Apertis integration testing with LAVA

Contents

LAVA[1] is a testing system allowing the deployment of operating systems to physical and virtual devices, sharing access to devices between developers. As a rule tests are started in non-interactive unattended mode and LAVA provides logs and results in a human-readable form for analysis.

As a common part of the development cycle we need to do some integration testing of the application and validate it's behavior on different hardware and software platforms. LAVA provides the ability for Apertis to share a pool of test devices, ensuring good utilization of these resources in addition to providing automated testing.

Integration testing example

Let's take the `systemd` service and `systemctl` CLI tool as an example to illustrate how to test an application with a D-Bus interface.

The goal could be defined as follows:

> As a developer of the `systemctl` CLI tool, I want to ensure that `systemctl` is able to provide correct information about the system state.

Local testing

To simplify the guide we are testing only the status of `systemd` with the command below:

```
$ systemctl is-system-running
running
```

It doesn't matter if `systemctl` is reporting some other status, `degraded` for instance. The goal is to validate if `systemctl` is able to provide a proper status, rather than to check the `systemd` status itself.

To ensure that the `systemctl` tool is providing the correct information we may check the system state additionally via the `systemd` D-Bus interface:

[1] https://www.lavasoftware.org/

```
$ gdbus call --system --dest=org.freedesktop.systemd1 --object-path "/org/freedesktop/systemd1" -
-method org.freedesktop.DBus.Properties.Get org.freedesktop.systemd1.Manager SystemState
(<'running'>,)
```

So, for local testing during development we are able to create a simple script validating that `systemctl` works well in our development environment:

```
#!/bin/sh

status=$(systemctl is-system-running)

gdbus call --system --dest=org.freedesktop.systemd1 \
  --object-path "/org/freedesktop/systemd1" \
 --method org.freedesktop.DBus.Properties.Get org.freedesktop.systemd1.Manager SystemState | \
  grep "${status}"

if [ $? -eq 0 ]; then
  echo "systemctl is working"
else
  echo "systemctl is not working"
fi
```

Testing in LAVA

As soon as we are done with development, we push all changes to GitLab and CI will prepare a new version of the package and OS images. But we do not know if the updated version of `systemctl` is working well for all supported devices and OS variants, so we want to have the integration test to be run by LAVA.

Since the LAVA is a part of CI and works in non-interactive unattended mode we can't use the test script above as is.

To start the test with LAVA automation we need to:

1. Adopt the script for LAVA
2. Integrate the testing script into Apertis LAVA CI

Changes in testing script

The script above is not suitable for unattended testing in LAVA due some issues:

- LAVA relies on exit code to determine if test a passed or not. The example above always return the `success` code, only a human-readable string printed by the script provides an indication of the status of the test
- if `systemctl is-system-running` call fails for some other reason (with a segfault for instance), the script will proceed further without that error being detected and LAVA will set the test as passed, so we will have a false positive result

- LAVA is able to report separately for any part of the test suite –just need to use LAVA-friendly output pattern

So, more sophisticated script suitable both for local and unattended testing in LAVA could be the following:

```
#!/bin/sh

# Test if systemctl is not crashed
testname="test-systemctl-crash"
status=$(systemctl is-system-running)
if [ $? -le 4 ]; then
  echo "${testname}: pass"
else
  echo "${testname}: fail"
  exit 1
fi

# Test if systemctl return non-empty string
testname="test-systemctl-value"
if [ -n "$status" ]; then
  echo "${testname}: pass"
else
  echo "${testname}: fail"
  exit 1
fi

# Test if systemctl is reporting the same status as
# systemd exposing via D-Bus
testname="test-systemctl-dbus-status"
gdbus call --system --dest=org.freedesktop.systemd1 \
  --object-path "/org/freedesktop/systemd1" \
  --method org.freedesktop.DBus.Properties.Get \
  org.freedesktop.systemd1.Manager SystemState | \
  grep "${status}"
if [ $? -eq 0 ]; then
  echo "${testname}: pass"
else
  echo "${testname}: fail"
  exit 1
fi
```

Now the script is ready for adding into LAVA testing. Pay attention to output format which will be used by LAVA to detect separate tests from our single script. The exit code from the testing script must be non-zero to indicate the test suite failure.

Create GIT repository for the test suite

We assume the developer is already familiar with GIT version control system[2] and has an account for the Apertis GitLab[3] as described in the Development Process guide[4]

The test script must be accessible by LAVA for downloading. LAVA has support for several methods for downloading but for Apertis the GIT fetch is preferable since we are using separate versions of test scripts for each release.

It is strongly recommended to create a separate repository with test scripts and tools for each single test suite.

As a first step we need a fresh and empty GIT repository somewhere (for example in your personal space of the GitLab instance) which needs to be cloned locally:

```
git clone git@gitlab.apertis.org:d4s/test-systemctl.git
cd test-systemctl
```

By default the branch name is set to `main` but Apertis automation require to use the branch name aimed at a selected release (for instance `apertis/v2022dev1`), so need to create it:

```
git checkout HEAD -b apertis/v2022dev1
```

Copy your script into GIT repository, commit and push it into GitLab:

```
chmod a+x test-systemctl.sh
git add test-systemctl.sh
git commit -s -m "Add test script" test-systemctl.sh
git push -u origin apertis/v2022dev1
```

Add the test into Apertis LAVA CI

Apertis test automation could be found in the GIT repository for Apertis test cases[5], so we need to fetch a local copy and create a work branch `wip/example` for our changes:

```
git clone git@gitlab.apertis.org:tests/apertis-test-cases.git
cd apertis-test-cases
git checkout HEAD -b wip/example
```

1. Create test case description.

   First of all we need to create the instruction for LAVA with following information:

   - where to get the test

[2] https://www.apertis.org/guides/version_control/
[3] https://gitlab.apertis.org/
[4] https://www.apertis.org/guides/development_process/
[5] https://gitlab.apertis.org/tests/apertis-test-cases

- how to run the test

Create the test case file `test-cases/test-systemctl.yaml` with your favorite editor:

```
metadata:
  name: test-systemctl
  format: "Apertis Test Definition 1.0"
  image-types:
    fixedfunction:  [ armhf, arm64, amd64 ]
  image-deployment:
    - OSTree
  group: systemctl
  type: functional
  exec-type: automated
  priority: medium
  maintainer: "Apertis Project"
  description: "Test the systemctl."

  expected:
    - "The output should show pass."

install:
  git-repos:
    - url: https://gitlab.apertis.org/d4s/test-systemctl.git
      branch: apertis/v2022dev1

run:
  steps:
    - "# Enter test directory:"
    - cd test-systemctl
    - "# Execute the following command:"
    - lava-test-case test-systemctl --shell ./test-systemctl.sh

parse:
  pattern: "(?P<test_case_id>.*):\\s+(?P<result>(pass|fail))"
```

This test is aimed to be run for an ostree-based fixedfunction Apertis image for all supported architectures. However the metadata is mostly needed for documentation purposes.

The `group` field is used to group test cases into the same LAVA job description. See the job templates below.

Action “install”points to the GIT repository as a source for the test, so LAVA will fetch and deploy this repository for us.

Action “run”provides the step-by-step instructions on how to execute the test. Please note that it is recommended to use wrapper for the test for integration with LAVA.

Action “parse”provides its own detection for the status of test results printed by script.

2. Push the test case to the GIT repository.

   This step is mandatory since the test case would be checked out by LAVA internally during the test preparation.

```
git add test-cases/test-systemctl.yaml
git commit -s -m "add test case for systemctl" test-cases/test-systemctl.yaml
git push --set-upstream origin wip/example
```

3. If needed, add a job template to be run in lava. Job templates contain all needed information for LAVA to boot the target device and deploy the OS image onto it.

   Job template files must be named `lava/group-[GROUP]-tpl.yaml`.

   e.g.: Create the simple template `lava/group-systemctl-tpl.yaml` with your lovely editor:

```
 job_name: systemctl test on {{release_version}} {{pretty}} {{image_date}}
 {% if device_type == 'qemu' %}
 {% include 'common-qemu-boot-tpl.yaml' %}
 {% else %}
 {% include 'common-boot-tpl.yaml' %}
 {% endif %}
   - test:
       timeout:
         minutes: 15
       namespace: system
       name: {{group}}-tests
       definitions:
 {%- for test_name in tests %}
           - repository: https://gitlab.apertis.org/tests/apertis-test-cases.git
             branch: 'wip/example'
             from: git
             name: {{test_name}}
             path: test-cases/{{test_name}}.yaml
 {%- endfor -%}
```

   If no template exists for a given group, the default template (`lava/group-default-tpl.yaml`) will be used, still creating a different job per group. It looks a lot like the example template above. This is useful if you do not need any specific variables set or special boot steps.

Hopefully you don't need to deal with the HW-related part, boot and deploy since we already have those instructions for all supported boards and Apertis OS images. See common boot template[6] for instance.

Please pay attention to `branch` –it must point to your development branch while you are working on your test.

It is highly recommended to use a temporary group specific to the test you are working on to avoid unnecessary workload on LAVA while you're developing the test.

4. Generate the job descriptions.

   Since LAVA is a part of Apertis OS CI, it requires some variables to be provided for using Apertis templates. Let's define the board we will use for testing, as well as the image release and variant:

```
release=v2023dev1
version=v2023dev1.0rc2
variant=fixedfunction
arch=armhf
board=uboot
baseurl="https://images.apertis.org"
imgpath="release/$release"
image_name=apertis_ostree_${release}-${variant}-${arch}-${board}_${version}
```

   To generate the test job description, `generate-jobs.py` is used:

```
./generate-jobs.py -d lava/devices.yaml --config lava/config.yaml
   --release ${release} --arch ${arch} --board ${board} --osname apertis
    --deployment ostree --type ${variant} --date ${version}
    --name ${image_name}
    -t visibility:"{'group': ['Apertis']}" -t priority:"medium"
```

   It will generate one job description file for each group that is found compatible with those parameters.

   `generate-jobs.py` can be found here[7]

   There should not be any error or warning. You may want to add the `-v` argument to see the generated LAVA job.

   If the test definition is on an external git repository, you can specify the folder to load the test cases from with `--tests-dir` or, for debugging one specific test case, specify it with `--test-case`.

[6] https://gitlab.apertis.org/tests/apertis-test-cases/-/blob/apertis/v2022/lava/common-boot-tpl.yaml

[7] https://gitlab.apertis.org/tests/apertis-test-cases/-/blob/apertis/v2023dev1/generate-jobs.py

It is recommended to set `visibility` variable to "Apertis"group during development to avoid any credentials/passwords leak by occasion. Setting the additional variable `priority` to `high` allows you to bypass the jobs common queue if you do not want to wait for your job results for ages.

The `generate-jobs.py` tool generates the test job from local files, so you don't need to push your changes to GIT until your test job is working as designed.

5. Configure and test the `lqa` tool.

   For interaction with LAVA you need to have the `lqa` tool installed and configured as described in LQA[8] tutorial.

   The tool is pretty easy to install in the Apertis SDK:

   ```
   $ sudo apt-get update
   $ sudo apt-get install -y lqa
   ```

   To configure the tool you need to create file `~/.config/lqa.yaml` with the following authentication information:

   ```
   user: '<REPLACE_THIS_WITH_YOUR_LAVA_USERNAME>'
   auth-token: '<REPLACE_THIS_WITH_YOUR_AUTH_TOKEN>'
   server: 'https://lava.collabora.co.uk/'
   ```

   where `user` is your login name for LAVA and `auth-token` must be obtained from LAVA API: https://lava.collabora.co.uk/api/tokens/

   To test the setup just run command below, if the configuration is correct you should see your LAVA login name:

   ```
   $ lqa whoami
   d4s
   ```

6. Submit your first job to LAVA.

   Jobs can be submitted with `lava-submit.py`. It can be found here[9].

   You can select the job files you want to send, here it will be the one for our new test group `systemctl`:

   ```
   job-apertis_ostree_v2023dev1-fixedfunction-armhf-uboot_v2023dev1.0rc2-
   systemctl.yaml
   ```

   and can be sent with:

   ```
   $ ./lava-submit.py -c ~/.config/lqa.yaml submit
      job-apertis_ostree_v2023dev1-fixedfunction-armhf-uboot_v2023dev1.0rc2-
   systemctl.yaml
   ```

[8] https://www.apertis.org/qa/lqa/

[9] https://gitlab.apertis.org/tests/apertis-test-cases/-/blob/apertis/v2023dev1/lava-submit.py

```
Submitted job job-apertis_ostree_v2023dev1-fixedfunction-armhf-
uboot_v2023dev1.0rc2-systemctl.yaml with id 3463731
```

It is possible to check the job status by URL with the ID returned by the above command: https://lava.collabora.co.uk/scheduler/job/3463731

The `lava-submit.py` tool is currently only a wrapper around the `lqa` tool. It is also capable to communicate the tested image to the QA Report App[10].

7. Push your template changes.

   Once your test case works as expected you should make sure it is in the right group, change the `branch` key in file `lava/group-systemctl-tpl.yaml` to a suitable target branch and submit your changes:

   ```
   git add lava/group-systemctl-tpl.yaml
   git commit -a -m "hello world template added"
   git push
   ```

As a last step you need to create a merge request in GitLab. As soon as it gets accepted your test becomes part of Apertis testing CI.

Details on test job templates

The boot process for non-emulated devices and for QEMU differs, and due to the amount of differences the definitions are split into two separate template files.

`common-boot-tpl.yaml` contains definition needed to boot Apertis images on real (non-emulated devices). Since they cannot boot images directly, the boot process is separated in two stages: flashing the image onto a device from which the board can boot, and booting into the image and running tests.

The first stage boots over NFS into a (currently) Debian stretch image with a few extra tools needed to flash the image, downloads the image using HTTP, flashes it and reboots. This stage is defined using `namespace: flash` in the job YAML file. In most cases you won't need to edit bits related to this stage. The second stage is common for both non-emulated devices and QEMU, despite them having certain differences. It is used to boot the image itself, prepare the LAVA test runner and run tests. This stage is defined using `namespace: system`. You *normally* don't need to edit this stage either. The exception to this is when you need to load an image from a different source than `images.apertis.org`.

Image URLs are defined in the `deploy` action. For `common-boot-tpl.yaml`, it is necessary to specify URLs to both image itself and its *bmap* file, which is used to speed up the flashing process and avoid unnecessary excessive device wear. For `common-qemu-boot-tpl.yaml`, only the URL to the image itself is needed, as QEMU doesn't support *bmap* files yet.

[10]https://qa.apertis.org/

The second stage always performs two tests: `sanity-check`, which basically checks that the system actually works, and `add-repo`, which isn't actually a test, and is used to add repositories to `/etc/apt/sources.list` on certain devices.

Using short-lived CI tokens

Gitlab provides a short-lived token called `CI_JOB_TOKEN` which can be used to give access to the contents of internal and private repositories during CI runs. From `apertis/v2023dev3` we can make use of this token, using a different approach to job submission to the one described in the previous sections. That is, so far in this document, we've run `lava-submit.py` to batch upload the jobs generated by `generate-jobs.py` to LAVA. If we do the same thing in our CI pipeline, then the CI job will terminate shortly after the jobs are uploaded, invalidating our job token.

Do not expose `CI_JOB_TOKEN` to the wider public by submitting publicly visible jobs. You should submit jobs with tokens in them as `private`. You should also reduce the privileges of job tokens[11] when using `CI_JOB_TOKEN` in LAVA jobs.

For this reason, instead of using `lava-submit.py`, we use a different tool, `generate-test-pipeline.py`, from the same repository when running CI tests. This makes a dynamic Gitlab pipeline to run the generated jobs. Each LAVA job will have its own Gitlab job to track it, and that means there is a short-lived token available that will remain valid for as long as the LAVA job runs. `generate-test-pipeline.py` can be found here[12].

There are two different places you might want to use such tokens with LAVA, and they require slightly different approaches. To use a short-lived token to gain access to a repository from a LAVA job description, for example to obtain test files from a private repository, the repository URL needs to be altered to show where to substitute the token. For example:

```
  https://gitlab-ci-token:{{ '{{job.CI_JOB_TOKEN}}' }}@gitlab.apertis.org/tests/apertis-
test-cases.git
```

The odd appearance is because two rounds of templating are occurring: we escape the template for the job token so that `generate-jobs.py` will preserve it. When our dynamic pipeline runs, the LAVA runner will substitute its own value for `CI_JOB_TOKEN`.

To use a short-lived token from within a test-case, we need to do two things. First, we need to add a parameter to the test's group template with the full URL for the repository we wish to include. The group templates form part of the job definition, and so we can modify the URL in exactly the same way as before.

[11]https://docs.gitlab.com/ee/ci/jobs/ci_job_token.html#configure-the-job-token-scope-limit

[12]https://gitlab.apertis.org/tests/apertis-test-cases/-/blob/apertis/v2023dev3/generate-test-pipeline.py

Secondly, we need to replace the repository URL in the test case with the new parameter. You cannot use templating within test cases themselves, you must setup a parameter or environment variable in the job definition that the test case can use. Parameters are preferable because they can be used in the `install` section of a test.

Putting things together, let's look at a section of a group template that:

- Pulls test case files from `apertis-test-cases` using a short-lived token.
- Sets up a parameter which contains the URL to clone `glib-gio-fs` using a short-lived token as authentication. We can use this parameter in a test case to obtain our test data.

```
 - test:
   timeout:
    minutes: 180
   namespace: system
   name: {{group}}-tests
   definitions:
{%- for test_name in tests %}
    - repository: https://gitlab-ci-token:{{ '{{job.CI_JOB_TOKEN}}' }}@gitlab.apertis.org/tests/ap
     branch: 'apertis/v2023dev3'
     history: False
     from: git
     name: {{test_name}}
     path: test-cases/{{test_name}}.yaml
     parameters:
      EXAMPLE_REPO_URL: |-
       https://gitlab-ci-token:{{ '{{job.CI_JOB_TOKEN}}' }}@gitlab.apertis.org/tests/glib-gio-f
{%- endfor -%}
```

We could then amend our test-case in `apertis-test-cases` to use the parameter like this (note that there is no `$` when substituting the parameter in an `install` section):

```
install:
 git-repos:
  - url: EXAMPLE_REPO_URL
   branch: 'apertis/v2023dev3'
```

Non-public jobs

These instructions are written to submit LAVA jobs for **ONLY PUBLIC** Apertis images. If you need to submit a LAVA job for a private image, there are few things that need to be taken into consideration and few changes need to be made to these instructions: `personal` or `group` visibility should be selected for your jobs.

If you really need to submit a private job, please contact the Apertis QA team.