Architecture Analysis and Implementation of an Adaptable Satellite Sniffing Network

Martin Böh

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Bachelor Thesis

Architecture Analysis and Implementation of an Adaptable Satellite Sniffing Network

by

Martin Böh
contact@martb.dev

Grade: 1.0

29.03.2022

Technische Universität Kaiserslautern
Department of Computer Science
Distributed Systems Lab

Supervisor: M.Sc. Eric Jedermann
Examiner: Prof. Dr.-Ing. Jens B. Schmitt

Abstract

With the ever-increasing amount of satellite-backed communication, constellations covering the entire world, and the rise of Software Defined Radios (SDRs), satellite signals have already become prime targets for scientific research all over the globe. However, due to logistical challenges like capture time/location and peripheral/system management for the sensors and the wide variety of protocols/encoding schemes used, no one-fits-all sniffing solution exists for capturing their wide variety of signals. Therefore, this thesis aims to analyze, design, and implement a system that makes it possible to study LEO (Low Earth Orbit) L-Band satellite signals with readily available Single Board Computers (SBCs) in a widely distributed, location, and time-aware way. The key design factors were useability, maintainability, adaptability, and security in a centrally managed client-server architecture. The research presented yielded a Satellite probe Operating System called SATOS, which aims to implement on-sensor data decoding driven by GNU Radio and secure Over The Air (OTA) updates inside the Buildroot build environment. Its intended use case is the future deployment of DISCOSAT on a university working group scale.

Auszug

Durch die ständig wachsende Anzahl satellitengestützter Kommunikation, Konstellationen, die die ganze Welt abdecken, und dem Aufkommen von Software Defined Radios (SDRs), sind Satellitensignale bereits zu Hauptzielen für die wissenschaftliche Forschung auf der ganzen Welt geworden. Aufgrund der logistischen Herausforderungen wie Erfassungszeit/Ort und das Peripherie-/Systemmanagement für die Sensoren sowie der großen Vielfalt an verwendeten Protokollen/Codierungsschemata gibt es jedoch keine Universallösung für die Erfassung der großen Vielfalt an Signalen. Daher zielt diese Arbeit darauf ab, bestehende Systeme zu analysieren um mit dem gewonnenem Wissen ein neues zu entwerfen und zu implementieren, welches es ermöglicht, LEO L-Band Satellitensignale mit leicht verfügbaren SBCs in einer weit verteilten, orts- und zeitabhängigen Weise zu untersuchen. Die wichtigsten Entwurfsfaktoren waren Verwendbarkeit, Wartbarkeit, Anpassungsfähigkeit und Sicherheit in einer zentral verwaltete Client-Server-Architektur. Die vorgestellte Arbeit führte zu einem Satellitensonden-Betriebssystem namens SATOS, das darauf abzielt, die Dekodierung von Sensordaten durch GNU Radio und sichere Over The Air (OTA)-Updates innerhalb der Buildroot Umgebung zu implementieren. Der vorgesehene Anwendungsfall ist der zukünftige Einsatz von DISCOSAT im Rahmen eines Universitätsarbeitsgruppenprojekts.

Dedicated to my family, my cat Ben and all friends
that helped me on my way to this thesis.

Contents

List of Figures

List of Tables

Listings

1. Introduction

With the availability of affordable SDRs like the HackRF One, analyzing satellite signals at a low cost is now becoming a reality. However, due to the wide variety of new research possibilities, it is necessary to adopt a more solid satellite transmission capturing approach when studying time and location-dependent signal effects. The significant challenges to overcome include the need for an independent Adaptable Satellite Sniffing Network (ASSN) that can perform preplanned capture tasks independently without streaming data back in real-time, alleviating the need for constant connectivity and allowing the use of wireless network connectivity and battery-powered deployments. Moreover, feature- and security updates are also required for the probes. Ideally, such an ASSN should feature easily deployable sensors with a solid software stack, permitting users to focus on the scientific research data and not on the complexity of the data acquisition, entirely abstracting it away.

1.1. Objective

This thesis is part of a satellite sniffing research project conducted at the Technische Universität Kaiserslautern Department of Computer Science Distributed Systems Lab (DISCO). The main focus of work lies in the sensor (client) architecture and its corresponding implementation. Furthermore, at the time of writing, the project is still under active development, and therefore no final evaluation on practical, real-world implementation scenarios can be provided yet. Hence, work will start from the ground up by factoring in the core operating system tooling and tweaks needed to get all aspects covered.

1.2. Motivation

This section justifies the choice of critical parts of this thesis through the use of feasibility examples and under-researched topics encountered during the DISCOSAT design process.

Data Volume When talking about raw satellite signal capturing done by an SDR, *I/Q* samples are what gets sent to the host computer. These samples consist of an in-phase (I) and quadrature-phase (Q) sample of the same signal. These components are needed to recover amplitude and phase information from the received signal for later demodulation [1]. For the HackRF One, each *I/Q* sample consists of two separate integers with a resolution of 8-bit [2]. Consider the following real-world scenario to get an intuition for the network bandwidth needed to stream this data in real-time, which helps narrow down possible system designs.

Iridium® L-Band The Iridium satellite constellation (Iridium®) L-Band links operate at 1,616 - 1,626.5 Mhz and use Quadrature Phase Shift Keying (QPSK) as their modulation scheme [3]. The main channels in use are between 1,616 and 1,626 Mhz, with an additional ring alert covering up the last 500 kHz [4]. Due to the 10 MHz bandwidth covering the main channels and the usage of quadrature sampling by standard SDR receivers [5], applying the Nyquist–Shannon sampling theorem yields a needed minimum sampling rate of $R_{sample} \geq 10$ MS/s. [1].

Thus if each sample consists of I and Q data with a resolution of *8-bit* the size of one *I/Q-Sample* is precisely $L_{IQ} = 2 \times 8bit = 16bit$ and therefore the amount of uncompressed raw data streamed when capturing Iridium® with the above theoretical settings is $R_{iridium} = L_{IQ} * 10000000\ hz * 1 = 20\ MBps$.

If the above capture runs for 8 hours a day, this will accumulate to $L_{total} = 20\ MBps * 28800s = 576\ GB$ of data streamed. Unfortunately, transmitting or storing this large amount of raw data for centralized processing is not feasible due to mobile network data caps in the low double-digit GB range and device storage performance constraints due to the ordinary SD-Cards used in embedded boards.

So how to ideally design payload transmission and data capture in a ASSN to offload as much work as possible to the sensor and therefore reduce data flow?

[1] In a real-world scenario, higher sampling rates should be used to combat filtering issues inside the SDR. For a hypothetical example, it is sufficient.

GNU Radio As a way to fulfill their signal processing needs, research projects in the SDRs field often directly deploy GNU Radio [6] on suitable embedded devices [7] [8]. However, given its target dependency graph in Figure 1.1 and its direct dependency on boost [2], and python3 (in case of the enabled python API), it is safe to classify GNU Radio as a large and complex project. Furthermore, when dealing with constellations like Iridium® or different SDR hardware, custom plugins like *gr-iridium* used to capture Iridium® [10], and *gr-osmosdr* used for SDR hardware support also extend this graph further, as seen later on.

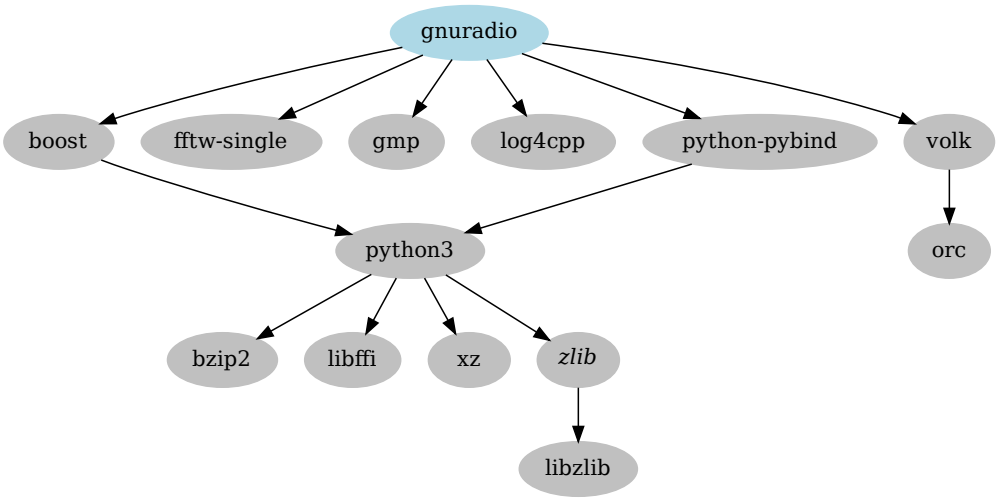


Figure 1.1.: GNU Radio dependencies (excluding system- and build-time)

Operating Systems Throughout the research for this thesis, it became clear that little to no information about the system software used in "Demonstration Abstract: OpenSky: A Large-scale ADS-B Sensor Network for Research"[11] and "Low Cost Nano Satellite Communication System Using GNURadio, HackRF, and Raspberry Pi"[7] is provided. Therefore, it is assumed that the authors used custom-baked software solutions on top of the factory-provided operating systems for embedded devices like Raspbian in the case of the Raspberry Pi embedded board or a proprietary one running on fixed-feature devices.

However, with additional ASSN requirements like WiFi and LTE backhaul for mobile communication, GNSS for position tracking, custom GNU Radio plug-

[2] A collection of peer-reviewed portable C++ source libraries for (network) I/O, threading, and more. [9]

ins for on-device processing, and project longevity, this could turn out to become a logistical management challenge. Moreover, entirely standard Linux distributions like Fedora Core or Ubuntu used in projects like "KUAR: A Flexible Software-Defined Radio Development Platform"[8] require a stronger focus on the underlying operating system configuration and its needed modifications, yielding the following fundamental research question. Which system base is suitable for use and covers the required packages and tools for an ASSN in an adaptable and maintainable way?

1.3. Outline

First, the fundamentals needed to understand the DISCOSAT project and the real-world constraint environment that applies to it will be provided. Furthermore, a literature review will be conducted on existing projects that help answer the core questions outlined above. Moreover, some intuition on which aspects used in these projects were adopted or improved upon in DISCOSAT will be provided, keeping the encountered limitations in mind. Said literature is going to establish the base needed to understand the choices made for the design and implementation of DISCOSATs architecture. Then, the architecture design for DISCOSAT will be described, and implementation specifics for the system- and peripheral management will be provided. Lastly, a brief analysis based on SATOSs useability in a real-world environment will be performed. The broader scope for this analysis contains maintenance tasks like adding new packages, deploying firmware updates, and more general system operations.

2. DISCOSAT fundamentals

This chapter is the base for the literature analysis and provides the guidelines for further development. Moreover, it is going to outline the essential responsibilities and requirements for a functional ASSN called DISCOSAT which is programmed but not limited to analyzing the traffic of Iridium®.

2.1. Goals

Sniffing packet-based signals from Iridium® in remote locations during a given time window, decoding them, and later uploading the decoded data to a remote server. Said server has to be able to store the data so researchers can later access it. One example use-case would be verifying the active spot beam patterns from Iridium® by analyzing decoded protocol and metadata from sensors in different regions. Remote management is limited to OTA-update support through the use of secured firmware binaries.

2.2. Stakeholders

The stakeholder analysis in Table 2.1 plays a role in understanding the involved parties. All parties besides the sensor hosts are primary stakeholders as they have a direct interest in the collected data and the project's success. Moreover, volunteers are secondary stakeholders, as they only provide replaceable infrastructure to the project. However, some double assignments are to be expected, especially in the case of a researcher volunteering to host a sensor.

	Role	Assumptions and Risks	Impact
Network Owner	Owner	- Provides and maintains the network - Has fundamental embedded development knowledge - Distributes sensors to volunteers - Plans new hardware / software revisions	High
DISCO researchers	System user	- Need access to collected satellite data - Must be able to specify new data collection jobs - Provide feedback for missing tooling - Test future software updates - Do not care about the "how", just need the data	High
Sensor hosts	Volunteer	- Provide power and location to set up the sensor - Can drop out at any time	Low
Other researchers	System user	- Access to historical satellite data - Can submit requests for new data	Medium

Table 2.1.: DISCOSAT Stakeholders

2.3. Requirements and constraints

To provide a realistic example scope for this thesis, some high-level requirements and constraints are given in this section.

2.3.1. Maintainability / Adaptability

The system needs to be maintainable and adaptable by non-industry professionals. This includes the work required for new firmware updates and updates for the infrastructure used in the project. In addition, every component used should be understandable after some basic training.

2.3.2. System requirements

Sensors

1. can be deployed all over Germany
2. guaranteed to have a working power source at all times
3. need to have network connectivity, including backup WiFi or LTE to upload the captured data and receive tasks reliably
4. need time synchronization (sub-second precision)

5. need GNSS support for accurate location reporting and as *offline* time source
6. need to be securely updateable remotely

Backend

1. ability to set sniffing job type, capture start- and end time, and metadata [1]
2. ability to manage participating devices (firmware updates, status)
3. ability to save decoded Iridium® frames or other payloads transmitted by the sensors
4. provides an interface to access captured data for researchers

2.3.3. Hardware constraints

The following hardware is used in DISCOSAT. It provides a solid base for sniffing parts of the spectrum from LEO satellites like Iridium® successfully. This constraint is primarily influenced by availability and cost, as no other similarly priced hardware was readily available while writing this thesis.

Sensor

1. Raspberry Pi 3B+ (4x 1.4 GHZ, 1 GB RAM) AArch64 (64-bit ARM cpu architecture) [2]
2. SIM7600E-H 4G HAT for Raspberry Pi, LTE Cat-4 4G / 3G / 2G, GNSS including antennas.
3. Great Scott Gadgets HackRF One SDR including a TAOGLAS IAA.01 Iridium Antenna
4. Miscellaneous (power supply, additional antennas, USB-Sticks, SD-Cards)

[1] More advanced job data like center frequency, bandwidth, and device-specific settings
[2] The faster successor Raspberry Pi 4 was originally planned, but was not available.

Backend

For the backend, the only available hardware is a Linux-powered virtual machine in the university network with a public IP-Address and the possibility to open network ports and install custom software on demand.

2.4. System overview

The high level system component overview is described in Figure 2.1. The sensors will be distributed and connected to the server using ordinary internet links with the help of basic access technologies like Ethernet, WiFi or LTE.

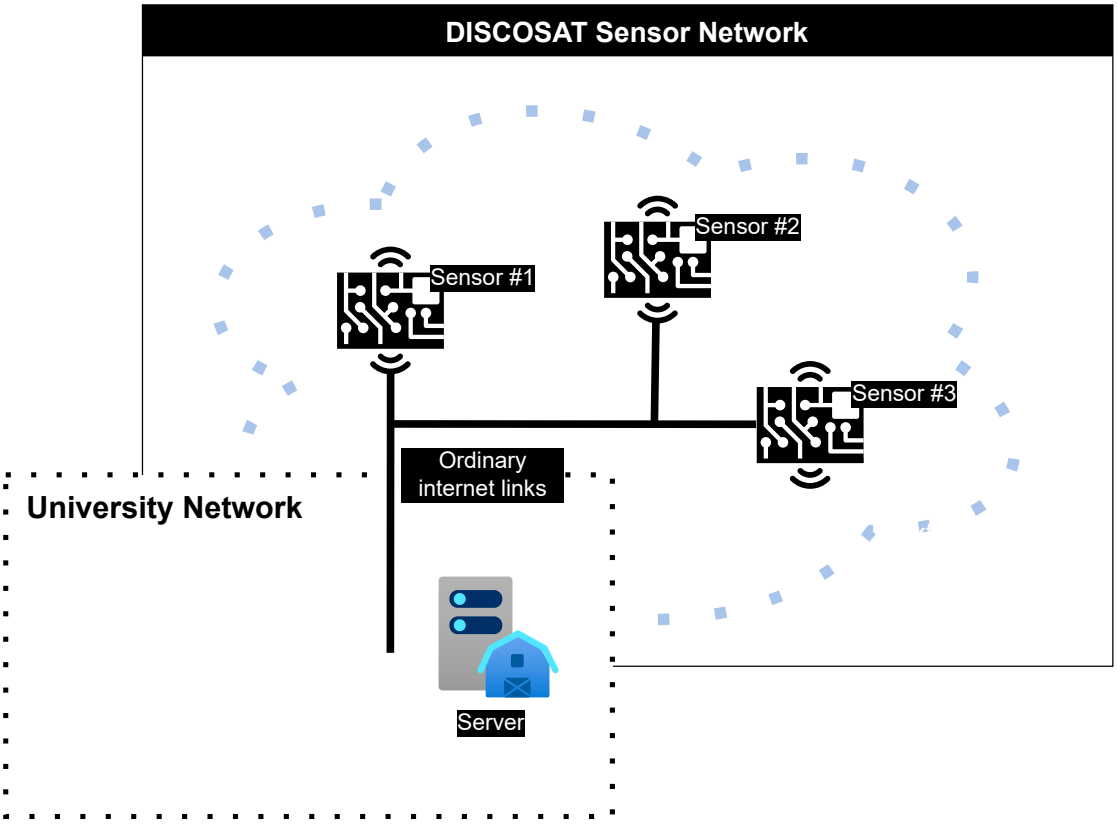


Figure 2.1.: DISCOSAT high level system overview

3. Literature Review

In this chapter, existing solutions in the context of **Distributed Computing Projects** and the **Operating Systems** used in related fields are analyzed. They serve as an aid in determining the amount of work needed to answer the research questions of this thesis. The general motivation is re-using and only slightly modifying existing system designs and implementations for all aspects of DISCOSAT.

3.1. Distributed Computing Projects

Some proven distributed computing projects with centralized management supporting a wide variety of configurable work tasks for client devices exist. Notable ones will be presented based on the following key questions and compared with the approaches used for DISCOSAT. Furthermore, the required adaptations and iterations of the designs for use in DISCOSAT will also be provided.

System Architecture

What is the system architecture used in the project?

Work Distribution and Results How does task distribution to client devices work? How do the obtained results get delivered, and which data do they contain?

Clients

What is a client in the project?

Payload Are raw scientific data payloads being computed by a fixed toolchain on the device, or is an additional Compute Core (CC) or similar on-demand downloading approach used that contains the required processing logic?

Computation What are the rough layouts of the payloads and computation-related data, and how is the computation carried out?

Security Which mechanisms for preventing attacks like Man In The Middle (MITM) and general data integrity issues are in use?

Supported systems Which operating systems and CPU architectures are supported, and how is adoption for other systems handled?

3.1.1. Folding@Home

F@h is a distributed computing project released in the year 2000. It aims to provide computational power for various mostly bio-informatics-related subprojects through "Volunteer Computing (VC)" [12]. The use of F@h in this thesis can be seen as a representation for similar distributed computing projects like Berkeley Open Infrastructure for Network Computing (BOINC), as they exhibit similar core functionality in our narrowed down analysis scope.

System Architecture

As shown in Figure 3.1 F@h uses a client-server architecture. The multiple single-purpose servers exist for scalability and fault tolerance reasons as F@h is a large-scale project with more than 400.000 clients [12] exceeding a total compute capacity of 100 PetaFLOPS (Floating Point Operations Per Second) in the year 2016 [13].

The multi-server architecture allows F@h to alleviate downtime of single servers and attempts to guarantee successful result delivery as there is no single point of failure in the design. Therefore, making this approach a viable candidate upon further expansion of our DISCOSAT server infrastructure.

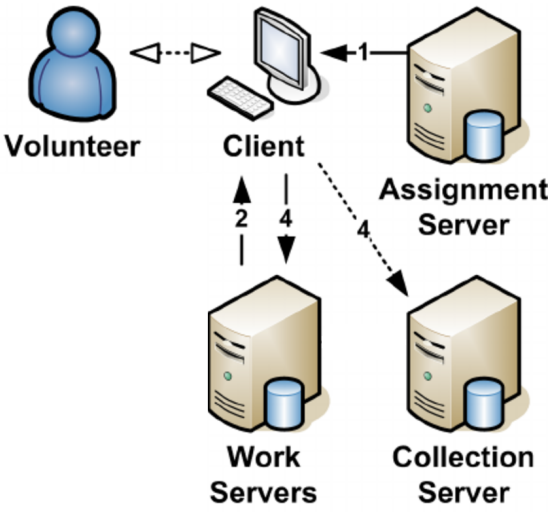


Figure 3.1.: Modified F@h architecture[1], arrows show dataflow, based on [12]

Work Distribution and Results In F@h tasks are called *Work Units (WUs)* and are exclusively planned by an *Assignment Server*. Upon assignment (1), clients download the *WU* and, if needed, an additional *computational core* from the *Work Server* listed in the assignment payload (2). When finished with the work, clients either deliver the results back to their original *Work Server* or attempt to reach a fallback *Collection Server* in case the former is unreachable (4). With this, the WU is considered complete, and the next one can be assigned. As F@h primarily uses GPU and CPU intensive calculations, multiple WUs can be assigned and worked on in parallel if the assigned client system resources allow it [12].

Albeit not using multiple servers for fault tolerance, DISCOSAT adopts the significant components for work distribution and result sharing needed in a distributed computing project. Moreover, F@hs work scheduling, and processing functionality can be considered state-of-the-art since only slightly different implementations are in use in other large-scale distributed computing projects like BOINC [14]. Furthermore, neither does DISCOSAT need support for multiple tasks at the same time, nor is it planned to be deployed on a large enough scale to warrant various independent servers.

[1]Steps 3 and 5 of the original are not of concern, as they are related to the credit system used in F@h

Clients

Clients in F@h are software-based and get installed by volunteers on ordinary computer hardware like desktops, laptops, tablets, smartphones, and even game consoles all over the world [12]. The client component called *bastet* is primarily written in the C++ programming language. Its source code is available on GitHub under the GPL-v3.0 license [15].

In DISCOSAT the entire hardware and software stack can be considered part of the client, as APOGEE needs to handle auxiliary system setup and peripheral management. However, *bastet* only deals with computation and other work-related tasks.

Payload The raw payload-data transmitted inside WUs is sub-project specific and is later being worked on by clients using the specified *CC* in the WU metadata.

Computation F@h deploys the aforementioned CCs that package the tooling and processing code required for the client to work on one or more assigned WUs. These cores are platform- and operating system dependent and get delivered as binary files for later execution by the client. They come with metadata fields for the identification and verification of the core. This metadata approach allows having multiple revisions of the same core or specific cores for different target architectures. During the setup of WUs, authors can choose the required CCs for their data. Due to its flexibility, F@h needs to rely on the on-demand download of additional software as the client computer does not belong to them, and preloading every required package is not feasible on volunteer machines.

In an attempt to simplify and improve upon the maintainability and data intensity of the CC approach used by F@h, DISCOSAT is relying on payload-less WUs to achieve a higher level of system architecture independence and alleviate the need for additional downloads. Furthermore, the WUs execute pre-installed software on the sensors, made possible by the used operating system SATOS providing all the tooling and processing capabilities ahead of time directly on the device.

Security As seen in Listing A.1 all payloads like *WUs* and *CCs* are cryptographically signed. Their corresponding signature gets checked against a

well-known Certificate Authority (CA)-File distributed with the client upon retrieval, making sure that an authorized person created the package. This mechanism prevents unauthorized payloads from being executed on a volunteer's machine, preventing malicious malware deployment as a result of attacks like MITM as long as the private keys used are kept secure. Moreover, as an additional data measure, payload integrity gets verified by SHA-256 checksums, preventing malformed payloads from crashing the client software or, worse, the system in the case of dangerous buffer under-/overruns.

For DISCOSAT, the same Public Key Infrastructure (PKI) powered security and integrity approach outlined above is planned, as the security measures implemented by F@h are considered sufficiently advanced at the current time.

Supported systems F@h clients are available for multiple different systems like Windows, Linux, Mac OS, and architectures like x86, x64, PowerPC. However, not all projects are executable on all platforms [12], even though WUs are architecture-independent, the *CCs* are not. Still, as briefly mentioned before, *CCs* are adaptable for different target architectures by porting the required processing software stack and packaging it. Allowing authors to add new target systems on the fly if the F@h client supports the device.

For DISCOSAT, the operating system SATOS is the core component and runs on a fixed set of needed architectures. On the other hand, F@h does run on all major operating systems and does not provide any custom operating system base, not yielding any pointers for analysis.

Takeaways

F@h does have a broader scope than DISCOSAT as it does heavily distributed computation on existing chunk-data, supporting all major operating systems since only CPU and GPU compute power is required as a resource. For DISCOSAT, real-time data retrieval on a fixed set of architectures with specialized peripherals is needed. Therefore, the implementation choices differ slightly, and the goals diverge, but the general task scheduling and security aspects presented above are reusable and sufficiently advanced. With the usage of lighter WUs and the possibility to control the target systems as a whole, multiple new ways of improving project maintainability and data intensity are achievable. Furthermore, DISCOSAT requires special attention in the fields of system management like network connectivity and peripheral management (GNSS or SDR hardware), as they are required for successful and reliable data capturing, a field not touched upon by F@h.

3.1.2. RIPE Atlas

RIPE Atlas is an open-data distributed measurement platform for internet metrics, including latency and internet reachability. After being established in 2010 by RIPE NCC (a not-for-profit Regional Internet Registry (RIR) for multiple internet service regions, including Europe), [16] it now features over 11,000 connected probes all over the world, delivering over 13,000 measurement results per second [17]. In addition, Atlas deploys off-the-shelf hardware like modified routers and embedded boards preloaded with custom firmware based on BusyBox[18], making it a candidate for system-related research.

System Architecture

Albeit being ten years newer, Atlas still uses a classic client-server approach, just like F@h. Its core components can be seen in Figure 3.2. The infrastructure heavily uses Message Queuing (MQ) [16], but this technique and the nodes upstream of the *brain* are of no concern for this thesis, as its focus lies on the client system architecture. Therefore, only the probe-to-controller, probe- to-registration server communication, and the measurement-scheduling by the *brain* is relevant.

Atlas improves upon certain aspects like message buffering and fault-tolerance, but no improvements exist for the high-level client system architecture analyzed in this thesis.

Work Distribution and Results After connecting to the *registration servers* seen in Figure 3.2 the probe network geo-location gets analyzed and used to point it to an adequate *controller*. Said *controller* then is in charge of talking to the connected probe and, based on location and capacity, assigns it to requested measurements and performs system management. The controllers themselves get managed by the *brains* which coordinate the measurements and process part of the collected data.

New work tasks get transferred to the client by *controllers* using a Secure Shell Protocol (SSH) port forward exposing a telnetd daemon running on the probe to the assigned *controller* [19]. Said protocol gets used to invoke the on-device measurement system at the specified time, which starts the computation of the tasks. Once finished, results are delivered back to the upstream *controller* using a single HTTP JavaScript Object Notation (JSON)-reply for each measurement

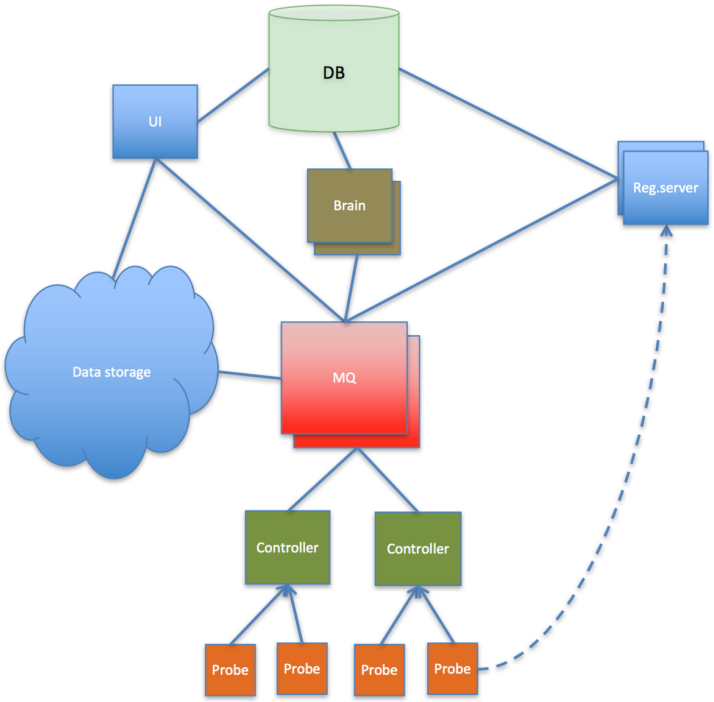


Figure 3.2.: RIPE NCC Atlas architecture, arrows indicate dataflow [16]

containing the *firmware version* of the device and the collected raw data from the measurement-tool run [20].

Atlas uses an interactive no-payload, only instructions approach powered by telnetd. For DISCOSAT, the no-payload strategy was adopted as much as possible for specific job types. For example, a periodic-polling approach does not rely on long-lived tunnel or web-socket-based real-time connectivity to save data. Moreover, a similarly designed result delivery mechanism is used, but as DISCOSAT also needs binary result data just like F@h a hybrid approach is required.

Clients

There are two types of clients in the Atlas project *Software-* and *Hardware Probes*. Both deploy the core measurement framework *eperd* which is based on crond (Command Run On Daemon) [21]. However, *Software Probes* are installable on volunteer controlled devices whereas *Hardware Probes* are exclusively controlled and managed by RIPE NCC. *eperd* is written in C and integrated into a BusyBox tooling environment [18]. All measurement related projects (*ripe-atlas-*

probe-busybox, *ripe-atlas-software-probe*) are available on GitHub under the GPLv2 license [20]. Not a lot of details about the firmware exist, but the *ripe-atlas-software-probe* repository contains scripts (*bin/arch/openwrt-atlas-probe-v[3-5]*) for building an OpenWRT (Open wireless router) based image.

Atlas also deploys hardware probes as clients, lining up closer with DISCOSATs goals. Furthermore, the usage of BusyBox indicates the existence of a custom operating system image [18]. With the additional information obtained above, it seems likely that the OpenWRT build environment is in use for building the probe operating system, bridging the gap to the Buildroot toolchain used in DISCOSAT.

Payload Atlas does not use conventional payloads in the sense of F@h WUs, making it payload-less for measurement tasks.

Computation Atlas uses a plaintext-based protocol for managing and triggering measurements, as the annotated excerpt in Listing A.2 shows. However, the format is not easily readable by humans. The text lines contain program invocations with parameters that get parsed by *eperd*. They have all the needed information like time, target address, and the specific tool command line to start the computation. Moreover, no CCs or similar techniques get used, as all the processing is happening directly on the probes.

The protocol used in Atlas that invokes arbitrary cron commands is unadaptable and hard to maintain due to its loose fixed-purpose structure. Therefore, a more descriptive representation with different task types for the jobs is used in DISCOSAT. The computation makes use of pre-installed on-device software closely lining up with the processing approach chosen for DISCOSAT

Security The transport layer security provided by the established SSH tunnel between the *controller* and the telnetd instance on the probe offers strong protection against MITM-Attacks and unauthorized access. Moreover, Atlas uses additional command sanity-checking logic, preventing the creation of arbitrary files on the device and protecting security-critical operations like firmware upgrades through message-digest hashes and signature checks. [22]

DISCOSAT also relies on additional transport layer security. Moreover, the firmware updater also enforces similar state-of-the-art signature checks to prevent unauthorized updates.

Supported systems Atlas was primarily supposed to run on embedded hardware handed out to volunteers by RIPE NCC. Nowadays, multiple OpenWRT based hardware probes exist in Atlas, most notably the v5 revision based on the Turris Mox router [2]. Moreover, *software probes* are also available for major Linux-based operating systems like CentOS and Debian. Additionally, *Docker* images also exist, making the installation possible on all architectures supported by it, providing access to a vast arsenal of potential target systems. Porting the software to other architectures in the case of *hardware probes* is achieved through OpenWRT build-scripts, as they feature the required cross-compilation toolchain for building the customized BusyBox package that contains the *eperd* measurement framework. For *software probes* only the latter part concerning the *eperd* framework is relevant.

DISCOSAT uses Buildroot, the base behind OpenWRT, a less router-software-inspired build environment. Additionally, the project-specific build scripts used in Atlas are not reusable for DISCOSAT. Instead, a more maintainable approach based on Buildroot's best practices is used to achieve a saner and well-defined repository structure.

Takeaways

The scope of the RIPE NCC Atlas project is firm in the range of network-related measurements. Therefore, all required parameters for data collection like time, target, and tools are adjustable remotely. The protocol used is plain-text based and something that needs more flexibility for usage in DISCOSAT. Additionally, suppose an additional mechanism is necessary. In that case, the measurement-framework *eperd* integrated into BusyBox needs an update, indicating the need for *hardware probe* firmware update creation through the OpenWRT build environment. No further details exist, so the general mechanisms used to create such an update are subject to further analysis. Just like F@h, Atlas relies on PKI, and digest-checksums for its security needs embodying the current state-of-the-art.

[2] See Atlas probe v5 device scripts commit message at `https://github.com/RIPE-NCC/ripe-atlas-software-probe/commit/d08a06496e3dc01843eed46667163a2918e4732f`

3.2. Operating Systems

As seen in the previous analysis of related *Distributed Computing Projects*, the operating system details are mostly out-of-scope, as they either run on multiple existing platforms like Windows / Linux / macOS or, in the case of Atlas, only require application-level adjustments. Furthermore, if mentioned, the build systems and toolchains used for embedded device development and the necessary production readiness work are primarily left unexplained. Therefore, this section will provide information on the core systems encountered in other projects capable of running on-device processing software stacks based on GNU Radio.

3.2.1. Limitations

This section will provide information on the limitations found during the research of this thesis. For example, some analyzed projects did not meet the scope of DISCOSAT and were discarded. Others sufficiently overlapped with other more suitable approaches and were therefore not evaluated to the same degree.

Desktop Linux distributions

Operating systems for general-purpose use like Debian, Fedora, Ubuntu, ArchlinuxARM, which support different architectures like AArch64 are not suitable for use in DISCOSAT. Their overall scope and the package-manager-based update designs make them unsuitable for use in an embedded device, as network traffic and filesystem size must be minimal. Moreover, in the case of Debian and others, no readily available reproducible build support exists, as past package versions are deleted from the primary upstream servers regularly and need to be fetched from snapshot mirrors instead [23]. Furthermore, the amount of manual work required to get the build environment for the system and the target device running depends on the exact project implementation and can only be evaluated practically. All in all, desktop Linux distributions do not align with the maintainability targets set for DISCOSAT, as modifications are not stored in an easy-to-understand and reproducible way.

Yocto Project®

Much like Buildroot [24], the open-source Yocto Project® founded in 2010, enables developers to create an embedded operating system from scratch. Unfortunately, albeit showing satisfying results during research, the overall implementation complexity due to its non-simple layer approach [25] and the inherent steeper learning curve made it unsuitable for practical evaluation. Hence, it does not meet ease-of-use and maintainability requirements when used by non-subject matter experts, making it more suited for specialty tailored commercial device fleets deployed on a larger scale. Fortunately, its significant feature overlap with Buildroot made it possible to focus on this type of build system instead.

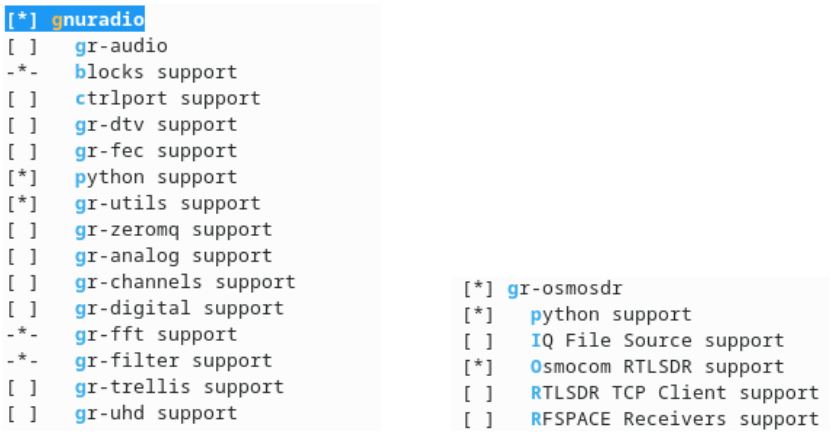
3.2.2. Buildroot

"Buildroot is a tool that simplifies and automates building a complete Linux system for an embedded system, using cross-compilation."[26] Its development started in 2005, making heavy use of Make, Python, and shell-based scripts. Buildroot is maintained in a public GIT repository and is available under the GPLv2 license [27]. It aims to provide reproducible builds and the most miniature root filesystem possible for a particular project and allows tuning every step in the build process.

Provided Packages

Buildroot version *2022.02 LTS* currently comes with hundreds of packages including a dedicated section of GNU Radio 3.8 related packages [28] as seen in Figure 3.3. Unnecessary packages can be disabled if they serve no purpose in the build. More use-case specific plugins like *gr-iridium* are non-existent but can be added. Albeit without support for the HackRF One used in DISCOSAT, the *gr-osmosdr* library needed for common SDRs operations and data capturing in DISCOSAT does exist. For hardware and service management, recent versions of the *Linux* kernel, the *U-Boot* bootloader and *systemd* are available out of the box.

Customizability and Maintainability Buildroot uses a simple text-based makefile approach, which makes it heavily customizable. Furthermore, it supports additional changes and new packages in an external tree that stacks on top of the one provided by Buildroot [26]. The ability to place custom files on the



(a) Core GNU Radio packages

(b) *gr-osmosdr* SDR support packages

Figure 3.3.: Buildroot GNU Radio related packages in 2022.02 LTS

filesystem of the target devices also exists through the use of *rootfs-overlays*. Buildroot also comes with a LTS release that only ships critical bug and security fixes. However, daily snapshots for testing new developments and more frequent stable updates are also available [26]. If desired, reproducible build support can be configured, which aims to provide identical binary build results given the same version is used, even if compiled on a different host device [26].

Software updates

Build system Buildroot releases one LTS version every year. Said release then gets further point releases with security, build and bug fixes over the year. The additionally available stable updates are more frequent and may contain breaking changes that need adjustments depending on the chosen setup [26].

Target system Builds created with Buildroot do not come with an update mechanism out of the box. However, Buildroot does have support for state-of-the-art atomic update tools like *Mender*, SWUpdate (Software Update for Embedded System) and RAUC for binary firmware updates. As the build chain in Buildroot creates flashable filesystem images instead of single packages [26], it is advisable to use a full image block-by-block update.

Notable uses

Buildroot is in use by a wide variety of projects. Notably, the OpenWRT build system deployed by significant projects like RIPE NCC Atlas uses it as a base [29]. In addition, Conseil Européen pour la Recherche Nucléaire (CERN) Fermilab researchers rebuilt a real-time data acquisition system to use Buildroot, yielding better maintainability, less multi-device support complexity, improved performance, and storage footprint when compared to their previous Scientific Linux-based setup. Moreover, the use of a customized real-time kernel was also made easier through the application of custom patches managed inside the Buildroot toolchain [30].

Summary

Buildroot offers a wide range of packages, including some of the ones needed for on-device processing based on GNU Radio [28]. Moreover, due to the demonstrated real-world readiness in terms of ease-of-use, in a research context, [30], and the project-specific limitations of Yocto, Buildroot is the build system of choice for DISCOSAT. More specific packages like *gr-iridium* used in similar Iridium® research [10] and the required drivers for the HackRF One SDR need to be added, but thanks to the low entry barrier and the ease-of-use first approach, this is a feasible task within the Buildroot build environment [30].

4. DISCOSAT ASSN

4.1. Design

This section will outline the significant design aspects for DISCOSAT. The primary focus is on the peripheral and probe management part, as this is underrepresented in current literature [8][11][30].

4.1.1. Aspects not covered

The following aspects were not part of the design due to exceeding the time constraints for this thesis.

Probe provisioning No design for automatic probe provisioning is provided.

Measurements No measurement execution, data access, or result storing exists. This is mainly because the required server backend is not finalized yet.

WiFi connectivity setup Setting up WiFi connectivity is not currently implemented, as that would require *volunteer* interaction. No suitable way of configuring the required parameters was agreed on. Writing a custom parser for network-specific config files stored on a USB-Stick seems like the easiest way to deal with this. However, a more robust and secure approach would be storing per-probe encrypted WiFi settings on the device through a one-time wired network setup routine done by the *volunteers*.

Fault tolerance/scalability The current DISCOSAT client-server communication design requires perfect uptime. No mechanisms exist to prevent problems caused by controller connectivity disruptions during network operations like

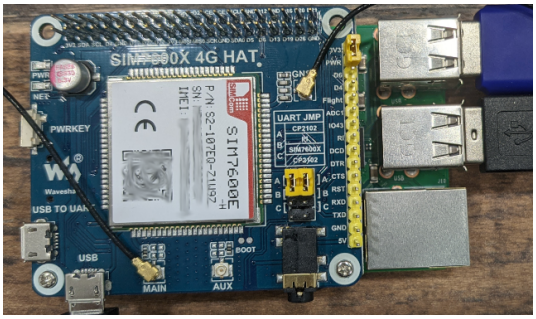
task retrieval or result uploading due to system maintenance or similar outage events. Moreover, a growing network could increase the load on the controller, prompting a re-design with load balancing features for scalability reasons.

4.1.2. Sensor Network

The simple client-server structure as shown in Figure 2.1 is suitable for the scope of DISCOSAT in this thesis. Moreover, the previously analyzed work for Atlas and F@h shows that this approach is well accepted. Additionally, the sensor will only support decoded data transmission powered by on-device processing, eliminating the need for real-time streaming data and keeping the setup simple. Furthermore, certificates pre-registered on the server are stored on each probe to establish the needed trust chain between client and server without relying on a zero-configuration provisioning mechanism.

Server The project constraints (section 2.3) limit DISCOSAT to the use of a single server. However, the server and network architectures are out-of-scope for this thesis, leaving a black box that mostly needs to satisfy aspects for the client.

Client The sensor used in DISCOSAT and the additional HackRF One SDR can be seen in Figure 4.1. It consists of the hardware previously mentioned (subsubsection 2.3.3).



(a) RPI 3b+ with SIM7600E-H LTE/GNSS hat

(b) HackRF One SDR

Figure 4.1.: DISCOSAT hardware and peripherals

4.1.3. Probe

Startup logic The high level startup-sequence can be seen in Figure 4.3. The bring up of the *DateTime*, *location*, *OTA* and additional system services is done by the operating system itself and described in more detail during the implementation phase. Once the time synchronization for required tasks like certificate validity checking happens, APOGEE takes over the management of the device. This service handles all the non-system initialization and provisioning required on the probe.

Failsafe Booting To allow for failsafe booting and bad-flash recovery, there are two copies of the operating system stored on the device. As shown in Figure 4.2, the *bootloader* tracks failed boot attempts and switches to the other copy if three consecutive start attempts fail due to system errors. For example, these errors are caused by the system daemon APOGEE not starting or more severe problems with the operating system ending in a device reboot before its start. The counter only gets reset by APOGEE if the system state is considered sane.

That means the internet connectivity is active, the probe has checked in with the server, and no critical tooling errors exist, see Figure 4.3 *ResetBootCount*.

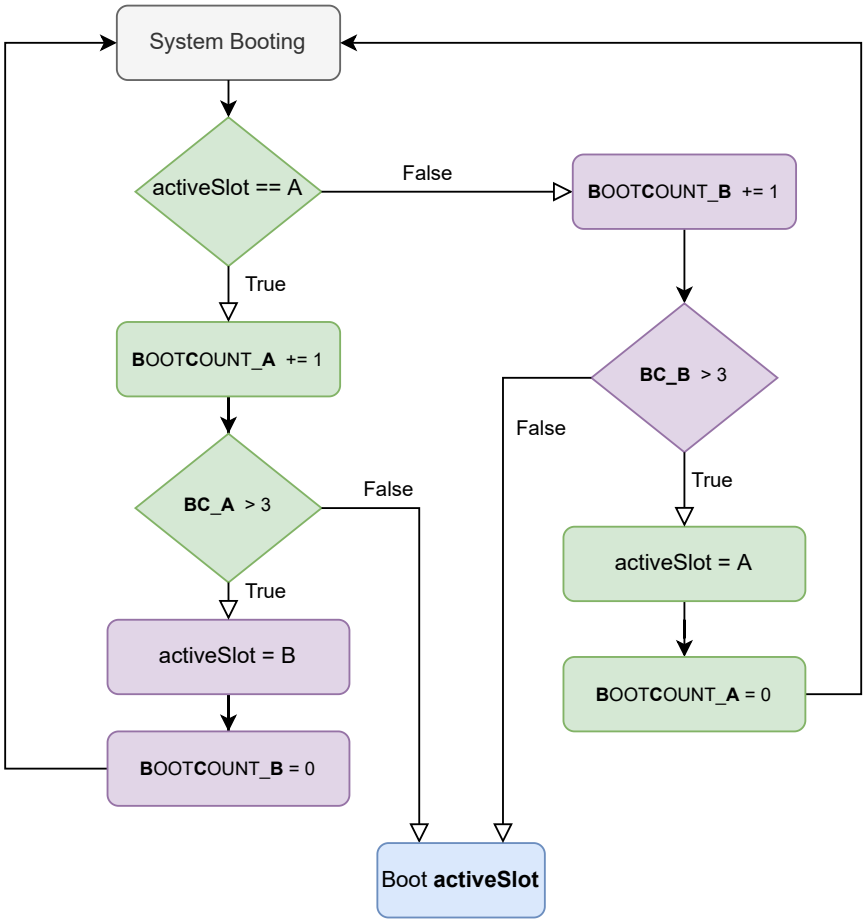


Figure 4.2.: DISCOSAT fail-safe booting flow

System Services DISCOSAT relies on functionality provided by pre-existing services as much as possible. This means, that tasks like network time synchronization *(DateTimeService)*, GNSS location tracking *(LocationService)*, network connectivity (including LTE and WiFi), and OTA-Updates *(OTA-Controller)* are not to be implemented solely by application code. Hence, guaranteeing a maintainable architecture, as the amount of custom maintenance scripting on the device itself is restricted to a minimum. Suitable interfaces for the necessary data and configuration retrieval from these services exist and are in use where applicable.

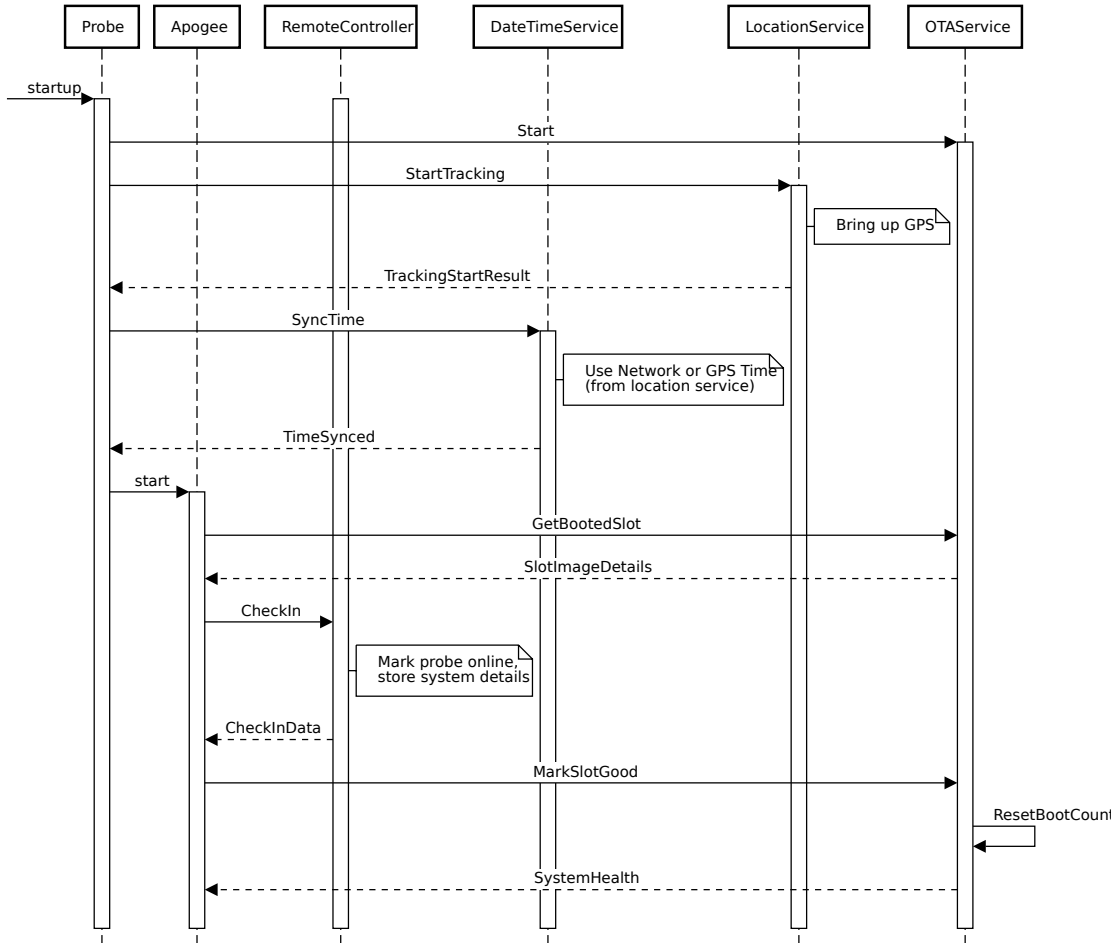


Figure 4.3.: DISCOSAT probe init sequence diagram

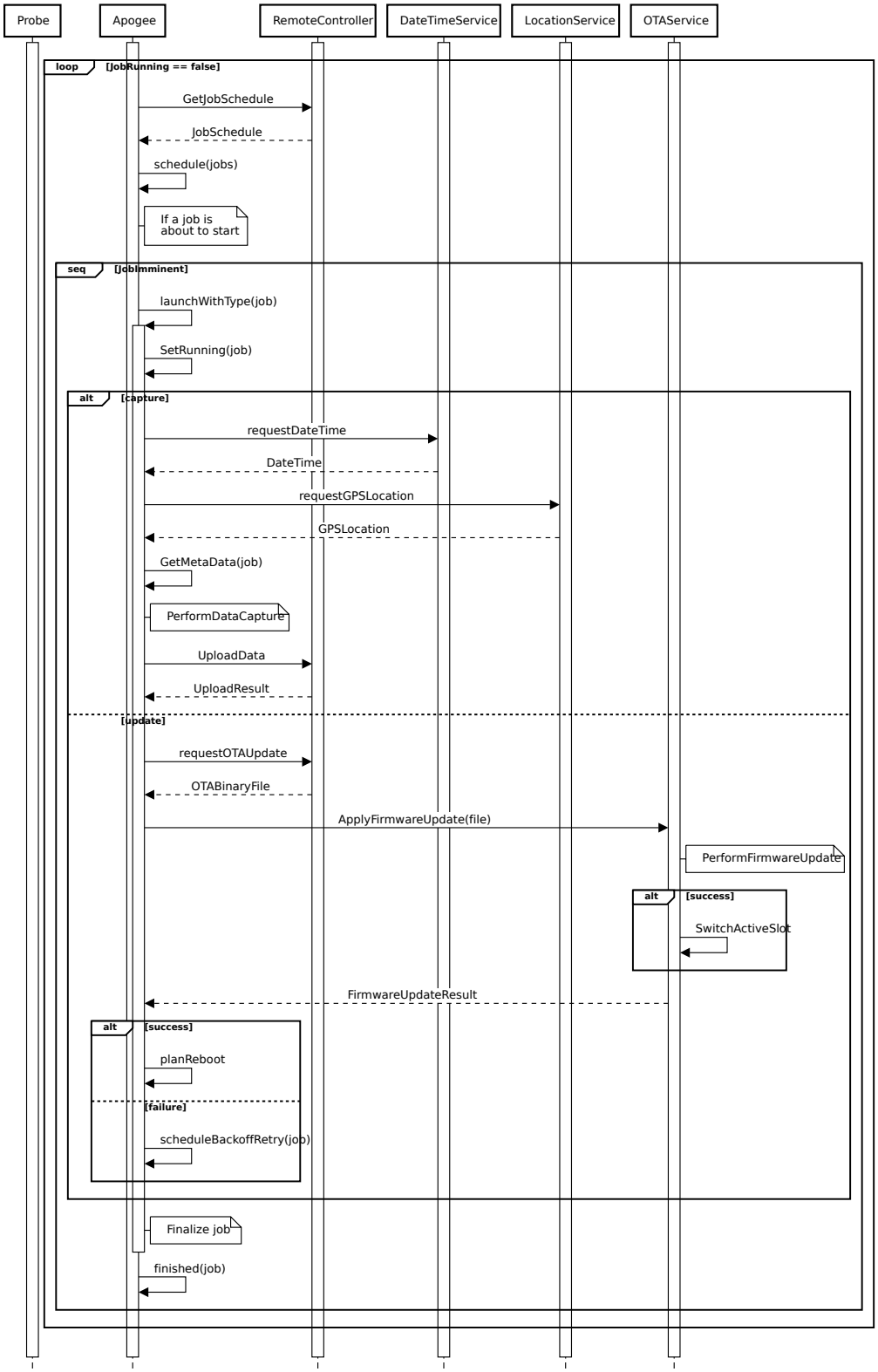


Figure 4.4.: DISCOSAT high-level probe job diagram

Jobs The job processing outlined in Figure 4.4 can be described as follows. Probes in the DISCOSAT network handle one job at a time, as multiple captures are not possible with the current hardware setup. Moreover, special job types exist for specific tasks. They will be extendable in the future. The following two main types are currently designed.

Decoded-Data-Capture The core of the job system is the data capture job that contains the metadata to set up the decoding and capture tools. It stores the GPS location of the probe and a start timestamp with second precision in the result data. Lastly, it invokes the specified decoder with the settings provided, runs until the specified end time, and packages the results for later upload.

Metadata The required parameter data defined for the capture job.

1. Start time [Timestamp]
2. End time [Timestamp]
3. Decoder tool [Enumeration]
4. Center frequency [Number in hz]
5. Sample rate [Number samples per second]
6. Bandwidth [Number in hz]
7. SDR gain and antenna settings

Additionally, optional parameters for each *decoder tool* also exist. In the case of *gr-iridium* the *decimation [Number]* can be specified.

The result contains the captured data in a binary format and additional metadata identifying the probe and the currently running firmware on the device.

Update This job type performs a firmware update on the device. It must be applied through the *OTA Service* as soon as possible if no job is currently running, as the same one job at a time limit applies. Firmware updates are schedulable by the *system manager* through a web interface running on the *controller*. Hence, updates are **not** sent out automatically to each online probe, limiting the data usage and allowing different firmware versions to co-exist.

Metadata The required parameter data defined for the firmware update job.

1. Firmware version [string]
2. Download location [string]

3. File Size [Number in byte]

The transferred update package contains an embedded signature from the *network owner*. Therefore, the *OTAService* checks this signature against a known authority file stored on the probe. If it is invalid and the update is still in the job list sent from the controller, the update job gets retried later. In addition, the *OTAService* also verifies update integrity through the use of checksums, triggering the exact on-failure retry mechanism mentioned before.

4.1.4. System and peripheral management

The core system management and maintenance aspect in long-term projects is barely touched upon in existing embedded sensor literature and therefore needs more in-depth consideration for DISCOSAT. In addition, requirements like millisecond-accurate time synchronization, secure and failsafe firmware updates, and on-device processing come into play when capturing satellite data on multiple independent devices.

Network connectivity Network connectivity is achieved by using the following decision logic shown in Figure 4.5. For simplicity, no error conditions are described, but failed WiFi and LTE connection attempts will be retried. Network cable hotplug events and WiFi roaming are also handled accordingly. The general approach of using pre-existing solutions heavily applies here.

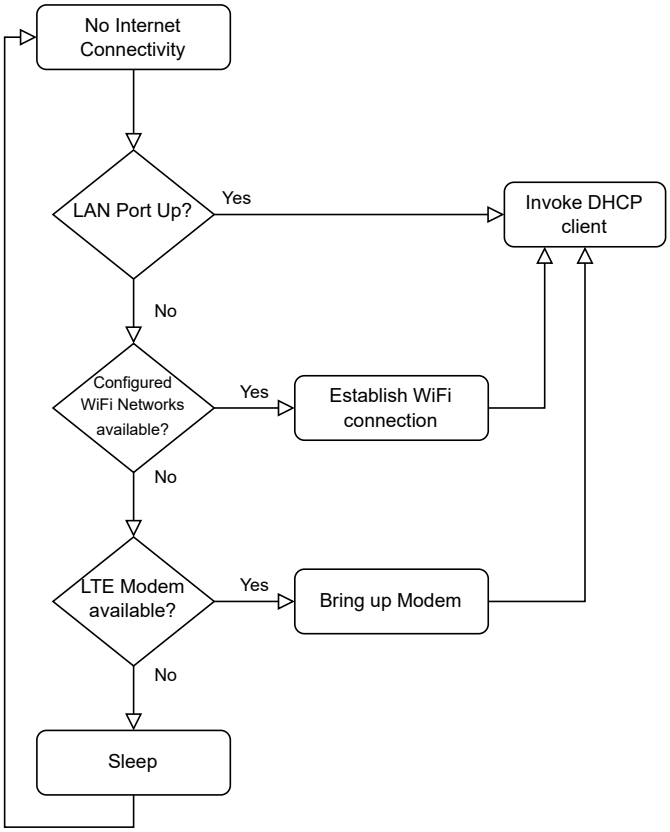


Figure 4.5.: DISCOSAT network connectivity decision logic

Time/Location synchronisation For DISCOSATs probe measurements to be valid, the exact sensor location and time need to be accurate to the second. If the time is incorrect, no synchronized work is possible, so the system management daemon APOGEE will not start. On the other hand, if no accurate location fix within *50 Meters* can be achieved, planned *Decoded-Data-Capture* jobs will still execute but indicate a lack of GNSS accuracy in the result.

Firmware DISCOSAT probes use a two-slot A/B root filesystem (*rootfs*) layout similar to the one used in recent versions of the Android operating system [31]. Additionally, the system has one primary *bootloader* in charge of booting the system and establishing the safety mechanisms needed for the fail-safe device initialization described in subparagraph 4.1.3. This *bootloader* is considered feature-complete upon deployment and only receives occasional security-relevant updates.

Security OTA-update integrity is verified through checksums to prevent the flashing of corrupt firmware files rendering the device unbootable. Moreover, only authorized and adequately cryptographically signed payloads get accepted by the *OTA-Controller* to prevent a hostile takeover of probes by malicious actors.

Size Firmware and update sizes are minimal due to the minimum base system approach. Furthermore, compression is in use for all OTA files transferred over the network.

Updating The type of firmware update *bootloader* or *rootfs* is stored in the downloaded firmware binary itself, making it possible to update the individual parts of the system without any additional metadata. Furthermore, adopting the seamless A/B approach from android [31], DISCOSATs firmware flashing logic ensures not to overwrite the current slot. **Example:** If slot A is currently executing the firmware flash, slot B is the target for the new system and vice-versa. A successful update sets the boot target slot to the presently inactive one and resets its *bootcounter* to zero, allowing the new system to take over on the next boot.

4.1.5. On-Device Processing

As already mentioned before, the network traffic hurdles to overcome when dealing with real-time raw SDR traffic make it necessary to not only capture the traffic but also decode it on the device. Currently, only one design that enables the parsing of Iridium® frames is required.

Iridium® The data required by the *DISCO Researchers* consists of the decoded packet-based data from Iridium®. Hence our device will only provide this data and run pre-processing straight on the device based on the *Decoded-Data-Capture* job execution pipeline.

4.2. Implementation

The following section will provide some of the core steps done while implementing the DISCOSAT design goals. It starts with a quick overview of the development status, followed by the main course of work, the implementation of the SATOS operating system, and all the features it provides. Finally, it ends with the future system daemon work planned for APOGEE.

4.2.1. Development status

APOGEE is in the early concept stage. No task scheduling, remote system management, or other automatic data-acquisition-related functionality exists. The implementation in this thesis mainly focuses on the SATOS part of DISCOSAT, as the system work required extensive research and development efforts. A prior work analysis for implementing state-of-the-art task scheduling and a rough design is part of this thesis so that development efforts can continue in the future.

4.2.2. SATOS - External Tree

SATOS is using the Buildroot build system to support multiple devices and make use of the required software and firmware tweaks necessary to achieve a performant and stable on-device processing chain. The code for SATOS is available upon request. SATOS adopts Buildroot's external-tree best-practice approach, which allows leaving custom modifications outside of the Buildroot sources, keeping the upstream repository as clean and mergeable as possible [26].

The *$(BR2_EXTERNAL_SATOS_PATH)* variable is commonly seen in the following sections. It contains the root directory of the used external tree and is needed to support directory lookups from inside the native Buildroot sources [26].

Organizing the tree One of the most important aspects was splitting the tree into multiple reusable components. For example, the *common* directory contains all the shared scripts and configurations reusable between the different

hardware probes available. More specific configurations are possible in the respective *vendor* folder like *raspberrypi* in the case of vendor adjustments like kernel or firmware configuration fragments needed for multiple devices of a single vendor. Furthermore, for adjustments of a particular probe model, the *device* subfolders like *rpi3-64* exist. For a more in-depth overview of the external tree structure used in SATOS, refer to the annotated version in Listing B.1.

Config fragments Diversions from the default configurations are kept to a bare minimum for maintainability. Hence, SATOS only uses fragments configurations for the Linux kernel, BusyBox and U-Boot. They get applied on top of the default configuration specified by Buildroot and therefore only need to contain the symbol changes required by the device. An example of such a fragment would be the non-standard U-Boot build configuration (Listing 4.4), which is explained in more detail later on.

Packages For DISCOSAT a few new custom packages and adaptation of existing ones is needed. The adjustments concerning already existing packages were applied in a fork of the upstream Buildroot GIT repository. However, custom and entirely new packages were added in the SATOS external tree to keep this work out of the GPLv2 licensed project and allow for a cleaner base without too many tweaks. In addition, this improves maintainability and supports easier updating to newer LTS versions due to the reduced merge conflicts on version changes and clean distinction between modified and completely new packages.

4.2.3. Hardware and Peripherals

When starting with the work on Buildroot, the available hardware was analyzed first. Furthermore, its requirements and limitations were figured out based on the designs for connectivity and processing required in DISCOSAT.

Raspberry Pi 3B+ The core of the DISCOSAT sensor is the RPI 3B+. Its USB2 connectivity allows it to only capture parts of the Iridium® spectrum as the achievable real-world USB2 Bandwidth is below the *I/Q* traffic load calculated in subparagraph 1.2. In [32, RPiDS: Raspberry Pi IDS — A Fruitful Intrusion Detection System for IoT] a network bandwidth hard-limit of $70Mbps$ was seen due to the shared USB2-bus, putting it below the theoretically-required $20MBps \equiv 160Mbps$ for real-time data streaming. Hence, it further enforces the need for

on-device processing. However, even without the network aspect, the raw processing power of the device is not high enough to capture the entire Iridium® L-band, as shown through an experiment conducted later on.

HackRF One The HackRF One SDR is capable of sniffing the entire Iridium® L-Band, as it has an operating frequency between 1 MHz to 6 GHz and a sample rate of 20 MS/s, which is more than enough to capture the 10 MHz bandwidth used in the Iridium® L-Band example given in subparagraph 1.2 [2]. It uses a USB2 connection, making it compatible with the RPI and other more powerful hardware platforms, making it reusable in the future.

LTE/GNSS - SIM7600E-H 4G HAT This RPI modem accessory made by Wave-Share delivers LTE connectivity and GNSS localizing with support for the GPS, BeiDou, and Glonass satellites. It sits on top of the RPI General Purpose Input Output (GPIO) header and makes use of the power, ground, and (optional) Universal Asynchronous Receiver/Transmitter (UART) connections provided by the RPI. The module also connects to the RPI through a USB2 connection providing multiple USB serial ports. Moreover, it is configurable through the use of *AT* commands sent to its management port. Lastly, it also features external Sub-Miniature Version A (SMA) antenna ports for LTE and GNSS antennas, allowing for a better reception when external antennas are in use. [33]

The exposed serial ports used by DISCOSAT are as follows:

1. */dev/ttyUSB1* - Dedicated National Marine Electronics Association (NMEA) output (needs to be started)
2. */dev/ttyUSB2* - Management interface for the modem (AT commands)

Summary The chosen hardware together with the previous constraints leaves the following list of essential steps needed for successful SATOS implementation.

1. Basic operating system setup
2. Time/Location synchronization (including GNSS)
3. LTE USB-Modem connectivity
4. GNURadio, HackRF One support and plugins
5. Network connectivity
6. OTA updates

4.2.4. Basic operating system setup

For creating an image, a specific device needs to be selected. Hence, the already existing default configuration file for the RPI 3B+ *buildroot/configs/raspberrypi3_64_defconfig* is used as a base, as it contains all required settings to get the system to a bootable state [27]. However, some extensive tweaking is needed to implement the design goals for DISCOSAT. The important ones are mentioned in the paragraphs below.

Boards The adjusted board configuration files are stored in the *config* folder, currently only the RPI 3B+ is supported through *rpi3_64_defconfig*. It is useable for multiple boards that share the same **RPI3** prefix and supports them by using different Linux kernel device trees.

Filesystem layout

By default, Buildroot uses a single EXT4 *rootfs* with an additional FAT32 *boot* partition [34]. However, this does not align with the two-slot system and failsafe booting approach, so tweaking was needed. This yielded the configuration provided in Listing B.2[1] to establish the needed layout by duplicating the *boot* and *rootfs* partitions. For DISCOSAT the *rootfs* partition has a fixed size of 512*MB* and is mounted read-only through applying the settings seen in Listing 4.1 inside *rpi3_64_defconfig*.

```
BR2_TARGET_ROOTFS_EXT2=y
BR2_TARGET_ROOTFS_EXT2_4=y
BR2_TARGET_ROOTFS_EXT2_SIZE="512M"
#BR2_TARGET_GENERIC_REMOUNT_ROOTFS_RW is not set
```

Listing 4.1: Buildroot board config *rootfs* tweaks

Moreover, the *boot* partition is also mounted read-only by default. These changes prevent accidental system modification and data loss caused by unexpected power dropouts, as no data gets written to the crucial partitions. However, a writeable *data* partition with a size of 512*MB* to store persistent configuration and measurement data on the probe was added. If desired by a developer, the *rootfs* can also be remounted in read-write mode by issuing the *mount -o remount, rw /* command, like for any standard EXT4 partition.

[1] The offset calculations seen in the boot partitions are specific to the U-Boot environment storing mentioned in the U-Boot paragraph.

Startup and initialization

The startup can be divided into 4-Phases.

1. Firmware
2. Bootloader (U-Boot)
3. Linux Kernel
4. Userspace

Firmware The system startup of the RPI uses a three-stage process. A first-stage bootloader is executed from Read Only Memory (ROM), it then loads and executes the second-stage bootloader called *bootcode.bin* from the SD-Card only for said second-stage to enable more system components and transferring the third-stage bootloader *loader.bin* to RAM, which in turn gets executed and runs the *start.elf* script, which is in charge of loading the optional *config.txt* file for system configuration parameters, the *cmdline.txt* file for kernel start parameters, and lastly a *Image* that contains an executable to start [35].

For DISCOSAT the ***c**ut-**d**own* versions of the firmware files are used because no GPU support is required. Furthermore, the *config.txt* firmware config location is adjusted as it needs to be changed later on. These changes are set in the board config file for the probe and are listed in Listing 4.2.

```
BR2_PACKAGE_RPI_FIRMWARE=y
BR2_PACKAGE_RPI_FIRMWARE_BOOTCODE_BIN=y
BR2_PACKAGE_RPI_FIRMWARE_VARIANT_PI_CD=y
BR2_PACKAGE_RPI_FIRMWARE_CONFIG_FILE="$(BR2_EXTERNAL_SATOS_PATH)/
    board/raspberrypi/rpi3-64/config_fw.txt"
```

Listing 4.2: Buildroot board config *firmware* tweaks

Bootloader - U-Boot U-Boot is a primary bootloader used for embedded devices that enables scriptable booting of the device's main operating system kernel. [36]

Instead of having a Linux kernel in the *Image* file, a secondary startup phase through the U-Boot bootloader is used. This is achieved by specifying *kernel=u-boot.bin* in the customized *config.txt* for the RPI, allowing implementation of the required failsafe booting design. Configuration started by enabling a redundant U-Boot variable environment stored on the SD-Card *(Multi Media Card (MMC))*.

The tweaks needed were specified in the fragment configuration stored in *board-/raspberrypi/uboot.fragment*. These changes are shown in Listing 4.3 and are needed to hold the boot order and attempts left. Moreover, network booting support was disabled to decrease system startup time.

```
CONFIG_ENV_OFFSET=0x4000
CONFIG_ENV_OFFSET_REDUND=0x8000
# CONFIG_ENV_IS_IN_FAT is not set
CONFIG_ENV_IS_IN_MMC=y
CONFIG_SYS_REDUNDAND_ENVIRONMENT=y
CONFIG_NET=n
```

Listing 4.3: U-Boot redundant environment config fragment

The *uboot.ush* script used for achieving the failsafe booting is based on the one provided by RAUC and can be found in Listing A.3. Besides using an implementation dependent inverted boot count approach when compared to the design flowchart in Figure 4.2, its core functionality is identical. Furthermore, depending on the active slot, it boots a different partition, either */dev/mmcblk0p2* in case of slot **A** or */dev/mmcblk0p3* in case of slot **B**.

Both of these customizations get applied through the board configuration file, so Buildroot knows how to build U-Boot with the required changes.

```
1 BR2_TARGET_UBOOT=y
2 BR2_TARGET_UBOOT_BOARD_DEFCONFIG="rpi_arm64"
3 BR2_TARGET_UBOOT_CONFIG_FRAGMENT_FILES="$(BR2_EXTERNAL_SATOS_PATH)
    /board/raspberrypi/uboot.fragment"
4
5 BR2_PACKAGE_HOST_UBOOT_TOOLS=y
6 BR2_PACKAGE_HOST_UBOOT_TOOLS_BOOT_SCRIPT=y
7 BR2_PACKAGE_HOST_UBOOT_TOOLS_BOOT_SCRIPT_SOURCE="$(
    BR2_EXTERNAL_SATOS_PATH)/board/raspberrypi/uboot.ush"
```

Listing 4.4: Buildroot board settings for U-Boot

Kernel The Linux kernel plays a critical role in supporting auxiliary devices and chips like WiFi and UART. While most of the RPI SBCs run on the *mainline* Linux kernel as well, they never get shipped with these kernels when they get released, as is indicated by the lack of support for the RPI 4 SBC. Moreover, it also continues to lack features that are available on the kernel provided by the RPI foundation as they are not merged yet [37]. In order to prevent pitfalls by missing functionality, Buildroot is using the customized Linux kernel sources from the RPI foundation [27, buildroot/configs/raspberrypi3_64_defconfig].

Using 5.15 LTS With the release of their Linux *5.15* LTS branch, the decision was made to switch out the currently in use *5.10* kernel sources within Buildroot to get the system on a LTS kernel release. This was achieved by modifying the following lines in the board configuration seen in Listing 4.2.4.

```
- BR2_PACKAGE_HOST_LINUX_HEADERS_CUSTOM_5_10=y
+ BR2_PACKAGE_HOST_LINUX_HEADERS_CUSTOM_5_15=y
BR2_LINUX_KERNEL_CUSTOM_TARBALL_LOCATION="$(call github,
    raspberrypi,linux,0efbe86e7248ad9b80a42b37a91c44860f91eee4)/
    linux-0efbe86e7248ad9b80a42b37a91c44860f91eee4.tar.gz"
```

The *TARBALL* directive contains the download location of the kernel. Buildroot parses it in the following format *github* indicates that *github.com* should be used as a download source, *raspberrypi* is the project name, and *linux* the repository. Furthermore, *0efbe86e7248ad9b80a42b37a91c44860f91eee4* is the full SHA-1 git commit hash of the kernel sources. This corresponds to the 5.15.25 LTS release [38]. The *call* helper then downloads the sources, and Buildroot uses them in the build process. The correct *HEADERS* directive also needs to be set, as they differ between kernel versions.

Checking the running kernel on the target system after building with the modified configuration yields the output in Listing 4.5, indicating a successful kernel update.

```
5.15.25-v8 aarch64 GNU/Linux
```

Listing 4.5: Output of the *uname -rom* command on SATOS

Configuration The kernel configuration also gets configured through a fragment file located at

```
BR2_LINUX_KERNEL_CONFIG_FRAGMENT_FILES="$(BR2_EXTERNAL_SATOS_PATH)
    /board/raspberrypi/linux.fragment"
```

Listing 4.6: Board configuration change for the Linux fragment configuration

The base configuration is almost sufficient but needs tweaks for certain aspects. For example, the CPU governor was set to *performance* by default to prevent CPU frequency switching and maximize performance as energy-saving is not required [39]. Some other required options for OTA update handling and USB modem support also need to be set, as annotated in Listing 4.7.

```
# Switch frequency governor to performance mode
CONFIG_CPU_FREQ_DEFAULT_GOV_PERFORMANCE=y

# Enable support for SQUASHFS images to apply OTAs
```

```
CONFIG_SQUASHFS=y
CONFIG_SQUASHFS_XATTR=y
CONFIG_SQUASHFS_ZSTD=y

# Enable DM_VERITY for the firmware updater
CONFIG_DM_VERITY=y
CONFIG_DM_VERITY_VERIFY_ROOTHASH_SIG=y
CONFIG_DM_VERITY_FEC=y

# Enable usb serial, WWAN and USBNET support for the modem.
CONFIG_USB_SERIAL=y
CONFIG_USB_SERIAL_WWAN=y
CONFIG_USB_SERIAL_OPTION=y
CONFIG_USBNET=y
```

Listing 4.7: RPI annotated Linux Kernel config fragment

Userspace Handling all the service startup dependencies is achieved by using the *systemd* init system. **Systemd** is a software collection that provides many different system components. It aims to provide a unified service configuration on all its supported systems [35]. In DISCOSAT the main task for *systemd* is managing system initialization and user-space service bootstrapping. Systemd uses *.unit* files to specify dependencies between services and control their startup. An example unit file is provided later on. Moreover, a simple command-line interface for checking the status of services exists. For example, executing *systemctl status* yields the output in Listing B.3 which shows the successful startup of all units and their command-line arguments.

4.2.5. Time/Location synchronisation

For DISCOSATs measurements, precise time down to the second is a hard requirement. As the used RPI does not have offer a Real Time Clock (RTC) that prevents clock drift, the following time synchronization features are used in SATOS.

NTP - Network Time Protocol

The main design goal for Network Time Protocol (NTP) is the distribution of time information over packet-based internet links that allows system operators to have a reliable time source. It also features forwarding and routing features

and has shown a historical time accuracy of a few milliseconds [40]. For DISCOSAT, it is used as the reference time, as the used GNSS module does not support the more accurate pulse per second operation mode [33].

GNSS - Time and Location

If no network connectivity is available at bootup, the system might stall as a reasonably accurate time is required to start the services correctly. The SIM7600E-H HAT features an NMEA format GPS data output that provides time and location-related data to clients. However, the SIM7600E NMEA output at */dev/ttyUSB1* needs to be started first. But since GPS is always required, the modem can be requested to start the NMEA output on startup. This only needs to be done once during the initial probe assembly,by issuing the *AT+CGPSAUTO=1* command seen in Listing 4.8 to the modem management serial port at */dev/ttyUSB2*.

```
> AT+CGPSAUTO?
+CGPSAUTO: 0
OK

> AT+CGPSAUTO=1
OK

// Verify it sticks after rebooting
> AT+CGPSAUTO?
+CGPSAUTO: 1
OK
```

Listing 4.8: AT commands used for SIM7600E GPS autostart

System integration

Support for both of these time syncing mechanisms exists in the **chrony** daemon available in Buildroot.

chronyd *chronyd* allows using the NTP protocol and the GPS NMEA data to synchronize the system time based through a single configuration file. The NMEA data is provided to **chronyd** by **gpsd** using a Shared Memory (SHM) interface. Moreover, the configuration file is provided in Listing B.4 and contains the NTP pool *0.pool.ntp.org* which allows DNS-based global resolving of

the nearest server addresses [2] and the settings for shared-memory powered data pulling from a GPS NMEA source like **gpsd**.

gpsd Is part of Buildroot and does also manage the location tracking support for DISCOSAT. The path to the GPS data port of the modem is specified in the */etc/default/gpsd rootfs-overlay* using the *DEVICES="/dev/ttyUSB1"* line. If the probe or GNSS hardware changes, this needs to be adjusted to reflect the new system configuration

Service startup Both, *chrony* and *gpsd* come with a *systemd unit* file that controls their startup. *chrony* needs to be started before *gpsd*, so the shared memory section mentioned above gets created correctly. Moreover, support for a systemd time sync dependency on *chrony* also exists, which delays or prevents services like APOGEE from starting if the time is not synchronized yet. These techniques can be seen in Listing 4.9. The main *unit* definition provides a human readable description and specified the required startup dependency on *chronyd.service*. The environment variables *$GPSD_OPTIONS* get loaded automatically from the */etc/(default | sysconfig)/gpsd Environment* files specified inside the unit.

```
1 [Unit]
2 Description=GPS (Global Positioning System) Daemon
3 After=chronyd.service
4
5 [Service]
6 Type=forking
7 EnvironmentFile=-/etc/default/gpsd
8 EnvironmentFile=-/etc/sysconfig/gpsd
9 ExecStart=/usr/sbin/gpsd $GPSD_OPTIONS $OPTIONS $DEVICES
10
11 [Install]
12 WantedBy=multi-user.target
```

Listing 4.9: Simplified and annotated *gpsd systemd* unit file

Timesynchronization accuracy

The accuracy achieved solely through the GPS data is obtainable by analyzing the *chronyc* tracking output seen in Figure 4.6. The $*$ next to *NMEA* indicates

[2] For large scale projects, a custom NTP pool should be used instead of the free and public infrastructure of the NTP project

that it is the selected source. Compared to the reference and more accurate NTP servers below, it achieves an offset of less than $\approx 7000\mu s - 1000\mu s = 6ms$. For this calculation, the measured offsets in the brackets are substracted to compare the currently selected NMEA reference to the more accurate NTP time sources. Nevertheless, this is enough to achieve the sub-second accuracy needed for DISCOSATs data acquisition to work reliably.

```
# chronyc tracking
Reference ID    : 4E4D4541 (NMEA)
Stratum         : 1
Ref time (UTC)  : Thu Mar 24 15:30:05 2022
System time     : 0.000850052 seconds fast of NTP time
Last offset     : +0.001471169 seconds
RMS offset      : 0.001324951 seconds
Frequency       : 7.922 ppm fast
Residual freq   : +75.152 ppm
Skew            : 0.692 ppm
Root delay      : 0.000000001 seconds
Root dispersion : 0.001990973 seconds
Update interval : 8.0 seconds
Leap status     : Normal
# chronyc sources
MS Name/IP address         Stratum Poll Reach LastRx Last sample
===============================================================================
#* NMEA                          0   3   377     9   -460us[+1011us] +/- 1000us
^- mail.rrz.cc                   2   6   377    16  +4946us[+6950us] +/-   11ms
^- router.gunnarhofmann.de       2   6   377    18  +4939us[+6944us] +/-   11ms
^- spacys.de                     2   6   377    16  +4952us[+6956us] +/-   11ms
^- ntp1.kashra-server.com        2   6   377    17  +4950us[+6954us] +/-   11ms
```

Figure 4.6.: Chrony time syncing sources and tracking offsets

Location accuracy

Tracking error estimates can be obtained by executing *cgps*, a client for *gpsd*. The output provided in Figure 4.7 shows a *2D* and *3D* error of below twenty meters. However, these estimates are calculated by algorithms inside the receiver and should be taken with a grain of salt [41]. Due to privacy reasons, the exact location is not provided, but the tracked location was within the estimated *2D* tracking error range.

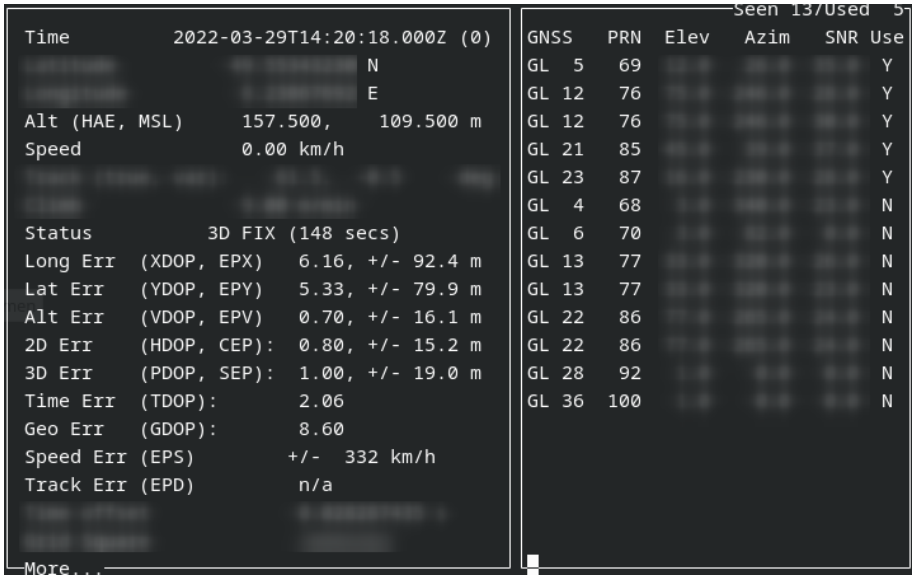


Figure 4.7.: GNSS tracking error estimates as seen by *cgps*

4.2.6. GNU Radio

A version bump of *GNU Radio* was necessary to get the required plugins for the on-device processing pipeline to run. *GNU Radio* was upgraded from version 3.8 (available in Buildroot 2022.02 LTS [27]) to **3.9** to get a more recent version that uses *PyBind11* instead of *swig* to bind C++ code for python for better compatibility [42]. The main motivation behind this is that some plugins like *gr-iridium* already made the transition, and their long-term stability for older GNU Radio versions is not tested nor guaranteed [43].

Required version upgrade from 3.8 to 3.9

The upstream GNU Radio package inside Buildroot mainly needed a change in the version number field. Moreover, the dependencies also changed, GNU Radio 3.9 now requires *numpy* on the compiling host as-well, *swig* got replaced with *pybind* and the *python-mako* and *python-six* host dependencies were dropped. Moreover, the required *volk* vector optimized math library is not shipped with GNU Radio anymore [44] and also needs a new package. Some special workaround is required to bypass cross-compilation issues encountered with the new *pybind11* build chain [45]. The full patch required to bring GNU Radio to 3.9 inside Buildroot is too large for this thesis and can be obtained from the Buildroot fork used in this thesis [46].

Plugins

Some existing GNU Radio plugins also needed modification to work with the hardware and software stack used by DISCOSAT.

gr-osmosdr A modified *gr-osmosdr* version with enabled HackRF One support is necessary, as the one shipped by Buildroot lacks support, as shown previously in Figure 3.3. Fortunately, the required functionality could be achieved by adding the necessary compile-time definitions for building the HackRF driver inside *gr-osmosdr*. Buildroots structure made this particularly easy, as only package file adjustments were needed. The changes are also obtainable through the Buildroot fork used in this thesis [46] and are provided with additional commentary in Listing A.4. Furthermore, the *hackrf* package used by *gr-osmosdr* is already part of Buildroot, and therefore no additional work is required.

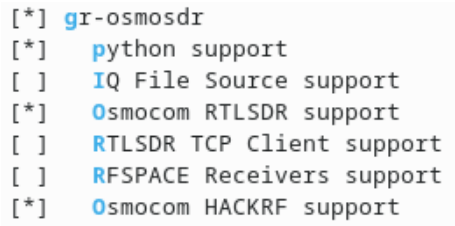


Figure 4.8.: *gr-osmosdr* - HackRF support in the configuration menu

gr-iridium The Python-based command-line tool *iridium-extractor* that is bundled with *gr-iridium* [47] is required for decoding and extracting the frames received through the HackRF One. The inherent use of Python packages, as seen in Figure 4.9 made this a challenging task. The *gr-iridium* package does not exist within Buildroot. It was added based on the three-file package creation mechanism outlined in the Buildroot manual [26]. The changes are available in the SATOS external tree under *package/gr-iridium*. Furthermore, the package can be added to board configuration files using one or both of the lines in Listing 4.10 with the second line depending on the need for python-support.

```
BR2_PACKAGE_GR_IRIDIUM=y
BR2_PACKAGE_GR_IRIDIUM_PYTHON=y
```

Listing 4.10: Board configuration snippet for *gr-iridium*

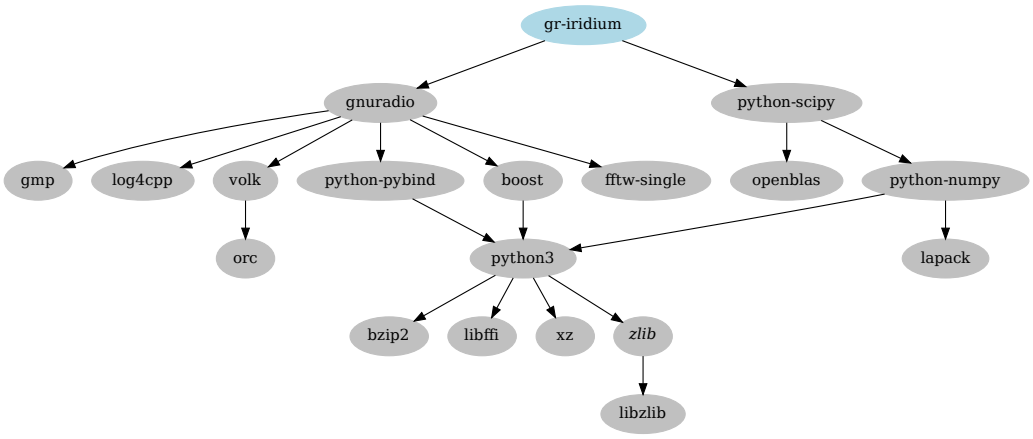


Figure 4.9.: *gr-iridium* - Target System Dependencies

The left *gnuradio* subtree is already displayed in Figure 1.1, but *gr-iridium* requires the addition of a previously not existing package called *python-scipy*.

python-scipy is a python package that provides highly-optimized scientific computing algorithm implementations usable through python [48]. It is not available within Buildroot but was staged a few times over the past years, with the most recent attempt done in February 2022 [49]. An older version of this patch was used for the inclusion of *python-scipy* in SATOS as the newest version was not out at the time of writing this thesis. Furthermore, it requires three new python packages *python-pythtran*, *python-beniget* and *python-gast* as host dependencies that were also added based on the above patch set. They also reside in the external tree *package/python-[pythran | beniget | gast]* folders and can be replaced by the upstream patch as soon as it gets accepted.

With all these dependencies handled, the *iridium-extractor* toolkit is now useable for on-device decoding later on.

4.2.7. Network connectivity

Network connectivity management got offloaded to industry-proven solutions, primarily relying on the de-facto standard Linux solution *NetworkManager* for control of different connection types through a unified interface. Network manager allows access to network functionality like enabling or disabling network

devices through the use of *nmcli*. Furthermore, it provides an extensive Desktop Bus (D-Bus) integration for automation through external applications making it the perfect candidate for usage within APOGEE later on [50].

LTE - Modem Manager

Only adding *ModemManager* to the Buildroot board configuration file was not enough to get the modem to work. Hence, this is where the network and modem-related kernel tweaks seen in Listing 4.7 come into play. Moreover, a configuration for the LTE sim card itself is needed. It can be found in Listing B.5 and primarily consists of the required *gsm* Access Point Name (APN) configuration. The connection is configured to only start manually through the use of the *autoconnect=false* directive, allowing it to be only enabled if desired following the sequence in Figure 4.5. All the low-level implementation specifics on how to control the modem are abstracted away by the use of *Modem and NetworkManager* and their tight integration.

Bringing up LTE Connecting to the LTE Network is achieved through a simple command *nmcli con up congstar* yielding the following response *Connection successfully activated (D-Bus active path: /org/freedesktop/NetworkManager/ActiveConnection/3)* and bringing up the *wwan0* interface automatically as shown in Figure 4.10.

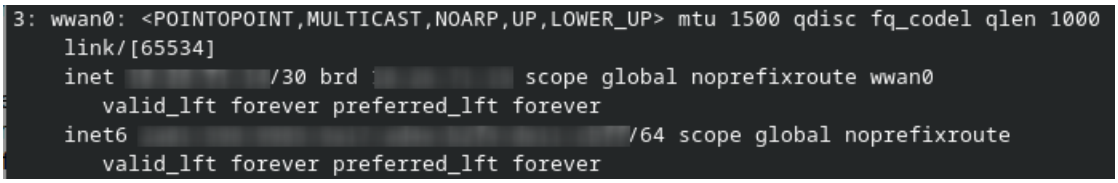


Figure 4.10.: LTE connectivity through Network- and ModemManager

Disabling the LTE network connection is done by *nmcli con down congstar*. Moreover, these tasks can also be performed by directly talking to the *ActivateConnection* or *DeactivateConnection* endpoints of the NetworkManager D-Bus interface [51].

wpa_supplicant - WiFi

WiFi network connectivity is also controlled through NetworkManager and its integration with *wpa_supplicant*. *wpa_supplicant* is a free implementation of an

IEEE 802.11i supplicant, hence its name. It has support for modern WiFi protocols like WPA3 and WPA2 and all commonly used encryption schemes [52].

Show available WiFi Networks The NetworkManager command *nmcli dev wifi* provides a list of all WiFi networks in range and works on SATOS as shown in Figure 4.11.

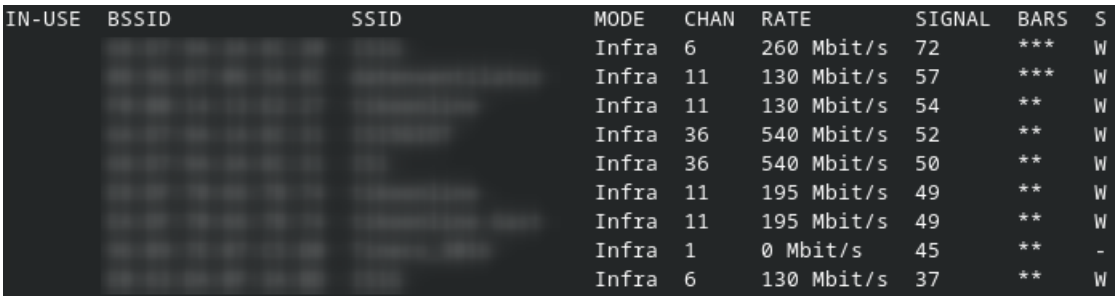


Figure 4.11.: WiFi network list demo

4.2.8. OTA-Updates

For OTA-Updates, the A/B slot firmware design needs to be taken into account. This is achieved through RAUC, a firmware update solution primarily used in embedded devices. It consists of a client in charge of applying the update which is running on the device itself and the tools that allow the building of the OTA-update bundles. *rauc* was chosen over *swupd* and *mender* due to its tight integration into Buildroot and the readily available well-documented example project *br2rauc* licensed under GPL-v2 that covered the groundwork and was adaptable to fit DISCOSATs needs [53]. The provided packages can be enabled using the annotated options shown in 4.11.

```
1    # Enable the RAUC client on the device
2    BR2_PACKAGE_RAUC=y
3    # Enable the client dbus support, for automation
4    BR2_PACKAGE_RAUC_DBUS=y
5    # Enable the host tools for firmware creation.
6    BR2_PACKAGE_HOST_RAUC=y
```

Listing 4.11: RAUC annotated Buildroot board config additions

Client configuration

The client portion of RAUC gets configured by a single configuration file for each individual device. This configuration can be seen in Listing B.7. The *system* section describes the *compatible* target system for which to accept update image and sets the *bootloader* to U-Boot. Additionally, the *mountprefix* for firmware update staging and the *statusfile* that keeps track of update-history and installed slots is set. Moreover, the *plain bundle* format for slots was disabled to only allow use of *crypt* and *verity*, two secured and integrity checked formats [54]. In addition, the *keyring* file stored on the device is set, that is needed for secure verification of the update packages. Lastly, the slot definitions are equal to the partitioning layout in Listing B.2.

Security and Integrity

Only bundles that contain the new *dm-verity* SHA-256 hash tree support over the generated SquashFS filesystem images are created. This is why the Kernel fragment tweaks for *DM* and *SQUASHFS* mentioned in Listing 4.7 are needed. Moreover, *verity* enables integrity checking of every single file inside the package and prevents any tampering with the update images [54]. Each signed update image needs to verify against the *keyring* stored on the device. If the signature is not valid, the update will not apply.

Creating bundles

The build process automatically creates bundles through the use of the main post-build and post-image scripts in Listing B.1. These scripts then invoke functions provided by a helper script called *scripts/rauc.sh* which in turn generates the bundles and stores the *keyring* file used for the build on the device. It requires the presence of a key and a certificate authority file for the development and production mode. This CA gets generated through standard PKI setup commands, e.g., by using *openssl*, making the process easy to replicate but too lengthy for this thesis. Once the required PKI files exist, the *rauc_generate_(root | boot)_bundle* functions create the required *manifest* files specifying the bundle contents, version and the device they are compatible with. This metadata is embedded and later used by the device to verify if the update is for the right board. Next, the RAUC tooling gets invoked and creates a finished bundle from the metadata and image *rootfs/bootfs* images available. These

images then get stored in the */release* folder of the SATOS tree. The update packages are named like *satos-rpi3-64-dev-20220309-b122c30-rootfs.raucb*, containing the compatible device *rpi3-64*, the build type *dev* the data of the build *20220309*, the short GIT revision *b122c30* and the contained type of filesystem *rootfs*.

Applying updates

In DISCOSAT, the automatic transfer of bundles is supposed to happen through APOGEE. However, right now, the updates need to be applied manually by transferring them to the device and invoking the install commands seen in Listing 4.12. First, the working directory is changed to the */tmp* folder, as the *rootfs* is read-only, and the subsequent download commands need to store the files on the device. Lastly, the installation is invoked for the *bootfs* and *rootfs* slots.

```
1 cd /tmp
2
3 wget https://ota-server/satos-rpi3-64-dev-20220309-b122c30-bootfs.
    raucb -O bootfs.raucb
4 wget https://ota-server/satos-rpi3-64-dev-20220309-b122c30-rootfs.
    raucb -O rootfs.raucb
5
6 rauc install bootfs.raucb
7 rauc install rootfs.raucb
```

Listing 4.12: RAUC manual firmware update installation

RAUC then handles the installation and all the required boot logic changes transparently, as shown in the update job in Figure 4.4. This is made possible through the bootloader integration with U-Boot and the use of the already presented customized boot script (Listing A.3).

Verifying state

The currently booted slot can be obtained through issuing the *rauc status* command, it generates the output shown in Figure 4.12 and indicates the booted slot, status and system info.

```
# rauc status
=== System Info ===
Compatible:  satos-rpi3-64
Variant:
Booted from: rootfs.0 (A)

=== Bootloader ===
Activated: rootfs.0 (A)

=== Slot States ===
  [bootloader.0] (/dev/mmcblk0, boot-mbr-switch, inactive)

o [rootfs.1] (/dev/mmcblk0p3, ext4, inactive)
        bootname: B
        boot status: good

x [rootfs.0] (/dev/mmcblk0p2, ext4, booted)
        bootname: A
        mounted: /
        boot status: good
```

Figure 4.12.: RAUC - System status

If more details are desired, *rauc status --detailed* provides the exact version information, checksum and size of the booted slot, the installation and activation date and count, and the current status as portrayed for *slot A* in Figure 4.13.

```
x [rootfs.0] (/dev/mmcblk0p2, ext4, booted)
       bootname: A
       mounted: /
       boot status: good
     slot status:
         bundle:
             compatible=satos-rpi3-64
             version=dev-20220309-b122c30
         checksum:
             sha256=e19b2f0e438dfd148dad09cad7811571b1278c44ebf0f9a50a4ed39a6223f05a
             size=536870912
         installed:
             timestamp=2022-03-09T19:42:59Z
             count=15
         activated:
             timestamp=2022-03-09T19:42:59Z
             count=15
         status=ok
```

Figure 4.13.: RAUC - System status detailed

The above data and all operations are also available through a D-Bus API provided by RAUC [54] that is going to be used by APOGEE.

4.2.9. APOGEE - Client daemon

GO is the language of choice for APOGEE, as it provides a higher-level interface reducing the required development effort and providing additional functionality through the use of pre-existing packages like *go-dbus*, which allows connecting to the D-Bus system and application buses [55].

Scope

First of all, APOGEE will be in charge of marking the currently booted system as *good* after the check-in to the server has been performed (see Figure 4.3). However, as it does not exist yet, this has to be performed manually through the use of the *rauc status mark-good* command to prevent switching slots after three startups. Furthermore, in the future, APOGEE will implement the task scheduling and auxiliary system management design for DISCOSAT through the use of D-Bus bindings to the chosen implementations in SATOS.

D-Bus

D-Bus is a middleware mechanism that allows programs to communicate with each other over one shared system or multiple point-to-point buses. It features a connection-based and stateful approach to transmitting binary data messages between processes [56]. APOGEE will use it to obtain location data through *gpsd*, manage networks through *NetworkManager* and perform updates through *rauc*.

In addition to the already previously mentioned uses of D-Bus, some more details on which data gets exchanged over D-Bus in the case of OTA-Updates and GNSS location tracking is provided below.

RAUC The D-Bus interface in RAUC comes with full support for system status setting and update bundle installation. The annotated commands in Listing 4.13 show how APOGEE can use D-Bus to perform OTA related tasks.

```
# Mark rootfs.0 as good using DBUS
busctl call de.pengutronix.rauc / de.pengutronix.rauc.Installer
   Mark ss "good" "rootfs.0"

# Install an update bundle over DBUS
busctl call de.pengutronix.rauc / de.pengutronix.rauc.Installer
   InstallBundle sa{sv} "/path/to/satos-bundle" 0
```

```
# Retrieve progress from the installer
busctl get-property de.pengutronix.rauc / de.pengutronix.rauc.
    Installer Progress

# Get status of all slots
busctl call de.pengutronix.rauc / de.pengutronix.rauc.Installer
    GetSlotStatus
```

Listing 4.13: *dbus* commands for RAUC [54]

GPSD *gpsd* outputs its satellite data to the system bus each time a location fix happens [57]. An example of the delivered data is in Listing 4.14. This data will be used in APOGEE by mapping it to the right fields based on the table in Table 4.1.

```
signal time=1648397962.024116 sender=:1.3 -> destination=(null
    destination) serial=19085 path=/org/gpsd; interface=org.gpsd;
    member=fix
     double 1.6484e+09
     int32 3
     double 0.005
     double 49.5534
     double 8.23865
     double 17.1
     double 108.5
     double 18.4
     double 217.7
     double nan
     double 0
     double nan
     double nan
     double nan
     string "/dev/ttyUSB1"
```

Listing 4.14: *dbus-monitor --system* output for gpsd

4.3. Usage

This section will provide some details on how to use the SATOS implementation to perform build and research tasks.

DBUS_TYPE_DOUBLE	Time (seconds since Unix epoch)
DBUS_TYPE_INT32	mode
DBUS_TYPE_DOUBLE	Time uncertainty (seconds).
DBUS_TYPE_DOUBLE	Latitude in degrees.
DBUS_TYPE_DOUBLE	Longitude in degrees.
DBUS_TYPE_DOUBLE	Horizontal uncertainty in meter.
DBUS_TYPE_DOUBLE	Altitude MSL in meters.
DBUS_TYPE_DOUBLE	Altitude uncertainty in meters.
DBUS_TYPE_DOUBLE	Course in degrees from true north.
DBUS_TYPE_DOUBLE	Course uncertainty in meters
DBUS_TYPE_DOUBLE	Speed, meters per second.
DBUS_TYPE_DOUBLE	Speed uncertainty in meters per second.
DBUS_TYPE_DOUBLE	Climb, meters per second.
DBUS_TYPE_DOUBLE	Climb uncertainty in meters per second.
DBUS_TYPE_STRING	Device name

Table 4.1.: GPSD Satellite object data mapping (taken from [57])

4.3.1. Image building

Creating the different images for the target devices is achieved through a *Makefile* in SATOS, based on the one used in the HomeAssistant Operating system project [58]. When invoking *make help*, the supported targets are listed. These targets can then either be built all at once by issuing the *make* command without any parameters or individually, e.g., by invoking *make rpi3_64* to build the RPI 3B+ image.

4.3.2. Initial installation

The build process generates the *satos_rpi3_64_sdcard.img* file, which is a full block-level system image containing all required partitions. It can be flashed to any sufficiently large SD-Card using the command given in Listing 4.15. It copies the image to the target and flushes the storage buffers. It is required only once, as further firmware changes are done by OTA-Updates.

```
sudo dd if=release/satos_rpi3_64_sdcard.img of=/dev/mmcblk0 bs=4M
sync
```

Listing 4.15: SATOS SD-Card installation command

4.3.3. Probe access

The only way to interact with the operating system of the device is through the use of the RPI serial UART connection, as no remote access through *SSH* or similar techniques exist. This further limits the attack surface, as no ports get exposed to the internet, and no security-critical services like *ssh* or *dropbear* run directly on the device.

4.3.4. Data capture

The capture and decoding process can be started by executing the commands seen in Listing 4.16 on the probe. First, the *home* folder of the user needs to be overwritten by exporting a new *HOME* environment variable before the start of *iridium-extractor*. Without this change, GNU Radio tries to write lock-files to the read-only partitions, causing the startup to fail with *RuntimeError: Failed to create FFTW wisdom lockfile: /root/.gr_fftw_wisdom.lock*.

```
export HOME="/data/"
iridium-extractor -D 4 "hackrf_br_rpi3.conf" > data.log
```

Listing 4.16: *iridium-extractor* sniffing example

Decoding configuration

A custom SDR dependent configuration is required for *iridium-extractor*. It is used for setting the required parameters needed to capture the Iridium® satellite L-band. The file is shown in Listing B.6 and was tweaked by experimentally testing the limits of the device.

Performance limitations When invoked with a higher sample rate like $4MS/s$, the probe started dropping samples as seen in Listing B.8 indicating that the python-based GNU Radio processing pipeline runs too slow or the USB2 bus is overwhelmed with the inrush of data. Hence, the current setup is not fast enough to capture the entire Iridium® spectrum.

The output seen in Figure 4.14 shows the on-device decoding pipeline dumping the captured data into the *data.log* file and outputting status messages on the interactive console.

```
# iridium-extractor -D 4 "hackrf_br_rpi3.conf" > data.log
gr-osmosdr 0.2.0.0 (0.2.0) gnuradio 3.9.5.0
built-in source types: rtl hackrf
Using HackRF One with firmware 2021.03.1
(RF) Gain: 14.0 (Requested 14)
IF Gain: 40.0 (Requested 40)
BB Gain: 20.0 (Requested 20)
Warning: Setting bandwidth to 1750000.0
Warning: Setting antenna to TX/RX
/bin/iridium-extractor:356: DeprecationWarning: setDaemon() is deprecated, set the dae
mon attribute instead
  statistics_thread.setDaemon(True)
1648396571 | i:   0/s | i_avg:   0/s | q_max:    0 | i_ok:   0% | o:    0/s | ok:   0%
 | ok:   0/s | ok_avg:   0% | ok:          0 | ok_avg:   0/s | d: 0
1648396572 | i:   0/s | i_avg:   0/s | q_max:    1 | i_ok: 100% | o:    0/s | ok: 100%
 | ok:   0/s | ok_avg: 100% | ok:          1 | ok_avg:   0/s | d: 0
1648396573 | i:   1/s | i_avg:   1/s | q_max:    1 | i_ok: 100% | o:    1/s | ok: 100%
 | ok:   1/s | ok_avg: 100% | ok:          3 | ok_avg:   1/s | d: 0
1648396574 | i:   0/s | i_avg:   1/s | q_max:    1 | i_ok: 100% | o:    0/s | ok: 100%
 | ok:   0/s | ok_avg: 100% | ok:          4 | ok_avg:   1/s | d: 0
1648396575 | i:   5/s | i_avg:   2/s | q_max:    2 | i_ok:  83% | o:    5/s | ok:  83%
 | ok:   4/s | ok_avg:  90% | ok:          9 | ok_avg:   2/s | d: 0
1648396576 | i:  10/s | i_avg:   4/s | q_max:    3 | i_ok:  81% | o:   10/s | ok:  81%
 | ok:   8/s | ok_avg:  85% | ok:         18 | ok_avg:   3/s | d: 0
1648396577 | i:   8/s | i_avg:   4/s | q_max:    1 | i_ok:  77% | o:    8/s | ok:  77%
 | ok:   6/s | ok_avg:  83% | ok:         25 | ok_avg:   4/s | d: 0
Done.
#
```

Figure 4.14.: *iridium-extractor* running on the probe

It provides proof that the raw data HackRF One capture using GNU Radio 3.9.5.0 and *gr-osmosdr* 0.2.0 works on the probes. Furthermore, the python-based on-device decoding through *gr-iridium* is also working, as *iridium-extractor* executed correctly.

5. Usecase analysis for SATOS

In this chapter, SATOS will briefly be analyzed based on the useability, adaptability, and maintainability requirements set for DISCOSAT.

5.1. Useability

Currently, the system is only accessible through a USB serial connection allowing no remote sensor work. Still, the *researchers* do not need to pay attention to the complex processing readiness-work required for getting *gr-iridium* decoding to work on a RPI SBC inside a sane and modern GNU Radio and Linux environment, as they get a finished image that is installable on any RPI 3B+. Furthermore, the complexity required to get features like LTE and GNSS working out of the box is hidden behind a well-structured system service setup.

5.1.1. Adding new packages

Adding new packages to the external SATOS tree is done through simple makefiles. The structure is in the external device tree layout given in Listing B.1. The optional *.hash*, *Config.in* for the inclusion of config options and the main *.mk* file for specifying the toolchain and steps required for building the packages are easy to understand and well documented in the Buildroot manual [26].

5.1.2. Modifying configurations

Modifying major package configurations is primarily done by overriding the existing ones through the use of *fragment files*. As seen before, these provide a minimal difference view of the changed options. In addition, system software configuration adjustments on the target device are possible by using the *rootfs-overlay* mechanism provided by Buildroot, which allows placing arbitrary files everywhere in the system.

5.1.3. Adding new boards

Adding new hardware to SATOS is a nontrivial task, as the device needs to be supported by Buildroot already, and the appropriate board configuration files need creation. In the case of a different RPI, common files shared can be reused. Still, the device hardware itself needs a hand-crafted new board configuration file that contains the same core adjustments outlined for the RPI 3B+ in the implementation section.

5.1.4. Firmware Updates

The build scripts automatically create firmware OTA-Updates on every build. Moreover, the security mechanisms deployed are similar to the ones used by F@h and Atlas. Firmware updates are integrity checked using state-of-the-art mechanisms like *dm-verity* based on *SHA-256* checksums. Furthermore, they are signed and only flashable if their signature is valid. The rollout of updates is not implemented, but as the bundles consist of one file for the respective *rootfs* or *bootfs* slot, installation is possible through a single command.

5.2. Maintainability

Maintenance work heavily relies on the used GIT version control. Updating to a newer Buildroot release is done by rebasing the used submodule on the latest version. The task described before is a common GIT task for developers and should be manageable by the *network owner*. In the case of merge conflicts, the custom patches need evaluation leaving one source of potential work open. However, this is unavoidable in any version-controlled project, and GIT provides adequate tooling to handle these challenges.

5.3. Adaptability

One primary research concern is the addition of new satellite data decoding pipelines. But, again, Buildroot makes this simple boiling down to the same dependency management and new package configuration work already outlined for the *gr-iridium* pipeline in SATOS above.

5.3.1. Operating system size

Due to the fixed *rootfs* partitions, it is also important to keep system and update sizes in check so enough free space is available for further tooling expansion. The uncompressed factory SD-Card image generated by Buildroot is $1.6GB$ in size. Presently the pre-allocated $512MB$ *rootfs* partition only uses $\approx$ $291MB$ of storage as seen in Figure 5.1, allowing bigger *rootfs* updates to be applied without requiring repartitioning.

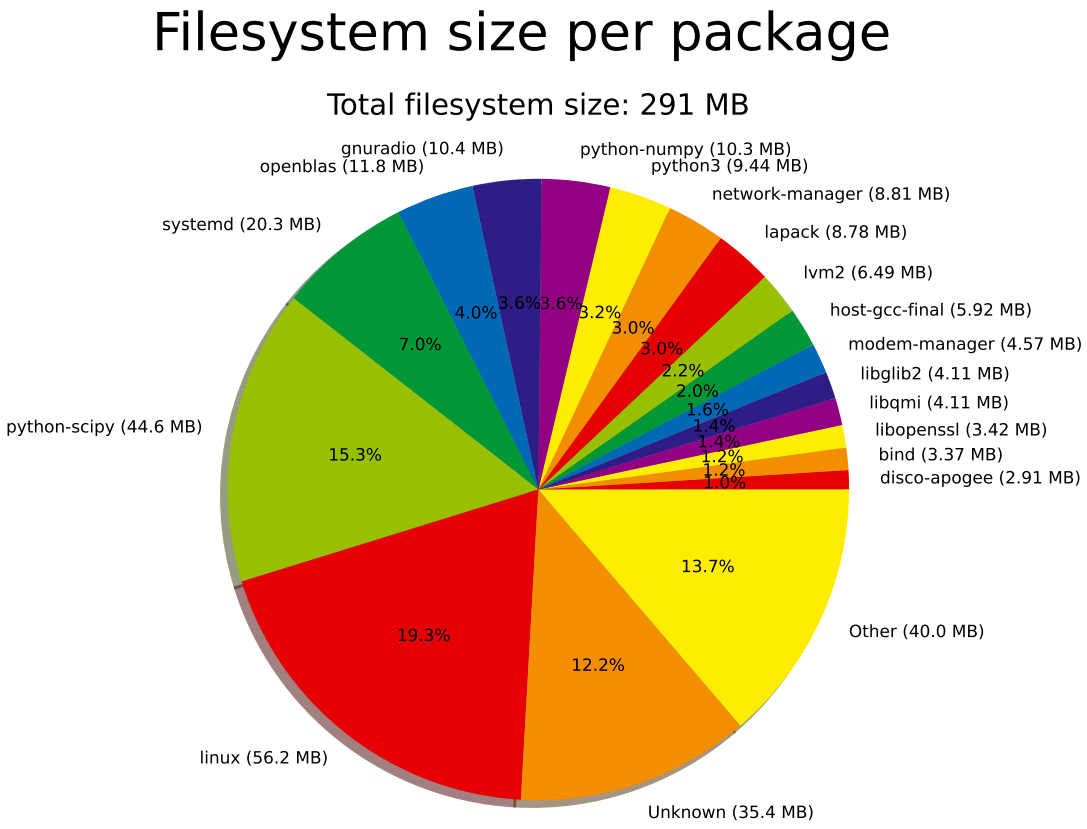


Figure 5.1.: SATOS size distribution of target files

According to the high-level breakdown of the filesystem size per package in Figure 5.1, the relevant main contributors are the following.

1. linux $\approx 56MB$
2. systemd $\approx 20MB$
3. gnuradio $\approx 96MB$ (including the following subitems)

python-scipy (Optimized scientific computing algorithms)

openblas (Basic Linear Algebra Library)

python-numpy (Fundamental mathematical functions)

python3

lapack (Linear Algebra Library)

OTA updates The bundles generated by RAUC are *squashfs* system images compressed using *zlib* [59]. They currently require $\approx 100MB$ of storage which is an acceptable size to transfer over network.

6. Future Work

With the groundwork for DISCOSAT done and the system running on a stable base, the following research topics could be of interest in the future.

6.1. Work areas

Some interesting future work areas arose during this thesis. First of all, the Yocto Project®build environment was primarily discarded due to its high entry barrier, but a comparison between these two approaches could be of interest for future projects. Moreover, provisioning the probe is a manual process and could benefit from automation in the future to ease deployment. Ideally, the PKI device secrets used in this process could be stored in a secure environment like a Trusted Platform Module (TPM) or similar to keep them secure and untampered with by third parties. Moreover, firmware updates for the clients contain the entire *rootfs* image. A delta-update approach based on a copy of the installed system might be worth investigating to minimize download size over cellular links. Additionally, no remote access to the probes exists. Therefore the design of a secure remote web shell for management and troubleshooting might be worth investigating. Lastly, an additional solution for binary task payloads similar to the Compute Core used by F@h could improve job execution performance. Moreover, they would also allow reconfiguring a probe through a smaller *appfs* update package that only contains the required files for executing the task.

6.2. Pending implementations

APOGEEs entire system management and task scheduling capabilities are pending implementation with the concepts learned and adapted from F@h and RIPE NCC Atlas. Moreover, SATOS itself needs some more maintenance work to be ready for real-world usage. Most notably, it needs automated USB storage mounting and a ramdisk based research data storage.

7. Conclusion

In this thesis, it became clear that no one-fits-all solution for a SBC powered ASSN exists. Moreover, such networks always depend on a particular software-decoding stack with specific adjustments due to their limited processing power. Current related literature does not include the software foundation work needed for an adaptable implementation. Hence, the SATOS operating system for the DISCOSAT ASSN was designed and implemented in a reproducible Buildroot build environment. It provides state-of-the-art system management software with tight integration into the D-Bus message bus, offering network connectivity (wired, WiFi, LTE), sensor localization (GNSS), and time synchronization (NTP + GNSS). Furthermore, SDR related software like *GNU Radio 3.9*, *gr-osmosdr* and *gr-iridium* with support for the HackRF One platform is available on the probe. Moreover, it allows for easy extension through simple-to-understand Makefile configuration mechanisms and comes with many advantages over a standard Linux operating system approach. To achieve this, SATOS combines adaptability with maintainability through a well-defined tree layout, allowing true logical separation of the required build system configuration. Additionally, a degree of failsafe operation is provided, as the minimal system image used by SATOS allows storing and updating two full copies of the system on the device. The small encrypted update packages are delivered through an A/B slot-based Over The Air-Update mechanism allowing reconfiguration and updating of existing probes remotely. All in all, with the future addition of the system and task management daemon APOGEE, SATOS provides a reliable and open-source software foundation for *GNU Radio* based satellite data sniffing powered by Single Board Computers.

Acronyms

AArch64	64-bit ARM cpu architecture
APN	Access Point Name
ASSN	Adaptable Satellite Sniffing Network
BOINC	Berkeley Open Infrastructure for Network Computing
CA	Certificate Authority
CC	Compute Core
CERN	Conseil Européen pour la Recherche Nucléaire
crond	Command Run On Daemon
D-Bus	Desktop Bus
DISCOSAT	Distributed Computer Systems Satellite Operating System
F@h	Folding@Home
FLOPS	Floating Point Operations Per Second
GNSS	Global Navigation Satellite System
GPIO	General Purpose Input Output
Iridium®	Iridium satellite constellation
JSON	JavaScript Object Notation
LEO	Low Earth Orbit
LTS	Long Term Support
MITM	Man In The Middle
MMC	Multi Media Card
MQ	Message Queuing
MS/s	Mega Samples per Second
NMEA	National Marine Electronics Association
NTP	Network Time Protocol
OpenWRT	Open wireless router
OTA	Over The Air
PKI	Public Key Infrastructure
QPSK	Quadrature Phase Shift Keying
RAUC	Robust Auto-Update Controller
RIPE NCC	Réseaux IP Européens Network Coordination Centre

RIR	Regional Internet Registry
ROM	Read Only Memory
RPI	Raspberry Pi
RTC	Real Time Clock
SATOS	Satellite probe Operating System
SBC	Single Board Computer
SDR	Software Defined Radio
SHM	Shared Memory
SMA	Sub-Miniature Version A
SSH	Secure Shell Protocol
SWUpdate	Software Update for Embedded System
telnetd	Teletype Network daemon
TPM	Trusted Platform Module
U-Boot	Universal Boot Loader
UART	Universal Asynchronous Receiver/Transmitter
VC	Volunteer Computing
WU	Work Unit

A. Source code listings

```
void Unit::downloadResponse(const JSON::ValuePtr &data) {
  // Check certificate, F@H usage & signature
  auto request = data->get("request");
  auto assign = data->get("assignment");
  auto wu = data->get("wu");
  string cert = wu->getString("certificate");
  string inter = wu->getString("intermediate");
  string sig64 = wu->getString("signature");
  wu = wu->get("data");
  string sigData = request->toString() + assign->toString() + wu->
    toString();

  // <Martin Boeh> The following calls App::validate later in the
    chain
  app.checkBase64SHA256(cert, inter, sig64, sigData, "WS");
  ...
  // Check data hash
  if (wu->getString("sha256") != Digest::base64(wuData, "sha256"))
    THROW("WU data hash does not match");
  ...
}

void App::validate(const Certificate &cert,
                   const Certificate &intermediate) const {
  CertificateChain chain;
  chain.add(intermediate);
  CertificateStore store;
  store.add(caCert);
  CertificateStoreContext(store, cert, chain).verify();
}
```

Listing A.1: F@h work unit signature checks [15]

```
// A payload for creating a crontab line would be
CRONTAB /home/atlas/crons/7

// File: ripe-atlas-probe-busybox/networking/telnetd.c
#define CMD_CRONTAB  "CRONTAB "
#define CMD_CRONLINE "CRONLINE "
#define CMD_ONEOFF   "ONEOFF "
```

```
#define CMD_REBOOT   "REBOOT"
...

// It gets parsed by the telnetd code:
len= strlen(CMD_CRONTAB);
if (strncmp(line, CMD_CRONTAB, len) == 0)
{
  r= start_crontab(ts, line);
    ...
  ts->state= DO_CRONTAB;
  goto skip3;
}

// This is followed up with a measurement (this is for a ping)
CRONLINE 240 342 1577836800 UNIFORM 14 evping -6 -c 3 -A "2001" -O
    /home/atlas/data/new/7 ipv6addr

// A firmware update looks like this:
FIRMWARE_APPS 4550
1b0fb537ec86f99c9247bf84bce29570ddf53ca9f82cd0efa8f21196ffe9d6b2
   app_tlmr3020_4550.img.bz2
```

Listing A.2: RIPE NCC Atlas telnetd protocol excerpt taken from [22] and [20]

```
test -n "${BOOT_ORDER}" || setenv BOOT_ORDER "A B"
test -n "${BOOT_A_LEFT}" || setenv BOOT_A_LEFT 3
test -n "${BOOT_B_LEFT}" || setenv BOOT_B_LEFT 3
test -n "${bootargs_satos}" || setenv bootargs_satos rootwait
    console=tty1 console=ttyAMA0,115200 fsck.repair=yes panic=2

setenv bootargs_a "root=/dev/mmcblk0p2 rootfstype=ext4 ro"
setenv bootargs_b "root=/dev/mmcblk0p3 rootfstype=ext4 ro"

# Preserve origin bootargs
setenv bootargs_rpi
setenv fdt_org ${fdt_addr}
fdt addr ${fdt_org}
fdt get value bootargs_rpi /chosen bootargs

setenv bootargs
for BOOT_SLOT in "${BOOT_ORDER}"; do
  if test "x${bootargs}" != "x"; then
    # skip remaining slots
  elif test "x${BOOT_SLOT}" = "xA"; then
    if test ${BOOT_A_LEFT} -gt 0; then
      setexpr BOOT_A_LEFT ${BOOT_A_LEFT} - 1
      echo "Trying to boot slot A, ${BOOT_A_LEFT} attempts
    remaining. Loading kernel ..."
      if load ${devtype} ${devnum}:2 ${kernel_addr_r} boot/Image;
    then
          setenv bootargs "${bootargs_satos} ${bootargs_rpi} ${
    bootargs_a} rauc.slot=A"
      fi
    fi
  elif test "x${BOOT_SLOT}" = "xB"; then
    if test ${BOOT_B_LEFT} -gt 0; then
      setexpr BOOT_B_LEFT ${BOOT_B_LEFT} - 1
      echo "Trying to boot slot B, ${BOOT_B_LEFT} attempts
    remaining. Loading kernel ..."
      if load ${devtype} ${devnum}:3 ${kernel_addr_r} boot/Image;
    then
          setenv bootargs "${bootargs_satos} ${bootargs_rpi} ${
    bootargs_b} rauc.slot=B"
      fi
    fi
  fi
done

setenv fdt_addr
if test -n "${bootargs}"; then
  saveenv
else
  echo "No valid slot found, resetting tries to 3"
```

```
43   setenv BOOT_A_LEFT 3
44   setenv BOOT_B_LEFT 3
45   saveenv
46   reset
47 fi
48
49 echo "Starting kernel"
50 booti ${kernel_addr_r} - ${fdt_org}
51
52 echo "Boot failed, resetting..."
53 reset
```

Listing A.3: U-Boot script for failsafe booting (inspired by RAUC)[60]

```
# Changes in package/gr-osmosdr/Config.in
config BR2_PACKAGE_GR_OSMOSDR_HACKRF
    bool "Osmocom HACKRF support"
    depends on BR2_PACKAGE_HACKRF
    help
      Enable Osmocom HackRF support

# Compile time def. added in package/gr-osmosdr/gr-osmosdr.mk
ifeq ($(BR2_PACKAGE_GR_OSMOSDR_HACKRF),y)
    # Enable the hackrf backend
    GR_OSMOSDR_CONF_OPTS += -DENABLE_HACKRF=ON

    # Depend on the hackrf driver/firmware package
    GR_OSMOSDR_DEPENDENCIES += hackrf
else
    GR_OSMOSDR_CONF_OPTS += -DENABLE_HACKRF=OFF
endif

# If python is selected, add the host-python-six dependency
ifeq ($(BR2_PACKAGE_GR_OSMOSDR_PYTHON),y)
...
GR_OSMOSDR_DEPENDENCIES += python3 host-python-six
...
endif
```

Listing A.4: *gr-osmosdr* HackRF support

```
#!/bin/bash
set -e

RAUC_PKI_OPTIONS="--cert ${BR2_EXTERNAL_SATOS_PATH}/ota/dev-ca.pem --key ${BR2_EXTERNAL_SATOS_PATH}/ota/dev-key.pem"
RAUC_CERT_NAME="${BR2_EXTERNAL_SATOS_PATH}/ota/dev-ca.pem"
RAUC_BUNDLE_BASE_FILENAME="satos-${BOARD_NAME}-${VERSION}"

if [ "$DEPLOYMENT_MODE" == "production" ]; then
    RAUC_PKI_OPTIONS="--cert ${BR2_EXTERNAL_SATOS_PATH}/ota/prod-ca.pem --key ${BR2_EXTERNAL_SATOS_PATH}/ota/prod-key.pem"
    RAUC_CERT_NAME="${BR2_EXTERNAL_SATOS_PATH}/ota/prod-ca.pem"
fi


function rauc_generate_root_bundle {
    ROOTFS_PATH=${BINARIES_DIR}/${RAUC_BUNDLE_BASE_FILENAME}-rootfs.raucb
    [ -e ${ROOTFS_PATH} ] && rm -rf ${ROOTFS_PATH}
    [ -e ${BINARIES_DIR}/temp-rootfs ] && rm -rf ${BINARIES_DIR}/temp-rootfs
    mkdir -p ${BINARIES_DIR}/temp-rootfs

    cat >> ${BINARIES_DIR}/temp-rootfs/manifest.raucm << EOF
[update]
compatible=satos-${BOARD_NAME}
version=${VERSION}
[bundle]
format=verity
[image.rootfs]
filename=rootfs.ext4
EOF

    ln -L ${BINARIES_DIR}/rootfs.ext4 ${BINARIES_DIR}/temp-rootfs/

    # Generate OTA for rootfs
    ${HOST_DIR}/bin/rauc bundle ${RAUC_PKI_OPTIONS} ${BINARIES_DIR}/temp-rootfs/ ${ROOTFS_PATH}
}


function rauc_generate_boot_bundle {
    # Generate a RAUC update bundle for the boot filesystem
    BOOTFS_PATH=${BINARIES_DIR}/${RAUC_BUNDLE_BASE_FILENAME}-bootfs.raucb
    [ -e ${BOOTFS_PATH} ] && rm -rf ${BOOTFS_PATH}
    [ -e ${BINARIES_DIR}/temp-bootfs ] && rm -rf ${BINARIES_DIR}/temp-bootfs
    mkdir -p ${BINARIES_DIR}/temp-bootfs
```

```
43
44      cat >> ${BINARIES_DIR}/temp-bootfs/manifest.raucm << EOF
45  [update]
46  compatible=satos-${BOARD_NAME}
47  version=${VERSION}
48  [bundle]
49  format=verity
50  [image.bootloader]
51  filename=boot.vfat
52  EOF
53
54      ln -L ${BINARIES_DIR}/boot.vfat ${BINARIES_DIR}/temp-bootfs/
55
56      # Generate rauc bundle for bootfs
57      ${HOST_DIR}/bin/rauc bundle ${RAUC_PKI_OPTIONS} ${BINARIES_DIR
        }/temp-bootfs/ ${BOOTFS_PATH}
58  }
59
60
61  function rauc_copy_keyring {
62      cp "${RAUC_CERT_NAME}" "${TARGET_DIR}/etc/rauc/keyring.pem"
63  }
```

Listing A.5: RAUC helper script for bundle creation (based on *br2rauc*)[53]

B. Logs and Configs

```
├── board
│   ├── common # Files used by multiple boards.
│   │   │   busybox.fragment # Busybox config fragment
│   │   │   device_table.txt # Special permissions for systemfiles
│   │   └── rootfs-overlay # Overlay for the file system
│   │       ├── etc
│   │       │   │   chrony.conf # chronyd configuration
│   │       │   ├── NetworkManager # NM configuation
│   │       │   │   │   NetworkManager.conf
│   │       │   │   └── system-connections
│   │       │   │           congstar.nmconnection # SIM conf.
│   └── raspberrypi # Files for multiple RPI models
│       │   linux.fragment # Conf. Fragment for the Linux Kernel
│       │   uboot.fragment # Conf. Fragment for U-Boot
│       │   uboot.ush # Bootloader script containing the A/B logic
│       └── rpi3-64 # Config files/scripts for RPI 3 (aarch64)
│           │   config_fw.txt # RPI firmware configuration
│           │   genimage-rpi3-64.cfg # Filesystem layout
│           │   post-[build|image].sh # Device specific scripting
│           └── rootfs-overlay
│               ├── etc
│               │   └── rauc # OTA-Firmware updater configuration
│               └── ...
├── buildroot # Custom BR fork submodule
├── configs # Configuration files for supported boards
│       rpi3_64_defconfig
├── package # Definitions for our new packages, used by BR
│   ├── disco-apogee
│   ├── gr-iridium
│   │       Config.in # Contains deps. and optional parameters
│   │       gr-iridium.hash # Contains source hashes for integrity
│   │       gr-iridium.mk # Makefile, tells BR how to build
│   └── ...
├── scripts # Generic scripts for image building
│       post-[build|image].sh # see above, just generalized
│       rauc.sh # Helper for the OTA-Firmware update images
```

```
└──src # Custom packages available in source code
    └──apogee # Submodule for our system daemon
```

Listing B.1: Buildroot structure of SATOS

```
image boot.vfat {
  vfat {
    files = {
      "bcm2710-rpi-3-b.dtb",
      "bcm2710-rpi-3-b-plus.dtb",
      "bcm2837-rpi-3-b.dtb",
      "rpi-firmware/bootcode.bin",
      "rpi-firmware/cmdline.txt",
      "rpi-firmware/config.txt",
      "rpi-firmware/fixup_cd.dat",
      "rpi-firmware/start_cd.elf",
      "rpi-firmware/overlays",
      "boot.scr",
      "u-boot.bin"
    }
  }

  size = 32M
}

image data.ext4 {
  name = "data"
  ext4 {
    use-mke2fs = true
    label = "data"
    features = "^64bit"
  }
  size = 512M
}

image satos_rpi3_64_sdcard.img {
  hdimage {
  }

  partition boot0 {
    partition-type = 0xC
    bootable = true
    image = "boot.vfat"

    # Leave some space for the U-Boot environment
    offset = 64K
  }

  partition boot1 {
    image = "boot.vfat"
    in-partition-table = false

    # 32M + 64K
    offset = 32832K
```

```
50    }
51
52    partition rootfs0 {
53      partition-type = 0x83
54      image = "rootfs.ext4"
55    }
56
57    partition rootfs1 {
58      partition-type = 0x83
59      image = "rootfs.ext4"
60    }
61
62    partition data {
63      partition-type = 0x83
64      image = "data.ext4"
65    }
66  }
```

Listing B.2: Buildroot configuration for *genimage* defining probe partitions

```
discosatspy
    State: running
     Jobs: 0 queued
   Failed: 0 units
    Since: Thu 1970-01-01 00:00:02 UTC; 52 years 2 months ago
   CGroup: /
           ├─init.scope
           │ └─1 /sbin/init
           └─system.slice
             ├─ModemManager.service
             │ ├─176 /usr/sbin/ModemManager
             │ └─218 /usr/libexec/qmi-proxy
             ├─NetworkManager.service
             │ └─178 /usr/sbin/NetworkManager --no-daemon
             ├─chrony.service
             │ └─206 /usr/sbin/chronyd -n
             ├─dbus.service
             │ └─177 /usr/bin/dbus-daemon --system --address=
    systemd: --nofork --nopidfile --systemd-activation --syslog-
    only
             ├─gpsd.service
             │ └─340 /usr/sbin/gpsd -n /dev/ttyUSB1
             ├─polkit.service
             │ └─180 /usr/lib/polkit-1/polkitd --no-debug
             ├─rauc.service
             │ └─181 /usr/bin/rauc --mount=/run/rauc service
             ├─system-serial\x2dgetty.slice
             │ └─serial-getty@ttyAMA0.service
             │   ├─182 -sh
             │   ├─356 systemctl status
             │   └─357 less
             ├─systemd-journald.service
             │ └─117 /usr/lib/systemd/systemd-journald
             ├─systemd-networkd.service
             │ └─144 /usr/lib/systemd/systemd-networkd
             ├─systemd-resolved.service
             │ └─183 /usr/lib/systemd/systemd-resolved
             ├─systemd-udevd.service
             │ └─134 /usr/lib/systemd/systemd-udevd
             └─wpa_supplicant.service
               └─185 /usr/sbin/wpa_supplicant -u
```

Listing B.3: *systemctl status* output on a DISCOSAT probe

```
driftfile /data/chrony/drift
makestep 1 3

# The pool used for ntp synchronization
pool 0.pool.ntp.org iburst

# Enable rtc syncing if we have an real time clock
rtcsync

# This uses shared memory and contains the calibrated EST offset.
refclock SHM 0 refid NMEA offset 2.94e-2 precision 1e-3 poll 3
```

Listing B.4: *chrony* configuration file

```
[connection]
id=congstar
uuid=a1385913-6ccc-4d83-bd3a-272325d6c60c
type=gsm
autoconnect=false
metered=1
permissions=

[gsm]
apn=internet.telekom
password=cs
username=congstar

[ipv4]
dns-search=
method=auto

[ipv6]
addr-gen-mode=stable-privacy
dns-search=
method=auto

[proxy]
```

Listing B.5: *NetworkManager* config for a Congstar SIM card

```
#hackrf_br_rpi3.conf
[osmosdr-source]
# Center frequency for iridium l band
center_freq=1623000000

# Turn on the pre-amp
gain=14

# Moderate gains
if_gain=40
bb_gain=20

# Decrease sample-rate to 3 MS/s due to perf. issues on the RPI3b+
sample_rate=3000000

# Bandwidth defaults to auto which is sample_rate / 1.72
# bandwidth=1750000
```

Listing B.6: The gr-iridium config for RPI and the HackRF One

```
[system]
compatible=satos-rpi3-64
mountprefix=/run/rauc
statusfile=/data/rauc.db
bootloader=uboot
bundle-formats=-plain

[keyring]
path=/etc/rauc/keyring.pem

[slot.bootloader.0]
device=/dev/mmcblk0
type=boot-mbr-switch
region-start=64K
region-size=64M

[slot.rootfs.0]
device=/dev/mmcblk0p2
type=ext4
bootname=A

[slot.rootfs.1]
device=/dev/mmcblk0p3
type=ext4
bootname=B
```

Listing B.7: RAUC configuration (based on *br2rauc*)[53]

```
1 WARNING: your SDR seems to be losing samples. ~303k samples lost
    (8%)
2 01647882413 | i:   0/s | i_avg:   3/s | q_max:    0 | i_ok:   0% |
     o:    0/s | ok:   0% | ok:   0/s | ok_avg:   0% | ok:
     2 | ok_avg:   0/s | d: 0
3 WARNING: your SDR seems to be losing samples. ~164k samples lost
    (4%)
4 001647882414 | i:   0/s | i_avg:   3/s | q_max:    0 | i_ok:   0%
     | o:    0/s | ok:   0% | ok:   0/s | ok_avg:   0% | ok:
       2 | ok_avg:   0/s | d: 0
5 WARNING: your SDR seems to be losing samples. ~404k samples lost
    (10%)
6 001647882415 | i:   0/s | i_avg:   3/s | q_max:    0 | i_ok:   0%
    | o:    0/s | ok:   0% | ok:   0/s | ok_avg:   0% | ok:
      2 | ok_avg:   0/s | d: 0
7 WARNING: your SDR seems to be losing samples. ~306k samples lost
    (8%)
8 001647882416 | i:   0/s | i_avg:   3/s | q_max:    0 | i_ok:   0%
    | o:    0/s | ok:   0% | ok:   0/s | ok_avg:   0% | ok:
      2 | ok_avg:   0/s | d: 0
9 WARNING: your SDR seems to be losing samples. ~249k samples lost
    (6%)
10 001647882417 | i:   0/s | i_avg:   3/s | q_max:    0 | i_ok:   0%
     | o:
```

Listing B.8: SATOS RPI sample loss at 4 MS/s

Bibliography

[1] L. Doolittle, H. Ma, and M. S. Champion, "Digital low-level rf control using non-iq sampling," in *Proceedings of LINAC*, Citeseer, vol. 568, 2006, p. 570.

[2] G. S. GADGETS, *Hackrf one*, https://greatscottgadgets.com/hackrf/one/. (visited on 03/29/2022).

[3] K. Maine, C. Devieux, and P. Swan, "Overview of iridium satellite network," in *Proceedings of WESCON'95*, Nov. 1995, pp. 483–. DOI: 10.1109/WESCON.1995.485428.

[4] S. R. Pratt, R. A. Raines, C. E. Fossa, and M. A. Temple, "An operational and performance overview of the iridium low earth orbit satellite system," *IEEE Communications Surveys*, vol. 2, no. 2, pp. 2–10, 1999. DOI: 10.1109/COMST.1999.5340513.

[5] H. Liu, A. Ghafoor, and P. Stockmann, "A new quadrature sampling and processing approach," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 25, no. 5, pp. 733–748, Sep. 1989, ISSN: 1557-9603. DOI: 10.1109/7.42090.

[6] *GNU Radio Website*. [Online]. Available: http://www.gnuradio.org (visited on 03/29/2022).

[7] R. P. Hudhajanto, H. Wijanarko, M. Arifin, *et al.*, "Low cost nano satellite communication system using gnuradio, hackrf, and raspberry pi," in *2018 International Conference on Applied Engineering (ICAE)*, 2018, pp. 1–4. DOI: 10.1109/INCAE.2018.8579395.

[8] G. J. Minden, J. B. Evans, L. Searl, *et al.*, "Kuar: A flexible software-defined radio development platform," in *2007 2nd IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks*, Apr. 2007, pp. 428–439. DOI: 10.1109/DYSPAN.2007.62.

[9] *Boost c++ libraries*, https://www.boost.org/. (visited on 03/29/2022).

[10] G. Oligeri, S. Sciancalepore, and R. Di Pietro, "Gnss spoofing detection via opportunistic iridium signals," in *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '20, Linz, Austria: Association for Computing Machinery, 2020, pp. 42–52, ISBN: 9781450380065. DOI: 10.1145/3395351.3399350. [Online]. Available: https://doi.org/10.1145/3395351.3399350.

[11] M. Schäfer, M. Strohmeier, V. Lenders, I. Martinovic, and M. Wilhelm, "Demonstration abstract: Opensky: A large-scale ads-b sensor network for research," in *Proceedings of the 13th International Symposium on Information Processing in Sensor Networks (IPSN '14)*, Berlin, Germany: IEEE Press, Apr. 2014, pp. 313–314, ISBN: 978-1-4799-3146-0. [Online]. Available: https://opensky-network.org/files/publications/ipsn2014_demo.pdf.

[12] A. Beberg, D. Ensign, G. Jayachandran, S. Khaliq, and V. Pande, "Folding@home: Lessons from eight years of volunteer distributed computing," May 2009, pp. 1–8. DOI: 10.1109/IPDPS.2009.5160922.

[13] Folding@Home, *A significant milestone – 100 petaflops*, https://foldingathome.org/2016/07/19/a-significant-milestone-100-petaflops/?lng=en. (visited on 03/29/2022).

[14] D. P. Anderson, "BOINC: A platform for volunteer computing," *CoRR*, vol. abs/1903.01699, 2019. arXiv: 1903.01699. [Online]. Available: http://arxiv.org/abs/1903.01699.

[15] FoldingAtHome, *Fah-client-bastet*, https://github.com/FoldingAtHome/fah-client-bastet. (visited on 03/29/2022).

[16] M. K. Robert Kisteleki, *Ripe atlas probes as iot devices*, https://labs.ripe.net/author/kistel/ripe-atlas-probes-as-iot-devices/, Oct. 2017. (visited on 03/29/2022).

[17] R. NCC, *Ripe atlas - statistics*, https://atlas.ripe.net/. (visited on 03/29/2022).

[18] *Busybox website*, https://busybox.net/. (visited on 03/29/2022).

[19] *Ripe atlas telnetd source code*, https://github.com/RIPE-NCC/ripe-atlas-probe - busybox / blob / fb8bc976a326f894b16e859124dd27e865fd3ccc / networking/telnetd.c. (visited on 03/29/2022).

[20] *Ripe atlas source code repositories*, https://github.com/RIPE-NCC/. (visited on 03/29/2022).

[21] P. Homburg, *Releasing ripe atlas measurements source code*, https://labs.ripe.net/author/philip_homburg/releasing-ripe-atlas-measurements-source-code/, Oct. 2013. (visited on 03/29/2022).

[22] ——, *Ripe ncc atlas internals*, https://www.ietf.org/proceedings/interim/2013/10/14/nmrg/slides/slides-interim-2013-nmrg-1-0.pdf, Oct. 2013. (visited on 03/29/2022).

[23] D. Project, *Snapshot.debian.org*, https://snapshot.debian.org/. (visited on 03/29/2022).

[24] *The yocto project®*, https://www.yoctoproject.org/. (visited on 03/29/2022).

[25] *The yocto project®- overview*, https://www.yoctoproject.org/software-overview/. (visited on 03/29/2022).

[26] Buildroot, *The buildroot user manual*, https://buildroot.org/downloads/manual/manual.html. (visited on 03/29/2022).

[27] *Buildroot lts 2022.02 source code*, https://github.com/buildroot/buildroot/tree/2022.02. (visited on 03/29/2022).

[28] G. Goavec-Merou, "Gnuradio running on embedded boards: Porting to buildroot," *Proceedings of the GNU Radio Conference*, vol. 1, no. 1, 2021. [Online]. Available: https://pubs.gnuradio.org/index.php/grcon/article/view/86.

[29] *Openwrt - about*, https://openwrt.org/about. (visited on 03/29/2022).

[30] J. Diamond and K. Martin, "Managing a Real-time Embedded Linux Platform with Buildroot," in *Proc. of International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'15), Melbourne, Australia, 17-23 October 2015*, (Melbourne, Australia), ser. International Conference on Accelerator and Large Experimental Physics Control Systems, doi:10.18429/JACoW-ICALEPCS2015-WEPGF096, Geneva, Switzerland: JACoW, Dec. 2015, pp. 926–929, ISBN: 978-3-95450-148-9. DOI: doi:10.18429/JACoW-ICALEPCS2015-WEPGF096. [Online]. Available: http://jacow.org/icalepcs2015/papers/wepgf096.pdf.

[31] E. Blázquez, S. Pastrana, Á. Feal, *et al.*, "Trouble over-the-air: An analysis of fota apps in the android ecosystem," in *2021 IEEE Symposium on Security and Privacy (SP)*, May 2021, pp. 1606–1622. DOI: 10.1109/SP40001.2021.00095.

[32] A. Sforzin, F. G. Mármol, M. Conti, and J.-M. Bohli, "Rpids: Raspberry pi ids — a fruitful intrusion detection system for iot," in *2016 Intl IEEE Conferences on Ubiquitous Intelligence Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, Jul. 2016, pp. 440–448. DOI: 10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.0080.

[33] Waveshare, *Sim7600ce-t/e-h/a-h/sa-h/g-h 4g modules*, https://www.waveshare.com/wiki/SIM7600E-H_4G_HAT, 2022. (visited on 03/29/2022).

[34] *Buildroot lts 2022.02 rpi partitioning*, https://github.com/buildroot/buildroot/blob/2022.02/board/raspberrypi/genimage-raspberrypi3-64.cfg. (visited on 03/29/2022).

[35] I. Amirtharaj, T. Groot, and B. Dezfouli, "Profiling and improving the duty-cycling performance of linux-based iot devices," *Journal of Ambient Intelligence and Humanized Computing*, vol. 11, no. 5, pp. 1967–1995, May 2020, ISSN: 1868-5145. DOI: 10.1007/s12652-019-01197-2. [Online]. Available: https://doi.org/10.1007/s12652-019-01197-2.

[36] *The u-boot documentation*, https://u-boot.readthedocs.io/en/latest/. (visited on 03/29/2022).

[37] *Upstream raspberry pi 4 b support github.com/lategoodbye/rpi-zero*, https://github.com/lategoodbye/rpi-zero/issues/43. (visited on 03/29/2022).

[38] *Raspberry pi foundation linux kernel*, https://github.com/raspberrypi/linux/tree/0efbe86e7248ad9b80a42b37a91c44860f91eee4. (visited on 03/29/2022).

[39] K. Rao, J. Wang, S. Yalamanchili, Y. Wardi, and H. Ye, "Application-specific performance-aware energy optimization on android mobile devices," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2017, pp. 169–180. DOI: 10.1109/HPCA.2017.32.

[40] D. Mills, "Internet time synchronization: The network time protocol," *IEEE Transactions on Communications*, vol. 39, no. 10, pp. 1482–1493, Oct. 1991, ISSN: 1558-0857. DOI: 10.1109/26.103043.

[41] *Gpsd cgps*, https://gpsd.gitlab.io/gpsd/cgps.html. (visited on 03/29/2022).

[42] *Gnu radio swig to pybind11 transition*, https://github.com/gnuradio/greps/blob/main/grep-0015-remove-swig.md. (visited on 03/29/2022).

[43] *Gr-iridium: 3.8 maintenance error*, https://github.com/muccc/gr-iridium/issues/125. (visited on 03/29/2022).

[44] *Vector-optimized libary of kernels*, https://www.libvolk.org/. (visited on 03/29/2022).

[45] *Gnu radio issue 5455*, https://github.com/gnuradio/gnuradio/issues/5455. (visited on 03/29/2022).

[46] *Mirror of the satos buildroot fork*, https://github.com/MartB/buildroot-satos. (visited on 03/29/2022).

[47] *Gr-iridium source code repository*, https://github.com/muccc/gr-iridium/tree/2022-02-02_maint-3.9. (visited on 03/29/2022).

[48] *Scipy - website*, https://scipy.org/. (visited on 03/29/2022).

[49] *Buildroot patchworks python-scipy*, https://patchwork.ozlabs.org/project/buildroot/patch/20220222125724.11079-4-guillaume.bressaix@gmail.com/. (visited on 03/29/2022).

[50] *Networkmanager*, https://networkmanager.dev/. (visited on 03/29/2022).

[51] *Go d-bus bindings for networkmanager*, https://github.com/Wifx/gonetworkmanager. (visited on 03/29/2022).

[52] *Linux wpa_supplicant*, https://w1.fi/wpa_supplicant/. (visited on 03/29/2022).

[53] *Cdsteinkuehler/br2rauc: Buildroot + rauc*, https://github.com/cdsteinkuehler/br2rauc. (visited on 03/29/2022).

[54] *Rauc reference*, https://rauc.readthedocs.io/en/latest/reference.html. (visited on 03/29/2022).

[55] *Go d-bus*, https://github.com/godbus/dbus. (visited on 03/29/2022).

[56] *Introduction to dbus*, https://www.freedesktop.org/wiki/IntroductionToDBus/. (visited on 03/29/2022).

[57] *Gpsd*, https://gpsd.gitlab.io/gpsd/gpsd.html. (visited on 03/29/2022).

[58] *Haos buildroot external makefile*, https://github.com/home-assistant/operating-system/blob/7.5/Makefile. (visited on 03/29/2022).

[59] L.-C. Duca, A. Duca, and C. Popescu, "Ota secure update system for iot fleets," *International Journal of Advanced Networking and Applications*, vol. 13, no. 03, pp. 4988–4992, 2021. DOI: 10.35444/ijana.2021.13307.

[60] *Rauc - uboot.sh*, https://github.com/rauc/rauc/blob/v1.6/contrib/uboot.sh. (visited on 03/29/2022).