WebSphere XD Compute Grid Parallel Job Manager

System Design & Architecture Document

APPROVED EDITION

Notice of Currency

This documentation is current for WebSphere XD Compute Grid Versions 6.1.0.1 and 6.1.0.2 only. Refer to WebSphere XD Compute Grid Product Wiki for all future levels of Compute Grid at:

http://www.ibm.com/developerworks/wikis/display/xdcomputegrid/Home?S_TACT=105AGX10&S_CMP=ART

TABLE OF CONTENTS

0.0 INTRODUCTION ... 3
0.1 Concepts ... 3
0.1.1 Parallel Job ... 3
0.1.2 Logical Transaction ... 3
0.2 System Programming Interfaces ... 3
1.0 ARCHITECTURE ... 4
2.0 DESIGN ... 6
2.1 xJCL and Naming Conventions ... 6
2.2 Parallelization Control ... 6
2.3 Job Life Cycle ... 8
2.4 Logical Transactions ... 9
2.5 Sub-job Collector-Analyzer ... 11
2.6 Context Objects ... 13
2.7 Operations ... 13
2.8 High Availability ... 14
2.9 Problem Determination ... 14
2.10 Job Restart ... 15
2.11 Disaster Recovery ... 15

.

0.0 Introduction

The Parallel Job Manager, PJM, provides a facility and framework for submitting and managing transactional batch jobs that execute as a coordinated collection of independent parallel sub-jobs.

0.1 Concepts

0.1.1 Parallel Job

In the context of this design specification, a parallel job is a job that is dispatched as multiple jobs (called sub-jobs) executing the same job definition, but each with potentially distinct inputs. All sub-jobs are managed together as a single logical job.

0.1.2 Logical Transaction

A logical transaction is a unit of work demarcation that spans the execution of a parallel job. Its lifecycle corresponds to the combined lifecycle of the parallel job's sub-jobs. An extension mechanism enables customization so that application-managed resources can be controlled in this unit of work scope for commit and rollback purposes.

0.2 System Programming Interfaces

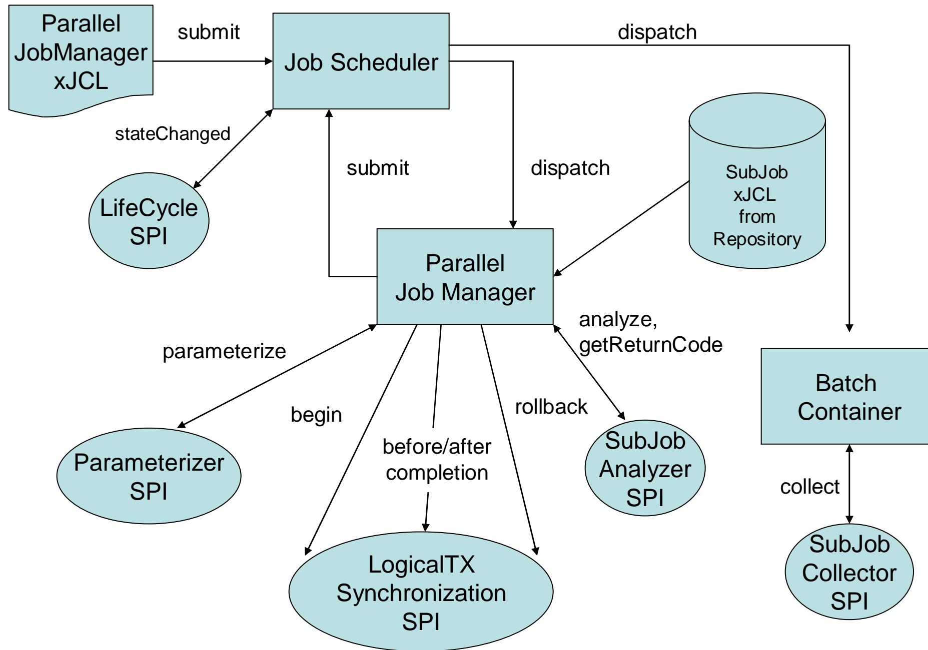
A System Programming Interface (SPI) is a pluggable extension to the execution environment. Compute Grid SPIs are configured through a properties file that identifies which SPIs are installed and what their implementation class names are.

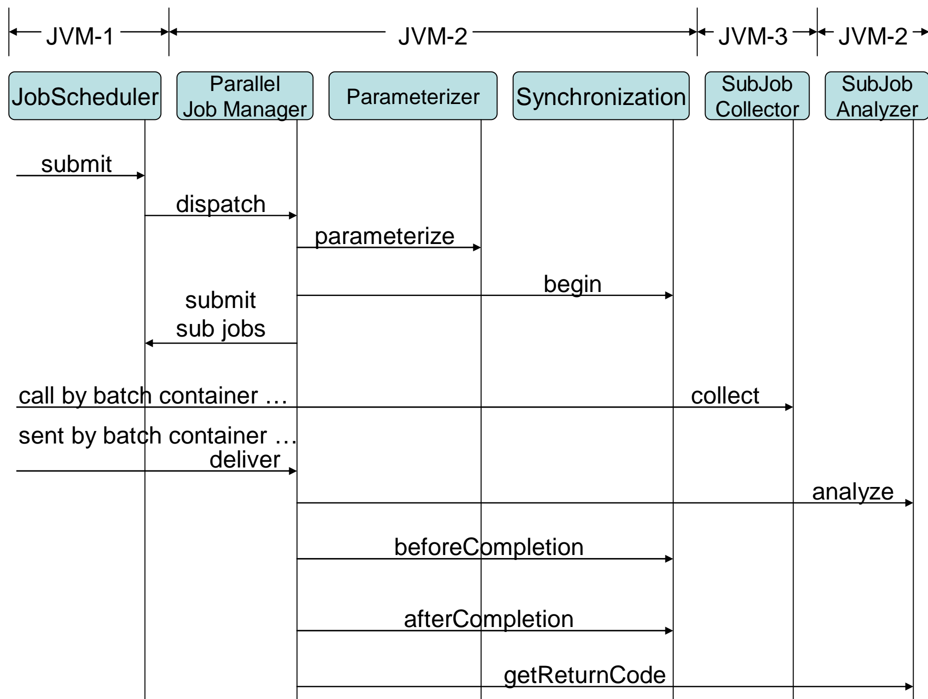
Property file name: xd.spi.properties
Property file location: <WAS install directory>/properties
Property file format: <SPI name>=<SPI implementation class>

Note that all SPIs are instantiated in a WebSphere Application Server as singleton objects.

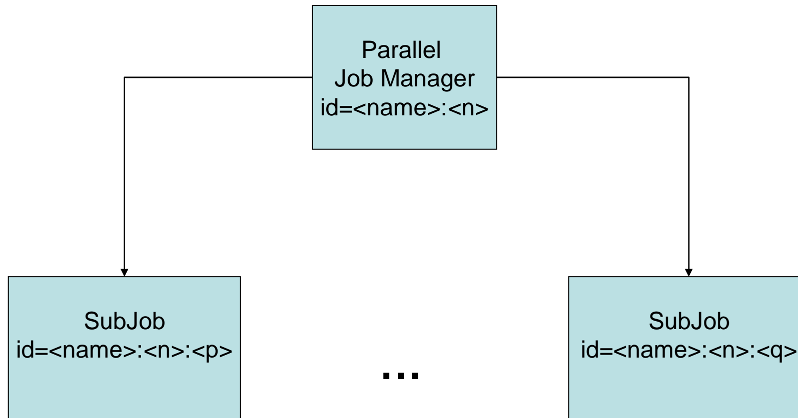
1.0 Architecture

The following diagram summarizes the Parallel Job Manager architecture:





The following diagram depicts the job structure of a parallel job:



Where:

name	= user-given jobname
n	= system generated identifier
p	= system generated identifier, p > n
q	= system generated identifier, q > q

A parallel job is comprised of a job, called the top-level job, that executes the ParallelJobManager application and a set of sub-jobs that execute the actual business logic.

Separate xJCL definitions are required for both the top-level and sub-jobs. All sub-jobs execute using the same xJCL definition; each sub-job instance may be parameterized with distinct substitutions properties.

2.0 Design

2.1 xJCL and Naming Conventions

Jobs have jobnames and jobids. The jobname is defined in the job definition (xJCL) on the name attribute of the job element. A jobid is a unique identifier that represents an instance of a job definition in the job processing system.

The jobname of a parallel job is specified in the xJCL that defines a top-level job. The top-level job xJCL can originate from the file system or from the Compute Grid job repository. The jobname *may* be parameterized using standard substitution property notation:

<job name="${jobname}" … >

A jobid is formed by concatenating a jobname with a system generated sequence number. Jobname and sequence number are colon separated.

The jobname of a sub-job is the parallel job's jobid. The xJCL for a sub-job can originated only from the Compute Grid job repository. Sub-job xJCL ***must*** supply the jobname substitution property:

<job name="${jobname}" … >

Summary:

Parallel job jobname= <jobname>
Parallel job jobid=<jobname>:<n>
Sub-job jobname= <Parallel job jobid>
Sub-job jobid = <Parallel job jobid>:<m>

Where

- jobname is user specified jobname of parallel job.
- n is system-generated sequence number
- m is another system-generated sequence number (m > n)

2.2 Parallelization Control

The system must determine the following for a parallel job:

1. name of sub-job xJCL (from repository)
2. optional logical transaction ID
3. number of sub-jobs
4. input to each sub-job

This information comes from the following sources:

1. Parallel Job xJCL

   The ParallelJobManager defines the following input properties:

   a. com.ibm.wsspi.batch.parallel.subjob.name

      Required: specifies name of sub-job xJCL from repository.

   b. com.ibm.wsspi.batch.parallel.logicalTXID

      Optional: specifies logical transaction ID for a parallel job.

      Important: a logical transaction ID must be unique across all parallel job submissions. User assumes responsibility for ensuring uniqueness.

   Example xJCL snippet:

```
<job name=”${jobname}” default-application-name=”ParallelJobManager” >
    …
    <job-step name=”STEP1” >
        …
        <prop name=” com.ibm.wsspi.batch.parallel.subjob.name”
                value=”<subjob xJCL name from repository>” />
        <prop name=” com.ibm.wsspi.batch.parallel.logicalTXID”
                value=”<user-specified logical transaction ID>” />
        other properties …
        …
    </job-step>
    …
```

2. Parameterizer SPI

   The Parameterizer SPI extends the Compute Grid environment to provide parallelization parameters to the ParallelJobManager.

   The Parameterizer SPI receives the following inputs:

a. Parallel job name
b. Logical transaction id
c. ParallelJobManager step input properties

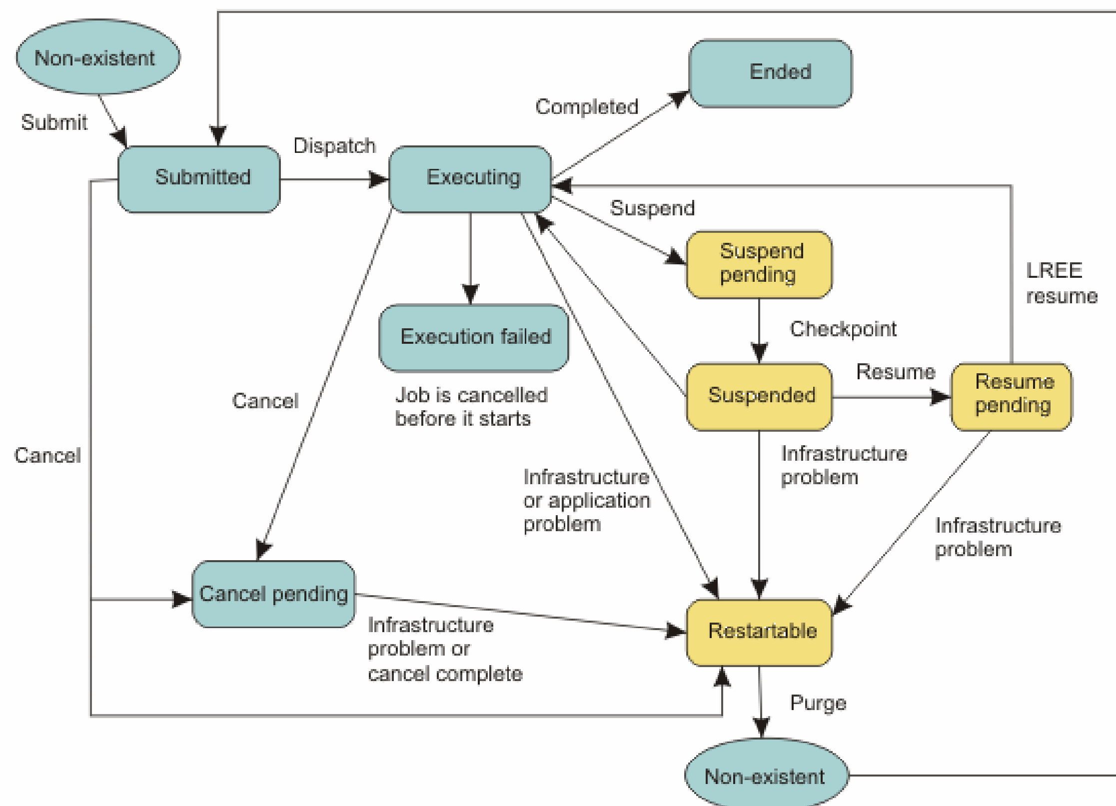
The Parameterizer SPI returns the following outputs:

a. number of sub-jobs to run concurrently for this parallel job
b. Properties object for each subjob, containing substitution properties for subjob xJCL

Configuration details for this SPI is found in the accompanying javadoc.

2.3 Job Life Cycle

Transactional batch jobs execute according to the following state diagram:



Both the top-level job and sub-jobs execute according to this state diagram. The top-level job submits the sub-jobs and monitors their completion. The top-level job end state is influenced by the outcome of the sub-jobs as follows:

1. If all subjobs complete in Ended state (i.e. successful completion), then top-level job will complete in Ended state.

2. If any subjob completes in Restartable state and no subjob has ended in the Failed state, then top-level job will complete in Restartable state.

   Note a job ends in the restartable state due to explicit interruption, such as a cancel or stop command; a job also ends in the restartable state if it ends unexpectedly after processJobStep has received control.

   Note also the JTA transaction of a job that enters restartable state is rolled back, with the exception of a job interrupted by the stop command, since the stop commands waits until the next checkpoint is reached before halting the job.

   The ParallelJobManager's logical transaction is always rolled back. See section 0 Configuration details for this SPI is found in the accompanying javadoc.
3. Logical Transactions for details on logical transaction rollback.

4. If any subjob completes in Failed state the top-level job will complete in Failed state.

   Note a job ends in the Failed state if it fails for any reason before processJobStep is invoked for the first time.

   Note the JTA transaction of a job that ends in the Failed state is always rolledback.

   The ParallelJobManager's logical transaction is always rolled back. See section 0 Configuration details for this SPI is found in the accompanying javadoc.
5. Logical Transactions for details on logical transaction rollback.

The LifeCycle SPI extends the Compute Grid environment to provide callback for extension code that processes job state change events. This SPI has a single method, *stateChanged*, that receives the following inputs:

1. jobid – this uniquely identifies the job, parallel job, or sub-job
2. new state – defined by JobStatusConstants (see javadoc)

Configuration details for this SPI is found in the accompanying javadoc.

2.4 Logical Transactions

A logical transaction provides a unit-of-work scope across all sub-jobs belonging to a parallel job. The logical transaction begins before sub-jobs are submitted and completes after all sub-jobs have ended. The Parallel Job Manager is not a resource manager, so no transactional resources are actually enlisted into a logical transaction. A logical transaction serves only to demarcate the execution of a parallel job's sub-jobs.

The Synchronization SPI extends the Compute Grid environment to enable extension code to process logical transaction lifecycle events. This SPI provides the following callbacks:

1. begin

   The begin callback demarcates the start of a new logical transaction. The input to this callback is a logical transaction ID.

   There is no return value.

2. beforeCompletion

   The beforeCompletion callback is invoked after all sub-jobs have completed. If any sub-jobs terminate unexpectedly, this callback is not invoked, rather only rollBack is invoked. The input to this callback is a logical transaction ID.

   This callback may throw the RollbackLogicalTXException to signal to the ParallelJobManager to rollback the logical transaction.

3. afterCompletion

   The afterCompletion callback is invoked to demarcate the end of a logical transaction. The afterCompletion callback indicates whether the logical transaction has committed or rolledback. This callback is always invoked at the conclusion of a logical transaction, whether or not the beforeCompletion callback has been invoked. The inputs to this callback are:

   a. logical transaction ID
   b. logical transaction status (i.e. outcome): commit or rollback

4. rollBack

   The rollBack callback is invoked as part of logical transaction rollback processing. The input to this callback is a logical transaction ID and a boolean indicating whether or not the parallel job that started this logical transaction is restartable..

   Rollback occurs for the following reasons:

   a. at least one subjob ends in the Restartable or Failed state
   b. beforeCompletion callback throws RollbackLogicalTXException
   c. any of the SubJobAnalyzer SPI analyze methods throws RollbackLogicalTXException (see section 2.5 Sub-job Collector-Analyzer)

   The ParallelJobManager performs the following steps during rollback processing:

a. Cancels all remaining sub-jobs. This includes any sub-job that has not yet been dispatched to an endpoint.

b. Invokes rollBack callback.

   The rollBack method is passed the logical transaction id for the current logical transaction and flag indicating whether the parallel job executing this logical transaction is restartable after rollback processing is complete.

   Consistent with the definitions in section 2.3 Job Life Cycle, a parallel job is restartable if all subjobs complete in either ended and/or restartable states. A parallel job is not restartable if any of its subjobs complete in the failed state.

   The rollback method returns a RestartInstructions object, that indicates whether to:
   a. Restart only subjobs that completed in the restartable state.
   b. Restart all subjobs.
   c. Restart both subjobs that completed in the restartable state and a list of specified subjobs that completed in the ended state.

   If the parallel job is not restartable, the ParallelBatchManager ignores the RestartInstructions object.

c. Ends parallel job with following state

   a. Failed if rollback occurred due to subjob ending in Failed state.
   b. Restartable otherwise.

Configuration details for this SPI is found in the accompanying javadoc.

2.5 Sub-job Collector-Analyzer

The sub-job collector-analyzer SPI extends the Compute Grid environment with a two part mechanism that enables collection of application specific data on the endpoint where a sub-job executes and receipt/analysis of that data at the point of coordination, which is where the ParallelJobManager executes.

The analyzer SPI additionally provides two additional functions:

1. Receives control for each sub-job completion. Sub-job return code is passed as parameter.
2. Provides overall return code for parallel job.

The collector is invoked by the batch container where a sub-job is executing immediately before a checkpoint is taken for that sub-job. This occurs according to the sub-job's checkpoint algorithm class or when the sub-job's processJobStep method forces a checkpoint. If the sub-job checkpoints multiple times, the collector is called before each checkpoint. The collector returns a Java Externalizable for transports to the ParallelJobManager. This transmission may occur immediately; however, the batch container reserves the right to batch transmissions. The Parallel Job Manager ensures all transmissions are received before the beforeCompletion method on the Synchronization SPI is invoked.

The analyzer is called by the ParallelJobManager each time a sub-job collector Externalizable is received and each time a sub-job ends. All Externalizables collected for a sub-job are delivered before the analyzer is notified of sub-job end.

The SubJobCollector's collect method receives the following inputs:

1. parallel job name
2. logical transaction ID
3. subjob jobid

The SubJobCollector collect method returns an Externalizable.

The SubJobAnalyzer methods receive the following inputs:

1. analyze (1)
    a. parallel job name
    b. logical transaction ID
    c. subjob jobid
    d. Externalizable (from collector SPI)
2. analyze (2)
    a. parallel job name
    b. logical transaction ID
    a. subjob jobid
    b. return code (from sub-job)
3. getReturnCode
    a. parallel job name
    b. logical transaction ID

Only the getReturnCode returns a value, an integer which specifies the return code for the parallel job.

Configuration details for these SPIs are found in the accompanying javadoc.

2.6 Context Objects

The Compute Grid runtime will provide context objects that offer a common workarea among SPIs and batch application programming model artifacts. A context object allows user code to save and retrieve a java Object and share it within the context's scope. The Compute Grid runtime will ensure proper cleanup of context objects and end of scope.

There are two context object types:

1. ParallelJobManagerContext

   This context object exists in the scope of a parallel job. The Parameters, SubJobAnalyzer, and Synchronization SPIs all have access to this context for a given parallel job instance.

2. SubJobContext

   This context object exists in the scope of a sub job. The SubJobCollector, and batch application programming model artifacts, BatchDataStream, BatchJobStepInterface, CheckpointPolicyAlgorithm, and ResultsAlgorithm all have access to this context for a given sub job instance.

2.7 Operations

Compute Grid provides a set of operational commands to manage jobs. These commands will work against a parallel job and its sub-jobs as follows:

1. cancel/stop

   Cancel/stop is applied to both the parallel job and its sub-jobs.

2. restart

   Restarting a parallel job will result in a restart of any of its sub-jobs in the Restartable state.

3. suspend

   Suspend is applied to both the parallel job and its sub-jobs. Any sub-job that has not yet been dispatched will be "held" until the parallel job is resumed.

4. resume

Resume is applied to both the parallel job and its sub-jobs. Any “held” sub-job will be “released”.

5. remove

   Remove is applied to both the parallel job and its sub-jobs.

2.8 High Availability

The Compute Grid environment provides high availability for:

1. job scheduler

   Job Scheduler HA is accomplished through clustering. This HA function is insensitive to parallel jobs and remains available to a deployment in which parallel jobs are used.

2. jobs

   Compute Grid supports a failover model for transactional batch jobs (not for compute intensive or native execute type batch applications). Parallel jobs are built on the transactional batch model and are therefore restartable after interruption. When a parallel job is restarted, all sub-jobs in the restartable state are restarted.

2.9 Problem Determination

Problem determination for parallel jobs depends the same mechanism as regular jobs. This includes:

1. job end status
2. job return code
3. job log

These indicators are accessible through the Compute Grid job console, command line, and scheduler APIs.

Additionally, the new LifeCycle SPI allows extension code to receive notification of all job state change events.

The parallel job naming conventions enables correlation of a parallel job and its sub-jobs. This in turn, enables monitoring to – whether human-interactive or automated – make decisions on the basis of the combined state of a parallel job and its sub-jobs.

Examples:

- Job console filtering allows all jobs with jobids of the form <jobname>:<n>* to be viewed together. This would include a parallel job and its sub-jobs.

- LifeCycle SPI and job scheduler APIs can be used in combination to a) catalog parallel job/sub-job jobids and their state changes, and b) use job scheduler APIs to obtain end status, return code, and job log for each job.

2.10 Job Restart

A parallel job maintains persistent state of all its sub-jobs, including sub-job end status. When a parallel job is restarted, all its sub-jobs in the Restartable state are restarted also.

If the top-level job of parallel job ends unexpectedly, it is possible that subjobs are still running and that a logical transaction is in doubt (i.e. not completed). Logical transaction state is logged, therefore upon restart, if the top-level job detects an unresolved logical transaction, it will take the following recovery actions:

1. canceling any of its surviving subjobs
2. rolling back logical transaction

2.11 Disaster Recovery

Compute Grid will support two disaster recovery models:

1. Multi-site Cell

   The multi-site cell relies on basic WebSphere failover, based on application server clusters to provide a cross-site failover capability, using clusters that span sites.

   This model requires high speed communication links between sites in order to support the multi-site model.

   The sites can be operated in hot/hot, hot/warm, or hot/cold modes, following these general guidelines:

   a. Hot/hot

      This mode is automatic by clusters job scheduler and endpoints across sites. By having members of these clusters active on both sites and by sending jobs to

each job scheduler instance, the configuration executes work on both sites.

b. Hot/warm

This mode is enabled by clustering job scheduler end endpoints across both sites, but then only send jobs to primary site. The job scheduler on the other site can be initialized, but not receiving any work.

Endpoint cluster members can be initialized on the standby site and not receiving any work using job scheduling criteria that limits which nodes are eligible to receive jobs. The DR takeover procedure would have to include a step to change this criteria to begin sending jobs to the recovery site.

c. Hot/cold

This mode is enabled by clustering job scheduler and endpoints across both sites, and leaving the jobs scheduler and endpoint cluster members on the backup site in the stopped state. Work would be sent only to the primary site. Upon DR takeover, the takeover procedure would require a step to start the job scheduler and endpoint cluster members on the recovery site.

2. Multi-domain

The multi-domain model uses a separate WebSphere cell on each site. The XD Compute Grid domain exists within a WebSphere cell, hence there would be at least one Compute Grid (aka job scheduling) domain on each site. The two domains must be clones of one another – same topology, same-named WebSphere administrative artifacts. The multi-domain model supports only only hot/cold operational mode. A shared or replicated database may be used.

Hot/cold operational mode works in the multi-domain model by running a job scheduling domain on the primary site and then enacting a DR takeover procedural step in order to activate the backup site. The essential steps to carry this out are:

a. Ensure primary job scheduling domain is down (i.e. stop any servers still running)
b. Execute "job scheduler take over" script – to be provided by IBM This script updates the job scheduler database to mark the jobs left in the "executing" state on the failed primary site as "restartable", so the job scheduling domain on the backup site can take over and restart them.

The multi-domain model *can* provide a form of hot/hot operation, by provide two cell pairs that span the two sites using a primary/backup, backup/primary organization, meaning that cell pair 1 has its primary cell on site 1 and its backup on site 2, whereas cell pair 2 has its primary cell on site 2 and its backup on site 1.