Promise user guide

Release 2015.1.1 (82e0ab3)

OPNFV

August 22, 2016

This user guide document provides the users with:

1. Short description of Promise project and key features implemented in Colorado
2. Instructions to execute Promise test cases manually

CHAPTER

ONE

PROMISE DESCRIPTION

Promise is a resource reservation and management project to identify NFV related requirements and realize resource reservation for future usage by capacity management of resource pools regarding compute, network and storage.

The following are the key features provided by this module:

- Capacity Management
- Reservation Management
- Allocation Management

CHAPTER

TWO

PROMISE CAPABILITIES AND USAGE

The Colorado implementation of Promise is built with the YangForge data modeling framework [1] , using a shim-layer on top of OpenStack to provide the Promise features. This approach requires communication between Consumers/Administrators and OpenStack to pass through the shim-layer. The shim-layer intercepts the message flow to manage the allocation requests based on existing reservations and available capacities in the providers. It also extracts information from the intercepted messages in order to update its internal databases. Furthermore, Promise provides additional intent-based APIs to allow a Consumer or Administrator to perform capacity management (i.e. add providers, update the capacity, and query the current capacity and utilization of a provider), reservation management (i.e. create, update, cancel, query reservations), and allocation management (i.e. create, destroy instances).

Detailed information about Promise use cases, features, interface specifications, work flows, and the underlying Promise YANG schema can be found in the Promise requirement document [2] .

2.1 Promise features and API usage guidelines and examples

This section lists the Promise features and API implemented in OPNFV Colorado.

Note: The listed parameters are optional unless explicitly marked as "mandatory".

2.1.1 Reservation management

The reservation management allows a Consumer to request reservations for resource capacity. Reservations can be for now or a later time window. After the start time of a reservation has arrived, the Consumer can issue create server instance requests against the reserved capacity. Note, a reservation will expire after a predefined *expiry* time in case no allocation referring to the reservation is requested.

The implemented workflow is well aligned with the described workflow in the Promise requirement document [1] (Section 6.1) except for the "multi-provider" scenario as described in *(Multi-)provider management* .

create-reservation

This operation allows making a request to the reservation system to reserve resources.

The operation takes the following input parameters:

- start: start time of the requested reservation
- end: end time of the requested reservation
- capacity.instances: amount of instances to be reserved

[1] YangForge framework, http://github.com/opnfv/yangforge
[2] Promise requirement document, http://artifacts.opnfv.org/promise/docs/requirements/index.html

- capacity.cores: amount of cores to be reserved
- capacity.ram: amount of ram in MB to be reserved

Promise will check the available capacity in the given time window and in case sufficient capacity exists to meet the reservation request, will mark those resources “reserved” in its reservation map.

update-reservation

This operation allows to update the reservation details for an existing reservation.

It can take the same input parameters as in *create-reservation* but in addition requires a mandatory reference to the *reservation-id* of the reservation that shall be updated.

cancel-reservation

This operation is used to cancel an existing reservation.

The operation takes the following input parameter:

- reservation-id (mandatory): identifier of the reservation to be canceled.

query-reservation

The operation queries the reservation system to return reservation(s) matching the specified query filter, e.g., reservations that are within a specified start/end time window.

The operation takes the following input parameters to narrow down the query results:

- without: excludes specified collection identifiers from the result
- elements.some: query for ResourceCollection(s) that contain some or more of these element(s)
- elements.every: query for ResourceCollection(s) that contain all of these element(s)
- window.start: matches entries that are within the specified start/
- window.end: end time window
- **window.scope: if set to ‘exclusive’, only reservations with start AND end time** within the time window are returned. Otherwise (‘inclusive’), all reservation starting OR ending in the time windows are returned.
- show-utilization: boolean value that specifies whether to also return the resource utilization in the queried time window or not

2.1.2 Allocation management

create-instance

This operation is used to create an instance of specified resource(s) for immediate use utilizing capacity from the pool. *Create-instance* requests can be issued against an existing reservation, but also allocations without a reference to an existing reservation are allowed. In case the allocation request specifies a reservation identifier, Promise checks if a reservation with that ID exists, the reservation start time has arrived (i.e. the reservation is ‘active’), and the required capacity for the requested flavor is within the available capacity of the reservation. If those conditions are met, Promise creates a record for the allocation (VMState=”INITIALIZED”) and update its databases. If no *reservation_id* was provided in the allocation request, Promise checks whether the required capacity to meet the request can be provided from the available, non-reserved capacity. If yes, Promise creates a record for the allocation with an unique *instance-id* and update its databases. In any other case, Promise rejects the *create-instance* request.

In case the *create-instance* request is rejected, Promise responds with a "status=rejected" providing the reason of the rejection. This will help the Consumer to take appropriate actions, e.g., send an updated *create-instance* request. In case the *create-instance* request was accepted and a related allocation record has been created, the shim-layer issues a *createServer* request to the VIM Controller (i.e. Nova) providing all information to create the server instance.

The operation takes the following input parameters:

- name (mandatory): Assigned name for the instance to be created
- image (mandatory): the image to be booted in the new instance
- flavor (mandatory): the flavor of the requested server instance
- networks: the list of network uuids of the requested server instance
- provider-id: identifier of the provider where the instance shall be created
- reservation-id: identifier of a resource reservation the *create-instance*

The Colorado implementation of Promise has the following limitations:

- All create server instance requests shall pass through the Promise shim-layer such that Promise can keep track of all allocation requests. This is necessary as in the current release the sychronization between the VIM Controller and Promise on the available capacity is not yet implemented.
- *Create-allocation* requests are limited to "simple" allocations, i.e., the current workflow only supports the Nova compute service and *create-allocation* requests are limited to creating one server instance at a time
- Prioritization of reservations and allocations is yet not implemented. Future version may allow certain policy-based conflict resolution where, e.g., new allocation request with high priority can "forcefully" terminate lower priority allocations.

destroy-instance

This operation request to destroy an existing server instance and release it back to the pool.

The operation takes the following input parameter:

- instance-id: identifier of the server instance to be destroyed

2.1.3 Capacity management

The capacity management feature allows the Consumer or Administrator to do capacity planning, i.e. the capacity available to the reservation management can differ from the actual capacity in the registered provider(s). This feature can, e.g., be used to limit the available capacity for a given time window due to a planned downtime of some of the resources, or increase the capacity available to the reservation system in case of a planned upgrade of the available capacity.

increase/decrease-capacity

This operations allows to increase/decrease the total capacity that is made available to the Promise reservation service between a specified window in time. It does NOT increase the actual capacity of a given resource provider, but is used for capacity management inside Promise.

This feature can be used in different ways, like

- Limit the capacity available to the reservation system to a value below 100% of the available capacity in the VIM, e.g., in order to leave "buffer" in the actual NFVI to be used outside the Promise reservation service.
- Inform the reservation system that, from a given time in the future, additional resources can be reserved, e.g., due to a planned upgrade of the available capacity of the provider.
- Similarily, the "decrease-capacity" can be used to reduce the consumable resources in a given time window, e.g., to prepare for a planned downtime of some of the resources.
- Expose multiple reservation service instances to different consumers sharing the same resource provider.

The operation takes the following input parameters:

- start: start time for the increased/decreased capacity
- end: end time for the increased/decreased capacity
- capacity.cores: Increased/decreased amount of cores
- capacity.ram: Increased/decreased amount of RAM
- capacity.instances: Increased/decreased amount of instances

Note, increase/decreasing the capacity in Promise is completely transparent to the VIM. As such, when increasing the virtual capacity in Promise (e.g. for a planned upgrade of the capacity), it is in the responsibility of the Consumer/Administrator to ensure sufficient resources in the VIM are available at the appropriate time, in order to prevent allocations against reservations to fail due to a lack of resources. Therefore, this operations should only be used carefully.

query-capacity

This operation is used to query the available capacity information of the specified resource collection. A filter attribute can be specified to narrow down the query results.

The current implementation supports the following filter criteria:

- time window: returns reservations matching the specified window
- window scope: if set to 'exclusive', only reservations with start AND end time within the time window are returned. Otherwise, all reservation starting OR ending in the time windows are returned.
- metric: query for one of the following capacity metrics:
  - 'total': resource pools
  - 'reserved': reserved resources
  - 'usage': resource allocations
  - 'available': remaining capacity, i.e. neither reserved nor allocated

2.1.4 (Multi-)provider management

This API towards OpenStack allows a Consumer/Administrator to add and remove resource providers to Promise. Note, Promise supports a multi-provider configuration, however, for Colorado, multi-provider support is not yet fully supported.

add-provider

This operation is used to register a new resource provider into the Promise reservation system.

Note, for Colorado, the add-provider operation should only be used to register one provider with the Promise shim-layer. Further note that currently only OpenStack is supported as a provider.

The operation takes the following input parameters:

- provider-type (mandatory) = ‘openstack’: select a specific resource provider type.
- endpoint (mandatory): target URL endpoint for the resource provider.
- username (mandatory)
- password (mandatory)
- region: specified region for the provider
- tenant.id: id of the OpenStack tenant/project
- tenant.name: name of the OpenStack tenant/project

CHAPTER

THREE

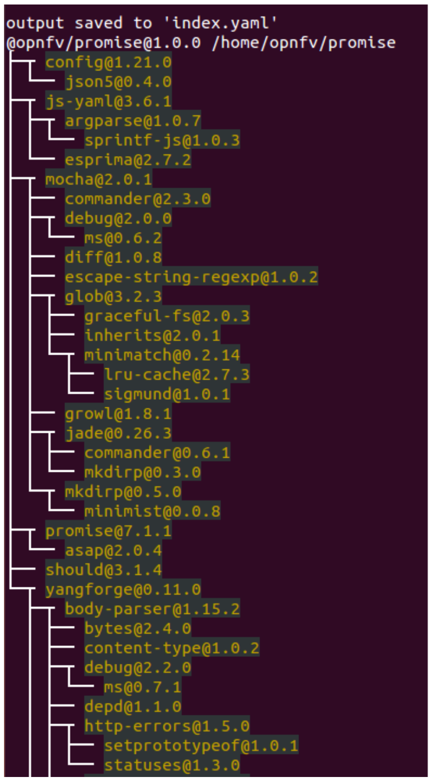
MANUAL TESTING

3.1 Promise installation

Install nodejs, npm and promise

```
curl -sL https://deb.nodesource.com/setup_4.x | sudo -E bash -
sudo apt-get install -y nodejs
sudo npm -g install npm@latest
git clone https://github.com/opnfv/promise.git
cd promise
npm install
```

Please note that the last command 'npm install' will install all needed dependencies for promise (including yangforge and mocha)



```
output saved to 'index.yaml'
@opnfv/promise@1.0.0 /home/opnfv/promise
├─┬ config@1.21.0
│ └── json5@0.4.0
├─┬ js-yaml@3.6.1
│ ├─┬ argparse@1.0.7
│ │ └── sprintf-js@1.0.3
│ └── esprima@2.7.2
├─┬ mocha@2.0.1
│ ├── commander@2.3.0
│ ├─┬ debug@2.0.0
│ │ └── ms@0.6.2
│ ├── diff@1.0.8
│ ├── escape-string-regexp@1.0.2
│ ├─┬ glob@3.2.3
│ │ ├── graceful-fs@2.0.3
│ │ ├── inherits@2.0.1
│ │ └─┬ minimatch@0.2.14
│ │   ├── lru-cache@2.7.3
│ │   └── sigmund@1.0.1
│ ├── growl@1.8.1
│ ├─┬ jade@0.26.3
│ │ ├── commander@0.6.1
│ │ └── mkdirp@0.3.0
│ └─┬ mkdirp@0.5.0
│   └── minimist@0.0.8
├─┬ promise@7.1.1
│ └── asap@2.0.4
├── should@3.1.4
└─┬ yangforge@0.11.0
  ├─┬ body-parser@1.15.2
  │ ├── bytes@2.4.0
  │ ├── content-type@1.0.2
  │ ├─┬ debug@2.2.0
  │ │ └── ms@0.7.1
  │ ├── depd@1.1.0
  │ ├─┬ http-errors@1.5.0
  │ │ ├── setprototypeof@1.0.1
  │ │ └── statuses@1.3.0
```

3.2 Validation

Please perform the following preparation steps:

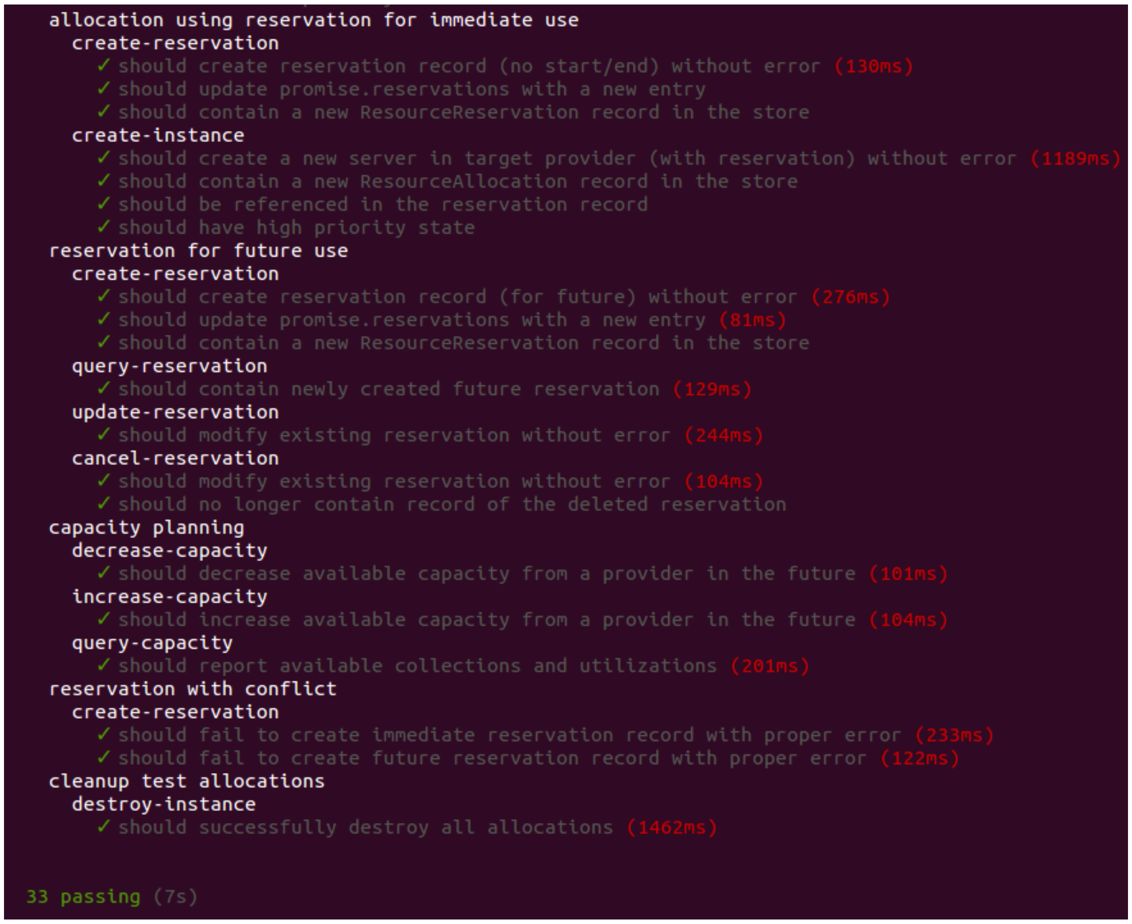
1. Set OpenStack environment parameters properly (e.g. source openrc admin demo in DevStack)
2. Create OpenStack tenant (e.g. promise) and tenant user (e.g. promiser)
3. Create a flavor in Nova with 1 vCPU and 512 MB RAM
4. Create a private network, subnet and router in Neutron
5. Create an image in Glance

Once done, the promise test script can be invoked as follows (as a single line command):

```
NODE_ENV=mytest \
OS_TENANT_NAME=promise \
OS_USERNAME=promiser \
OS_PASSWORD=<user password from Step 2> \
OS_TEST_FLAVOR=<flavor ID from Step 3> \
OS_TEST_NETWORK=<network ID from Step 4> \
OS_TEST_IMAGE=<image ID from Step 5> \
npm run -s test -- --reporter json > promise-results.json
```

The results of the tests will be stored in the promise-results.json file.

The results can also be seen in the console ("npm run -s test")



```
  allocation using reservation for immediate use
    create-reservation
      ✓ should create reservation record (no start/end) without error (130ms)
      ✓ should update promise.reservations with a new entry
      ✓ should contain a new ResourceReservation record in the store
    create-instance
      ✓ should create a new server in target provider (with reservation) without error (1189ms)
      ✓ should contain a new ResourceAllocation record in the store
      ✓ should be referenced in the reservation record
      ✓ should have high priority state
  reservation for future use
    create-reservation
      ✓ should create reservation record (for future) without error (276ms)
      ✓ should update promise.reservations with a new entry (81ms)
      ✓ should contain a new ResourceReservation record in the store
    query-reservation
      ✓ should contain newly created future reservation (129ms)
    update-reservation
      ✓ should modify existing reservation without error (244ms)
    cancel-reservation
      ✓ should modify existing reservation without error (104ms)
      ✓ should no longer contain record of the deleted reservation
  capacity planning
    decrease-capacity
      ✓ should decrease available capacity from a provider in the future (101ms)
    increase-capacity
      ✓ should increase available capacity from a provider in the future (104ms)
    query-capacity
      ✓ should report available collections and utilizations (201ms)
  reservation with conflict
    create-reservation
      ✓ should fail to create immediate reservation record with proper error (233ms)
      ✓ should fail to create future reservation record with proper error (122ms)
  cleanup test allocations
    destroy-instance
      ✓ should successfully destroy all allocations (1462ms)


33 passing (7s)
```

All 33 tests passing?! Congratulations, promise has been successfully installed and configured.