Linaro Connect
MADRID 2024 | MAY 14-17 2024

What does it take to ease SCMI development and testing

Cristian Marussi - Kernel Developer @arm

Agenda

- Whoami
- What is this about
- SCMI Overview
- SCMI stack layout: i.e. what needs to be tested
- SCMI server testing (platform FW)
- SCMI clients testing (focus on Linux agent)
- SCMI Prototyping and Development Issues
- Client development issues and the proliferation of SCMI ‘Hack’ Servers
- The Mother of all issues...
- ...and a surely controversial proposal

What is this about

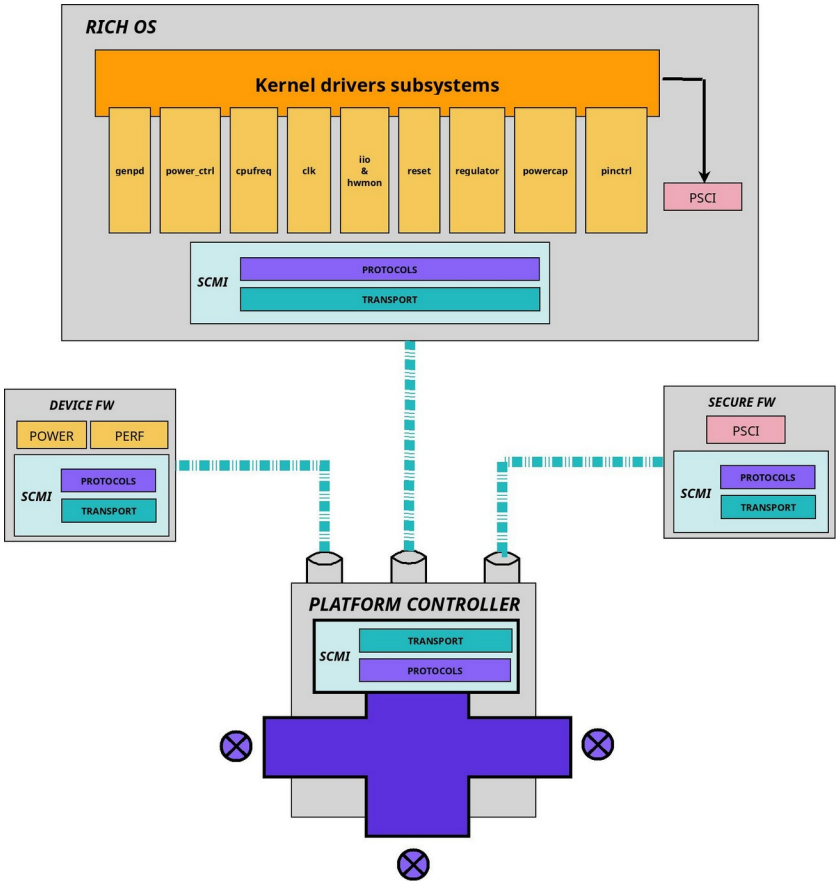
Evolution of the specification and improvements of the implementation led to **increased adoption** of the SCMI stack by vendors under a number of new deployment scenarios…

…this is good…but it means also that it is **increasingly difficult** to develop and test all the bits and pieces of the SCMI “distributed” stack.

The aim of this talk is to present a **walk-through** of the current state of the SCMI stack, looking at **issues**, shortcomings and pain-points around the SCMI development and testing process.

(...so this is NOT about missing SCMI features, planned refactoring etc etc...)

SCMI – System Control and Management Interface – Overview



- protocol to **abstract**/unify control & power mgmt
- **client/server** model:
  - a platform server (SCP/SCMI Arm reference)
  - multiple agents, identified by their channel, as clients issuing requests
- OS agnostic
- **extensible** by design
- **delegation** of mgmt control actions and **policy** enforcement to an external entity, the SCMI platform server, which lives in a distinct, isolated, trusted code base: it acts as the final arbiter on any request (deny/ignore)
  - → good for security: implausible/malicious requests from agents can be ignored
- the SCMI server, being the centralized endpoint of agents' requests, can **harmonize** conflicting requests from multiple clients around **shared** resources
  - → good for virtualization purposes

SCMI – Increasingly complex deployment scenarios

Thanks to the work carried out by Linaro, Arm and contributing vendors the SCMI stack has evolved/matured to support an increasing number of deployment scenarios.

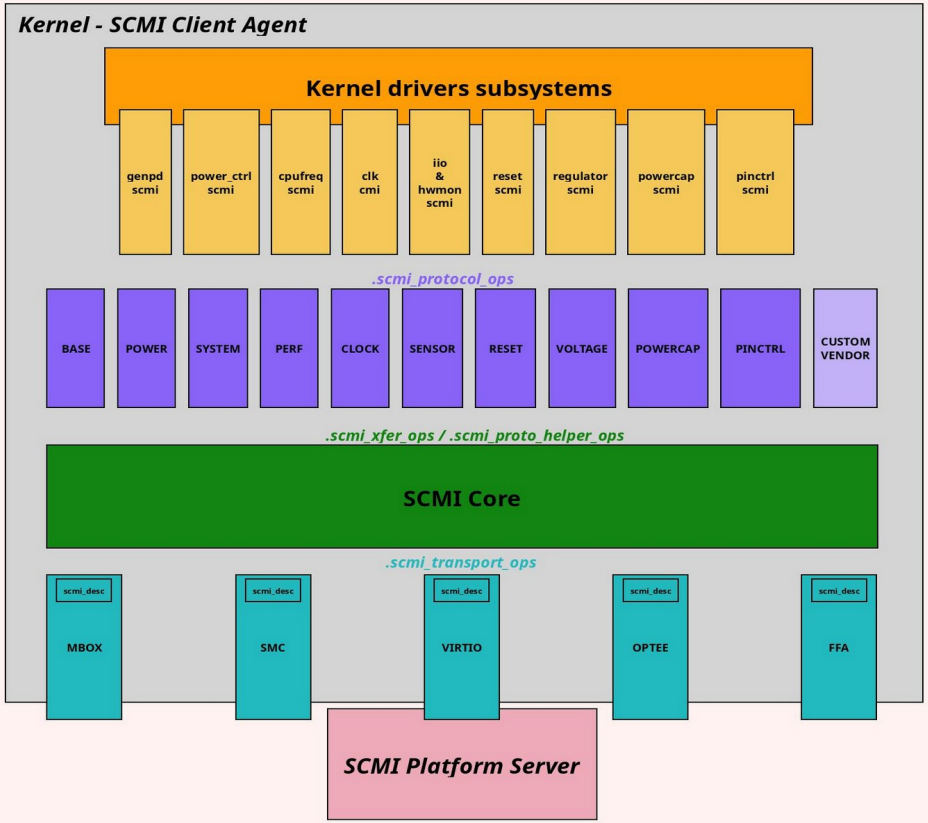
IOW, the SCMI server can live in a variety of different places:

- @dedicated MCU (SCP)
- @optee secure app
- @secure partition
- @VM

… using a number of different transports: mbox / smc / optee / ff-a / virtio

This by itself complicates validating the SCMI server compliance and, moreover, a number of proprietary vendor SCMI servers has appeared, carrying potentially a variety of IMP_DEF characteristics that we should be able to test against in the Linux SCMI agent.

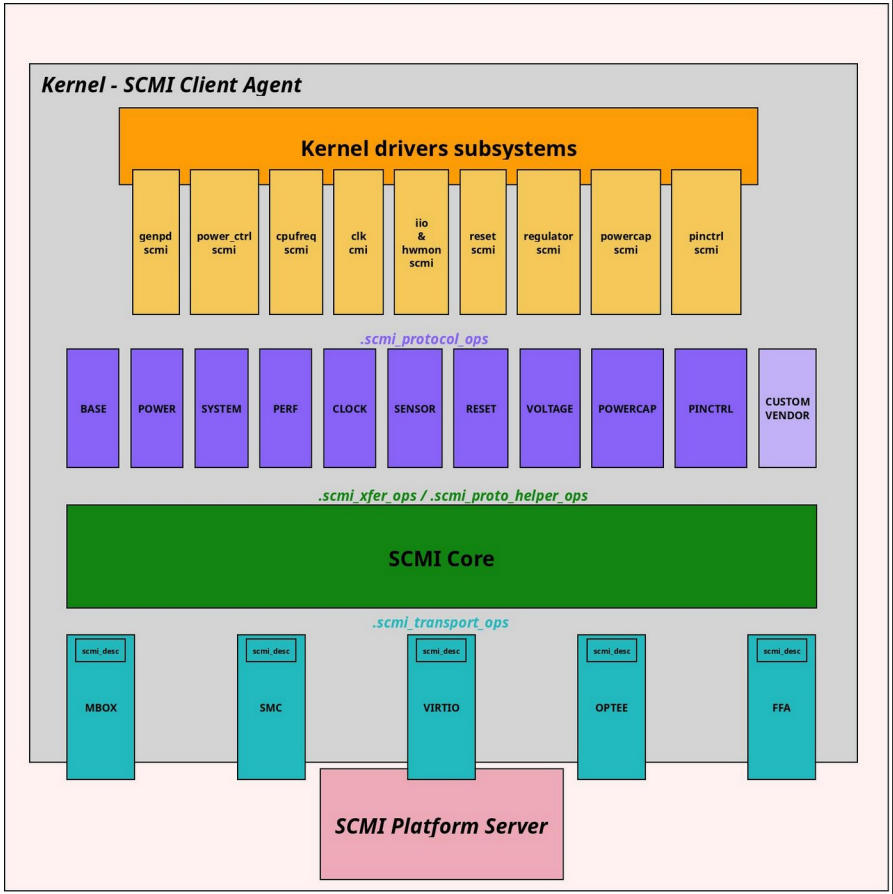
SCMI stack - Linux Agent perspective



Layered design:

- **SCMI drivers:** plugged into the related Linux subsytems, use the available **protocol operations** to issue SCMI requests.
- **SCMI Protocols:** implement the **protocol operations**, knowing how to build and send the appropriate SCMI messages using the **xfer operations**.
- **SCMI Core:** implement **xfer operations** and takes care of in-flight message tracking, reply timeouts, polling mode, transmission errors while using the configured **transport operations.**
- **SCMI Transports:** implement transport specific methods to send messages and fetch responses.

SCMI – Testing what exactly ?



Looking at the ugly diagram, we have a hint at what we want to test:

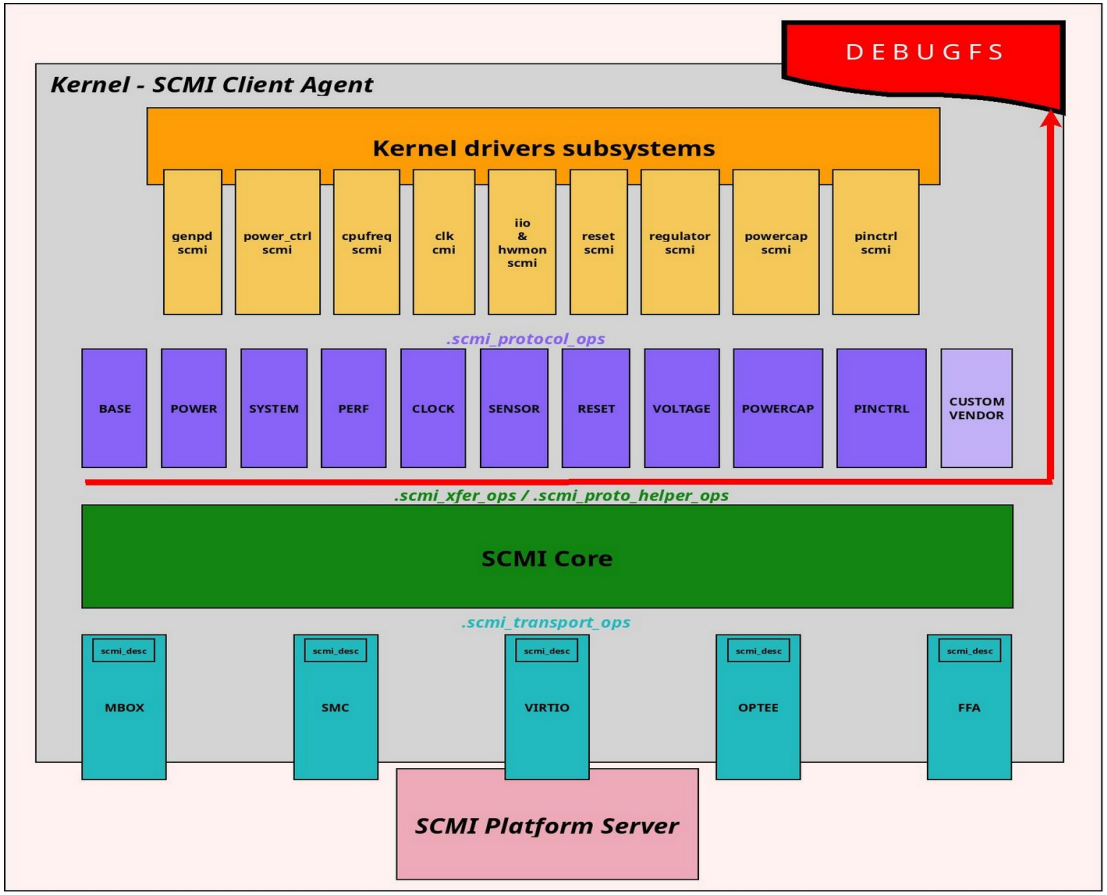
- **Server** (wherever it lives):
  - SCMI **Compliance** to the specification
    - ACS test-suite: scmi-tests
  - Stress testing ?
  - Fuzzing ?
- **Client**: cannot use regular SCMI drivers to test SCMI interfaces:
  - cannot exercise the API to its full extent
  - limited means of interaction from drivers

The aim is to test only:

- **.scmi_protocol_ops**
- **SCMI Core** functionalities
  - message build/send/tracking
  - errors, timeouts

Do NOT aim to test .scmi_transport_ops alone.

SCMI Raw – v6.3



A debugfs interface to inject/snoop bare LE binary SCMI messages.
Uses the standard core facilities and transport to route the injected messages to the SCMI server wherever it lives.
No need of custom transport test hooks.
Regular SCMI drivers are inhibited in Raw mode to avoid false-positives.

```
# tree /sys/kernel/debug/scmi/0/raw/
/sys/kernel/debug/scmi/0/raw/
├── channels
│   ├── 0x10
│   │   ├── message
│   │   └── message_async
│   └── 0x13
│       ├── message
│       └── message_async
├── errors
├── message
├── message_async
├── notification
└── reset
```

SCMI Raw – Server testing

The principal use-case for SCMI Raw injection capabilities is, of course, Server-side testing:

- **SCMI ACS compliance suite** now supports also Raw mode [1]
  - *NOTE:* using ACS in Raw mode limits compliance testing to the perspective of a Normal world agent
- Easy to write a tool for stress-testing or fuzzing...or just go with a poor man solution:
  - *dd if=/dev/random of=/sys/kernel/debug/scmi/0/raw/message bs=128 count=1*

➔ ***ISSUE****:* Running SCMI compliance on a real platform of some kind means risking unwanted interactions with other SCMI agents acting in background.

[1]:https://gitlab.arm.com/tests/scmi-tests

SCMI Raw – Linux SCMI Agent core functionalities testing

SCMI Raw access can also be used for testing of **client SCMI core functionalities** to verify the core (and transport) support for:

- Basic Sync & Async commands handling
- Notifications handling
- Timeouts and late replies
- Unexpected/malformed replies
- OoO replies
- Handling of multiple parallel request (depending on underlying transport)

This can be done really in a few ways: as an example an **SCMI optional Test protocol** could provide some sort of **“echo service”** to exercise all of the above in a controlled manner.

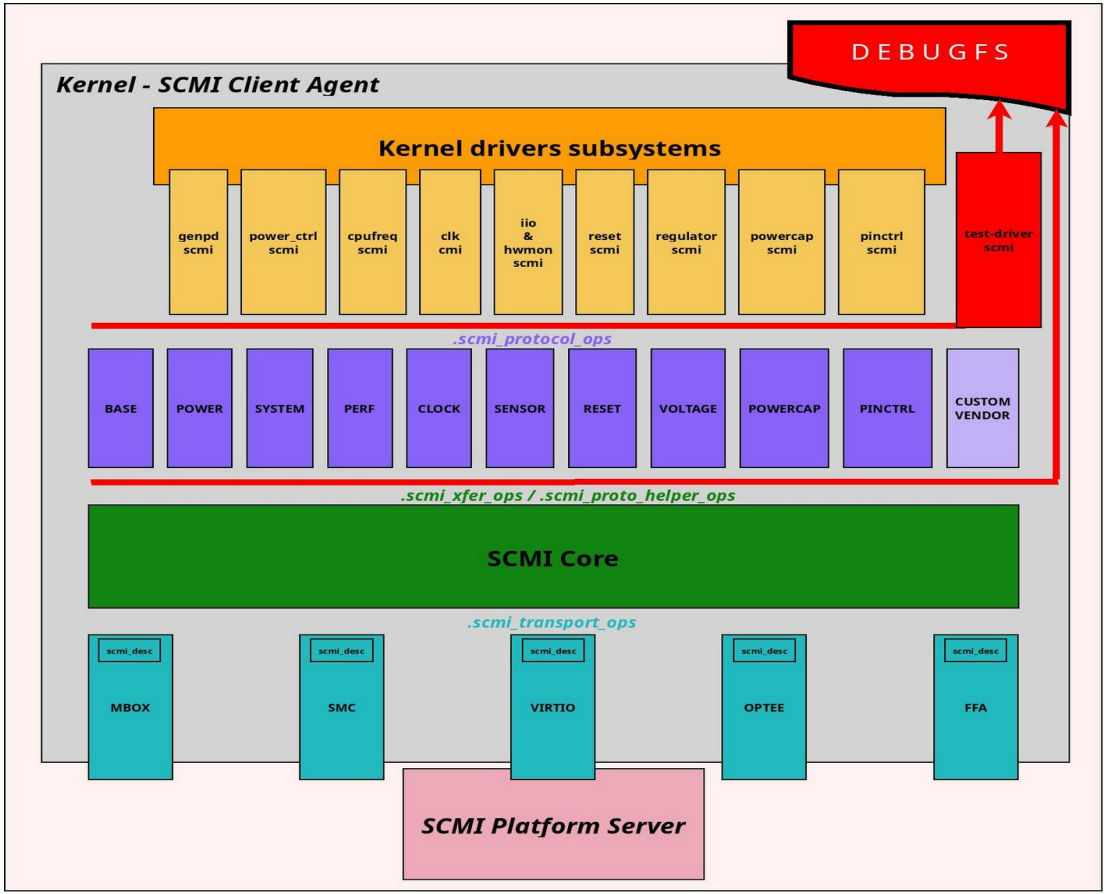
➔ ***ISSUE*:** for all of the above to work we need an SCMI backend-server that can be made to misbehave at will and can support effective multi-threading.
Real SCMI servers (proprietary or not) are hardly designed to misbehave on demand.

SCMI Test Driver – Linux SCMI Agent protocol_ops testing - WIP

An SCMI driver that:

- Registers with each SCMI protocol found supported by the platform
- **Exposes all the protocol_ops** and resources descriptors, basically all the content of include/linux/scmi_protocol.h, via **debugfs** to allow for extensive **Kselftest** scripting.
- Excludes regular SCMI drivers to avoid unwanted interactions while testing
- Only posted once as an RFC a few years ago...
- ...currently 90% complete in term of protocol coverage...
- ...but with all of the zillions Kselftest to be written
- Also generally useful for runtime ‘introspection’ and basic system interactions
- Is it worth ? (a lot to write and maintain…)
- Kunit maybe an alternative way to do this ? (not investigated)

SCMI Test Driver – WIP – Is it worth ?




```
# tree -L 1 /sys/kernel/debug/scmi/0/testing/
/sys/kernel/debug/scmi/0/testing/
|-- info
|-- protocol_0x11
|-- protocol_0x12
|-- protocol_0x13
|-- protocol_0x14
|-- protocol_0x15
|-- protocol_0x16
|-- protocol_0x17
|-- protocol_0x18
|-- protocol_0x19
`-- protocol_0xFF
```

```
sys/kernel/debug/scmi/0/testing/protocol_0x15/
|-- 000
|   |-- enable
|   |-- info
|   |   |-- async
|   |   |-- continuos_updates
|   |   |-- intervals
|   |   |-- name
|   |   |-- num_axis
|   |   |-- num_trip_points
|   |   |-- scale
|   |   |-- sensor_config
|   |   |-- timestamped
|   |   |-- tstamp_scale
|   |   `-- type
|   |-- reading_get
|   |-- reading_get_timestamped
|   |-- sensor_config
|   |-- sensor_trip_notifs
|   `-- trips
|       |-- 000
|       `-- 001
|-- 001
|   |-- enable
```

SCMI Test Driver - WIP

- ***ISSUE***: what to use as a backend SCMI server ?
  - Needs something that implements all possible SCMI protocols (hardly implemented in full on any real platform due, trivially, to space concerns)
  - Notifications ? Not all notifications can be easily triggered on demand
  - Difficult to configure tests expectations for the general case on a real platform
  - Is it safe to exercise freely all the possible protocol_ops invocations ?
    - Will it kill/freeze the DUT during the run by mistake ?
    - Will it fry the DUT all-together due to some misconfiguration/bug ?

SCMI New Protocols: Prototyping & Development Lifecycle

- **IDEA** for a new SCMI protocol results in ATG working on its specification for SCMI-next:
- ATG prototypes new protocol: FVP or real platform + mocking
- ***OR***
- Vendor prototypes the new protocol in its own proprietary server and platform

- ATG TestTeam starts adding new test-cases to the ACS compliance suite
- KernelTeam starts planning/working on upstream Linux Agent support
- SCP FW team starts planning/working on upstream SCP/SCMI support

All of these steps happen on **heavily different/disjoint timelines/deadlines**….as is normal, given different teams and companies are involved, each one with its own priorities…

… most probably, before any complete SCMI client/server upstream support is ready

- ATG releases the new Beta-public spec and, ideally, the updated ACS test-suite

- ***ISSUE:*** ATG TestTeam had nothing reasonably stable and complete on the server side to validate their new ACS test-cases.

SCMI Agent development process is the most problematic

- **Server side development** for *SCMI-next* just needs:
  - A published *SCMI-next* specification
  - SCMI Raw access from a Linux Agent userspace
  - (possibly an updated ACS to produce SCMI requests through Raw)
  - no need to have any kind of Linux *SCMI-next* driver support to start server development

Instead, **SCMI Agent development** process is more tricky: it **needs** some sort of **SCMI** speaking **entity** somewhere to play the server-role.

For **this** reason **and** the out-of-sync development **timelines**:

- A number of **simplified 'hack' SCMI server** were borne, scattered around various codebase:
  - U-boot mock-server
  - TF-A mock-server
  - ACS mock-server (!)
  - scmi_emu: userspace emulated mock-server (that's me!): kvmtool + guest + scmi_emu

...beside the more usual setup:

- SCP + mocking + QEMU/FVP

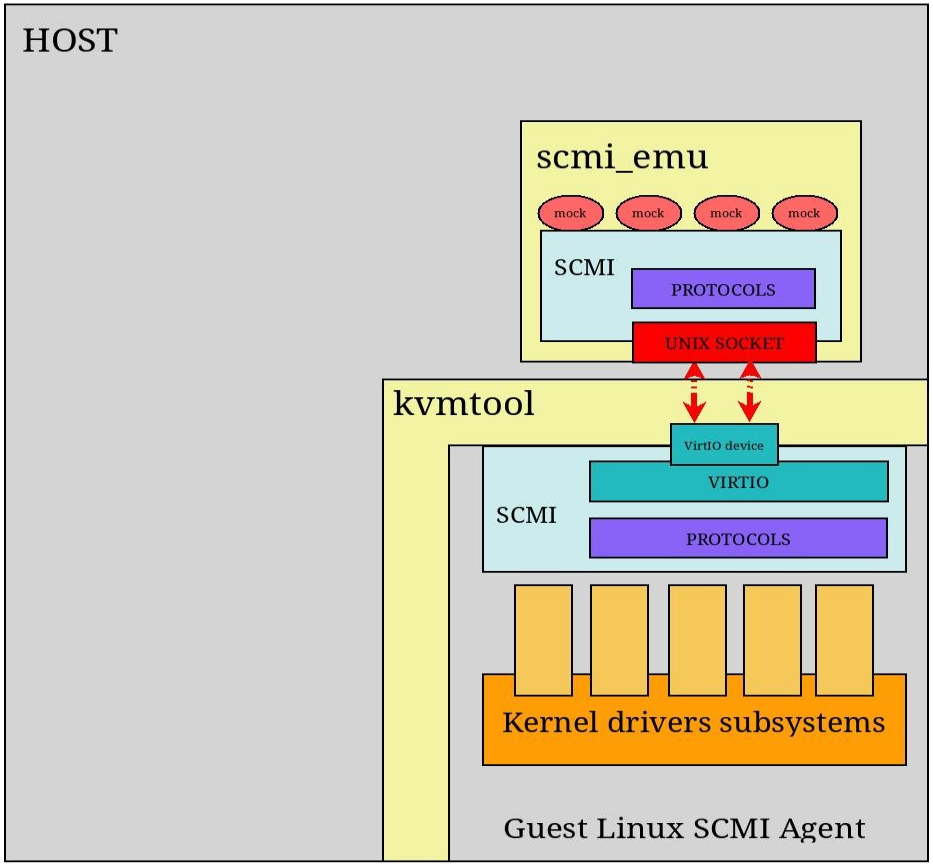
SCMI ‘Hack’ Servers

All of these hacks share the common **aim** to provide a **simplified SCMI server** entity which:

- Does **NOT** have to cope with real **HW support** (all mocked)
- Does **NOT** need proper extensive **abstractions** in its design (it is not a product)
- Does **NOT** need to support **multiple platforms/architectures**
- Does **NOT** need extensive, flexible, well abstracted configurability
- Does **NOT** have to follow any specific **IMP_DEF choice** that limits its test-ability (mono-thread)
- Code base is **small enough** to be understandable: easy and quick to prototype new additions
- Can cope with any kind of out-of-spec requests without risks...all is mocked, **nothing can fry**
- Can be easily reconfigured at build/run time to address specific testing needs

scmi_emu: Userspace Emulation

- SCMI Agent - a standard Linux Guest using SCMI/VirtIO
- SCMI Server - a Linux userspace multi-threaded application running on the host receiving and sending SCMI messages through Unix datagram sockets
- kvmtool exposes a VirtIO SCMI device to the guest:
  - intercepts the SCMI traffic from/to the guest
  - re-routes the SCMI messages back and forth between the virtqueues and the SCMI emulation Unix datagram sockets



scmi_emu: Userspace Emulation – Pros

1) **Tiny build**-system: few-lines makefile just supports arm64 and x86 cross-compilation
2) Heavily **multi-threaded** by design
3) **Thin abstractions** just to provide an aid in message build and transmission
4) All is mocked and easily re-configurable: **nothing to fry** (look mama no HW !)
5) Small enough code-base that a new protocol support can be drafted in one afternoon
6) Full control of packet scheduling logic
   ➔can be **made to misbehave at will**, event by re-configuring it at runtime (echo service)
7) Server behavior **can be reconfigured** at runtime potentially using ad-hoc **debug** protocol:
   ➔trigger **notification** generation (e.g. SytemPower Graceful shutdown)
   ➔configure **values to be read-back** at runtime out-of-band (no fixed expectation in CI)
8) **Easy to setup and run**...a Linux userspace app running in the host + a kvmtool guest
9) Can be used to **host any SCMI guest** for development purposes:
   ➔**freeBSD** SCMI support is growing….starting with VirtIO transport
10) **Throw-away test-code** is welcome ! e.g. powercap recursion (...with the option to refine/merge later)
11) Can be tested for **compliance with ACS** like any other SCMI server (not saying that it passes now)
12) Can be used to **test VirtIO Transport** misbehavior
13) **No other interacting agent** on the system: only Linux OSPM agent requests are seen

SCMI Userspace Emulation – Cons

1) Massive duplication of work (the elephant in the room)
2) Limited to VirtIO Transport based setups: agents in guests
3) Only testing from Normal world view
4) Not performant as the QEMU-based virtualized SCMI which uses vhost-users
5) Only support for “Unices” guest SCMI agents (:P)
6) Developing/testing both side of the world together by the same team/person is bad process
7) Cannot snoop messages from other agents, so it can support only Linux Agent (at of now…)

At the end, **scmi_emu** is really just a different (worst) way to achieve the same virtualized SCMI scenario as realized by Linaro with QEMU and the SCP/SCMI reference platform firmware: the virtual setup is more limited in scope, tailored for debug/test, and **worst in performance** for sure….but...

… beside the different virtualization architecture, the main difference that makes it more desirable to ease Client development, is in fact the **usage of a fake, simplified, stripped-down SCMI server.**

Mother of all Issues (related to SCMI Agents development / testing)

IMHO, **using a real** (even virtual) **platform** embedding a real fully fledged SCMI server of any kind (proprietary or open) for prototyping and development is the **origin of all** client side development **issues**.
Since, in this way, we are in fact using a **full-fledged "product"** which, by its nature:

- has to support a number of real platforms and its HW which brings complexity/abstractions
- adopts a number of IMP_DEF implementation choices that can limit testability on client side
- was not designed to misbehave easily/safely at will
- cannot (or want not) support a complete SCMI stack (all protocols / all optional features)

...while, at the end, we just **needed a minimal subset of its functionalities**, i.e.

- a ***compliant SCMI-speaking entity*** supporting all of the protocols and features

Now, I am not saying that we must necessarily find and use one single virtual SCMI platform to rule them all ... neither I am trying to sell my solution...

… since each "hack" SCMI server has probably its own merits...but there is something could be done to **reduce the massive amount of work duplication**...

A pure SCMI library (a long shot :P)

In an ideal (SCMI) world, we would have a **pure SCMI library** which:

- can be plugged on any backend to use it as data source
- can be connected to any frontend for the effective SCMI tx/rx message processing
- has no included and pre-baked message scheduling logic

...with that available we could:

- write **unit tests** upfront and run it in CI
- use it as **a base to build your preferred simplified-server** to fit your needs, without duplication of work, and knowing that is reasonably SCMI compliant
- **prototype/draft** new features on it early on and **then, maybe, refine** that same code base for final library support (instead of throwing all away)
- have a reasonably compliant SCMI server against which **validate the ACS test-suite**

And you could end up releasing a bundle of:

➔ ***SCMI Specification + validated ACS test suite + SCMI reference library***

Linaro Connect
MADRID 2024 | MAY 12-17 2024

Thank you