RIOT OS: O Amigável Sistema Operacional para a Internet das Coisas

Matheus Gonçalves Stigger, UCPEL

Resumo—Com o surgimento da Internet das Coisas (IoT), sistemas operacionais compactos são necessários em dispositivos low-end para facilitar o desenvolvimento e a portabilidade de aplicativos IoT. RIOT é um sistema operacional livre e de código aberto proeminente neste espaço. Neste documento, é mostrada uma visão geral abrangente do RIOT, com os principais componentes de interesse para desenvolvedores e usuários em potencial: o kernel, a abstração de hardware e a modularidade do software. Também são explicados aspectos operacionais como inicialização do sistema, gerenciamento de energia e uso de rede.

Palavras-Chave—Internet das Coisas (IoT), Sistema Operacional, Dispositivos Low-end, Sistema de Tempo Real.

1 Introdução

A Internet está se expandindo com o advento da Internet das Coisas (IoT) - espera-se que bilhões de entidades físicas em nosso planeta sejam instrumentadas e interconectadas por padrões de protocolo aberto. Em particular, a IoT irá aproveitar sensores e atuadores de próxima geração para interoperar com o mundo físico. Esses sistemas ciberfísicos não apenas realizarão a aquisição e o processamento de dados, mas também provavelmente controlarão mais e mais elementos de nosso ambiente.

1.1 Histórico

O RIOT OS é um sistema operacional que foi oficialmente lançado em 2013, porém teve seu início em 2008 com o FeuerWare, um sistema operacional para redes de sensores sem fio.

1.2 Motivações

Os dispositivos IoT não apenas se interconectam, mas também estendem a comunicação para além dos gateways, na Internet de hoje (por exemplo, a nuvem) que não lidou antes com tantos dispositivos.

- ***Matheus Gonçalves Stigger**: Engenharia de Computação, Centro de Politécnico - CPoli. Universidade Católica de Pelotas - UCPEL.*
E-mail: matheus.goncalves@sou.ucpel.edu.br

Dispositivos IoT neste contexto são muito restritos em termos de recursos de hardware. Em particular, dispositivos IoT low-end não têm recursos suficientes para executar sistemas operacionais convencionais como Linux, BSD ou Windows, ou mesmo executar derivados de sistema operacional (SO) otimizado como OpenWRT, uClinux, Brillo ou Windows 10 IoT Core. Com essas limitações de recursos em dispositivos IoT low-end, uma variedade de sistemas operacionais mais compactos foram projetados recentemente. Um dos sistemas operacionais de destaque neste espaço é o RIOT, que foi escrito do zero para dispositivos IoT low-end. RIOT é executado em memória mínima na ordem de 10 kByte e pode ser executado em dispositivos sem MMU (unidade de gerenciamento de memória) nem MPU (unidade de proteção de memória).

1.3 Objetivos

RIOT é um sistema operacional de código aberto, baseado em uma arquitetura modular construída em torno de um kernel minimalista e desenvolvido por uma comunidade mundial de desenvolvedores. Antes de detalharmos os conceitos em ação no RIOT, é importante observar os objetivos que motivaram o design do RIOT em primeiro lugar:

- Minimizar o uso de recursos em termos de RAM, ROM e consumo de energia;

- Suporte para configurações versáteis com MCUs de 8 a 32 bits, ampla variedade de placas e casos de uso;
- Duplicação de código minimizada nas configurações;
- Portabilidade da maior parte do código, em todo o hardware suportado;
- Fornecer uma plataforma de software fácil de programar;
- Fornecer recursos em tempo real;

2 Anatomia de um Sistema Operacional IoT

Um sistema operacional para dispositivos IoT deve incluir uma série de recursos, incluindo (mas não se limitando à) abstração de hardware, rede e gerenciamento de energia, na presença de consumo espartano de recursos de hardware. A seguir, é mostrado com mais detalhes os requisitos para um sistema operacional em execução em dispositivos IoT low-end.

2.1 Dispositivos IoT Low-end vs. High-end

Em comparação com dispositivos IoT de ponta, como smartphones e RaspberryPis, os dispositivos IoT low-end normalmente têm um fator de 10^6 menos memória, 10^3 menos capacidade de unidade central de processamento (CPU), consomem 10^3 menos energia e usam redes com 10^5 a menos de rendimento. Dispositivos IoT low-end são baseados em três componentes principais:

- Um microcontrolador (MCU) - uma única peça de hardware contendo a CPU, alguns kB de memória de acesso aleatório (RAM) e memória somente leitura (ROM), bem como seus periféricos mapeados por registro.
- Diversos dispositivos externos, como sensores, atuadores ou armazenamento, que são conectados ao MCU por meio de uma variedade de padrões de entrada/saída (E/S), como UART, SPI ou I2C.
- Uma ou mais interfaces de rede que conectam o dispositivo à Internet, normalmente usando uma tecnologia de transmissão de baixa potência. Esses transceptores podem fazer parte do MCU (neste caso, é conhecido como system-on-chip (SoC)) ou ser conectados como um dispositivo externo por meio de um barramento de E/S.

2.2 Requisitos de Sistema Operacional para Dispositivos IoT Low-end

Seguindo a partir desse ambiente, um sistema operacional para dispositivos IoT low-end deve encontrar um equilíbrio entre as várias direções de design para atender aos requisitos descritos abaixo.

1) **Requisitos básicos de desempenho**: os principais requisitos de desempenho para tal sistema operacional são (i) eficiência de memória, (ii) eficiência de energia e (iii) reatividade. Para caber no orçamento de RAM e ROM em dispositivos IoT low-end, o sistema operacional deve ocupar uma pequena área de memória. Em geral, espera-se que os dispositivos IoT durem anos com uma única carga de bateria. Um sistema operacional deve, portanto, fornecer mecanismos integrados de economia de energia, explorando, tanto quanto possível, os modos de baixo consumo de energia disponíveis no hardware IoT. Em vários contextos, como detecção de alarmes ou reação a comandos remotos, espera-se que os dispositivos IoT reajam em (quase) tempo real. Um sistema operacional deve, portanto, ser capaz de fornecer pelo menos recursos soft real-time.
2) **Requisitos de interoperabilidade de rede**: espera-se que dispositivos IoT low-end sejam conectados à rede. As tecnologias Linklayer usadas na IoT incluem várias tecnologias sem fio de baixa potência, como IEEE 802.15.4, Bluetooth Low-Energy (BLE) ou LoRa. As camadas de link relevantes na IoT também incluem tecnologias com fio, como BACnet, Power-line Communication (PLC), Ethernet ou Controller Area Network (CAN). Um sistema operacional deve, portanto, oferecer suporte para rádio heterogêneo e transceptores com fio, bem como várias camadas de link. Além disso, muitas vezes se espera uma conectividade perfeita com a Internet dos dispositivos IoT. Portanto, um sistema operacional deve oferecer suporte para a pilha IP

padrão de baixa potência, incluindo 6LoWPAN, IPv6 e UDP. O suporte para protocolos adicionais também pode ser necessário para rotear pacotes, gerenciar dispositivos e interoperar na Web of Things (WoT).

3) **Requisitos de interoperabilidade do sistema**: o software em dispositivos IoT pode ser complexo e geralmente deve cumprir rigorosas restrições de qualidade. Por outro lado, não apenas o hardware IoT low-end disponível atualmente é diversificado, mas também evolui rapidamente com o tempo. Portanto, o software IoT não portátil deve ser mínimo. Em baixo nível, um sistema operacional deve fornecer abstração de hardware para que a maior parte do código seja reutilizável em todo o hardware IoT compatível com o sistema operacional. No nível de aplicativo e biblioteca de software, um sistema operacional deve fornecer interfaces padrão para conectar uma ampla variedade de módulos de software de terceiros.

4) **Requisitos de segurança e privacidade**: as implantações de IoT devem penetrar tanto em vidas privadas quanto em processos industriais, aumentando as demandas por segurança e privacidade de sistemas e aplicativos. Por outro lado, o grande número de dispositivos IoT implantados representa uma grave ameaça à infraestrutura geral da Internet. Um sistema operacional para a IoT deve ser cuidadosamente desenvolvido e continuamente revisado para minimizar sua superfície de ataque. As primitivas criptográficas devem ser selecionadas com sabedoria e adaptadas especificamente para atender às restrições do dispositivo. Além disso, para evitar violações de privacidade indesejadas ou vazamento de dados, a transparência é desejável em relação ao manuseio de dados do usuário final, com os usuários finais permanecendo no controle.

3 Características Técnicas

Os objetivos acima levam a uma série de princípios que explicam partes do design do RIOT. Esses princípios são os seguintes:

1) **Padrões de rede**: RIOT concentra-se em especificações de protocolo de rede padrão aberto, por exemplo, Protocolos IETF.

2) **Padrões do sistema**: RIOT visa cumprir os padrões relevantes, por exemplo, o padrão ANSI C (C99), para aproveitar ao máximo o maior conjunto de software de terceiros. O uso extensivo da linguagem C atende aos requisitos de poucos recursos e fácil programação.

3) **APIs unificadas**: RIOT visa fornecer consistência de APIs em todo o hardware suportado, mesmo para APIs de acesso a hardware, para atender a portabilidade de código e minimizar a duplicação de código.

4) **Modularidade**: RIOT visa definir blocos de construção autocontidos, a serem combinados de todas as maneiras imagináveis, para atender tanto a casos de uso versáteis quanto a restrições de memória.

5) **Memória estática**: RIOT faz uso extensivo de estruturas pré-alocadas para fornecer confiabilidade, validação e verificação simplificadas, bem como requisitos em tempo real.

6) **Fornecedor e Independência de Tecnologia**: bibliotecas de fornecedores são normalmente evitadas, para impedir o aprisionamento do fornecedor e minimizar a duplicação de código. Além disso, as decisões de design não devem vincular a RIOT a uma tecnologia específica.

7) **Comunidade de Código Aberto e Inclusivo**: RIOT tem como objetivo permanecer livre e aberto para todos, e agregar uma comunidade com processos 100% transparentes.

Em relação a sua estrutura, o RIOT é estruturado em módulos de software que são agregados em tempo de compilação, em torno de um kernel que fornece funcionalidade minimalista. Essa abordagem permite construir o sistema completo de maneira modular, incluindo apenas os módulos exigidos pelo caso de uso. Isso não apenas minimiza o consumo de memória, mas também a complexidade do sistema em campo. Ainda assim, a modularidade é balanceada para evitar convolução estrutural não gerenciável no

longo prazo devido a componentes de software excessivamente refinados.

O projeto é dividido em quatro grandes partes. A primeira é a abstração de hardware, para o sistema se fazer compatível com diversas plataformas diferentes. Essa camada faz com que o RIOT consiga dividir bem o código específico para o hardware e o código específico para a aplicação, fazendo com que o mesmo código funcione em diferentes plataformas. A segunda parte é o Kernel, que é composto basicamente por três principais partes: o escalonador preemptivo, o IPC e o mutex para gerenciar a memória compartilhada. A terceira parte é o pilha de rede. O RIOT possui uma camada genérica que busca abstrair e trazer um acesso único a todas as interfaces de redes suportadas pelo RIOT. Por último, tem uma camada que possui bibliotecas de integração do sistema com a aplicação.

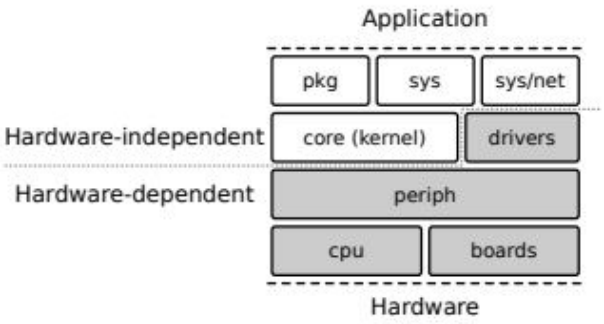

Figura 1. Elementos Estruturais do RIOT

O código RIOT é estruturado de acordo com os grupos representados na Figura 1:

- O core implementa o kernel e suas estruturas de dados básicas, como listas vinculadas, LIFOs e ringbuffers.
- A abstração de hardware distingue quatro partes: (i) cpu que implementa funcionalidades relacionadas ao microcontrolador, (ii) boards que principalmente seleciona, configura e mapeia a CPU e drivers usados, (iii) drivers que implementam drivers de dispositivo, e (iv) periph que fornece acesso unificado a periféricos de microcontroladores e é usado por drivers de dispositivo.
- O sys implementa bibliotecas de sistema além das funcionalidades do kernel, como funcionalidades criptográficas, suporte a sistema de arquivos e rede.
- O pkg importa componentes de terceiros (bibliotecas que não estão incluídas no repositório de código principal).
- O application implementa a lógica de alto nível do caso de uso real.

O RIOT OS também é um sistema operacional microkernel. Implementa um escalonador preemptivo, tickless baseando em prioridades.

Funciona com sistemas de threads, sincronizadas via mutex e utilizando um sistema de comunicação entre processos (IPC) para trocar informações entre as threads, de forma síncrona ou assíncrona.

RIOT OS tem em seus destaques a grande gama de tecnologias de rede e comunicação que implementa, como IPv6, 6LoWPAN, RPL, UDP, CoAP, CBOR, além da pilha tradicional TCP/IP. O RIOT também possui suporte ao OpenThread.

O sistema ocupa em média 2 kB de RAM (Random Access Memory) e em média 5 kB de ROM (Read-only Memory).

O RIOT OS possui suporte a microcontroladores de 8, 16 e 32 bits. Possui porte para placas das arquiteturas AVR, ARM, ARC, x86, ESP8266, ESP32, MIPS32, MSP430, PIC32 e RISC-V.

4 O Kernel do RIOT

O kernel do RIOT inicialmente evoluiu a partir do projeto FireKernel. Ele fornece funcionalidade básica para multi-threading: comutação de contexto, agendamento, comunicação entre processos (IPC) e primitivos de sincronização (mutex etc.). Todos os outros componentes, como drivers de dispositivo, componentes de pilha de rede ou lógica de aplicativo, são mantidos separados do kernel. Normalmente, a interação entre esses componentes é implementada por meio da API básica minimalista fornecida pelo kernel.

4.1 Multi-Threading

Um tópico no RIOT é semelhante a um tópico no Linux. Usando a API principal RIOT, cada componente - seja um driver para um transceptor de rede ou alguma lógica específica de aplicativo

- pode ser executado em um contexto de thread separado com um nível de prioridade de thread atribuído a ele. Multi-threading foi integrado para fornecer as seguintes vantagens: (a) separação lógica limpa entre várias tarefas, (b) priorização simples entre tarefas e (c) importação mais fácil de código. Ignorando a complexidade adicional inerente necessária para gerenciar a simultaneidade, a desvantagem mais proeminente do multi-threading em dispositivos embarcados low-end é a sobrecarga de memória. Este overhead se decompõe em (i) memória para o bloco de controle de thread (TCB), (ii) memória para espaço de pilha e (iii) memória para contexto de CPU (registradores). Enquanto (iii) é determinado pela arquitetura da CPU (por exemplo, 64 bytes no Cortex-M, menos em 16 bits e 8 bits), (i) e (ii) podem ser influenciados pelo software.

a) **Sincronização leve entre processos**: o kernel fornece vários primitivos de sincronização, como mutex, semáforo e mensagens (msg). Esses mecanismos são modelados como submódulos do kernel e, portanto, opcionais e compilados apenas sob demanda. Em termos de consumo de memória, o tamanho desses submódulos é pequeno. Em termos de velocidade, o atraso incorrido pelo uso do IPC se decompõe no tempo para (i) salvar e restaurar contextos de thread, (ii) o tempo de execução do planejador e (iii) o tempo de execução do próprio submódulo IPC. Enquanto (i) é inteiramente determinado pela arquitetura da CPU, (ii) é constante, e um design estreito de msg torna (iii) pequena sobrecarga em comparação com (i) e (ii).

b) **Multi-threading é opcional**: para alguns cenários onde o uso de memória extremamente baixo é obrigatório, um aplicativo single-threaded pode ser desejável. O RIOT não força o uso de vários threads de aplicativo. A menos que um módulo de sistema selecionado precise executar um encadeamento, o aplicativo do usuário pode ser o único encadeamento em execução no sistema. Nesse caso, é possível remover a maioria dos requisitos de memória do escalonador. Dessa forma, é possível construir firmwares extremamente eficientes em termos de memória de forma semelhante ao Arduino, ainda se beneficiando dos recursos do RIOT, incluindo abstração de hardware, drivers de dispositivo e ferramentas.

4.2 Escalonador & Propriedades em Tempo Real

O kernel do RIOT usa um escalonador baseado em prioridades fixas e preempção com operações *O(1)*, permitindo recursos soft real-time. Em mais detalhes: o tempo necessário para interromper e alternar para um thread diferente não excederá um (pequeno) limite superior, uma vez que salvar o contexto, encontrar o próximo thread a ser executado e restaurar o contexto são operações determinísticas. Uma política de agendamento de execução para conclusão baseada em classe é usada: o encadeamento de maior prioridade (ativo) é executado, apenas interrompido por rotinas de serviço de interrupção (ISRs). Com esse agendador, o RIOT fornece uma maneira limpa de priorizar tarefas e antecipar o manuseio de tarefas de baixa prioridade para lidar com eventos de alta prioridade.

A política de agendamento usada no RIOT simplifica o agendamento em tempo real porque, se um evento requer ação por um thread de alta prioridade, os threads de prioridade mais baixa são eliminados e o thread de alta prioridade é executado até que o evento seja tratado. Observe que, para minimizar o tempo de processamento e o consumo de energia, não há mudanças de contexto que imitem a execução paralela de tarefas de mesma prioridade. Essas características permitem um comportamento determinístico do sistema em tempo real - desde que as prioridades das tarefas sejam configuradas de forma coerente. Além disso, essas características também suportam bons casos de uso com criticidade mista, em que tarefas em tempo real são executadas juntamente com tarefas de melhor esforço - por exemplo, uma tarefa de controle do motor (tempo real) em execução na mesma placa com uma pilha de rede IP (não em tempo real).

O agendador usado no RIOT é tickless. Não depende de fatias de tempo da CPU e tiques periódicos do cronômetro do sistema. O sistema, portanto, não precisa acordar periodicamente, a menos que algo esteja realmente acontecendo, por exemplo, uma interrupção acionada pelo hardware conectado. Um despertar pode ser iniciado

por um transceptor quando um pacote chega, por temporizadores quando eles disparam, por botões sendo pressionados ou similar. Se nenhum outro thread estiver em execução e nenhuma interrupção estiver pendente, o sistema muda por padrão para o thread inativo - que tem a prioridade mais baixa. A thread ociosa, por sua vez, muda para o modo de economia de energia mais possível, otimizando assim o consumo de energia.

5 Inicialização do Sistema

O RIOT fornece todos os elementos para bootstrap de hardware IoT desde a primeira instrução de software chamada (normalmente algum tipo de manipulador de interrupção de reinicialização) até o ponto em que a função principal real é chamada em um contexto de thread. Sempre que possível, todo o código de inicialização inicial é implementado em C simples, o que leva a um melhor nível de capacidade de manutenção do que a abordagem típica baseada no código assembly. A sequência de inicialização do sistema consiste tipicamente nas seguintes etapas: (i) inicialização da memória, (ii) inicialização da placa e da CPU, (iii) inicialização da biblioteca C usada (esta etapa é opcional) e, finalmente, (iv) configuração e inicialização do sistema operacional real.

A inicialização da memória simplesmente se encarrega de copiar as variáveis inicializadas na RAM (seção de dados) e definir todas as variáveis não inicializadas para zero (seção bss). Esta etapa pode ser opcionalmente estendida com subtarefas específicas de hardware, por exemplo, aplicação de correções para bugs de hardware, conforme especificado nas folhas de errata do fornecedor.

A inicialização da placa então se encarrega de inicializar os elementos de hardware específicos da placa, como LEDs integrados, expansores de E/S ou dispositivos de gerenciamento de energia externos à CPU. Neste ponto, a inicialização da CPU também é acionada, que se encarrega de configurar o sistema de interrupção, a árvore do relógio, drivers periféricos compartilhados, etc.

Em seguida, a inicialização da biblioteca C cuida do mapeamento de funções libc (por exemplo, funções para alocação de memória dinâmica ou acesso stdio) para as syscalls RIOT correspondentes.

Neste estágio, o sistema básico é inicializado e o controle é transferido para o kernel RIOT. Após configurar os threads *idle* e *main* e alternar para o último, o RIOT inicializa todos os módulos do sistema e drivers de dispositivo configurados por meio de seu módulo auto_init. Por último, a função *main* atual é chamada, que é o ponto de entrada para o aplicativo do usuário real.

6 Gerenciamento de Energia

Para projetar o gerenciamento de energia, é crucial primeiro entender as fontes de consumo de energia e a influência que o software pode ter sobre elas. O consumo total de energia de um dispositivo é a soma da energia consumida por (i) a CPU, (ii) dispositivos conectados por meio de periféricos e (iii) vários outros componentes passivos externos à CPU. Embora (iii) seja fortemente influenciado pelo design de hardware de uma placa, ambos (i) e (ii) são influenciados pelo software.

Por um lado, o sistema de gerenciamento de energia no RIOT define automaticamente a CPU no modo de hibernação mais profundo possível quando ociosa, o que diminui o consumo de energia devido a (i). Por outro lado, espera-se que os aplicativos do usuário ou outros módulos do sistema lidem com (ii) gerenciando o estado dos periféricos. Por exemplo, um transceptor de rede é gerenciado por um módulo de protocolo de camada MAC, enquanto um sensor é gerenciado por um aplicativo de detecção.

O elemento central de design no sistema de gerenciamento de energia da CPU no RIOT é o chamado thread inativo. Este encadeamento é criado durante a inicialização do sistema e é agendado quando nenhum outro encadeamento precisa ser executado, ou seja, quando a CPU está ociosa. O gerenciamento de energia da CPU é construído em torno de uma única função, que aciona a CPU para entrar no estado de energia mais baixo possível (modo de espera). Chamar essa função é a única tarefa que o thread inativo executa. Este mecanismo funciona de forma transparente para aplicativos de usuário e outros módulos do sistema.

7 Cenários de Uso

O RIOT tem sido utilizado por diversas empresas como, Continental[1], Locha[2], Savoir-faire Linux[3], SSV Software Systems GmbH[4], wolfSSL[5] e também em universidades como a Sapienza University of Rome[6].

7.1 Compartilhamento de Carros sem Chave

A Continental lançou um produto baseado no RIOT, um módulo para acesso de veículos e telemática, para soluções de car sharing.

7.2 Converse e Envie Bitcoin sem Internet

A Locha Mesh desenvolve uma tecnologia de transmissão de dados alternativa e resiliente para comunicações móveis e pagamentos sem a necessidade de uma conexão com a Internet, usando a topologia de rede mesh AODVv2 para permitir conexões P2P diretas entre nós. O Locha Mesh tem suporte total a IPv6, portanto, a maioria dos aplicativos atuais pode ser executado nele.

7.3 Ensino da Internet das Coisas

Na Sapienza University of Rome, os alunos de Engenharia da Computação são treinados com base no RIOT para adquirir habilidades práticas e conhecimento sobre a IoT. A ampla gama de ferramentas e recursos de experimentação para depuração de camada cruzada e criação de perfil é valiosa para eles desenvolverem, testarem e avaliarem aplicativos de IoT em profissões futuras.

8 Considerações Finais

Este documento apresentou uma visão geral abrangente do RIOT, um sistema operacional de código aberto para dispositivos embarcados low-end.

Ao contrário de outros sistemas operacionais proeminentes no domínio, o RIOT adota uma abordagem propositalmente semelhante à filosofia GNU do Linux em termos de licença de código, independência do fornecedor e transparência. De uma perspectiva técnica, entretanto, RIOT é escrito do zero (sem usar bibliotecas de fornecedores).

O RIOT é baseado nos seguintes princípios de design: eficiência de energia, recursos em tempo real, baixo consumo de memória, modularidade e acesso uniforme à API, independente do hardware subjacente (essa API oferece conformidade parcial com POSIX).

1. https://www.continental.com/en
2. https://locha.io/
3. https://savoirfairelinux.com/en
4. https://www.ssv-embedded.de/en/
5. https://www.wolfssl.com/
6. https://www.uniroma1.it/en/

AGRADECIMENTOS

Este trabalho foi realizado com o objetivo de apresentar o RIOT OS para o Congresso de Sistemas Operacionais do Curso de Engenharia de Computação da UCPEL. Teve como proposta, realizar as buscas em materiais como, sites [1] [2], TCCs [3], dissertações, teses e/ou artigos em geral. Agradecimentos a [3] e [4], pois o texto deste artigo, foi totalmente construído com base em trechos de seus tremendos trabalhos.

Referências

[1] Riot os. https://www.riot-os.org/.

[2] Sempreupdate. riot 2018.10.1 – o so para iot. https://sempreupdate.com.br/e-lancado-o-riot-2018-10-1-o-so-para-iot/.

[3] Vítor barros aquino. análise comparativa de sistemas operacionais para nordic nrf52840 para implementação na meta-plataforma de internet das coisas knot, 2019. https://www.cin.ufpe.br/ tg/2019-2/TG_EC/TG_vba2.pdf.

[4] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S. Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C. Schmidt, and Matthias Wählisch. Riot: An open source operating system for low-end embedded devices in the iot. *IEEE Internet of Things Journal*, 5(6):4428–4440, 2018.

Matheus Gonçalves Stigger Graduando em Engenharia de Computação na Universidade Católica de Pelotas. Tem interesse nas áreas de Controle e Automação, Internet das Coisas, Microcontroladores, Sistemas Embarcados e Linguagem de Programação Python.