RFC-0001: Racklet

Collaborate on HackMD

(for docs.racklet.io)

RFC Metadata

Authors (in alphabetical order):

- Ayan Borthakur, @ayan1948
- Dennis Marttinen, @twelho

- Lucas Käldström, @luxas
- Verneri Hirvonen, @chiplet

Status (as defined here): `Implementable`

Creation Date: `2020-12-10`

Last Updated: `2020-06-08`

Version Number: `v1.1.1`

Summary

Racklet is a fully-integrated, miniature server rack. It is a scale model of a hyperscaler server rack loosely based on the Open Compute Project (OCP) rack designs. It consists of several pluggable "compute units", a Rack Management Controller (RMC) and shared power delivery (the so-called busbar). In addition, there is some functionality borrowed from OCP Edge Cloud implementations in place, such as a common interconnect for the compute units (SMBus).

Physically a Racklet rack is a bit larger than a one liter milk carton and hosts single board computers conforming to the Raspberry Pi 3/4 form factor. The defining features of Racklet compared to other "Raspberry Pi clouds" are the fully integrated and **secure** but still pluggable open source firmware/software/hardware solutions, and the scalability enabled by e.g. hotplug support as well as the inexpensive and available manufacturing techniques applied.

Racklet aims to inspire their users to explore how modern, advanced server architectures work in practice, in a tangible and educational way. With the new-found knowledge and inspiration, the user may apply their modernization skills on traditional server infrastructure, which improves the status quo and pushes the industry forward. The aim of the project is also to write modular pieces of software and firmware that can be re-used across a diverse set of systems, not only on Racklet itself.

Racklet is defined by its values and principles. Below you can read about the 9 values that shape this project, and what they mean in practice. One value to highlight here is *accessibility*. Racklet is 100% open source and should be accessible to a group as diverse as possible from all over the world. This means all parts of the system should be reproducible through open PCB designs, 3D-printed casing, and commodity, off-the-shelf hardware. We want to lower the barrier of entry for this domain.

In this RFC, we will outline what Racklet is, why create it (from the user's perspective), and the values and design constraints for the system.

In short, we'd like to say that

> Racklet aims to be for Cloud Computing what Raspberry Pi is for Programming, and Arduino for Electronics

Have fun tinkering with it!

Motivation

Problem statement (choose the one that appeals to you):

> Distributed systems of various kinds are steadily becoming the foundation for all important technological environments; and their backends require ever-increasing capacity. The world of cloud computing software is rapidly evolving towards dynamic, scalable and self-correcting systems. The amount of tools and services required to run high-performing cloud systems in a diverse range of environments are vast, and the integration between them complex[1]. Due to the complicated nature of this quickly-evolving cloud infrastructure, how can newcomers to the field of cloud computing get an idea of the landscape and workings of the systems in an effective way?

and/or

> Empirically, it seems that many mainstream server and infrastructure provisioning guides (or even fully-integrated solutions) often don't put enough effort into securing the firmware stack of the server, but focus more on the layers above. This, in combination with the firmware being proprietary, often leads to situations of unknown/random bugs and security flaws (caused by the user due to insufficient knowledge, or flaws in the firmware without patched versions). Different pieces and layers of firmware doing the same things (and often too much) in subtly different ways, but without it being possible to only activate what you need or want to extend. Firmware written in C suffers from many common memory errors and even security flaws. Network booting of servers often stick to legacy and unreliable protocols like TFTP or similar, and skip any verification of the payload's integrity or authenticity.

Proposed solution:

As detailed in the Summary section above, *"Racklet is a fully-integrated, miniature server rack"*. Building on top of good ideas and practices from OCP, the open source firmware community (e.g. LinuxBoot, u-boot, TF-A, etc.), the Raspberry Pi educational model, and the advancements in writing secure system software and firmware with Rust, we think we can push the state of the art here, and educate newcomers to the field of secure and open source cloud computing at the same time.

As pointed out in the first problem statement, it can be challenging to "get into" to the server infrastructure world, especially if you want to run your own servers, due to a multitude of reasons, including complexity, lack of standardization at several layers of the stack, and cost. Through Racklet, we want to explore these venues in a tangible and low-cost way with the help of Raspberry Pi's (or alternate single board computers). It has been shown earlier that Raspberry Pis can be helpful for teaching cloud computing[2], hence we believe this could be a good fit.

The goal of the project is to be comprehensive and "real-world" enough to feature, at least conceptually, most of what you would find in a modern hyperscaler server infrastructure environment, but still clear, well-documented, and user-friendly enough to attract and welcome new, future talents to the server and firmware worlds.

Goals of this RFC

1. Describe what Racklet is and what it might be used for.
2. Define the values used to guide the design and decision process.
3. Describe the purpose and goals of the system from a user point of view.
4. Define high-level layers of the system.

Non-Goals of this RFC

1. Go into details about what any layer, interface or API contains or does.
2. Describe what exact parts, technologies or interfaces should be used.
3. Define a timeline for the project.
4. Define project governance.

Proposal

This proposal consists of a detailed breakdown of the Values of the project, who we think will be using this project (what kind of user persona we are optimizing for, in User Perspectives), how we envision the users will use it (User Goals), and finally, a high-level overview of the

hardware/software Layers.

Any further technical details are out of scope for this RFC. Those will be covered by upcoming, more detailed RFCs.

Values

The following values apply to the whole system, and are sorted roughly in priority order, as a guideline when making decisions. Future RFCs should outline in their "Motivation" chapter what values have (or have not) been adhered in the RFC and how.

> **Disclaimer:** All of these values are aspirational, they are not literal guarantees that can be used for liability claims. We expect to iteratively improve towards and get closer to these goals as the project matures and new versions are released.

1. **Security**
    1. **Security by design**: Security should be at the top of our minds at every decision we make. All design proposals must consider how the proposed change affects security concerns.
    2. **No old/insecure protocols**: We won't accept old/insecure protocols or ways of doing things (e.g. TFTP). If we need to choose between interoperability and an insecure standard, we choose the more secure alternative, although that would mean we go against the norm.
    3. **Improve status quo**: We aspire to improve the status quo of "secure-by-default" solutions and concepts available out there. When we find ways to improve the state of the art, we preferably contribute patches to the respective upstream, otherwise, depending on the situation, build re-usable pieces of code that bring the industry forward.
    4. **Defense in depth**: Design according to the "defense in depth" and "least privilege" methodologies. For example, the network is considered being an insecure channel, unless proven otherwise (Dolev-Yao adversary model).
2. **Interoperability**
    1. **Openness**: The truly most effective way of driving innovation forward in our minds is to define open (source) APIs, share code freely, and collaborate with fellow community members.
    2. **De-facto standards**: Implement well-known, existing and de-facto APIs instead of creating new ones when not needed.

3. **Declarative management**: This is prominent in the cloud native space, but not so much in the embedded and firmware space. We believe declarative APIs are very useful and powerful, especially as it becomes "obvious" to write state reconciliation loops that follow the observe-diff-act pattern.
4. **Consistency**: We want to expose consistent (declarative) APIs across the stack for the same "look and feel". Use common meta-protocols like JSON and YAML.

3. **Accessibility / Reproducibility**

    1. **Open source**: Racklet is 100% open source software and hardware. Anyone can contribute, improve, fork and access the project. The dependencies of the project will also be openly accessible.
    2. **Common off-the-shelf parts**: Only use commonly available components that can be aquired in most parts of the world in a frictionless manner. In other words, no exotic hard-to-reproduce designs.
    3. **3D printed parts**: For non-off-the-shelf casing, we will provide 3D-printable designs that can easily be reproduced. Modelling is done in software that does not require paid-for subscriptions. Both printable STL output and the underlying save files are published to GitHub.
    4. **Reproducible PCBs**: For non-off-the-shelf PCBs, we will release schematics freely reproducible, made in open source software such as KiCAD. We will try to make sure that the PCB can be ordered from major PCB manufacturing/assembly services.
    5. **Documentation**: Documentation will be made available through our mdBook site, our GitHub organization, code-autogenerated documentation services such as crates.io and pkg.go.dev as well as our blog detailing the development process and important decisions made, featuring these design proposals. This documentation will lower the bar to entry in order to increase accessibility.

4. **Modularity / Compatibility**

    1. **Modular design**: Our designs, both hardware and software, strive to be as modular and extensible as possible. We strive to follow the Unix philosophy. This will allow for portability between e.g. different hardware modules implementing the same interfaces, or extensibility where the user demands other features than the default.
    2. **Raspberry Pi compatibility**: The Raspberry Pi physical design (mounting holes, GPIO layout, dimensions, HAT spec) has established a "de facto" standard, and any other single-board computer implementing this interface should be compatible with the system with minimal modifications.
    3. **Portability**: The code we write includes parameters for the platform it's running on so it is fairly easy to port the code to a new alternate architecture. We primarily support ARMv8 for the compute units.

4. **Loose coupling**: We strive towards loose coupling. This means that each component has as little knowledge of and hard dependencies on other components. Components should be easily interchangeable with alternate implementations.

5. **Transparency**

    1. **Observability**: All data logging/aggregating components in the system must expose metrics compliant to the OpenMetrics specification.
    2. **Debuggability**: We expose standardized debug headers (e.g. UART/JTAG) from our PCBs. Low-level firmware troubleshooting is accessible and documented for both our microcontrollers as well as the compute.
    3. **Energy monitoring**: Energy usage should be measured individually for the various components in the system, in order to transparently and automatically be able to track where power is consumed. This also allows for higher-order aggregation and data processing related to energy.

6. **Maintainability / Upgradability**

    1. **Hot swappability**: Modules of the system (especially the compute) should be able to be hot-swapped without disturbing the operation of other modules in the rack while the rack is operating.
    2. **Upgradability**: The modularity of the system should allow that individual pieces of the system (e.g. compute, storage, network switches, power supplies) should be upgradable without having to disrupt the rest of the rack, or disregard existing, functioning parts. This will minimize E-waste produced over time.
    3. **Auto-upgradability**: It should be possible to enable automated (security) upgrades safely using e.g. A/B partitioning schemes.
    4. **One-time hardware setup**: The hardware setup steps (3D-printing, PCB assembly, firmware flashing) are only performed once per rack. None of them are performed when dealing with software, even when resetting the entire cluster. This will allow for fast and less error-prone reconfigurations with faster setup/teardown cycle times.

7. **Affordability**

    1. **Sensible rack cost**: The price point of the Racklet Bill-of-Materials should be low enough to be accessible for hobbyists and educational organizations. Our target price range (VAT-exclusive, all essentials included) is 400-500€ per rack (containing 5 Pis). If this configuration is too costly it should be possible to switch parts out for a lower total price.

User Perspectives

> **Main User Persona**: *"Racklet is for a student, hobbyist, teacher or industry professional who wants to learn and understand modern, increasingly important distributed and cloud computing skills to foster education, research and work opportunities."*

Tangible Cloud Teaching

A University distributed computing class could use one or multiple Racklets as a prototyping platform to enable fast learning augmented by practical training.

Mobile Cluster for Conferences

Companies demoing their software and hardware at conferences could use this as an innovative way to showcase their solutions.

CTF contests

As this aims to be a scale model of a real cloud environment, it forms a good target for capture-the-flag hacking contests.

Workshops

In the same spirit as the Tangible Cloud Teaching use-case above, we anticipate it would also be a good fit for commercial trainings, when time is limited and you quickly need to demonstrate how some specific piece of technology works in detail. The instructor can easily engage their audience in a practical way.

Homelabs

Hobbyists could use one or multiple Racklets to establish home infrastructure while learning more about and further developing the platform.

Research and Development

Racklet can be used for Research and Development purposes in Computer Labs where some specific application's architecture is being validated on a real-world but inexpensive system. This is an alternative to data center simulation programs. Racklet captures the real-world aspects that a simulation might not take into account or can not realistically represent. Realistic outages and partial failure modes (e.g. power outages or network partitions) can easily be applied to the system in order to research how the tested application reacts.

User Goals

1. **Achieve user goals through containers and Kubernetes**: The user wants to use containers and Kubernetes as their preferred way of running applications, and hence some base functionality of that should be provided in a "batteries included, but swappable"-sense. At the end of the day, we're building this project so that the user can build something nice on top of it through these standard interfaces.

2. **Fast reconfiguration / turn-around time**: The user wants to configure their hardware once, and after that be able to set up and/or recreate the whole software stack from the ground up multiple times over with minimal hassle. For example, an educator may want to rebuild the rack configuration, trusted certificates, etc. or do a "factory reset" for every class/workshop they run.

3. **Secure firmware[3] and software updates**: When the user gets notified that a new release is available, the user *doesn't* want to do it the "classical" way of downloading some hex binary and flashing it manually for each server. Instead, they want the upgrades to be atomic (e.g. A/B partitioning), secure (payload is signed), automated and defend against common upgrading attacks (e.g. rollback attack). Optionally, automatic deployment of upgrades can be enabled.

4. **Network boot in a zero-trust environment**: The user should feel ready to plug Racklet in to (almost) any existing network, without the system interfering with existing devices on the network or vice versa. This goes strongly in hand with #3 as ensuring security, especially in the boot and upgrade process, is of paramount importance[4].

5. **End-to-end encryption and authentication**: The user wants to feel comfortable running software on top of their Racklet without worrying about e.g. MITM attacks in the surrounding untrusted network it is connected to. Hence, all TCP/IP traffic should be end-to-end encrypted, authenticated or preferably, both.

6. **Hot swappability**: The user wants to be able to upgrade their racks often for newer hardware as they enter the market. The user also wants to be able to maintain and service the rack while it is running, and dynamically expand capacity at runtime.

7. **Keep track of power usage and efficiency**: For educational purposes and intelligent power control, it is important to know how compute utilization translates to power consumption across the system. The user wants transparently reported metrics so that they can analyze the data and utilize higher-level power control routines to optimize power draw.

8. **Physical portability**: Racklet should be lightweight enough to be carried by hand and should not require specialized equipment or disassembly for transportation.

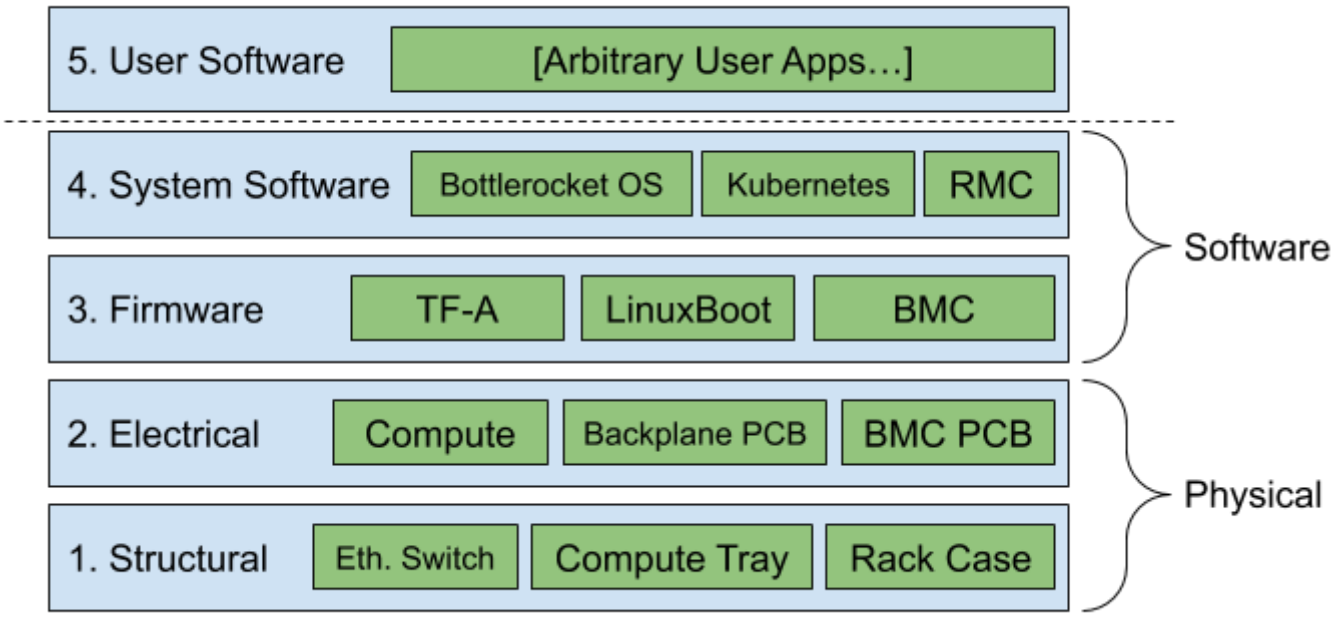
9. **Commodity power and I/O**: The user wants to be able to utilize commodity resources they already have at hand, instead of needing to buy specialized equipment only for Racklet. Examples of these commodities include: Laptop chargers instead of a custom power cable/transformer, USB instead of some proprietary high-speed interconnect, and Ethernet switches & cables instead of e.g. expensive SFP+.

Design Details

The only design detail in-scope for this document is defining the layers of the system at a high level.

High-level Layers

The following section will go through the various layers of the system and the requirements/contract of each of the items.



Layer 1: Structural

This layer of the "stack" consists mainly of the 3D-printed casing and trays of the rack. The Ethernet switch optionally attached on the side of the rack can also be considered a structural item.

Layer 2: Electrical

This layer consists of compute capacity (e.g. a Raspberry Pi with an attached SSD), our reproducible Baseboard Management Controller PCB attached to it in some way (as also can be found in mainstream servers), and our reproducible backplane PCB/wiring which feeds the common busbar power rails, and the SMBus interconnect between compute units.

Layer 3: Firmware

The firmware layer is defined as the code that is running in "bare metal" environments, i.e. on the compute before the primary OS has been loaded, or on the BMC microcontroller. Examples of code that is capable of (and specialized at) running before the primary OS includes the (proprietary) Raspberry Pi firmware, LinuxBoot, u-boot, and TF-A. We strive to use open source Embedded Rust due to the language's suitability for memory safe firmware and good support for most popular microcontrollers.

Layer 4: System Software

The system software includes everything the system needs to run in order to fulfil the user goals. All applications at this layer are built for and depend on Linux. As per above, we expect the user to utilize Kubernetes, and hence there is a default (but replaceable) installation of that. We will also pre-install a (configurable) operating system (OS), e.g. Bottlerocket, so that the user can get going without too much preliminary work. In addition, we will implement the Rack Management Controller features at this level.

Layer 5: User Software

At this layer are the user-deployable workloads running in containers, we consider them as "user-space applications". This layer is not part of the Racklet project, it is entirely user-defined. This is also a good place for users to extend Racklet and add extra functionality of their liking.

Test Plan

Unit tests will be created for individual software components of the system. Integration tests will be created for cross-component communications. Automated end-to-end tests will be conducted by a physical Racklet instance that is continuously "upgraded" to the latest development version and reports feedback. This way we will assure the stability and resilience of the software/firmware stack.

Furthermore, we will rely on the developer community to test out many different hardware, software and firmware combinations other than the reference implementation.

Graduation Criteria

For this project to be considered successful and graduated, we mandate the following:

1. There is a vibrant open source community around Racklet
2. Racklet fulfils all of the above mentioned User Goals to a sufficient degree (as determined by the RFCs addressing respective functionality)
3. It has been end-to-end tested and verified working by following the documentation by someone external to the core contributors team
4. It has been successfully used for educational purposes, e.g. in a university course and/or a workshop

Implementation History

1. `2020-12-10` : First version of this RFC has been accepted.
2. `2021-06-07` : Values have been refined, misc. clarifications and readability improvements.
3. `2021-06-08` : All values, subvalues and user goals have been given IDs for referring to them.

---

[1] See "8 ways the cloud is more complex than you think | CIO." (https://www.cio.com/article/3430760/8-ways-the-cloud-is-more-complex-than-you-think.html) and "Cloud Computing, Once Loved For Its Simplicity, Is Now A Complex Beast." (https://www.forbes.com/sites/joemckendrick/2018/09/12/cloud-computing-once-loved-for-its-simplicity-is-now-a-complex-beast/#4fbb3641747c) (accessed Dec. 05, 2019).

[2] "KubeCloud: A Small-Scale Tangible Cloud Computing Environment". Master's thesis in Computer Engineering at Aarhus University by Kasper Nissen and Martin Jensen. Published June 6th, 2016. Download PDF here (https://github.com/KubeCloud/thesis/raw/master/master.pdf)

[3] For example the 1st stage bootloader of the Raspberry Pi 4 (https://github.com/raspberrypi/rpi-eeprom) is currently closed source software which we cannot audit or modify, and hence cannot use as a "complete end to end" hardware root of trust. However, such non-idealities don't stop us from getting as close as possible to full hardware root of trust, and more importantly, conceptually being consistent in the way we work with these SBCs and "normal" servers.

[4] At least initially this does not mean that the system is 100% secure, there are both some practical limits[3] and software/hardware features that need to be explored for improved security (e.g. ARM TrustedFirmware (https://www.trustedfirmware.org/)).