

Symphony WorkManager Administrator's Guide

Version 1.2

January, 2009

Authors

Hongliang Li, Jilin University, China

Shaocheng Xing, Jilin University, China

Ji Li, Jilin University, China

Zane Hu, Platform Computing

Reviewers

Xiaohui Wei, Jilin University, China

Zane Hu, Platform Computing, Canada

| | |
|--|----|
| Symphony WorkManager Administrator's Guide..... | 1 |
| 1 Introduction..... | 1 |
| 1.1 Distributed Thread Pooling..... | 2 |
| 1.2 A JSR237 WorkManager Implementation Using Platform Symphony..... | 3 |
| 2 Setup the Environment..... | 4 |
| 2.1 Install JRE1.5 or Higher Version..... | 4 |
| 2.2 Setup Platform Symphony Environment..... | 4 |
| 2.3 Obtain Commonj.WorkManager Package..... | 5 |
| 3 Symphony WorkManager Package..... | 5 |
| 4 Deploy SymService onto Symphony Cluster..... | 7 |
| 5 Test Symphony WorkManager Application..... | 8 |
| 5.1 Deploy SymWorkManager onto JBoss Server..... | 8 |
| 5.2 Test Symphony and JBoss environment..... | 9 |
| 6 References..... | 11 |

1 Introduction

This document provides a step-by-step guide to install Symphony WorkManager package. And we will also give you a demo of how to design a JSR237 CommonJ WorkManager application using Symphony WorkManager under a Symphony-JBoss environment.

Terms:

CommonJ^[1] Timer and WorkManager for Application Servers specification (JSR237 for J2EE, Java) provides a concurrent programming API for use within managed environments on the Java TM platform, such as Servlets and EJBs.

WorkManager^[2] is a part of the CommonJ specifications. It provides a simple API for application-server-supported concurrent execution of WorkItems. This enables J2EE-based applications (including Servlets and EJBs) to schedule work items for concurrent execution, which will provide greater throughput and increased response time.

Work and **WorkItem**^[2] are interfaces that provided in WorkManager. Work contains the main logic of a user program. CommonJ consumers use WorkManager to schedule Works. WorkItem is returned once a **Work request** is submitted to a WorkManager. It can be used to check the status of Work after it's finished and to gather executing results of the Work.

Symphony Java API^[3] is provided by Platform Symphony for applications to communicating with Symphony Cluster. We use this API to implement CommonJ WorkManager.

Symphony WorkManager is a high performance computing (HPC) implementation of CommonJ WorkManager using Symphony API. It consists of two parts: SymWorkManager and SymService.

SymWorkManager is a front-end client application that implements all kernel functions of WorkManager. It could be deployed on any Symphony client hosts.

SymService: is a typical Symphony service program that runs on Symphony compute hosts under SOAM's management. SymService executes Works and return results to SymWorkManager. It is the back-end service application of Symphony WorkManager.

1.1 Distributed Thread Pooling

Thread Pool is a concurrency programming pattern as shown in Figure 1.

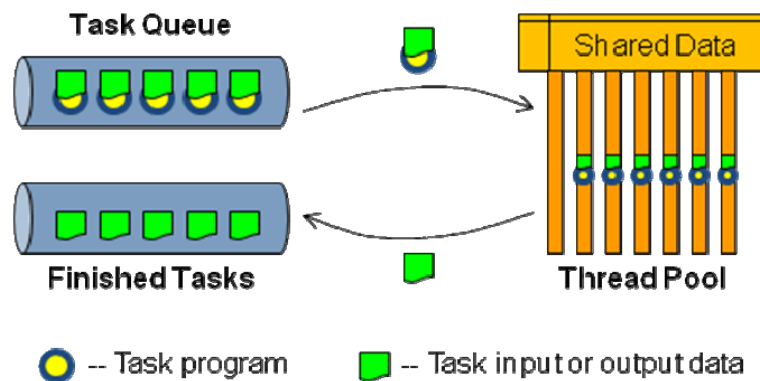


Figure 1: Thread Pool concurrency programming pattern

Here are simple steps when writing a Thread Pool application.

```
// Start a thread pool with a min/max number of threads
myThreadPool.Start(min, max);

// Pack task objects, and enqueue tasks for the pool
myThreadPool.EnqueueTasks();
```

```
// Wait for tasks to finish, and get task output data
myThreadPool.WaitForTasks();

// Stop the pool if no more tasks
myThreadPool.Stop();
```

Thread Pool is easily scalable. Once an application is programmed in this pattern, it can easily scale on a multi-CPU or multi-core computer. The number of threads in a pool can be configured and dynamically adjusted based on the actual task workloads and number of available CPUs or cores.

Not only on a single SMP box, if not much memory or data needs to be shared among threads, this application can also scale out to a server farm of many computers *without* changing the application code, providing the underneath infrastructure supports dynamic distribution of the task execution context, task program, and task input/output data at runtime. Such *distributed thread pooling* is something interesting done by a JSR237 WorkManager implementation described in this document.

1.2 A JSR237 WorkManager Implementation Using Platform Symphony

Unmanaged ad hoc multi-thread execution may either overload or underutilize system resources. It is not recommended and can only be used in unmanaged Java environments like J2SE.

Loosely coupled JMS is designed for easy integration among heterogeneous and distributed applications. Although it could be used as an unmanaged workaround to write concurrent and asynchronous applications, doing so may however be complicated. Application developers have to define message destinations, create messages, implement message beans, and etc. JMS itself does not provide any managed way to control how many workers/actors need to start without overloading or underutilizing system resources, and what messages these workers/actors should work on.

Traditional J2EE application developers are limited to using single-thread execution, because threads in J2EE are container-managed resources like CPUs and DB connections. J2EE application developers should not use threads directly. So, unless seeking for other workarounds, tasks such as generating independent reports can only be executed sequentially, even if they can be done concurrently by multiple threads.

To solve these problems, JSR237 WorkManager provides a specification of concurrency programming APIs for application developers to run concurrent tasks with implicit thread pooling managed by the J2EE container. It enables J2EE-based applications (including Servlets and EJBs) to have the tasks being scheduled by the system for concurrent execution. It can actually be viewed as a JSR of Thread Pool for J2EE applications, although how the thread pool is implemented on the J2EE server side is transparent to the application developers.

This administrator's guide is for setting up and using a concrete system implementation of JSR237

WorkManager, which is done by leveraging Platform Symphony so that distributed thread pooling is supported. Figure 2 shows an overall system architecture of this implementation.

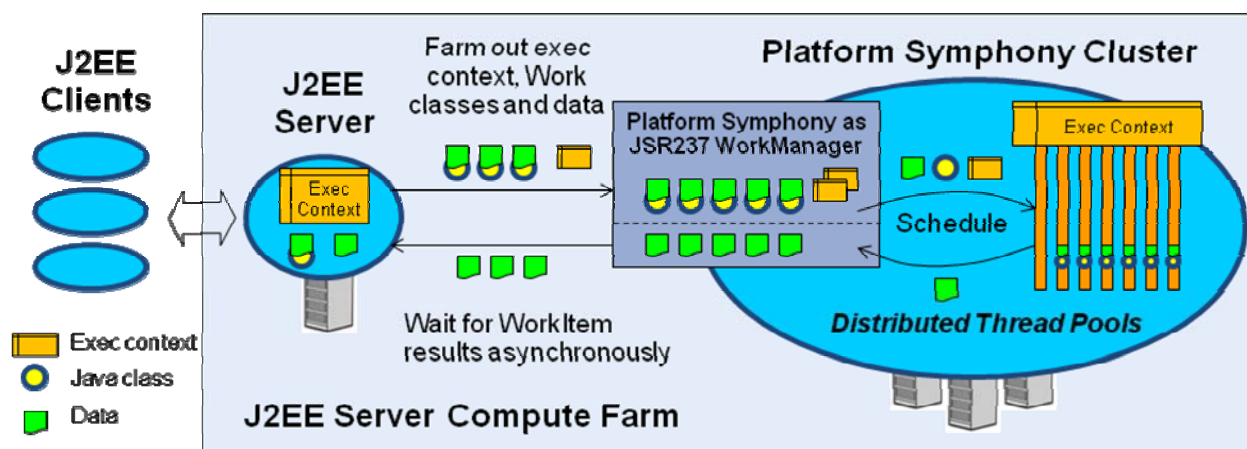


Figure 2: Platform Symphony implementation of JSR237 WorkManager

2 Setup the Environment

This section will describe the software required to run Symphony WorkManager applications:

2.1 Install JRE1.5 or Higher Version

Symphony WorkManager applications are based on java language. JRE is required to be installed as the runtime environment of java language. You can download the JRE installation package from SUN's website for free:

<http://java.cun.com/j2se/downloads/index.html>

Please follow the installation guide to install JRE and setup your environment properly.

2.2 Setup Platform Symphony Environment

Platform Symphony is the computing platform of Symphony WorkManager. Before using Symphony WorkManager, you should prepare a Platform Symphony environment.

In this documentation, we assume that the Platform Symphony environment is installed already. And the SymWorkManager package will be installed on a Platform Symphony client hosts.

Please obtain the Symphony installation package and license by contacting with Platform Computing Inc. (<http://www.platform.com>)

Note: Platform Symphony version 3.1 is used by our developers, and the other versions are not tested

yet. For more information of installation and configuration of Platform Symphony Client please see [sym_client_install_unix.pdf^{\[4\]}](#) and [sym_client_install_windows.pdf^{\[5\]}](#) shipped with the Platform Symphony Client package.

2.3 Obtain Commonj.WorkManager Package

Commonj.WorkManager specification provides a high-level programming model that enables applications to logically execute multiple work items concurrently under the control of the container. The following specification gives an overview of the Commonj model.

CommonJ the BEA and IBM Joint Specifications^[1]

<http://dev2dev.bea.com/wlplatform/commonj/>

The Commonj consists of four sub-specifications in total. Commonj.WorkManager is one of them, please refer to

Commonj-TimerAndWorkManager-Specification-v1.1^[2]

<http://dev2dev.bea.com/wlplatform/commonj/twm.html>

In order to run your SymWorkManager applications, you must have the **commonj-twm.jar** package, which provides the interface definitions of *Commonj.WorkManager*. It can be downloaded from the website above. The other three sub-specifications are not used in this project.

3 Symphony WorkManager Package

Symphony WorkManager is a high performance computing (HPC) implementation of CommonJ WorkManager specification using Symphony Java API. It could provide application improved throughput and reduced response time.

The package is consisted of:

| File name | Description |
|---|---|
| Jar_packages\SymWorkManager.jar | SymWorkManager binary package which contains front-end application and Symphony Java API. This package is used by both system administrators and CommonJ consumers. |
| Jar_packages\commonj-twm.jar | Commonj WorkManager package |
| Jar_packages\jbossall-client.jar | This package is provided by JBoss jar package. Developers can use this package to build JBoss client application. |
| Service_application\upload.zip | SymService application package. This package will be deployed onto Symphony cluster. |
| Service_application\test.xml | Configure file of SymService application |
| Test_programs\AdminTest | Admin test examples for system administrators |
| Test_programs\CommonTestCTX | Client test examples |
| Source files | Source code for the package |
| Documents\SymphonyWorkManager Admin Guide.doc | Admin Guide documentation |

Table 1: Overall package information

Note: Before running your client program, you must make sure that the *classpath* has included the package of SymWorkManager.jar and commonj-twm.jar.

As we can see in the table that Symphony WorkManager consist of two parts: the front-end WorkManager (SymWorkManager) and the service application (SymService). The service application will be deployed onto Symphony cluster. The front-end application can be deployed on any J2EE servers. These servers also have to be Symphony client hosts. The relationship between this two parts and a CommonJ client application is shown as Figure 3 below.

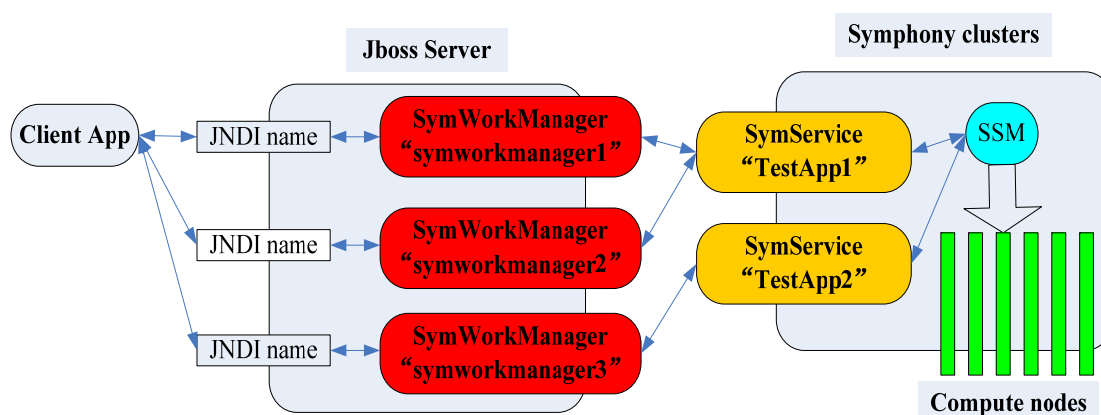


Figure 3: Relationships between client app, SymWorkManager front-end and server app

As shown in the picture, three instances of the front-end application are deployed onto a JBoss server. “TestApp1” and “Test App2” are two instance of SymService that deployed onto Symphony Cluster. A client program can use different SymWorkManager. A SymService can serve multiple front-end SymWorkManager.

4 Deploy SymService onto Symphony Cluster

System administrators can maintain different instances of SymService on Symphony Cluster. The service application package is named upload.zip. This package will be deployed onto Symphony Cluster. The package consists of these three jar files.

| File name | Description |
|----------------------------------|--|
| Symphony-workmanager-service.jar | Symphony WorkManager service package which contains SymService |
| JavaSoamApi.jar | Symphony Java API package |
| commonj-twm.jar | Commonj WorkManager package |

Table 2: Service application package information

Note: Before running your Symphony service program, do make sure that all the Symphony hosts share the same directory path of JRE. In our test, the “JAVA_HOME” is “c:\java”. And the application can find “java.exe” in “c:\java\bin”. This path information can be configured in the configure file of an application.

The deploy process is as follow:

1. Go to the directory that contains the SymService package.

```
cd \your own directory\SymphonyWorkManager\Service application
```

2. Deploy the SymService package with **soamdeploy** command.

```
soamdeploy add TestApp -p upload.zip -c /WorkManagerServices/SymTest
```

The application package is deployed. “TestApp” is the name of the application.

Note: Before deployment, a consumer named /WorkManagerServices/SymTest has to be created in Symphony cluster.

3. Check the list of deployed services with **soamdeploy view** command:

```
soamdeploy view -c /WorkManagerServices/SymTest
```

4. Register SymService configure file with the **soamreg** command:

```
soamreg test.xml
```

The application is registered.

Note: If you want to use your own consumer name and soam application name, you have to edit the configure file like this:

```
applicationName="your app name" consumerId="/your group/your consumer name"
```

5. Check the list of registered applications with the **soamview** app command:

soamview app

You should be able to see an enabled application named “TestApp” on the list now.

After the steps above you have deployed your Symphony WorkManager service application (SymService) onto Symphony cluster successfully.

5 Test Symphony WorkManager Application

5.1 Deploy SymWorkManager onto JBoss Server

SymWorkManager is a front-end client application that implements all kernel functions of WorkManager. It could be deployed on any Symphony client hosts. In this section, we will demo how to use SymWorkManager in Symphony-JBoss environment. We will deploy SymWorkManager instance onto a JBoss server step by step in this section. This computer is also a Symphony client host.

1 Firstly, install and configure a JBoss server on your computer. In our test environment, we use jboss-4.2.2.GA. For more information about JBoss server, please refer to JBoss organization official website:

<http://www.jboss.org/services/index>

2 Then you have to add the control flags in your system environment.

| Variable Name | Values | Description |
|---------------------------|------------------|---|
| When_Remove_ClassFile | Never(default) | Do not remove class files |
| | OnSessionLeave | Remove class files in <i>onSessionLeave()</i> |
| | OnServiceDestroy | Remove class files in <i>OnServiceDestroy()</i> |
| Dynamic_Passing_Class | True(default) | Use dynamic passing class files feature |
| | False | Do not use dynamic passing class files feature |
| Concurrent_Invoke_Enabled | True(default) | Enable concurrent invoke |
| | False | Disable concurrent invoke |

Table 3: Control flags

SymWorkManager will get these flags in its constructor. Also you can use setting and getting functions to configure these flags. Additionally, a function named *refreshEnvironmentFlags()* is provided to reset all these flags to current system settings.

3 Finally, you can use our AdminTest program to deploy a SymWorkManager instance onto the server.

AdminTest package will help you to bind a new SymWorkManager instance onto JBoss JNDI server. In our test, the new SymWorkManager instance is named "symworkmanager1".

AdminTest package contains two class files:

- **AdminClient . NewBind** is an example about how to bind a new SymWorkManager instance to JNDI.
- **AdminClient . ResetVars** is an example about how to reset SymWorkManager's flags.

After these steps, developers can use JNDI *lookup()* function to get the instance of SymWorkManager now.

Note: In order to run AdminTest package, you must have SymWorkManager.jar, commonj-twm.jar and jbossall-client.jar included in the classpath. "jbossall-client.jar" is a jar package provided by JBoss server for developers to build JBoss client applications.

5.2 Test Symphony and JBoss environment

In this section, we will give you an example about how to use the Symphony WorkManager environment. The instance of SymWorkManager is deployed onto JBoss server as discussed in the previous sections. We present you a test package: CommonTestCTX. The files contained in the package are shown below:

| File name | Description |
|---------------------------------------|--|
| UserWork.AddUserWork.java | A user-defined Work to do a addition This Work will read a string from numbers.txt |
| UserWork.MulUserWork.java | A user-defined Work to do a multiplication |
| WorkListener.AddUserWorkListener.java | WorkListener of AddUserWork |
| WorkListener.MulUserWorkListener.java | WorkListener of MulUserWork |
| Test1.SingleClientMulUserWork.java | Simple test of multiple userWork application |
| Test1.AnotherThread.java | A new Thread to schedule AddUserWork and MulUserWork |
| Test1.MulThreadSingleWMMulWork.java | A test that start multi-thread client to use SymWorkManager. Each Thread schedule AddUserWork and MulUserWork. |
| Test2.RunAddUserWork.java | A new Thread to schedule AddUserWork |
| Test2.RunMulUserWork.java | A new Thread to schedule MulUserWork |
| Test2.MulThreadRunMulWork.java | A test that start multi-thread client to use SymWorkManager. One Thread schedule just one kind of userWork. |
| numbers.txt | Input files |

Table 3: Client application package information

In the client programs, users can get SymWorkManager instance from JBoss server using a JNDI name like this:

```
WorkManager swm = null;
Hashtable properties = new Hashtable();
properties.put(Context.INITIAL_CONTEXT_FACTORY,
"org.jnp.interfaces.NamingContextFactory");
properties.put(Context.URL_PKG_PREFIXES,
"org.jboss.naming:org.jnp.interfaces");
properties.put(Context.PROVIDER_URL, "jnp://localhost");
String wmkey = "symWorkManager1";
try {
    InitialContext ctx = new javax.naming.InitialContext(properties);
    swm = (WorkManager) ctx.lookup(wmkey);
    ctx.close();
    if (swm == null)
        System.out.println("swm name is: failed");
    else
        System.out.println("swm name is: " + swm.getClass().getName());
} catch (NamingException e) {
    e.printStackTrace();
}
```

Note: In order to use CommonJ WorkManager, SymWorkManager, and JBoss Server, the client programs have to include commonj-twm.jar, SymWorkManager.jar and jbossall-client.jar in their classpath.

You can test your Symphony WorkManager and JBoss environment using these test programs:

1 Test1.SingleClientMulUserWork.java

This program sends 3 AddUserWork and 3 MulUserWork requests.

Outputs:

```
swm name is: com.platform.soam.commonj.workmanager.SymWorkManager
SingleClientMultiplWork with workID = 3 is accepted by
AddUserWorkListener.
SingleClientMultiplWork with workID = 13 is accepted by
MulUserWorkListener.
wait for any-----
mul Work 11 Result:1
wait for all-----
SingleClientMultiplWork with workID = 13 is completed by
MulUserWorkListener.
SingleClientMultiplWork with workID = 3 is completed by
AddUserWorkListener.1111
add Work 2 Result:4
add Work 2 Result(str):1111
```

```
add Work 3 Result:6
add Work 3 Result(str):1111
mul Work 12 Result:4
Finish successfully
```

2 Test1.MulThreadSingleWMMulWork.java

The program starts 20 Threads by default. Each thread sends 3 AddUserWork and 3 MulUserWork requests. This test sends 120 Work requests.

Output:

```
.....
All works finished successfully!!!
The total result (globalAddUserWorkResult)= 240
The total result (globalMulUserWorkResult)= 280
The work finished in * hours * minutes * seconds.
```

3 Test2.MulThreadRunMulWork.java

This program starts 40 RunAddUserWork and 40 RunMulUserWork Threads by default, each Thread sends 3 Work requests. This test sends 240 Work requests.

Output:

```
... ..
All works finished successfully!!!
The total result (globalAddUserWorkResult)= 480
The total result (globalAddUserWorkResult)= 560
The work finished in * hours * minutes * seconds.
```

Additionally, you can reset the variable “*clientTotal*” in MulThreadSingleWMMulWork and MulThreadRunMulWork to configure the numbers of Threads you want to start.

6 References

[1] CommonJ the BEA and IBM Joint Specifications.

<http://dev2dev.bea.com/wlplatform/commonj>

[2] Commonj-TimerAndWorkManager-Specification-v1.1

<http://dev2dev.bea.com/wlplatform/commonj/twm.html>

[3] Symphony Developer's Guide

[4] sym_client_install_unix.pdf

[5] sym_client_install_windows.pdf