Using Genode OS for Remotely Updated Embedded IoT Devices

Maximilian Matthé*
Paul Kühne*
maximilian.matthe@barkhauseninstitut.org
paul.kuehne@barkhauseninstitut.org
Barkhausen Institut gGmbH
Dresden, Saxony, Germany

Johannes Schlatow
johannes.schlatow@genode-labs.com
Genode Labs GmbH
Dresden, Saxony, Germany

ABSTRACT

This paper demonstrates the usage of the Genode OS for wirelessly connected, embedded IoT devices. We show that the modular Genode OS makes the system robust against corrupted or erroneous updates and provides builtin rollback functionality. It enables an increased uptime of the device compared to using an embedded Linux OS. We compare the performance of such device against an embedded Linux. The claimed properties are presented in the form of a demonstrator which can initiate discussions among experts from academia and industry.

KEYWORDS

Resilience, Microkernel, Genode, FPGA, Real-time wireless, Robotics, Demonstrator

1 INTRODUCTION

With the advent of applications in the areas of smart home, smart factory or e-Health, the Internet of Things (IoT) became part of our daily lives. Such network consists of multitudes of connected devices, ranging from low-cost microcontrollers in smart plugs to full-fledged complex computers in modern vehicles. Besides the compute engine running the actual application on an embedded IoT device such as sensor value preprocessing, IoT devices need wireless connectivity functions such as WiFi, 4G/5G or Zigbee modems to communicate with the cloud services. Moreover, with emerging real-time applications in 5G, edge computing [13] to reduce latency became a vital part of the 5G infrastructure. For 5G femto cells [2], small 5G radio access and edge cloud computers are integrated into a single small device, similar to an embedded IoT device [6]. Therefore, such femto cell base stations also need both connectivity and compute power.

Due to real-time and power consumption constraints, most signal processing for wireless connectivity is performed on dedicated hardware such as ASICs within the IoT device. However, given the complexity of modern communication standards, an error-free hardware implementation nearly impossible. Here, using reconfigurable hardware such as FPGAs for the wireless modem allows to update the signal processing on demand. Moreover, with such hardware, an embedded device can change the employed wireless standard on the fly or adapt its signal processing to changing channel conditions [17]. Using a flexible hardware is the core part of Software Defined Radios (SDRs) and the concept of using SDRs for 5G base stations or terminals has often been proposed [1, 16].

Naturally, on highly integrated embedded IoT devices, multiple components such as e.g. IP-connectivity and sensor processing need to share a common compute hardware. Moreover, often components are provided by third parties which just use the IoT platform as a vehicle to implement their applications. However, untrusted third-party components pose the risk of interfering with other applications or the device's operating system by either malicious means or accident. Nowadays, most IoT devices employ some form of (embedded) Linux as their OS. The risk of interfering applications is well known and is usually addressed by running each component in a separate container or virtual machine [8]. However, on cheap IoT devices the resource overhead of virtualization can become prohibitively high and hence other means for isolation need to be applied.

Additionally, when embedded devices such as IoT sensors are deployed in the field it is often impossible or at least very risky to deploy updates to these remote devices [15]. Despite having been tested thoroughly, such update can potentially fail and leave the device in a bricked, unconnected and hence often unreachable state. On the other hand, with increasing complexity of modern firmwares and applications, security breaches or functional bugs become more likely also for embedded devices. Therefore, remote updates become increasingly important for IoT devices.

*Both authors contributed equally to this research.

FGBS '23, March 6–7, 2023, Dresden, Germany
© 2023 Copyright held by the authors.
https://doi.org/10.18420/fgbs2023f-01

In this setting, modular microkernel based operating systems (OSs) [14], in contrast to commonly used monolithic embedded Linux systems, offer two main advantages. First, their modularity and fine-grained control mechanisms allow to isolate different software components from each other and hence inhibit unwanted interference. Moreover, by providing fine-grained access rights to each component, the blast-radius of a failing hardware driver is limited to the clients of the driver and not the entire system. Second, many modular OSs allows individual restarting, replacing and monitoring of hardware drivers and software applications at runtime, e.g. [5, 7]. In particular, the microkernel or a dedicated management software allows updating components and hardware drivers separately, makes sure they run properly or rolls back to versions known-to-be-working without interrupting the functionality of unaffected components. Although micro-rebooting and live-patching has been retrofitted into Linux, the resulting solutions are complex and require considerable care and expertise.

Given these two main benefits, namely component isolation and resilient, component-wise updates, modular OSs are very suitable for remotely deployed IoT devices [10]. This paper evaluates the integration of a modular microkernel OS (the Genode OS framework [7]) with a software defined radio device (USRP E310) to build a generic embedded IoT device that supports reliable remote updates. To this end, we implemented a generic real-time wireless transmission scheme [9] and integrated it into the operating systems IP stack via a TAP device. Moreover, we extended the Genode OS to support the Xilinx Zynq7020 architecture of the USRP E310 and in particular enable usage of the FPGA of the device. To evaluate the resulting performance, we compare communication latency and throughput against a monolithic embedded Linux system. Moreover, we employ the device in an abstract factory automation scenario and evaluate above claims of reliability and resilience in a real use case.

The remainder of this paper is structured as follows: Section II describes the system architecture and provides fundamendal performance comparison with a monolithic Linux system. Section III introduces the demonstrator of a factory automation use-case and provides application-specific measurements, namely HTTP round trip latency. The paper is concluded in Section IV.

2 SYSTEM ARCHITECTURE

The Genode OS framework [4] is a toolkit that implements the Genode architecture [7]. Within this architecture, a microkernel sits at the root of a tree of software components. Each node in the tree is granted specific access rights and a ressource budget from their parent, and each node can forward resources or access rights to its children. This leads to a recursive structure of component hierarchies. Within each node, a software component runs in its own sandbox, and communication with other nodes is only possible via the granted access rights. On the one hand, this rigid structure reduces the attack surface of security-critical functions by orders of magnitude. On the other hand, it allows exchanging, restarting and rolling back failing software components at runtime without affecting non-related nodes.

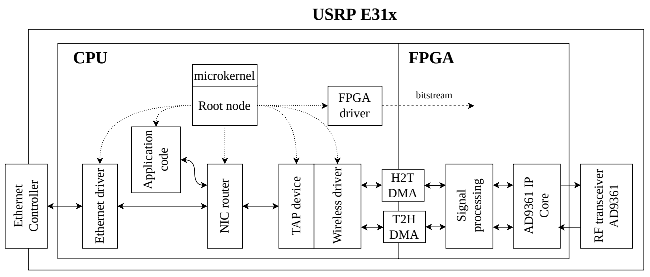

Figure 1: System structure of embedded IoT device.

The hardware platform used for our implementation is the USRP E310, which combines a Zynq7020 SoC with an AD9361 radio frontend. The ressources of such device in terms of FPGA slices and compute power are relatively limited which makes it a good example for a remotely deployed, small IoT hardware. The Zynq7020 SoC consists of a CPU (PS, programming system) and an FPGA (PL, programmable logic) well integrated into a single SoC. The Host part of the system runs the Genode OS. The Genode OS integrates the the FPGA via register and DMA interfaces, where details are documented in [11, 12]. Within the FPGA, the Analog Devices AD9361 IP core [3] is integrated and the signal processing of the wireless system [9] is implemented. The FPGA performs channel coding and decoding, symbol mapping, synchronization, channel estimation and equalization and CRC checking. Higher-layer wireless functionality such as PHY layer packet fragmentation or packet loss detection is performed in the Host system. The wireless system exposes itself as a TAP device within the OS system, which allows transparent IP traffic to be routed via the wireless link. 1 illustrates the implemented structure.

Fig. 2 compares the round trip time on the MAC layer for different MAC packet sizes between two wirelessly connected devices. Since the difference is very small, no real performance difference is observed. However, for small packets, the Genode system obtains a 0.1ms shorter latency, whereas for larger packets, Linux becomes faster. Since the difference is minimal and not relevant for the use case at hand, we have not further investigated the root cause.

Fig. 3 compares wireless throughput of MAC frames of different sizes between two wirelessly connected devices. Again, the difference is marginal and the overall obtained

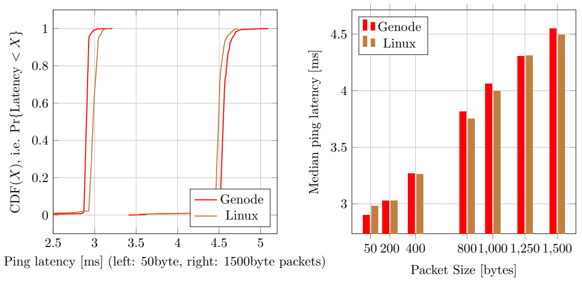

Figure 2: Round-Trip Latency on the MAC layer.

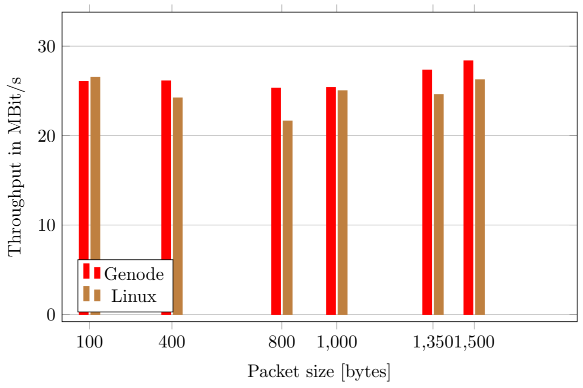

Figure 3: Throughput comparison.

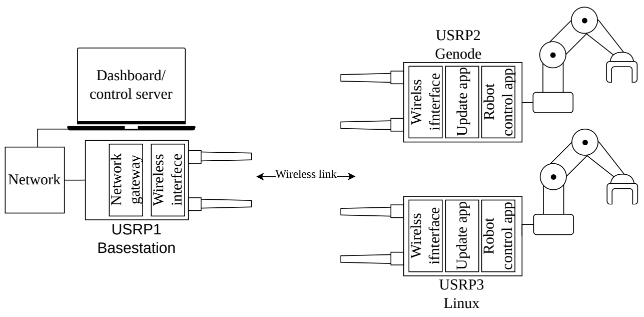

Figure 4: Demonstrator structure.

throughput matches the maximum rate of our wireless connection. Having a constant throughput over packet sizes shows that the limiting factor is indeed the wireless network, i.e. both systems can fully load the wireless interface. The variations are due to differing channel qualities during the measurements and RF effects such as frequency offsets which were not completely calibrated. The obtained throughput was fully sufficient for the application at hand and hence the wireless interface was not optimized further.

3 APPLICATION SCENARIO

In the present demonstrator, we directly compare the properties of an IoT device running monolithic Linux and a Genode-powered device. The demonstrator resembles a generic factory automation scenario. A wireless network connects a base station and two terminals, one running Linux, one running Genode. Each terminal is attached to a robotic arm. A 3rd party robot application running on each terminal controls the robot arm's movement. Attached to the base station is a central control server which communicates with the 3rd party apps on the terminals and instructs them to start or stop their movement. As the robot application on the terminals directly controls the robot arm's movement, a moving arm indicates that the robot app is currently active and running. In addition, a management application is running on each terminal which periodically asks a central server if it should perform an update of the wireless driver on the terminal. In reality, such updates could either be a bug-fixing release for the wireless driver or a change of the wireless system to adapt to different channel conditions. If an update should be installed, the management app downloads the update binary, stops the wireless driver, replaces the binary and restarts the wireless driver. While the wireless driver is down, the device is not reachable over wireless.

The user can interact with the demonstrator in two ways: First, the movement of each robot can be started or stopped from a dashboard which also displays the current state of the setup. Moreover, update requests can be set up for both systems. The user can choose between a good, i.e. error-free update and a bad update, i.e. an update that contains an error which inhibits the wireless component to start.

While interacting with the demonstrator, the visitor can directly experience the claimed benefits of a microkernel based OS. First, during normal operation, no difference between Linux and Genode is visible. The robot movements can be stopped and started freely from the dashboard and the dashboard indicates that both systems are online. The interesting part happens during updates:

When an error-free update is to be installed, both systems download the wireless driver binary, stop the wireless driver, and replace the binary. As soon as the driver is stopped, the wireless connection is not available for both systems. During the installation, the robot control app still runs, letting the robot arm still move. To restart the driver, the Linux system needs to be rebooted since the updates touched critical system components that require a reboot to be updated. During the reboot time, the robot app is also stopped, which also stops the robot arm's movement. After the Linux system has rebooted, wireless connectivity becomes available again and the robot arm starts moving again. Due to modularity of the Genode system, its wireless driver component can be exchanged and restarted without rebooting the system or interfering with other applications on the system. Therefore, during the entire update the robot arm keeps moving, indicating that the overall system is alive.

In case, an erroneous update is installed, the same process as before runs. However, as soon as Linux reboots, the device is bricked, because the update corrupted a core system component which inhibits start of the system. Since the device is not booting, no wireless connectivity is available and hence it is not reachable from the outside. The only way to fix this state is to literally go to the device, check the error status and fix it manually. Such event will incur a long downtime to the device and hence to the manufacturing line. In strong contrast, the Genode OS only restarts the driver component which also fails completely. However, since a second management process on the modular OS is running, the driver is overseen and rolled back to a working version in case the driver crashes. During the entire process, the robot performs its movement and the system is not bricked by the update. Hence, no downtime of the manufacturing line is incurred.

It is important to note that also in Linux such fail-safe methods and rollback mechanisms can be established. Often, such rollback is implemented in the bootloader which boots into two different firmwares, depending on a flag. This flag is set when an update is performed and cleared when the system booted correctly. The bootloader can oversee the boot process and roll back to a known version, if the flag is not cleared within an expected time frame after booting. However, despite being simple to grasp conceptually, the implementation of such fail-safe mechanisms is complex, needs to be tested thoroughly on each new hardware device and can itself introduce bugs or vulnerabilities. In contrast, the properties of the Genode OS allow reusing the management component on all hardware and no special rollback implementation for different hardware is required.

In the demonstrator, due to simplicity the applications communicate via the HTTP protocol. The dashboard server runs the HTTP server and the terminals periodically ask the dashboard for new movement and update commands. Fig. 5 shows the measurement results of the HTTP request latency from the robot control app to the dashboard server. Genode uses a port of the Linux IP Stack from Linux kernel v4.4.3. Linux uses the default IP stack with kernel v4.14.0. All IP stacks were in default configuration. As visible, the latencies vary mainly in the low latency-regions, i.e. where no TCP retransmissions where requested. There, Linux outperforms Genode as median latencies are 18ms for Linux and 27ms for Genode. The root cause was not of practical relevance for the use case at hand and therefore not investigated further.

The latencies ranged up to 5s, before the HTTP request timed out. Considering the fact that the jumps in the CDF

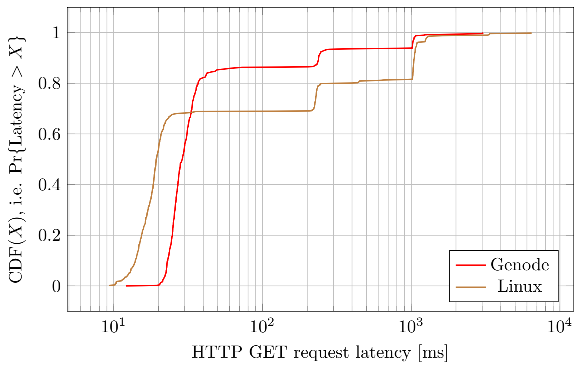

Figure 5: Comparison of HTTP request latency.

of Linux and Genode latency occur at the same times confirms that the TCP timeout of both stacks were configured equally. In fact, the main jumps at 200ms and 1000ms in the CDF correspond to TCP's default retransmission timeouts for established and non-established connections[1]. The height difference of the plateaus is caused by different packet loss in Layer 1 for the two measurements. Packet loss varied between 15%-30% due to different channel qualities and carrier frequency offset calibrations of the wireless system. It is clear that TCP is not built for such high loss rates and therefore the results might differ in more realistic scenarios. However, for the application at hand, the results obtained from Genode with the Linux IP stack were sufficient and the latencies were not further investigated.

4 CONCLUSION

This paper presented a demonstrator for remote updates for microkernel based operating systems, namely the Genode OS. It compared its characteristics with these of a monolithic embedded Linux system. The implementation shows that by using a robust modular OS, the system downtime due to updating parts of the system can be significantly reduced. Moreover, the Genode OS allows to rollback failing components, leading to higher availability after a failing update. The presented demonstrator shows these effects in reality and hence can be used to illustrate benefits and initiate discussions with experts from academia and industry.

ACKNOWLEDGEMENT

Gefördert durch: Bundesministerium für Wirtschaft und Energie aufgrund eines Beschlusses des deutschen Bundestages. FuE-Projekt "Modulare Plattform für sicheren Echtzeitfunk".

[1]Consider e.g. `include/net/tcp.h`, lines 135, 136 in Linux Kernel v4.4

REFERENCES

[1] Wei Chen, Dake Liu, Yong Bai, and Rifei Yang. 2021. Design Space Exploration of SDR Vector Processor for 5G Micro Base Stations. *IEEE Access* 9 (2021), 141367–141377. https://doi.org/10.1109/ACCESS.2021.3119292

[2] Massimo Condoluci, Mischa Dohler, Giuseppe Araniti, Antonella Molinaro, and Kan Zheng. 2015. Toward 5G densenets: Architectural advances for effective machine-type communications over femtocells. *IEEE Communications Magazine* 53, 1 (2015), 134–141. https://doi.org/10.1109/MCOM.2015.7010526

[3] Adrian Costina. 2022. AXI_AD9361. https://wiki.analog.com/resources/fpga/docs/axi{_}ad9361

[4] Genode Developers. 2023. Genode OS Framework. https://github.com/genodelabs/genode

[5] Hatem A. El-Azab, Moheb R. Girgis, Essam F. Elfakharany, and Mohamed E. Shehab. 2017. Design and implementation of multicore support for a highly reliable self-repairing μ-kernel OS. *2017 IEEE 8th International Conference on Intelligent Computing and Information Systems, ICICIS 2017* 2018-Janua, Icicis (2017), 13–22. https://doi.org/10.1109/INTELCIS.2017.8260021

[6] I. Farris, A. Orsino, L. Militano, M. Nitti, G. Araniti, L. Atzori, and A. Iera. 2017. Federations of connected things for delay-sensitive IoT services in 5G environments. *IEEE International Conference on Communications* (2017), 1–6. https://doi.org/10.1109/ICC.2017.7996644

[7] Norman Feske. 2022. *The Genode Operating System Framework 22.05*. https://genode.org/documentation/genode-foundations-22-05.pdf

[8] Chris Greamo and Anup Ghosh. 2011. Sandboxing and Virtualization. *IEEE Security Privacy Magazine* 9, 2 (2011), 79–82. http://ieeexplore.ieee.org/xpls/abs{_}all.jsp?arnumber=5739643

[9] Paul Kühne. 2023. LightOFDM-FPGA-transceiver. https://gitlab.barkhauseninstitut.org/lightofdm-fpga-transceiver/lightofdm-fpga-transceiver

[10] Friedrich Pauls, Sebastian Haas, Stefan Köpsell, Michael Roitzsch, Nils Asmussen, and Gerhard Fettweis. 2022. On Trustworthy Scalable Hardware/Software Platform Design. In *Smart Systems Integration*.

[11] Johannes Schlatow. 2021. Genodians: Zynq guide #1 - getting started. http://genodians.org/jschlatow/2021-11-29-zynq-guide-1

[12] Johannes Schlatow. 2022. Genodians: Zynq guide #2 - enabling the programmable logic. http://genodians.org/jschlatow/2022-10-06-zynq-guide-2

[13] Meryem Simsek, Adnan Aijaz, Mischa Dohler, Joachim Sachs, and Gerhard Fettweis. 2016. 5G-Enabled Tactile Internet. *IEEE Journal on Selected Areas in Communications* 8716, c (2016), 1–1. https://doi.org/10.1109/JSAC.2016.2525398

[14] Andrew S Tanenbaum, Jorrit N Herder, and Herbert Bos. 2006. Can We Make Operating Systems Reliable and Secure? *Computer* (2006).

[15] Mónica M. Villegas, Cristian Orellana, and Hernán Astudillo. 2019. A study of over-the-air (OTA) update systems for CPS and IoT operating systems. *ACM International Conference Proceeding Series* 2 (2019), 269–272. https://doi.org/10.1145/3344948.3344972

[16] Xinbo Wang, Cicek Cavdar, Lin Wang, Massimo Tornatore, Hwan Seok Chung, Han Hyub Lee, Soo Myung Park, and Biswanath Mukherjee. 2017. Virtualized Cloud Radio Access Network for 5G Transport. *IEEE Communications Magazine* 55, 9 (2017), 202–209. https://doi.org/10.1109/MCOM.2017.1600866

[17] Muhammad Zeeshan and Shoab Ahmed Khan. 2016. A Novel Fuzzy Inference-Based Technique for Dynamic Link Adaptation in SDR Wideband Waveform. *IEEE Transactions on Communications* 64, 6 (2016), 2602–2609. https://doi.org/10.1109/TCOMM.2016.2560164

Received 1 February 2023; accepted 7 February 2023