DOCTOR

Release Latest

Nov 13, 2018

Contents

CHAPTER 1

Doctor

1.1 Doctor User Guide

1.1.1 Doctor capabilities and usage

figure1 shows the currently implemented and tested architecture of Doctor. The implementation is based on OpenStack and related components. The Monitor can be realized by a sample Python-based implementation provided in the Doctor code repository. The Controller is realized by OpenStack Nova, Neutron and Cinder for compute, network and storage, respectively. The Inspector can be realized by OpenStack Congress, Vitrage or a sample Python-based implementation also available in the code repository of Doctor. The Notifier is realized by OpenStack Aodh.

Immediate Notification

Immediate notification can be used by creating 'event' type alarm via OpenStack Alarming (Aodh) API with relevant internal components support.

See: - Upstream spec document: https://specs.openstack.org/openstack/ceilometer-specs/specs/liberty/event-alarm-evaluator.html - Aodh official documentation: https://docs.openstack.org/aodh/latest

An example of a consumer of this notification can be found in the Doctor repository. It can be executed as follows:

```
git clone https://gerrit.opnfv.org/gerrit/doctor
cd doctor/doctor_tests/consumer
CONSUMER_PORT=12346
python sample.py "$CONSUMER_PORT" > consumer.log 2>&1 &
```

Consistent resource state awareness

Resource state of compute host can be changed/updated according to a trigger from a monitor running outside of OpenStack Compute (Nova) by using force-down API.

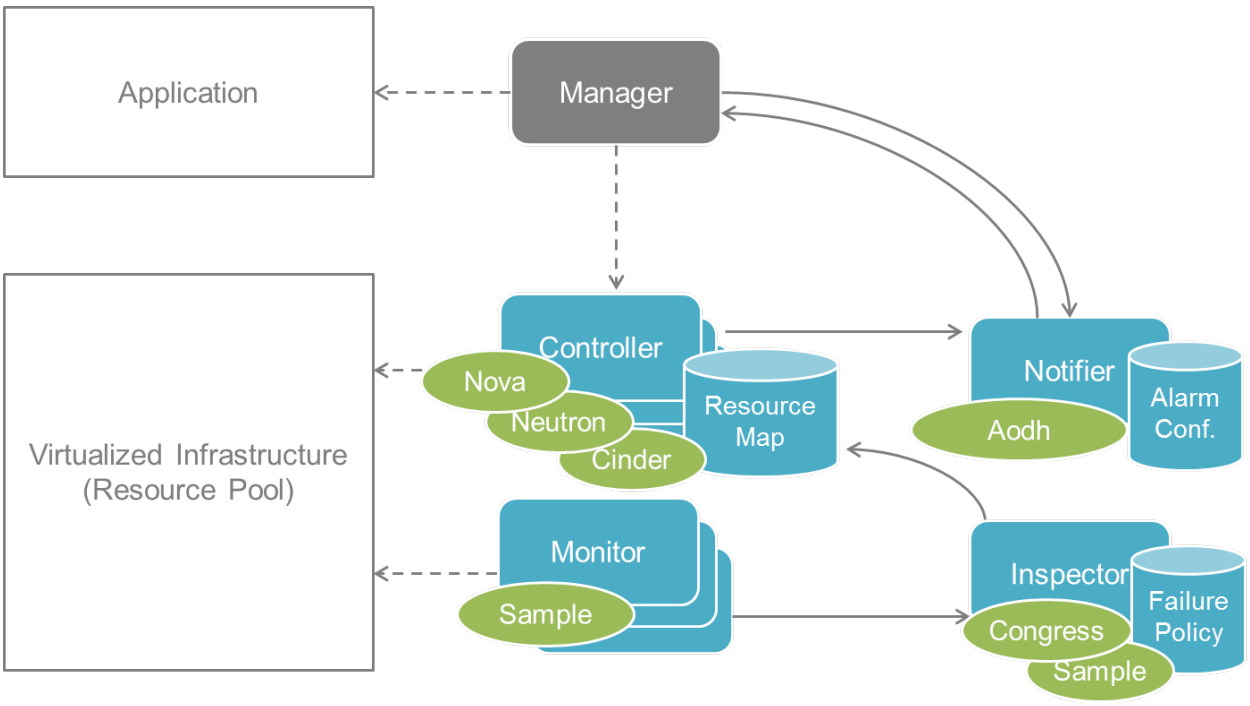


Fig. 1: Implemented and tested architecture

See: * Upstream spec document: https://specs.openstack.org/openstack/nova-specs/specs/liberty/implemented/mark-host-down.html * Upstream Compute API reference document: https://developer.openstack.org/api-ref/compute * Doctor Mark Host Down Manual: https://git.opnfv.org/doctor/tree/docs/development/manuals/mark-host-down_manual.rst

Valid compute host status given to VM owner

The resource state of a compute host can be retrieved by a user with the OpenStack Compute (Nova) servers API.

See: * Upstream spec document: https://specs.openstack.org/openstack/nova-specs/specs/mitaka/implemented/get-valid-server-state.html * Upstream Compute API reference document: https://developer.openstack.org/api-ref/compute * Doctor Get Valid Server State Manual: https://git.opnfv.org/doctor/tree/docs/development/manuals/get-valid-server-state.rst

Port data plane status update

Port data plane status can be changed/updated in the case of issues in the underlying data plane affecting connectivity from/to Neutron ports.

See: * Upstream spec document: https://specs.openstack.org/openstack/neutron-specs/specs/pike/port-data-plane-status.html * Upstream Networking API reference document: https://developer.openstack.org/api-ref/network

Doctor driver (Congress)

The Doctor driver can be notified about NFVI failures that have been detected by monitoring systems.

See: * Upstream spec document: https://specs.openstack.org/openstack/congress-specs/specs/mitaka/push-type-datasource-driver.html * Congress official documentation: https://docs.openstack.org/congress/latest

Event API (Vitrage)

With this API, monitoring systems can push events to the Doctor datasource.

See: * Upstream spec document: https://specs.openstack.org/openstack/vitrage-specs/specs/ocata/event-api.html * Vitrage official documentation: https://docs.openstack.org/vitrage/latest

Doctor datasource (Vitrage)

After receiving events from monitoring systems, the Doctor datasource identifies the affected resources based on the resource topology.

See: * Upstream spec document: https://specs.openstack.org/openstack/vitrage-specs/specs/ocata/doctor-datasource.html

CHAPTER 2

Doctor

2.1 Design Documents

This is the directory to store design documents which may include draft versions of blueprints written before proposing to upstream OSS communities such as OpenStack, in order to keep the original blueprint as reviewed in OPNFV. That means there could be out-dated blueprints as result of further refinements in the upstream OSS community. Please refer to the link in each document to find the latest version of the blueprint and status of development in the relevant OSS community.

See also https://wiki.opnfv.org/requirements_projects .

Note: This is a specification draft of a blueprint proposed for OpenStack Nova Liberty. It was written by project member(s) and agreed within the project before submitting it upstream. No further changes to its content will be made here anymore; please follow it upstream:

- Current version upstream: https://review.openstack.org/#/c/169836/
- Development activity: https://blueprints.launchpad.net/nova/+spec/mark-host-down

Original draft is as follow:

2.1.1 Report host fault to update server state immediately

https://blueprints.launchpad.net/nova/+spec/update-server-state-immediately

A new API is needed to report a host fault to change the state of the instances and compute node immediately. This allows usage of evacuate API without a delay. The new API provides the possibility for external monitoring system to detect any kind of host failure fast and reliably and inform OpenStack about it. Nova updates the compute node state and states of the instances. This way the states in the Nova DB will be in sync with the real state of the system.

Problem description

- Nova state change for failed or unreachable host is slow and does not reliably state compute node is down or not. This might cause same instance to run twice if action taken to evacuate instance to another host.
- Nova state for instances on failed compute node will not change, but remains active and running. This gives user a false information about instance state. Currently one would need to call "nova reset-state" for each instance to have them in error state.
- OpenStack user cannot make HA actions fast and reliably by trusting instance state and compute node state.
- As compute node state changes slowly one cannot evacuate instances.

Use Cases

Use case in general is that in case there is a host fault one should change compute node state fast and reliably when using DB servicegroup backend. On top of this here is the use cases that are not covered currently to have instance states changed correctly: * Management network connectivity lost between controller and compute node. * Host HW failed.

Generic use case flow:

- The external monitoring system detects a host fault.
- The external monitoring system fences the host if not down already.
- The external system calls the new Nova API to force the failed compute node into down state as well as instances running on it.
- Nova updates the compute node state and state of the effected instances to Nova DB.

Currently nova-compute state will be changing "down", but it takes a long time. Server state keeps as "vm_state: active" and "power_state: running", which is not correct. By having external tool to detect host faults fast, fence host by powering down and then report host down to OpenStack, all these states would reflect to actual situation. Also if OpenStack will not implement automatic actions for fault correlation, external tool can do that. This could be configured for example in server instance METADATA easily and be read by external tool.

Project Priority

Liberty priorities have not yet been defined.

Proposed change

There needs to be a new API for Admin to state host is down. This API is used to mark compute node and instances running on it down to reflect the real situation.

Example on compute node is:

- When compute node is up and running: vm_state: active and power_state: running nova-compute state: up status: enabled
- When compute node goes down and new API is called to state host is down: vm_state: stopped power_state: shutdown nova-compute state: down status: enabled

vm_state values: soft-delete, deleted, resized and error should not be touched. task_state effect needs to be worked out if needs to be touched.

Alternatives

There is no attractive alternatives to detect all different host faults than to have a external tool to detect different host faults. For this kind of tool to exist there needs to be new API in Nova to report fault. Currently there must have been some kind of workarounds implemented as cannot trust or get the states from OpenStack fast enough.

Data model impact

None

REST API impact

- Update CLI to report host is down

  nova host-update command

  usage: nova host-update [–status <enable|disable>] [–maintenance <enable|disable>] [–report-host-down] <hostname>

  Update host settings.

  Positional arguments

  <hostname> Name of host.

  Optional arguments

  –status <enable|disable> Either enable or disable a host.

  –maintenance <enable|disable> Either put or resume host to/from maintenance.

  –down Report host down to update instance and compute node state in db.

- Update Compute API to report host is down:

  /v2.1/{tenant_id}/os-hosts/{host_name}

  Normal response codes: 200 Request parameters

  Parameter Style Type Description host_name URI xsd:string The name of the host of interest to you.

  {

  “host”: { “status”: “enable”, “maintenance_mode”: “enable” “host_down_reported”: “true”

  }

  }

  {

  “host”: { “host”: “65c5d5b7e3bd44308e67fc50f362aee6”, “maintenance_mode”: “enabled”, “status”: “enabled” “host_down_reported”: “true”

  }

  }

- New method to nova.compute.api module HostAPI class to have a to mark host related instances and compute node down: set_host_down(context, host_name)
- class novaclient.v2.hosts.HostManager(api) method update(host, values) Needs to handle reporting host down.
- Schema does not need changes as in db only service and server states are to be changed.

Security impact

API call needs admin privileges (in the default policy configuration).

Notifications impact

None

Other end user impact

None

Performance Impact

Only impact is that user can get information faster about instance and compute node state. This also gives possibility to evacuate faster. No impact that would slow down. Host down should be rare occurrence.

Other deployer impact

Developer can make use of any external tool to detect host fault and report it to OpenStack.

Developer impact

None

Implementation

Assignee(s)

Primary assignee: Tomi Juvonen Other contributors: Ryota Mibu

Work Items

- Test cases.
- API changes.
- Documentation.

Dependencies

None

Testing

Test cases that exists for enabling or putting host to maintenance should be altered or similar new cases made test new functionality.

Documentation Impact

New API needs to be documented:

- Compute API extensions documentation. http://developer.openstack.org/api-ref-compute-v2.1.html
- Nova commands documentation. http://docs.openstack.org/user-guide-admin/content/novaclient_commands.html
- Compute command-line client documentation. http://docs.openstack.org/cli-reference/content/novaclient_commands.html
- nova.compute.api documentation. http://docs.openstack.org/developer/nova/api/nova.compute.api.html
- High Availability guide might have page to tell external tool could provide ability to provide faster HA as able to update states by new API. http://docs.openstack.org/high-availability-guide/content/index.html

References

- OPNFV Doctor project: https://wiki.opnfv.org/doctor
- OpenStack Instance HA Proposal: http://blog.russellbryant.net/2014/10/15/openstack-instance-ha-proposal/
- The Different Facets of OpenStack HA: http://blog.russellbryant.net/2015/03/10/ the-different-facets-of-openstack-ha/

2.1.2 Notification Alarm Evaluator

Note: This is spec draft of blueprint for OpenStack Ceilomter Liberty. To see current version: https://review.openstack.org/172893 To track development activity: https://blueprints.launchpad.net/ceilometer/+spec/notification-alarm-evaluator

https://blueprints.launchpad.net/ceilometer/+spec/notification-alarm-evaluator

This blueprint proposes to add a new alarm evaluator for handling alarms on events passed from other OpenStack services, that provides event-driven alarm evaluation which makes new sequence in Ceilometer instead of the polling-based approach of the existing Alarm Evaluator, and realizes immediate alarm notification to end users.

Problem description

As an end user, I need to receive alarm notification immediately once Ceilometer captured an event which would make alarm fired, so that I can perform recovery actions promptly to shorten downtime of my service. The typical use case is that an end user set alarm on "compute.instance.update" in order to trigger recovery actions once the instance status has changed to 'shutdown' or 'error'. It should be nice that an end user can receive notification within 1 second after fault observed as the same as other helth- check mechanisms can do in some cases.

The existing Alarm Evaluator is periodically querying/polling the databases in order to check all alarms independently from other processes. This is good approach for evaluating an alarm on samples stored in a certain period. However, this is not efficient to evaluate an alarm on events which are emitted by other OpenStack servers once in a while.

The periodical evaluation leads delay on sending alarm notification to users. The default period of evaluation cycle is 60 seconds. It is recommended that an operator set longer interval than configured pipeline interval for underlying metrics, and also longer enough to evaluate all defined alarms in certain period while taking into account the number of resources, users and alarms.

Proposed change

The proposal is to add a new event-driven alarm evaluator which receives messages from Notification Agent and finds related Alarms, then evaluates each alarms;

- New alarm evaluator could receive event notification from Notification Agent by which adding a dedicated notifier as a publisher in pipeline.yaml (e.g. notifier://?topic=event_eval).
- When new alarm evaluator received event notification, it queries alarm database by Project ID and Resource ID written in the event notification.
- Found alarms are evaluated by referring event notification.
- Depending on the result of evaluation, those alarms would be fired through Alarm Notifier as the same as existing Alarm Evaluator does.

This proposal also adds new alarm type "notification" and "notification_rule". This enables users to create alarms on events. The separation from other alarm types (such as "threshold" type) is intended to show different timing of evaluation and different format of condition, since the new evaluator will check each event notification once it received whereas "threshold" alarm can evaluate average of values in certain period calculated from multiple samples.

The new alarm evaluator handles Notification type alarms, so we have to change existing alarm evaluator to exclude "notification" type alarms from evaluation targets.

Alternatives

There was similar blueprint proposal "Alarm type based on notification", but the approach is different. The old proposal was to adding new step (alarm evaluations) in Notification Agent every time it received event from other OpenStack services, whereas this proposal intends to execute alarm evaluation in another component which can minimize impact to existing pipeline processing.

Another approach is enhancement of existing alarm evaluator by adding notification listener. However, there are two issues; 1) this approach could cause stall of periodical evaluations when it receives bulk of notifications, and 2) this could break the alarm portioning i.e. when alarm evaluator received notification, it might have to evaluate some alarms which are not assign to it.

Data model impact

Resource ID will be added to Alarm model as an optional attribute. This would help the new alarm evaluator to filter out non-related alarms while querying alarms, otherwise it have to evaluate all alarms in the project.

REST API impact

Alarm API will be extended as follows;

- Add "notification" type into alarm type list
- Add "resource_id" to "alarm"
- Add "notification_rule" to "alarm"

Sample data of Notification-type alarm:

```
{
    "alarm_actions": [
        "http://site:8000/alarm"
```

(continues on next page)

(continued from previous page)


```
    ],
    "alarm_id": null,
    "description": "An alarm",
    "enabled": true,
    "insufficient_data_actions": [
        "http://site:8000/nodata"
    ],
    "name": "InstanceStatusAlarm",
    "notification_rule": {
        "event_type": "compute.instance.update",
        "query" : [
            {
                "field" : "traits.state",
                "type" : "string",
                "value" : "error",
                "op" : "eq",
            },
        ]
    },
    "ok_actions": [],
    "project_id": "c96c887c216949acbdfbd8b494863567",
    "repeat_actions": false,
    "resource_id": "153462d0-a9b8-4b5b-8175-9e4b05e9b856",
    "severity": "moderate",
    "state": "ok",
    "state_timestamp": "2015-04-03T17:49:38.406845",
    "timestamp": "2015-04-03T17:49:38.406839",
    "type": "notification",
    "user_id": "c96c887c216949acbdfbd8b494863567"
}
```


"resource_id" will be refered to query alarm and will not be check permission and belonging of project.

Security impact

None

Pipeline impact

None

Other end user impact

None

Performance/Scalability Impacts

When Ceilomter received a number of events from other OpenStack services in short period, this alarm evaluator can keep working since events are queued in a messaging queue system, but it can cause delay of alarm notification to users and increase the number of read and write access to alarm database.

"resource_id" can be optional, but restricting it to mandatory could be reduce performance impact. If user create "notification" alarm without "resource_id", those alarms will be evaluated every time event occurred in the project. That may lead new evaluator heavy.

Other deployer impact

New service process have to be run.

Developer impact

Developers should be aware that events could be notified to end users and avoid passing raw infra information to end users, while defining events and traits.

Implementation

Assignee(s)

Primary assignee: r-mibu

Other contributors: None

Ongoing maintainer: None

Work Items

- New event-driven alarm evaluator
- Add new alarm type "notification" as well as AlarmNotificationRule
- Add "resource_id" to Alarm model
- Modify existing alarm evaluator to filter out "notification" alarms
- Add new config parameter for alarm request check whether accepting alarms without specifying "resource_id" or not

Future lifecycle

This proposal is key feature to provide information of cloud resources to end users in real-time that enables efficient integration with user-side manager or Orchestrator, whereas currently those information are considered to be consumed by admin side tool or service. Based on this change, we will seek orchestrating scenarios including fault recovery and add useful event definition as well as additional traits.

Dependencies

None

Testing

New unit/scenario tests are required for this change.

Documentation Impact

- Proposed evaluator will be described in the developer document.
- New alarm type and how to use will be explained in user guide.

References

- OPNFV Doctor project: https://wiki.opnfv.org/doctor
- Blueprint "Alarm type based on notification": https://blueprints.launchpad.net/ceilometer/+spec/alarm-on-notification

2.1.3 Neutron Port Status Update

Note: This document represents a Neutron RFE reviewed in the Doctor project before submitting upstream to Launchpad Neutron space. The document is not intended to follow a blueprint format or to be an extensive document. For more information, please visit http://docs.openstack.org/developer/neutron/policies/blueprints.html

The RFE was submitted to Neutron. You can follow the discussions in https://bugs.launchpad.net/neutron/+bug/1598081

Neutron port status field represents the current status of a port in the cloud infrastructure. The field can take one of the following values: 'ACTIVE', 'DOWN', 'BUILD' and 'ERROR'.

At present, if a network event occurs in the data-plane (e.g. virtual or physical switch fails or one of its ports, cable gets pulled unintentionally, infrastructure topology changes, etc.), connectivity to logical ports may be affected and tenants' services interrupted. When tenants/cloud administrators are looking up their resources' status (e.g. Nova instances and services running in them, network ports, etc.), they will wrongly see everything looks fine. The problem is that Neutron will continue reporting port 'status' as 'ACTIVE'.

Many SDN Controllers managing network elements have the ability to detect and report network events to upper layers. This allows SDN Controllers' users to be notified of changes and react accordingly. Such information could be consumed by Neutron so that Neutron could update the 'status' field of those logical ports, and additionally generate a notification message to the message bus.

However, Neutron misses a way to be able to receive such information through e.g. ML2 driver or the REST API ('status' field is read-only). There are pros and cons on both of these approaches as well as other possible approaches. This RFE intends to trigger a discussion on how Neutron could be improved to receive fault/change events from SDN Controllers or even also from 3rd parties not in charge of controlling the network (e.g. monitoring systems, human admins).

2.1.4 Port data plane status

https://bugs.launchpad.net/neutron/+bug/1598081

Neutron does not detect data plane failures affecting its logical resources. This spec addresses that issue by means of allowing external tools to report to Neutron about faults in the data plane that are affecting the ports. A new REST API field is proposed to that end.

Problem Description

An initial description of the problem was introduced in bug #159801 [*1*]. This spec focuses on capturing one (main) part of the problem there described, i.e. extending Neutron's REST API to cover the scenario of allowing external tools to report network failures to Neutron. Out of scope of this spec are works to enable port status changes to be received and managed by mechanism drivers.

This spec also tries to address bug #1575146 [*2*]. Specifically, and argued by the Neutron driver team in [*3*]:

- Neutron should not shut down the port completly upon detection of physnet failure; connectivity between instances on the same node may still be reachable. Externals tools may or may not want to trigger a status change on the port based on their own logic and orchestration.
- Port down is not detected when an uplink of a switch is down;
- The physnet bridge may have multiple physical interfaces plugged; shutting down the logical port may not be needed in case network redundancy is in place.

Proposed Change

A couple of possible approaches were proposed in [*1*] (comment #3). This spec proposes tackling the problema via a new extension API to the port resource. The extension adds a new attribute 'dp-down' (data plane down) to represent the status of the data plane. The field should be read-only by tenants and read-write by admins.

Neutron should send out an event to the message bus upon toggling the data plane status value. The event is relevant for e.g. auditing.

Data Model Impact

A new attribute as extension will be added to the 'ports' table.

Attribute Name	Type	Access	Default Value	Validation/ Conversion	Description
dp_down	boolean	RO, tenant RW, admin	False	True/False	

REST API Impact

A new API extension to the ports resource is going to be introduced.

```
EXTENDED_ATTRIBUTES_2_0 = {
    'ports': {
        'dp_down': {'allow_post': False, 'allow_put': True,
                    'default': False, 'convert_to': convert_to_boolean,
                    'is_visible': True},
    },
}
```

Examples

Updating port data plane status to down:

```
PUT /v2.0/ports/<port-uuid>
Accept: application/json
{
    "port": {
        "dp_down": true
    }
}
```

Command Line Client Impact

```
neutron port-update [--dp-down <True/False>] <port>
openstack port set [--dp-down <True/False>] <port>
```

Argument –dp-down is optional. Defaults to False.

Security Impact

None

Notifications Impact

A notification (event) upon toggling the data plane status (i.e. 'dp-down' attribute) value should be sent to the message bus. Such events do not happen with high frequency and thus no negative impact on the notification bus is expected.

Performance Impact

None

IPv6 Impact

None

Other Deployer Impact

None

Developer Impact

None

Implementation

Assignee(s)

- cgoncalves

Work Items

- New 'dp-down' attribute in 'ports' database table
- API extension to introduce new field to port
- Client changes to allow for data plane status (i.e. 'dp-down' attribute') being set
- Policy (tenants read-only; admins read-write)

Documentation Impact

Documentation for both administrators and end users will have to be contemplated. Administrators will need to know how to set/unset the data plane status field.

References

2.1.5 Inspector Design Guideline

Note: This is spec draft of design guideline for inspector component. JIRA ticket to track the update and collect comments: DOCTOR-73.

This document summarize the best practise in designing a high performance inspector to meet the requirements in OPNFV Doctor project.

Problem Description

Some pitfalls has be detected during the development of sample inspector, e.g. we suffered a significant performance degrading in listing VMs in a host.

A patch set for caching the list has been committed to solve issue. When a new inspector is integrated, it would be nice to have an evaluation of existing design and give recommendations for improvements.

This document can be treated as a source of related blueprints in inspector projects.

Guidelines

Host specific VMs list

While requirement in doctor project is to have alarm about fault to consumer in one second, it is just a limit we have set in requirements. When talking about fault management in Telco, the implementation needs to be by all means optimal and the one second is far from traditional Telco requirements.

One thing to be optimized in inspector is to eliminate the need to read list of host specific VMs from Nova API, when it gets a host specific failure event. Optimal way of implementation would be to initialize this list when Inspector start by reading from Nova API and after this list would be kept up-to-date by `instance.update` notifications received from nova. Polling Nova API can be used as a complementary channel to make snapshot of hosts and VMs list in order to keep the data consistent with reality.

This is enhancement and not perhaps something needed to keep under one second in a small system. Anyhow this would be something needed in case of production use.

This guideline can be summarized as following:

- cache the host VMs mapping instead of reading it on request
- subscribe and handle update notifications to keep the list up to date
- make snapshot periodically to ensure data consistency

Parallel execution

In doctor's architecture, the inspector is responsible to set error state for the affected VMs in order to notify the consumers of such failure. This is done by calling the nova reset-state API. However, this action is a synchronous request with many underlying steps and cost typically hundreds of milliseconds. According to the discussion in mailing list, this time cost will grow linearly if the requests are sent one by one. It will become a critical issue in large scale system.

It is recommended to introduce **parallel execution** for actions like `reset-state` that takes a list of targets.

Shortcut notification

An alternative way to improve notification performance is to take a shortcut from inspector to notifier instead of triggering it from controller. The difference between the two workflow is shown below:

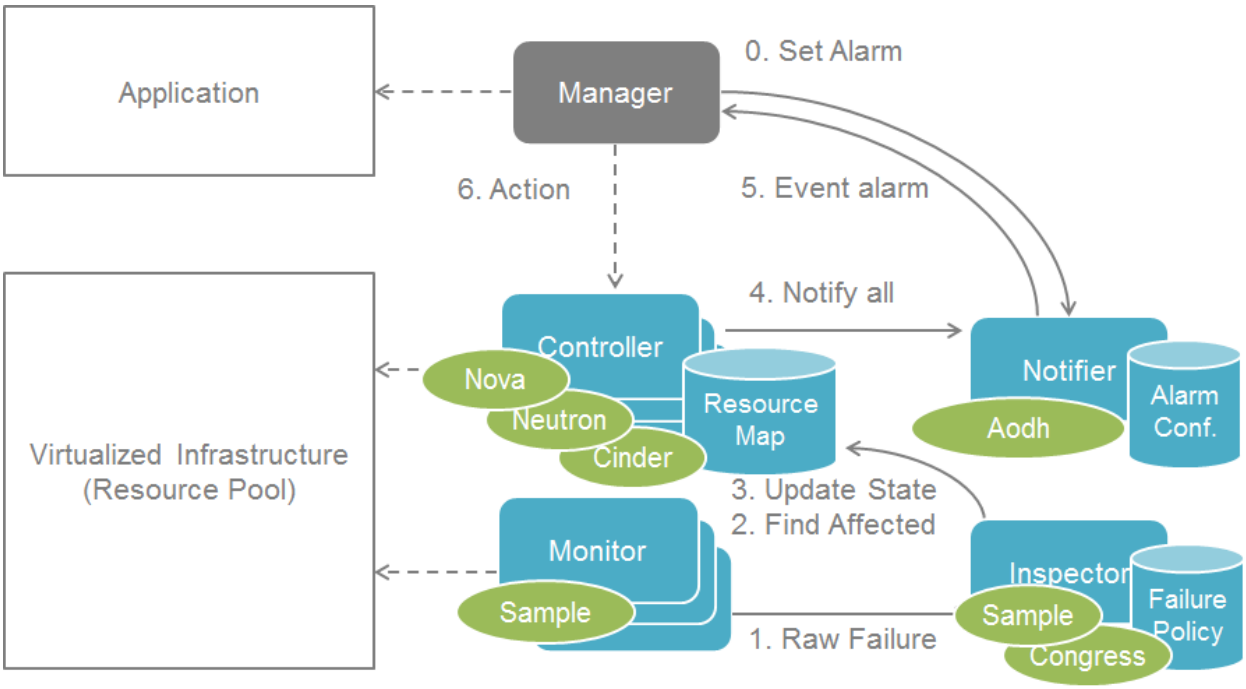


Fig. 1: Conservative Notification

It worth noting that the shortcut notification has a side effect that cloud resource states could still be out-of-sync by the time consumer processes the alarm notification. This is out of scope of inspector design but need to be taken consideration in system level.

Also the call of "reset servers state to error" is not necessary in the alternative notification case where the "host forced down" is still called. "get-valid-server-state" was implemented to have valid server state while earlier one couldn't get it unless calling "reset servers state to error". When not having "reset servers state to error", states are more unlikely to be out of sync while notification and force down host would be parallel.

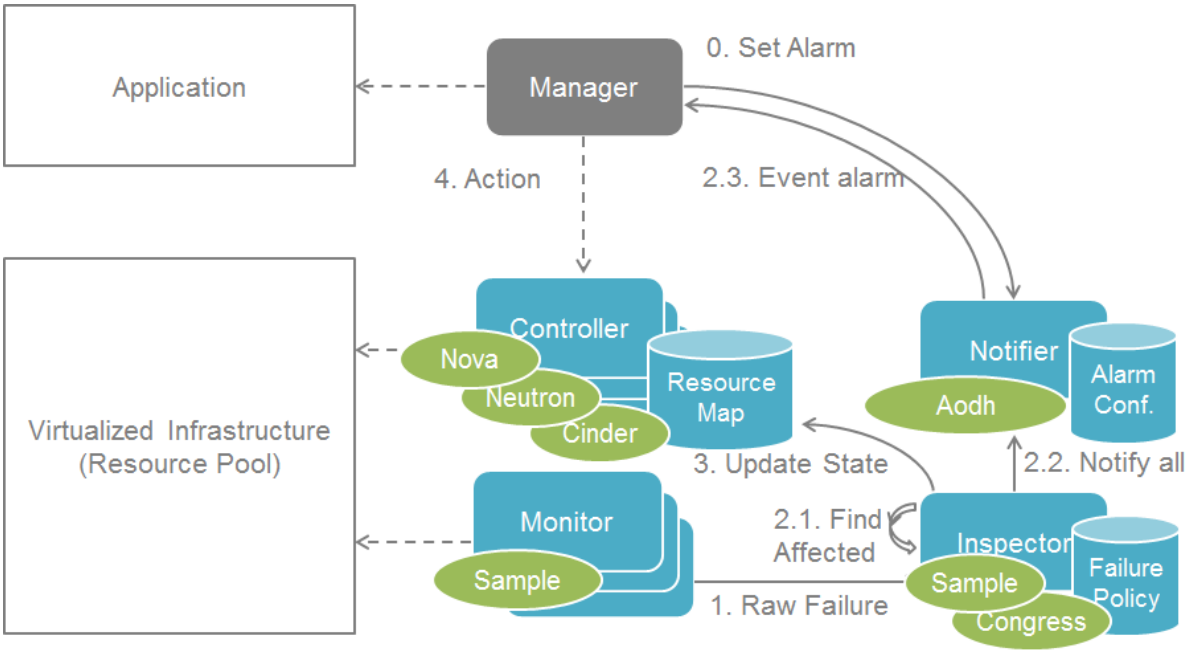


Fig. 2: Shortcut Notification

Appendix

A study has been made to evaluate the effect of parallel execution and shortcut notification on OPNFV Beijing Summit 2017.

Download the full presentation slides here.

2.1.6 Performance Profiler

https://goo.gl/98Osig

This blueprint proposes to create a performance profiler for doctor scenarios.

Problem Description

In the verification job for notification time, we have encountered some performance issues, such as

1. In environment deployed by APEX, it meets the criteria while in the one by Fuel, the performance is much more poor. 2. Signification performance degradation was spotted when we increase the total number of VMs

It takes time to dig the log and analyse the reason. People have to collect timestamp at each checkpoints manually to find out the bottleneck. A performance profiler will make this process automatic.

Proposed Change

Current Doctor scenario covers the inspector and notifier in the whole fault management cycle:

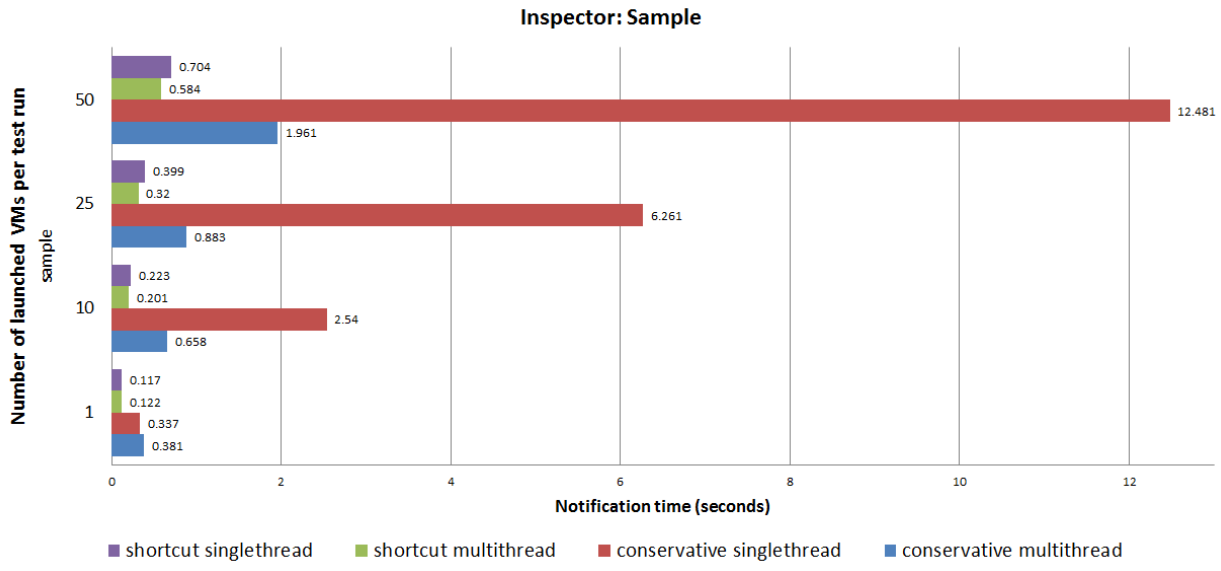


Fig. 3: Notification Time

```
start                                              end
  +       +         +        +       +           +
  |       |         |        |       |           |
  |monitor|inspector|notifier|manager|controller|
  +------>+         |        |       |           |
occurred  +-------->+        |       |           |
  |     detected    +------->+       |           |
  |       |     identified   +-------+           |
  |       |                notified   +--------->+
  |       |                   |    processed  resolved
  |       |                   |                  |
  |       +<-----doctor------>+                  |
  |                                              |
  |                                              |
  +<---------------fault management------------>+
```

The notification time can be split into several parts and visualized as a timeline:

```
start                                            end
  0----5---10---15---20---25---30---35---40---45--> (x 10ms)
  +    +   +   +   +    +      +   +   +   +   +
0-hostdown |   |   |    |      |   |   |   |   |
  +--->+   |   |   |    |      |   |   |   |   |
  |  1-raw failure |    |      |   |   |   |   |
  |    +-->+   |   |    |      |   |   |   |   |
  |    | 2-found affected      |   |   |   |   |
  |    |   +-->+   |    |      |   |   |   |   |
  |    |     3-marked host down|   |   |   |   |
  |    |       +-->+    |      |   |   |   |   |
  |    |         4-set VM error|   |   |   |   |
  |    |           +--->+      |   |   |   |   |
  |    |           |  5-notified VM error  |   |
  |    |           |    +----->|   |   |   |   |
```

(continues on next page)

(continued from previous page)

```
  |     |            |     |     6-transformed event
  |     |            |     |       +-->+    |    |    |
  |     |            |     |       | 7-evaluated event
  |     |            |     |       |    +-->+    |    |
  |     |            |     |       |      8-fired alarm
  |     |            |     |       |        +-->+    |
  |     |            |     |       |          9-received alarm
  |     |            |     |       |            +-->+
sample | sample     |     |       |             |10-handled alarm
monitor| inspector |nova| c/m   |     aodh     |
  |                                             |
  +<-----------------doctor---------------->+
```

Note: c/m = ceilometer

And a table of components sorted by time cost from most to least

Component	Time Cost	Percentage
inspector	160ms	40%
aodh	110ms	30%
monitor	50ms	14%
...		
...		

Note: data in the table is for demonstration only, not actual measurement

Timestamps can be collected from various sources

1. log files
2. trace point in code

The performance profiler will be integrated into the verification job to provide detail result of the test. It can also be deployed independently to diagnose performance issue in specified environment.

Working Items

1. PoC with limited checkpoints
2. Integration with verification job
3. Collect timestamp at all checkpoints
4. Display the profiling result in console
5. Report the profiling result to test database
6. Independent package which can be installed to specified environment

2.1.7 Planned Maintenance Design Guideline

This document describes how one can implement infrastructure maintenance in interaction with VNFM by utilizing the OPNFV Doctor project framework and to meet the set requirements. Document concentrates to OpenStack and VMs while the concept designed is generic for any payload or even different VIM. Admin tool should be also for controller and other cloud hardware, but that is not the main focus in OPNFV Doctor and should be defined better in the upstream implementation. Same goes for any more detailed work to be done.

Problem Description

Telco application need to know when infrastructure maintenance is going to happen in order to guarantee zero down time in its operation. It needs to be possible to make own actions to have application running on not affected resource or give guidance to admin actions like migration. More details are defined in requirement documentation: use cases, architecture and implementation.

Guidelines

Concepts used:

- *event*: Notification to rabbitmq with particular event type.
- *state event*: Notification to rabbitmq with particular event type including payload with variable defined for state.
- *project event*: Notification to rabbitmq that is meant for project. Single event type is used with different payload and state information.
- *admin event*: Notification to rabbitmq that is meant for admin or as for any infrastructure service. Single event type is used with different state information.
- *rolling maintenance*: Node by Node rolling maintenance and upgrade where a single node at a time will be maintained after a possible application payload is moved away from the node.
- *project* stands for *application* in OpenStack contents and both are used in this document. *tenant* is many times used for the same.

Infrastructure admin needs to make notification with two different event types. One is meant for admin and one for project. Notification payload can be consumed by application and admin by subscribing to corresponding event alarm trough alarming service like OpenStack AODH.

- Infrastructure admin needs to make a notification about infrastructure maintenance including all details that application needs in order to make a decisions upon his affected service. Alarm Payload can hold a link to infrastructure admin tool API for reply and for other possible information. There is many steps of communication between admin tool and application, thus the payload needed for the information passed is very similar. Because of this, the same event type can be used, but there can be a variable like *state* to tell application what is needed as action for each event. If a project have not subscribed to alarm, admin tool responsible for the maintenance will assume it can do maintenance operations without interaction with application on top of it.
- Infrastructure admin needs to make an event about infrastructure maintenance telling when the maintenance starts and another when it ends. This admin level event should include the host name. This could be consumed by any admin level infrastructure entity. In this document we consume this in *Inspector* that is in OPNFV Doctor project terms infrastructure entity responsible for automatic host fault management. Automated actions surely needs to be disabled during planned maintenance.

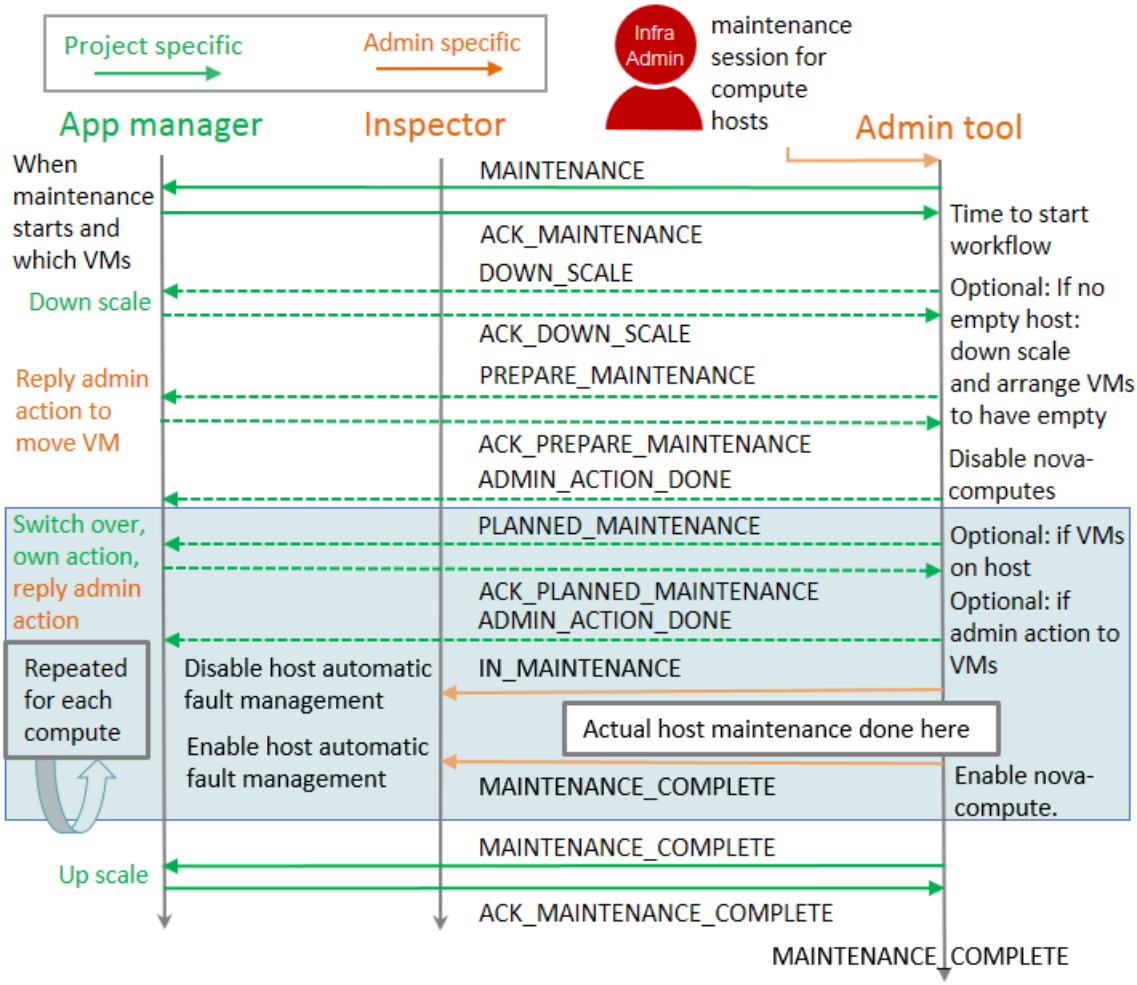
Before maintenance starts application needs to be able to make switch over for his ACT-STBY service affected, do operation to move service to not effected part of infrastructure or give a hint for admin operation like migration that can be automatically issued by admin tool according to agreed policy.

There should be at least one empty host compatible to host under maintenance in order to have a smooth *rolling maintenance* done. For this to be possible also down scaling the application instances should be possible.

Infrastructure admin should have a tool that is responsible for hosting a maintenance work flow session with needed APIs for admin and for applications. The Group of hosts in single maintenance session should always have the same physical capabilities, so the rolling maintenance can be guaranteed.

Flow diagram is meant to be as high level as possible. It currently does not try to be perfect, but to show the most important interfaces needed between VNFM and infrastructure admin. This can be seen e.g. as missing error handling that can be defined later on.

Flow diagram:



Flow diagram step by step:

- Infrastructure admin makes a maintenance session to maintain and upgrade certain group of hardware. At least compute hardware in single session should be having same capabilities like the amount number of VCPUs to ensure the maintenance can be done node by node in rolling fashion. Maintenance session need to have a *session_id* that is a unique ID to be carried throughout all events and can be used in APIs needed when interacting with the session. Maintenance session needs to have knowledge about when maintenance will start and what capabilities the possible upgrade to infrastructure will bring to application payload on top of it. It will be matter of the implementation to define in more detail whether some more data is needed when creating a session or if it is defined in the admin tool configuration.

  There can be several parallel maintenance sessions and a single session can include multiple projects payload. Typically maintenance session should include similar type of compute hardware, so you can guarantee moving of instances on top of them can work between the compute hosts.

- State *MAINTENANCE project event* and reply *ACK_MAINTENANCE*. Immediately after a maintenance session is created, infrastructure admin tool will send a project specific 'notification' which application manager can consume by subscribing to AODH alarm for this event. As explained already earlier all 'project event's will only be sent in case the project subscribes to alarm and otherwise the interaction with application will simply not be done and operations could be forced.

  The state *MAINTENANCE* event should at least include:

  - *session_id* to reference correct maintenance session.
  - *state* as *MAINTENANCE* to identify event action needed.
  - *instance_ids* to tell project which of his instances will be affected by the maintenance. This might be a link to admin tool project specific API as AODH variables are limited to string of 255 character.
  - *reply_url* for application to call admin tool project specific API to answer *ACK_MAINTENANCE* including the *session_id*.
  - *project_id* to identify project.
  - *actions_at* time stamp to indicate when maintenance work flow will start. *ACK_MAINTENANCE* reply is needed before that time.
  - *metadata* to include key values pairs of a capabilities coming over the maintenance operation like 'openstack_version': 'Queens'

- Optional state *DOWN_SCALE project event* and reply *ACK_DOWN_SCALE*. When it is time to start the maintenance work flow as the time reaches the *actions_at* defined in previous *state event*, admin tool needs to check if there is already an empty compute host needed by the *rolling maintenance*. In case there is no empty host, admin tool can ask application to down scale by sending project specific *DOWN_SCALE state event*.

  The state *DOWN_SCALE* event should at least include:

  - *session_id* to reference correct maintenance session.
  - *state* as *DOWN_SCALE* to identify event action needed.
  - *reply_url* for application to call admin tool project specific API to answer *ACK_DOWN_SCALE* including the *session_id*.
  - *project_id* to identify project.
  - *actions_at* time stamp to indicate when is the last moment to send *ACK_DOWN_SCALE*. This means application can have time to finish some ongoing transactions before down scaling his instances. This guarantees a zero downtime for his service.

- Optional state *PREPARE_MAINTENANCE project event* and reply *ACK_PREPARE_MAINTENANCE*. In case still after down scaling the applications there is still no empty compute host, admin tools needs to analyze the situation on compute host under maintenance. It needs to choose compute node that is now almost empty or has otherwise least critical instances running if possible, like looking if there is floating IPs. When compute host is chosen, a *PREPARE_MAINTENANCE state event* can be sent to projects having instances running on this host to migrate them to other compute hosts. It might also be possible to have another round of *DOWN_SCALE state event* if necessary, but this is not proposed here.

  The state *PREPARE_MAINTENANCE* event should at least include:

  - *session_id* to reference correct maintenance session.
  - *state* as *PREPARE_MAINTENANCE* to identify event action needed.
  - *instance_ids* to tell project which of his instances will be affected by the *state event*. This might be a link to admin tool project specific API as AODH variables are limited to string of 255 character.
  - *reply_url* for application to call admin tool project specific API to answer *ACK_PREPARE_MAINTENANCE* including the *session_id* and *instance_ids* with list of key value pairs with key as *instance_id* and chosen action from allowed actions given via *allowed_actions* as value.
  - *project_id* to identify project.
  - *actions_at* time stamp to indicate when is the last moment to send *ACK_PREPARE_MAINTENANCE*. This means application can have time to finish some ongoing transactions within his instances and make possible switch over. This guarantees a zero downtime for his service.

  - *allowed_actions* to tell what admin tool supports as action to move instances to another compute host. Typically a list like: *['MIGRATE', 'LIVE_MIGRATE']*

- Optional state *INSTANCE_ACTION_DONE project event*. In case admin tool needed to make action to move instance like migrating it to another compute host, this *state event* will be sent to tell the operation is complete.

  The state *INSTANCE_ACTION_DONE* event should at least include:

  - *session_id* to reference correct maintenance session.
  - *instance_ids* to tell project which of his instance had the admin action done.
  - *project_id* to identify project.

- At this state it is guaranteed there is an empty compute host. It would be maintained first trough *IN_MAINTENANCE* and *MAINTENANCE_COMPLETE* steps, but following the flow chart *PLANNED_MAINTENANCE* will be explained next.

- Optional state *PLANNED_MAINTENANCE project event* and reply *ACK_PLANNED_MAINTENANCE*. In case compute host to be maintained has instances, projects owning those should have this *state event*. When project receives this *state event* it knows instances moved to other compute host as resulting actions will now go to host that is already maintained. This means it might have new capabilities that project can take into use. This gives the project the possibility to upgrade his instances also to support new capabilities over the action chosen to move instances.

  The state *PLANNED_MAINTENANCE* event should at least include:

  - *session_id* to reference correct maintenance session.
  - *state* as *PLANNED_MAINTENANCE* to identify event action needed.
  - *instance_ids* to tell project which of his instances will be affected by the event. This might be a link to admin tool project specific API as AODH variables are limited to string of 255 character.
  - *reply_url* for application to call admin tool project specific API to answer *ACK_PLANNED_MAINTENANCE* including the *session_id* and *instance_ids* with list of key value pairs with key as *instance_id* and chosen action from allowed actions given via *allowed_actions* as value.
  - *project_id* to identify project.
  - *actions_at* time stamp to indicate when is the last moment to send *ACK_PLANNED_MAINTENANCE*. This means application can have time to finish some ongoing transactions within his instances and make possible switch over. This guarantees a zero downtime for his service.
  - *allowed_actions* to tell what admin tool supports as action to move instances to another compute host. Typically a list like: *['MIGRATE', 'LIVE_MIGRATE', 'OWN_ACTION'] OWN_ACTION* means that application may want to re-instantiate his instance perhaps to take into use the new capability coming over the infrastructure maintenance. Re-instantiated instance will go to already maintained host having the new capability.
  - *metadata* to include key values pairs of a capabilities coming over the maintenance operation like 'openstack_version': 'Queens'

- *State IN_MAINTENANCE* and *MAINTENANCE_COMPLETE admin event's. Just before host goes to maintenance the IN_MAINTENANCE state event* will be send to indicate host is entering to maintenance. Host is then taken out of production and can be powered off, replaced, or rebooted during the operation. During the maintenance and upgrade host might be moved to admin's own host aggregate, so it can be tested to work before putting back to production. After maintenance is complete *MAINTENANCE_COMPLETE state event* will be sent to know host is back in use. Adding or removing of a host is yet not included in this concept, but can be addressed later.

  The state *IN_MAINTENANCE* and *MAINTENANCE_COMPLETE* event should at least include:

  - *session_id* to reference correct maintenance session.
  - *state* as *IN_MAINTENANCE* or *MAINTENANCE_COMPLETE* to indicate host state.
  - *project_id* to identify admin project needed by AODH alarm.
  - *host* to indicate the host name.
- State *MAINTENANCE_COMPLETE project event* and reply *MAINTENANCE_COMPLETE_ACK*. After all compute nodes in the maintenance session have gone trough maintenance operation this *state event* can be send to all projects that had instances running on any of those nodes. If there was a down scale done, now the application could up scale back to full operation.
  - *session_id* to reference correct maintenance session.
  - *state* as *MAINTENANCE_COMPLETE* to identify event action needed.
  - *instance_ids* to tell project which of his instances are currently running on hosts maintained in this maintenance session. This might be a link to admin tool project specific API as AODH variables are limited to string of 255 character.
  - *reply_url* for application to call admin tool project specific API to answer *ACK_MAINTENANCE* including the *session_id*.
  - *project_id* to identify project.
  - *actions_at* time stamp to indicate when maintenance work flow will start.
  - *metadata* to include key values pairs of a capabilities coming over the maintenance operation like 'openstack_version': 'Queens'
- At the end admin tool maintenance session can enter to *MAINTENANCE_COMPLETE* state and session can be removed.

Benefits

- Application is guaranteed zero downtime as it is aware of the maintenance action affecting its payload. The application is made aware of the maintenance time window to make sure it can prepare for it.
- Application gets to know new capabilities over infrastructure maintenance and upgrade and can utilize those (like do its own upgrade)
- Any application supporting the interaction being defined could be running on top of the same infrastructure provider. No vendor lock-in for application.
- Any infrastructure component can be aware of host(s) under maintenance via 'admin event's about host state. No vendor lock-in for infrastructure components.
- Generic messaging making it possible to use same concept in different type of clouds and application payloads. *instance_ids* will uniquely identify any type of instance and similar notification payload can be used regardless we are in OpenStack. Work flow just need to support different cloud infrastructure management to support different cloud.
- No additional hardware is needed during maintenance operations as down- and up-scaling can be supported for the applications. Optional, if no extensive spare capacity is available for the maintenance - as typically the case in Telco environments.
- Parallel maintenance sessions for different group of hardware. Same session should include hardware with same capabilities to guarantee *rolling maintenance* actions.
- Multi-tenancy support. Project specific messaging about maintenance.

Future considerations

- Pluggable architecture for infrastructure admin tool to handle different clouds and payloads.
- Pluggable architecture to handle specific maintenance/upgrade cases like OpenStack upgrade between specific versions or admin testing before giving host back to production.
- Support for user specific details need to be taken into account in admin side actions (e.g. run a script, . . .).
- (Re-)Use existing implementations like Mistral for work flows.
- Scaling hardware resources. Allow critical application to be scaled at the same time in controlled fashion or retire application.

POC

There was a Maintenance POC demo 'How to gain VNF zero down-time during Infrastructure Maintenance and Upgrade' in the OCP and ONS summit March 2018. Similar concept is also being made as OPNFV Doctor project new test case scenario.

2.2 Doctor: Fault Management and Maintenance

Project Doctor, https://wiki.opnfv.org/doctor

Editors Ashiq Khan (NTT DOCOMO), Gerald Kunzmann (NTT DOCOMO)

Authors Ryota Mibu (NEC), Carlos Goncalves (NEC), Tomi Juvonen (Nokia), Tommy Lindgren (Ericsson), Bertrand Souville (NTT DOCOMO), Balazs Gibizer (Ericsson), Ildiko Vancsa (Ericsson) and others.

Abstract Doctor is an OPNFV requirement project *[DOCT]*. Its scope is NFVI fault management, and maintenance and it aims at developing and realizing the consequent implementation for the OPNFV reference platform.

This deliverable is introducing the use cases and operational scenarios for Fault Management considered in the Doctor project. From the general features, a high level architecture describing logical building blocks and interfaces is derived. Finally, a detailed implementation is introduced, based on available open source components, and a related gap analysis is done as part of this project. The implementation plan finally discusses an initial realization for a NFVI fault management and maintenance solution in open source software.

Definition of terms

Different SDOs and communities use different terminology related to NFV/Cloud/SDN. This list tries to define an OPNFV terminology, mapping/translating the OPNFV terms to terminology used in other contexts.

ACT-STBY configuration Failover configuration common in Telco deployments. It enables the operator to use a standby (STBY) instance to take over the functionality of a failed active (ACT) instance.

Administrator Administrator of the system, e.g. OAM in Telco context.

Consumer User-side Manager; consumer of the interfaces produced by the VIM; VNFM, NFVO, or Orchestrator in ETSI NFV *[ENFV]* terminology.

EPC Evolved Packet Core, the main component of the core network architecture of 3GPP's LTE communication standard.

MME Mobility Management Entity, an entity in the EPC dedicated to mobility management.

NFV Network Function Virtualization

NFVI Network Function Virtualization Infrastructure; totality of all hardware and software components which build up the environment in which VNFs are deployed.

S/P-GW Serving/PDN-Gateway, two entities in the EPC dedicated to routing user data packets and providing connectivity from the UE to external packet data networks (PDN), respectively.

Physical resource Actual resources in NFVI; not visible to Consumer.

VNFM Virtualized Network Function Manager; functional block that is responsible for the lifecycle management of VNF.

NFVO Network Functions Virtualization Orchestrator; functional block that manages the Network Service (NS) lifecycle and coordinates the management of NS lifecycle, VNF lifecycle (supported by the VNFM) and NFVI resources (supported by the VIM) to ensure an optimized allocation of the necessary resources and connectivity.

VIM Virtualized Infrastructure Manager; functional block that is responsible for controlling and managing the NFVI compute, storage and network resources, usually within one operator's Infrastructure Domain, e.g. NFVI Point of Presence (NFVI-PoP).

Virtual Machine (VM) Virtualized computation environment that behaves very much like a physical computer/server.

Virtual network Virtual network routes information among the network interfaces of VM instances and physical network interfaces, providing the necessary connectivity.

Virtual resource A Virtual Machine (VM), a virtual network, or virtualized storage; Offered resources to "Consumer" as result of infrastructure virtualization; visible to Consumer.

Virtual Storage Virtualized non-volatile storage allocated to a VM.

VNF Virtualized Network Function. Implementation of a Network Function that can be deployed on a Network Function Virtualization Infrastructure (NFVI).

2.2.1 Introduction

The goal of this project is to build an NFVI fault management and maintenance framework supporting high availability of the Network Services on top of the virtualized infrastructure. The key feature is immediate notification of unavailability of virtualized resources from VIM, to support failure recovery, or failure avoidance of VNFs running on them. Requirement survey and development of missing features in NFVI and VIM are in scope of this project in order to fulfil requirements for fault management and maintenance in NFV.

The purpose of this requirement project is to clarify the necessary features of NFVI fault management, and maintenance, identify missing features in the current OpenSource implementations, provide a potential implementation architecture and plan, provide implementation guidelines in relevant upstream projects to realize those missing features, and define the VIM northbound interfaces necessary to perform the task of NFVI fault management, and maintenance in alignment with ETSI NFV *[ENFV]*.

Problem description

A Virtualized Infrastructure Manager (VIM), e.g. OpenStack *[OPSK]*, cannot detect certain Network Functions Virtualization Infrastructure (NFVI) faults. This feature is necessary to detect the faults and notify the Consumer in order to ensure the proper functioning of EPC VNFs like MME and S/P-GW.

- EPC VNFs are often in active standby (ACT-STBY) configuration and need to switch from STBY mode to ACT mode as soon as relevant faults are detected in the active (ACT) VNF.
- NFVI encompasses all elements building up the environment in which VNFs are deployed, e.g., Physical Machines, Hypervisors, Storage, and Network elements.

In addition, VIM, e.g. OpenStack, needs to receive maintenance instructions from the Consumer, i.e. the operator/administrator of the VNF.

- Change the state of certain Physical Machines (PMs), e.g. empty the PM, so that maintenance work can be performed at these machines.

Note: Although fault management and maintenance are different operations in NFV, both are considered as part of this project as – except for the trigger – they share a very similar work and message flow. Hence, from implementation perspective, these two are kept together in the Doctor project because of this high degree of similarity.

2.2.2 Use cases and scenarios

Telecom services often have very high requirements on service performance. As a consequence they often utilize redundancy and high availability (HA) mechanisms for both the service and the platform. The HA support may be built-in or provided by the platform. In any case, the HA support typically has a very fast detection and reaction time to minimize service impact. The main changes proposed in this document are about making a clear distinction between fault management and recovery a) within the VIM/NFVI and b) High Availability support for VNFs on the other, claiming that HA support within a VNF or as a service from the platform is outside the scope of Doctor and is discussed in the High Availability for OPNFV project. Doctor should focus on detecting and remediating faults in the NFVI. This will ensure that applications come back to a fully redundant configuration faster than before.

As an example, Telecom services can come with an Active-Standby (ACT-STBY) configuration which is a (1+1) redundancy scheme. ACT and STBY nodes (aka Physical Network Function (PNF) in ETSI NFV terminology) are in a hot standby configuration. If an ACT node is unable to function properly due to fault or any other reason, the STBY node is instantly made ACT, and affected services can be provided without any service interruption.

The ACT-STBY configuration needs to be maintained. This means, when a STBY node is made ACT, either the previously ACT node, after recovery, shall be made STBY, or, a new STBY node needs to be configured. The actual operations to instantiate/configure a new STBY are similar to instantiating a new VNF and therefore are outside the scope of this project.

The NFVI fault management and maintenance requirements aim at providing fast failure detection of physical and virtualized resources and remediation of the virtualized resources provided to Consumers according to their predefined request to enable applications to recover to a fully redundant mode of operation.

1. Fault management/recovery using ACT-STBY configuration (Triggered by critical error)
2. Preventive actions based on fault prediction (Preventing service stop by handling warnings)
3. VM Retirement (Managing service during NFVI maintenance, i.e. H/W, Hypervisor, Host OS, maintenance)

Faults

Fault management using ACT-STBY configuration

In figure1, a system-wide view of relevant functional blocks is presented. OpenStack is considered as the VIM implementation (aka Controller) which has interfaces with the NFVI and the Consumers. The VNF implementation is represented as different virtual resources marked by different colors. Consumers (VNFM or NFVO in ETSI NFV terminology) own/manage the respective virtual resources (VMs in this example) shown with the same colors.

The first requirement in this use case is that the Controller needs to detect faults in the NFVI ("1. Fault Notification" in figure1) affecting the proper functioning of the virtual resources (labelled as VM-x) running on top of it. It should be possible to configure which relevant fault items should be detected. The VIM (e.g. OpenStack) itself could be extended to detect such faults. Alternatively, a third party fault monitoring tool could be used which then informs the VIM about such faults; this third party fault monitoring element can be considered as a component of VIM from an architectural point of view.

Once such fault is detected, the VIM shall find out which virtual resources are affected by this fault. In the example in figure1, VM-4 is affected by a fault in the Hardware Server-3. Such mapping shall be maintained in the VIM, depicted as the "Server-VM info" table inside the VIM.

Once the VIM has identified which virtual resources are affected by the fault, it needs to find out who is the Consumer (i.e. the owner/manager) of the affected virtual resources (Step 2). In the example shown in figure1, the VIM knows that for the red VM-4, the manager is the red Consumer through an Ownership info table. The VIM then notifies (Step 3 "Fault Notification") the red Consumer about this fault, preferably with sufficient abstraction rather than detailed physical fault information.

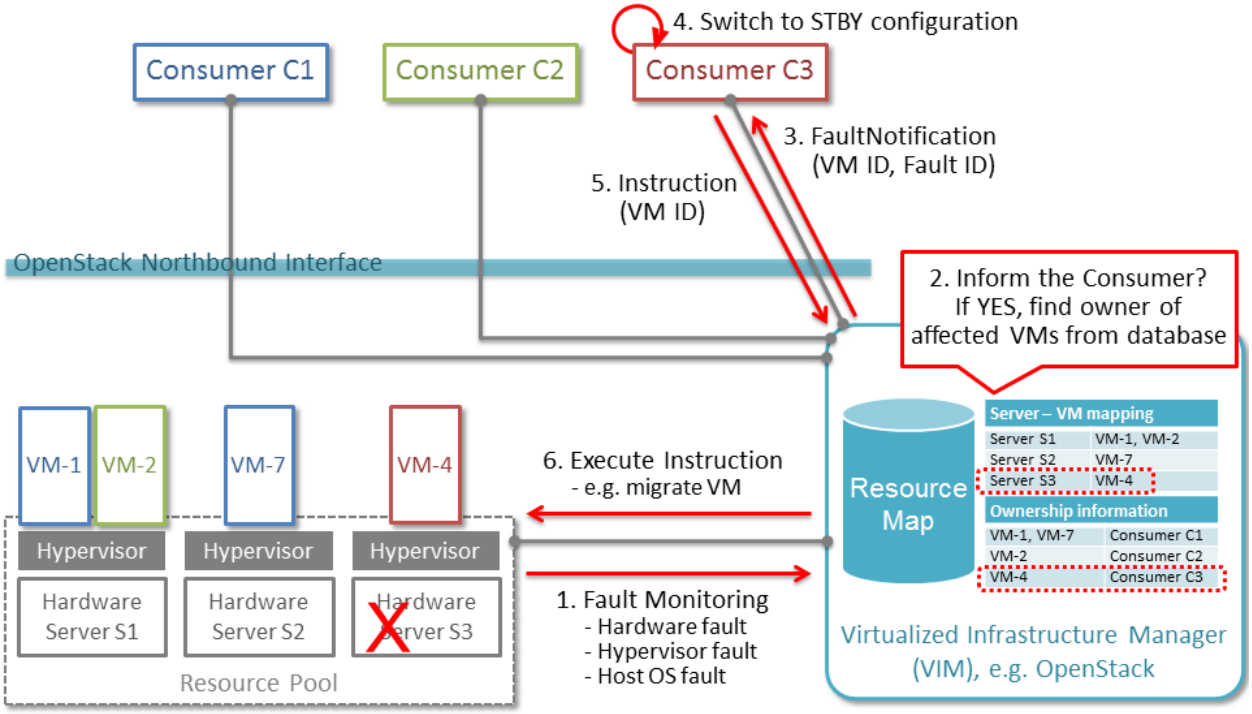


Fig. 4: Fault management/recovery use case

The Consumer then switches to STBY configuration by switching the STBY node to ACT state (Step 4). It further initiates a process to instantiate/configure a new STBY. However, switching to STBY mode and creating a new STBY machine is a VNFM/NFVO level operation and therefore outside the scope of this project. Doctor project does not create interfaces for such VNFM level configuration operations. Yet, since the total failover time of a consumer service

depends on both the delay of such processes as well as the reaction time of Doctor components, minimizing Doctor's reaction time is a necessary basic ingredient to fast failover times in general.

Once the Consumer has switched to STBY configuration, it notifies (Step 5 "Instruction" in `figure1`) the VIM. The VIM can then take necessary (e.g. pre-determined by the involved network operator) actions on how to clean up the fault affected VMs (Step 6 "Execute Instruction").

The key issue in this use case is that a VIM (OpenStack in this context) shall not take a standalone fault recovery action (e.g. migration of the affected VMs) before the ACT-STBY switching is complete, as that might violate the ACT-STBY configuration and render the node out of service.

As an extension of the 1+1 ACT-STBY resilience pattern, a STBY instance can act as backup to N ACT nodes (N+1). In this case, the basic information flow remains the same, i.e., the consumer is informed of a failure in order to activate the STBY node. However, in this case it might be useful for the failure notification to cover a number of failed instances due to the same fault (e.g., more than one instance might be affected by a switch failure). The reaction of the consumer might depend on whether only one active instance has failed (similar to the ACT-STBY case), or if more active instances are needed as well.

Preventive actions based on fault prediction

The fault management scenario explained in *Fault management using ACT-STBY configuration* can also be performed based on fault prediction. In such cases, in VIM, there is an intelligent fault prediction module which, based on its NFVI monitoring information, can predict an imminent fault in the elements of NFVI. A simple example is raising temperature of a Hardware Server which might trigger a pre-emptive recovery action. The requirements of such fault prediction in the VIM are investigated in the OPNFV project "Data Collection for Failure Prediction" *[PRED]*.

This use case is very similar to *Fault management using ACT-STBY configuration*. Instead of a fault detection (Step 1 "Fault Notification in" `figure1`), the trigger comes from a fault prediction module in the VIM, or from a third party module which notifies the VIM about an imminent fault. From Step 2~5, the work flow is the same as in the "Fault management using ACT-STBY configuration" use case, except in this case, the Consumer of a VM/VNF switches to STBY configuration based on a predicted fault, rather than an occurred fault.

NFVI Maintenance

VM Retirement

All network operators perform maintenance of their network infrastructure, both regularly and irregularly. Besides the hardware, virtualization is expected to increase the number of elements subject to such maintenance as NFVI holds new elements like the hypervisor and host OS. Maintenance of a particular resource element e.g. hardware, hypervisor etc. may render a particular server hardware unusable until the maintenance procedure is complete.

However, the Consumer of VMs needs to know that such resources will be unavailable because of NFVI maintenance. The following use case is again to ensure that the ACT-STBY configuration is not violated. A stand-alone action (e.g. live migration) from VIM/OpenStack to empty a physical machine so that consequent maintenance procedure could be performed may not only violate the ACT-STBY configuration, but also have impact on real-time processing scenarios where dedicated resources to virtual resources (e.g. VMs) are necessary and a pause in operation (e.g. vCPU) is not allowed. The Consumer is in a position to safely perform the switch between ACT and STBY nodes, or switch to an alternative VNF forwarding graph so the hardware servers hosting the ACT nodes can be emptied for the upcoming maintenance operation. Once the target hardware servers are emptied (i.e. no virtual resources are running on top), the VIM can mark them with an appropriate flag (i.e. "maintenance" state) such that these servers are not considered for hosting of virtual machines until the maintenance flag is cleared (i.e. nodes are back in "normal" status).

A high-level view of the maintenance procedure is presented in `figure2`. VIM/OpenStack, through its northbound interface, receives a maintenance notification (Step 1 "Maintenance Request") from the Administrator (e.g. a network

operator) including information about which hardware is subject to maintenance. Maintenance operations include replacement/upgrade of hardware, update/upgrade of the hypervisor/host OS, etc.

The consequent steps to enable the Consumer to perform ACT-STBY switching are very similar to the fault management scenario. From VIM/OpenStack's internal database, it finds out which virtual resources (VM-x) are running on those particular Hardware Servers and who are the managers of those virtual resources (Step 2). The VIM then informs the respective Consumer (VNFMs or NFVO) in Step 3 "Maintenance Notification". Based on this, the Consumer takes necessary actions (Step 4, e.g. switch to STBY configuration or switch VNF forwarding graphs) and then notifies (Step 5 "Instruction") the VIM. Upon receiving such notification, the VIM takes necessary actions (Step 6 "Execute Instruction" to empty the Hardware Servers so that consequent maintenance operations could be performed. Due to the similarity for Steps 2~6, the maintenance procedure and the fault management procedure are investigated in the same project.

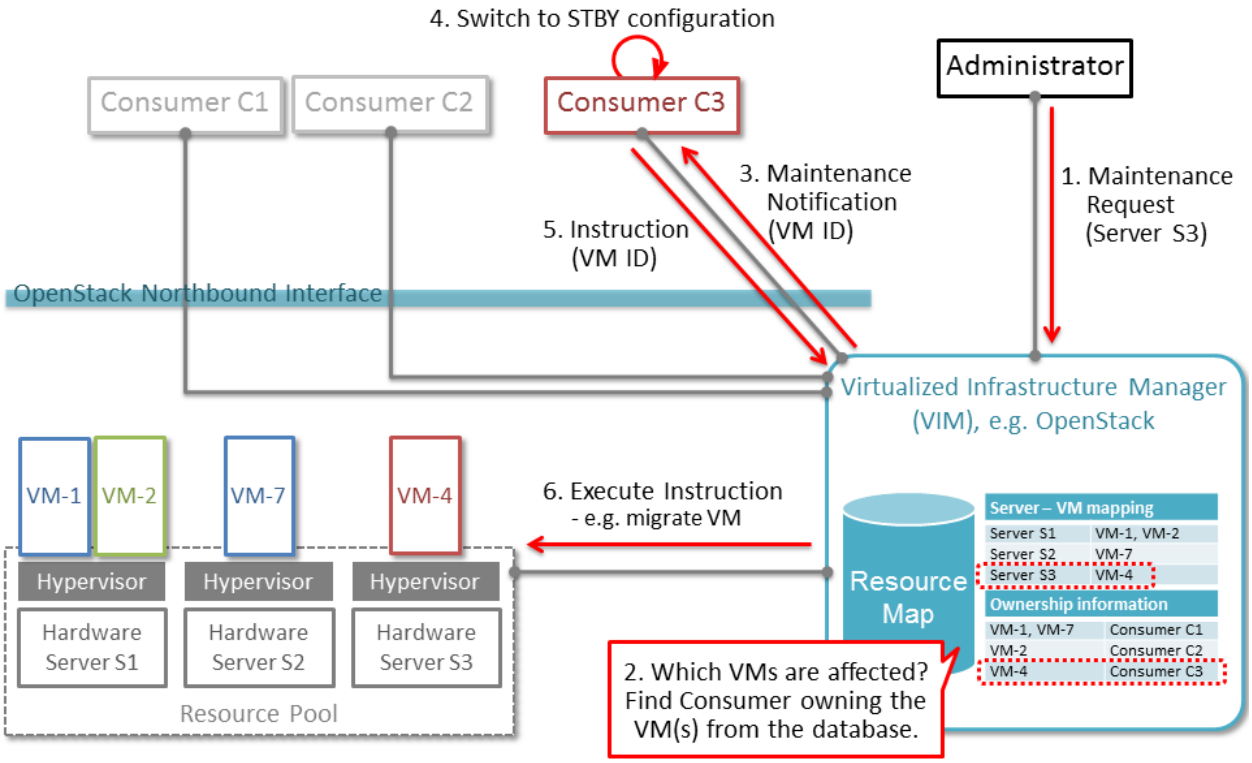


Fig. 5: Maintenance use case

2.2.3 High level architecture and general features

Functional overview

The Doctor project circles around two distinct use cases: 1) management of failures of virtualized resources and 2) planned maintenance, e.g. migration, of virtualized resources. Both of them may affect a VNF/application and the network service it provides, but there is a difference in frequency and how they can be handled.

Failures are spontaneous events that may or may not have an impact on the virtual resources. The Consumer should as soon as possible react to the failure, e.g., by switching to the STBY node. The Consumer will then instruct the VIM on how to clean up or repair the lost virtual resources, i.e. restore the VM, VLAN or virtualized storage. How much the applications are affected varies. Applications with built-in HA support might experience a short decrease in retainability (e.g. an ongoing session might be lost) while keeping availability (establishment or re-establishment of sessions are not affected), whereas the impact on applications without built-in HA may be more serious. How much

the network service is impacted depends on how the service is implemented. With sufficient network redundancy the service may be unaffected even when a specific resource fails.

On the other hand, planned maintenance impacting virtualized resources are events that are known in advance. This group includes e.g. migration due to software upgrades of OS and hypervisor on a compute host. Some of these might have been requested by the application or its management solution, but there is also a need for coordination on the actual operations on the virtual resources. There may be an impact on the applications and the service, but since they are not spontaneous events there is room for planning and coordination between the application management organization and the infrastructure management organization, including performing whatever actions that would be required to minimize the problems.

Failure prediction is the process of pro-actively identifying situations that may lead to a failure in the future unless acted on by means of maintenance activities. From applications' point of view, failure prediction may impact them in two ways: either the warning time is so short that the application or its management solution does not have time to react, in which case it is equal to the failure scenario, or there is sufficient time to avoid the consequences by means of maintenance activities, in which case it is similar to planned maintenance.

Architecture Overview

NFV and the Cloud platform provide virtual resources and related control functionality to users and administrators. `figure3` shows the high level architecture of NFV focusing on the NFVI, i.e., the virtualized infrastructure. The NFVI provides virtual resources, such as virtual machines (VM) and virtual networks. Those virtual resources are used to run applications, i.e. VNFs, which could be components of a network service which is managed by the consumer of the NFVI. The VIM provides functionalities of controlling and viewing virtual resources on hardware (physical) resources to the consumers, i.e., users and administrators. OpenStack is a prominent candidate for this VIM. The administrator may also directly control the NFVI without using the VIM.

Although OpenStack is the target upstream project where the new functional elements (Controller, Notifier, Monitor, and Inspector) are expected to be implemented, a particular implementation method is not assumed. Some of these elements may sit outside of OpenStack and offer a northbound interface to OpenStack.

General Features and Requirements

The following features are required for the VIM to achieve high availability of applications (e.g., MME, S/P-GW) and the Network Services:

1. Monitoring: Monitor physical and virtual resources.
2. Detection: Detect unavailability of physical resources.
3. Correlation and Cognition: Correlate faults and identify affected virtual resources.
4. Notification: Notify unavailable virtual resources to their Consumer(s).
5. Fencing: Shut down or isolate a faulty resource.
6. Recovery action: Execute actions to process fault recovery and maintenance.

The time interval between the instant that an event is detected by the monitoring system and the Consumer notification of unavailable resources shall be < 1 second (e.g., Step 1 to Step 4 in `figure4`).

Monitoring

The VIM shall monitor physical and virtual resources for unavailability and suspicious behavior.

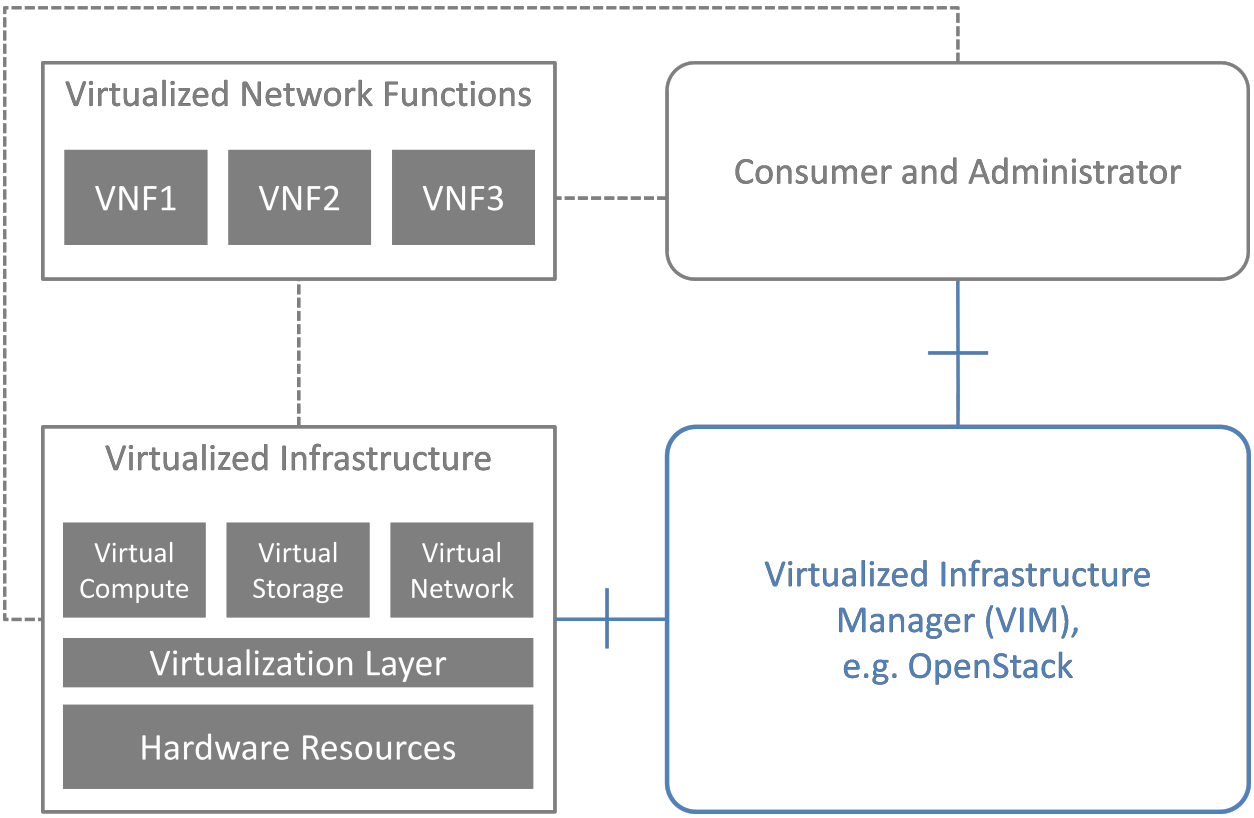


Fig. 6: High level architecture

Detection

The VIM shall detect unavailability and failures of physical resources that might cause errors/faults in virtual resources running on top of them. Unavailability of physical resource is detected by various monitoring and managing tools for hardware and software components. This may include also predicting upcoming faults. Note, fault prediction is out of scope of this project and is investigated in the OPNFV "Data Collection for Failure Prediction" project *[PRED]*.

The fault items/events to be detected shall be configurable.

The configuration shall enable Failure Selection and Aggregation. Failure aggregation means the VIM determines unavailability of physical resource from more than two non-critical failures related to the same resource.

There are two types of unavailability - immediate and future:

- Immediate unavailability can be detected by setting traps of raw failures on hardware monitoring tools.
- Future unavailability can be found by receiving maintenance instructions issued by the administrator of the NFVI or by failure prediction mechanisms.

Correlation and Cognition

The VIM shall correlate each fault to the impacted virtual resource, i.e., the VIM shall identify unavailability of virtualized resources that are or will be affected by failures on the physical resources under them. Unavailability of a virtualized resource is determined by referring to the mapping of physical and virtualized resources.

VIM shall allow configuration of fault correlation between physical and virtual resources. VIM shall support correlating faults:

- between a physical resource and another physical resource
- between a physical resource and a virtual resource
- between a virtual resource and another virtual resource

Failure aggregation is also required in this feature, e.g., a user may request to be only notified if failures on more than two standby VMs in an (N+M) deployment model occurred.

Notification

The VIM shall notify the alarm, i.e., unavailability of virtual resource(s), to the Consumer owning it over the northbound interface, such that the Consumers impacted by the failure can take appropriate actions to recover from the failure.

The VIM shall also notify the unavailability of physical resources to its Administrator.

All notifications shall be transferred immediately in order to minimize the stalling time of the network service and to avoid over assignment caused by delay of capability updates.

There may be multiple consumers, so the VIM has to find out the owner of a faulty resource. Moreover, there may be a large number of virtual and physical resources in a real deployment, so polling the state of all resources to the VIM would lead to heavy signaling traffic. Thus, a publication/subscription messaging model is better suited for these notifications, as notifications are only sent to subscribed consumers.

Notifications will be send out along with the configuration by the consumer. The configuration includes endpoint(s) in which the consumers can specify multiple targets for the notification subscription, so that various and multiple receiver functions can consume the notification message. Also, the conditions for notifications shall be configurable, such that the consumer can set according policies, e.g. whether it wants to receive fault notifications or not.

Note: the VIM should only accept notification subscriptions for each resource by its owner or administrator. Notifications to the Consumer about the unavailability of virtualized resources will include a description of the fault, preferably with sufficient abstraction rather than detailed physical fault information.

Fencing

Recovery actions, e.g. safe VM evacuation, have to be preceded by fencing the failed host. Fencing hereby means to isolate or shut down a faulty resource. Without fencing – when the perceived disconnection is due to some transient or partial failure – the evacuation might lead into two identical instances running together and having a dangerous conflict.

There is a cross-project definition in OpenStack of how to implement fencing, but there has not been any progress. The general description is available here: https://wiki.openstack.org/wiki/Fencing_Instances_of_an_Unreachable_Host

OpenStack provides some mechanisms that allow fencing of faulty resources. Some are automatically invoked by the platform itself (e.g. Nova disables the compute service when libvirtd stops running, preventing new VMs to be scheduled to that node), while other mechanisms are consumer trigger-based actions (e.g. Neutron port admin-state-up). For other fencing actions not supported by OpenStack, the Doctor project may suggest ways to address the gap (e.g. through means of resourcing to external tools and orchestration methods), or documenting or implementing them upstream.

The Doctor Inspector component will be responsible of marking resources down in the OpenStack and back up if necessary.

Recovery Action

In the basic *Fault management using ACT-STBY configuration* use case, no automatic actions will be taken by the VIM, but all recovery actions executed by the VIM and the NFVI will be instructed and coordinated by the Consumer.

In a more advanced use case, the VIM may be able to recover the failed virtual resources according to a pre-defined behavior for that resource. In principle this means that the owner of the resource (i.e., its consumer or administrator) can define which recovery actions shall be taken by the VIM. Examples are a restart of the VM or migration/evacuation of the VM.

High level northbound interface specification

Fault Management

This interface allows the Consumer to subscribe to fault notification from the VIM. Using a filter, the Consumer can narrow down which faults should be notified. A fault notification may trigger the Consumer to switch from ACT to STBY configuration and initiate fault recovery actions. A fault query request/response message exchange allows the Consumer to find out about active alarms at the VIM. A filter can be used to narrow down the alarms returned in the response message.

The high level message flow for the fault management use case is shown in `figure4`. It consists of the following steps:

1. The VIM monitors the physical and virtual resources and the fault management workflow is triggered by a monitored fault event.
2. Event correlation, fault detection and aggregation in VIM. Note: this may also happen after Step 3.
3. Database lookup to find the virtual resources affected by the detected fault.
4. Fault notification to Consumer.

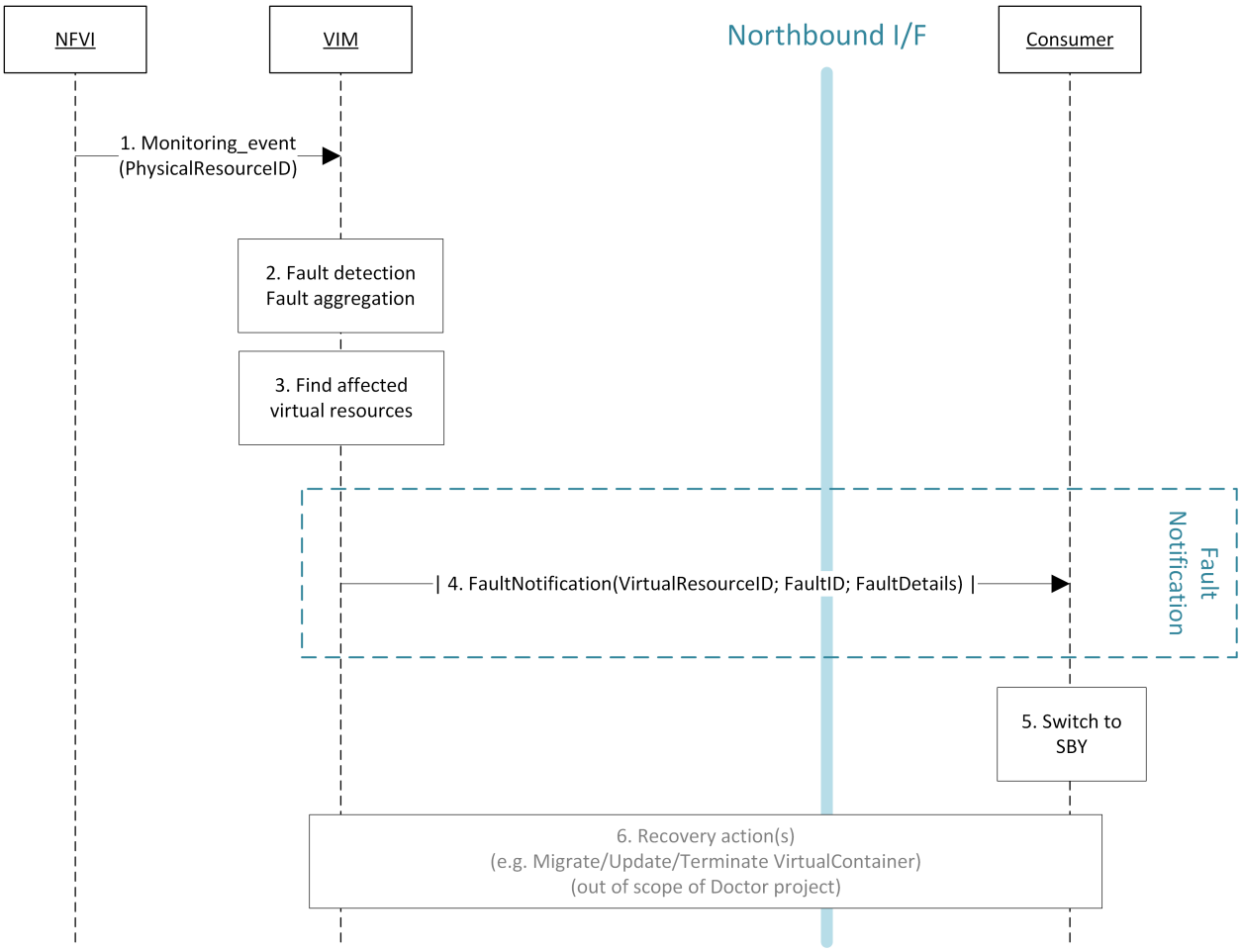


Fig. 7: High-level message flow for fault management

5. The Consumer switches to standby configuration (STBY).
6. Instructions to VIM requesting certain actions to be performed on the affected resources, for example migrate/update/terminate specific resource(s). After reception of such instructions, the VIM is executing the requested action, e.g., it will migrate or terminate a virtual resource.

NFVI Maintenance

The NFVI maintenance interface allows the Administrator to notify the VIM about a planned maintenance operation on the NFVI. A maintenance operation may for example be an update of the server firmware or the hypervisor. The MaintenanceRequest message contains instructions to change the state of the physical resource from 'enabled' to 'going-to-maintenance' and a timeout[1]. After receiving the MaintenanceRequest,the VIM decides on the actions to be taken based on maintenance policies predefined by the affected Consumer(s).

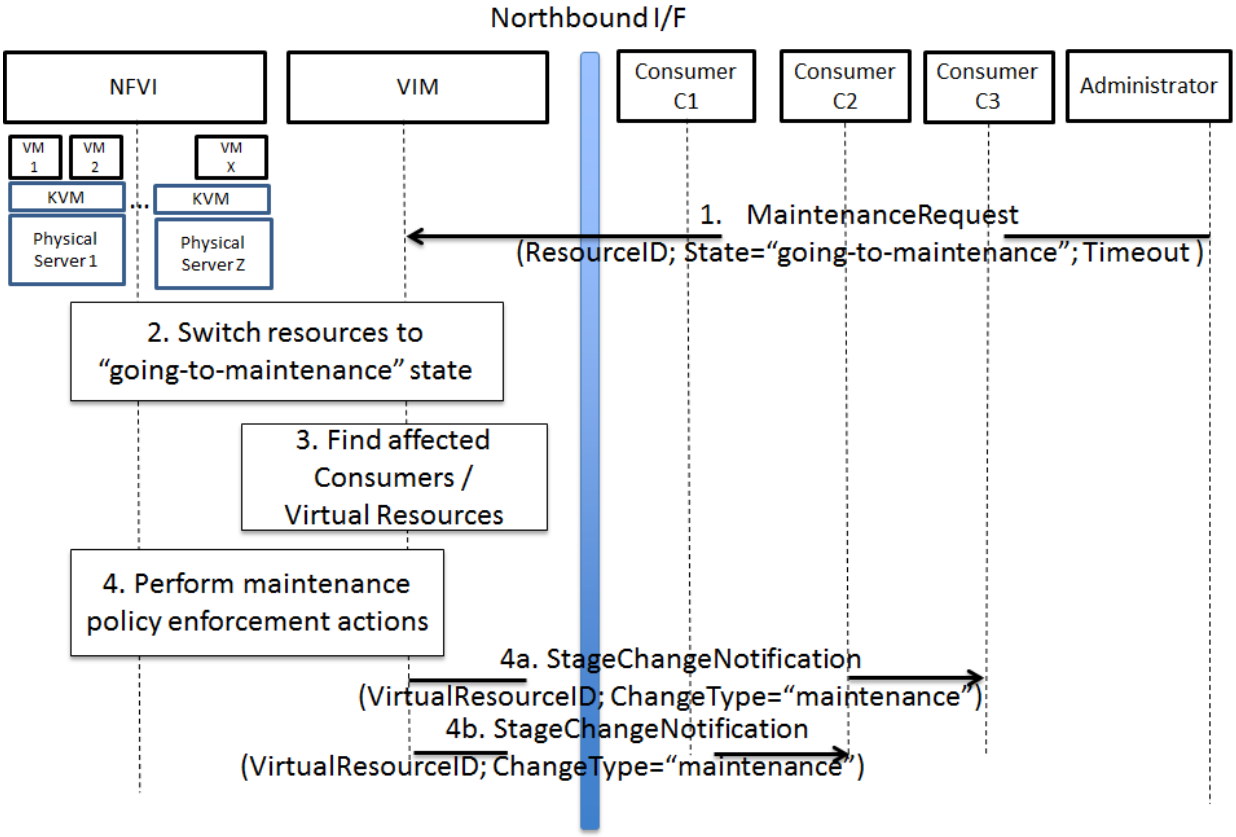


Fig. 8: High-level message flow for maintenance policy enforcement

The high level message flow for the NFVI maintenance policy enforcement is shown in `figure5a`. It consists of the following steps:

1. Maintenance trigger received from Administrator.
2. VIM switches the affected physical resources to "going-to-maintenance" state e.g. so that no new VM will be scheduled on the physical servers.
3. Database lookup to find the Consumer(s) and virtual resources affected by the maintenance operation.

[1] Timeout is set by the Administrator and corresponds to the maximum time to empty the physical resources.

4. Maintenance policies are enforced in the VIM, e.g. affected VM(s) are shut down on the physical server(s), or affected Consumer(s) are notified about the planned maintenance operation (steps 4a/4b).

Once the affected Consumer(s) have been notified, they take specific actions (e.g. switch to standby (STBY) configuration, request to terminate the virtual resource(s)) to allow the maintenance action to be executed. After the physical resources have been emptied, the VIM puts the physical resources in "in-maintenance" state and sends a MaintenanceResponse back to the Administrator.

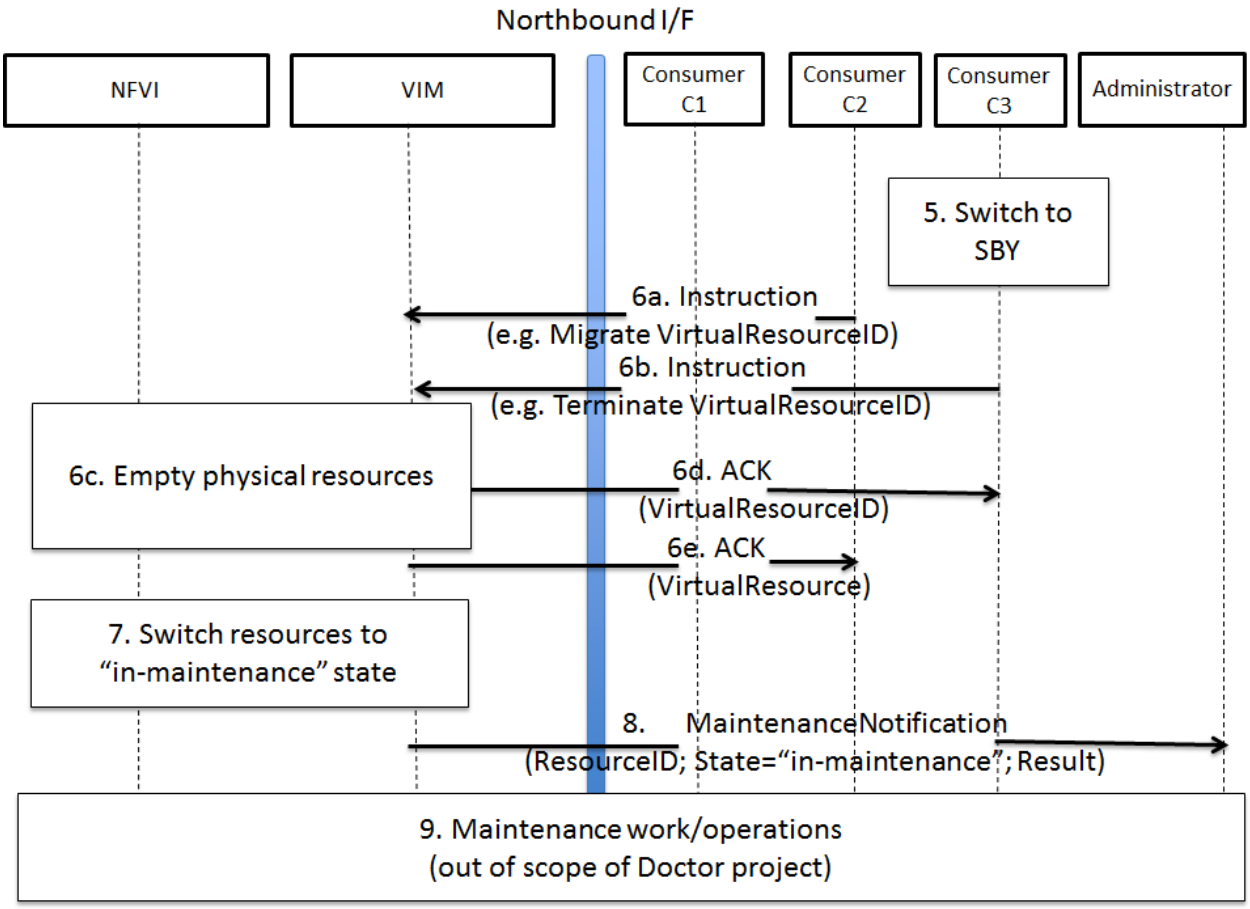


Fig. 9: Successful NFVI maintenance

The high level message flow for a successful NFVI maintenance is show in `figure5b`. It consists of the following steps:

5. The Consumer C3 switches to standby configuration (STBY).
6. Instructions from Consumers C2/C3 are shared to VIM requesting certain actions to be performed (steps 6a, 6b). After receiving such instructions, the VIM executes the requested action in order to empty the physical resources (step 6c) and informs the Consumer about the result of the actions (steps 6d, 6e).
7. The VIM switches the physical resources to "in-maintenance" state
8. Maintenance response is sent from VIM to inform the Administrator that the physical servers have been emptied.
9. The Administrator is coordinating and executing the maintenance operation/work on the NFVI. Note: this step is out of scope of Doctor project.

The requested actions to empty the physical resources may not be successful (e.g. migration fails or takes too long) and in such a case, the VIM puts the physical resources back to 'enabled' and informs the Administrator about the

problem.

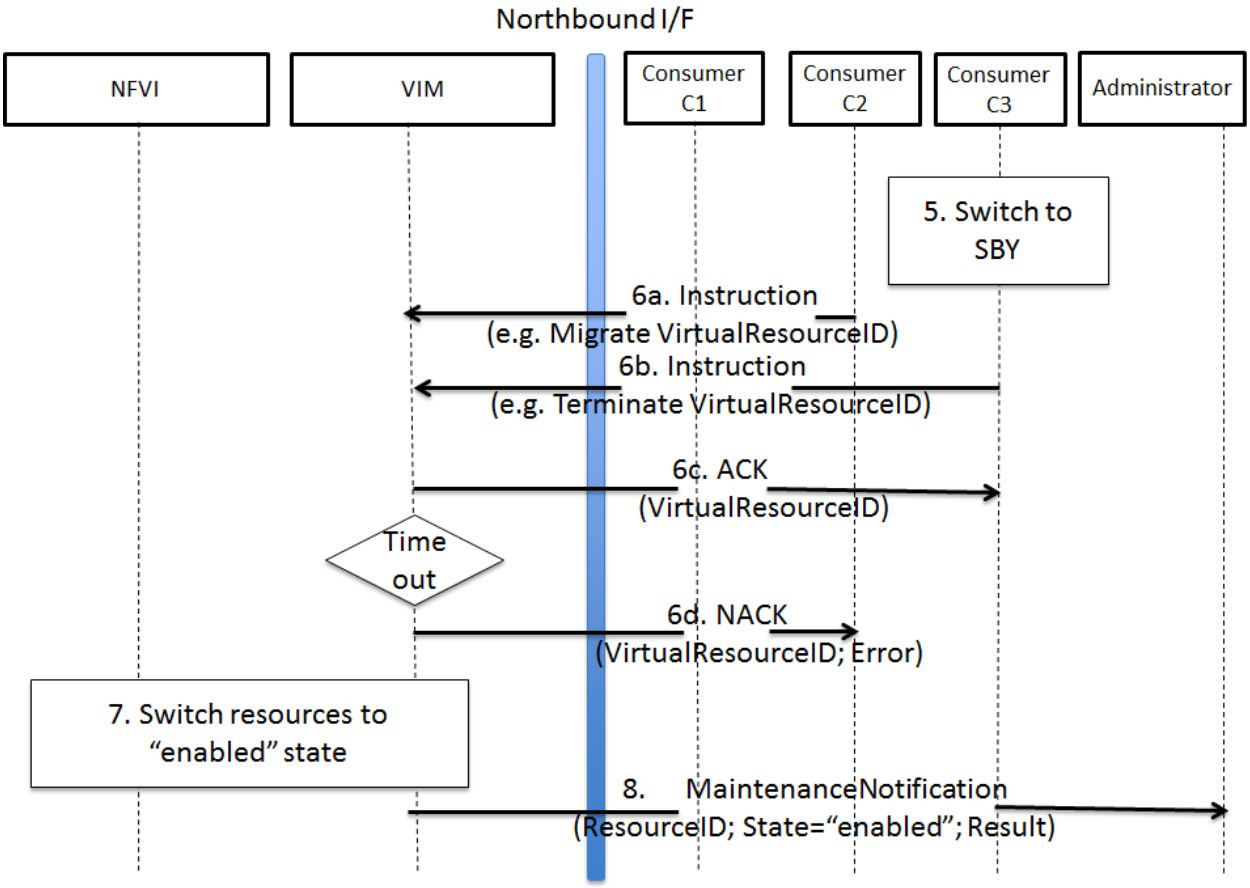


Fig. 10: Example of failed NFVI maintenance

An example of a high level message flow to cover the failed NFVI maintenance case is shown in `figure5c`. It consists of the following steps:

5. The Consumer C3 switches to standby configuration (STBY).
6. Instructions from Consumers C2/C3 are shared to VIM requesting certain actions to be performed (steps 6a, 6b). The VIM executes the requested actions and sends back a NACK to consumer C2 (step 6d) as the migration of the virtual resource(s) is not completed by the given timeout.
7. The VIM switches the physical resources to "enabled" state.
8. MaintenanceNotification is sent from VIM to inform the Administrator that the maintenance action cannot start.

2.2.4 Gap analysis in upstream projects

This section presents the findings of gaps on existing VIM platforms. The focus was to identify gaps based on the features and requirements specified in Section 3.3. The analysis work determined gaps that are presented here.

VIM Northbound Interface

Immediate Notification

- Type: 'deficiency in performance'
- Description
  - To-be
    - VIM has to notify unavailability of virtual resource (fault) to VIM user immediately.
    - Notification should be passed in '1 second' after fault detected/notified by VIM.
    - Also, the following conditions/requirement have to be met:
      - Only the owning user can receive notification of fault related to owned virtual resource(s).
  - As-is
    - OpenStack Metering 'Ceilometer' can notify unavailability of virtual resource (fault) to the owner of virtual resource based on alarm configuration by the user.
      - Ceilometer Alarm API: http://docs.openstack.org/developer/ceilometer/webapi/v2.html#alarms
    - Alarm notifications are triggered by alarm evaluator instead of notification agents that might receive faults
      - Ceilometer Architecture: http://docs.openstack.org/developer/ceilometer/architecture.html#id1
    - Evaluation interval should be equal to or larger than configured pipeline interval for collection of underlying metrics.
      - https://github.com/openstack/ceilometer/blob/stable/juno/ceilometer/alarm/service.py#L38-42
    - The interval for collection has to be set large enough which depends on the size of the deployment and the number of metrics to be collected.
    - The interval may not be less than one second in even small deployments. The default value is 60 seconds.
    - Alternative: OpenStack has a message bus to publish system events. The operator can allow the user to connect this, but there are no functions to filter out other events that should not be passed to the user or which were not requested by the user.
  - Gap
    - Fault notifications cannot be received immediately by Ceilometer.
- Solved by
  - Event Alarm Evaluator: https://specs.openstack.org/openstack/ceilometer-specs/specs/liberty/event-alarm-evaluator.html
  - New OpenStack alarms and notifications project AODH: http://docs.openstack.org/developer/aodh/

Maintenance Notification

- Type: 'missing'
- Description
  - To-be
    - VIM has to notify unavailability of virtual resource triggered by NFVI maintenance to VIM user.

        * Also, the following conditions/requirements have to be met:
            - VIM should accept maintenance message from administrator and mark target physical resource "in maintenance".
            - Only the owner of virtual resource hosted by target physical resource can receive the notification that can trigger some process for applications which are running on the virtual resource (e.g. cut off VM).
    - As-is
        * OpenStack: None
        * AWS (just for study)
            - AWS provides API and CLI to view status of resource (VM) and to create instance status and system status alarms to notify you when an instance has a failed status check. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/monitoring-instances-status-check_sched.html
            - AWS provides API and CLI to view scheduled events, such as a reboot or retirement, for your instances. Also, those events will be notified via e-mail. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/monitoring-system-instance-status-check.html
    - Gap
        * VIM user cannot receive maintenance notifications.
- Solved by
    - https://blueprints.launchpad.net/nova/+spec/service-status-notification

VIM Southbound interface

Normalization of data collection models

- Type: 'missing'
- Description
    - To-be
        * A normalized data format needs to be created to cope with the many data models from different monitoring solutions.
    - As-is
        * Data can be collected from many places (e.g. Zabbix, Nagios, Cacti, Zenoss). Although each solution establishes its own data models, no common data abstraction models exist in OpenStack.
    - Gap
        * Normalized data format does not exist.
- Solved by
    - Specification in Section *Detailed southbound interface specification*.

OpenStack

Ceilometer

OpenStack offers a telemetry service, Ceilometer, for collecting measurements of the utilization of physical and virtual resources *[CEIL]*. Ceilometer can collect a number of metrics across multiple OpenStack components and watch for variations and trigger alarms based upon the collected data.

Scalability of fault aggregation

- Type: 'scalability issue'
- Description
  - To-be
    - Be able to scale to a large deployment, where thousands of monitoring events per second need to be analyzed.
  - As-is
    - Performance issue when scaling to medium-sized deployments.
  - Gap
    - Ceilometer seems to be unsuitable for monitoring medium and large scale NFVI deployments.
- Solved by
  - Usage of Zabbix for fault aggregation *[ZABB]*. Zabbix can support a much higher number of fault events (up to 15 thousand events per second, but obviously also has some upper bound: http://blog.zabbix.com/scalable-zabbix-lessons-on-hitting-9400-nvps/2615/
  - Decentralized/hierarchical deployment with multiple instances, where one instance is only responsible for a small NFVI.

Monitoring of hardware and software

- Type: 'missing (lack of functionality)'
- Description
  - To-be
    - OpenStack (as VIM) should monitor various hardware and software in NFVI to handle faults on them by Ceilometer.
    - OpenStack may have monitoring functionality in itself and can be integrated with third party monitoring tools.
    - OpenStack need to be able to detect the faults listed in the Annex.
  - As-is
    - For each deployment of OpenStack, an operator has responsibility to configure monitoring tools with relevant scripts or plugins in order to monitor hardware and software.
    - OpenStack Ceilometer does not monitor hardware and software to capture faults.
  - Gap
    - Ceilometer is not able to detect and handle all faults listed in the Annex.

- Solved by
  - Use of dedicated monitoring tools like Zabbix or Monasca. See *Annex: NFVI Faults*.

Nova

OpenStack Nova *[NOVA]* is a mature and widely known and used component in OpenStack cloud deployments. It is the main part of an "infrastructure-as-a-service" system providing a cloud computing fabric controller, supporting a wide diversity of virtualization and container technologies.

Nova has proven throughout these past years to be highly available and fault-tolerant. Featuring its own API, it also provides a compatibility API with Amazon EC2 APIs.

Correct states when compute host is down

- Type: 'missing (lack of functionality)'
- Description
  - To-be
    - The API shall support to change VM power state in case host has failed.
    - The API shall support to change nova-compute state.
    - There could be single API to change different VM states for all VMs belonging to a specific host.
    - Support external systems that are monitoring the infrastructure and resources that are able to call the API fast and reliable.
    - Resource states are reliable such that correlation actions can be fast and automated.
    - User shall be able to read states from OpenStack and trust they are correct.
  - As-is
    - When a VM goes down due to a host HW, host OS or hypervisor failure, nothing happens in OpenStack. The VMs of a crashed host/hypervisor are reported to be live and OK through the OpenStack API.
    - nova-compute state might change too slowly or the state is not reliable if expecting also VMs to be down. This leads to ability to schedule VMs to a failed host and slowness blocks evacuation.
  - Gap
    - OpenStack does not change its states fast and reliably enough.
    - The API does not support to have an external system to change states and to trust the states are reliable (external system has fenced failed host).
    - User cannot read all the states from OpenStack nor trust they are right.
- Solved by
  - https://blueprints.launchpad.net/nova/+spec/mark-host-down
  - https://blueprints.launchpad.net/python-novaclient/+spec/support-force-down-service

Evacuate VMs in Maintenance mode

- Type: 'missing'
- Description
  - To-be
    - When maintenance mode for a compute host is set, trigger VM evacuation to available compute nodes before bringing the host down for maintenance.
  - As-is
    - If setting a compute node to a maintenance mode, OpenStack only schedules evacuation of all VMs to available compute nodes if in-maintenance compute node runs the XenAPI and VMware ESX hypervisors. Other hypervisors (e.g. KVM) are not supported and, hence, guest VMs will likely stop running due to maintenance actions administrator may perform (e.g. hardware upgrades, OS updates).
  - Gap
    - Nova libvirt hypervisor driver does not implement automatic guest VMs evacuation when compute nodes are set to maintenance mode (`$ nova host-update --maintenance enable <hostname>`).

Monasca

Monasca is an open-source monitoring-as-a-service (MONaaS) solution that integrates with OpenStack. Even though it is still in its early days, it is the interest of the community that the platform be multi-tenant, highly scalable, performant and fault-tolerant. It provides a streaming alarm engine, a notification engine, and a northbound REST API users can use to interact with Monasca. Hundreds of thousands of metrics per second can be processed *[MONA]*.

Anomaly detection

- Type: 'missing (lack of functionality)'
- Description
  - To-be
    - Detect the failure and perform a root cause analysis to filter out other alarms that may be triggered due to their cascading relation.
  - As-is
    - A mechanism to detect root causes of failures is not available.
  - Gap
    - Certain failures can trigger many alarms due to their dependency on the underlying root cause of failure. Knowing the root cause can help filter out unnecessary and overwhelming alarms.
- Status
  - Monasca as of now lacks this feature, although the community is aware and working toward supporting it.

Sensor monitoring

- Type: 'missing (lack of functionality)'

- Description
  - To-be
    - It should support monitoring sensor data retrieval, for instance, from IPMI.
  - As-is
    - Monasca does not monitor sensor data
  - Gap
    - Sensor monitoring is very important. It provides operators status on the state of the physical infrastructure (e.g. temperature, fans).
- Addressed by
  - Monasca can be configured to use third-party monitoring solutions (e.g. Nagios, Cacti) for retrieving additional data.

Hardware monitoring tools

Zabbix

Zabbix is an open-source solution for monitoring availability and performance of infrastructure components (i.e. servers and network devices), as well as applications *[ZABB]*. It can be customized for use with OpenStack. It is a mature tool and has been proven to be able to scale to large systems with 100,000s of devices.

Delay in execution of actions

- Type: 'deficiency in performance'
- Description
  - To-be
    - After detecting a fault, the monitoring tool should immediately execute the appropriate action, e.g. inform the manager through the NB I/F
  - As-is
    - A delay of around 10 seconds was measured in two independent testbed deployments
  - Gap
    - Cause of the delay is a periodic evaluation and notification. Periodicity is configured as 30s default value and can be reduced to 5s but not below. https://github.com/zabbix/zabbix/blob/trunk/conf/zabbix_server.conf#L329

2.2.5 Detailed architecture and interface specification

This section describes a detailed implementation plan, which is based on the high level architecture introduced in Section 3. Section 5.1 describes the functional blocks of the Doctor architecture, which is followed by a high level message flow in Section 5.2. Section 5.3 provides a mapping of selected existing open source components to the building blocks of the Doctor architecture. Thereby, the selection of components is based on their maturity and the gap analysis executed in Section 4. Sections 5.4 and 5.5 detail the specification of the related northbound interface and the related information elements. Finally, Section 5.6 provides a first set of blueprints to address selected gaps required for the realization functionalities of the Doctor project.

Functional Blocks

This section introduces the functional blocks to form the VIM. OpenStack was selected as the candidate for implementation. Inside the VIM, 4 different building blocks are defined (see `figure6`).

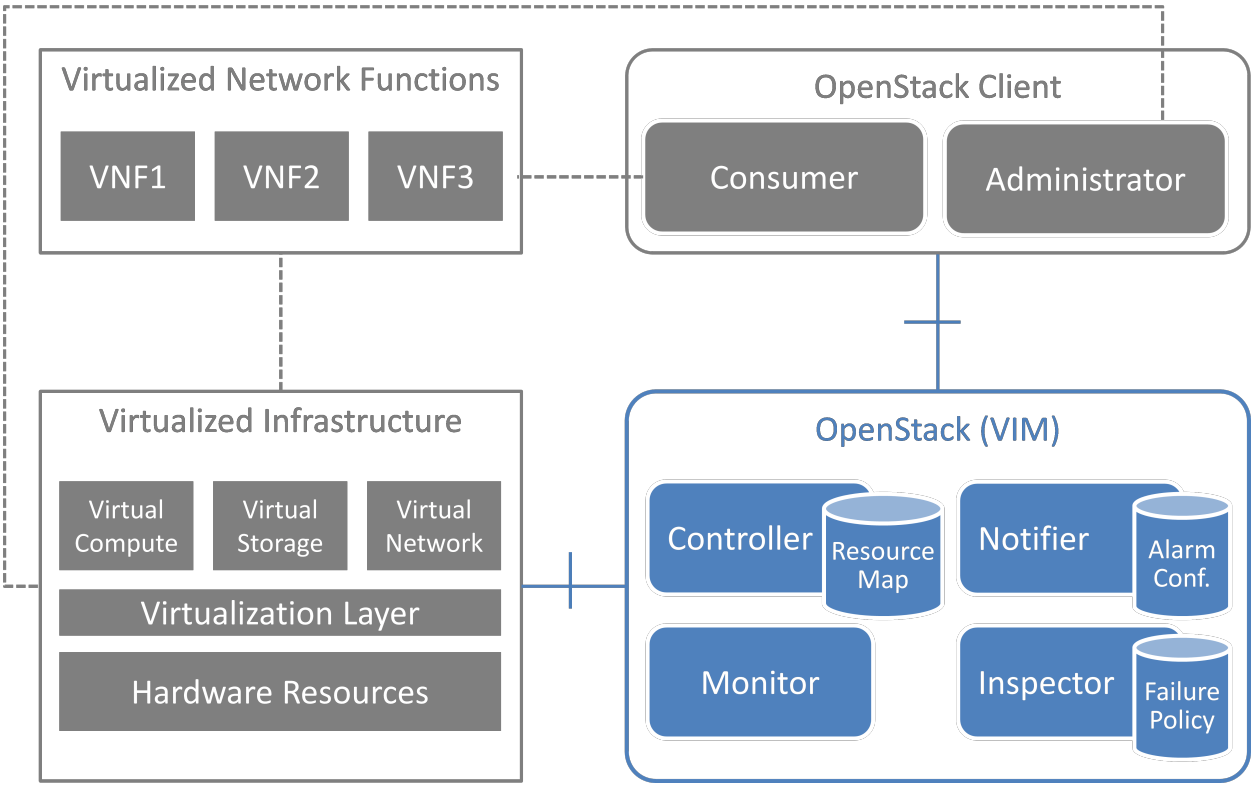


Fig. 11: Functional blocks

Monitor

The Monitor module has the responsibility for monitoring the virtualized infrastructure. There are already many existing tools and services (e.g. Zabbix) to monitor different aspects of hardware and software resources which can be used for this purpose.

Inspector

The Inspector module has the ability a) to receive various failure notifications regarding physical resource(s) from Monitor module(s), b) to find the affected virtual resource(s) by querying the resource map in the Controller, and c) to update the state of the virtual resource (and physical resource).

The Inspector has drivers for different types of events and resources to integrate any type of Monitor and Controller modules. It also uses a failure policy database to decide on the failure selection and aggregation from raw events. This failure policy database is configured by the Administrator.

The reason for separation of the Inspector and Controller modules is to make the Controller focus on simple operations by avoiding a tight integration of various health check mechanisms into the Controller.

Controller

The Controller is responsible for maintaining the resource map (i.e. the mapping from physical resources to virtual resources), accepting update requests for the resource state(s) (exposing as provider API), and sending all failure events regarding virtual resources to the Notifier. Optionally, the Controller has the ability to force the state of a given physical resource to down in the resource mapping when it receives failure notifications from the Inspector for that given physical resource. The Controller also re-calculates the capacity of the NVFI when receiving a failure notification for a physical resource.

In a real-world deployment, the VIM may have several controllers, one for each resource type, such as Nova, Neutron and Cinder in OpenStack. Each controller maintains a database of virtual and physical resources which shall be the master source for resource information inside the VIM.

Notifier

The focus of the Notifier is on selecting and aggregating failure events received from the controller based on policies mandated by the Consumer. Therefore, it allows the Consumer to subscribe for alarms regarding virtual resources using a method such as API endpoint. After receiving a fault event from a Controller, it will notify the fault to the Consumer by referring to the alarm configuration which was defined by the Consumer earlier on.

To reduce complexity of the Controller, it is a good approach for the Controllers to emit all notifications without any filtering mechanism and have another service (i.e. Notifier) handle those notifications properly. This is the general philosophy of notifications in OpenStack. Note that a fault message consumed by the Notifier is different from the fault message received by the Inspector; the former message is related to virtual resources which are visible to users with relevant ownership, whereas the latter is related to raw devices or small entities which should be handled with an administrator privilege.

The northbound interface between the Notifier and the Consumer/Administrator is specified in *Detailed northbound interface specification*.

Sequence

Fault Management

The detailed work flow for fault management is as follows (see also `figure7`):

1. Request to subscribe to monitor specific virtual resources. A query filter can be used to narrow down the alarms the Consumer wants to be informed about.
2. Each subscription request is acknowledged with a subscribe response message. The response message contains information about the subscribed virtual resources, in particular if a subscribed virtual resource is in "alarm" state.
3. The NFVI sends monitoring events for resources the VIM has been subscribed to. Note: this subscription message exchange between the VIM and NFVI is not shown in this message flow.
4. Event correlation, fault detection and aggregation in VIM.
5. Database lookup to find the virtual resources affected by the detected fault.
6. Fault notification to Consumer.
7. The Consumer switches to standby configuration (STBY)
8. Instructions to VIM requesting certain actions to be performed on the affected resources, for example migrate/update/terminate specific resource(s). After reception of such instructions, the VIM is executing the requested action, e.g. it will migrate or terminate a virtual resource.

1. Query request from Consumer to VIM to get information about the current status of a resource.
2. Response to the query request with information about the current status of the queried resource. In case the resource is in "fault" state, information about the related fault(s) is returned.

In order to allow for quick reaction to failures, the time interval between fault detection in step 3 and the corresponding recovery actions in step 7 and 8 shall be less than 1 second.

`figure8` shows a more detailed message flow (Steps 4 to 6) between the 4 building blocks introduced in *Functional Blocks*.

4. The Monitor observed a fault in the NFVI and reports the raw fault to the Inspector. The Inspector filters and aggregates the faults using pre-configured failure policies.
5. a) The Inspector queries the Resource Map to find the virtual resources affected by the raw fault in the NFVI. b) The Inspector updates the state of the affected virtual resources in the Resource Map. c) The Controller observes a change of the virtual resource state and informs the Notifier about the state change and the related alarm(s). Alternatively, the Inspector may directly inform the Notifier about it.
6. The Notifier is performing another filtering and aggregation of the changes and alarms based on the pre-configured alarm configuration. Finally, a fault notification is sent to northbound to the Consumer.

NFVI Maintenance

The detailed work flow for NFVI maintenance is shown in `figure9` and has the following steps. Note that steps 1, 2, and 5 to 8a in the NFVI maintenance work flow are very similar to the steps in the fault management work flow and share a similar implementation plan in Release 1.

1. Subscribe to fault/maintenance notifications.
2. Response to subscribe request.
3. Maintenance trigger received from administrator.
4. VIM switches NFVI resources to "maintenance" state. This, e.g., means they should not be used for further allocation/migration requests
5. Database lookup to find the virtual resources affected by the detected maintenance operation.
6. Maintenance notification to Consumer.
7. The Consumer switches to standby configuration (STBY)
8. Instructions from Consumer to VIM requesting certain recovery actions to be performed (step 8a). After reception of such instructions, the VIM is executing the requested action in order to empty the physical resources (step 8b).
9. Maintenance response from VIM to inform the Administrator that the physical machines have been emptied (or the operation resulted in an error state).
10. Administrator is coordinating and executing the maintenance operation/work on the NFVI.

1. Query request from Administrator to VIM to get information about the current state of a resource.
2. Response to the query request with information about the current state of the queried resource(s). In case the resource is in "maintenance" state, information about the related maintenance operation is returned.

`figure10` shows a more detailed message flow (Steps 3 to 6 and 9) between the 4 building blocks introduced in Section 5.1..

3. The Administrator is sending a StateChange request to the Controller residing in the VIM.

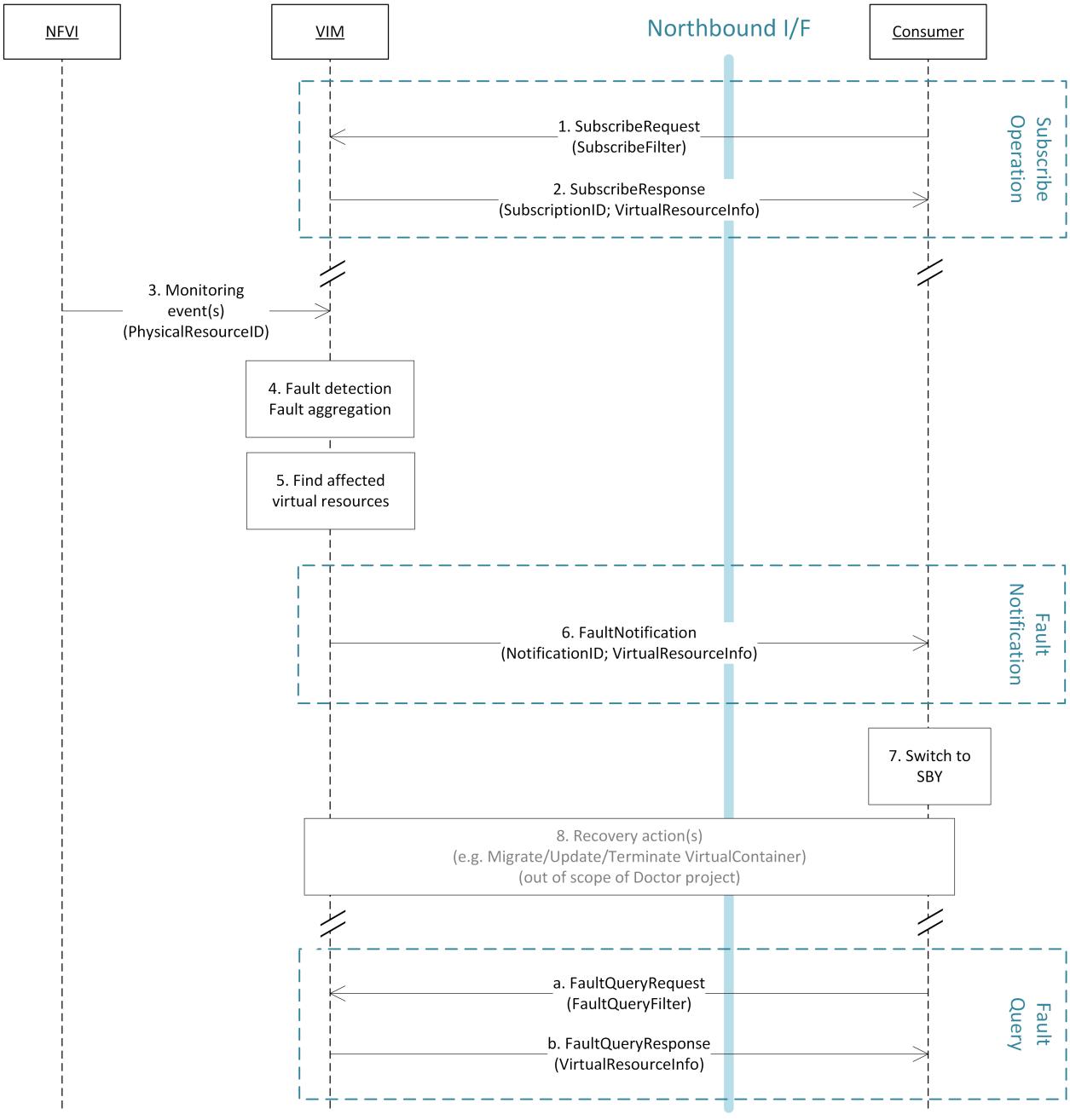


Fig. 12: Fault management work flow

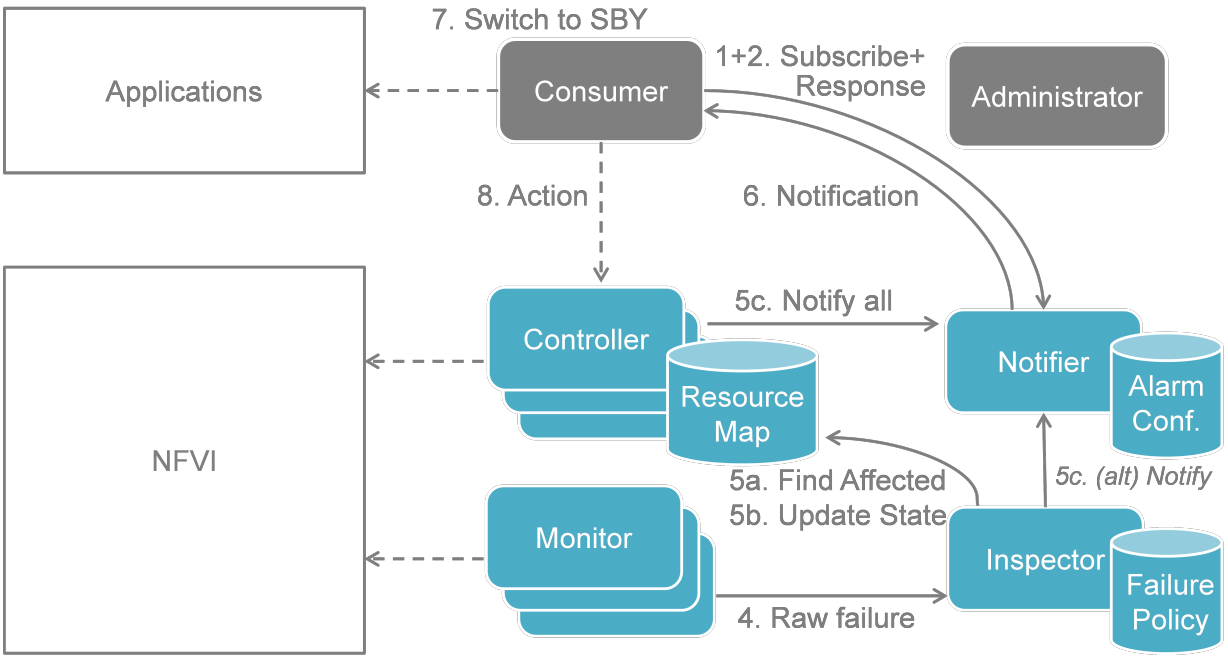


Fig. 13: Fault management scenario

4. The Controller queries the Resource Map to find the virtual resources affected by the planned maintenance operation.
5. a) The Controller updates the state of the affected virtual resources in the Resource Map database.

   b) The Controller informs the Notifier about the virtual resources that will be affected by the maintenance operation.
6. A maintenance notification is sent to northbound to the Consumer.

...

9. The Controller informs the Administrator after the physical resources have been freed.

Information elements

This section introduces all attributes and information elements used in the messages exchange on the northbound interfaces between the VIM and the VNFO and VNFM.

Note: The information elements will be aligned with current work in ETSI NFV IFA working group.

Simple information elements:

- SubscriptionID (Identifier): identifies a subscription to receive fault or maintenance notifications.
- NotificationID (Identifier): identifies a fault or maintenance notification.
- VirtualResourceID (Identifier): identifies a virtual resource affected by a fault or a maintenance action of the underlying physical resource.
- PhysicalResourceID (Identifier): identifies a physical resource affected by a fault or maintenance action.
- VirtualResourceState (String): state of a virtual resource, e.g. “normal”, “maintenance”, “down”, “error”.
- PhysicalResourceState (String): state of a physical resource, e.g. “normal”, “maintenance”, “down”, “error”.

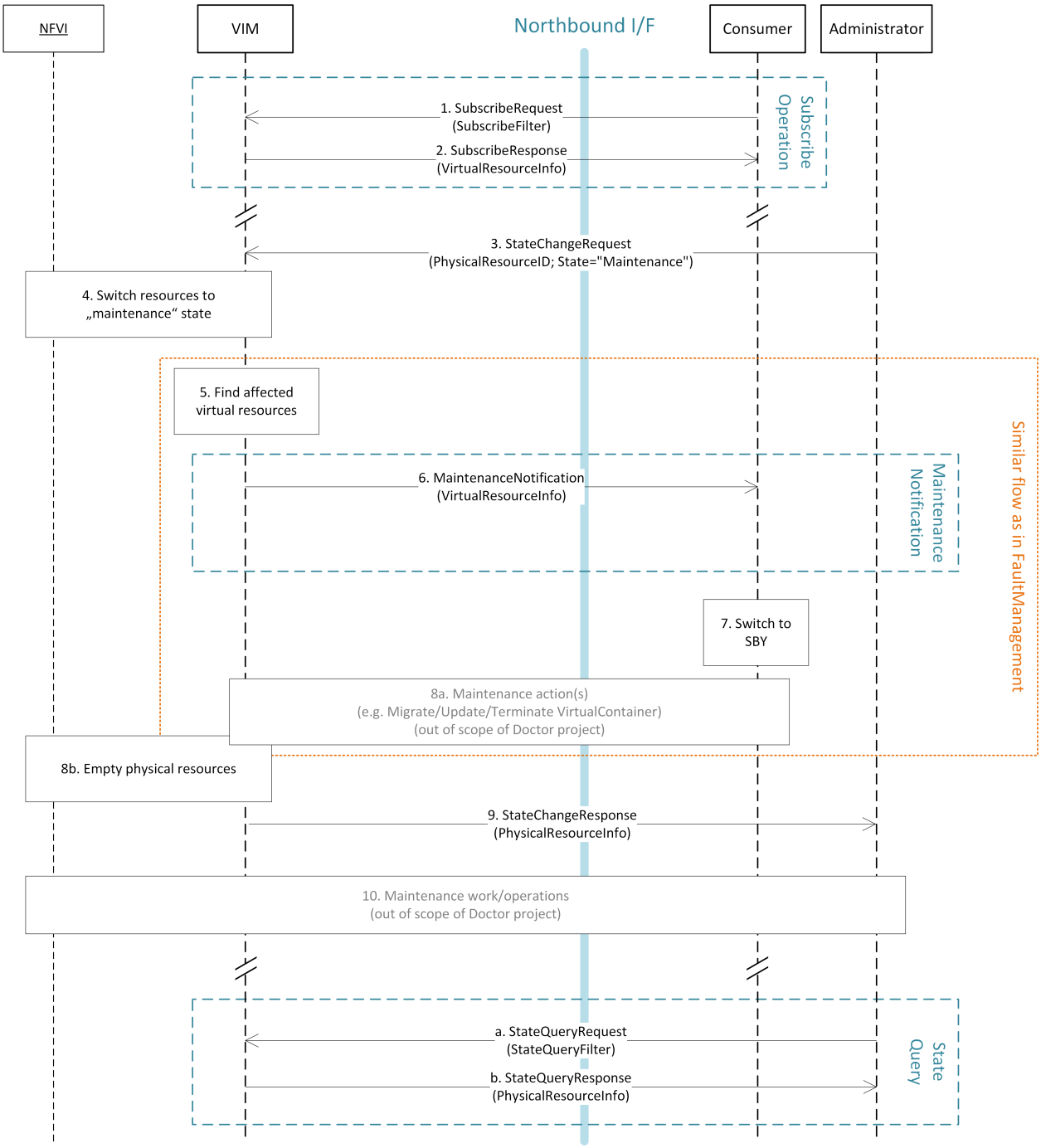


Fig. 14: NFVI maintenance work flow

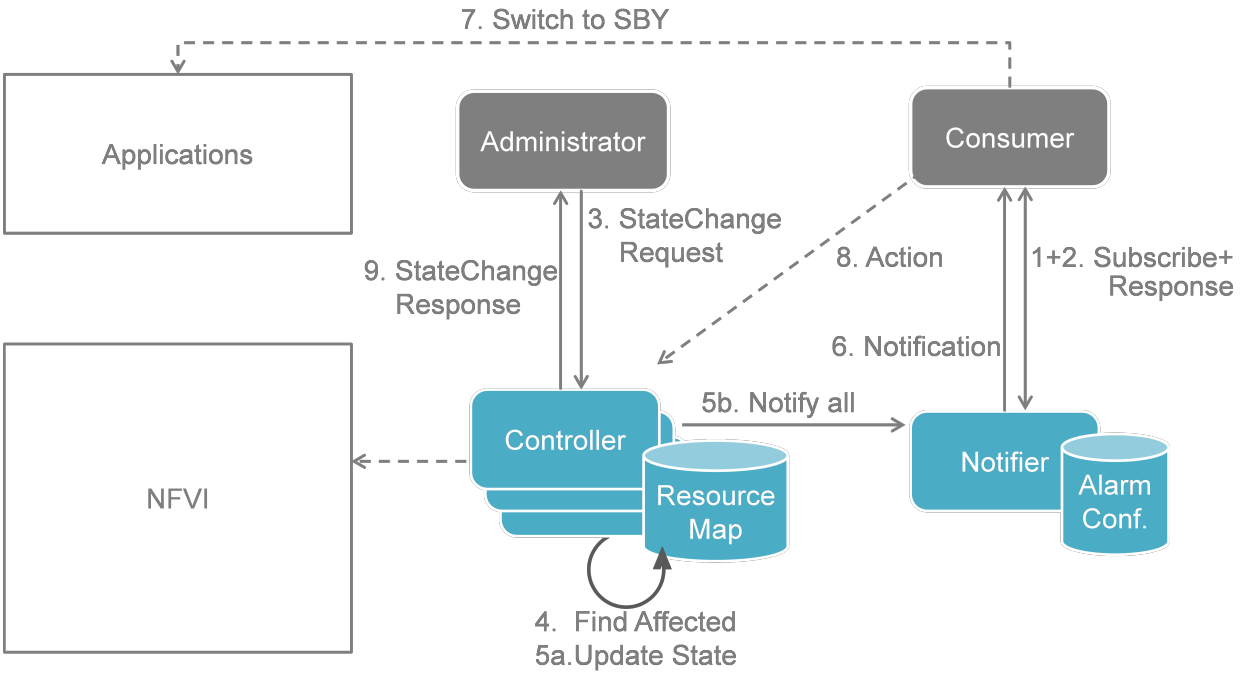


Fig. 15: NFVI Maintenance scenario

- VirtualResourceType (String): type of the virtual resource, e.g. "virtual machine", "virtual memory", "virtual storage", "virtual CPU", or "virtual NIC".
- FaultID (Identifier): identifies the related fault in the underlying physical resource. This can be used to correlate different fault notifications caused by the same fault in the physical resource.
- FaultType (String): Type of the fault. The allowed values for this parameter depend on the type of the related physical resource. For example, a resource of type "compute hardware" may have faults of type "CPU failure", "memory failure", "network card failure", etc.
- Severity (Integer): value expressing the severity of the fault. The higher the value, the more severe the fault.
- MinSeverity (Integer): value used in filter information elements. Only faults with a severity higher than the MinSeverity value will be notified to the Consumer.
- EventTime (Datetime): Time when the fault was observed.
- EventStartTime and EventEndTime (Datetime): Datetime range that can be used in a FaultQueryFilter to narrow down the faults to be queried.
- ProbableCause (String): information about the probable cause of the fault.
- CorrelatedFaultID (Integer): list of other faults correlated to this fault.
- isRootCause (Boolean): Parameter indicating if this fault is the root for other correlated faults. If TRUE, then the faults listed in the parameter CorrelatedFaultID are caused by this fault.
- FaultDetails (Key-value pair): provides additional information about the fault, e.g. information about the threshold, monitored attributes, indication of the trend of the monitored parameter.
- FirmwareVersion (String): current version of the firmware of a physical resource.
- HypervisorVersion (String): current version of a hypervisor.

- ZoneID (Identifier): Identifier of the resource zone. A resource zone is the logical separation of physical and software resources in an NFVI deployment for physical isolation, redundancy, or administrative designation.
- Metadata (Key-value pair): provides additional information of a physical resource in maintenance/error state.

Complex information elements (see also UML diagrams in `figure13` and `figure14`):

- VirtualResourceInfoClass:
  - VirtualResourceID [1] (Identifier)
  - VirtualResourceState [1] (String)
  - Faults [0..*] (FaultClass): For each resource, all faults including detailed information about the faults are provided.
- FaultClass: The parameters of the FaultClass are partially based on ETSI TS 132 111-2 (V12.1.0)*[0], which is specifying fault management in 3GPP, in particular describing the information elements used for alarm notifications.
  - FaultID [1] (Identifier)
  - FaultType [1] (String)
  - Severity [1] (Integer)
  - EventTime [1] (Datetime)
  - ProbableCause [1] (String)
  - CorrelatedFaultID [0..*] (Identifier)
  - FaultDetails [0..*] (Key-value pair)
- SubscribeFilterClass
  - VirtualResourceType [0..*] (String)
  - VirtualResourceID [0..*] (Identifier)
  - FaultType [0..*] (String)
  - MinSeverity [0..1] (Integer)
- FaultQueryFilterClass: narrows down the FaultQueryRequest, for example it limits the query to certain physical resources, a certain zone, a given fault type/severity/cause, or a specific FaultID.
  - VirtualResourceType [0..*] (String)
  - VirtualResourceID [0..*] (Identifier)
  - FaultType [0..*] (String)
  - MinSeverity [0..1] (Integer)
  - EventStartTime [0..1] (Datetime)
  - EventEndTime [0..1] (Datetime)
- PhysicalResourceStateClass:
  - PhysicalResourceID [1] (Identifier)
  - PhysicalResourceState [1] (String): mandates the new state of the physical resource.
  - Metadata [0..*] (Key-value pair)
- PhysicalResourceInfoClass:

[0] http://www.etsi.org/deliver/etsi_ts/132100_132199/13211102/12.01.00_60/ts_13211102v120100p.pdf

  - PhysicalResourceID [1] (Identifier)
  - PhysicalResourceState [1] (String)
  - FirmwareVersion [0..1] (String)
  - HypervisorVersion [0..1] (String)
  - ZoneID [0..1] (Identifier)
  - Metadata [0..*] (Key-value pair)
- StateQueryFilterClass: narrows down a StateQueryRequest, for example it limits the query to certain physical resources, a certain zone, or a given resource state (e.g., only resources in "maintenance" state).
  - PhysicalResourceID [1] (Identifier)
  - PhysicalResourceState [1] (String)
  - ZoneID [0..1] (Identifier)

Detailed northbound interface specification

This section is specifying the northbound interfaces for fault management and NFVI maintenance between the VIM on the one end and the Consumer and the Administrator on the other ends. For each interface all messages and related information elements are provided.

Note: The interface definition will be aligned with current work in ETSI NFV IFA working group .

All of the interfaces described below are produced by the VIM and consumed by the Consumer or Administrator.

Fault management interface

This interface allows the VIM to notify the Consumer about a virtual resource that is affected by a fault, either within the virtual resource itself or by the underlying virtualization infrastructure. The messages on this interface are shown in `figure13` and explained in detail in the following subsections.

Note: The information elements used in this section are described in detail in Section 5.4.

SubscribeRequest (Consumer -> VIM)

Subscription from Consumer to VIM to be notified about faults of specific resources. The faults to be notified about can be narrowed down using a subscribe filter.

Parameters:

- SubscribeFilter [1] (SubscribeFilterClass): Optional information to narrow down the faults that shall be notified to the Consumer, for example limit to specific VirtualResourceID(s), severity, or cause of the alarm.

SubscribeResponse (VIM -> Consumer)

Response to a subscribe request message including information about the subscribed resources, in particular if they are in "fault/error" state.

Parameters:

- SubscriptionID [1] (Identifier): Unique identifier for the subscription. It can be used to delete or update the subscription.

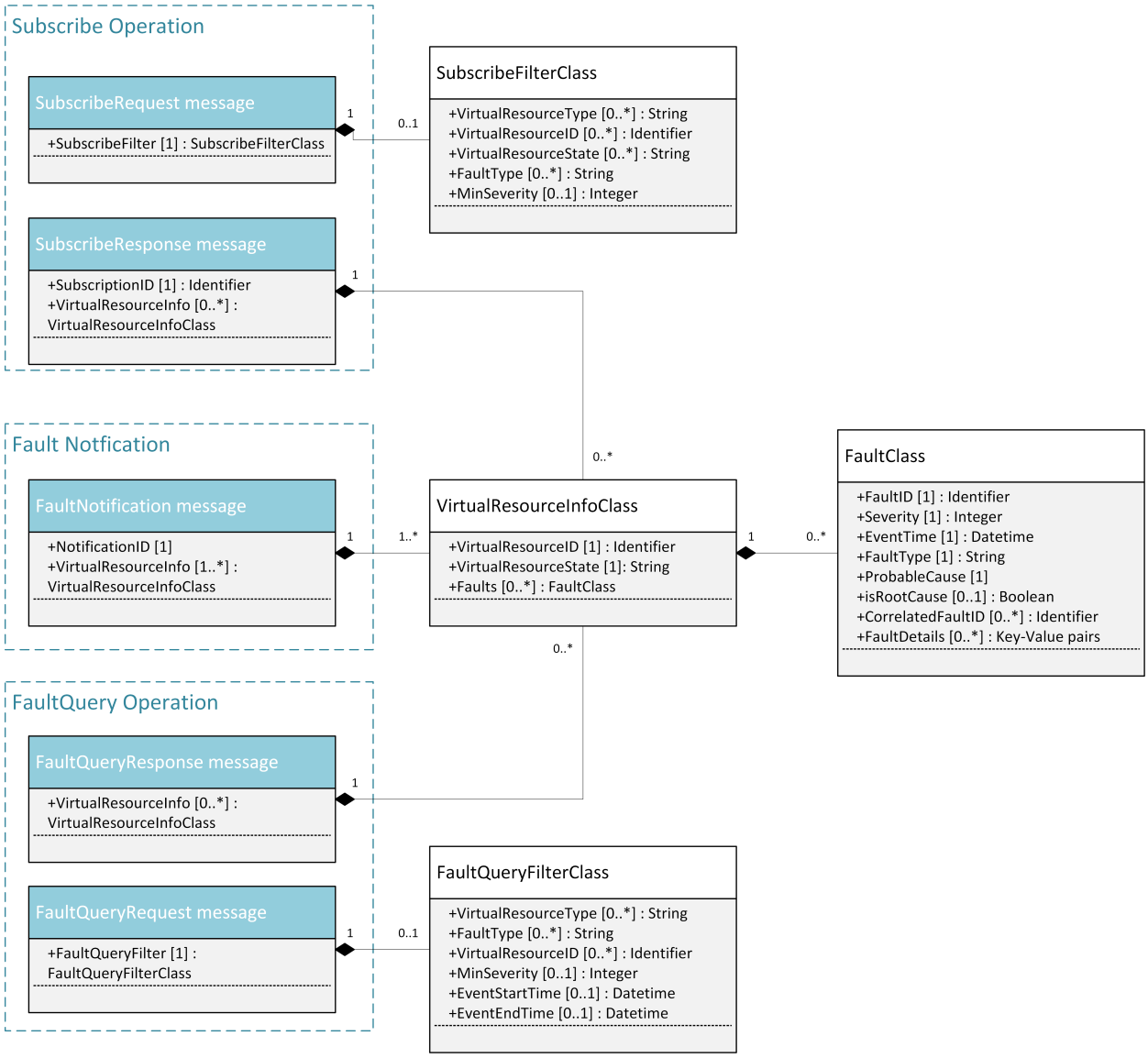


Fig. 16: Fault management NB I/F messages

- VirtualResourceInfo [0..*] (VirtualResourceInfoClass): Provides additional information about the subscribed resources, i.e., a list of the related resources, the current state of the resources, etc.

FaultNotification (VIM -> Consumer)

Notification about a virtual resource that is affected by a fault, either within the virtual resource itself or by the underlying virtualization infrastructure. After reception of this request, the Consumer will decide on the optimal action to resolve the fault. This includes actions like switching to a hot standby virtual resource, migration of the fault virtual resource to another physical machine, termination of the faulty virtual resource and instantiation of a new virtual resource in order to provide a new hot standby resource. In some use cases the Consumer can leave virtual resources on failed host to be booted up again after fault is recovered. Existing resource management interfaces and messages between the Consumer and the VIM can be used for those actions, and there is no need to define additional actions on the Fault Management Interface.

Parameters:

- NotificationID [1] (Identifier): Unique identifier for the notification.
- VirtualResourceInfo [1..*] (VirtualResourceInfoClass): List of faulty resources with detailed information about the faults.

FaultQueryRequest (Consumer -> VIM)

Request to find out about active alarms at the VIM. A FaultQueryFilter can be used to narrow down the alarms returned in the response message.

Parameters:

- FaultQueryFilter [1] (FaultQueryFilterClass): narrows down the FaultQueryRequest, for example it limits the query to certain physical resources, a certain zone, a given fault type/severity/cause, or a specific FaultID.

FaultQueryResponse (VIM -> Consumer)

List of active alarms at the VIM matching the FaultQueryFilter specified in the FaultQueryRequest.

Parameters:

- VirtualResourceInfo [0..*] (VirtualResourceInfoClass): List of faulty resources. For each resource all faults including detailed information about the faults are provided.

NFVI maintenance

The NFVI maintenance interfaces Consumer-VIM allows the Consumer to subscribe to maintenance notifications provided by the VIM. The related maintenance interface Administrator-VIM allows the Administrator to issue maintenance requests to the VIM, i.e. requesting the VIM to take appropriate actions to empty physical machine(s) in order to execute maintenance operations on them. The interface also allows the Administrator to query the state of physical machines, e.g., in order to get details in the current status of the maintenance operation like a firmware update.

The messages defined in these northbound interfaces are shown in `figure14` and described in detail in the following subsections.

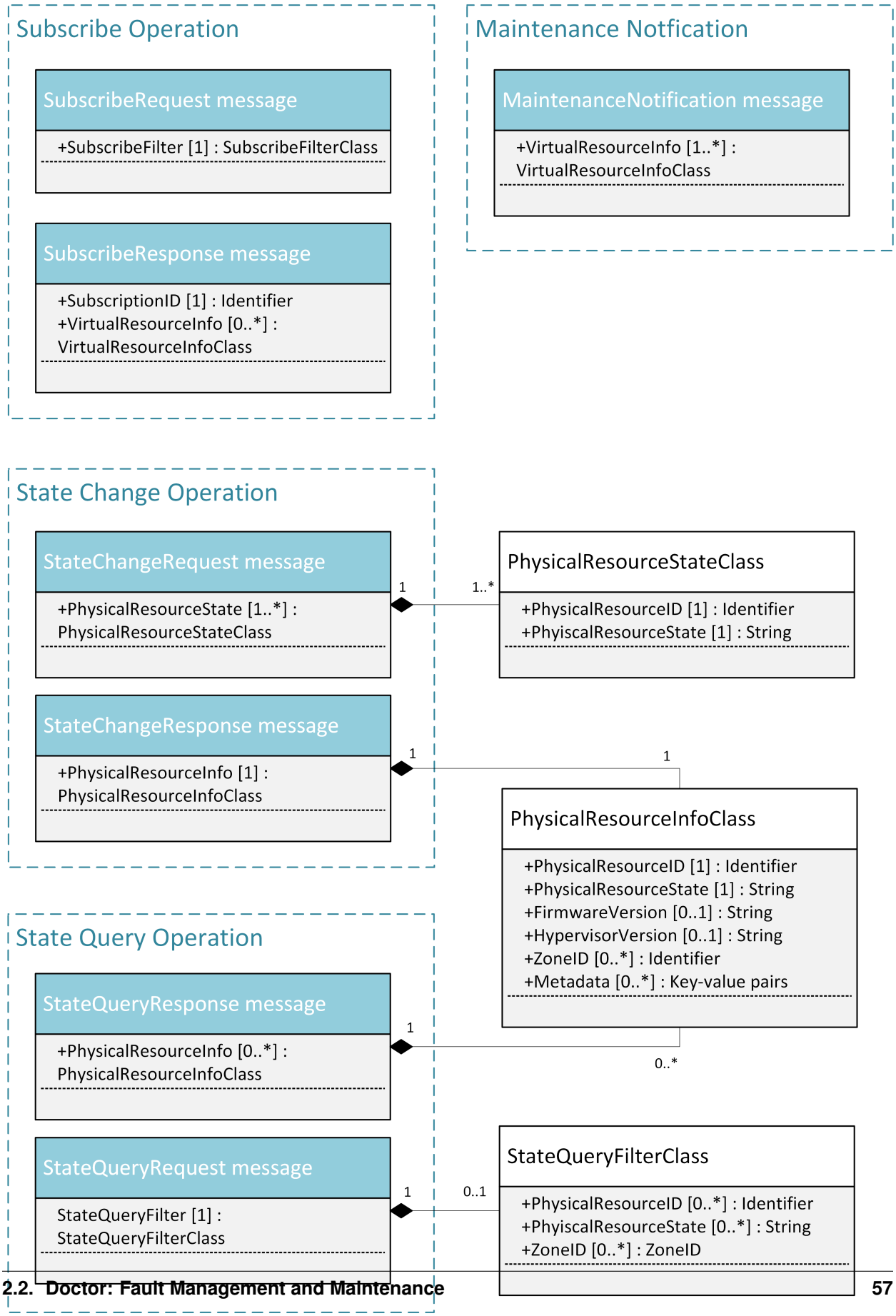


Fig. 17: NFVI maintenance NB I/F messages

SubscribeRequest (Consumer -> VIM)

Subscription from Consumer to VIM to be notified about maintenance operations for specific virtual resources. The resources to be informed about can be narrowed down using a subscribe filter.

Parameters:

- SubscribeFilter [1] (SubscribeFilterClass): Information to narrow down the faults that shall be notified to the Consumer, for example limit to specific virtual resource type(s).

SubscribeResponse (VIM -> Consumer)

Response to a subscribe request message, including information about the subscribed virtual resources, in particular if they are in "maintenance" state.

Parameters:

- SubscriptionID [1] (Identifier): Unique identifier for the subscription. It can be used to delete or update the subscription.
- VirtualResourceInfo [0..*] (VirtalResourceInfoClass): Provides additional information about the subscribed virtual resource(s), e.g., the ID, type and current state of the resource(s).

MaintenanceNotification (VIM -> Consumer)

Notification about a physical resource switched to "maintenance" state. After reception of this request, the Consumer will decide on the optimal action to address this request, e.g., to switch to the standby (STBY) configuration.

Parameters:

- VirtualResourceInfo [1..*] (VirtualResourceInfoClass): List of virtual resources where the state has been changed to maintenance.

StateChangeRequest (Administrator -> VIM)

Request to change the state of a list of physical resources, e.g. to "maintenance" state, in order to prepare them for a planned maintenance operation.

Parameters:

- PhysicalResourceState [1..*] (PhysicalResourceStateClass)

StateChangeResponse (VIM -> Administrator)

Response message to inform the Administrator that the requested resources are now in maintenance state (or the operation resulted in an error) and the maintenance operation(s) can be executed.

Parameters:

- PhysicalResourceInfo [1..*] (PhysicalResourceInfoClass)

StateQueryRequest (Administrator -> VIM)

In this procedure, the Administrator would like to get the information about physical machine(s), e.g. their state ("normal", "maintenance"), firmware version, hypervisor version, update status of firmware and hypervisor, etc. It can be used to check the progress during firmware update and the confirmation after update. A filter can be used to narrow down the resources returned in the response message.

Parameters:

- StateQueryFilter [1] (StateQueryFilterClass): narrows down the StateQueryRequest, for example it limits the query to certain physical resources, a certain zone, or a given resource state.

StateQueryResponse (VIM -> Administrator)

List of physical resources matching the filter specified in the StateQueryRequest.

Parameters:

- PhysicalResourceInfo [0..*] (PhysicalResourceInfoClass): List of physical resources. For each resource, information about the current state, the firmware version, etc. is provided.

NFV IFA, OPNFV Doctor and AODH alarms

This section compares the alarm interfaces of ETSI NFV IFA with the specifications of this document and the alarm class of AODH.

ETSI NFV specifies an interface for alarms from virtualised resources in ETSI GS NFV-IFA 005 *[ENFV]*. The interface specifies an Alarm class and two notifications plus operations to query alarm instances and to subscribe to the alarm notifications.

The specification in this document has a structure that is very similar to the ETSI NFV specifications. The notifications differ in that an alarm notification in the NFV interface defines a single fault for a single resource while the notification specified in this document can contain multiple faults for multiple resources. The Doctor specification is lacking the detailed time stamps of the NFV specification essential for synchronizaion of the alarm list using the query operation. The detailed time stamps are also of value in the event and alarm history DBs.

AODH defines a base class for alarms, not the notifications. This means that some of the dynamic attributes of the ETSI NFV alarm type, like alarmRaisedTime, are not applicable to the AODH alarm class but are attributes of in the actual notifications. (Description of these attributes will be added later.) The AODH alarm class is lacking some attributes present in the NFV specification, fault details and correlated alarms. Instead the AODH alarm class has attributes for actions, rules and user and project id.

ETSI NFV Alarm Type	OPNFV Doctor Requirement Specs	AODH Event Alarm Notification	Description / Comment	Recommendations
alarmId	FaultId	alarm_id	Identifier of an alarm.	-
-	-	alarm_name	Human readable alarm name.	May be added in ETSI NFV Stage 3.
managedObjectId	VirtualResourceId	(reason)	Identifier of the affected virtual resource is part of the AODH reason parameter.	-
-	-	user_id, project_id	User and project identifiers.	May be added in ETSI NFV Stage 3.
alarmRaisedTime	-	-	Timestamp when alarm was raised.	To be added to Doctor and AODH. May be derived (e.g. in a shimlayer) from the AODH alarm history.
alarmChangedTime	-	-	Timestamp when alarm was changed/updated.	see above
alarmClearedTime	-	-	Timestamp when alarm was cleared.	see above
eventTime	-	-	Timestamp when alarm was first observed by the Monitor.	see above
-	EventTime	generated	Timestamp of the Notification.	Update parameter name in Doctor spec. May be added in ETSI NFV Stage 3.
state: E.g. Fired, Updated Cleared	VirtualResourceState: E.g. normal, down maintenance, error	current: ok, alarm, insufficient_data	ETSI NFV IFA 005/006 lists example alarm states.	Maintenance state is missing in AODH. List of alarm states will be specified in ETSI NFV Stage 3.
perceivedSeverity: E.g. Critical, Major, Minor, Warning, Indeterminate, Cleared	Severity (Integer)	Severity: low (default), moderate, critical	ETSI NFV IFA 005/006 lists example perceived severity values.	List of alarm states will be specified in ETSI NFV Stage 3. **OPNFV: Severity (Integer):** • update OPNFV Doctor specification to *Enum* **perceivedSeverity=Indetermined:** • remove value *Indetermined* in IFA and map unde-

Table: Comparison of alarm attributes

The primary area of improvement should be alignment of the perceived severity. This is important for a quick and accurate evaluation of the alarm. AODH thus should support also the X.733 values Critical, Major, Minor, Warning and Indeterminate.

The detailed time stamps (raised, changed, cleared) which are essential for synchronizing the alarm list using a query operation should be added to the Doctor specification.

Other areas that need alignment is the so called alarm state in NFV. Here we must however consider what can be attributes of the notification vs. what should be a property of the alarm instance. This will be analyzed later.

Detailed southbound interface specification

This section is specifying the southbound interfaces for fault management between the Monitors and the Inspector. Although southbound interfaces should be flexible to handle various events from different types of Monitors, we define unified event API in order to improve interoperability between the Monitors and the Inspector. This is not limiting implementation of Monitor and Inspector as these could be extended in order to support failures from intelligent inspection like prediction.

Note: The interface definition will be aligned with current work in ETSI NFV IFA working group.

Fault event interface

This interface allows the Monitors to notify the Inspector about an event which was captured by the Monitor and may effect resources managed in the VIM.

EventNotification

Event notification including fault description. The entity of this notification is event, and not fault or error specifically. This allows us to use generic event format or framework build out of Doctor project. The parameters below shall be mandatory, but keys in 'Details' can be optional.

Parameters:

- Time [1]: Datetime when the fault was observed in the Monitor.
- Type [1]: Type of event that will be used to process correlation in Inspector.
- Details [0..1]: Details containing additional information with Key-value pair style. Keys shall be defined depending on the Type of the event.

E.g.:

```
{
    'event': {
        'time': '2016-04-12T08:00:00',
        'type': 'compute.host.down',
        'details': {
            'hostname': 'compute-1',
            'source': 'sample_monitor',
            'cause': 'link-down',
            'severity': 'critical',
            'status': 'down',
            'monitor_id': 'monitor-1',
            'monitor_event_id': '123',
```

(continues on next page)

(continued from previous page)

```
        }
    }
}
```

Optional parameters in 'Details':

- Hostname: the hostname on which the event occurred.
- Source: the display name of reporter of this event. This is not limited to monitor, other entity can be specified such as 'KVM'.
- Cause: description of the cause of this event which could be different from the type of this event.
- Severity: the severity of this event set by the monitor.
- Status: the status of target object in which error occurred.
- MonitorID: the ID of the monitor sending this event.
- MonitorEventID: the ID of the event in the monitor. This can be used by operator while tracking the monitor log.
- RelatedTo: the array of IDs which related to this event.

Also, we can have bulk API to receive multiple events in a single HTTP POST message by using the 'events' wrapper as follows:

```
{
    'events': [
        'event': {
            'time': '2016-04-12T08:00:00',
            'type': 'compute.host.down',
            'details': {},
        },
        'event': {
            'time': '2016-04-12T08:00:00',
            'type': 'compute.host.nic.error',
            'details': {},
        }
    ]
}
```

Blueprints

This section is listing a first set of blueprints that have been proposed by the Doctor project to the open source community. Further blueprints addressing other gaps identified in Section 4 will be submitted at a later stage of the OPNFV. In this section the following definitions are used:

- "Event" is a message emitted by other OpenStack services such as Nova and Neutron and is consumed by the "Notification Agents" in Ceilometer.
- "Notification" is a message generated by a "Notification Agent" in Ceilometer based on an "event" and is delivered to the "Collectors" in Ceilometer that store those notifications (as "sample") to the Ceilometer "Databases".

Instance State Notification (Ceilometer)†[0]

†The Doctor project is planning to handle "events" and "notifications" regarding Resource Status; Instance State, Port State, Host State, etc. Currently, Ceilometer already receives "events" to identify the state of those resources, but it does not handle and store them yet. This is why we also need a new event definition to capture those resource states from "events" created by other services.

This BP proposes to add a new compute notification state to handle events from an instance (server) from nova. It also creates a new meter "instance.state" in OpenStack.

Event Publisher for Alarm (Ceilometer)‡[0]

‡**Problem statement:**

The existing "Alarm Evaluator" in OpenStack Ceilometer is periodically querying/polling the databases in order to check all alarms independently from other processes. This is adding additional delay to the fault notification send to the Consumer, whereas one requirement of Doctor is to react on faults as fast as possible.

The existing message flow is shown in `figure12`: after receiving an "event", a "notification agent" (i.e. "event publisher") will send a "notification" to a "Collector". The "collector" is collecting the notifications and is updating the Ceilometer "Meter" database that is storing information about the "sample" which is capured from original "event". The "Alarm Evaluator" is periodically polling this databases then querying "Meter" database based on each alarm configuration.

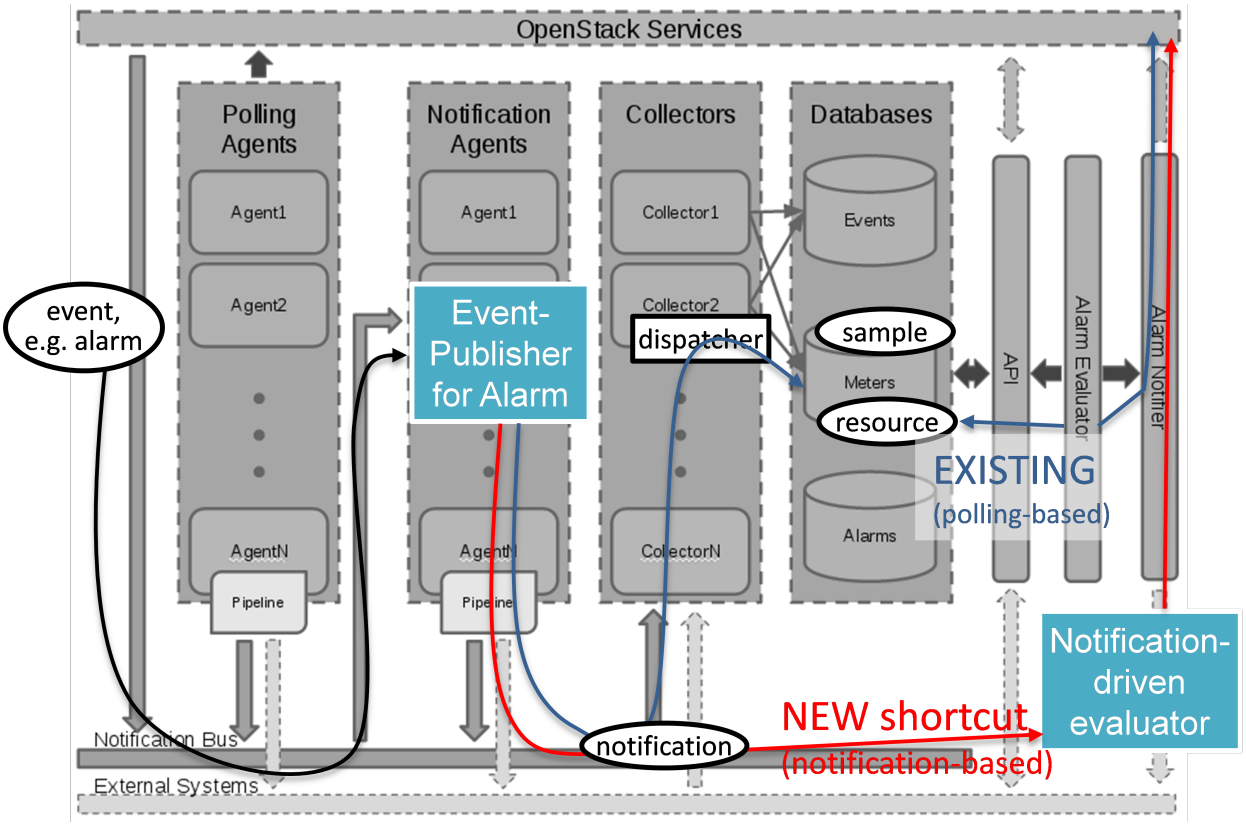


Fig. 18: Implementation plan in Ceilometer architecture

[0] https://etherpad.opnfv.org/p/doctor_bps
[0] https://etherpad.opnfv.org/p/doctor_bps

In the current Ceilometer implementation, there is no possibility to directly trigger the "Alarm Evaluator" when a new "event" was received, but the "Alarm Evaluator" will only find out that requires firing new notification to the Consumer when polling the database.

Change/feature request:

This BP proposes to add a new "event publisher for alarm", which is bypassing several steps in Ceilometer in order to avoid the polling-based approach of the existing Alarm Evaluator that makes notification slow to users. See `figure12`.

After receiving an "(alarm) event" by listening on the Ceilometer message queue ("notification bus"), the new "event publisher for alarm" immediately hands a "notification" about this event to a new Ceilometer component "Notification-driven alarm evaluator" proposed in the other BP (see Section 5.6.3).

Note, the term "publisher" refers to an entity in the Ceilometer architecture (it is a "notification agent"). It offers the capability to provide notifications to other services outside of Ceilometer, but it is also used to deliver notifications to other Ceilometer components (e.g. the "Collectors") via the Ceilometer "notification bus".

Implementation detail

- "Event publisher for alarm" is part of Ceilometer
- The standard AMQP message queue is used with a new topic string.
- No new interfaces have to be added to Ceilometer.
- "Event publisher for Alarm" can be configured by the Administrator of Ceilometer to be used as "Notification Agent" in addition to the existing "Notifier"
- Existing alarm mechanisms of Ceilometer can be used allowing users to configure how to distribute the "notifications" transformed from "events", e.g. there is an option whether an ongoing alarm is re-issued or not ("repeat_actions").

Notification-driven alarm evaluator (Ceilometer)[159]

Problem statement:

The existing "Alarm Evaluator" in OpenStack Ceilometer is periodically querying/polling the databases in order to check all alarms independently from other processes. This is adding additional delay to the fault notification send to the Consumer, whereas one requirement of Doctor is to react on faults as fast as possible.

Change/feature request:

This BP is proposing to add an alternative "Notification-driven Alarm Evaluator" for Ceilometer that is receiving "notifications" sent by the "Event Publisher for Alarm" described in the other BP. Once this new "Notification-driven Alarm Evaluator" received "notification", it finds the "alarm" configurations which may relate to the "notification" by querying the "alarm" database with some keys i.e. resource ID, then it will evaluate each alarm with the information in that "notification".

After the alarm evaluation, it will perform the same way as the existing "alarm evaluator" does for firing alarm notification to the Consumer. Similar to the existing Alarm Evaluator, this new "Notification-driven Alarm Evaluator" is aggregating and correlating different alarms which are then provided northbound to the Consumer via the OpenStack "Alarm Notifier". The user/administrator can register the alarm configuration via existing Ceilometer API¶[0]. Thereby, he can configure whether to set an alarm or not and where to send the alarms to.

Implementation detail

- The new "Notification-driven Alarm Evaluator" is part of Ceilometer.

[159] https://etherpad.opnfv.org/p/doctor_bps
[0] https://wiki.openstack.org/wiki/Ceilometer/Alerting

- Most of the existing source code of the "Alarm Evaluator" can be re-used to implement this BP
- No additional application logic is needed
- It will access the Ceilometer Databases just like the existing "Alarm evaluator"
- Only the polling-based approach will be replaced by a listener for "notifications" provided by the "Event Publisher for Alarm" on the Ceilometer "notification bus".
- No new interfaces have to be added to Ceilometer.

Report host fault to update server state immediately (Nova)[0]

Problem statement:

- Nova state change for failed or unreachable host is slow and does not reliably state host is down or not. This might cause same server instance to run twice if action taken to evacuate instance to another host.
- Nova state for server(s) on failed host will not change, but remains active and running. This gives the user false information about server state.
- VIM northbound interface notification of host faults towards VNFM and NFVO should be in line with OpenStack state. This fault notification is a Telco requirement defined in ETSI and will be implemented by OPNFV Doctor project.
- Openstack user cannot make HA actions fast and reliably by trusting server state and host state.

Proposed change:

There needs to be a new API for Admin to state host is down. This API is used to mark services running in host down to reflect the real situation.

Example on compute node is:

- When compute node is up and running::

```
vm_state: activeand power_state: running
nova-compute state: up status: enabled
```

- When compute node goes down and new API is called to state host is down::

```
vm_state: stopped power_state: shutdown
nova-compute state: down status: enabled
```

Alternatives:

There is no attractive alternative to detect all different host faults than to have an external tool to detect different host faults. For this kind of tool to exist there needs to be new API in Nova to report fault. Currently there must be some kind of workarounds implemented as cannot trust or get the states from OpenStack fast enough.

Other related BPs

This section lists some BPs related to Doctor, but proposed by drafters outside the OPNFV community.

[0] https://blueprints.launchpad.net/nova/+spec/update-server-state-immediately

pacemaker-servicegroup-driver[0]

This BP will detect and report host down quite fast to OpenStack. This however might not work properly for example when management network has some problem and host reported faulty while VM still running there. This might lead to launching same VM instance twice causing problems. Also NB IF message needs fault reason and for that the source needs to be a tool that detects different kind of faults as Doctor will be doing. Also this BP might need enhancement to change server and service states correctly.

2.2.6 Summary and conclusion

The Doctor project aimed at detailing NFVI fault management and NFVI maintenance requirements. These are indispensable operations for an Operator, and extremely necessary to realize telco-grade high availability. High availability is a large topic; the objective of Doctor is not to realize a complete high availability architecture and implementation. Instead, Doctor limited itself to addressing the fault events in NFVI, and proposes enhancements necessary in VIM, e.g. OpenStack, to ensure VNFs availability in such fault events, taking a Telco VNFs application level management system into account.

The Doctor project performed a robust analysis of the requirements from NFVI fault management and NFVI maintenance operation, concretely found out gaps in between such requirements and the current implementation of OpenStack. A detailed architecture and interface specification has been described in this document and work to realize Doctor features and fill out the identified gaps in upstream communities is in the final stages of development.

2.2.7 Annex: NFVI Faults

Faults in the listed elements need to be immediately notified to the Consumer in order to perform an immediate action like live migration or switch to a hot standby entity. In addition, the Administrator of the host should trigger a maintenance action to, e.g., reboot the server or replace a defective hardware element.

Faults can be of different severity, i.e., critical, warning, or info. Critical faults require immediate action as a severe degradation of the system has happened or is expected. Warnings indicate that the system performance is going down: related actions include closer (e.g. more frequent) monitoring of that part of the system or preparation for a cold migration to a backup VM. Info messages do not require any action. We also consider a type "maintenance", which is no real fault, but may trigger maintenance actions like a re-boot of the server or replacement of a faulty, but redundant HW.

Faults can be gathered by, e.g., enabling SNMP and installing some open source tools to catch and poll SNMP. When using for example Zabbix one can also put an agent running on the hosts to catch any other fault. In any case of failure, the Administrator should be notified. The following tables provide a list of high level faults that are considered within the scope of the Doctor project requiring immediate action by the Consumer.

Compute/Storage

[0] https://blueprints.launchpad.net/nova/+spec/pacemaker-servicegroup-driver

Fault	Severity	How to detect?	Comment	Immediate action to recover
Processor/CPU failure, CPU condition not ok	Critical	Zabbix		Switch to hot standby
Memory failure/ Memory condition not ok	Critical	Zabbix (IPMI)		Switch to hot standby
Network card failure, e.g. network adapter connectivity lost	Critical	Zabbix/ Ceilometer		Switch to hot standby
Disk crash	Info	RAID monitoring	Network storage is very redundant (e.g. RAID system) and can guarantee high availability	Inform OAM
Storage controller	Critical	Zabbix (IPMI)		Live migration if storage is still accessible; otherwise hot standby
PDU/power failure, power off, server reset	Critical	Zabbix/ Ceilometer		Switch to hot standby
Power degration, power redundancy lost, power threshold exceeded	Warning	SNMP		Live migration
Chassis problem (e.g. fan degraded/failed, chassis power degraded), CPU fan problem, temperature/ thermal condition not ok	Warning	SNMP		Live migration
Mainboard failure	Critical	Zabbix (IPMI)	e.g. PCIe, SAS link failure	Switch to hot standby
OS crash (e.g. kernel panic)	Critical	Zabbix		Switch to hot standby

Hypervisor

Fault	Severity	How to detect?	Comment	Immediate action to recover
System has restarted	Critical	Zabbix		Switch to hot standby
Hypervisor failure	Warning/ Critical	Zabbix/ Ceilometer		Evacuation/switch to hot standby
Hypervisor status not retrievable after certain period	Warning	Alarming service	Zabbix/ Ceilometer unreachable	Rebuild VM

Network

Fault	Severity	How to detect?	Comment	Immediate action to recover
SDN/OpenFlow switch, controller degraded/failed	Critical	Ceilometer		Switch to hot standby or reconfigure virtual network topology
Hardware failure of physical switch/router	Warning	SNMP	Redundancy of physical infrastructure is reduced or no longer available	Live migration if possible otherwise evacuation

2.2.8 References and bibliography

2.3 Manuals

2.3.1 OpenStack NOVA API for marking host down.

Related Blueprints:

https://blueprints.launchpad.net/nova/+spec/mark-host-down https://blueprints.launchpad.net/python-novaclient/+spec/support-force-down-service

What the API is for

This API will give external fault monitoring system a possibility of telling OpenStack Nova fast that compute host is down. This will immediately enable calling of evacuation of any VM on host and further enabling faster HA actions.

What this API does

In OpenStack the nova-compute service state can represent the compute host state and this new API is used to force this service down. It is assumed that the one calling this API has made sure the host is also fenced or powered down. This is important, so there is no chance same VM instance will appear twice in case evacuated to new compute host. When host is recovered by any means, the external system is responsible of calling the API again to disable forced_down flag and let the host nova-compute service report again host being up. If network fenced host come up again it should not boot VMs it had if figuring out they are evacuated to other compute host. The decision of deleting or booting VMs there used to be on host should be enhanced later to be more reliable by Nova blueprint: https://blueprints.launchpad.net/nova/+spec/robustify-evacuate

REST API for forcing down:

Parameter explanations: tenant_id: Identifier of the tenant. binary: Compute service binary name. host: Compute host name. forced_down: Compute service forced down flag. token: Token received after successful authentication. service_host_ip: Serving controller node ip.

request: PUT /v2.1/{tenant_id}/os-services/force-down { “binary”: “nova-compute”, “host”: “compute1”, “forced_down”: true }

response: 200 OK { “service”: { “host”: “compute1”, “binary”: “nova-compute”, “forced_down”: true } }

Example: curl -g -i -X PUT http://{service_host_ip}:8774/v2.1/{tenant_id}/os-services /force-down -H “Content-Type: application/json” -H “Accept: application/json ” -H “X-OpenStack-Nova-API-Version: 2.11” -H “X-Auth-Token: {token}” -d ‘{“b inary”: “nova-compute”, “host”: “compute1”, “forced_down”: true}’

CLI for forcing down:

nova service-force-down <hostname> nova-compute

Example: nova service-force-down compute1 nova-compute

REST API for disabling forced down:

Parameter explanations: tenant_id: Identifier of the tenant. binary: Compute service binary name. host: Compute host name. forced_down: Compute service forced down flag. token: Token received after successful authentication. service_host_ip: Serving controller node ip.

request: PUT /v2.1/{tenant_id}/os-services/force-down { “binary”: “nova-compute”, “host”: “compute1”, “forced_down”: false }

response: 200 OK { “service”: { “host”: “compute1”, “binary”: “nova-compute”, “forced_down”: false } }

Example: curl -g -i -X PUT http://{service_host_ip}:8774/v2.1/{tenant_id}/os-services /force-down -H “Content-Type: application/json” -H “Accept: application/json ” -H “X-OpenStack-Nova-API-Version: 2.11” -H “X-Auth-Token: {token}” -d ‘{“b inary”: “nova-compute”, “host”: “compute1”, “forced_down”: false}’

CLI for disabling forced down:

nova service-force-down –unset <hostname> nova-compute

Example: nova service-force-down –unset compute1 nova-compute

2.3.2 Get valid server state

Related Blueprints:

https://blueprints.launchpad.net/nova/+spec/get-valid-server-state

Problem description

Previously when the owner of a VM has queried his VMs, he has not received enough state information, states have not changed fast enough in the VIM and they have not been accurate in some scenarios. With this change this gap is now closed.

A typical case is that, in case of a fault of a host, the user of a high availability service running on top of that host, needs to make an immediate switch over from the faulty host to an active standby host. Now, if the compute host is forced down [1] as a result of that fault, the user has to be notified about this state change such that the user can react accordingly. Similarly, a change of the host state to “maintenance” should also be notified to the users.

What is changed

A new `host_status` parameter is added to the `/servers/{server_id}` and `/servers/detail` endpoints in microversion 2.16. By this new parameter user can get additional state information about the host.

`host_status` possible values where next value in list can override the previous:

- `UP` if nova-compute is up.
- `UNKNOWN` if nova-compute status was not reported by servicegroup driver within configured time period. Default is within 60 seconds, but can be changed with `service_down_time` in nova.conf.
- `DOWN` if nova-compute was forced down.
- `MAINTENANCE` if nova-compute was disabled. MAINTENANCE in API directly means nova-compute service is disabled. Different wording is used to avoid the impression that the whole host is down, as only scheduling of new VMs is disabled.
- Empty string indicates there is no host for server.

`host_status` is returned in the response in case the policy permits. By default the policy is for admin only in Nova policy.json:

```
"os_compute_api:servers:show:host_status": "rule:admin_api"
```

For an NFV use case this has to also be enabled for the owner of the VM:

```
"os_compute_api:servers:show:host_status": "rule:admin_or_owner"
```

REST API examples:

Case where nova-compute is enabled and reporting normally:

```
GET /v2.1/{tenant_id}/servers/{server_id}

200 OK
{
  "server": {
    "host_status": "UP",
    ...
  }
}
```

Case where nova-compute is enabled, but not reporting normally:

```
GET /v2.1/{tenant_id}/servers/{server_id}

200 OK
{
  "server": {
    "host_status": "UNKNOWN",
    ...
  }
}
```

Case where nova-compute is enabled, but forced_down:

```
GET /v2.1/{tenant_id}/servers/{server_id}

200 OK
{
  "server": {
    "host_status": "DOWN",
    ...
  }
}
```

Case where nova-compute is disabled:

```
GET /v2.1/{tenant_id}/servers/{server_id}

200 OK
{
  "server": {
    "host_status": "MAINTENANCE",
    ...
  }
}
```

Host Status is also visible in python-novaclient:

```
+-------+------+--------+------------+-------------+----------+-------------+
| ID    | Name | Status | Task State | Power State | Networks | Host Status |
+-------+------+--------+------------+-------------+----------+-------------+
| 9a... | vm1  | ACTIVE | -          | RUNNING     | xnet=... | UP          |
+-------+------+--------+------------+-------------+----------+-------------+
```

Links:

[1] Manual for OpenStack NOVA API for marking host down http://artifacts.opnfv.org/doctor/docs/manuals/mark-host-down_manual.html

[2] OpenStack compute manual page http://developer.openstack.org/api-ref-compute-v2.1.html#compute-v2.1

2.4 Indices

- search

Bibliography

[DOCT] OPNFV, “Doctor” requirements project, [Online]. Available at https://wiki.opnfv.org/doctor

[PRED] OPNFV, “Data Collection for Failure Prediction” requirements project [Online]. Available at https://wiki.opnfv.org/prediction

[OPSK] OpenStack, [Online]. Available at https://www.openstack.org/

[CEIL] OpenStack Telemetry (Ceilometer), [Online]. Available at https://wiki.openstack.org/wiki/Ceilometer

[NOVA] OpenStack Nova, [Online]. Available at https://wiki.openstack.org/wiki/Nova

[NEUT] OpenStack Neutron, [Online]. Available at https://wiki.openstack.org/wiki/Neutron

[CIND] OpenStack Cinder, [Online]. Available at https://wiki.openstack.org/wiki/Cinder

[MONA] OpenStack Monasca, [Online], Available at https://wiki.openstack.org/wiki/Monasca

[OSAG] OpenStack Cloud Administrator Guide, [Online]. Available at http://docs.openstack.org/admin-guide-cloud/content/

[ZABB] ZABBIX, the Enterprise-class Monitoring Solution for Everyone, [Online]. Available at http://www.zabbix.com/

[ENFV] ETSI NFV, [Online]. Available at http://www.etsi.org/technologies-clusters/technologies/nfv

Index