How to split and sync a dynamic number of job instances for execution with Agents

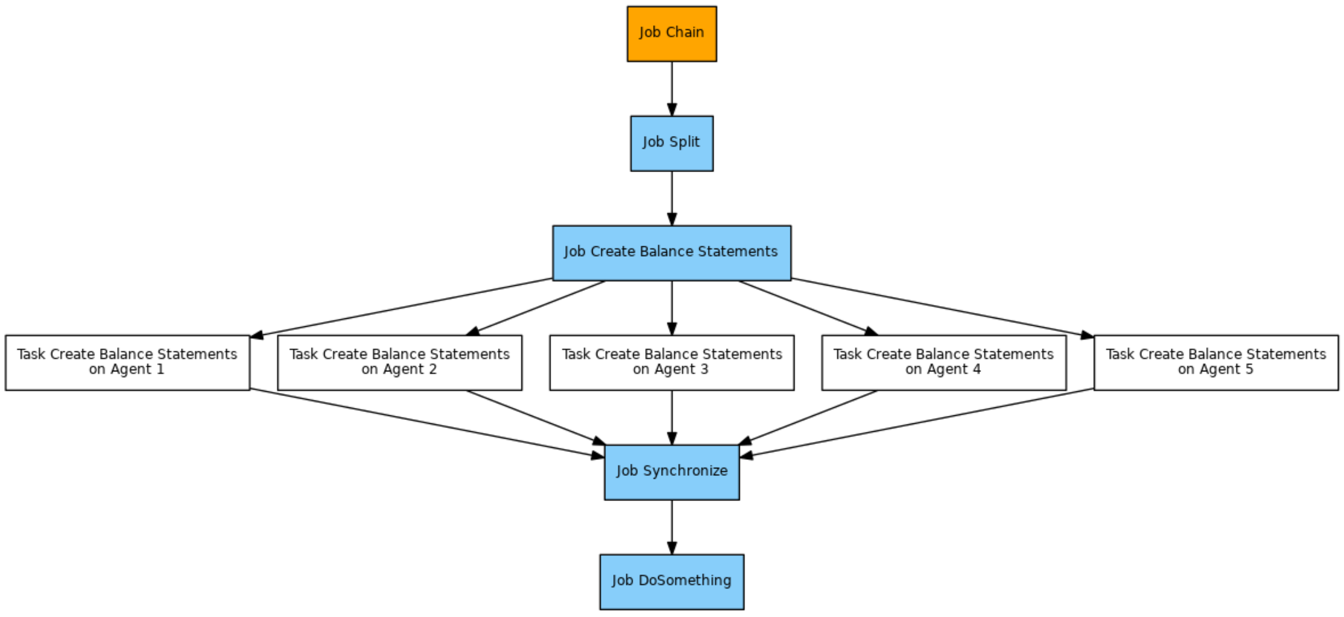
Scope

- Use Case
    - Run a specific job within a job chain in a number of parallel instances, i.e. split a job for parallel processing. Each instance is executed on a different Agent.
        - The use case is intended for execution of the same job within a job chain in parallel instances, not for parallel execution of different jobs.
        - We assume am application for end of day processing that creates balance statements for a larger number of accounts. The application is installed on a number of servers each equiped with an Agent. The processing is splitted in a way that each instance of the application processes a chunk of accounts. The chunk size is calculated from the available Agents, not just from configured Agents.
        - A similar use case is explained with the How to split and sync a dynamic number of job instances in a job chain article, however, this use case is about using a different Agent for each parallel process instead of running parallel processes on the same Master or Agent.
    - Synchronize jobs after parallel processing.
- Solution Outline
    - The number of parallel job instances is dynamically determined from the number of Agents that are available to run the application.
    - A number of orders is created for a specific job that would be executed in parallel on different Agents. The number of orders corresponds to the number of Agents and each order is parameterized to process a chunk of accounts.
    - Finally the parallel job instances are synchronized for further processing.
- References
    - How to split and sync jobs in a job chain
    - How to split and sync a dynamic number of job instances in a job chain

Solution

- Download parallel_job_instances_with_agents.zip
- Extract the archive to a folder `./config/live` of your JobScheduler installation.
- The archive will extract the files to a folder `parallel_job_instances_with_agents.`
- You can store the sample files to a any folder as you like, the solution does not make use of specific folder names or job names.

Pattern



Job Chain
Job Split
Job Create Balance Statements
Task Create Balance Statements on Agent 1
Task Create Balance Statements on Agent 2
Task Create Balance Statements on Agent 3
Task Create Balance Statements on Agent 4
Task Create Balance Statements on Agent 5
Job Synchronize
Job DoSomething

Implementation

Components

- The `end_of_day_split` job reads the process class configuration that is assigned to its successor job which is the `create_balance_statements` job. It creates the number of orders that corresponds to the number of Agents.
    - Each order is added the following parameters:
        - `number_of_orders`: the number of orders that have been created.
        - `<job_chain_name>_required_orders`: the number of orders that the `end_of_day_sync` job waits for. This includes the value of the `number_of_orders` parameter incremented by 1 for the main order. The prefix is made up of the name of the job chain to allow parallel use of this job with a number of job chains.
    - The orders are assigned the state that is associated with the next job node, the `create_balance_statements` job node, i.e. the orders will be executed starting with that state.
    - The orders are assigned the end state that is associated with the `end_of_day_sync` job.
- The `create_balance_statements` job is configured for a maximum number of 10 parallel tasks via the attribute `<job tasks="10">`. It could be configured for any number of parallel tasks.
- The `end_of_day_sync` job is used to synchronize splitted orders and is provided by the Sync JITL Job with the Java class `com.sos.jitl.sync.JobSchedulerSynchronizeJobChainsJSAdapterClass`. This job is used without parameters.
- The `end_of_day` process class configures a number of Agents.
- Hint: to re-use the `end_of_day_split` job you can
    - store the job to some central folder and reference the job in individual job chains.
    - move the JavaScript code of the job to some central location and use a corresponding `<include>` element for individual job scripts.

Job: end_of_day_split

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<job  order="yes" stop_on_error="no" title="split processing for execution on a number of agents">

    <params >
        <param  name="sync_state_name" value="sync"/>
    </params>

    <script  language="java:javascript">
        <![CDATA[
function spooler_process()
{
        var rc = true;
        var syncParameterName = spooler_task.order.job_chain.name + '_required_orders';

    // get job and order parameters
          params = spooler.create_variable_set();
          params.merge( spooler_task.params );
          params.merge( spooler_task.order.params );

        if ( !params.value( 'min_account' ) )
        {
                spooler_log.error( 'parameter missing: min_account' );
        }

        if ( !params.value( 'max_account' ) )
        {
                spooler_log.error( 'parameter missing: max_account' );
        }

        if ( !params.value( 'sync_state_name' ) )
        {
                spooler_log.error( 'parameter missing: sync_state_name' );
        }

    var minAccount = parseInt(params.value( 'min_account' ));
    var maxAccount = parseInt(params.value( 'max_account' ));
    var syncStateName = params.value( 'sync_state_name' );

    spooler_log.debug( '.. executing xml: <show_state subsystems="folder job" what="source folders
no_subfolders" path="' + spooler_task.order.job_chain_node.next_node.job.folder_path + '"/>' );
    var response = spooler.execute_xml( '<show_state subsystems="folder job" what="source folders
no_subfolders" path="' + spooler_task.order.job_chain_node.next_node.job.folder_path + '"/>' );

    if ( response )
    {
        spooler_log.debug( '.... show state for job: ' + response );
        var xmlDOM = new Packages.sos.xml.SOSXMLXPath( new java.lang.StringBuffer( response ) );

                // get process class assignment from job
                spooler_log.debug( '.... query process class assignment from job: //folder[@path = "' +
spooler_task.order.job_chain_node.next_node.job.folder_path + '"]/jobs/job[@path = "/' + spooler_task.order.
job_chain_node.next_node.job.name + '"]' );

                var jobNode = xmlDOM.selectSingleNode( '//folder[@path = "' + spooler_task.order.job_chain_node.
next_node.job.folder_path + '"]/jobs/job[@path = "/' + spooler_task.order.job_chain_node.next_node.job.name +
'"]' );
                if ( !jobNode )
                {
                        throw 'no configuraton found for job: ' + spooler_task.order.job_chain_node.next_node.
job.name;
                }

                var processClassPath = jobNode.getAttribute( 'process_class' );
                if ( !processClassPath )
                {
                        throw 'no process class assignment found for job: ' + spooler_task.order.job_chain_node.
next_node.job.name
                }

                if ( processClassPath.lastIndexOf( '/' ) > -1 )
                {
                        var processClassDirectory = processClassPath.substring( 0, processClassPath.lastIndexOf
( '/' ) );
                        var processClassName = processClassPath.substring( processClassPath.lastIndexOf( '/' )
+ 1 );
                } else {
                        var processClassDirectory = spooler_task.order.job_chain.path.substring( 0,
spooler_task.order.job_chain.path.lastIndexOf( '/' ) );
                        var processClassPath = processClassDirectory + '/' + processClassName;
                }
        }

    spooler_log.debug( '.. executing xml: <show_state subsystems="folder process_class" what="source folders
no_subfolders" path="' + processClassDirectory + '"/>' );
    var response = spooler.execute_xml( '<show_state subsystems="folder process_class" what="source folders
no_subfolders" path="' + processClassDirectory + '"/>' );

    if ( response )
    {
        spooler_log.debug( '.... show state for process class: ' + response );
        var xmlDOM = new Packages.sos.xml.SOSXMLXPath( new java.lang.StringBuffer( response ) );

                // get Agents from process class configuration
                var xmlAgents = xmlDOM.selectNodeList( '//process_classes/process_class[@path = "' +
processClassPath + '"]//remote_scheduler' );

                if ( xmlAgents.getLength() == 0 )
                {
                        spooler_log.error( 'no process class found for expected path: ' + processClassPath );
                }

                // check for available Agents
                var availableAgents = [];

                for( i=0; i < xmlAgents.getLength(); i++ )
                {
                        var agentId = xmlAgents.item(i).getAttribute( 'remote_scheduler' );
                        var agentService = agentId + '/jobscheduler/agent/api/';
                        spooler_log.debug( '.... Agent configuration found: ' + agentId );

            try {
                    var agentResponse = com.sos.jitl.restclient.JobSchedulerRestClient.executeRestService(
agentService );
                        } catch (e) {
                                spooler_log.info( '.... Agent Service [' + agentService + '] returns error: ' +
String(e) );
                                agentResponse = false;
                        }

                          var skipAgent = false;

                        if ( agentResponse )
                        {
                        eval( "var jsonObject = " + agentResponse + ";" );
                                spooler_log.debug( '.... output of Agent response: ' + agentResponse );

                                if ( jsonObject.isTerminating )
                                {
                                        spooler_log.info( '.... Agent Service [' + agentService + '] signals
termination' );
                                        skipAgent = true;
                                }
                    } else {
                        spooler_log.info( '.... no response from Agent web service at: ' + agentService );
                                skipAgent = true;
                    }

                        if ( skipAgent )
                        {
                                spooler_log.info( '.... skipping unavailable Agent: ' + agentId );
                        } else {
                                availableAgents.push( agentId );
                        }
                }

                // sample: initialize chunk size from the number of available Agents
                var currentOrderChunkSize = ( maxAccount-minAccount+1 ) / availableAgents.length;
                var currentOrderChunkSizeInt = parseInt( currentOrderChunkSize );
                var currentOrderChunkSizeReminder = currentOrderChunkSize % 1;
                var subMinAccount = 0;
                var subMaxAccount = minAccount-1;

                for( i=0; i < availableAgents.length; i++ )
                {
                        // sample: calculate chunk size for child order from the number of available Agents
                        subMinAccount = subMaxAccount + 1;
                        subMaxAccount = subMinAccount + currentOrderChunkSizeInt;

                        if ( i == 0 && currentOrderChunkSizeReminder )
                        {
                                subMaxAccount += 1;
                        } else if ( i == availableAgents.length-1 ) {
                                subMaxAccount = maxAccount;
                        }

                        // create child order
                    var subOrder = spooler.create_order();
                    var subParams = spooler.create_variable_set();
                        subParams.set_var( syncParameterName, ( availableAgents.length+1 ) );
                    subParams.set_var( 'sync_session_id', spooler_task.order.id );
                    subParams.set_var( 'number_of_orders', availableAgents.length );
                    subParams.set_var( 'min_account', subMinAccount );
                    subParams.set_var( 'max_account', subMaxAccount );
                    subOrder.params = subParams;
                    subOrder.id = spooler_task.order.id + '_child_order_' + ( i + 1 );
                        subOrder.title = 'child order of parent order: ' + spooler_task.order.id;
                    subOrder.state = spooler_task.order.job_chain_node.next_state;
                    subOrder.end_state = syncStateName;

                        // launch child order
                    spooler_task.order.job_chain.add_order( subOrder );
                    spooler_log.info( '.. child order ' + ( i + 1 ) + ' has been added for range: ' +
subMinAccount + ' - ' + subMaxAccount + ': ' + subOrder.id );
                }
    } else {
        spooler_log.error( 'no response from JobScheduler Master for command: <show_state>' );
        rc = false;
    }
```

```
        spooler_task.order.params.set_var( syncParameterName, ( availableAgents.length+1 ) );
        spooler_task.order.params.set_var( 'sync_session_id', spooler_task.order.id );
        spooler_task.order.state = syncStateName;

          return rc;
}
        ]]>
    </script>
    <run_time />
</job>
```

Explanations

- Line 5: the `sync_state_name` parameter is used when creating child orders to specify the end state of child orders.
- Line 15-37: the code is about parameter checking
- Line 39-70: the name of the process class is extracted from the successor job which is the `create_balance_statements` job. As process classes can be referenced by absolute or relative paths the code calculates the absolute path and creates local variables for the directory, name and path of the process class.
- Line 72-86: the code queries the JobScheduler Master to show the source of the process class. The answer is provided in XML format and is parsed accordingly. The xPath query in line 81 selects the Agent URLs from the process class.
- Line 88-127: each Agent URL that is extracted from the process class configuration is checked for an available Agent. This information is subsequently used to calculate the `min_account` and `max_account` parameters.
- Line 98: executes a REST request to the JobScheduler Agent.
    - The REST client library ships with JobScheduler and is explained with the How to implement a client for REST web services article.
    - The REST interface for JobScheduler Master and Agents is explained with the JOC Cockpit - REST Web Service article.
    - The technical documentation of the REST interface is available from the web site JOC Cockpit REST Web Services Technical Documentation (RAML Specification)
- Line 129-147: implements some arbitrary logic how to create chunks from the `min_account` and `max_account` parameters. The idea is to split the range into chunks that corresponds to the number of available Agents, however, any other logic could apply.
- Line 149-161: for each Agent found from the process class an order is created and is parameterized with the respective chunk of accounts.
- Line 172-174: the main order is moved to the `end_of_day_sync` job node and a `sync_session_id` parameter is added that allows the `end_of_day_sync` job to be used in parallel for a number of main orders.

Job: create_balance_statements

```
<job  process_class="end_of_day" order="yes" stop_on_error="no" tasks="10" stderr_log_level="error">
    <params/>
    <script language="shell">
        <![CDATA[
@echo "job running on Agent: %SCHEDULER_HOST%:%SCHEDULER_HTTP_PORT%"
@echo "creating balance statements for clients: %SCHEDULER_PARAM_MIN_ACCOUNT% - %SCHEDULER_PARAM_MAX_ACCOUNT%"
ping -n 5 localhost
        ]]>
    </script>
    <run_time />
</job>
```

Explanations

- Line 1: the job is assigned the `end_of_day` process class, see below configuration.
- Line 5-7: the job includes an arbitrary implementation to create balance statements.
    - For the sake of this sample the job simply uses some `echo` commands.
    - The job uses the built-in environment variables `SCHEDULER_HOST` and `SCHEDULER_HTTP_PORT` to display the host and port of the Agent that the job is executed for.
    - The job displays parameters for the range of accounts that should be processed from the `SCHEDULER_PARAM_MIN_ACCOUNT` and `SCHEDULER_PARAM_MAX_ACCOUNT` environment variables.
- The job is implemented for Windows. For Unix environments use the `$SCHEDULER_HOST` instead of `%SCHEDULER_HOST%` syntax to reference environment variables.

Process Class: end_of_day

```
<process_classes >
    <process_class  max_processes="10">
        <remote_schedulers  select="next">
            <remote_scheduler  remote_scheduler="http://server1:4445"/>
            <remote_scheduler  remote_scheduler="http://server2:4445"/>
            <remote_scheduler  remote_scheduler="http://server3:4445"/>
        </remote_schedulers>
    </process_class>
</process_classes>
```

Explanations

- Line 3: the process class defines an Active Cluster with round-robin scheduling by use of the `select="next"` attribute: each task for an order gets executed on the next Agent.
- Line 4-6: the process class specifies a number of Agents that are addressed by the http or https protocol. If one of the Agents is not available then this is considered by the `end_of_day_split` job when calculating chungs of work for the `create_balance_statements` job.
- The process class is assigned to the job `create_balance_statements`, see above job configuration.

Order: end_of_day

```
<order>
    <params>
        <param name="min_account" value="100000"/>
        <param name="max_account" value="200000"/>
    </params>
    <run_time/>
</order>
```

Explanations

- Line 3-4: the order includes the parameters that specify the range of accounts for which balance statements are created.
- LIne 6: any order configuration, e.g. run-time rule, can be added.

Usage

- Start the `end_of_day` order for the `end_of_day` job chain by use of JOC Cockpit.
- Consider the processing that would
  - split the execution into 3 subsequent orders that run for the `create_balance_statements` job each on a different Agent. Should one of the Agents not be available then the number of orders is reduced and the chunk size is calculated accordingly.
  - move the current order to the `end_of_day_sync` job node.
- The splitted orders for the `create_balance_statements` job will arrive in the `end_of_day_sync` job node and will wait for all orders to be completed. With all splitted orders being completed the processing will continue.