|  |  |  |
| --- | --- | --- |
| Huawei Technologies Co. Ltd. | Product version | Confidentiality level |
| C70 | Classified |
| Product name:  FusionInsight | Total pages： |

**FusionInsight Superior Planner TCC Integration**

**Prepared by:** Sam Sanjabi 00368570 Chong Chen C00320067

**Reviewed by:** -

**Approved by:** -

2016-09-23

  
Huawei Technologies Co., Ltd.   
All rights reserved

FusionInsight Superior Planner TCC Integration

For internal use only

|  |  |  |  |
| --- | --- | --- | --- |
| Prepared by | Sam Sanjabi  Chong Chen | Date | 2016-09-23 |
| Reviewed by |  | Date |  |
| Approved by |  | Date |  |
| Granted by |  | Date |  |



Huawei Technologies Co., Ltd.

All rights reserved

Revision record

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Revision version | Description | Author |
| 2015/08/26 | 1.0 | Initial draft | Sam Sanjabi |
| 2015/9/23 | 1.1 | Update doc to include overall architecture flow and componet interaction in more detail | Chong Chen |
| 2015/9/29 | 1.2 | Add RESTful interface for query plan  Update attempt id structure  Update graph for arrow | Chong Chen |

Table of Content

Contents

[FusionInsight Superior Planner TCC Integration 2](#_Toc462948318)

[Revision record 3](#_Toc462948319)

[Table of Content 4](#_Toc462948320)

[1 Introduction 5](#_Toc462948321)

[1.1 Background 5](#_Toc462948322)

[1.2 Superior Planner 6](#_Toc462948323)

[1.3 Superior Planner and TCC Job Scheduler System Integration 7](#_Toc462948324)

[2 TCC Adapter 8](#_Toc462948325)

[2.1 TCC Adapter Configuration 8](#_Toc462948326)

[2.2 TCC Adapter Architecture and Logic Flow 10](#_Toc462948327)

[2.2.1 Assumptions 12](#_Toc462948328)

[2.2.2 Questions 13](#_Toc462948329)

[2.2.3 TCC Adapter main functions 13](#_Toc462948330)

[3 Superior Planner External interfaces 15](#_Toc462948331)

[3.1 Orchestrator Action ID Format 15](#_Toc462948332)

[3.2 REST APIs 18](#_Toc462948333)

[3.2.1 Query Planner status 18](#_Toc462948334)

[3.2.2 Register and update external resources 21](#_Toc462948335)

[3.2.3 Query external resources 21](#_Toc462948336)

[3.2.4 Delete external resources 22](#_Toc462948337)

[3.2.5 Create and update a Plan 23](#_Toc462948338)

[3.2.6 Delete Plan 27](#_Toc462948339)

[3.2.7 Update Orchestrator Action Status 28](#_Toc462948340)

[3.2.8 Query planned start time for all actions 29](#_Toc462948341)

[4 Other Interfaces 31](#_Toc462948342)

[4.1 Hive Client Execution Variables 31](#_Toc462948343)

[4.2 Required Hadoop Configurations 32](#_Toc462948344)

[4.3 Recommended Hadoop Configurations 32](#_Toc462948345)

**FusionInsight Superior Planner TCC Integration**

# Introduction

## Background

In big data space, the industry moving trend is to consolidate multiple application or department specific silo big data systems into one system to support multi-tenant environment and improve system resource utilization. Although resource and workload consolidation are in right direction, it does introduce some challenge, in particular, in the consolidated environment, how to guarantee right resource for critical workloads to make sure important business workflow able to complete before deadline.

In a typical enterprise data process environment, there are normally two tiers of systems. One is business workflow tier (system like oozie, control-M, TCC from CBG), which manages workflow action dependency and life cycle. The other tier is resource management system (YARN), which schedules individual workloads based on various policies. These two tiers system have no connection, top tier only cares about higher level dependency, has no knowledge about underlying resource availability and when and how to allocate those resources to critical workloads. Underlying scheduler system only has knowledge of individual workloads, has no knowledge of higher level workload dependency and deadlines. This disconnection of business SLA definition and underlying resource allocation strategy leads to no guarantee of critical workflow SLA (deadline).

**Workflow**

**Orchestration**

**Big Data App**

Hive, Pig, …

**YARN**

**Workflow Orchestration**

* Encapsulates complex business logic
* Manages repeatable workloads, continuous processing
* Examples: Oozie, Control-M, Azkaban, …

**Gateway Cluster**

* Distribute workflow tasks to various underlying systems
* May or may not be under orchestrator control

**Enterprise subsystems**

* YARN-based data processing applications
* Legacy databases
* Data transfer services, file system operations, …
* Non-YARN applications

**YARN Scheduling**

* Multi-tenant policy support
* Features to monitor workload progress
* Features to allocate greater resources to critical work

**Data Transfer**

FTP, SSH, …

**Database**

Oracle, TeraData, …

**Other**

HBase, HDFS, …

**Query Planning**

* Mix of YARN / non-YARN stages (sub-tasks)

**BUSINESS TIER**

**CONTROL SYSTEM**

Figure 1 Typical Enterprise Data Processing Environment

## Superior Planner

Superior Planner is a new component added to the Big Data deployment context (FusionInsight HD) to help bridging business workflow tier and underlying big data system and guarantee SLA (deadline) of critical business workflows while maintaining good overall cluster utilization (reducing silos) and response time for unplanned (ad-hoc) jobs.

Superior Planner is a long running service running in big data FI environment with built-in high availability capability. It uses FI hadoop APIs to integrate with underlying hadoop YARN system and provides RESTful interfaces to integrate with any workflow orchestrator systems.

The diagram below shows the relationship of the planner to other elements in this context.

**Workflow**

**Orchestration**

**Big Data App**

Hive, Pig, …

**YARN**

**Data Transfer**

FTP, SSH, …

**Database**

Oracle, TeraData, …

**Other**

HBase, HDFS, …

**SLA Planner**

Prior to jobs being submitted to control system:

* Generate QoS ID for each control system job (or expected control system job)
* Assign resource requirement for each job
* Create resource allocation plan for each job
* Submit plan to underlying control system for enforcement

**BUSINESS TIER**

**CONTROL SYSTEM**

* **Workflow Dependencies**
* **Deadlines**
* **Resource allocation plans**

**Execution Status**

**Execution Status**

Figure 2. Superior SLA Planner

There are several key integration points from high level with Superior Planner:

* **Workflow Orchestrator** (e.g. Oozie, TCC, Control-M, etc.)
  + Use Planner RESTful interface to create workflow instance graph and trigger the plan.
  + Respect planned start time of workflow actions
  + Send planned workflow-level task status updates to planner, so that planner can continue to track progress of running workflow.
  + Set unique id of each workflow-level task into big data YARN environment, so that planner can automatically detect resource usage of different phases of each task.
* **Scheduler** (e.g. YARN)
  + Provide dynamic queue management interface to allow Planner to make workload specific resource reservation
  + Provide service interface to allow Planner to query workload status of resource reservation

## Superior Planner and TCC Job Scheduler System Integration

TCC Job Scheduler is a home grown workflow orchestrator designed and implemented by HUAWEI Consumer Business Group (CBG). Its main purpose is to orchestrate pre-defined tasks and execute them in order to do data processing and generate business reports.

TCC Job Scheduler system needs to integrate with Superior Planner to guarantee critical workflow SLA. Logically, there are four integration points, showed in red color:

Planner

TCC Job Scheduler

TCC

Adapter

RESTful

interface

TCC Database

Update status

Fetch data

**update task**

TCC\_task

FI YARN

FI Hive

Submit M/R

Superior Scheduler

Launch TCC task

**Set Unique Planner ID per TCC\_task**

Push down plan

Pull job and

cluster status

**Create plan**

**Query task start time**

FusionInsight HD

* TCC Job Scheduler needs to inform Superior Planner about workflow instance execution plan, so that Planner can make plan and push down to YARN Superior Scheduler
* During workflow runtime, TCC Job Scheduler needs to notify Planner about running task status, basically, task start time, finish time, so that Planner can do adjustment accordingly.
* TCC Job Scheduler can query Planner to get predicated task start time. Ideally it should follow start time to launch each task, this is recommended step.
* TCC Job Scheduler should set unique orchestrator workload ID into hive query environment, so that Planner can automatically analyze and group independent YARN jobs (Map Reduce) into single TCC task and predict sub-DAG resource usage.

The Planner provides 3 RESTful interfaces to allow workflow orchestrator to do integration. Please see following sections for detail.

For TCC scenario, since TCC Job Scheduler needs to access TCC database, in order to minimize impact on production system, we plan to introduce a new component (TCC Adapter) to access the same TCC database (read-only) to handle Planner integration, basically, workflow execution plan creation and update. TCC Adapter is a Java program that can run in a standalone process or can even tightly integrated into TCC Job Scheduler system eventually. For former case, TCC Adapter will be running on the same host as Planner component and Planner can manage its life cycle.

# TCC Adapter

TCC Adapter is a Java program that can run in standalone or in embedded mode. In standalone mode, TCC Adapter will be started and monitored by Planner component, while TCC Adapter can be part of TCC Job Scheduler in embedded mode. In later case, it will be TCC Job Scheduler responsible to monitor and control life time of Adapter component.

## TCC Adapter Configuration

|  |
| --- |
| **Filename:** tcc-adapter.xml |
| **Location:** planner installation directory |

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Type | Format | Description |
| lookAhead\_duration | String | HOURS:MINUTES:SECONDS | Determines how far ahead from current time should the adaptor look every time it creates instance dependency graph for the Planner.  HH: hour, MM: minute, SS: seconds |
| create\_plan\_period | String | HOURS:MINUTES:SECONDS | Defines time distance between consecutive calls to Planner createPlan interface |
| status\_update\_period | String | HOURS:MINUTES:SECONDS | Defines time distance between consecutive calls to sendStepStatusUpdate() method |
| cluster\_resource | String | clusterId:[resourceName1, resourceName2, …] | Determines capacity of which cluster resources of cluster with ID = clusterId, should be sent to planner.  Example:  1:[max\_running\_num] |
| default\_cluster\_resource | String | clusterId:[resourceName1, resourceName2, …] | Used if cluster\_resource is not defined or there was error retrieving cluster info from database.  The values should be equal to the ones in TccConfig.  Example:  ClusterInfoEntity.DEF\_CLUSTER\_ID:[TccConfig.getMaxRunningCycleTaskNum()] |
| predictor\_event\_log\_file | String | <path\_to\_file> | Path to the event log file used by Predictor |

The proposed schema of the configuration file is similar to the standard configuration file using name/value pairs for parameter definitions:

<?xml version="1.0"?>

<xs:schema elementFormDefault="qualified" targetNamespace="http://www.huawei.com/scheduler/superior/2015/07/schema" xmlns="http://www.huawei.com/scheduler/superior/2015/07/schema" xmlns:ss="http://www.huawei.com/scheduler/superior/2015/07/schema" xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="property">

<xs:complexType>

<xs:sequence>

<xs:element maxOccurs="1" minOccurs="0" name="description" type="xs:string"/>

<xs:element maxOccurs="1" minOccurs="1" name="name" type="xs:string"/>

<xs:element maxOccurs="1" minOccurs="1" name="value" type="xs:string"/>

</xs:sequence>

</xs:complexType>

</xs:element>

<xs:element name="configuration">

<xs:complexType>

<xs:sequence>

<xs:element maxOccurs="unbounded" minOccurs="0" ref="ss:property"/>

</xs:sequence>

</xs:complexType>

</xs:element>

</xs:schema>

## TCC Adapter Architecture and Logic Flow

TCC Adapter is a module sitting between TCC orchestrator and Planner module, which acts as a data converter. It access TCC orchestrator database and generates date in a format needed by Planner module.

**TccAdapter communicates with planner through the following rest APIs**

|  |  |  |
| --- | --- | --- |
| Planner rest API | TccAdapter Method calling API | Description |
| /planner/plans | createPlanDepGraph(Date) | Generates step-level instance dependency graph and ask planner to schedule tasks in this graph |
| /planner/workloads | sendStepStatusUpdate(Date) | Generates event log file used by predictor and sends step instances start/finish times to planner |
| /planner/resources | createClusterResourceCapacity(  Integer, List<String>) | Send external resources (i.e. gateway slots) to planner |

When TCC Adapter daemon starts it creates two threads which will be periodically executed to generate data required by the planner.

* The first thread, “CreatPlanThread”, is executed every ‘create\_plan\_period’ seconds and calls ‘createPlanDepGraph’. This method generates instance dependency graph of all task instances which can start before ‘current time + lookAhead\_duration’ and ask the Planner to schedule tasks in this graph.
* The second thread, “StatusUpdateThread”, is executed every ‘status\_update\_period’ seconds and calls ‘sendStepStatusUpdate’ and ‘sendClusterResource’ methods. The first method, ‘sendStepStatusUpdate’, retrieves start time, finish time and state of task step instances not retrieved since the last call to this method. It then logs these information into event log file, ‘predictor\_event\_log\_file’, and send same information to Planner. The second method, ‘sendClusterResource’, gets capacity of resources defined in ‘cluster\_resource’ for cluster identified by ‘clusterId’ and sends these external resource capacities to the Planner.

Following diagram demonstrates above procedures.

**re-runs every create\_plan\_period seconds**

**StatusUpdateThread**

// log step instance actions

// happened since logLastDate

logLastDate

= sendStepStatusUpdate(logLastDate)

// set cluster resource capacity

sendClusterResource(clusterId, resourceNames)

// create timer ‘planTimer’ with

// period = create\_plan\_period

// create CreatePlanThread

// add CreatePlanThread to planTimer

// create timer ‘statusTimer’ with

// period = status\_update\_period

// create StatusUpdateThread

// add StatusUpdateThread to statusTimer

**TCC Adapter Main Thread**

// initialize member variables

logLastDate = new Date(0,0,0,0,0,0)

create\_plan\_period = …

status\_update\_period = …

lookAhead\_duration = …

// parse cluster\_resource to set

// Integer clusterId

// List<String> resourceNames

// create timer ‘planTimer’ with

// period = create\_plan\_period

// create CreatePlanThread

// add CreatePlanThread to planTimer

// create timer ‘statusTimer’ with

// period = status\_update\_period

// create StatusUpdateThread

// add StatusUpdateThread to statusTimer

**re-runs every status\_update\_period seconds**

**CreatePlanThread**

// set planEndDate to system current time

// + lookAhead\_duration

// create dependency graph and call

// Planner create plan interface

createPlanDepGraph(planEndDate)

**Ways to implement TccAdaptor**

|  |  |  |
| --- | --- | --- |
| Integration Type | Pros. | Cons. |
| As a component of TCC orchestrator | * Directly access data structures and classes in TCC orchestrator * Does not need to independently connect to database * Minimum code duplication | * TCC orchestrator needs to be redeployed once new components is added |
| Communicate with TCC orchestrator through rest API | * Does not need to independently connect to database * Minimum code duplication | * TCC orchestrator needs to provide rest API for every method used by TccAdaptor * TCC orchestrator needs to be redeployed |
| As an independent component with direct read access to database | * No change to current TCC orchestrator * Does not require TCC orchestrator redeployment | * Need read access to database * Need duplicating part of TCC data structure and code * Any change in duplicated codes at TCC part should be applied at adaptor side (code maintenance problem) |

### Assumptions

The logic inside TCC Adapter assumes the following assumption on TCC orchestrator:

1. tcc\_task table has column ‘SLA\_Offset’ of type string with the same format as ‘Cycle\_Offset’ (i.e. 1M2D3h40m) which sets deadline of that task based on tasks cycleID. In this example, the deadline of the task is 1 month, 2 days, 3 hours and 40 minutes after it cucleID. TaskEntity should have getSlaOffset() method which returns this value.
2. Method TaskManage.getEnabledTaskList() returns list of enabled tasks.
3. NEW method TccDao.getMinCycleDate() returns earliest date which has unfinished task instance.
4. Method TccUtil.maxCycleIdTimeisOK(Date date, String cycleOffset) returns maximum cycleId that can be executed before ‘date’ based on ‘cycleOffset’.
5. Method TccUtil.covCycleID2Date(String cycleId) returns a date object which corresponds to input cycleId.
6. Method TccServiceImpl.generateCycleIDs(Date minDate, Date maxDate,

int cycleType, int cycleLength)

returns list of cycleIds falling in the range [minDate, maxDate], for a cycle with cycleType and cycleLength.

1. Method TccServiceImpl.getAllCycleDepRs(Long taskId, String cycleId,

TaskEntity task)

returns list of CycleDependRelation of task instance (taskId+cycleId) which are after variable maxStartTime defined in this method. This functions returns list of task instances that taskId depnends on.

1. Any TaskRunningState with RunningState other than RunningState.SUCCESS or RunningState.VSUCCESS is not finished gracefully and is being executed now or will be executed later. Therefore, such task instances should be included in dependency graph.
2. Method TccDao.getTaskRunningState(TaskRunningState) returns a TaskRunningState object corresponding to its input parameter which includes its most recent RunningState. By calling getState() method of returned object one can tell whether this task instance is successfully executed or not.
3. NEW method TccDao.getStepRunningStatesStartedAfter(Date fromTime)

returns List<StepRunningStateEntity> where runningStartTime >= fromTime.

1. NEW method TccDao.getStepRunningStatesEndedAfter(Date fromTime)

returns List<StepRunningStateEntity> where runningEndTime >= fromTime.

1. Retry\_Count in tcc\_step\_running\_state gives current attempt number. If it is null then current run is the first attempt.

### Questions

Following questions should be addressed and fully understood in order to make sure TCC Adapter generates valid date:

1. What does TccConfig.getMinCycleId() do? Can we use its returned value to get earliest date which has unfinished task instance (related to Assumption 3)?
2. In some functions (such as the one in Assumption 7) there is tccDomain, DomainDAOs, … . The purpose of section of code which uses these variables is not clear.
3. Each TCC TaskStep can have single or multiple batches. Explain how these batches are defined and executed in each TaskStep.

### TCC Adapter main functions

* **void createPlanDepGraph(Date planEndDate)**

Generate instance-level dependency graph of SLA jobs that can start execution before ‘planEndDate’. Dependency graph goes as far back in time as possible to include all task instances not successfully executed.

Detailed Steps:

1. Get list of enabled tasks
2. Set ‘minDate’ as the earliest date which has unfinished task instances
3. For each enabled task with SLA, get list of cycleIds in range [minDate, planEndDate].
   1. For each cycleId in above list, create new TaskRunningState object and add it to ‘remaining’ list.
4. While remaining list is not empty
   1. Pop first taskRS of type TaskRunningList from remaining list
   2. Get list of cycle dependency relations of taskRS which are after minDate. This function is similar to TccServiceImpl.getAllCycleDepRs(Long taskId, String cycleId, TaskEntity task) except the fact that variable maxStartTime is set to minDate rather than task.getStartTime()
   3. For each CycleDependRelation in above returned list create a new TaskRunningState object named newTaskRS. If this newTaskRS is not in processed list add it to remaining list.
   4. Add taskRS to processed list.
5. Generate dependency graph from task instances in processed list.
6. For each task instance in above graph get its RunningState by calling

TccDao.getTaskRunningState(TaskRunningState).getState() If RunningState is either RunningState.SUCCESS or runningState.VSUCCESS remove its corresponding node from graph (this job successfully finished).

For situations where the following dependency chain exists T1 -> T2 -> T3 and T2 is successfully finished but T1 is not, this function prints out a warning message, removes T2 and changes the graph dependency T1 -> T3.

1. Generate JSONObject from dependency graph and send it to Planner createPlan interface.

* **Date sendStepStatusUpdate(Date fromTime)**

Query tcc\_setp\_running\_state table to get list of StepRunningStateEntity which started after fromTime. Logs start time, finish time, attempt number and state of each step instance in a file to be used by predictor. Also sends start time and finish time of step instances to the planner. This function returns new fromTime to be used for the next call.

Detailed Steps:

1. Set newFromTime to current system time.
2. If fromTime is null, set fromTime = new Date(0, 0, 0, 0, 0 ,0)
3. Get list of step instances started after fromTime as stepRSStartlst by calling

TccDao.getStepRunningStateStartedAfter(fromTime)

1. For each StepRunningStateEntity in stepRSStartlst as stepRS
   1. attemptNum = stepRS.retryCount == null ? 0 : stepRes.retryCount
   2. actionID =

(stepRS.taskId, stepRS.cycleId, stepRS.stepId, attemptNum)

* 1. event = stepRS.runningStartTime //timestamp
  2. event += “START” // action type
  3. event += actionID // action ID
  4. event += “” // additional info
  5. Log event into predictor\_event\_log\_file\_<CURRENT\_DAY>

1. Get list of step instances finished after fromTime as stepRSFinishlst by calling TccDao.getStepRunningStateEndedAfter(fromTime)
2. For each StepRunningStateEntity in stepRSFinishlst as stepRS
   1. attemptNum = stepRS.retryCount == null ? 0 : stepRes.retryCount
   2. actionID = (stepRS.taskId, stepRS.cycleId, stepRS.stepId, attemptNum)
   3. actionType = “FAILED”
   4. if stepRS.state = RunningState.SUCCESS or RunninState.VSUCCESS

actionType = “SUCCESS”

* 1. event = stepRS.runningEndTime //timestamp
  2. event += actionType // action type
  3. event += actionID // action ID
  4. event += stepRS.runningJobId // additional info
  5. Log event into predictor\_event\_log\_file\_<CURRENT\_DAY>

1. From StepRunningStateEntity in stepRSFinishlst and stepRSFinishlst lists generate step instance status (action ID, start time, finish time) list and send it to Planner.
2. Return newFromDate

* **Void sendClusterResourceCapacity(Integer clusterId, List<String> resourceNames)**

Send list of <resourceName, capacity> of cluster with ID clusterId to planner.

Detailed Steps:

1. Get ClusterInfoEntity as clusterInfo by calling TccDoa.getClusterInfo(Integer clusterId)
2. If clusterInfo is null or it does not have one of the resources in resoureNames or the maximum value of any of the resources in resourceNames is invalid, set clusterInfo = defaultClusterInfo
3. Generate list of <resourceName, maxAmount> for each resource in resourceNames and send it to Planner updateExternalResources interface.

# Superior Planner External interfaces

## Orchestrator Action ID Format

The Superior Planner assumes a particular format for IDs that the workflow orchestrator provides. The purpose is to be able to help Planner predication system:

1. Group individual scheduler level *jobs* (e.g. Map-Reduce jobs, Spark contexts, etc...) associated with the same orchestrator level workload together.
2. Identify which workload *executions* are actually periodically executed instances of the same workload *definition*.

The Orchestrator Action ID format provided by the orchestrator must be separated into a couple of fields separated by the colon (‘:’) and (‘.’) character:

[ORCHESTRATOR\_NAME:]DEFINITION\_ID:INSTANCE\_ID[.SUBDAG\_PATH][:ATTEMPT\_NUMBER]

ORCHESTRATOR\_NAME is optional field, which is used to uniquely identify orchestrator system, in case there are multiple orchestrators running concurrently and using the same underlying hadoop services. If system has only one orchestrator, then we can leave this empty, for instance TCC case.

The DEFINITION\_ID field will stay the same across multiple execution instances of the orchestrator actions.

|  |
| --- |
| In the TCC case, this field should include:   * The TCC Task ID |

The INSTANCE\_ID field will change with each separate execution of the action, it is used by the prediction component to identify multiple executions of the same action into a time-series for the purposes of predicting information about future executions.

|  |
| --- |
| In the TCC case, this field should include:   * The Period Date * The Period Hour |

Each action may include multiple sub actions organized based on hierarchy DAG. Each sub actions can be managed and controlled independently, but all these sub actions may share some common life time external resources or subject to some external scheduling constraint. For instance, following diagram show a two level DAGs:



Action includes 2 level DAGs, and 3 sub actions, sub action “step1” will be executed first, followed by sub actions “step2” and “step3”. All of these sub actions will be subject to single constraint. To uniquely identify each sub action within action, it needs to use full path from action (root node) with ‘.’ as separator. For instance, for step1, it will be <ActionID>.level1\_1.step1, while step2 will be ActionID.level1\_2.step2 etc.

|  |
| --- |
| In TCC case, each action has only one level DAGs, which includes multiple sub actions (steps) executed in sequential order. Each sub action has unique identifier by an integer. |

The ATTEMPT\_NUMBER field is optional and should change each time an execution instance is retried by the orchestrator due to failure or other reasons. Orchestrator system can also have potential behavior to handle retry on different DAG level. In order to uniquely identify which level retry applies to, ATTEMPT\_NUMBER also needs to follow the same naming convention and separated by ‘.’ separator and leave. For instance, if retry attempt is only on leaf sub action level, then previously example will become “1.1.1” or “1.1.2”. Each level must be positive integer.

|  |
| --- |
| In TCC case, a full orchestrator action ID will be defined as following:  **${TCC\_TASK\_ID}:${TCC\_PERIOD\_DATE}-${TCC\_PERIOD\_HOUR}.${TCC\_STEP\_NUMBER}:1.${TCC\_STEP\_ATTEMPT\_NUM}**  Here is a concrete example:  **4010081:20131008-0000.1:1.1**  4010081 : TCC Task definition ID  20131008-0000 : TCC Task instance ID, includes cycle ID and also time.  1 : sub action(step) id  1.1 : attempt number |

**Notes**:

* When the ATTEMPT\_NUMBER field is included, we refer to the ID string as a complete Orchestrator action ID to distinguish it from a standard Orchestrator Action ID which does not include this field.
* It is assumed that the Orchestrator Action ID will be passed to the YARN job execution context (see section [4](#_Hive_Client_Execution)).

## REST APIs

The REST/HTTP server is part of the Superior Planner, it will bind to a web service port on the host it is running on. Assuming the web service is listening at *planner\_address:planner\_port*, below we specify the REST URLs relative to

https://planner\_address:planner\_port/v1

Only HTTPS will be supported.

The following APIs are intended for use by external orchestrator integration (e.g. Oozie, TCC, etc…) and end users:

* Planner status
* External resource management (create/update/delete/query)
* Plan management (create/update/delete/query)
* Update orchestrator workload events
* Query planning result of given plan and orchestrator

### Query Planner status

|  |
| --- |
| **Description** |
| Show overall status of planner, including   * + - Overall hadoop cluster name and status     - Total number of queues within cluster and their capacity     - A list of orchestrator registered and also registered plans, overall total actions within plan. |
| **URL** |
| GET /planner/status |
| **Input** |
| None |
| Output |
| JSON Response  {  “administrators” : [“hadoopadmin”, “user1” ],  “clusters” : [  {  “name” : “ShenZhen\_cluster”,  “operation\_queue” : “default”,  “status” : ok  “tenant\_queues” : [  {  “name” : “root.BU1”,  “status” : OPEN | ACTIVE,  “capacity” : {“vcores” : 100, “memory” : 45600}  },  {  “name” : “BU2”,  “status” : OPEN | INACTIVE,  “capacity” : {“vcores” : 200, “memory” : 56000}  }  ]  },  {  “name” : “Beijing\_cluster”,  “operation\_queue” : “default”,  “status” : disconnected  “tenant\_queues” :[  {  “name” : “root.Beijing\_BU1”,  “status” : OPEN | ACTIVE,  “capacity” : {“vcores” : 300, “memory” : 23000}  }  ]  }  ],  “orchestrators” : [  {  “name” : “tcc\_system”,  “number\_of\_plans” : 3,  “plans” : [  “critical\_report”,  “regression”,  “sales\_report”  ]  }  ],  “orchestrator\_adapters” :[  {  “name” : “tcc\_adapter”,  “host” : ykz003.huawei.com,  “start\_time” : 1243523535,  “number\_of\_starts” : 1,  “uptime” : 1234245,  “pid” : 132435  }  ]  }   |  |  |  | | --- | --- | --- | | **Attribute** | **Type** | **Description** | | adminstrators | array | An array of administrator names, each one is a string type. | | clusters | array | An array of hadoop cluster object | | Clusters:name | string | Cluster name | | Clusters:operation\_queue | string | Operation queue name used for planner to do predication calculation. | | Clusters:status | string | The value can be “ok”, “disconnected”. | | Clusters:tenant\_queues | array | An array of tenant queues | | Clusters:tenant\_queues:name | string | Name of tenant queue | | Clusters:tenant\_queues:status | string | The value can be “OPEN|CLOSE|ACTIVE|INACTIVE” | | Clusters:tenant\_queues:capacity | object | Resource capacity for queue | | Orchestrators:name | string | Name of orchestrator system | | Orchestrator:number\_of\_plans | int | Number of plans created under orchestrator system | | Orchestrator:plans | array | An array of registered plan names | | Orchestrator\_adapters | array | An array of orchestrator adapter object | | Orchestrator\_adapters:name | string | Adapter name | | Orchestrator\_adapters:host | string | Running host of adapter | | Orchestrator\_adapters:start\_time | long | latest start time of adapter | | Orchestrator\_adapters:number\_of\_starts | int | total number of start times of adapter | | Orchestrator\_adapters:uptime | long | Latest up time of adapter | | Orchestrator\_adapters:pid | long | Process identifier of latest instance of adapter | |

### Register and update external resources

|  |
| --- |
| **Description** |
| External resources will be used to represent non-hadoop cluster resource constraints, for instance, TCC gateway resource. It is share consumable resource.  Register or update a list of external share resource into Planner. For existing resource, it will update its capacity value. |
| **URL** |
| POST /planner/resources |
| **Input** |
| JSON Request:  {  “resourcelist” : [  {  “name” : “tcc-gateway”,  “capacity” : 120  },  {  “name” : “matlab-license”,  “capacity” : 100  }  }   |  |  |  | | --- | --- | --- | | **Attribute** | **Type** | **Description** | | resourcelist | array | A array of resource | | name | string | Resource name, must be unique | | capacity | int | Total number of capacity of resource | |
| Output |
| 201: created successfully |

### Query external resources

|  |
| --- |
| **Description** |
| Query all external resources |
| **URL** |
| GET /planner/resources  GET /planner/resources/${resource\_name} |
| **Input** |
| None |
| Output |
| JSON Response:  {  “resourcelist” : [  {  “name” : “tcc-gateway”,  “capacity” : 120  },  {  “name” : “matlab-license”,  “capacity” : 100  }  }   |  |  |  | | --- | --- | --- | | **Attribute** | **Type** | **Description** | | resourcelist | array | A array of resource | | name | string | Resource name, must be unique | | capacity | int | Total number of capacity of resource | |

### Delete external resources

|  |
| --- |
| **Description** |
| delete one external resource or all external resources |
| **URL** |
| DELETE /planner/resources/${resource\_name}  DELETE /planner/resources |
| **Input** |
| None |
| Output |
| 204: delete successfully |

### Create and update a Plan

|  |
| --- |
| **Description** |
| Create or update a plan for orchestrator actions with SLAs in the given dependency graph. Assumption is that there is one entry for each orchestrator execution unit (in the case of TCC, this is a single step in a task execution instance). |
| **URL** |
| PUT /planner/plans/${orchestrator.name}/${plan.name} |
| **Input** |
| {  “default\_cluster”: “hadoop1”,  "default\_queue": "root.production",  “default\_life\_time\_resources”: [  {  “unit” : {“tcc-gateway”:1, “tcc-gateway2”:1},  “total\_num”: 1,  “name” : “license”  }  ],  “timestep”: 1,  “update\_period”: 1,  “adjustment\_buffer”: 0.25,  “planning\_period”: 48,  “plan\_starttime”: time\_str,  “actions”: { ... }  }   |  |  |  | | --- | --- | --- | | **Attribute** | **Type** | **Description** | | default\_cluster | string | The hadoop cluster under which to create this plan. The cluster must be pre-defined in Planner system. | | default\_queue | string | The hadoop cluster queue under which to create this plan. | | default\_life\_time\_resources | array | *[optional]* The array of requested life time resource per each plan action | | name | string | [optional] External resource name, must be pre-created in the planner | | amount | int | *[Optional]* external resource amount | | update\_period | int | *[Optional]* Plan-level parameter to overwrite ss.planner.update.period if desired. | | adjustment\_buffer | float | *[Optional]* Plan-level parameter to overwrite ss.planner.adjustment.buffer if desired. | | planning\_period | int | *[Optional]* By default, planner will plan out every action within submitted plan. If plan graph is huge, planningperiod can be used to control how many hours for plan window. Unit is hours. | | plan\_starttime | String | *[Optional]* When plan will start, by default, it will be now. | | order\_algorithm\_class | string | *[Optional]* Plan-level parameter to overwrite planner.order-algorithm.class if desired. | | shape\_algorithm\_class | string | *[Optional]* Plan-level parameter to overwrite planner.shape-algorithm.class if desired. | | actions | object | The specific structure is described in the below subsection. | |

The below subsections only describe the format of the “actions” object

Graph Action Structure

|  |
| --- |
| **Description** |
| A generic list of actions, their dependencies, and their deadlines and minimum start times. |
| **Format** |
| "actions": [  {  “id” : "4010007:20160204-0000",  “cluster” : “hadoop2”,  “queue” : “queue1”,  “dependlist”: [],  “default\_alloc”:  {  “resources” : [  {  “name” : “vcores”,  “amount” : 100  },  {  “name” : “memory”,  “amount” : 1480  }  ],  “duration” : “8h”,  “force\_use” : 1  },  “life\_time\_resources” : [  {  “unit” : {“tcc-gateway”:0},  “total\_num” : 0,  “name” : “license”  }],  “dag” : [  {  “id” : “1”,  }  {  “id” : “2”,  “dependlist” : [“1”]  }  ]  },  {  “id” : "4010007:20160204-0000",  "dependlist": ["4010007:20160204-0000"],  “dag” : [ {  “id” : “1”,  }]  },  {  “id” : "4013002:20160204-0100”,  "min\_start": 1456790400000,  "dependlist": [],  “dag” : [ {  “id” : “1”,  }]  },  {  “id” : "4014523:20160204-0000",  "max\_finish": 1456790800000,  “sla\_priority”: 1,  "dependlist": [  "4010007:20160204-0000","4013002:20160204-0100"  ],  “dag” : [ {  “id” : “1”,  }]  }  }  The above specification corresponds to the following workflow:  4010007  4010007  1  2  1  4014523  4013002  1  1   |  |  |  | | --- | --- | --- | | **Attribute** | **Type** | **Description** | | actions | Array | An array of actions | | **Action instance object** | | | | id | string | *Unique identifier to identify action* | | default\_alloc | object | *[optional]* default resource allocation when there is no predication result.  Behavior: default\_alloc can be used to define required reservation resource for given action, it can be defined in each level. | | resources | array | *[optional]* an array of resource, with name and amount pair. For life time resource, it will inherit from plan level definitions. If action does not want to use any life time resource, it should define amount as 0. Any value with 0 or negative will be ignored. | | duration | string | *[optional]* The duration of resource this plan action will be used. | | Force\_use | boolean | *[optional]* If this flag is used, then planning engine will use default allocation and ignore getting predication. | | cluster | String | *[optional]* By default, all actions within plan will be placed within the single cluster defined from “default\_cluster” attribute. This parameter can overwrite the global default cluster attribute | | queue | String | *[optional]* By default, all actions within plan will be used under single queue defined from “default\_queue” attribute. This parameter can overwrite the global default queue attribute to make this particular action run within single queue. | | min\_start | long | *[Optional]* The earliest time at which this instance should be scheduled in the plan. | | max\_finish | long | *[Optional]* The latest time at which this instance should finish in the plan (i.e., the SLA of the action instance) | | sla\_priority | int | *[optional]* Priority of action. The default priority will be 0. Higher is better. High priority action SLA will be considered first during planning phase ahead of other actions. | | dependlist | array | List of orchestrator actions (indexed by their orchestrator Id) that this instance depends on. The planner will ensure that an instance is scheduled after the planned finish time of its dependent instance in the plan. | | dag | array | An array of dag sub action object | | dag:Life\_time\_resources | array | An array of life time resources used for sub action | | dag:id | string | Unique identifier to identify sub level action. Can use <actionid>.<subactionid> to address the corresponding sub action object, for instance, 4010007:20160204-0000.1 is the first step of TCC task instance 4010007:20160204-0000. | | dag:dependlist | array | An array of dependency sub actions within this sub dag object | |
| **Output** |
| JSON Response：  {  "rejected": [  "4014523:20160204-0000"  ]  }   |  |  |  | | --- | --- | --- | | **Attribute** | **Type** | **Description** | | rejected | array | List of action instances (indexed by their orchestrator Id) that the planner is not able to schedule before their requested max\_finish time. If empty, then all action instances have been successfully scheduled. | |

### Delete Plan

|  |
| --- |
| **Description** |
| Delete one plan or all plans within specified orchestrator or entire plans |
| **URL** |
| DELETE /planner/plans/${orchestrator\_name}  DELETE /planner/plans/${orchestrator\_name}/${plan\_name}  DELETE /planner/plans/ |
| **Input** |
| None |
| Output |
| 204: delete successfully |

### Update Orchestrator Action Status

|  |
| --- |
| **Description** |
| Update the planner about events relating to a given workflow action or sub action (e.g. start, finish, killed). This may trigger the planner to adjust the existing plan. Client orchestrators must at minimum notify the planner of actual action start and end times.   |  | | --- | | In TCC case, TCC Adapter will query the TCC database then post action events to Planner. | |
| **URL** |
| POST /planner/workloads/${orchestrator.name} |
| **Input** |
| JSON Request:  {  “status\_list”: [  {  "event": "START",  "time": 1456790400000,  “attemptid” : “.1”  “orchestrator\_action\_id”: “2583421:20160508-0000.1”,  “info” : {}  },  {  "event": "DONE",  "time": 1456790510000,  “attempted” : “.1”  “orchestrator\_action\_id”: “2583431:20160508-0000.1”,  “info” : {}  },  {  "event": "FAILED",  "time": 1456790100000,  “attemptid” : “.1”  “orchestrator\_action\_id”: “2583421:20160508-0000.2”,  "info": {}  }  {  "event": "START",  "time": 1456790100000,  “attemptid” : “”  “orchestrator\_action\_id”: “2583421:20160509-0000”,  "info": {}  }  ]  }   |  |  |  | | --- | --- | --- | | **Attribute** | **Type** | **Description** | | Status\_list | array | An array of status object | | event | string | Event type associated with the orchestrator action. Currently supporting:   * START * DONE (normal finish) * FAILED (abnormal finish) | | time | long | Event time. | | attemptid | string | Attempt id, number of times this action has been tried, the format will follow the same convention as workload ID definition. | | Orchestrator\_action\_id | string | Unique identifier. Can be action level or underlying sub dag level. In later case, id must include full sub dag path. | | info | object | Object used to pass additional event info, reserved for future use. | |
| **Output** |
| 201: created successfully |

### Query planned start time for all actions

|  |
| --- |
| **Description** |
| Query predicated time for all actions for given plan for all active actions (pending and running) based on orchestrator and plan name before given end time. |
| **URL** |
| GET /planner/status/actionstarttime/${orchestrator.name}/${plan.name}  GET /planner/status/actionstarttime  GET /planner/status/actionstarttime/${orchestrator.name}  GET /planner/status/actionstarttime/${orchestrator.name}/${plan.name}/${orchestrator\_action\_id} |
| **Input** |
| JSON Request:  {  “end\_time” : 1451607400000  }   |  |  |  | | --- | --- | --- | | **Attribute** | **Type** | **Description** | | End\_time | long | Time stamp of future time | |
| **Output** |
| JSON Response：  {  “action\_list” : [  {  “orchestrator\_name” : “tcc\_system”,  “plan\_name” : “critical\_report”,  “id” : “4010007:20160204-0000",  “plan\_start\_time” : 1451606400000  },  {  “orchestrator\_name” : “tcc\_system”,  “plan\_name” : “critical\_report”,  “id” : "4010007:20160204-0100",  “plan\_start\_time” : 1451606400000  }  ]  }   |  |  |  | | --- | --- | --- | | **Attribute** | **Type** | **Description** | | Action\_list | array | An array of action with id and its plan start time | | Action\_list:id | string | Action id | | Action\_list:orchestrator\_name | string | Orchestrator name of action | | Action\_list:plan\_name | string | Plan name of action | | Action\_list:Plan\_start\_time | long | Planned start time of the orchestrator action. | |

# Other Interfaces

## Hive Client Execution Variables

As described in section 3.1, the superior planner prediction module assumes that the Full Orchestrator workload ID is passed to every YARN workload in order to group multiple YARN workloads that are associated to the same orchestrator-level task together.

The assumption is that the following parameter is set in the YARN jobs:

|  |  |
| --- | --- |
| **Parameter Name** | **Value** |
| ss.planner.orchestrator.action.id | String containing the Full Orchestrator Action id of the workflow action instance that submitted this job. |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| In the case of TCC on Hive or MapReduce, this can be achieved as follows:   1. TCC task instance launcher should set the following environment variables in the TCC script execution environment:  |  |  | | --- | --- | | TCC\_TASK\_ID | The task ID, e.g. 4010007 | | TCC\_STEP\_NUMBER | The step number within the task. | | TCC\_PERIOD\_DATE | The period ID date portion, e.g. 20160204 | | TCC\_PERIOD\_HOUR | The period ID hour portion, e.g. 0400 | | TCC\_STEP\_ATTEMPT\_NUM | The retry count of step |  1. A wrapper script around the Hive / MapReduce execution launcher sets the above variable in the Hive execution environment using the --hiveconf option to beeline. The value should combine the above into a single string representing the Full orchestrator action ID is described in section [3.1](#_OrchestratorID_Format)   Here is an example code snippet:  beeline --hiveconf ss.planner.orchestrator.action.id="\  ${TCC\_TASK\_ID}:${TCC\_PERIOD\_DATE}-${TCC\_PERIOD\_HOUR}\  \.${TCC\_STEP\_NUMBER}:1.${TCC\_STEP\_ATTEMPT\_NUM}" |

## Required Hadoop Configurations

The planner module makes certain assumptions about the Hadoop installation that backs the orchestrator. These are listed here:

1. When using Hive on Map/Reduce, the Job History Server must be enabled
   * The planner’s prediction module depends on the job history logs generated by the Hive Map/Reduce workload to operate.

## Recommended Hadoop Configurations

There are certain Hadoop configurations that are not required, but are recommended for optimal operation of the planner. These are listed here:

1. mapreduce.job.reduce.slowstart.completedmaps = 1
   * The prediction and scheduling modules assume that reducers for a given job are only launched after all maps are completed.
   * In order to maximize the accuracy of initially created plans, the slow start feature should be disabled in order to respect this assumption.
   * This configuration is not required, just recommended to reduce future adjustments required by the planner to handle reducers that start earlier than planned.
2. hive.exec.parallel = false
   * Similarly, the prediction and scheduling modules assume that hive sub-jobs are submitted sequentially.
   * In order to maximize the accuracy of initially created plans, the parallel execution parameter in hive should be disabled.
   * As above, this configuration is not a hard requirement, just recommended to reduce future plan adjustments.