YCE Exchange gateway and API

The YCE exchange gateway is intended for integrations with north-bound systems although it can also be used to interface with peer systems. The gateway is an XML based request-response system where YCE can be instructed to perform an action or deliver information.

Several types of integrations have been realized to date and due to its highly flexible and extensible implementation, capable to include many custom or future functions. It functions include those of the YCE product's API.

At the core, each YCE server has a service running to accept incoming requests to execute specific tasks. These tasks can be specific to YCE or customer specific. YCE API functions available include the preparation of (standardized) changes (e.g. adding new devices, setting up services, manipulating topology) as well as the scheduling of the provisioning of these changes and their monitoring. Customer specific functions allow for the interaction with YCE-connected systems like Infoblox to perform IPAM, DNS and DHCP tasks.

Authorization

The request header requires both a `userid` and a `passwd`. The userid must match one of the local user-id's (not ldap!) or may be `xch` which is a built-in userid for the API only.

The password may be cleartext (not advised), the md5-hash from the NetYCE YCE.Users.Passwd column, or the des3 encrypted password that can be generated using the cli tool `/opt/yce/system/`**`api_crypt.sh`**

The md5-hash taken from the indicated table cannot be self-generated since it is a hash created using a concatenation of the userid and a secret realm string.

Implementation

The Exchange or API system consists of two parts, a daemon and a series of plugins.

First there is the xch-daemon that permanently runs in the background to accept new requests from remote systems using the network. It listens to port 8888 by default and is available on any of the YCE servers of its implementation.

The daemon can accept tcp socket calls over which it receives the request in XML format directly, but the method used most widely is the HTTP POST. In this case the XML formatted request will be issued as a parameter of the POST. During the processing of the request the network connection is kept alive until a response is sent. Depending on the transaction type, the response is available immediately or can take several minutes.

The Exchange daemon is multi-threaded so that requests are processed in parallel. Up to 30 requests can be executed in parallel, any additional requests are queued until a slot is available. During the queuing the connection remains open. From issuers perspective these calls are identical, just take a little longer.

The second part of the exchange gateway are the plugins. These plugins provide the actual implementation of the request and are therefore highly modular and easily extensible. Most of the integrations between NetYCE and external NMS systems to date are using xch-plugins. Also the various NetYCE API functions are realized as an xch-plugin.

The plugins currently available:

- NetYCE command jobs - xch_jobs
- NetYCE Service type and service task launcher - xch_st
- NetYCE NCCM function - xch_nccm
- NetYCE system maintenance functions - xch_system
- Infoblox IPAM and DHCP provisioning - xch_ib_dhcp
- Infoblox DNS provisioning - xch_ib_dns
- Other modules are customer specific and deal with Maintenance Event suppression, Event Enrichment and CMDB updates.

XCH configuration

Exchange plugins are registered in a configuration file, `/opt/yce/etc/xch_tasks.ini` This file is read by the xch daemon and maps the incoming request task-name to the plugin module and the function name.

```
[system_status]
auth_agent = internal
user_level = 5
task_module = xch_system
task_sub = system_status

[system_fput]
auth_agent = internal
user_level = 5
task_module = xch_system
task_sub = fput

[system_get]
auth_agent = internal
user_level = 5
task_module = xch_system
task_sub = fget
```

In the section above of the ini-file, three different tasks are exposed to the the xch server from the same module: system status, system_fput and system_get. All three are intended for internal use only, which is reflected in the authorization agent that is to be used in these tasks. The plugin module is 'xch_system' from which three different functions (subroutines) are called. The value for user-level refers to the authorization and the minimum user role that is required.

```
[command_job]
auth_agent = yce
```

```
user_level = 2
task_module = xch_jobs
task_sub = command_job

[job_status]
auth_agent = yce
user_level = 3
task_module = xch_jobs
task_sub = job_status
```

In this example the command jobs are made accessible through the API. The submission of a job and the retrieval of the job-status are registered in the plugin module `xch_jobs`, both requiring a level three authorization using the normal `yce` user administration.

XCH Transaction types

Service type execution

Service task

Part of the YCE modeling is defined in Service types. A Service type mirrors in high detail the actions a designer performs when defining how a device must be connected or a service implemented. The process can be visualized as making a drawing of the design where nodes are added, lines are drawn, ports are assigned, vlans created and addresses mapped.

YCE uses Service types to define the standardized actions and have them executed by engineers or operators where the design (as modeled) allows them to do so. In this way a single click can result in an entire device layer be added and properly hooked up to the core devices, including all (management) IP addresses, vlan setup and port configurations of all devices involved.

The XCH Service task request allows remote systems to initiate the execution of a Service type (or service task). An example of such a xml request is shown below. The set of attributes provided is highly customizable. In the case below, the minimal set is used.

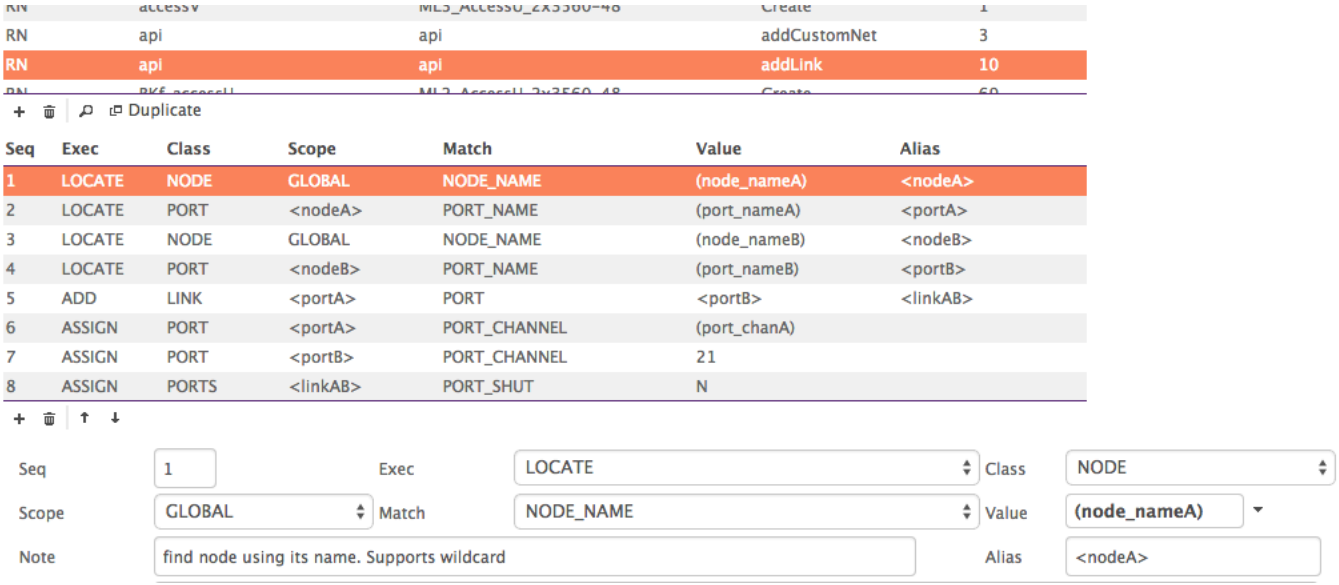
```
<task response="">
    <head
      passwd="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
      task_name="service_type"
      task_type="xml_request"
      userid="loginid"
  />
    <request
      client_type="SAM"
      service_class="Core"
      service_type="SAM_c6509_core"
      service_task="Create"
      client_code="SAM1"
```

```
      site_code="SAM1-BR01"
  />
</task>
```

Custom variables

In extension of the Service type above, custom variables can be inserted in the Service type using the API. The `value` parameter in most Service type records that make up a Service type can be supplied by the API.

The variable names of these custom variables in the API call can either be chosen freely or are pre-defined, depending on how the Service type was defined. If the designer of the service types used brackets, `()`, around the names in the `Value` parameter, that value MUST be provided by the API using that name as the custom name.



The Service type as defined above will need to be called by the API using the request below:

```
<task>
  <head passwd="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
     task_name="service_type" task_type="xml_request" userid="loginid" />
  <request
    client_type="SAM"
    service_class="api"
    service_task="addLink"
    service_type="api"
 >
    <custom name="node_nameA" type="name" value="TE--RN01001" />
    <custom name="node_nameB" type="name" value="TE--RN02002" />
    <custom name="port_nameA" type="name" value="Gi01" />
    <custom name="port_nameB" type="name" value="GI03" />
    <custom name="port_chanA" type="name" value="11" />
    <custom name="port_chanB" type="name" value="22" />
```

```
    <custom name="port_mode" type="name" value="Auto" />
    <custom name="port_shut" type="name" value="N" />
    <custom name="port_speed" type="name" value="Auto" />
  </request>
</task>
```

This method provides exact control over which API variable gets used in each of the Service type records, but will not allow the Service type to be used without the API.

A set of reserved names can be used where such a mix of API and front-end usage of the Service types is required. The service types can be designed including valid values but can be replaced by providing a reserved custom variable if provided by the API call. The variable names are named after their obvious use: `client_code, site_code, node_name, port_name, template, port_channel, port_shut, etc`. Actually, the example above already includes two of these to override the values provided in its last two lines.

Retrieving information

The Service types API also allows to retrieve information from the NetYCE network model. In the Service type, 'Added' objects and 'Located' objects are often manipulated using an 'Alias'. The operator chooses relevant names for these objects to clearly identify them. Each of these aliased NetYCE objects consists of a data-type that has a set of attributes. These Objects including all their attributes can be returned in the API response by setting a flag in the Service type API request.

By setting **`log_aliases`** to `yes` in the API request, all aliases used in the Service type are reported in full. Since many of the Objects include Custom Attributes, these are included in the data-set.

As an example, consider a Service type where a Node is located using Client_code and Site_code. The following XML call will retrieve the Client, Site, and Node data-sets.

```
<task response="">
  <head passwd="xxxxxxxxxxxxxxxx" task_name="service_type"
task_type="xml_request" userid="loginid" />
  <request
    client_type="ServiceProvider"
    service_class="SP_core"
    service_task="serverport"
    service_type="api"
    log_aliases="yes">
    <custom name="Hostname" type="name" value="node12" />
  </request>
</task>
```

```
<task>
    <head abort_on_error="1" error="0" log_level="0" passwd="xxxxxxxxxxxxxx"
req_host="::ffff:192.168.56.1" status="completed" task_id="0716_0021"
task_level="4" task_name="service_type" task_type="xml_request"
userid="loginid">
        <logs/>
    </head>
```

```
    <request
        client_type="ServiceProvider"
        log_aliases="yes"
        request_id="1"
        service_class="SP_core"
        service_task="serverport"
        service_type="api">
        <custom name="Hostname" type="name" value="node12"/>
    </request>
    <response client_type="ServiceProvider" log_aliases="yes"
request_error="0" request_id="1" request_status="completed"
service_class="SP_core" service_task="serverport" service_type="api">
        <alias name="&lt;node&gt;" type="node" value="node12"/>
        <alias name="&lt;port&gt;" type="port" value="33737"/>
        <alias_records>
            <node Boot_loader="" Boot_system="" ClientCode="ServiceProvider"
Console_line="0" DeviceStatus="2" Domain="SP" Enable_secret="cisco"
Hostname="node12" NodeType="0" Node_class="core"
Node_fqdn="node12.tmobile.local" Node_position="NA" Node_type="c3925"
Orig_node="" Par_group="Node" Redundant="1" Rtr_notes="" Service_key="20493"
Sid="36883" SiteCode="core" Template="c3925" Template_rev=""
Terminal_server="" Var_name="" Var_value="" Vendor_type="Cisco_IOS"/>
            <port Bandwidth_down="" Bandwidth_profile="" Bandwidth_up=""
Chan_id="" Hostname="node12" If_name="GigabitEthernet0/0/2"
Interface_id="33737" Port_class="Gi" Port_description="Server template"
Port_id="2" Port_mode="Full" Port_module="" Port_name="Gi00/00/02"
Port_reserve1="" Port_reserve2="" Port_reserve3="" Port_shut="N"
Port_speed="1000" Port_template="server" Port_type="GigabitEthernet" Sid="0"
Slot_id="0/0" Sys_slot="" Timestamp="2018-07-16 16:39:17"/>
        </alias_records>
        <custom name="Hostname" type="name" value="node12"/>
        <log>Pass1: syntax check completed</log>
        <log>Pass2: execution</log>
        <log>execution line 1</log>
        <log>1 parsed: LOCATE - NODE - GLOBAL - NODE_NAME - (Hostname) -
&lt;node&gt;</log>
        <log>1 custom resolve: 'node_name/(hostname)' as 'node12'</log>
        <log>1 exec: LOCATE - NODE - GLOBAL - NODE_NAME - node12 -
&lt;node&gt;</log>
        <log>set alias '&lt;node&gt;' to 'node12' as 'node'</log>
        <log>execution line 2</log>
        <log>2 parsed: LOCATE - PORT - &lt;node&gt; - PORT_TEMPLATE_FIRST -
int_unused - &lt;port&gt;</log>
        <log>2 alias resolve: '&lt;node&gt;' as 'node12'</log>
        <log>2 exec: LOCATE - PORT - node12 - PORT_TEMPLATE_FIRST -
int_unused - &lt;port&gt;</log>
        <log>set alias '&lt;port&gt;' to '33737' as 'port'</log>
        <log>execution line 3</log>
        <log>3 parsed: ASSIGN - PORT - &lt;port&gt; - PORT_TEMPLATE - server
```

```
- </log>
        <log>3 alias resolve: '&lt;port&gt;' as '33737'</log>
        <log>3 exec: ASSIGN - PORT - 33737 - PORT_TEMPLATE - server - </log>
        <log>Port_template must be for 'Cisco_IOS' of client_type
'ServiceProvider'</log>
        <log>execution line 4</log>
        <log>4 parsed: ASSIGN - PORT - &lt;port&gt; - PORT_SPEED - 1000 -
</log>
        <log>4 alias resolve: '&lt;port&gt;' as '33737'</log>
        <log>4 exec: ASSIGN - PORT - 33737 - PORT_SPEED - 1000 - </log>
        <log>execution line 5</log>
        <log>5 parsed: ASSIGN - PORT - &lt;port&gt; - PORT_MODE - Full -
</log>
        <log>5 alias resolve: '&lt;port&gt;' as '33737'</log>
        <log>5 exec: ASSIGN - PORT - 33737 - PORT_MODE - Full - </log>
        <log>execution line 6</log>
        <log>6 parsed: ASSIGN - PORT - &lt;port&gt; - PORT_SHUT - N - </log>
        <log>6 alias resolve: '&lt;port&gt;' as '33737'</log>
        <log>6 exec: ASSIGN - PORT - 33737 - PORT_SHUT - N - </log>
        <log>Done 6/6</log>
    </response>
</task>
```

Job execution

Command job

A command job is a generic tool to execute changes in the network. These changes are prepared in YCE using either the client(s) or remotely using the XCH Service task method. Once the modeled network has the desired change(s) incorporated, the operators prepare the jobs required to provision the network using the appropriate tasks and scenarios.

A range of tools is available for the operator to create these jobs. The Command job is the most versatile of these. The XCH Command_job task is its equivalent for remote use.

Standardized changes are available as “Stored jobs”, requiring only the device (node) selection and the stored_job name. For non-standard jobs, the complete set of commands (or template names) can be specified in the job request. The same is true for the desired scenario: it is either defined in the stored job or can be defined in the job request (step by step or by task name).

The job request example below uses the full version where all available options are defined. Note that here a stored_job_name is still defined although both command section and scenario sections are provided. In these cases the stored job functions as the default should one of these sections be left blank.

Because the commands and scenario sections support the full set of template and scenario syntaxes for parameter substitution and conditionals, a potential conflict in XML and YCE syntax arises. Encapsulating the actual scenario and commands in a CDATA construct circumvents this.

```
<task response="">
    <head
      passwd="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
      req_app="my_command_job"
      req_host="genie"
      task_name="command_job"
      task_type="xml_request"
      userid="loginid"
  />
    <request
      node_name="TESTRN01001"
      client_type=""
      site_code=""
      client_code="1006"
      stored_job_name="Default command job"
      sched_day="tomorrow"
      sched_time="5:05"
      sched_now="no"
      sched_queue="yce"
      verbose_log="yes"
                >
        <commands>
<![CDATA[
! Change enable from '<Enable_secret>' to '<Default_enable_secret>'
enable secret <Default_enable_secret>
!
]]>
        </commands>
        <scenario>
<![CDATA[
Description <node> Enable secret update

Import_cfg -q -n <node> -f <node>.cmd <verbose>
if Error
    LogAction -n <node> -a Command_job -m 'Job failed updating enable
secret'
    stop
endif

Db_update -t SiteRouter -f Enable_secret -v '<Default_enable_secret>' -w
Hostname='<node>'

Logaction -n <node> -a Enable_secret -m '<Enable_secret> =>
<Default_enable_secret>'
LogAction -n <node> -a Command_job -m 'Job completed updating enable secret'
]]>
        </scenario>
        </request>
</task>
```

The job request above is listing the full version, not using any defaults. In the request below, most of the defaults are used, only the host_name and the commands are specified. In this case the scenario used is the “Default command job”. In this case it is demonstrated that the use of the <![CDATA[…]]> encapsulation can be avoided by converting the ‘<’ and ‘>’ to `<` en `>` respectively.

The default schedule time is ‘tomorrow 05:05’. Other defaults are ‘verbose_log=”yes”’ and ‘sched_now=”no”.

```
<task response="">
    <head
      passwd="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
      task_name="command_job"
      task_type="xml_request"
      userid="loginid"
  />
    <request
      node_name="TESTRN01001"
>
        <commands>
!
my hostname is <hostname>
!
|hostname = 'testrn01001'|yes i'm <hostname>
!
        </commands>
        </request>
</task>
```

Sample response (full):

```
<task>
  <head
    error="0000"
    passwd="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    sched_status="client John Doe (operator) registered"
    status="completed"
    task_name="command_job"
    task_type="xml_request"
    user_func="Operator"
    user_level="6"
    user_name="John Doe"
    userid="loginid"
  />
  <request
    auth_agent="yce"
    client_level="5"
    commands="
        ! my hostname is <hostname>
        !
|hostname = 'testrn01001'|yes i'm <hostname>
        !"
```

```
    group_id="NetYCE"
    node_name="TESTRN01001"
    operator="myuserid"
    sched_day="tomorrow"
    sched_time="05:05"
    stored_job_description="Issue parameterized commands to the selected
nodes"
    stored_job_name="Default command job"
    task_module="xch_jobs.pl"
    task_sub="command_job"
    user_level="3"
    verbose_log="yes"
  />
  <response
    client_code="1006"
    client_type="RN"
    commands="
        ! my hostname is <hostname>
        !
|hostname = 'testrn01001'|yes i'm <hostname>
        !"
    job_descr=" ..."
    jobid="0913_0002"
    node_fqdn="testrn01001.netyce.net"
    node_name="TESTRN01001"
    node_type="RN+_3560G-48_DCoreHK"
    scenario="
        Description <node>
        Command_job... task = Command_job "
    sched_job="Sat 14-Sep-2013 05:05:00"
    sched_queue="yce"
    sched_req="tomorrow 05:05"
    site_code="TESTRN01"
    vendor_type="Cisco_IOS"
    verbose_log="-v"
  />
</task>
```

Parameters for a stored_job_name

When using a stored_job additional parameters may be provided. These will be treated as if they were parameters provided in the stored job '[parameter]' section.

> NOTE: They will not override existing set values!

```
<task>
  <head abort_on_error="1"
  passwd="password"
```

```
  req_app="/opt/yce/operate/command_job.pl"
  req_host="server"
  request_id="1"
  task_name="command_job"
  task_type="xml_request"
  userid="username"
  usr_type="local"
  xml_decode="yes" />

  <request
  change_id=""
  client_type=""
  commands=""
  description=""
  evaluate="no"
  node_name="your_node"
  sched_day="tomorrow"
  sched_epoch=""
  sched_now="yes"
  sched_queue="yce"
  sched_server="server"
  sched_time="5:05"
  stored_job_name="your_stored_job"
  verbose_log="yes"
  scenario="">
        <parameters parameter1="value1"
          parameter2="1100"
          some_name="2200"
           />
    <xml_decode>scenario</xml_decode>
  </request>
</task>
```

Basic Command job

The basic command job API call is exactly the same as the Command_job except for the task_name.

task_name is set to 'basic_command_job'

A basic command job can point to both CMDB nodes (the default) or YCE nodes.

Job status

The results of any job can be retrieved using it's jobID. While the job is in RUNNING state, the details will keep pace with its progress. The job results can be retrieved from any YCE server once it has become active.

```
<task response="">
    <head
```

```
      passwd="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
      task_name="job_status"
      task_type="xml_request"
      userid="loginid"
  />
    <request
      jobid="0912_0022"
        />
</task>
```

Sample response (full):

```
<task>
  <head
    error="0000"
    passwd="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    status="completed"
    task_name="job_status"
    task_type="xml_request"
    userid="loginid"
  />
  <request
    auth_agent="yce"
    jobid="0912_0022"
    status_timestamp="2013-09-13 14:32:22"
    task_module="xch_jobs.pl"
    task_sub="job_status"
    user_level="3"
    xch_server="genie"
  />
  <response
    job_state="ABORTED"
    jobid="0912_0022"
    log_details="0912_0022 2013-09-12 16:20:01 <b>Command_job on
TESTRN01001</b>
      Tasks: Command_job 03-Import_cfg (-q -n TESTRN01001 –f TESTRN01001.cmd
-v)
      00-ARGUMENTS
      Command: import
      Starting import on TESTRN01001
      Session stopped
      Node TESTRN01001 is unreachable at 10.10.62.192.
      Aborted 2013-09-12 16:20:09 10.34.62.192 finished with Errors
      ERROR import_cfg failed: Node TESTRN01001 is unreachable at
10.10.62.192.
      Aborted 05-Logaction (-n TESTRN01001 -a Command_job -m "
      Failed executing commands")
      06-Stop ()
      2013-09-12 16:20:09 ABORTED after 8 seconds "
```

```
      log_head="0912_0022 2013-09-12 16:20:01 TESTRN01001 Command_job..."
      log_info="TESTRN01001 Command_job..."
      log_server="genie"
      log_tail="0912_0022 2013-09-12 16:20:09 ABORTED after 8 seconds "
      log_timestamp="2013-09-12 16:20:09"
      operator="NetYCE support"
      />
</task>
```

If the job exists, but waits execution by the scheduler, there are no job results as yet, but schedule information can be retrieved if the XCH request was directed at the server where the job was scheduled.

The results for a job scheduled, but not yet active:

```
<task>
  <head
    error="0000"
    passwd="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    status="completed"
    task_name="job_status"
    task_type="xml_request"
    userid="loginid"
  />
  <request
    auth_agent="yce"
    jobid="0913_0002"
    task_module="xch_jobs.pl"
    task_sub="job_status"
    user_level="3"
  />
  <response
    job_status="SCHEDULED"
    jobid="0913_0002"
    log_info="<span class=node>TESTRN01001</span> Command job ..."
    operator="John Doe"
    sched_job="Sat 14-Sep-2013 05:05:00"
    sched_queue="yce"
    sched_start=""
  />
</task>
```

Reports

Fetch report

Previously generated custom reports can be retrieved using the API. The filename or query name is the only required attribute to fetch the CSV report and have it converted to XML.

using the URL

All reports are created in a CSV format and are converted to html (when viewing) or XML (for the API) when needed. If the original CSV is required, the download link is included when viewing the report. Generated reports can be downloaded as a CSV file directly using the URL below. Note that the file path is case sensitive but the report-name is not. To download the file using Dos formatting append `&type=dos` to the url.

Created reports are deleted automatically after 30 days, or the period in days defined by the Lookup tweak 'Age_custom_reports'.

```
https://<netyce.server>/report/<report-name>
```

using XCH API

To retrieve the CSV report in XML format using the API, the **`fetch_report`** request can be used. The **`report_name`** attribute of the `fetch_report` request can contain:

- the case-insensitive custom report name
- the path and filename of the custom report csv file

The latter format ('report_name="/var/opt/yce/output/my-report-name.csv"') is supported for historic reasons only. When using this format, the report-name is extracted from the argument and its results retrieved from the database, if located.

If the report settings indicate it may not be overwritten, the resulting report-name has the date appended (format: '<myreport>-yyyymmdd'). The `report_name` should then also include this date for the desired report. If the report-name has no date appended, the latest generated (dated) report will be returned.

Sample request:

```
<task response="">
    <head
        userid="my-login-id"
        passwd="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
        task_type="xml_request"
        task_name="fetch_report"
    />
    <request
        report_name="supernet_usage"
    />
</task>
```

The request resulted in the response below. Note that each row in the report is represented as a hash keyed with `row_nnn` where the nnn is the row number. The row number is padded with an appropriate number of zeros to allow alphanumeric sorting.

Included are the timestamp of the generated report, as is the number of rows. The report columns are listed in their original (sql) order under the key `cols`.

Column names starting with a digit that do not comply with the XML tag-formats will be automatically protected by prepending the string _x_ (e.g. column '1st' will be named '_x_1st' because it starts with a number).

```
<task>
  <head error="0000" passwd="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
status="completed" task_id="0121_0006" task_name="fetch_report"
task_type="xml_request" userid="my-login-id">
    <logs>
      <log>Read and converted csv report 'Supernet_usage' on 'yceone'</log>
    </logs>
  </head>
  <request auth_agent="yce" report_name="supernet usage"
task_module="xch_system.pl" task_sub="fetch_csv" user_level="2" />
  <response file_date="2019-01-10 04:05:02" report_name="supernet_usage"
file_name="Supernet_usage-20190110" query_name="Supernet_usage" rows="12">
    <cols>Ip_supernet</cols>
    <cols>Net_name</cols>
    <cols>Net_index</cols>
    <cols>ClientCode</cols>
    <cols>SiteCode</cols>
    <cols>Net_address</cols>
    <cols>Net_size</cols>
    <cols>Service_type</cols>
    <row_01 ClientCode="1476" Ip_supernet="10.21.0.0"
Net_address="10.21.63.192" Net_index="0" Net_name="Inrol" Net_size="30"
Service_type="ML31_Core_5800-24" SiteCode="BEEKRN02" />
    <row_02 ClientCode="1476" Ip_supernet="10.21.0.0"
Net_address="10.21.63.204" Net_index="3" Net_name="Inrol" Net_size="30"
Service_type="ML31_Core_2x5800-24" SiteCode="STD-RN01" />
    <row_03 ClientCode="1476" Ip_supernet="10.21.0.0"
Net_address="10.21.63.208" Net_index="4" Net_name="Inrol" Net_size="30"
Service_type="ML31_Core_5800-24" SiteCode="BORNRN01" />
    <row_04 ClientCode="1476" Ip_supernet="10.21.0.0"
Net_address="10.21.63.212" Net_index="5" Net_name="Inrol" Net_size="30"
Service_type="ML31_Core_5800-24" SiteCode="STD-RN03" />
    <row_05 ClientCode="1476" Ip_supernet="10.21.0.0"
Net_address="10.21.9.128" Net_index="3" Net_name="IPT" Net_size="25"
Service_type="ML31_Core_5800-24" SiteCode="BORNRN01" />
    <row_06 ClientCode="1476" Ip_supernet="10.21.0.0"
Net_address="10.21.48.0" Net_index="0" Net_name="IPT_srv" Net_size="27"
Service_type="ML31_Core_5800-24" SiteCode="STD-RN03" />
    <row_07 ClientCode="1476" Ip_supernet="10.21.0.0"
Net_address="10.21.48.128" Net_index="4" Net_name="IPT_srv" Net_size="27"
Service_type="ML31_Core_5800-24" SiteCode="BORNRN01" />
    <row_08 ClientCode="1476" Ip_supernet="10.21.0.0"
Net_address="10.21.63.12" Net_index="3" Net_name="p2p" Net_size="30"
Service_type="ML31_Core_DBC" SiteCode="BORNRN01" />
```

```
    <row_09 ClientCode="1476" Ip_supernet="10.21.0.0"
Net_address="10.21.47.0" Net_index="0" Net_name="Servers_bk" Net_size="28"
Service_type="ML31_Core_5800-24" SiteCode="BEEKRN02" />
    <row_10 ClientCode="1476" Ip_supernet="10.21.0.0"
Net_address="10.21.47.64" Net_index="4" Net_name="Servers_bk" Net_size="28"
Service_type="ML31_Core_5800-24" SiteCode="BORNRN01" />
    <row_11 ClientCode="1476" Ip_supernet="10.21.0.0"
Net_address="10.21.47.80" Net_index="5" Net_name="Servers_bk" Net_size="28"
Service_type="ML31_Core_5800-24" SiteCode="STD-RN03" />
    <row_12 ClientCode="1476" Ip_supernet="10.21.0.0"
Net_address="10.21.4.0" Net_index="8" Net_name="Users" Net_size="25"
Service_type="ML31_Core_5800-24" SiteCode="BORNRN01" />
  </response>
</task>
```

Run report

Similar to `fetch_report`, the **`run_report`** XCH request runs the custom report before downloading it in XML format. The request is identical, save for the `task_name` attribute which must read `run_report`. The request section has only one attribute, **`report_name`**, that holds the name of an existing custom report.

Since custom reports can be defined not to overwrite any results from previous days, the resulting csv report will have the date appended to the report name ('<report_name>-YYYYMMDD.csv'). When executing and fetching these reports using this call, a possibly existing file will be overwritten with todays date.

The response message will be identical to the `fetch_report` call.

Sample request:

```
<task response="">
    <head
        userid="my-login-id"
        passwd="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
        task_type="xml_request"
        task_name="run_report"
        log_level="1"
    />
    <request
        report_name="supernet_usage"
    />
</task>
```

Infoblox DNS

IPAM and DNS report

» This section has been replaced by the article on the Infoblox DNS API plugin.

Infoblox DNS registration

» his section will shortly be replaced by the article on the Infoblox DNS API plugin.

Add Host

The add_host request finds and allocates an IP-address for a new host name in a pre-existing zone. A free IP-address is located in the included set of IPAM subnets where ‘free’ means no that DNS entry exists, nor is part of DHCP range. A new ‘Host’-type DNS record is created by default, or an A-record if specified. When aliases are specified, those are added to the host record or created as Cname-records as is appropriate.

```
<task response="">
  <head
        passwd="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
        task_name="infoblox_dns"
        task_type="xml_request"
        userid="myuserid"
  />
  <request action_type="Add_host">
    <host
        comment="RFC 1234"
        host_domain="existing.zone.name"
        host_name="my-new-host"
        record_type="host"
        request_id="101"
     >
       <subnet_addr>10.16.238.0/25</subnet_addr>
       <subnet_addr>10.16.239.0/28</subnet_addr>
       <alias>new-host-alias1.existing.zone.name</alias>
       <alias>new-host-alias2.another.zone.name</alias>
    </host>
  </request>
</task
```

The changes are directly made to the life DNS GridMaster. The allocated ip-address and the registered DNS entry is returned. The task rejects non-existing zones and applies restrictions on the hostnames; e.g. no dotted hosts, hosts starting with a numeric digit or the use of underscores.

In the request, the host name and zone can be provided as two attributes ‘host_name’ and

'host_domain', but also combined as a single attribute 'host_fqdn'. When borh are provided, the 'host_fqdn' takes precedence.

Multiple host requests may be included in the task. Each is expected to have a unique request_id (within the task). These hosts are processed in sequence before the task responds.

Add alias

The add alias request updates an existing host record to include the aliases listed in the request. Existing or overlapping aliases are ignored. The response lists the resulting set of aliases. When no aliases are provided in the request, the existing set of aliases for this host are listed.

```
<task response="">
  <head
        passwd="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
        task_name="infoblox_dns"
        task_type="xml_request"
        userid="myuserid"
  />
  <request action_type="Add_alias">
    <host
        comment="Dummy RFC 1234"
        host_domain="a1006.some.zone"
        host_name="te--rn01003"
        request_id="101"
    >
      <alias>te--rn01003b.some.other.zone</alias>
      <alias>te--rn01003c.some.zone</alias>
    </host>
  </request>
</task>
```

Clear alias

The clear_alias request updates an existing host record to remove the aliases listed in the request. Existing aliases named in the request are removed, others ignored. The response lists the resulting set of aliases. When no aliases are provided in the request, the existing set of aliases for this host are listed.

```
<task response="">
  <head
        passwd="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
        task_name="infoblox_dns"
        task_type="xml_request"
        userid="myuserid"
  />
  <request action_type="Clear_alias">
```

```
    <host
        comment="Dummy RFC 1234"
        host_domain="a1006.some.zone"
        host_name="te--rn01003"
        request_id="101"
      >
      <alias>te--rn01003b.some.other.zone</alias>
      <alias>te--rn01003c.some.zone</alias>
    </host>
  </request>
</task>
```

Clear host

The clear host request removes the host record including all its ip-addresses and aliases if it is a host-record. When the DNS name belongs to an A-record or Cname, the appropriate record is removed from the DNS.

```
<task>
  <head
        passwd="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
        task_name="infoblox_dns"
        task_type="xml_request"
        userid="myuserid"
  />
  <request action_type="Clear_host">
    <host
        comment="Dummy RFC 1234"
        host_domain="a1006.some.zone"
        host_name="te--rn01003"
        request_id="101"
      >
    </host>
  </request>
</task>
```

Infoblox IPAM and DHCP

Client IPAM tree and DHCP configuration

The IPAM and DHCP report is used to feed an IPAM and/or DHCP configuration tool. YCE includes such a tool for Infoblox where this report is used internally, but is also used externally.

The report requires the name (ClientCode) of an YCE-client that is fully modeled and uses the YCE ip-plan(s). Combined with the information found in an IPAM definition table within YCE, a report is generated where both the IPAM subnet tree and the associated DHCP scopes are fully defined. The DHCP definition includes the (customer defined) options and their calculated values.

The intended use for the report is to automate the (Infoblox) IPAM subtree's en DHCP scope provisioning. When, for example, an operator adds a new location or some devices requiring ip-subnets, these are automatically assigned using the YCE ip-plans for this customer and used in the respective configurations. Next, the operator initiates the IPAM/DHCP update function for this customer which results in having the assigned subnets added to the IPAM tree, but also activating the required DHCP scopes including all their options. Similarly, when removing or freeing a subnet, the same process removes both DHCP definitions and returns the subnet to the 'free' pool.

IPAM/DHCP tree

This report can also be extended to include the IPAM trees of all clients in a client type.

```
<task>
  <head
  passwd="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
  task_name="infoblox_dhcp"
  task_type="xml_request"
  userid="loginid"
  />
  <request
  client="1005"
  client_type=""
  />
</task>
```

Sample report - below a single subnet record out of tens of thousands:

```
<tree
  ddns="no"
  line_number="110"
  net_address="10.25.236.0"
  net_mask="255.255.255.224"
  net_name="Wifi_ap 0"
  net_options="1,15,43,44,46,60,241"
  net_size="27"
  net_tier="2"
  net_type="network"
  site_type=""
>
  <net_members>10.10.254.26</net_members>
  <net_members>10.10.254.58</net_members>
  <option
    option_name="subnet-mask"
    option_number="1" option_val="255.255.255.224"
  />
  <option
    option_name="domain-name"
    option_number="15"
```

```
    option_val="netyce.net"
  />
  <option
    option_name="vendor-encapsulated-options"
    option_number="43"
    option_val="F1:04:0A:0C:10:3C"
  />
  <option
    option_name="netbios-name-servers"
    option_number="44"
  >
    <option_val>10.233.18.77</option_val>
  </option>
  <option
    option_name="netbios-node-type"
    option_number="46"
    option_val="2"
  />
  <option
    option_name="vendor-class-identifier"
    option_number="60"
    option_val="Cisco AP c1140"
  />
  <option
    option_name="WLC-Servers"
    option_number="241"
    option_space="WiFi"
  >
    <option_val>10.12.16.60</option_val>
  </option>
</tree>
```