IllegalArgumentException` if the `Collection` is null or timeout is negative.

The `waitForAny` method now returns an empty Collection object instead of a null.

WorkItem

The `getResult()` method was added and the interface changed to extend Comparable.

RemoteWorkItem

The `getResult()` method was removed and added to the `WorkItem` super interface.

Timer Architecture

The Timer API is comprised of three primary interfaces: `TimerManager`, `Timer`, and `TimerListener`. Applications use a `TimerManager` to schedule `TimerListeners`. Each of the TimerManager schedule methods returns a `Timer` object. The returned `Timer` can then be queried and/or cancelled. Applications are required to implement the `TimerListener` interface and may optionally implement one or both of the `CancelTimerListener` and `StopTimerListener` interfaces. When a timer expires, the `timerExpired()` method on the provided `TimerListener` instance is executed. This execution is always in the same JVM as the thread that scheduled the timer with the `TimerManager`. `TimerManager` provides a set of `schedule()` and `scheduleAtFixedRate()` methods which take a `TimerListener` instance along with other parameters (including absolute first execution time, relative delays before first execution, and execution periods) and returns a `Timer` instance.

It is important to note the difference between *fixed-delay* execution, provided by the series of `schedule()` methods that take a `period` parameter, and *fixed-rate* execution, provided by the series of `scheduleAtFixedRate()` methods. Fixed-delay means that the `period` parameter specifies the time between actual execution time of the last `timerExpired()` method call. If the `timerExpired()` call was delayed for any reason (e.g., TimerManager suspension, garbage collection or other background activity), this is taken into account. This is contrasted by fixed-rate execution, which tries to keep `timerExpired()` "caught up" and on schedule. Thus, under fixed-rate execution, the actually time interval between `timerExpired()` executions may be much smaller than the specified period.

The `Timer` instance returned by the `TimerManager` can be used to manipulate the timer (e.g., cancel, determine time to next execution, etc.).

A managed environment can support an arbitrary number of independent `TimerManager` instances. The common method for obtaining a `TimerManager` instance is through a JNDI lookup to the local Java environment (i.e., `java:comp/env/timer/[timername]`). Thus, Timer Managers are configured at deployment time through deployment descriptors, and may be further configured through implementation-specific management features. Each JNDI `lookup()` for a `TimerManager` returns a *new* logical instance of `TimerManager`. Thus, applications need to cache copies of `TimerManager` if they intend to reuse the same instance. `TimerManager` is thread-safe.

This specification places no requirements on persistence of timers: if the managed environment is shut down or fails, the timers will be irrevocably lost unless the implementation supports a higher quality of service.

`TimerManager` may also be suspended and resumed via the `suspend()` and `resume()` methods. When a `TimerManager` is suspended, all pending timers are deferred until the `TimerManager` is resumed and all in-flight `TimerListeners` are allowed to complete.

`TimerManager` can also be destroyed via the `stop()` method. After `stop()` has been called, all `Timers` will be stopped and the `TimerManager` instance will never expire another timer.

Timer Interface

The `Timer` interface, instances of which are returned when timers are scheduled with the `TimerManager`, provides several capabilities:

- `cancel()`: Cancels the timer that is currently pending. If the listener associated with this timer implements the `CancelTimerListener` interface, the listener will be notified via the `timerCancel()` callback.
- `getPeriod()`: This returns the period that is used to compute the next time the timer will expire.
- `getScheduledExecutionTime()`: This returns the absolute time in milliseconds that the timer is scheduled to expire. If this method is executed while the associated TimerListener execution is in progress, this value will be that of the current TimerListener execution.
- `getTimerListener()`: Returns the `TimerListener` associated with the timer.

Timer Listener Interfaces

The base `TimerListener` interface provides the `timerExpired()` callback. It is anticipated that this is sufficient for many applications. However, additional callbacks for timers being cancelled and TimerManagers being stopped are sometimes necessary. Listener classes can implement `CancelTimerListener` if they want the `timerCancel()` callback in the case that the application cancels a `Timer`. Listener classes can implement the `StopTimerListener` if they want the `timerStop()` callback in the case that the `TimerManager` on which the `Timer` was scheduled is stopped. Listener classes can also implement both `CancelTimerListener` and `StopTimerListener` if desired.

Work Manager Architecture

The Work Manager API is comprised of six primary interfaces: `WorkManager`, `Work`, `WorkItem`, `RemoteWorkItem`, `WorkListener`, and `WorkEvent`. The `WorkManager` interface provides a set of `schedule`() methods whereby `Work` can be scheduled for execution. The `WorkManager` then returns a `WorkItem`, which can be used to get the status of the in-flight work. The `WorkManager` executes the scheduled work using an implementation-specific strategy. Most implementations will use thread pools. Configuration of `WorkManager` thread pools or other resources is vendor-dependent.

A managed environment can support an arbitrary number of independent `WorkManager` instances. The primary method for obtaining a `WorkManager` instance is through a JNDI lookup to the local Java environment (i.e., `java:comp/env/wm/[work manager name]`). Thus, Work Managers are configured at deployment time through deployment descriptors as `resource-refs` (see **Deployment** below). Each JNDI `lookup()` of a specific `WorkManager` (e.g. `wm/MyWorkManager`) returns a shared instance of that `WorkManager`. `WorkManager` is a thread-safe.

This specification places no requirements on persistence of in-flight `Work`: if the managed environment is shut down or fails, the work will be irrevocably lost unless the particular implementation in use supports a higher quality of service.

Work Lifecycle

`Work` objects can be defined as long-lived (daemon) by returning a value of true from the isDaemon() method. Daemon `Works` can outlive the Servlet request or EJB method that scheduled it, but will automatically be released when the application is stopped. These `Works` do not use a thread from a pool.

Short-lived, non-daemon `Works` are allocated from a thread pool. Normally, short-lived `Works` should complete before the submitting Servlet request or EJB method terminates. The `WorkManager.waitForAll()` method can be used to wait for the `Works` to complete, or the Works can be released using the `Work.release()` or `RemoteWorkItem.release`() methods. Short-lived `Works` may exceed the the life of the submitting request method as long as the `Work` doesn't utilize resources that are tied to the method's duration. For example, a `javax.servlet.ServletResponse` object is only valid during the request and is invalid after the request completes.

Remote Execution of Work

The Work Manager API supports, but by no means mandates, implementation strategies whereby `Work` can be executed in a JVM that is remote with respect to the JVM on which the `WorkManager` is executing. Implementations may choose to farm out `Work` to remote JVMs when the underyling platform is a parallel architecture and supports high-speed communication between JVMs, for example.

If a `Work` instance that is scheduled on a `WorkManager` implements `java.io.Serializable`, this indicates to the `WorkManager` that remote execution (in a separate JVM) of that `Work` is possible. In this case, the `WorkManager` returns a `RemoteWorkItem`, and thus the client can reliably downcast from `WorkItem` to `RemoteWorkItem`. Note that many implementations of `WorkManager` will execute the `Work` locally even if the `Work` instance implements `java.io.Serializable`.

If the client's `Work` instance implements `java.io.Serializable`, the client must not rely on the `Work` instance submitted to the `WorkManager` to be current as it may be executing remotely. Rather, the client should use the `getResult()` method on the `RemoteWorkItem`. This returns the `Work` instance after it has been deserialized from remote execution. Note that in some implementations, the `Work` instance submitted to the `WorkManager` may be fresh, but this is not guaranteed behavior.

Work Listener

A `WorkListener` can be specified when work is being scheduled. The `WorkManager` will call back on `WorkListener` for various work events (e.g. accepted, rejected, started, completed).

`WorkListener` instances are always executed in the same JVM as the thread that scheduled the `Work` with the `WorkManager`.

Waiting for Completion of Work

`WorkManager` also provides simple APIs for common join tasks. `WorkManager` provides two semantics:

- `waitForAll()`: blocks until all specified `WorkItem`s complete, or until the specified timeout. Returns `true` if all items completed within the specified timeout value, and `false` otherwise.
- `waitForAny()`: blocks until any of the specified `WorkItem`s complete until the specified timeout and returns the Collection of completed `WorkItem`s. If no `WorkItem`s completed within the specified timeout, `an empty Collection` object is returned.

Two special timeout values are defined:

- `WorkManager.INDEFINITE`: Waits indefinitely for all/any of the work to complete.
- `WorkManager.IMMEDIATE`: Checks the current state for all/any of the work to complete and returns immediately.

Timer Deployment

Applications signal their need for a timer manager through including a `resource-ref` in the appropriate deployment descriptor (e.g., web.xml, ejb-jar.xml, ra.xml, etc.). The suggested name prefix for the JNDI namespace for `TimerManager` objects is `java:comp/env/timer`.

The following provides an example resource-ref fragment configuring a `TimerManager` named `MyTimer`:

```
<resource-ref>
  <res-ref-name>timer/MyTimer</res-ref-name>
  <res-type>commonj.timer.TimerManager</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Unshareable</res-sharing-scope>
</resource-ref>
```

Work Manager Deployment

Applications signal their need for a work manager by including a resource-ref in the appropriate deployment descriptor (e.g., web.xml, ejb-jar.xml, ra.xml, etc.). The suggested name prefix for the JNDI namespace for `WorkManager` objects is `java:comp/env/wm`.

The following provides an example resource-ref fragment configuring a WorkManager named MyWorkManager:

```
<resource-ref>
  <res-ref-name>wm/MyWorkManager</res-ref-name>
  <res-type>commonj.work.WorkManager</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

Timer Examples

The following example shows a `TimerManager` being looked up in JNDI and used to schedule a timer that fires in 60 seconds.

```
InitialContext ctx = new InitialContext();
TimerManager mgr = (TimerManager)
      ctx.lookup("java:comp/env/timer/MyTimer");
TimerListener listener =
      new StockQuoteTimerListener("QQQ", "johndoe@example.com");
```

```
// schedule timer to expire 60 seconds from now
mgr.schedule(listener, 1000*60);
```

The above code relies on the `StockQuoteTimerListener` class, which could be defined as follows:

```
import commonj.timers.Timer;
import commonj.timers.TimerListener;

public class StockQuoteTimerListener implements TimerListener {
	private String ticker;
	private String email;

	public StockQuoteTimerListener(String ticker, String email) {
		this.ticker = ticker;
		this.email = email;
	}

	public void timerExpired(Timer timer) {
		// retrieve stock quote for ticker and
		// email quote to recipient
		System.out.println("sent stock quote for " +
			ticker + " to " + email);

		System.out.println("timer will fire again: " +
			timer.getScheduledExecutionTime());
	}
}
```

The TimerManager allows other fixed-delay schedule methods, as shown below:

```
// schedule timer to expire 60 seconds from now
mgr.schedule(listener, 1000*60);

// schedule timer to expire 60 seconds from now
// and repeat every 30 seconds
mgr.schedule(listener, 1000*60, 1000*30);

// schedule timer to expire at noon today
Calendar cal = Calendar.getInstance();
cal.set(Calendar.HOUR, 12);
mgr.schedule(listener, cal.getTime());

// schedule timer to expire at noon today
// and repeat every hour thereafter
cal = Calendar.getInstance();
cal.set(Calendar.HOUR, 12);
mgr.schedule(listener, cal.getTime(), 1000*60*60);
```

The `scheduleAtFixedRate()` method can also be used:

```
// schedule timer to expire 60 seconds from now
```

```
// and repeat every 30 seconds
mgr.scheduleAtFixedRate(listener, 1000*60, 1000*30);

// schedule timer to expire at noon today
// and repeat every hour thereafter
cal = Calendar.getInstance();
cal.set(Calendar.HOUR, 12);
mgr.scheduleAtFixedRate(listener, cal.getTime(), 1000*60*60);
```

The following shows an example listener class similar to the previous listener class, but it implements both `StopTimerListener` and `CancelTimerListener`:

```
import commonj.timers.CancelTimerListener;
import commonj.timers.StopTimerListener;
import commonj.timers.Timer;

public class StockQuoteTimerListener2
        implements StopTimerListener, CancelTimerListener {

        private String ticker;
        private String email;

        public StockQuoteTimerListener2(String ticker, String email) {
                this.ticker = ticker;
                this.email = email;
        }

        public void timerStop(Timer timer) {
                System.out.println("Timer stopped: " + timer);
        }

        public void timerCancel(Timer timer) {
                System.out.println("Timer cancelled: " + timer);
        }

        public void timerExpired(Timer timer) {
                // retrieve stock quote for ticker and
                // email quote to recipient
                System.out.println("sent stock quote for " +
                        ticker + " to " + email);

                System.out.println("timer will fire again: " +
                        timer.getScheduledExecutionTime());
        }

}
```

Here is an example deployment descriptor that configures the `TimerManager` used above:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app …>
```

```
  <display-name>A Simple Application</display-name>
  <servlet>
    <servlet-name>OrderTracking</servlet-name>
    <servlet-class>com.mycorp.OrderTracking</servlet-class>
  </servlet>
  <resource-ref>
    <res-ref-name>timer/MyTimer</res-ref-name>
    <res-type>commonj.timer.TimerManager</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Unshareable</res-sharing-scope>
  </resource-ref>
</web-app>
```

Work Manager Examples

The following example shows a `WorkManager` being looked up in JNDI and used to schedule work:

```
import commonj.work.*;
...
RetrieveDataWork work1 =
      new RetrieveDataWork(new URI("http://www.example.com/1"));
RetrieveDataWork work2 =
      new RetrieveDataWork(new URI("http://www.example.com/2"));
InitialContext ctx = new InitialContext();
WorkManager mgr = (WorkManager)
      ctx.lookup("java:comp/env/wm/MyWorkManager");
WorkItem wi1 = mgr.schedule(work1);
WorkItem wi2 = mgr.schedule(work2);
```

This example uses a `RetrieveDataWork` class, which is a fictitious worker classes that retrieves data from a resource specified by a URI:

```
public class RetrieveDataWork implements Work {
      private URI uri;
      private String data;

      public RetrieveDataWork(URI uri) {
            this.uri = uri;
      }

      public void release() {
            // release my resources
      }

      public boolean isDaemon() {
            return false;
      }

      public void run() {
            // do the actual work here
            data = "Hello, World";
      }
```

```
	public String getData() {
		return data;
	}

	public String toString() {
		return "RetrieveDataWork(" + uri + ")";
	}
}
```

The following example shows an example deployment descriptor for a Servlet that configures the `WorkManager` used above.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app …>
  <display-name>A Simple Application</display-name>
  <servlet>
    <servlet-name>OrderTracking</servlet-name>
    <servlet-class>com.mycorp.OrderTracking</servlet-class>
  </servlet>
  <resource-ref>
    <res-ref-name>wm/MyWorkManager</res-ref-name>
    <res-type>commonj.work.WorkManager</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
  </resource-ref>
</web-app>
```

The following example, building on the prior example, shows how the application can block waiting for these work items to complete:

```
// block until all items are done
Collection coll = new ArrayList();
coll.add(wi1);
coll.add(wi2);
mgr.waitForAll(coll, WorkManager.INDEFINITE);
```

Once the application knows that work is completed, the data can be retrieved from the `Work` object:

```
System.out.println("work1 data: " + work1.getData());
System.out.println("work2 data: " + work2.getData());
```

The next example is a slight variation on the example above: the application blocks waiting for *any* of the items to complete. `waitForAny()` returns the `WorkItem`(s) that completed, at which point we can extract the result and continue:

```
String result = null;
Collection coll = new ArrayList();
coll.add(wi1);
coll.add(wi2);
Collection finished = mgr.waitForAny(coll, WorkManager.INDEFINITE);
```

```
if(finished.size() != 0) {
	Iterator i = finished.iterator();
	if(i.hasNext()) {
		WorkItem wi = (WorkItem) i.next();
		if(wi.equals(wi1)) {
			result = work1.getData();
		} else if(wi.equals(wi2)){
			result = work2.getData();
		}
	}
}
```

Alternatively, the `WorkItem` can be used directly to get the resulting `Work` object to avoid correlating the objects. This is always necessary if the `Work` was executed remotely since the result needs to be serialized back to the submitter. `Work` objects can optionally be executed remotely if the `Work` implements `Serializable`.

```
// block until any of the items are done
String result = null;
Collection coll = new ArrayList();
coll.add(wi1);
coll.add(wi2);
Collection finished = mgr.waitForAny(coll, WorkManager.INDEFINITE);
Iterator i = finished.iterator();
if(i.hasNext()) {
	RemoteWorkItem wi = (RemoteWorkItem) i.next();
	RetrieveDataWork work = (RetrieveDataWork) wi.getResult();
	result = work.getData();
}
```

The application can also check the status of the `WorkItem` instances at any time:

```
if(wi1.getStatus() == WorkEvent.WORK_COMPLETED) {
	System.out.println("wi1 completed");
}
```

When scheduling work with a `WorkManager`, a `WorkListener` can be used. To use a `WorkListener`, a concrete class first needs to be defined that implements the `WorkListener` interface. This example illustrates how the listener can use a synchronized `TreeMap` to correlate `WorkItem` to `Work` objects:

```
import commonj.work.Work;
import commonj.work.WorkEvent;
import commonj.work.WorkItem;
import commonj.work.WorkListener;

public class ExampleListener implements WorkListener {
    protected java.util.Map workMap =
        java.util.Collections.synchronizedMap(new java.util.TreeMap());
```

```
    public void workAccepted(WorkEvent we) {
        System.out.println("Work Accepted: " + getWork(we.getWorkItem()));
    }

    public void workRejected(WorkEvent we) {
        System.out.println("Work Rejected: " + removeWork(we.getWorkItem()));
    }

    public void workStarted(WorkEvent we) {
        System.out.println("Work Started: " + getWork(we.getWorkItem()));
    }

    public void workCompleted(WorkEvent we) {
        System.out.println("Work Completed: " + removeWork(we.WorkItem()));
    }

    public void addWork(WorkItem wi, Work w) {
        workMap.put(wi, w);
    }

    public Work getWork(WorkItem wi) {
        return (Work)workMap.get(wi);
    }

    public Work removeWork(WorkItem wi) {
        return (Work)workMap.remove(wi);
    }
}
```

Once the listener class is defined, it can be used in conjunction with the `WorkManager`:

```
RetrieveDataWork work1 =
       new RetrieveDataWork(new URI("http://www.example.com/1"));
RetrieveDataWork work2 =
       new RetrieveDataWork(new URI("http://www.example.com/2"));
InitialContext ctx = new InitialContext();
WorkManager mgr = (WorkManager)
       ctx.lookup("java:comp/env/wm/MyWorkManager");
WorkListener listener = new ExampleListener();
WorkItem wi1 =
       mgr.schedule(work1, listener);
listener.addWork(wi1,work1);
WorkItem wi2 =
       mgr.schedule(work2,listener);
listener.addWork(wi2,work2);
```

References

[1] JSR 112, J2EE Connector Architecture 1.5. http://www.jcp.org/en/jsr/detail?id=112

Trademarks