RIOT - das freundliche Echtzeitbetriebssystem für das IoT

Peter Kietzmann[1], Thomas C. Schmidt[1] und Matthias Wählisch[2]

[1] Internet Technologies Group, Department Informatik
Hochschule für Angewandte Wissenschaften Hamburg, 20099 Hamburg
{peter.kietzmann|t.schmidt}@haw-hamburg.de
[2] Institut für Informatik,
Freie Universität Berlin, 14195 Berlin
m.waehlisch@fu-berlin.de

Zusammenfassung. Das „Internet der Dinge“ (IoT) beschreibt die Entwicklung, wie maschinengebundene, eingebettete Systeme schrittweise Standardprotokolle der Internet-Welt adaptieren und damit selbst Teil des globalen Inter-Netzwerks werden. Das IoT wächst gegenwärtig sehr schnell und eine zunehmende Professionalisierung erfordert den Einsatz einer Systemarchitektur, die Hardware und Kommunikationskomponenten in der Abstraktionsschicht eines Betriebssystems zusammenführt.
In dieser Arbeit stellen wir die Grundarchitektur von RIOT vor und fokussieren insbesondere auf (i) den echtzeitfähigen Mikro-Kernel mit seinem tickless Scheduler sowie (ii) den frisch aktualisierten Netzwerk-Stack von RIOT, welcher heterogene Interface-Treiber mit gängigen IoT-Protokollen in einer modular geschichteten Architektur vereint.

1 Einleitung

Das Aufkommen des „Internet der Dinge“ (IoT) [1] erfordert den Einsatz von Software, welche die Hardwareressourcen von Kleinstgeräten verwaltet und die verschiedenen „Dinge“ vernetzt. Die Wiederverwendbarkeit von Code und eine bekannte Programmierumgebung für Entwickler, beschleunigen den Entwicklungsprozess. RIOT, das freundliche Betriebssystem für das Internet der Dinge, bildet eine solche professionelle Abstraktionsschicht; seit seiner Einführung auf der IEEE INFOCOM 2013 erfreut sich dieses vollständig offene System einer schnell wachsenden Beliebtheit. RIOT zielt auf eine modulare Architektur mit anwenderfreundlicher Abstraktionsschicht, welche u.a. auf einer weitgehenden POSIX-Kompatibilität mit Unterstützung von C und C++ aufbaut.

Zu Unterscheiden sind zwei Gerätetypen: (i) Desktop Rechner, Smartphones oder Router, welche ausreichend Ressourcen haben, um herkömmliche Betriebssysteme wie Linux, BSD oder deren Derivate (z.B. uClinux oder OpenWRT) zu unterstützten, und (ii) eingebettete IoT-Geräte [2] mit eng begrenzten Ressourcen, welche trotzdem Standardprotokolle der Internetwelt adaptieren. Die Ressourcenknappheit macht es den IoT-Geräten unmöglich, etablierte Standard-Betriebssysteme, wie beispielsweise Linux, zu betreiben.

RIOT [13] gilt als Linux-ähnliches Betriebssystem für IoT-Geräte und ist im Fokus dieser Arbeit. In Kapitel 2 geben wir einen Überblick über die Anforderungen an ein IoT-Betriebssystem und die Lösungsansätze verfügbarer Alternativen. In Kapitel 3 wird das Mikro-Kernel Konzept von RIOT erläutert und seine Echtzeitfähigkeit untersucht. In Kapitel 4 stellen wir das Netzwerk-System von RIOT vor. Für den Standard Netzwerk-Stack in RIOT (GNRC) werden detaillierte Leistungsmessungen dargestellt. Die Ergebnisse werden mit Messungen existierender Netzwerk-Stacks verglichen, welche in RIOT portiert wurden.

2 Anforderungen an ein IoT-Betriebssystem

Ein Betriebssystem hat vorrangig zwei Aufgaben: (i) die Bereitstellung von Schnittstellen zu Hardware-Ressourcen und (ii) die Verwaltung dieser Ressourcen. Neben der Hardware-Verwaltung stellen Betriebssysteme Software-Stacks bereit, um Geräte miteinander zu vernetzen. Die Heterogenität verfügbarer Hardware und die knappen Ressourcen typischer Klasse 0 bis 2-Geräte [2], erschweren die Entwicklung. Durch einen hohen Grad an Hardwareabstraktion unterstützt RIOT verschiedene Hardwarearchitekturen und macht Gerätetreiber wiederverwendbar. Sein modularer Mikro-Kernel erlaubt es, auch strengste Speicheranforderungen zu erfüllen. RIOT implementiert standardisierte IoT-Protokolle. Der Scheduler erlaubt lange Schlafzyklen zur Energieoptimierung und ist trotzdem echtzeitfähig. RIOT ist unter der freien Lizenz LGPLv2 veröffentlicht.

Hardwareabstraktion Typische IoT-Geräte bestehen aus Mikrocontroller (MCU) und externer Peripherie (z.B. Sensoren oder Radios) welche über I/O Standards wie SPI oder I2C verbunden sind. Die MCU besteht aus CPU (8 bit, 16 bit oder 32 bit), Speicher und interner Peripherie. Aufgrund der umfangreichen und heterogenen Hardware, unterstützen verschiedene Betriebssysteme teilweise nur einzelne Architekturen, z.B. mbedOS [3] für 32 bit ARM Cortex M Plattformen. Contiki [4] und TinyOS [5] unterstützen heterogene Plattformen [6].

Speichereffizienz Im Gegensatz zu herkömmlichen internetfähigen Geräten, haben z.B. Klasse 1 Geräte nur ca. 10 kB RAM und 100 kB ROM [2]. Das Betriebssystem muss den verfügbaren Speicher einhalten und ausreichend Reserven für Netzwerk-Stack und Anwendung allozieren. IoT-Betriebssysteme greifen häufig auf nicht-standardisierte Programmier-Paradigmen zurück, um den Speicheraufwand zu reduzieren. Contiki basiert auf einer leichtgewichtigen, Ereignisbasierten Programmierung mittels sog. „Protothreads“ [8], während TinyOS in der C-angelehnten Sprache „nesC“ [7] programmiert wird, welche die Speicheranforderungen minimiert. Nachteile sind die Notwendigkeit besonderer Fachkenntnis der Programmierung und eine geringe Wiederverwendbarkeit des Codes. Ansätze wie Zephyr [9] setzen auf flexible Mikro- oder Nano-Kernel Architekturen, welche lediglich einige hundert Bytes in Anspruch nehmen.

Netzwerkfähigkeit Die Übertragung von IPv6 stellt Anforderungen an die Schicht 2 im Netzwerk-Stack, welche typische Übertragungsmedien im IoT nicht erfüllen (z.B. 802.15.4 oder BLE). Dies erfordert den Einsatz von standardisierten Adaptionsprotokollen [12] wie beispielsweise 6LoWPAN [11].

Energieeffizienz IoT-Geräte sind für den Batteriebetrieb ausgelegt [16]. Dies erfordert den Einsatz effizienter Hardware und ein Energie-Management, welches sicher stellt, dass: (i) MCU, Radio und Peripherie möglichst lange im Ruhemodus bleiben. Das Radio ist dabei im „Idle Sleep" Zustand zu halten, um das Empfangen von Paketen zu gewährleisten. (ii) Der Netzwerk-Stack eine effiziente Sicherungsschicht [17] und Nachrichten-sparsame Protokolle auf Netzwerk-, Transport- und Anwendungsschicht implementiert. Contiki und TinyOS unterstützen solche Mechanismen und bieten Netzwerk-Stacks mit gängigen, energiesparenden IoT-Protokollen.
Echtzeitfähigkeit und Zuverlässigkeit IoT-Szenarien wie beispielsweise taktiles Internet [18] erfordern deterministische Reaktionszeiten. Mikro-Kernel-Ansätze wie FreeRTOS [19] oder ITRON [20] erfüllen Echtzeitanforderungen und definieren eine obere Grenze für Latenzen, sind jedoch nicht für lange Schlafzeiten ausgelegt. Die Anforderung der Energieeffizienz steht oft in Konflikt mit der Reaktionszeit des Systems. Contiki und TinyOS erfüllen keine Echtzeitanforderungen.
Sicherheit & Modifizierbarkeit Um einen hohen Sicherheitsstandard zu gewährleisten, wird die Verwendung von Open Source empfohlen [21]. Contiki und TinyOS werden unter der MIT/BSD Lizenz verbreitet welche es Herstellern erlaubt, den Quelltext des Betriebssystems in proprietäre Software umzuwandeln. Dies kann die Interoperabilität gefährden und erschwert die Weiterentwicklung sowie das Fortbestehen des Codes.

3 Der Betriebssystemkern in RIOT

Mikro-Kernel Der Betriebssystemkern in RIOT ist ein Mikro-Kernel, welcher aus den Basisfunktionen Scheduler, Interprozesskommunikation (IPC) und Thread-Synchronisation (Mutex) besteht [10]. Weitere Funktionalitäten werden als Module hinzugefügt, welche über eine schlanke API mit dem Kernel kommunizieren. Diese modulare Architektur ist speicherplatzeffizient und erlaubt einfaches Konfigurieren und Erweitern des Systems. Weiterhin führt der Mikro-Kernel Ansatz zu einem hohen Grad an Stabilität, da Fehler in externen Modulen, z.B. in Treibern oder dem Netzwerk-Stack, den Kernel nicht zwangsweise zum Absturz bringen.
Scheduling und Echtzeit IoT-Geräte benötigen Mechanismen, um den Energieverbrauch zu minimieren. Der Scheduler in RIOT implementiert keine zyklischen Systemticks. Wenn das Betriebssystem keine Aufgaben abzuarbeiten hat, können MCU und angeschlossene Geräte (insbesondere Radios) beliebig lange im Ruhemodus gehalten werden, um Energie zu sparen. Das System kann nur durch externe Interrupts aufgeweckt werden, z.B. durch den I/O-Pin eines Radios oder einen Hardware-Timer.

RIOT implementiert präemptives, klassenbasiertes Prioritätsscheduling. Threads haben statische Prioritäten und werden entsprechend dieser abgearbeitet. Threads mit niedriger Priorität können unterbrochen werden, was die Notwendigkeit einer Kontextsicherung mit sich bringt. Threads mit gleicher Priorität

sind möglich und müssen die CPU-Zeit kooperativ teilen. Ansonsten behält der aktive Thread mit höchster Priorität den CPU-Zugriff bis zu seiner Abarbeitung.

In RIOT stehen 16 Prioritäten zur Verfügung. Jede Priorität wird abgebildet auf eine zirkulare, verlinkte Liste fester Grüße, welche Referenzen (Zeiger) auf abzuarbeitende Threads dieser Priorität enthält. Bei einem Kontextwechsel wird immer der erste Eintrag aus der Liste der höchsten Priorität ausgewählt. Kontextwechsel können durch unterschiedliche Ereignisse ausgelöst werden.

`thread_create()` und `thread_wakeup()`: Wird ein neuer Thread erstellt oder ein schlafender Thread aufgeweckt, wird dieser anhand seiner Priorität in die entsprechende Liste eingetragen. Hat der aufgeweckte Thread eine höhere Priorität als der derzeit aktive Thread, wird auch dieser als ausstehend in seine zugehörige Liste aufgenommen und es geschieht ein Kontextwechsel.

`thread_yield()`: Ein Thread gibt aktiv den CPU-Zugriff ab. Sein Eintrag wird der Liste an letzter Stelle angehängt. Die Referenz auf den nächsten abzuarbeitenden Thread, wird auf den zweiten Eintrag in der Liste gesetzt, womit dieser der nächste Thread ist, welcher den CPU-Zugriff erhält. Alle weiteren ausstehenden Threads in dieser Liste rücken in der Abarbeitungsreihenfolge automatisch um eine Position vor.

`thread_sleep()`: Das Schlafenlegen entfernt einen Thread aus seiner Liste. Der nächste Thread, welcher CPU-Zugriff erhält, ist der nächste abzuarbeitende Thread in dieser Prioritätsliste, oder der erste Thread aus der Liste der nächstniedrigen Priorität, usw. .

Die wesentlichen Operationen (i) Umsetzen der Referenz des nächsten abzuarbeitenden Threads in einer Liste und (ii) Auffinden des Threads mit der höchsten Priorität, sind für die Laufzeit eines Kontextwechsels verantwortlich. (i) Ist unabhängig von der Anzahl an Threads, da lediglich die Referenz auf den ersten Eintrag einer Liste verschoben und der vorangegangene Thread der Liste angehängt wird. (ii) Ist unabhängig von der Anzahl an Threads, da der Scheduler maximal 16 Prioritäten durchlaufen muss und immer den ersten Eintrag der Liste auswählt. Das Scheduling ist somit unabhängig der Systemlast und garantiert Laufzeiten in O(1). Bei geeigneter Vergabe von Thread-Prioritäten, ermöglicht das deterministische System Echtzeitfähigkeit mit RIOT.

Interrupt Abwicklung und Latenzen Die Interrupt-Latenz ist ein wichtiger Leistungsparameter für Betriebssysteme. In einem präemptiven Multi-Threading Betriebssystem müssen Interrupts durch den Kernel geleitet werden, um den Kontext laufender Threads vor Abarbeitung der Interrupt Service Routine (ISR) zu sichern. Dies ist besonders wichtig, da neue Threads mit höherer Priorität aus dem Interrupt-Kontext erstellt werden können, welche nach dem Beenden der ISR den CPU-Zugriff erhalten müssen. In RIOT ist die Abarbeitungsreihenfolge folgendermaßen geregelt: Ein Interrupt tritt auf, das System springt in den Kernel, wo der aktuelle Kontext gesichert wird, und die ISR wird im Interrupt-Kontext aufgerufen. Diese Abfolge garantiert eine begrenzte Interrupt-Latenz, unabhängig von der Systemlast. Nach der Abarbeitung kehrt die ISR in den Kernel zurück. Dort wird ermittelt, ob ein Kontextwechsel nötig ist, oder der unterbrochene Thread wieder den CPU-Zugriff erhält. Im zweiten Fall muss der

Scheduler nicht aufgerufen werden und der unterbrochene Thread erhält sofort den CPU-Zugriff. Dies trägt maßgeblich zur Verringerung der Scheduling-Latenz nach einem Interrupt bei.

4 Netzwerk-Stacks in RIOT

RIOT unterstützt verschiedene Netzwerk-Stacks, welche u.A. auf IPv6, ICN (Information-Centric Networking) [14] und 6TiSCH (IPv6 over the TSCH mode of IEEE 802.15.4e) [15] basieren. In diesem Kapitel wird der Standard IPv6 Netzwerk-Stack (GNRC) in RIOT vorgestellt, vermessen und anschließend zwei alternativen Implementierungen gegenübergestellt.

4.1 GNRC Netzwerk-Stack Aufbau

Der schematische Aufbau des GNRC Netzwerk-Stacks ist in Abbildung 1 dargestellt. *conn* ist eine Programmierschnittstelle, welche die Benutzung von Standardprotokollen von der unterliegenden Netzwerk-Stack Implementierung abstrahiert. Die *conn* API wird außerdem von einem Software-Wrapper umfasst, um POSIX Kompatibilität herzustellen. GNRC kapselt jede Protokollimplementierung in einem eigenen Thread. Zwischen den Netzwerk-Modulen kommunizieren die heterogenen Komponenten über ein einheitliches Interface, die *netapi* Schnittstelle. Der Nachrichtenaustausch basiert auf der IPC-Infrastruktur des Betriebssystems mit einer Message Queue in jedem Thread. Dies erlaubt die Einbindung unterschiedlicher Software- und Hardwaremodule oder ganzer IP-Stacks in RIOT. Ähnlich zu UNIX STREAMS [22] ist eine einfache Verkettung von Modulen durch die gemeinsame API jedes Moduls möglich. *netdev* ist eine generalisierte Gerätetreiber API zur Abstraktion von Hardware-Spezifika. Der Treiber besteht aus den Basisfunktionen `init()`, `send()/recv()`, `get()/set()` und `isr()`. *netreg* (hier nicht dargestellt) ist eine Koordinierungsinstanz, bei der sich jedes Modul/Thread des Stacks registriert. Sie regelt die Verarbeitungskette von Modul zu Modul. *pktbuf* (hier nicht dargestellt) ist der zentrale Paketspeicher mit de-duplizierender Datenhaltung. Er verwaltet Datenblöcke (*snips*) verschiedener Größen, welche einzeln bearbeitet werden können. Ein *snip* repräsentiert typischerweise die Payload bzw. die einzelnen Protokoll-Header.

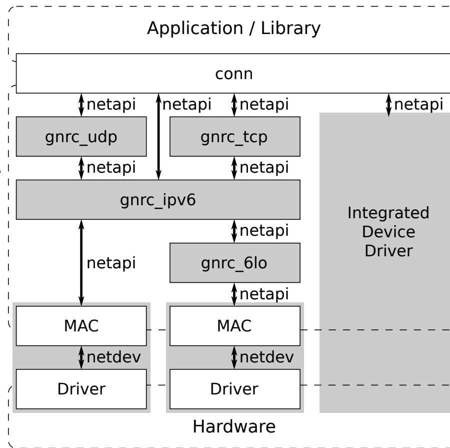

Abb. 1. Rekursive Architektur von GNRC.

4.2 Leistung einzelner Software-Schichten

Im Folgenden wird die Leistungsfähigkeit von GNRC in RIOT evaluiert und der Aufwand einzelner Komponenten quantifiziert. Die modulare Architektur des GNRC Netzwerk-Stacks mit der Aufteilung in mehrere Threads und uniformen Schnittstellen, ermöglicht einfaches Austauschen, Erweitern und Konfigurieren der Software-Komponente. Dies induziert jedoch einen Mehraufwand gegenüber monolithischen Ansätzen, insbesondere durch die Verwendung von IPC. Alle nachfolgend aufgeführten Messungen wurden auf einem iotlab-m3 IoT-Board [23] durchgeführt (Klasse 2 Gerät).

Die empirische Ermittlung der Leistungfähigkeit einzelner Software-Schichten im Netzwerk-Stack erfolgt mithilfe eines Testprogramms. Dieses versendet Datenpakete, welche an unterschiedlichen Positionen im Netzwerk-Stack reflektiert und wieder empfangen werden. Gemessen wurden Konfigurationen über (i) das IP Loopback und (ii) einen minimalen Reflektor auf Schicht 2 (L2 Reflektor). Der Gerätetreiber ist nicht Teil dieser Messung. Durch das Eingeben von Paketen über verschiedene *conn* APIs, hier UDP- und *RAW*- IPv6, kann der Aufwand einzelner Schichten geschätzt werden.

Datendurchsatz In Abbildung 2 ist der effektive Datendurchsatz einer UDP-Anwendung dargestellt. Die Ergebnisse für IP Loopback und L2 Reflektor weichen deutlich voneinander ab, was auf die 6LoWPAN Implementierung zurückzuführen ist. Der Aufwand durch die Paketfragmentierung wird durch den stufenförmigen Verlauf der L2 Reflektor Graphen sichtbar. Mit annähernd 10 Mbit/s (unidirektional) auf der IP-Schicht, ist das IoT-Gerät trotz stark begrenzter Hardware-Ressourcen in der Lage, an 802.11a/b Netzwerken teilzunehmen. Der Durchsatz von einigen Mbit/s auf Schicht 2 ist ausreichend hoch für LoWPANs. Der Vergleich zwischen RIOT *Conn* und der POSIX API zeigt einen geringen Einfluss.

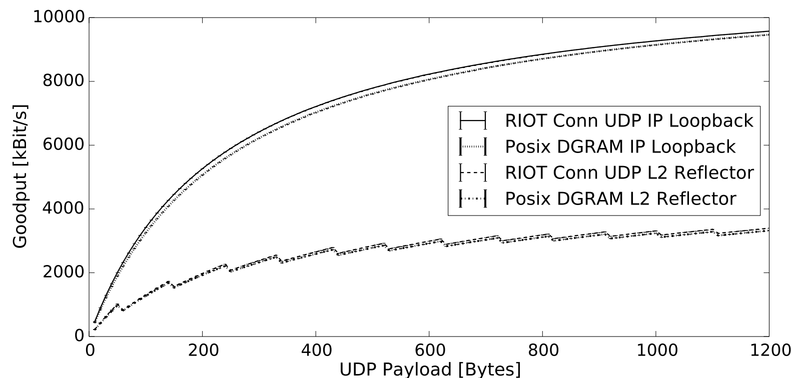

Abb. 2. Effektiver UDP-Durchsatz (Goodput), gemessen in einer Schleife. Stack auf-/abwärts mit und ohne 6LoWPAN und POSIX.

4.3 Leistung bei der Paketübertragung

Nachfolgend wird der GNRC Netzwerk-Stack inklusive Funkübertragung vermessen. Der Gerätetreiber bindet das Atmel AT86RF231 Radio an den Netzwerk-Stack an. Die Messungen wurden in einer EMV-Messkabine durchgeführt, um Interferenzen mit umliegenden Radios auf dem selben Frequenzband zu vermeiden. Mit maximal möglicher Paketrate werden UDP-Pakete von und zu einem IoT-Gerät (RIOT) übertragen. Der Kommunikationspartner ist ein Raspberry Pi (Linux), ausgestattet mit einem AT86RF233 Radio, dessen Rechenleistung Größenordnungen höher ist als die des IoT-Geräts. Der Raspberry Pi wurde gewählt, um bei der Übertragung nicht das Bremsen einer langsameren Gegenstelle auszumessen. Zusätzlich wurde eine Messung zwischen zwei Raspberry Pi Boards durchgeführt.

Datendurchsatz Abbildung 3 vergleicht den UDP-Durchsatz der Datenübertragung mit der theoretisch oberen Grenze. Diese wurde anhand der Datenrate von 802.15.4 ermittelt (250 kbit/s), abzüglich aller Header sowie dem „Inter Frame Gap“ und unter Ausschluss zusätzlicher Laufzeiten der MAC-Schicht.

Der Durchsatz der Paketübertragung liegt bei ca. 2/3 der theoretischen Kurve, was auf die CSMA/CA und ACK Koordination der MAC-Schicht zurückzuführen ist. Der Durchsatz eines RIOT Senders mit deaktivierten MAC-Komponenten nähert sich der theoretischen Kurve an (unter Vernachlässigung von Paketverlusten).

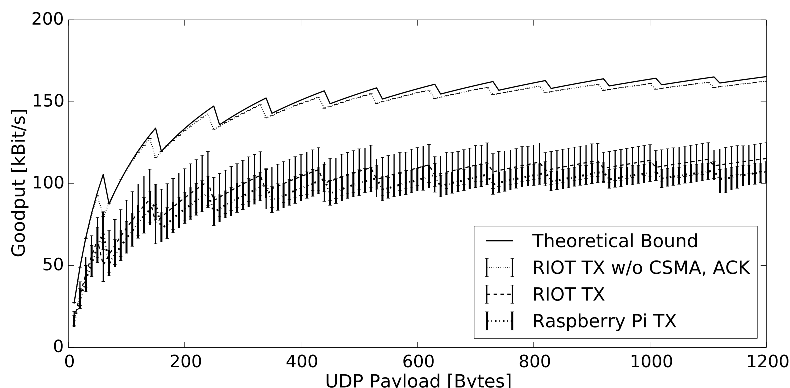

Abb. 3. Effektiver UDP-Durchsatz (Goodput) mit 802.15.4 Radio vs Theorie.

Die Ergebnisse zeigen, dass der GNRC Netzwerk-Stack den Gerätetreiber mit maximaler Geschwindigkeit versorgt (die reine Software-Leistung in Abb. 2 ist ca. 50 mal schneller). Der effektive Datendurchsatz wird durch die 802.15.4 Übertragungsgeschwindigkeit begrenzt. Die Vergleichsmessung zwischen zwei Raspberry Pi Boards zeigt den selben Effekt und hat unerwarteterweise einen geringfügig niedrigeren UDP-Durchsatz. Die Messungen für die Empfangsrichtung (hier nicht dargestellt) gleichen denen des Senders annähernd.

Energiebedarf Der Energiebedarf des Senders und Empfängers wurde getrennt gemessen. Das Senden eines UDP-Pakets mit 1000 Bytes Payload verbraucht im Mittel ca. 10,7 mJ, das Empfangen eines solchen Pakets ca. 10,3 mJ. Im Vergleich mit Energiemessungen des reinen Netzwerk-Stacks, ohne Treiber und Datenübertragung, ist der Aufwand hier ca. 30 mal höher.

4.4 GNRC im Vergleich zu anderen IP-Stacks

In diesem Kapitel wird GNRC (w/ und w/o RPL) mit den IPv6-Stacks lwIP (w/o RPL) und emb6 (w/ und w/o RPL) verglichen. Für die Messungen wurden die Stacks in das Netzwerk-System von RIOT integriert. Die Größen der jeweiligen Paketspeicher wurden so dimensioniert, dass die Netzwerk-Stacks ein volles IPv6-Paket darin verarbeiten können.

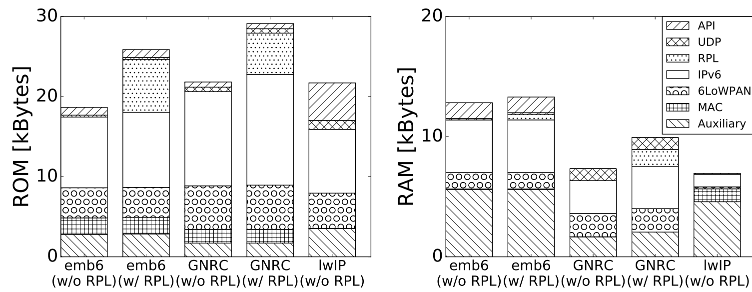

Abb. 4. Speicheranforderungen verschiedener Network-Stacks in RIOT

Speicherbedarf In Abbildung 4 sind die Speicheranforderungen der o.g. Netzwerk-Stacks in RIOT gegenübergestellt. GNRC beaufschlagt den größten Speicher im ROM. Der Mehraufwand gegenüber emb6 und lwIP liegt zwischen 1 kB bis 4 kB. Im Vergleich des RAM-Bedarfs liegt GNRC nur knapp oberhalb von lwIP (ca. 1 kB) und weiter unterhalb von emb6 (ca. 5 kB). Dies verwundert zunächst, da die Thread-basierte Modulbauweise in GNRC einen Mehraufwand mit sich bringt. Der Vorsprung ist auf die zentrale und de-duplizierende Datenhaltung des Paketspeichers *pktbuf* in GNRC zurückzuführen, welcher variable Datenblocklängen erlaubt.

Weitere Messungen von GNRC zur Laufzeit zeigen, dass der allozierte Stack-Speicher aller Threads in Summe, zu ca. 61 % unbenutzt bleibt. Das Optimierungspotenzial liegt in der Standardgröße für den kontextbezogenen Stack-Speicher bei der Thread-Initialisierung.

Rechenzeit In Abbildung 5 ist die relative Laufzeit der externen Stacks in Bezug zu GNRC (w/ oder w/o RPL) dargestellt. Die Paketprozessierung in emb6 ist ca. 20-40 % langsamer als in GNRC, für UDP-Pakete $<$ 200 Bytes. Eine Ursache hierfür ist die intervallgesteuerte Abfrage einer Event Queue in emb6.

Die Leistung des Empfängers in lwIP ist vergleichbar zu GNRC. Die periodisch auftretenden Spitzen in der Grafik sind der 6LoWPAN Fragmentierung

geschuldet, welche nicht für lwIP-optimale Payloadgrößen durchgeführt wurde (bekanntes Problem dieses Netzwerk-Stacks). Die Leistung des Senders in lwIP übertrifft GNRC um bis zu 40 %. In GNRC sind mehrfache IPC-Aufrufe bei der Paketprozessierung durch den Stack nötig, was einen Mehraufwand verursacht. lwIP blockiert während der Fragmentierung und dem Versenden aller Fragmente eines Datagramms sodass alle in einem Durchlauf, ohne Unterbrechung, abgearbeitet werden.

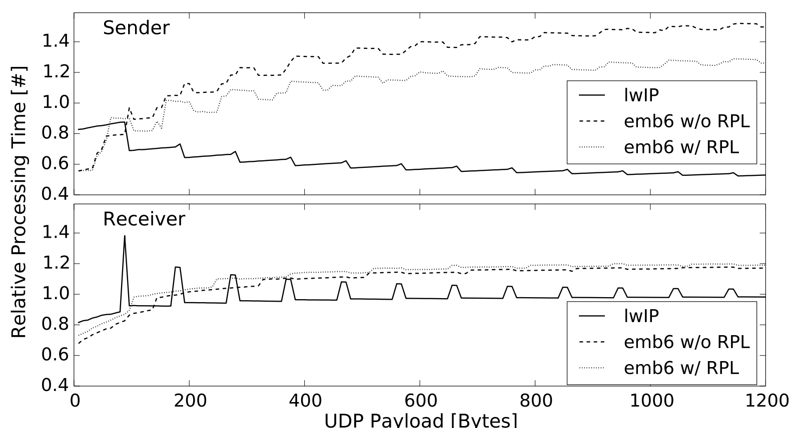

Abb. 5. Rechenzeit verschiedener IP-Stacks relativ zu GRNC.

5 Fazit und Ausblick

In diesem Beitrag haben wir das Open Source IoT-Betriebssystem RIOT vorgestellt. Dabei wurde gezeigt, dass adaptierte Konzepte vollwertiger Betriebssysteme auch auf kleinen IoT-Geräten eine annehmbare Leistung erbringen können.

Wir haben den Betriebssystemkern im Detail vorgestellt und seine Echtzeitfähigkeit erläutert. Darüber hinaus wurde der Standard Netzwerk-Stack GNRC analysiert, welcher sich durch eine stark modularisierte Architektur auszeichnet. Es wurde gezeigt, dass der nötige Mehraufwand für den hohen Grad an Konfigurierbarkeit des Stacks relativ gering ist, verglichen mit monolithischen Netzwerk-Stacks. Des Weiteren haben wir herausgefunden, dass die reine Software-Leistung Größenordnungen über der spezifizierten Datenrate des unterliegenden 802.15.4 Übertragungsmediums liegt. D.h. der effektive Datendurchsatz wird durch den Mehraufwand der Multi-Threading Architektur kaum beeinflusst.

Künftig werden wir weitere Vergleichsmessungen des Energieverbrauchs und der Rechenzeit mit RIOT und Contiki durchführen. Außerdem werden wir an der Weiterentwicklung des GNRC Netzwerk-Stacks hinsichtlich der Implementierung neuer Internet- und IoT- Konzepte wie beispielsweise LPWAN (Low Power

Wide Area Network) [24] oder ICN, arbeiten. Ferner gehört die Pflege und Wartung des gesamten Betriebssystems sowie der Community zu einer der zentralen Aufgaben für die Zukunft von RIOT. Die künftigen Arbeiten werden im Rahmen der Forschungsprojekte *Industrial Internet* und *I3* weitergeführt und von den Zielen dieser Projekte gelenkt.

Literaturverzeichnis

1. L. Atzori et al. . The Internet of Things: A survey. *Computer Networks*, 2010.
2. C. Bormann et al. . Terminology for Constrained-Node Networks. RFC 7228, 2014.
3. mbed OS. `https://mbed.org/technology/os`.
4. A. Dunkels et al. . Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. *Local Computer Networks*, 2004.
5. P. Levis. Experiences from a Decade of TinyOS Development. *OSDI*, 2012.
6. O. Hahm et al. . Operating Systems for Low-End Devices in the Internet of Things: a Survey. *IEEE IoT Journal*, 2015.
7. D. Gay et al. . The NesC Language: A Holistic Approach to Networked Embedded Systems. *ACM SIGPLAN PLDI*, 2003.
8. A. Dunkels et al. . Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. *ACM SenSys*, 2006.
9. Zephyr Project. `https://www.zephyrproject.org`.
10. H. Will et al. . A Real-Time Kernel for Wireless Sensor Networks Employed in Rescue Scenarios. *IEEE LCN*, 2009.
11. N. Kushalnagar et al. . IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals. RFC 4919.
12. Z. Sheng et al. . A Survey on the IETF Protocol Suite for the Internet of Things: Standards, Challenges, and Opportunities. *IEEE Wireless Communications*, 2013.
13. E. Baccelli et al. . RIOT OS: Towards an OS for the Internet of Things. *IEEE INFOCOM*, 2013.
14. E. Baccelli et al. . Information Centric Networking in the IoT: Experiments with NDN in the Wild. *ACM ICN*, 2014.
15. O. Hahm et al. . A Case for Time Slotted Channel Hopping for ICN in the IoT. *Open Archive: Nr. arXiv:1602.08591*, 2016.
16. R. Min et al. .A. Chandrakasan. Energy-centric Enabling Technologies for Wireless Sensor Networks. *IEEE Wireless Communications*, 2002.
17. T. Watteyne et al. . Industrial Wireless IP-Based Cyber–Physical Systems. *IEEE.*
18. K. Moskvitch. Tactile Internet: 5g and the Cloud on Steroids. *Engineering & Technology*, 2015.
19. R. Barry. FreeRTOS, a FREE open source RTOS for small embedded real time systems. `http://www.freertos.org`.
20. ITRON project archive. `http://www.ertl.jp/ITRON/home-e.html`.
21. J.-H. Hoepman and B. Jacobs. Increased security through open source. *Communications of the ACM*, 2007.
22. D. Ritchie. The UNIX System: A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal*, 1984.
23. C. Adjih et al. . FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed. *IEEE WF-IoT*, 2015.
24. A. Minaburo et al. . LPWAN GAP Analysis. IETF Std. draft-minaburo-lpwan-gap-analysis-01 [work-in-progress], 2016.