RFC-0002: Layer Architecture

Collaborate on HackMD

(for docs.racklet.io)

RFC Metadata

Authors (in alphabetical order):

- Dennis Marttinen, @twelho
- Lucas Käldström, @luxas

Status (as defined here): `Provisional`

Creation Date: `2021-04-15`

Last Updated: `2021-07-02`

RFC Handle: `0002-layer-architecture`

Initial Pull Request: racklet/racklet#20

Tracking Issue: racklet/racklet#21

Summary

This RFC describes the overall Racklet architecture, its defining layers, and requirements for each such layer, derived from RFC-0001. For each layer the defining components are described at a high level (avoiding implementation details). The components are associated with their role and five highlighted key requirements from the values and user goals of RFC-0001.

Motivation

With this RFC we aim to clearly define the layers Racklet consists of to provide a clear overview of the system for all contributors and maintainers. Additionally this document concisely presents the techniques and technologies used in the various layers to achieve the goals stated in RFC-0001.

Goals

- Define well-known layers of Racklet.
- Describe the requirements for each layer.
- Briefly discuss "Racklet conformance" from an architectural perspective.
- Highlight some differences between Racklet and other similar alternatives.

Non-Goals

- Describe the details and/or technical implementations of the various layers. See the detailed RFCs for the layers if looking for that information.
- Cover every minor component or implementation-specific components, this RFC is designed to only give an overview.

Proposal

Racklet is divided into 5 distinct layers, from highest-level to lowest-level:

5. **User Software**
6. **System Software**
7. **Firmware**
8. **Electrical**
9. **Structural**

There is some overlap between these defined layers, mostly due to individual components contributing to multiple layers, but we aim to keep a clear distinction in this definition. If for example a microcontroller is part of both the electrical and firmware layer, the electrical layer only considers its electical properties and the firmware layer only its firmware.

The architecture is designed with the layers and their interaction as the primary focus. The requirements of a layer drive the design of the layer below it, which aims to satisfy the dependencies according to the values and user goals of the project. The layers are described here in reverse order (layer 5 first), since the highest layer starts the dependency chain by directly fulfilling the user goals.

5. User Software layer

Summary: The user software layer should allow the user to schedule workloads of choice using either containers or VMs. There should be an accessible and observable graphical user interface in place for the user to monitor and manage the Racklet system and workloads.

Goals:

- Enable the user to observe and manage a Racklet cluster
- Enable easy deployment of container/VM workloads
- (Optionally) make a Kubernetes cluster accessible for the user

Layer components:

Component	Role	Key Requirements
Micro Virtual Machine orchestration	Define and run VMs declaratively	Improve status quo, Openness, Declarative management, Documentation, Fast reconfiguration
Kubernetes deployment automation	Consume/use a Kubernetes cluster	De-facto standards, Declarative management, Loose coupling, Upgradability, Utilize Kubernetes
Racklet dashboard	Monitor rack and cluster state, deploy workloads	Security by design, Declarative management, Open source, Portability, Observability

4. System Software layer

Summary: The system software layer is responsible for enabling the container/VM solutions of the user software layer. There should be a hypervisor in place for the virtual machines and a container orchestration solution (Kubernetes) for container workloads. Kubernetes is also leveraged for orchestrating the Racklet rack and performing managemental operations in a declarative fashion.

Goals:

- Support running containers/VMs securely and scalably
- Be fully declaratively configured using version control
- Enable secure communications inside the cluster

Layer components:

Component	Role	Key Requirements
System Kubernetes installation	Run container workloads, perform management	Declarative management, Consistency, Modular design, Portability, Loose coupling
Hypervisor operating system	Run VM workloads, enable kernel-level security	Defense in depth, De-facto standards, Declarative management, Raspberry Pi compatibility, Portability
CNI compliant networking	Network the Racklet cluster compute units	Security by design, No old/insecure protocols, Openness, Observability, End-to-end encryption
GitOps tooling	Declarative management of the Racklet stack	Improve status quo, De-facto standards, Declarative management, Observability, Auto-upgradability

3. Firmware layer

Summary: The firmware helps in securely booting and configuring Racklet compute, for example it is declaratively managed and performs cryptographic verification of payloads to boot. The firmware should also help with collecting hardware observability data and telemetry for monitoring and debugging.

Goals:

- Enable secure access to the declarative configuration in Git
- Verify payloads to be booted by the compute
- Enable debugging and observability of the hardware and compute

- Store keys and signatures for the above layers

Layer components:

Component	Role	Key Requirements
u-root based bootstrap environment	Secure Git access, firmware updates and payload booting	Security by design, Improve status quo, Open source, Secure updates, Zero-trust network boot
BMC (Baseboard Management Controller) firmware	Compute booting and debugging, key and signature storage for software layers	Security by design, No old/insecure protocols, Declarative management, Debuggability, One-time hardware setup
RMC (Rack Management Controller) firmware	Rack hardware control and observability, e.g. fans	Openness, Declarative management, Loose coupling, Observability, Secure updates

2. Electrical layer

Summary: The electrical layer backs the computational, power delivery and physical networking requirements of the compute. It also provides a means to run the firmware on the BMC and RMC (microcontrollers).

Goals:

- Provide computing capacity for the software layer
- Provide power for all components in a Racklet rack
- Provide a physical networking device for the software layer
- Provide a means to run the firmware for the compute and rack

Layer components:

Component	Role	Key Requirements
Compute unit	Run the bootstrap and hypervisor operating systems and compute workloads	Common off-the-shelf parts, Raspberry Pi compatibility, Hot swappability, One-time hardware setup, Physical portability
BMC PCB	Host the BMC microcontroller and deliver power to the compute unit	Open Source, Reproducible PCBs, Modular design, Raspberry Pi compatibility, Energy monitoring

Component	Role	Key Requirements
Backplane PCB	Rack level power distribution and inter-BMC connectivity	Common off-the-shelf parts, Reproducible PCBs, Physical portability, Hot swappability, Upgradability
Network switch	Provides networking for the rack (and cluster)	De-facto standards, Common off-the-shelf parts, Sensible rack cost, Physical portability, Commodity power and I/O

1. Structural layer

Summary: The structural layer consists of physical components that form the structure of the Racklet rack. The structural layer enables Racklet to be compact, modular and easily transportable. The rack consists of a casing that hosts the backplane, network switch and slots for slide-in trays. The compute unit with its storage is attached to modular compute trays, that have matching rails for the slide-in slots in the rack.

Goals:

- Provide a rigid structure for hosting all components
- Enable component hot-swap and modularity

Layer Components:

Component	Role	Key Requirements
Compute tray	Enable mounting of a compute unit in a hot-swappable and modular way	Open source, 3D printed parts, Modular design, Raspberry Pi compatibility, Hot swappability
Rack case	Contain the network switch, a power backplane and multiple compute trays	Open source, 3D printed parts, Modular design, Sensible rack cost, Physical portability

Guide-level explanation

The layer architecture described in the proposals introduces some new named concepts and components. By layer, they can be explained as follows:

5. **User Software**

- **Micro virtual machine (microVM)**: A very light-weight virtual machine that is optimized for low resource consumption by omitting unnecessary features. Racklet uses these to enable low-overhead kernel-level isolation of applications.

- **Kubernetes**: A production-grade container orchestration system for runnign containerized applications across multiple (physical) compute unit. Kubernetes is leveraged for both running applications and managing the Racklet cluster.
- **Dashboard**: A (usually web-based) graphical user interface for monitoring and controlling software/hardware. Racklet incorporates dashboards for accessibility and observability.

4. **System Software**

- **Hypervisor operating system**: An operating system base that is ready to run (micro) virtual machines using e.g. KVM. The operating system used on Racklet compute units should be light-weight and have hypervisor support.
- **CNI networking**: CNI is a standardized way to network containers for example in Kubernetes. It can however be leveraged across physical compute units and VMs as well, as is done in Racklet.
- **GitOps**: A way to declaratively manage Kubernetes (and other components) by storing declarative configuration files in a Git repository. This enables traceability of configuration changes and easy state transitions.

3. **Firmware**

- **Bootstrap environment**: A small integrated Linux kernel and userspace responsible for securely resolving and booting a hypervisor operating system from the network. Racklet leverages u-root for the userspace component, and GitOps for accessing the rack configuration.
- **BMC**: Short for *Baseboard Management Controller*, it is a microcontroller helping with adminstrative tasks on the compute unit level. In the case of Racklet the BMC helps with tasks such as booting securely, verifying integrity and debugging boot issues. The firmware for the BMC will do much of the heavy lifting.
- **RMC**: Short for *Rack Management Controller*, it is an additional microcontroller on the rack level that manages shared resources such as cooling (fans), indicator lights/displays and optionally rack-level power measurement.

2. **Electrical**

- **Compute unit**: the "server" of Racklet, i.e. a Linux-compatible computer in the Racklet rack that runs the VMs/containers/applications in a cluster setup. The reference implementation of Racklet focuses on the SBC (Single Board Computer) form factor for affordability, portability and accessibility.
- **PCB**: Short for *Printed Circuit Board*, both the BMC and the backplane of Racklet are PCB-based instead of having the user hand-wire the components together. Although PCBs and the associated SMD (surface mount device) components are more difficult to work with and require some expertise, due to their good state of accessibility and affordability

nowadays Racklet is ready to take the tradeoff for a considerably more streamlined user experience.
- **Network switch**: As a cluster computer implementation Racklet requires computer networking between the compute nodes. The network switch enables connecting the compute nodes and racks together (and to the Internet). While the type of network switch is not limited here, Racklet aims to only require OSI layer 2 compatibility from the switch. This is why the switch is categorized the electrical layer since that type of switch has no distinct firmware and other "moving parts".

1. **Structural**

- **Compute tray**: The compute tray is analogous to a server casing with rails in a traditional server rack. Since Racklet compute units are mostly of the SBC form factor, they most often have no integrated casing and no way to mount storage (i.e. an SSD/HDD) like a traditional computer case. Thus, Racklet has the compute tray to facilitate exactly that, it is a case (or plate) that mounts a compute unit and its storage and provides "rails" to slide into the Racklet rack with hot-plug support.
- **Rack case**: The physical structure hosting all compute units, PCBs, networking and power (conversion). In the reference implementation this resembles a down-scaled server rack for portability and educational value, but the form factor is not strict.

Note: These RFCs target a "reference" implementation of Racklet, as envisioned by its authors. The components and key requirements for them are described from the perspective of this reference implementation, and thus "community" implementations of Racklet (e.g. in a different physical form factor) don't need to strictly adhere to the requirements laid out here. A "Racklet compliant" system ultimately only required to follow the values laid out in RFC-0001 and the loose coupling hardware/software interfaces of the project. That said, it is still advised that variations of Racklet follow the layers, high-level components and key requirements in this document.

Risks and Mitigations

The Racklet team aims to adapt to community requirements and adaptations to keep the Racklet ecosystem cohesive. The project has three strategies to mitigate against the risk of the ecosystem fragmenting with incompatible hardware/software implementations of Racklet:

1. *Community contributions and suggestions are taken into account and encouraged.*
   - The project adapts to the usecases of its userbase to avoid community implementations steering different directions.
2. *Loose coupling is leveraged to the greatest possible extent.*
   - All components of Racklet shall depend on each other only through standardized interfaces, which enables the use of alternative implementations following those

specifications.

3. *The layer architecture described here is not fixed.*
    - The layers are used to guide the design, but are not fixed bounds that require to be strictly adhered to. For example, a community-made component can be both part of the user software and system software layer without issue. The Racklet team is also open to feedback regarding the layer structure if you have improvement suggestions to the model.

Rationale and alternatives

As stated in Risks and Mitigations, Racklet is (one of) the first of its kind with regards to its specification-first architecture. The initial layer separation presented here is the result of an iterative thought process by the core Racklet authors. The five layers are chosen to clearly separate roles and responsibilities of components, without going into too much detail (too many layers) or causing excessive overlap (too few layers). Firmware and system software are separated to achieve loose coupling and clear, secure communication between them. User software is separated from system software to define a border between software mostly provided by the Racklet project and external software that the user introduces (workloads).

Loose coupling plays a very important role in the architecture presented here. Racklet could have been designed as a fully integrated system with implementations that are strictly defined by the project, but while this potentially could make the system more compact and simple, it also faces many drawbacks that make it incompatible with the values and goals of the project. For example, Racklet relies heavily on various different projects in the Open Source Firmware and Cloud Native ecosystems, many of which evolve quickly and provide alternative implementations complying to standard APIs. We want Racklet to be accessible, transparent and modular, which means supporting a wide variety of hardware, and enabling user customization to a great extent. If loose coupling is implemented properly, we believe that the standardized architecture presented here will be relatively simple to maintain and extend, and community-built Racklet solutions will also be able to use the modules and different software implementations effortlessly. In summary, to fulfill the values defined in RFC-0001 and to avoid ecosystem fragmentation the Racklet project aims to provide interfaces, not implementations.

Prior art

At the time of Racklet creation the history of Raspberry Pi (and other single board computer) based cluster computers is already very rich. Various private persons, educational insistutes and companies have come up with a wide variety of designs (e.g. KubeCloud[1]) for different use cases for at least the past 8 years. What sets Racklet apart from these mostly one-off implementations is it's **specification**. Instead of deriving a specification from some implementation, Racklet as a system is *primarily* defined as a set of RFC documents. This specification is intended to define a **standardized** way to build a miniature compute cluster, from the lowest-level hardware details up to a state-of-the-art software stack. Since the specification is defined from the ground up, we prioritize basing it on the most *secure* and *modern* technologies available today, essentially merging the core concepts of prior SBC cluster computer implementations with the state of the art security and fleet management models of large-scale cloud providers.

Unresolved questions

The architecture described in this document is prone to encounter changes as the detailed RFCs describing individual components/layers are established. It is also unclear if this particular layered architecture with the chosen high-level components is optimal, and thus the reference implementation will likely influence the structure here once it is better known what works and what doesn't.

Racklet is also a complex system, and this document in its current state can likely not provide the full picture of the architecture to an unfamiliar reader. To combat this, additional graphical elements such as architecture diagrams could be embedded into this document in a future revision (**TODO**).

The concept of "Racklet conformance" briefly disussed in Risks and Mitigations is not expanded upon here, but might warrant its own RFC specifically for community implementations.

Future possibilities

The layer definitions presented here are expected to evolve with the project. This document serves as a starting point for discussion, and records the current consensus. In the future the scope of this document might also include a thorough introduction to the architecture for newcomers to the project, as well as improved reasoning for particular high-level architectural decisions and how they are derived.

Implementation History

- `2021-07-20`: This RFC has been accepted.

---

[1] "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).