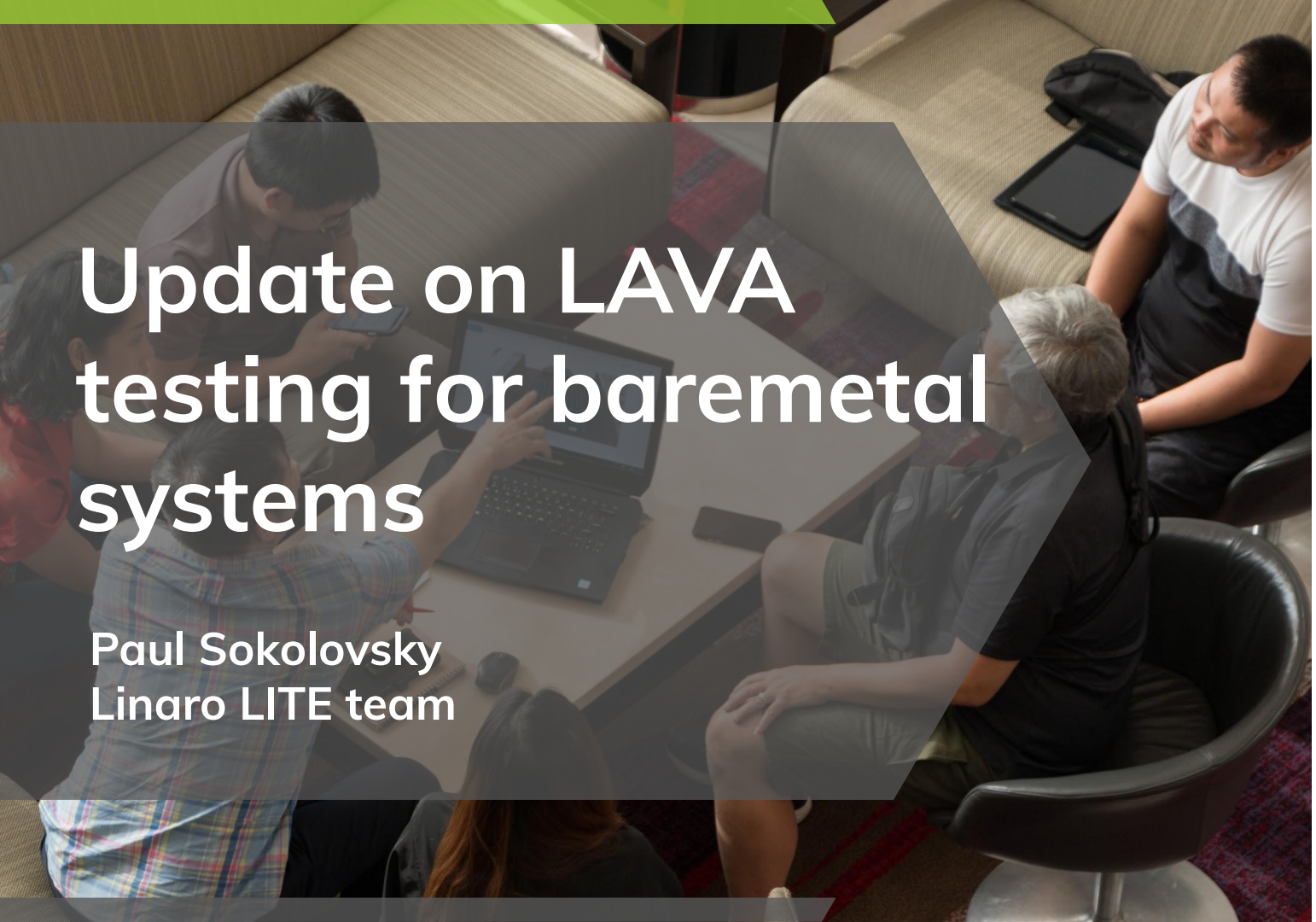


Update on LAVA testing for baremetal systems

Paul Sokolovsky
Linaro LITE team

Linaro TECH DAYS

Why test?

- Confirm that particular code (still) works as expected.
- The same at the large scale: system-level regression testing.
- Aid during development (TDD, etc.)
- Prove that code adheres to particular quality metrics/requirements.

Challenges in testing resource-constrained devices

Let’s consider how a typical Linux system is used:

- You get a generic ”starter” configuration (enough of general-purpose software already preinstalled).
- Install more software for your specific needs.
- Configure software.
- Install custom components.
- Run it.
- At any time, can easily install new software or new versions of existing.
- At any time, you may have good insight into system functioning, and can install additional software for analysis/debugging.

Challenges in testing resource-constrained devices

Testing of Linux systems parallels the “usage” approach:

- You install needed software.
- Then software needed to get tests running.
- Then drop (custom or pre-existing) tests
- And run them.

This is valid approach, because it mirrors a typical usage scenario, e.g. the above implicitly tests that:

- System allows installation of additional software during “use-time”.
- Particular software can actually be installed.
- And system can be configured for specific needs (running tests in this case).

Challenges in testing resource-constrained devices

Now let's compare with a deeply embedded:

1. You have a fixed-function device.
2. Oftentimes, there's no p.2, because you can't install additional software on the device.
3. Perhaps, you can add additional debugging features - as long as your device resources allow (e.g. free ROM/RAM size).
4. But that has security implications.
5. And in the end - you effectively would be testing a different device, not one intended to be used in production.
6. Add maintenance concerns - who's going to maintain all those adhoc changes to allow code be more testable on deeply embedded systems?

Challenges in testing resource-constrained devices

While techniques like additional debugging features and unit-testing are definitely applicable and important for deeply embedded development, the baseline is:

Black-box integration testing

Apply external stimuli (can also be none) to a device under test (DUT), check for expected reactions.

The approach also scales well for when we have debugging features and/or unit testing: a stimulus can be: set debugging option, activate functions, etc.

LAVA vs deeply embedded testing

The Challenge (semi-serious):

Make LAVA test what we have on our hands, don't let LAVA make us bend what we have to its whims and be burdened with maintaining LAVA-stricken versions afterwards.

LAVA vs deeply embedded testing

The Problem:

LAVA is largely designed, developed, and used for Linux-level testing.

3 test modes:

- `test: definitions` - “Install test scripts on target device” approach, suitable for Linux-level systems, used 90% of time in LAVA.
- `test: monitors` - No input, parse regularly-structured output from device.
- `test: interactive` - Feed input, check for a particular output - just the approach advocated in previous slides.

LAVA vs deeply embedded testing

3 test modes:

- `test: definitions` - “Install test scripts on target device” approach, suitable for Linux-level systems, used 90% of time in LAVA.
- `test: monitors` - No input, parse regularly-structured output from device.
- `test: interactive` - Feed input, check for a particular output - just the approach advocated in previous slides.

- Can also “test by proxy”, where you run a container which interacts with DUT in any way it wants, and just produces test results in a way expected by LAVA - very powerful and flexible, but also hardest to develop and maintain way of things. We keep this in store for when it’s really needed, concentrating on elaborating standard LAVA test modes for deeply embedded usage.

Improving “test: interactive”

- Adding support for discarding echoed input, which could lead to false positive output detection and breakdown of further test actions.
- Improved and extended documentation, describing the behavior more exactly and completely, with usage patterns and hints.
- Enabled basic multinode testing support.

Multinode testing

Consider e.g. a typical networking test: we need at least 2 devices participate in test (or it's not a "real" network).

A baseline scenario: DUT runs some kind of networking application, a "host" makes networking requests to it to verify that it behaves as expected.

"Host" part is represented by a docker container (we can't run anything directly on LAVA dispatcher for security reasons, and don't want to have a dedicated "host" device (for basic tests, maybe later we'll need one)).

Multinode testing was available for "test: definitions" (read: Linux), and was quickly-patched to enable it for "test: interactive", thus enabling (basic) networking tests for Zephyr.

Multinode synchronization

Consider the actual process of a simple network testing:

1. Start booting host.
2. Start booting device.
3. Wait for device to be booted.
4. Wait for host to be booted.
5. Run test

Requires “multinode synchronization”, which is effectively a core multinode feature (just booting something on 2 devices is easy!).

Multinode synchronization is currently available only for “test: definitions” (read: Linux).

Multinode synchronization for “test: interactive”

Consider the actual process of a simple network testing:

1. Start booting host.
2. Start booting device.
3. ~~Wait for device to be booted.~~ - Can’t do that, but can “sleep NN”, and hope that then it did.
4. ~~Wait for host to be booted.~~ - Can’t do that, so just hope it did. (Well, ok, can configure boot delay in Zephyr, requires patching at build time.)
5. Run test - and hope that everything is ready for it.

Multinode synchronization for “test: interactive”

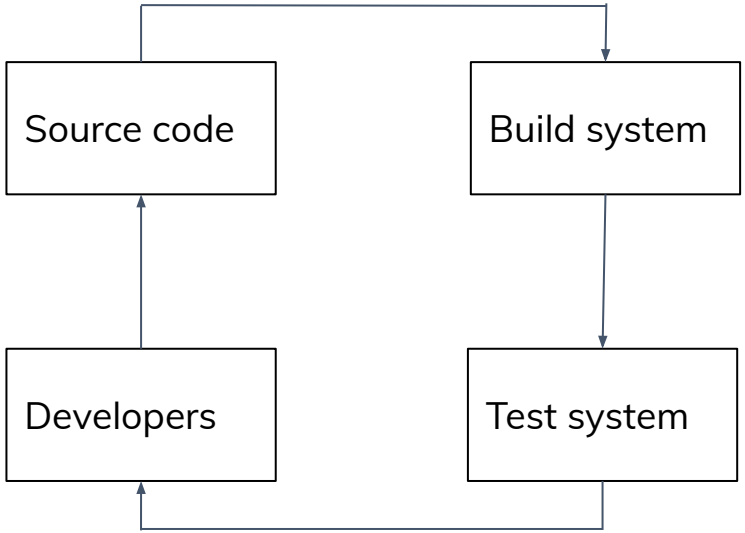
The patchset for multinode synchronization was developed in timeframe for LAVA 2020.03, but didn’t fit it due to amount of other changes. Then 2020.03 was delayed and cancelled to become 2020.04. The patchset is thus targeted to LAVA 2020.05.

Implementation is based on “test: definitions” way, syntax is closely followed. There’re 2 basic sync actions: `lava-send`, sending a message to other systems in a multinode group, and `lava-wait`, waiting for a particular message. Convenience actions of `lava-wait-all` ad `lava-sync`.

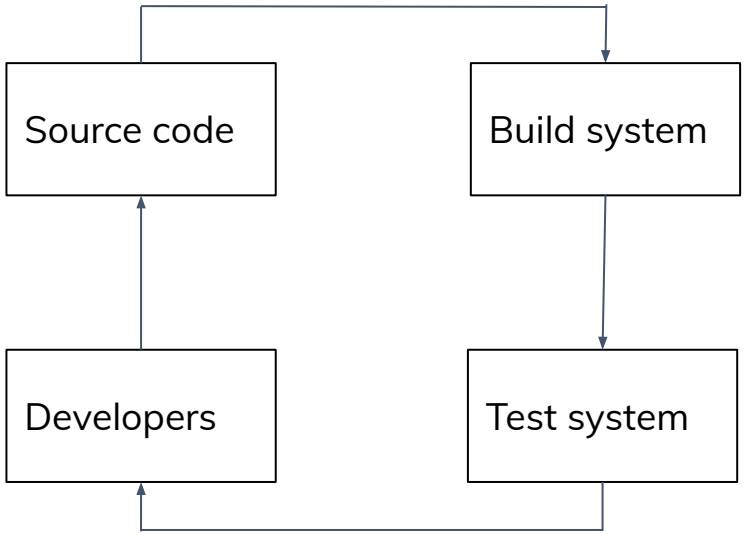
Extra: capturing device output by regex to variables, pass variables as additional payload with `lava-send` action. -> Server-side can print its (DHCP) IP address, which can be parsed and passed to client-side, so it knew where to connect.

Final extra: delay primitive, based on observation that we can’t easily run “sleep” on a deeply embedded device.

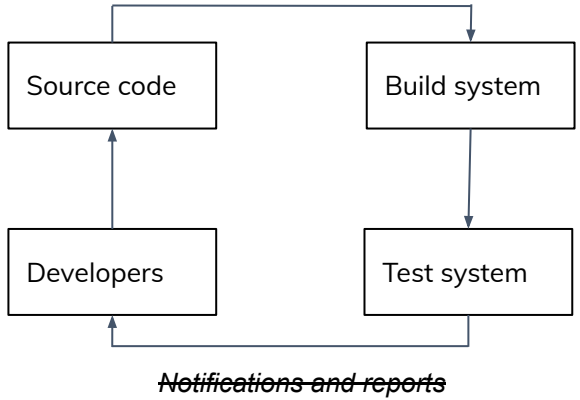
The CI loop

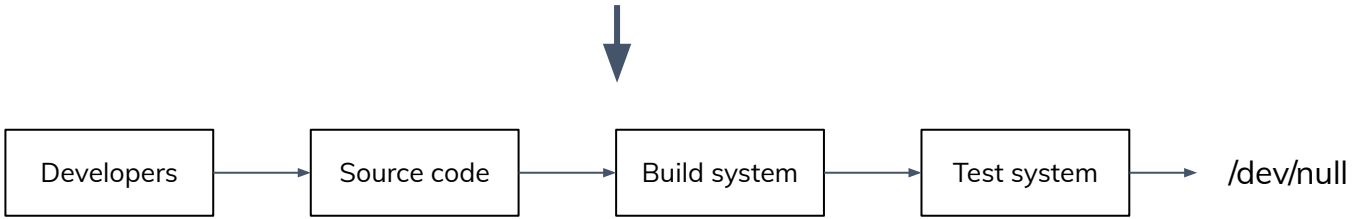


The CI loop


Source code
Build system
Developers
Test system
Notifications and reports

The CI loop


Source code
Build system
Developers
Test system
~~Notifications and reports~~


Developers
Source code
Build system
Test system
/dev/null

Case study: build notifications and reports

We use Jenkins as our build system:

1. When your build works, you don't receive anything (work undisturbed).
2. When build breaks, you receive a clear email notification.
3. While it's broken, you receive daily notifications (nagged into fixing).
4. When it's fixed, you stop receiving notifications.
5. Can go any time to web UI to check build status:

Configuration Matrix	gnuarmemb	zephyr
cc3220sf_launchxl	!	!
disco_l475_iot1	✓	✓
frdm_k64f	✓	✓
frdm_kw41z	✓	✓
mps2_an385	✓	✓
nucleo_f103rb	✓	✓
nucleo_f401re	✓	✓
qemu_cortex_m3	✓	✓

LAVA vs notification and reporting

The Challenge #2:

Get the same workflow from LAVA (as from Jenkins).

Turns out, you can't.

"Where're my failed tests, dudes?" (actual subject of email posted to lava-users mailing list)

Outcome:

- As it's known, git is "stupid content tracker". Then LAVA is "stupid test runner".
- Some rudimentary reporting capabilities, if you find your way thru (not exactly intuitive) UI.
- Some rudimentary

LAVA vs notification and reporting

The Challenge #2:

Get the same workflow from LAVA (as from Jenkins).

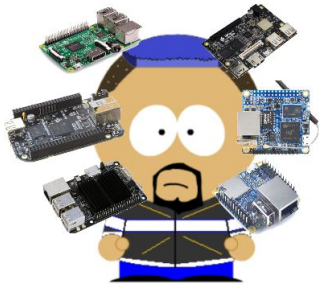
Turns out, you can’t.

LAVA vs notification and reporting

“Where’re my failed tests, dudes?” (actual subject of email posted to lava-users mailing list)

Outcome of the discussion:

- As it’s known, git is “stupid content tracker”. Then LAVA is “stupid test runner”.
- Some rudimentary reporting capabilities, if you find your way thru (not exactly intuitive) UI.
- Some rudimentary notification capabilities - you get daily spam regardless tests fail or not, have to grep email content to differentiate.
- “Write your own LAVA frontend”.



WAT?

SQUAD frontend

SQUAD to the rescue.

“Software QUAlity Dashboard”

Runs at https://qa-reports.linaro.org/

The message: “Don’t try to patch LAVA, try tp patch SQUAD”.

SQUAD frontend

We actually use SQUAD for Zephyr unittests. But, can you make much out of it?:

Myself, not sure (why for example there’re such wide jumps in all figures across runs?)

Augean Stables of Zephyr unittests

Turns out, SQUAD isn't first to blame (well, its UI is definitely too much data-packed, and you need wade your way thru it).

The initial problem lies in Zephyr in-tree unittest testing job. Problem: we have low-resource device, so each test is a separate executable. And we have hundreds of tests, some in different variants, for several boards. There're hundreds to thousands individual testcases. And for such scale and such numbers, process is not reliable, starting with publishing test executables for LAVA to consume. Then with LAVA UI, it's not manageable on its side, and even in SQUAD, what you see are some random jumping numbers.

Cleaning up Augean Stables

- “Ignore” Zephyr unittests test project for now.
- Set up SQUAD projects for appframework tests (MicroPython, JerryScript).
- Set up SQUAD projects for recently developed Zephyr networking tests.
- Use the experience from these to renovate Zephyr unittest project.

Conclusion and future work

- Looking forward to the “test: interactive” multinode synchronization patch being merged and LAVA version with it released. (Hopefully 2020.05.) With this, hopefully, majority of our “LAVA backend” issues will be solved. (Well, there’re more (un)known issues of course.)
- Learn/leverage SQUAD, using appframework and network tests as examples.
- Add missing features to SQUAD.
- Fix Zephyr unittest job.
- Write MOAR TESTS!

Thank you

Accelerating deployment in the Arm Ecosystem

Linaro

TECH DAYS