

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

**DESENVOLVIMENTO DE UMA APLICAÇÃO IOT UTILIZANDO COAP
E DTLS PARA TELEMETRIA VEICULAR**

LUKAS DERNER GRÜDTNER

FLORIANÓPOLIS

2020/1

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

**DESENVOLVIMENTO DE UMA APLICAÇÃO IOT UTILIZANDO COAP
E DTLS PARA TELEMETRIA VEICULAR**

LUKAS DERNER GRÜDTNER

FLORIANÓPOLIS

2020/1

DESENVOLVIMENTO DE UMA APLICAÇÃO IOT UTILIZANDO COAP E DTLS PARA TELEMETRIA VEICULAR

Trabalho de Conclusão de Curso de Graduação em Ciências da Computação, do Departamento de Informática e Estatística, do Centro Tecnológico da Universidade Federal de Santa Catarina, requisito parcial à obtenção do título de Bacharel em Ciências da Computação.

Orientadora: Carla Merkle Westphall, Profa. Dra.

Coorientador: Leandro Loffi, Me.

FLORIANÓPOLIS

2020/1

DESENVOLVIMENTO DE UMA APLICAÇÃO IOT UTILIZANDO COAP E DTLS PARA TELEMETRIA VEICULAR

LUKAS DERNER GRÜDTNER

Trabalho de Conclusão de Curso de Graduação em Ciências da Computação, do Departamento de Informática e Estatística, do Centro Tecnológico da Universidade Federal de Santa Catarina, requisito parcial à obtenção do título de Bacharel em Ciências da Computação.

Aprovado em:

Por:

Profa. Dra. Carla Merkle Westphall
Orientadora

Me. Leandro Loffi
Coorientador

Prof. Dr. Carlos Becker Westphall
Membro da banca

Me. Caciano dos Santos Machado
Membro da banca

AGRADECIMENTOS

Agradeço primeiramente aos meus pais, que sempre estiveram por perto para me apoiar e incentivar.

Aos meus amigos, pelo companheirismo ao longo destes anos, pelas risadas e bons momentos que passamos juntos e por muitas oportunidades que ainda virão.

Aos meus professores, pelos momentos de aprendizado e lições ensinadas.

À minha orientadora, Carla Merkle Westphall, e ao meu coorientador, Leandro Loffi, pela paciência, incentivo e apoio no desenvolvimento deste trabalho.

“The only true wisdom is in knowing you know nothing.”

— Socrates

RESUMO

A utilização da tecnologia de IoT nos mais diversos ambientes tem crescido muito nos últimos anos. A grande quantidade de dispositivos em rede tem trazido problemas devido a escalabilidade de sistemas e tráfego de dados gerado. No entanto, outro problema que requer atenção é a implantação de mecanismos de segurança para garantir a privacidade dos usuários destes sistemas. Devido ao seus recursos restritivos, os dispositivos IoT necessitam de mecanismos especiais para a proteção dos dados. Este trabalho teve como objetivo realizar um estudo sobre os dois principais protocolos propostos para tratar o problema de comunicação em ambientes IoT, CoAP e MQTT. Para a validação do modelo, foi implementada uma aplicação IoT no contexto de telemetria veicular, fazendo uso do protocolo CoAP com DTLS para fazer a comunicação entre os dispositivos e garantir a segurança na troca de mensagens. Esta aplicação simulou a coleta de dados de sensores de um veículo, transmitindo estes dados para um outro dispositivo IoT para processamento via rede sem fio, compreendendo uma abordagem M2M. Os dados são coletados e processados em tempo real, então é importante que os protocolos trabalhem de maneira rápida e eficiente, para garantir maior precisão da análise. Por fim, experimentos foram realizados utilizando a aplicação desenvolvida, mensurando os tempos de processamento e o consumo de energia dos protocolos CoAP e DTLS. O presente trabalho apresentou resultados satisfatórios em relação à mensuração dos tempos de processamento e consumo energético, apresentando comparações quantitativas entre as principais *cipher suites* utilizadas no protocolo DTLS e o seu impacto na comunicação.

Palavras-chave: *Internet of Things*; segurança; CoAP; DTLS; telemetria veicular

ABSTRACT

The use of IoT technology in the most diverse environments has grown a lot in recent years. The large number of networked devices has brought problems due to the scalability of systems and the data traffic generated. However, another problem that requires attention is the implementation of security mechanisms to guarantee the privacy of the users of these systems. Due to their constrained characteristics, IoT devices require special mechanisms for data protection. This work aimed to carry out a study on the two main protocols proposed to address the communication problem in IoT environments, CoAP and MQTT. For the validation of the model, an IoT application was implemented in the context of vehicular telemetry, using the CoAP protocol with DTLS to communicate between devices and ensure security in the exchange of messages. This application simulated the collection of sensor data from a vehicle, transmitting this data to another IoT device for processing via wireless network, comprising an M2M approach. The data is collected and processed in real time, so it is important that the protocols work quickly and efficiently, to ensure greater accuracy of the analysis. Finally, experiments were performed using the developed application, measuring the processing times and energy consumption of the CoAP and DTLS protocols. The present work presented satisfactory results in relation to the measurement of processing times and energy consumption, presenting quantitative comparisons between the main cipher suites used in the DTLS protocol and their impact on communication.

Key-words: Internet of Things; security; CoAP; DTLS; vehicular telemetry

LISTA DE ILUSTRAÇÕES

1	Topologia genérica de um sistema IoT	20
2	Modos de se atingir uma relação de confiança	22
3	Mensagens de <i>handshake</i> do DTLS	24
4	Arquitetura CoAP	26
5	Arquitetura de pacote de uma mensagem CoAP	27
6	Arquitetura MQTT	29
7	Tempos de atraso de mensagens CoAP	33
8	Tempo necessário para estabelecer uma conexão segura entre nós <i>fog</i> .	34
9	Ilustração de um câmbio CVT	38
10	Organização das aplicações do modelo	38
11	Dispositivo ESP32 WROOM-32 DevKit V1	39
12	Representação do protótipo incluindo as aplicações de coleta de dados e cliente	40
13	Detalhes da pinagem do ESP32 WROOM-32 DevKit V1	40
14	Fluxo de execução da aplicação cliente	41
15	Sensor de temperatura e pressão barométrica BMP280	42
16	Raspberry Pi 3 Model B	44
17	Configuração do endereço do servidor CoAP	59
18	Configuração de <i>cipher suites</i>	59
19	Pacote contendo a <i>cipher suite</i> utilizada na comunicação (Wireshark) .	61
20	Tempo médio de processamento de diferentes <i>cipher suites</i> no cliente .	67
21	Tempo médio de processamento de diferentes <i>cipher suites</i> no lado do servidor	67
22	Tempo médio de processamento de diferentes <i>cipher suites</i> no servidor	68
23	Consumo médio por execução (mA) de diferentes <i>cipher suites</i>	69
24	Menu de configuração do projeto exemplo <code>coap_client</code>	81
25	Submenu para configurações de rede	82
26	Submenu para configurações do protocolo CoAP	82
27	Submenu para configurações do modo de segurança do CoAP	83
28	Submenu para configurações do módulo UART	84

LISTA DE TABELAS

1	<i>Cipher suites</i> analisadas no trabalho correlato 1	31
2	<i>Cipher suites</i> analisadas pelo trabalho correlato 4	34
3	Contribuições deste e de outros trabalhos correlatos	36
4	Especificações dos dispositivos utilizados nos experimentos no lado do servidor	62
5	<i>Cipher suites</i> analisadas	62
6	RSA (<i>notebook</i>): Tempos de execução (ms)	65
7	DHE (<i>notebook</i>): Tempos de execução (ms)	65
8	ECDHE (<i>notebook</i>): Tempos de execução (ms)	65
9	PSK (<i>notebook</i>): Tempos de execução (ms)	65
10	RSA (Raspberry Pi 3): Tempos de execução (ms)	66
11	DHE (Raspberry Pi 3): Tempos de execução (ms)	66
12	ECDHE (Raspberry Pi 3): Tempos de execução (ms)	66
13	PSK (Raspberry Pi 3): Tempos de execução (ms)	66
14	Consumo energético de diferentes <i>cipher suites</i>	69

LISTA DE ABREVIATURAS E SIGLAS

6LowPAN	<i>IPv6 over Low power Wireless Personal Area</i>
AES	<i>Advanced Encryption Standard</i>
AMQP	<i>Advanced Message Queueing Protocol</i>
BLE	<i>Bluetooth Low Energy</i>
CA	<i>Certificate Authority</i>
CoAP	<i>Constrained Application Protocol</i>
CVT	<i>Continuously Variable Transmission</i>
DDoS	<i>Distributed Denial of Service</i>
DHE	<i>Diffie-Hellman Ephemeral</i>
DNS	<i>Domain Name System</i>
DoS	<i>Denial of Service</i>
DTLS	<i>Datagram Transport Layer Security</i>
EAP	<i>Extensible Authentication Protocol</i>
ECDHE	<i>Elliptic Curve Diffie-Hellman Ephemeral</i>
GCM	<i>Galois/Counter Mode</i>
GPS	<i>Global Positioning System</i>
HTTP	<i>Hyper Text Transfer Protocol</i>
I2C	<i>Inter-Integrated Circuit</i>
IEEE	<i>Institute of Electrical and Electronics Engineers</i>
IoT	<i>Internet of Things</i>
IP	<i>Internet Protocol</i>
LoRaWAN	<i>Long Range Wide Area Network</i>
M2M	<i>Machine to Machine</i>

MQTT	<i>Message Queueing Telemetry Transport</i>
MQTT-SN	<i>Message Queueing Telemetry Transport for Sensor Network</i>
NB-IoT	<i>Narrowband Internet of Things</i>
NVS	<i>Non-Volatile Storage</i>
PFC	<i>Perfect Forward Secrecy</i>
PKI	<i>Public Key Infrastructure</i>
PSK	<i>Pre-Shared Key</i>
QoS	<i>Quality of Service</i>
RAN	<i>Radio Access Network</i>
REST	<i>Representational State Transfer</i>
RFC	<i>Request for Comments</i>
RSA	<i>Rivest-Shamir-Adleman</i>
SAE	<i>Society of Automotive Engineers</i>
SHA	<i>Secure Hash Algorithm</i>
SPI	<i>Serial Peripheral Interface</i>
SRAM	<i>Static Random-Access Memory</i>
SSL	<i>Secure Sockets Layer</i>
TCP	<i>Transmission Control Protocol</i>
UART	<i>Universal Asynchronous Receiver Transmitter</i>
UDP	<i>User Datagram Protocol</i>
ULP	<i>Ultra Low Power</i>
WEP	<i>Wired Equivalent Privacy</i>
WPA	<i>Wi-Fi Protected Access</i>
WSN	<i>Wireless Sensor Network</i>

SUMÁRIO

1	INTRODUÇÃO	14
1.1	Motivação	15
1.2	Problematização	15
1.2.1	Solução proposta	16
1.3	Justificativa	16
1.4	Objetivos	17
1.4.1	Objetivos gerais	17
1.4.2	Objetivos específicos	17
1.5	Método de pesquisa e trabalho	17
1.6	Organização do trabalho	18
2	FUNDAMENTAÇÃO TEÓRICA	19
2.1	Internet of Things	19
2.2	Autenticação	21
2.3	DTLS	22
2.3.1	Handshake	23
2.4	CoAP	25
2.5	MQTT	28
3	ESTADO DA ARTE	31
3.1	Primeiro trabalho correlato	31
3.2	Segundo trabalho correlato	32
3.3	Terceiro trabalho correlato	32
3.4	Quarto trabalho correlato	33
3.5	Comparação com este trabalho	35
4	DESENVOLVIMENTO	37
4.1	Contexto	37
4.2	Modelo	38
4.2.1	Aplicação: Coleta de dados	39
4.2.2	Aplicação: Cliente	40
4.2.3	Aplicação: Servidor	41
4.3	Equipamentos utilizados	42

4.3.1	Sensor BMP280	42
4.3.2	Dispositivos ESP32	43
4.3.3	Raspberry Pi 3 Model B	43
4.4	Detalhes de implementação	45
4.4.1	Bibliotecas utilizadas	45
4.4.1.1	Adafruit BMP280	45
4.4.1.2	ESP-IDF	45
4.4.1.3	libcoap	46
4.4.2	Geração de certificados	46
4.4.3	Implementação da aplicação de coleta de dados	47
4.4.3.1	main.cpp	47
4.4.4	Implementação da aplicação cliente	48
4.4.4.1	payload.h	49
4.4.4.2	coap.c	49
4.4.4.3	uart.c	52
4.4.4.4	main.c	54
4.4.4.5	Kconfig.projbuild	55
4.4.4.6	Certificados	55
4.4.5	Implementação da aplicação do servidor	56
4.4.5.1	coap-server.c	56
4.5	Testes experimentais da aplicação	58
4.6	Testes experimentais e mensurações quantitativas	62
4.6.1	Descrição das <i>cipher suites</i>	62
4.6.2	Resultados referentes ao tempo de processamento	64
4.6.3	Resultados referentes ao consumo energético	68
	5 CONCLUSÃO	71
	REFERÊNCIAS	72
	Apêndices	77
	A ESP-IDF	78
A.1	Instalação	78
A.2	Menu de configuração	81
A.2.1	Configuração do módulo Wi-Fi	81

A.2.2	Configuração do módulo CoAP	82
A.2.3	Configuração do módulo UART	83
	B libcoap	85
B.1	Instalação	85
B.2	Utilização	85
	C Aplicação 1: uart-sender	87
C.1	Estrutura do projeto	87
C.2	main.cpp	87
	D Aplicação 2: coap-client	89
D.1	Estrutura do projeto	89
D.2	Certificados	89
D.2.1	coap_ca.pem	89
D.3	coap_client.crt	90
D.4	coap_client.key	90
D.5	payload.h	91
D.6	coap.c	92
D.7	uart.c	103
D.8	main.c	105
D.9	Kconfig.projbuild	105
	E Aplicação 3: coap-server	108
E.1	Estrutura do projeto	108
E.2	Certificados	108
E.2.1	certfile.pem	108
E.2.2	keyfile.pem	109
E.3	coap-server.c	109
	F Artigo	112

1 INTRODUÇÃO

Internet of Things (IoT) é um paradigma relativamente recente que tem tido um enorme crescimento nos últimos anos. Ambientes que integram redes IoT têm por objetivo a coleta de dados através de pequenos sensores acoplados em objetos físicos. Estes sensores são geridos por um pequeno microcontrolador, que tem sob sua supervisão um conjunto de sensores e atuadores. Ao receber os dados, este microcontrolador os envia pela rede, com destino à bancos de dados remotos, a fim de armazená-los. Estes microcontroladores trabalham em baixas frequências e geralmente possuem apenas alguns KiB de memória, o que restringe seu uso à aplicações que não demandam alto processamento. Além disso, os microcontroladores também são capazes de processar comandos através de extensões chamadas de atuadores.

Como exemplo, cita-se a utilização de dispositivos IoT na área médica, onde pequenos dispositivos vestíveis vêm sendo utilizados para o monitoramento de pacientes, com o objetivo de detectar precocemente o surgimento de determinadas doenças (Hassanalieragh *et al.*, 2015). Ambientes industriais têm apresentado um forte aumento na produção devido ao emprego de dispositivos IoT para o controle e automação de processos em suas fábricas. Porém, IoT não se restringe apenas à estes ambientes. Fazendas experimentais estão começando a utilizar biossensores para o monitoramento da saúde de animais, o que fornece dados sobre o estado do animal em tempo real. O monitoramento constante e automatizado possibilita o diagnóstico antecipado de doenças nestes animais (Neethirajan, 2017).

Ao realizar a coleta e a gerência destes dados, torna-se fundamental a utilização de mecanismos para garantir a sua segurança. É essencial manter o sigilo das informações que trafegam de um ponto a outro na rede. A comunicação entre estes dispositivos até seu uso final deve ser realizada através de protocolos leves, que promovam a autenticação mútua de ambas as partes comunicantes, a confidencialidade e a integridade da mensagem (Liu; Xiao; Chen, 2012).

1.1 Motivação

Em 21 de outubro de 2016, a empresa provedora de DNS (*Domain Name System*), Dyn, sofreu um ataque DDoS (*Distributed Denial of Service*) levando estruturas fundamentais ao colapso e comprometendo parte da Internet. O que chamou atenção neste caso foi o fato de o ataque DDoS ter sido efetuado utilizando pequenos dispositivos, como impressoras, *webcams*, *gateways* residenciais e câmeras de monitoramento, ou seja, basicamente dispositivos IoT (Kim; Lee, 2017).

De acordo com uma estimativa da *IHS Markit* – líder mundial em informações críticas, análises e especialização para as principais indústrias e mercados que impulsionam economias em todo o mundo – a quantidade de dispositivos IoT está crescendo a uma taxa de aproximadamente 12% ao ano, e espera-se que até 2030 125 bilhões de dispositivos estejam conectados à rede. Em 2017, contava-se com aproximadamente 27 bilhões de dispositivos (IHS Markit, 2017).

1.2 Problematização

Na indústria automotiva, a telemetria veicular tem sido extensamente utilizada para realizar o monitoramento de diversos aspectos de um veículo, com a finalidade de efetuar análises em busca de pontos de desperdício, possíveis otimizações e também aumentar a segurança para seus usuários. Estas informações geralmente dizem respeito ao comportamento do veículo em estado de funcionamento, tais como velocidade, aceleração, rotação do motor, pressão dos pneus, geoposicionamento, consumo de combustível, entre diversas outras informações que são possíveis de se coletar através de sensores (Harrington *et al.*, 2004).

Estas informações são coletadas tanto em tempo real, de modo que o veículo é submetido à uma bateria de testes e os dados são analisados ao passo em que vão sendo coletados, como também em séries históricas, em que grandes conjuntos de dados são coletados ao longo de um determinado período de tempo, e com isso é possível inferir o comportamento médio do veículo durante o período da amostra. Estas análises têm como objetivo avaliar o comportamento do veículo sob diferentes aspectos, como, por exemplo: situações de estresse, batidas de alta velocidade ou situações normais de uso. Estudos recentes propuseram o uso da telemetria veicular para analisar o comportamento de motoristas enquanto dirigem, com o intuito de evitar acidentes (Silva *et al.*, 2019).

Em análises de tempo real, é muito importante que o sistema de telemetria consiga responder as requisições/transmissões dentro de prazos bem definidos, aumentando a precisão da análise. Para tanto, é imprescindível que o sistema de telemetria realize uma comunicação rápida, eficiente e segura, evitando também inconveniências como invasões e ataques de terceiros.

1.2.1 Solução proposta

Dadas as restrições temporais e de processamento intrínsecas de sistemas de telemetria de tempo real, o presente trabalho propôs o desenvolvimento e implementação de um sistema de telemetria simples compreendendo um ambiente IoT, utilizando-se sensores de temperatura e pressão barométrica para a coleta de informações, e dispositivos *ESP32* para atuarem como clientes, levando os dados dos sensores até o servidor.

Considerando as restrições de comunicação, que por sua vez devem ser rápidas, eficientes e seguras, a solução fará uso dos protocolos CoAP (*Constrained Application Protocol*) e DTLS (*Datagram Transport Layer Security*) para reduzir a sobrecarga da transmissão e também garantir a segurança entre os dispositivos, que é o ideal para ambientes restritos.

1.3 Justificativa

Estamos cercados de dispositivos vulneráveis que podem comprometer nosso bem estar. Em um estudo conduzido pelo *Kaspersky Lab*, mais de 7.000 amostras de *malwares* foram descobertas só nos primeiros 5 meses de 2017, representando 74% a mais do que todas as amostras detectadas entre os anos de 2013 à 2016 (Kaspersky Lab, 2017).

Tanto sensores quanto os servidores são potenciais alvos de ataques em sistemas IoT, o que significa que ambos devem ser autenticados antes de realizar qualquer troca de dados. Conseqüentemente, protocolos de autenticação mútua devem ser leves devido aos recursos limitados de dispositivos IoT (Li; Liu; Nepal, 2017).

Um dos principais desafios para engenheiros de sistemas é escolher o protocolo apropriado que se enquadre nas restrições de sistemas IoT específicos, ao mesmo tempo em que aproveita os avanços nas áreas de *fog* e *cloud computing* (Dizdarević *et al.*, 2019).

1.4 Objetivos

Nesta seção foram definidos os objetivos deste trabalho. Como objetivos gerais, descreveu-se de maneira ampla o escopo de trabalho, e logo após, definiu-se objetivos específicos alcançados por este trabalho.

1.4.1 Objetivos gerais

Este trabalho teve como objetivo geral o desenvolvimento de uma aplicação IoT, fazendo uso dos protocolos CoAP e DTLS para estabelecer um canal de comunicação seguro entre cliente e servidor. Um microcontrolador ESP32 foi adotado como cliente, o qual faz a coleta de dados do ambiente por meio de sensores e os envia até um servidor utilizando ambos os protocolos para garantir confidencialidade e baixa sobrecarga na transmissão. Por intermédio desta aplicação, experimentos foram realizados com a finalidade de mensurar e comparar algumas das principais *cipher suites* utilizadas para prover segurança à comunicação.

1.4.2 Objetivos específicos

Como objetivos específicos, pode-se listar:

1. Pesquisar sobre os principais protocolos de comunicação projetados para ambientes IoT.
2. Pesquisar sobre aplicação de métodos de segurança para a comunicação em ambientes IoT.
3. Desenvolver um protótipo de aplicação que faça uso de um dos protocolos de comunicação segura, simulando um ambiente real.
4. Conduzir testes e avaliações acerca do desempenho do protocolo.

1.5 Método de pesquisa e trabalho

A seguir são apresentados os métodos utilizados para realizar as pesquisas:

1. Estudar sobre os principais trabalhos sobre protocolos de comunicação em ambientes IoT propostos no estado da arte.
 - Pesquisar sobre protocolos eficientes de transmissão de dados para pequenos microcontroladores.
 - Pesquisar sobre algoritmos de criptografia otimizados para uso em IoT.
2. Escolher um protocolo de comunicação para ser utilizado nos experimentos do trabalho.
3. Desenvolver um protótipo de telemetria veicular, fazendo uso do protocolo escolhido.
4. Validar a implementação em uma simulação de um ambiente real.
5. Avaliar o desempenho do protocolo através de experimentos.

1.6 Organização do trabalho

Este trabalho está dividido em 5 capítulos, incluído o capítulo introdutório. O Capítulo 2 apresenta a fundamentação teórica deste trabalho, com uma abordagem mais aprofundada sobre o tema. O Capítulo 3 detalha alguns trabalhos correlatos no estado da arte. Na sequência, o Capítulo 4 apresenta detalhes de implementação da proposta, incluindo resultados obtidos através de experimentos. Por fim, as conclusões deste trabalho são apresentadas no Capítulo 5.

2 FUNDAMENTAÇÃO TEÓRICA

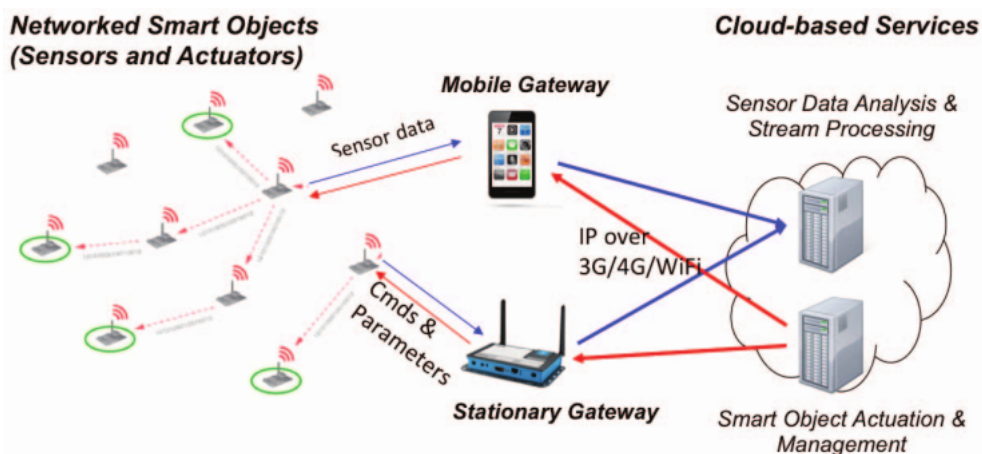
2.1 Internet of Things

Internet of Things é um conceito que abrange toda e qualquer tipo de rede capaz de conectar objetos físicos entre si e que, além disso, permite obter e manipular dados gerados por estes objetos. IoT utiliza a Internet como sistema de comunicação para estabelecer uma interação inteligente entre pessoas e objetos do ambiente (Ray, 2016). São características de dispositivos IoT uma reduzida capacidade de processamento e memória, devido principalmente ao seu tamanho, e a capacidade de acesso à redes sem fio. Ambientes IoT habilitam uma representação virtual de objetos físicos, permitindo-lhes trocar informações contextuais entre si para coordenar ações, obtendo respostas mais rápidas e precisas em relação às mudanças de seus ambientes e, deste modo, tornando-se aptos a utilizar seus recursos de forma mais eficiente (Lara *et al.*, 2020). Estes dispositivos, associados a sensores e/ou atuadores, são capazes de coletar dados do ambiente e enviá-los para uma máquina remota, sendo então armazenados e processados. Definimos sensores como aqueles dispositivos capazes de realizar a coleta de dados, e atuadores como dispositivos que recebem uma entrada e, com base nela, geram uma determinada ação.

Dispositivos IoT são muito mais limitados em termos de recursos do que outras máquinas conectadas à Internet, especialmente em termos de memória. Estes dispositivos normalmente dispõem de alguns KiB de memória (Sabri; Kriaa; Azzouz, 2017). A principal característica de redes IoT é que seus dispositivos são interconectáveis através de uma rede comum, seja ela, a Internet ou uma intra-rede, permitindo que troquem informações entre si, otimizando o sistema como um todo. Para isso, é necessário que estes dispositivos sejam equipados com interfaces de rede. Nestes casos, o mais comum são interfaces de rede sem fio, como IEEE 802.15.4 (*Institute of Electrical and Electronics Engineers*) e BLE (*Bluetooth Low Energy*), embora existam outros meios de rádio-frequência para transmissão de dados. Alguns dispositivos também dispõem de interfaces de rede cabeadas, como Ethernet.

Na Figura 1 podemos observar a topologia de um sistema IoT, composto de dispositivos de borda (sensores e atuadores), os quais se comunicam com *gateways*, dispositivos fog localizados na camada intermediária. Por fim, os *gateways* se comunicam com os dispositivos *cloud*, onde seu objetivo será armazenar e processar os dados recebidos.

Figura 1 – Topologia genérica de um sistema IoT



Fonte: Endler, Silva & Cruz (2017)

A redução do consumo energético de dispositivos IoT é imprescindível para aumentar a vida útil dos dispositivos, pois estes geralmente se localizam em ambientes de difícil acesso, e portanto a sua reposição torna-se complexa. Além disso, dispositivos que possuem sua autonomia dependente de baterias podem se beneficiar ao otimizar a utilização de seus recursos. Em sistemas de baixa potência, a redução do consumo é traduzida em manter o *hardware* de radio-frequência desligado a maior quantidade de tempo possível utilizando protocolos de baixa potência como 6LoWPAN (*IPv6 over Low power Wireless Personal Area Networks*), LoRaWAN (*Long Range Wide Area Network*), entre outros (Sabri; Kriaa; Azzouz, 2017).

Dispositivos IoT são conectados à Internet seja por seus próprios recursos ou através de outros dispositivos, conhecidos como *gateways*. No entanto, a conexão com a Internet os torna vulneráveis à ataques de adversários de qualquer lugar, o que pode acarretar no roubo de informações sensíveis tais como localização, dados médicos ou financeiros. Além disso, ataques podem utilizar atuadores para causar danos ao sistema e até mesmo ameaçar a segurança de seus usuários (Lara *et al.*, 2020).

2.2 Autenticação

Ao utilizar um canal de comunicação inseguro para troca de mensagens, tal como a Internet, ou qualquer outra rede, estamos sujeitos à interceptação de terceiros. Para garantir que o destinatário da nossa mensagem é realmente quem pensamos, somos obrigados a conhecer sua identidade. Intrinsecamente, a autenticação é baseada em confiança (Kim; Lee, 2017). Uma das formas de garantir a identidade de uma pessoa é através da confiança em uma entidade que dê a garantia da autenticidade desta identidade, tal como um órgão governamental.

Em termos computacionais, para estabelecermos uma relação de confiança, é necessário usar uma entidade conhecida como *root of trust*, que torna-se a base da relação de confiança. Esta entidade pode ser um componente de *hardware* especial, ou um PKI (*Public Key Infrastructure*). PKI é um sistema de recursos, políticas, e serviços que suportam a utilização de criptografia de chave pública para autenticar as partes envolvidas na transação. Para dispositivos conectados em rede, existem duas maneiras gerais de estabelecer esta relação (Kim; Lee, 2017), sendo elas:

- **Utilizando uma autoridade de confiança central**

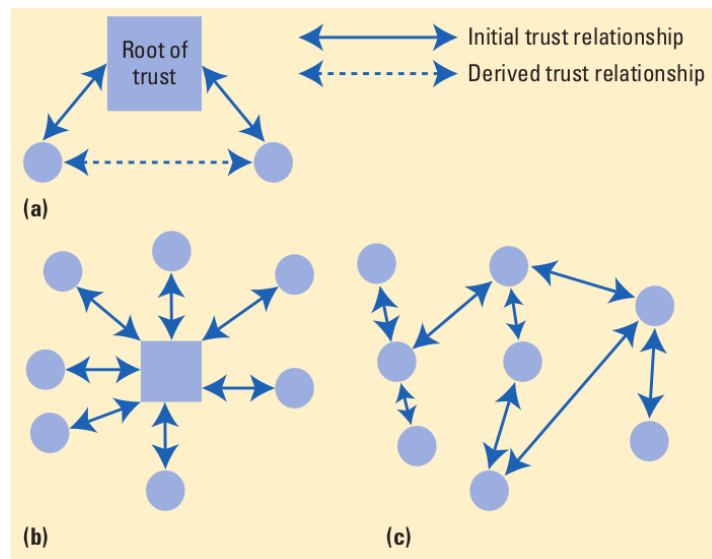
Neste modelo, a autoridade central não participa ativamente da comunicação, e portanto é chamada de terceira parte de confiança. Esta autoridade central dispõe da confiança das outras partes e emite certificados para as outras entidades. O certificado garante a autenticidade da chave pública armazenada, uma vez que é assinado digitalmente por uma autoridade certificadora. Este modelo é largamente utilizado no protocolo SSL/TLS (*Secure Sockets Layer/Transport Layer Security*) baseado em PKI.

- **Utilizando entidades de confiança distribuídas**

Este modelo não sofre do problema de um único ponto de falha, frequente em sistemas centralizados. Nesta opção não há uma autoridade central, e sim um conjunto de entidades distribuídas e autônomas, que trabalham em conjunto para estabelecer níveis de confiança para os participantes. O conceito de uma rede de confiança é utilizado pelo OpenPGP, padrão de criptografia largamente utilizado para cifragem de *e-mails*. O OpenPGP utiliza certificados, mas que, neste caso, são assinados por outros usuários de confiança. Assim, quanto maior o nível de confiabilidade de um usuário, mais confiança sua assinatura transmite ao detentor do certificado assinado.

A Figura 2 ilustra as duas maneiras discutidas anteriormente para o estabelecimento de uma relação de confiança entre entidades de uma rede. Em (a) observamos que dois participantes que possuem a confiança de um mesmo *root of trust* possuem uma relação de confiança entre si. Em (b) podemos visualizar o estabelecimento de confiança fazendo uso de uma autoridade central, e em (c) o estabelecimento de uma rede de confiança distribuída.

Figura 2 – Modos de se atingir uma relação de confiança



Fonte: Kim & Lee (2017)

2.3 DTLS

O DTLS é um protocolo responsável por prover segurança em comunicações baseadas em datagramas. Ele foi introduzido por Nagendra Modadugu e Eric Rescorla em 2014 e foi padronizado na RFC 4347 (*Request for Comments*) (Rescorla; Modadugu, 2006). O DTLS utiliza o UDP (*User Datagram Protocol*) como camada de transporte e é fortemente baseado no TLS, que por sua vez utiliza o TCP (*Transmission Control Protocol*). Ele reutiliza a infraestrutura de protocolo presente no TLS e provê algumas funcionalidades adicionais para habilitar suporte ao UDP (Roepke *et al.*, 2016).

O protocolo DTLS garante autenticação, confidencialidade e integridade. Devido às suas similaridades, o DTLS é considerado tão seguro quanto o TLS em configurações equivalentes e sob a suposição de que primitivas criptográficas semelhantes são utilizadas. Ele também suporta um protocolo de *handshake* para estabelecer um canal de comunicação seguro entre cliente e servidor, além de fornecer funcionalidades semelhantes ao TLS, tais

como mensagens de alerta e possibilidade de acordo de *cipher suites* entre cliente e servidor (Roepke *et al.*, 2016).

2.3.1 Handshake

Mensagens individuais são agrupadas no que é chamado de *message flights*, em que cada *flight* deve ser visto como um conjunto de mensagens único, de modo que a perda de mensagens dentro de um mesmo *flight* durante o *handshake* requer que todo o *flight* seja retransmitido (Rescorla; Modadugu, 2012). As mensagens trocadas durante o *handshake* são apresentadas na Figura 3.

Os *flights* 1 e 2 consistem de uma funcionalidade opcional presente no DTLS para prover proteção contra ataques DoS (*Denial of Service*). Ao enviar a mensagem *ClientHello* para o servidor, este pode responder com uma mensagem *HelloVerifyRequest*. Esta mensagem contém um *cookie* gerado pelo servidor, de modo que ele possa ser verificado sem a necessidade de reter qualquer estado no servidor (Rescorla; Modadugu, 2012).

Ao receber este *cookie*, o cliente deve retransmitir a mensagem *ClientHello* em conjunto com o *cookie* recebido, que é feito no *flight* 3. O servidor então verifica se o *cookie* é válido e, em caso afirmativo, prossegue com o *handshake*. Este mecanismo faz com que o cliente (ou atacante) seja forçado a receber uma mensagem contendo o *cookie*, inviabilizando ataques DoS com endereços IP (*Internet Protocol*) mascarados (Rescorla; Modadugu, 2012). A mensagem *ClientHello* inclui a versão do protocolo e uma lista de *cipher suites* suportadas pelo cliente (Kothmayr *et al.*, 2013).

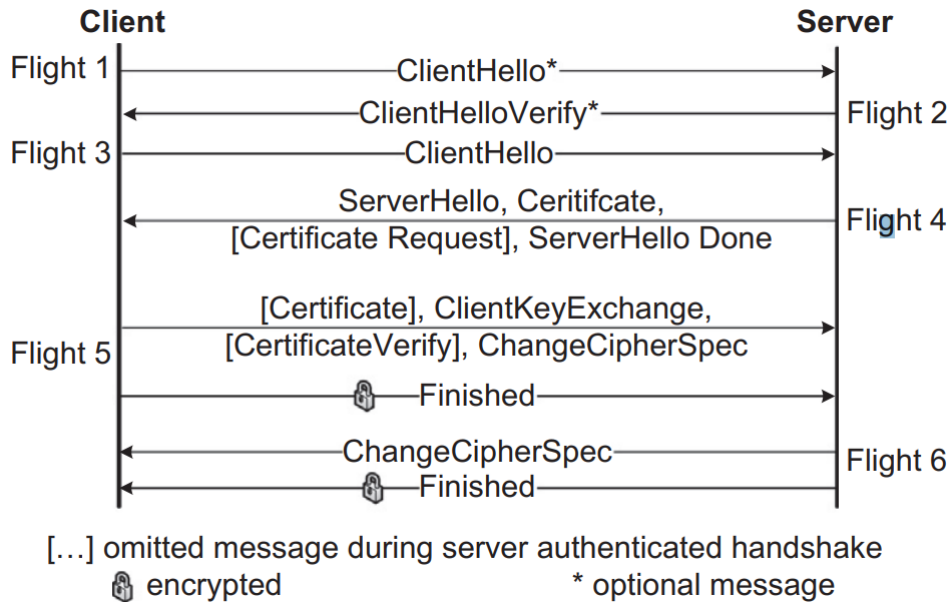
Em seguida, no *flight* 4, o servidor responde com uma mensagem *ServerHello* contendo a *cipher suite* escolhida. O servidor também inclui um certificado X.509 para se autenticar, seguindo de uma mensagem *CertificateRequest* caso o servidor requeira que o cliente também se autentique. Por último, a mensagem *ServerHelloDone* indica o término do *flight* 4 (Kothmayr *et al.*, 2013).

A mensagem *ClientKeyExchange*, enviada pelo cliente no *flight* 5, contém metade do *pre-master secret* cifrado com a chave pública RSA (*Rivest-Shamir-Adleman*) do certificado do servidor. *Pre-master secret* é um valor obtido ao realizar a troca de chaves ($g^{ab} \pmod{p}$), caso o algoritmo *Diffie-Hellman* seja utilizado). Caso haja necessidade, o cliente se autentica por meio da mensagem *CertificateVerify*. (Kothmayr *et al.*, 2013).

A mensagem *ChangeCipherSpec* indica que todas as mensagens enviadas a partir dali serão cifradas com a *cipher suite* negociada pelo *handshake*. Então, o cliente envia

uma mensagem *Finished*. Por fim, o servidor responde com seu próprio *ChangeCipherSpec* e envia uma mensagem *Finished* para completar o *handshake* (Kothmayr *et al.*, 2013).

Figura 3 – Mensagens de *handshake* do DTLS



Fonte: Kothmayr *et al.* (2013)

O protocolo DTLS foi projetado para se adaptar ao UDP como camada de transporte. No entanto, alguns problemas surgem à este respeito: o *handshake* do TLS é um *handshake* criptográfico de etapa única, em que mensagens devem ser transmitidas e recebidas em uma ordem definida. Claramente isto é incompatível com reordenação e perda de pacotes. Além disso, suas mensagens são significativamente maiores que qualquer datagrama UDP, criando problemas de fragmentação (Rescorla; Modadugu, 2012). Dito isto, o *handshake* do DTLS deve adicionar mecanismos adicionais para lidar com estes problemas.

Perda de pacotes

Para lidar com a perda de pacotes, o DTLS utiliza um temporizador para retransmissão de pacotes caso perdas sejam detectadas. Uma vez que o cliente transmite a mensagem *ClientHello* ao servidor, é esperado uma resposta com a mensagem *HelloVerifyRequest*. Caso a mensagem não tenha chegado antes da expiração do temporizador, então o cliente pode inferir que, ou a sua mensagem se perdeu, ou então a resposta do servidor se perdeu. Em qualquer um dos casos, ele retransmite seu *ClientHello* ao servidor. Quando o servidor receber a retransmissão, ele sabe que deverá retransmitir a resposta. Além disso, o servidor também mantém um temporizador, retransmitindo quando este temporizador expira (Rescorla; Modadugu, 2012).

Reordenação

Em DTLS, cada mensagem do *handshake* possui um número sequencial específico dentro daquele *handshake*. Quando uma das partes recebe uma destas mensagens, esta parte é capaz de inferir rapidamente se determinada mensagem é a próxima a ser processada. Se for, a parte a processa. Caso contrário, esta mensagem é adicionada em uma fila, e será processada assim que todas as mensagens anteriores já tiverem chegado e sido processadas (Rescorla; Modadugu, 2012).

Tamanho de mensagem

Mensagens TLS e DTLS podem ser bem grandes (em teoria, até $2^{24} - 1$ bytes). Por outro lado, datagramas UDP são frequentemente limitados a 1500 bytes caso a ocorrência de fragmentação nos pacotes IP não seja desejada. Para compensar esta limitação, cada mensagem de *handshake* do DTLS pode ser fragmentado em vários registros DTLS, cada qual como tamanho suficiente para se encaixar em um simples datagrama IP. Para isto, cada mensagem do *handshake* DTLS contém um *offset* de fragmento e o comprimento deste fragmento. Conseqüentemente, o destinatário com posse de todos os bytes de uma mensagem de *handshake* poderá remontar a mensagem original (Rescorla; Modadugu, 2012).

2.4 CoAP

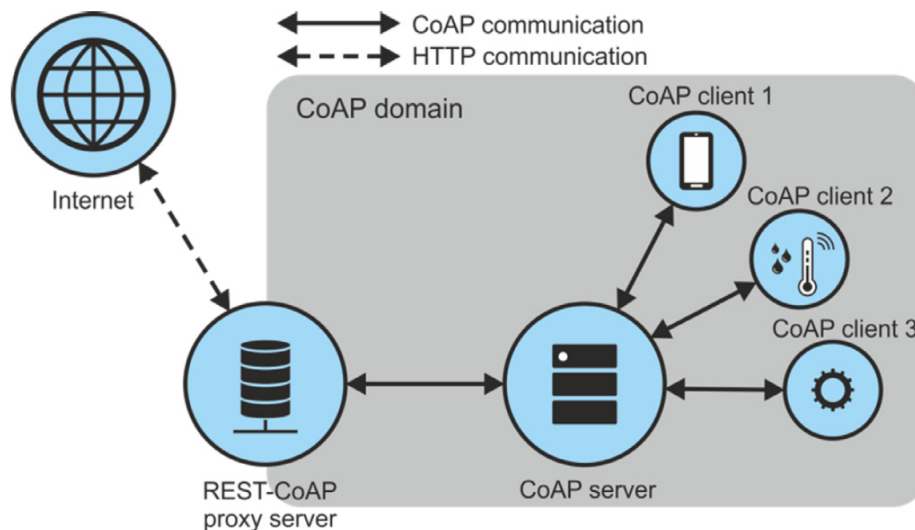
Para transferir dados de sensores coletados por um *gateway* para um servidor, é necessário um protocolo que seja otimizado em termos de largura de banda e eficiência energética, além de ser capaz de trabalhar com recursos de *hardware* limitados (ou seja, capacidade de processamento e espaço de memória reduzidos). Como resultado, protocolos tais como MQTT (*Message Queuing Telemetry Transport*) e CoAP foram propostos para tratar especificamente os difíceis requisitos de cenários de implantação de uma WSN (*Wireless Sensor Network*) no mundo real (Thangavel *et al.*, 2014).

CoAP é um protocolo da camada de aplicação que trabalha a nível M2M (*Machine to Machine*) (Tiburski; Matos; Hessel, 2019), sendo definido pela RFC 7252 (Shelby; Hartke; Bormann, 2014). O CoAP foi originalmente desenvolvido para transmissão de dados entre dispositivos com limitações de recursos em ambientes IoT (Rahman; Shah, 2016).

Este protocolo é fortemente baseado no protocolo HTTP (*Hyper Text Transfer Protocol*), utilizando também a arquitetura REST (*Representational State Transfer*), que

usa os métodos *GET*, *POST*, *PUT* e *DELETE* para manter uma interface uniforme e padronizada. Estes métodos servem para buscar, inserir, atualizar e deletar dados de aplicação em servidores remotos. Devido à estas características, o CoAP é facilmente conversível para o HTTP, permitindo que *gateways* intermediários façam a conversão entre ambos os protocolos, conforme a necessidade. A Figura 4 ilustra a arquitetura do CoAP.

Figura 4 – Arquitetura CoAP



Fonte: Glaroudis, Iossifides & Chatzimisios (2020)

No entanto, à medida que o HTTP é um protocolo robusto e pesado, dado que sua implementação requer bastante espaço em memória e produz mais tráfego na rede, o CoAP foi especialmente projetado para utilizar poucos recursos, tanto de *hardware* quanto da própria rede, adequando-se melhor a pequenos dispositivos e microcontroladores. Enquanto que o HTTP utiliza o TCP como camada de transporte, o CoAP faz uso do UDP, método mais leve para realizar a transmissão de dados.

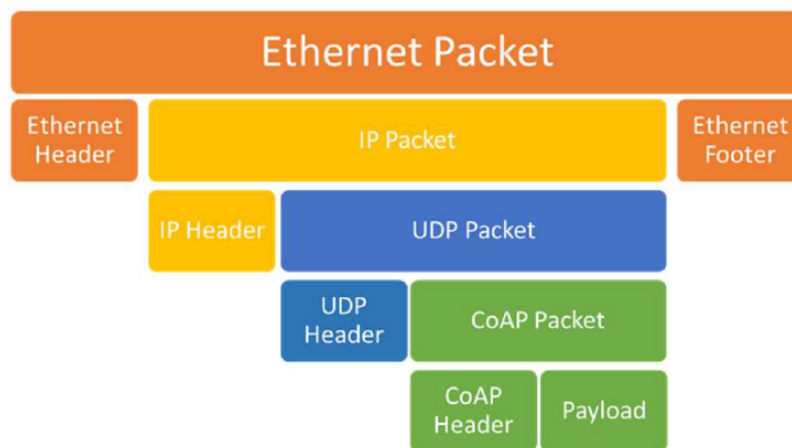
Além da arquitetura *request/response*, principal característica do HTTP, o CoAP também implementa a arquitetura *resource/observe*, uma variante do *publish/subscribe*, utilizando uma extensão do método *GET* (Thangavel *et al.*, 2014).

Por utilizar o protocolo UDP para a entrega de pacotes, não há garantias de que as mensagens serão entregues aos destinatários, o que é um grande problema em cenários onde a rede conta com uma taxa elevada de perda de pacotes. Portanto, torna-se responsabilidade do CoAP implementar estes mecanismos de qualidade. Para isso, sua arquitetura é dividida em duas camadas: camada de mensagem e camada *request/response*. A primeira camada é responsável por controlar a troca de mensagens sobre o UDP entre dois nós comunicantes. A segunda camada, por sua vez, lida com a parte de requisição

e resposta, armazenando o *method code* e o *response code* com a finalidade de evitar problemas como mensagens fora de ordem, perda e/ou duplicação de pacotes. Deste modo, CoAP oferece mecanismos confiáveis para retransmissão, detecção de duplicações e também funções de *multicast*.

O CoAP é considerado um protocolo leve no sentido de que o *overhead* do cabeçalho e demais métodos é significativamente menor do que em muitos outros protocolos a nível de aplicação. Os pacotes do CoAP são muito menores que os pacotes TCP do HTTP, e mapeamentos entre *strings* e números são extensivamente utilizados para reduzir a quantidade de *bits* dos campos (Glaroudis; Iossifides; Chatzimisios, 2020). A Figura 5 apresenta detalhes da arquitetura de pacote de uma mensagem CoAP.

Figura 5 – Arquitetura de pacote de uma mensagem CoAP



Fonte: Çavuşoğlu *et al.* (2020)

As principais bibliotecas que disponibilizam implementações do protocolo CoAP são as seguintes: Californium (Eclipse Foundation, c2020), libcoap (libcoap, c2020) e CoAPthon (Tanganelli; Vallati; Mingozi, 2015).

Para prover segurança em sua comunicação, o CoAP faz uso do protocolo DTLS. De acordo com a especificação RFC 7252 (Shelby; Hartke; Bormann, 2014) e com Rahman & Shah (2016), há quatro modos de segurança definidos para o CoAP, que são:

- **NoSec:** Neste modo não há nenhum nível de segurança no protocolo, e o DTLS é desabilitado.
- **PreSharedKey:** Este modo é habilitado por dispositivos sensíveis com chaves de criptografia simétrica pré-compartilhadas. Assim, é adequado para aplicações integradas à dispositivos que são incapazes de empregar criptografia de chave pública.

Além disso, aplicações podem usar uma chave por dispositivo, ou então uma chave por grupo de dispositivos.

- **RawPublicKey:** Este modo é obrigatório para dispositivos que requerem autenticação baseada em chave pública. Os dispositivos são programados com uma lista de chaves para que os dispositivos possam iniciar uma sessão DTLS sem certificado.
- **Certificates:** Este modo suporta autenticação baseada em chave pública e aplicações que participam da cadeia de certificação. A suposição desse modo é que a infraestrutura de segurança está disponível. Dispositivos que incluem chave assimétrica e têm certificados X.509 desconhecidos podem ser validados usando o modo de certificado e provisionando chaves raízes confiáveis.

A segurança do CoAP ainda está em discussão, embora o DTLS seja combinado como uma camada de proteção. O debate é o alto custo de computação e o pesado *handshake*, o que causa fragmentação de mensagens. Além disso, o gerenciamento de chaves é outro problema na segurança do CoAP, sendo um problema comum em quase todos os protocolos (Rahman; Shah, 2016).

2.5 MQTT

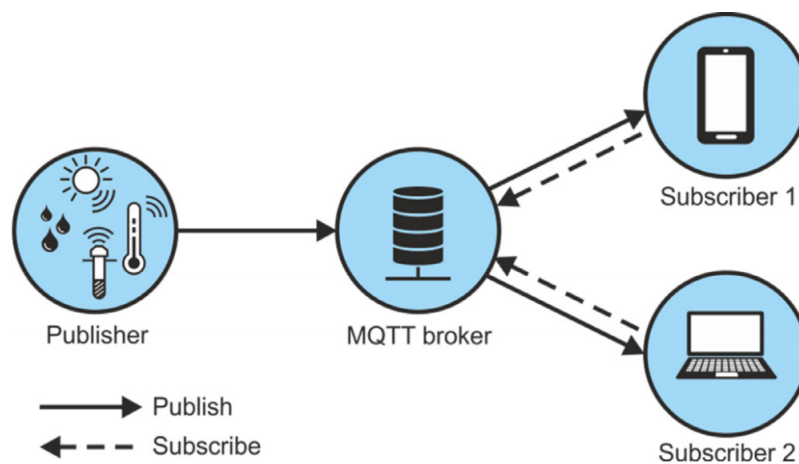
O MQTT é um protocolo de aplicação construído na arquitetura *publish/subscribe*, projetado para comunicações M2M em redes com recursos limitados. Ele é um protocolo binário e normalmente requer um cabeçalho fixo de 2 *bytes*, com *payload* máximo de até 256MB (Naik, 2017). O MQTT utiliza o TCP como camada de transporte e, conseqüentemente, utiliza o TLS para prover segurança.

A arquitetura do protocolo MQTT é composta por dois tipos de entidades: cliente e servidor (*broker*). A comunicação ocorre entre cliente e *broker*. O papel do *broker* é realizar o gerenciamento das mensagens entre clientes.

Diferentemente do modelo *request/response*, muito utilizado em arquiteturas cliente/servidor, o modelo *publish/subscribe* trabalha de maneira um pouco diferente. Para enviar dados, um dispositivo cliente deve enviar uma mensagem *publish* para o *broker*, informando o tópico da mensagem e seu conteúdo. O tópico da mensagem é uma *string* que referencia um determinado canal de mensagens. O sistema de tópicos utiliza um mecanismo de hierarquia, onde *strings* separadas por barras (/) indicam os níveis e subníveis do tópico (Dizdarević *et al.*, 2019).

Caso algum dispositivo cliente queira receber dados de um tópico específico, ele deve enviar uma mensagem *subscribe* ao *broker*, informando o tópico desejado. Neste caso, sempre que um cliente enviar dados referentes à este tópico, o *broker* encaminhará a mensagem imediatamente para o dispositivo que realizou o *subscribe*. Por utilizar o TCP, a comunicação entre cliente e *broker* é orientada a conexão. A Figura 6 ilustra arquitetura do protocolo MQTT.

Figura 6 – Arquitetura MQTT



Fonte: Glaroudis, Iossifides & Chatzimisios (2020)

Em relação à qualidade de serviço, o MQTT fornece três níveis de QoS (*Quality of Service*) (Mosquitto, c2020), sendo eles:

- **QoS 0:** Este é o nível mínimo de QoS, e não oferece garantia de entrega da mensagem. O destinatário não confirma o recebimento da mensagem e a mensagem não é armazenada e retransmitida pelo remetente. As garantias desse nível contam somente com as garantias fornecidas pelo protocolo TCP.
- **QoS 1:** O nível 1 de QoS garante que a mensagem seja entregue pelo menos uma vez ao destinatário. O remetente armazena a mensagem até obter um pacote do destinatário, confirmando o recebimento da mensagem. Há a possibilidade de a mensagem ser enviada mais de uma vez.
- **QoS 2:** O nível 2 de QoS é o nível mais alto no MQTT. Este nível garante que cada mensagem seja recebida apenas uma única vez pelo destinatário. Este é o nível de qualidade mais seguro e lento.

Mesmo que o MQTT tenha sido projetado para ser um protocolo leve e eficiente para utilização em dispositivos IoT, ele apresenta dois problemas que são inerentes à sua

implementação. Um deles é a utilização do TCP, o que força com que todos os clientes também tenham que suportá-lo, tornando a sua utilização onerosa para dispositivos muito pequenos. Além disso, o TCP mantém uma conexão aberta com o *broker* a cada troca de mensagens, consumindo recursos destes dispositivos.

O outro problema diz respeito ao sistema de tópicos utilizados em sua arquitetura, já que geralmente são cadeias de caracteres muito longas, tornando-se impraticável para alguns protocolos de camadas mais baixas.

No entanto, ambos os problemas estão sendo tratados pelo protocolo MQTT-SN (*Message Queuing Telemetry Transport for Sensor Network*), uma versão mais leve e otimizada do MQTT que foi proposto recentemente, projetado especificamente para reduzir o consumo de recursos e energia. Esta versão do MQTT utiliza o UDP como protocolo de transporte, em vez do TCP, e também provê configurações adicionais como menor tamanho de *payload*, suporte para tópicos com nomes indexáveis por parte do *broker* (onde as entradas passam a ser numéricas), entre outras melhorias (Glaroudis; Iossifides; Chatzimisios, 2020).

3 ESTADO DA ARTE

3.1 CoAP + DTLS: A Comprehensive Overview of Cryptographic Performance on an IOT Scenario

No trabalho de Westphall *et al.* (2020), os autores desenvolveram uma aplicação IoT para a área de agricultura com a finalidade de mensurar a quantidade de pesticida aplicado em plantações, e também coletar informações sobre a temperatura em pontos específicos do terreno, utilizando sensores de temperatura e GPS (*Global Positioning System*). Esta aplicação faz parte de um ambiente *fog* e *cloud*, e utilizou os protocolos CoAP e DTLS.

Sensores foram conectados diretamente a uma Raspberry Pi 3, a qual servia de cliente. Os dados coletados eram enviados para o servidor, executando na *cloud*, por meio dos protocolos CoAP e DTLS. Para a implementação do cliente e servidor, os autores optaram pela utilização da biblioteca CoAPthon (Tanganelli; Vallati; Mingozzi, 2015).

Os autores também realizaram uma série de testes quantitativos acerca do desempenho de diversas *cipher suites* aplicadas à seu modelo. As *cipher suites* analisadas são apresentadas na Tabela 1.

Tabela 1 – *Cipher suites* analisadas no trabalho correlato 1

<i>Cipher suite</i>
ECDHE-RSA-AES256-GCM-SHA384
ECDHE-RSA-AES128-GCM-SHA256
ECDHE-ECDSA-AES256-GCM-SHA384
ECDHE-RSA-AES256-SHA384
ECDHE-RSA-AES128-SHA256
ECDHE-ECDSA-AES256-SHA384

Fonte: Westphall *et al.* (2020)

Em seus testes, os autores utilizaram apenas mensagens *POST* com *confirmable requests*. Os resultados mostraram que a utilização de *cipher suites* com AES256, ou

AES128, tende a ser mais veloz se aplicadas ao modo GCM, quando comparadas com outras *cipher suites* com o mesmo algoritmo mas que utilizam modos diferentes.

3.2 Design and Implement of a Weather Monitoring Station using CoAP on NB-IoT Network

No trabalho de Kaewwongsri & Silanon (2020), os autores descreveram a implementação de um protótipo de uma estação climática, a qual é capaz de monitorar e coletar dados climáticos em tempo real. Esta estação utilizou uma placa Arduino e alguns sensores para mensurar variáveis como temperatura, umidade, velocidade e direção do vento, gás ozônio, pressão atmosférica e dados de chuva.

Para isto, o protótipo dos autores trabalhou com NB-IoT (*Narrowband Internet of Things*). NB-IoT é uma tecnologia de rádio-frequência desenvolvida para suportar um grande conjunto de casos de uso em comunicações M2M. Comparado com tecnologias 4G, o NB-IoT apresenta como características-chave maiores áreas de cobertura, maior economia de energia e um conjunto reduzido de funcionalidades. Estes recursos permitem a conectividade de dispositivos em posições mais difíceis, aumentando a vida útil da bateria e reduzindo a complexidade do dispositivo (Feltrin *et al.*, 2019).

Para realizar a transmissão dos dados entre a placa Arduino e o servidor de forma eficiente, os autores optaram pela utilização do protocolo CoAP. Assim que os dados chegam ao servidor, eles são armazenados em um banco de dados MySQL.

Por fim, os autores utilizam uma ferramenta de código aberto chamada Grafana (Grafana Labs, c2020), para exibir os dados coletados em forma de gráficos interativos.

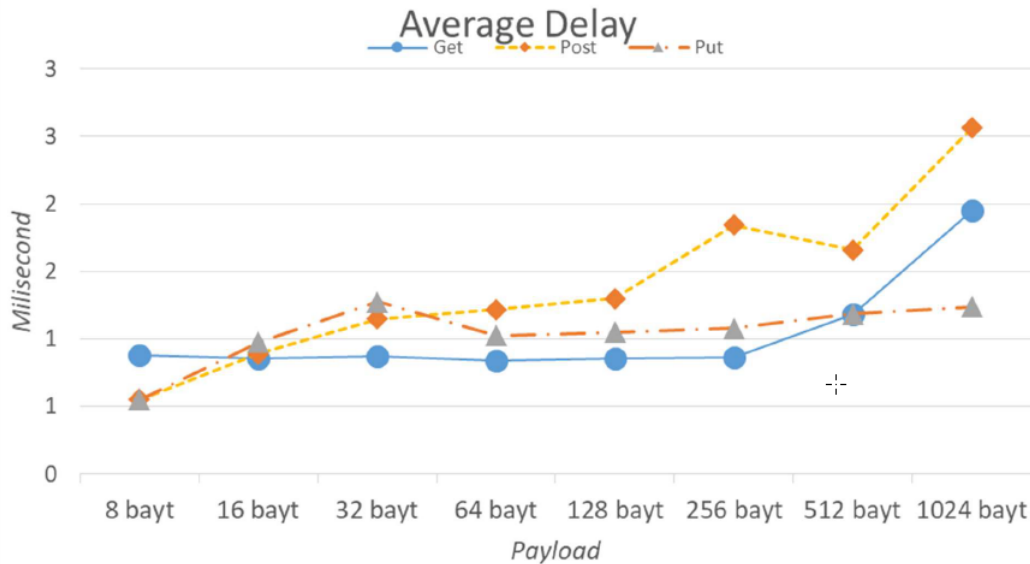
3.3 CoAP and Its Performance Evaluation

Em seu trabalho, Çavuşoğlu *et al.* (2020) avaliaram o desempenho do protocolo CoAP, analisando parâmetros como tempos de atraso, taxas de vazão e consumo de energia por mensagem. Estes critérios foram analisados utilizando os métodos *GET*, *POST* e *PUT* do protocolo CoAP.

Para isso, os autores utilizaram um dispositivo WeMOS D1, operando na frequência de 80/160 MHz, com 4 MB de memória e um módulo Wi-Fi ESP8266EX. O servidor contava com um processador *i7*, de 2.4 GHz e 8 GB de memória RAM.

Os testes conduzidos pelos autores utilizaram vários tamanhos de mensagem, de acordo com diferentes métodos do CoAP, e são apresentados na Figura 7.

Figura 7 – Tempos de atraso de mensagens CoAP



Fonte: Çavuşoğlu *et al.* (2020)

3.4 Evaluating the DTLS Protocol from CoAP in Fog-to-Fog Communications

Em seu trabalho, Tiburski, Matos & Hessel (2019) avaliaram o protocolo CoAP com DTLS em comunicações *fog-to-fog* operando em RANs (*Radio Access Networks*) com diferentes taxas de perda de pacotes e latência. Seus experimentos levaram em conta parâmetros de desempenho, *overhead* e problemas de *handshake*.

Para compor o ambiente, os autores adotaram uma infraestrutura composta de dois dispositivos *fog*, utilizando o mesmo *software* e a mesma configuração de *hardware* para ambos. Estes dispositivos utilizaram o sistema operacional Ubuntu 16.04 LTS, de 64 *bits*, com um processador *dual core* de 2.2 GHz e 2 GB de memória RAM. Estes dois dispositivos *fog* se comunicavam entre si por meio de uma RAN. Além disso, um aparelho celular foi utilizado como *gateway* para intermediar a comunicação entre os dispositivos *fog*, fazendo a conexão com a RAN.

Um dos dispositivos *fog* foi utilizado como cliente DTLS, enquanto que o outro foi utilizado como servidor. Ambas as entidades foram implementadas em Java por meio da biblioteca Scandium (parte do projeto *Californium Eclipse Project*).

Os experimentos conduzidos pelos autores foram compostos de 1000 interações entre ambos os dispositivos através da RAN, e foram aplicados em três tipos diferentes de RANs: EDGE (2.75G), HSPA+ (3.5G) e LTE (4G).

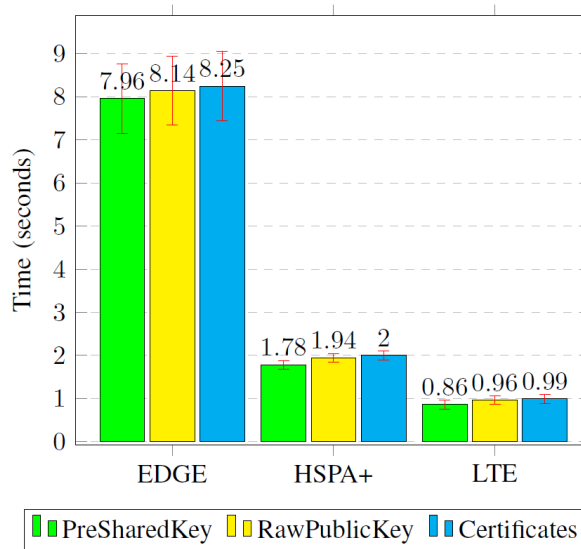
Os testes foram voltados apenas ao *handshake* do DTLS, utilizando os três modos de segurança do CoAP – *PreSharedKey*, *RawPublicKey* e *Certificates*. A Tabela 2 lista as *cipher suites* utilizadas nos testes, e a Figura 8 apresenta os resultados obtidos.

Tabela 2 – *Cipher suites* analisadas pelo trabalho correlato 4

Modo	<i>Cipher suite</i>
<i>PreSharedKey</i>	TLS_PSK_WITH_AES_128_CCM_8
<i>RawPublicKey</i>	TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8
<i>Certificates</i>	TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8

Fonte: Tiburski, Matos & Hessel (2019)

Figura 8 – Tempo necessário para estabelecer uma conexão segura entre nós *fog*



Fonte: Tiburski, Matos & Hessel (2019)

Com base nos resultados de seus testes, os autores concluíram que o uso do protocolo DTLS em comunicações *fog-to-fog* é viável em redes RAN mais velozes e robustas, apresentando um desempenho pior em redes mais lentas. O *overhead* adicionado pelo DTLS ao CoAP não é relacionado somente à camada de segurança. Uma vez que o DTLS não foi projetado para ambientes IoT, adaptações foram realizadas para lidar com problemas de confiabilidade (Tiburski; Matos; Hessel, 2019).

Como trabalhos futuros, os autores pretendem expandir sua avaliações analisando comunicações entre dispositivos de capacidades mais limitadas.

3.5 Comparação com este trabalho

Este trabalho realizou o desenvolvimento de uma aplicação IoT, tendo em vista a utilização de protocolos energeticamente eficientes e de baixo processamento. Para isto, adotou-se o protocolo CoAP, que tem demonstrado ser mais eficiente e econômico que seus pares, ideal para cenários mais restritivos. Todos os trabalhos discutidos nesta seção adotaram o CoAP como protocolo de comunicação.

Para incluir mecanismos de segurança e garantir a confidencialidade dos dados trocados entre as partes, este trabalho adotou o protocolo DTLS, que é uma versão do TLS adaptada para o uso com o UDP. Este protocolo não foi projetado especificamente para ambientes restritos, e por isso apresenta alguns problemas de desempenho nestes ambientes. Os autores Westphall *et al.* (2020) e Tiburski, Matos & Hessel (2019) também incluíram o DTLS em seus trabalhos, realizando testes de desempenho acerca do protocolo e da utilização de diferentes *cipher suites* aplicadas ao modelo. Por fim, estes trabalhos apresentaram resultados obtidos por meio destes experimentos.

Este trabalho também deve como objetivo trabalhar com dispositivos que, embora possuam recursos suficientes para a utilização de protocolos mais robustos, tais como o HTTP, ainda são considerados limitados em relação à maioria dos dispositivos conectados à Internet. Contudo, estes dispositivos também se beneficiam do uso de protocolos mais leves e eficientes. Entre os trabalhos correlatos, apenas Kaewwongsri & Silanon (2020) e Çavuşoğlu *et al.* (2020) incorporaram dispositivos mais restritos em seus trabalhos, porém sem tratar sobre questões de segurança.

A Tabela 3 resumiu as principais contribuições empreendidas pelos trabalhos correlatos analisados neste capítulo, tendo em consideração os seguintes pontos:

- Utilização de modelos incorporando dispositivos IoT, ou seja, microcontroladores e dispositivos com restrições de recursos;
- Utilização do protocolo CoAP para realizar a comunicação entre as entidades;
- Utilização do protocolo DTLS como mecanismo de segurança para a comunicação;
- Realização de testes quantitativos aplicados ao modelo, apresentando seus resultados.

As contribuições alcançadas neste trabalho englobaram a utilização dos protocolos CoAP e DTLS em um ambiente IoT, composto de dispositivos com restrições de energia e

processamento. Por fim, testes foram efetuados sob o modelo examinando o impacto de diferentes *cipher suites* no tempo total de processamento e no consumo de energia destes protocolos, apresentando os resultados obtidos. Os tópicos cobertos por este trabalho foram explorados apenas parcialmente pelos demais trabalhos correlatos.

Tabela 3 – Contribuições deste e de outros trabalhos correlatos

Trabalho	Dispositivos IoT	CoAP	DTLS	Mensurações
Westphall <i>et al.</i> (2020)	-	✓	✓	✓
Kaewwongsri & Silanon (2020)	✓	✓	-	-
Çavuşoğlu <i>et al.</i> (2020)	✓	✓	-	✓
Tiburski, Matos & Hessel (2019)	-	✓	✓	✓
Este trabalho	✓	✓	✓	✓

Fonte: Elaborado pelo autor (2020)

4 DESENVOLVIMENTO

4.1 Contexto

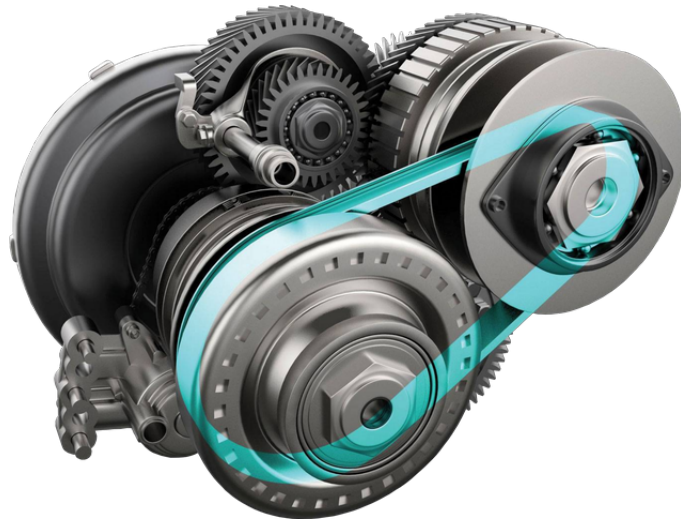
Baja SAE é uma competição conduzida pela *Society of Automotive Engineers International* (SAE *International*), na qual equipes de universidades de todo o mundo projetam e constroem pequenos carros *off-road*, em que todos os carros possuem motores com as mesmas especificações. O objetivo destas competições é projetar, construir e conduzir veículos *off-road* que possam suportar os elementos mais adversos de terrenos acidentados (SAE *International*, c2020). No Brasil, a SAE Brasil é a organização responsável por promover os eventos nacionais e regionais (SAE Brasil, c2020). As equipes vencedoras do campeonato nacional recebem o direito de competir as finais nos Estados Unidos.

Durante estas competições, os veículos construídos por estas equipes passam por diversos testes de segurança e provas dinâmicas para avaliar aspectos dos veículos como suspensão, tração, aceleração e retomada, entre outros. No fim da competição há uma prova de resistência, conhecida como Enduro, que possui quatro horas de duração.

Durante estas provas de resistência, é muito comum o superaquecimento do câmbio CVT (*Continuously Variable Transmission*), presente em alguns veículos. Dentro deste câmbio existe uma correia que tende a sofrer danos quando a temperatura ultrapassa um certo limiar (Sano, 2013), (Hardy *et al.*, 2019). O monitoramento da temperatura interna do câmbio CVT em tempo real é importante para evitar complicações durante a corrida, alertando o piloto e a equipe antecipadamente para efetuar a troca da correia. A Figura 9 apresenta uma ilustração do câmbio CVT.

Além disso, também é possível monitorar a pressão interna dos pneus dos veículos por meio de sensores acoplados em seu interior. A pressão dos pneus causa um grande impacto na dinâmica do veículo, e assim também é possível detectar furos e vazamentos durante a corrida.

Figura 9 – Ilustração de um câmbio CVT

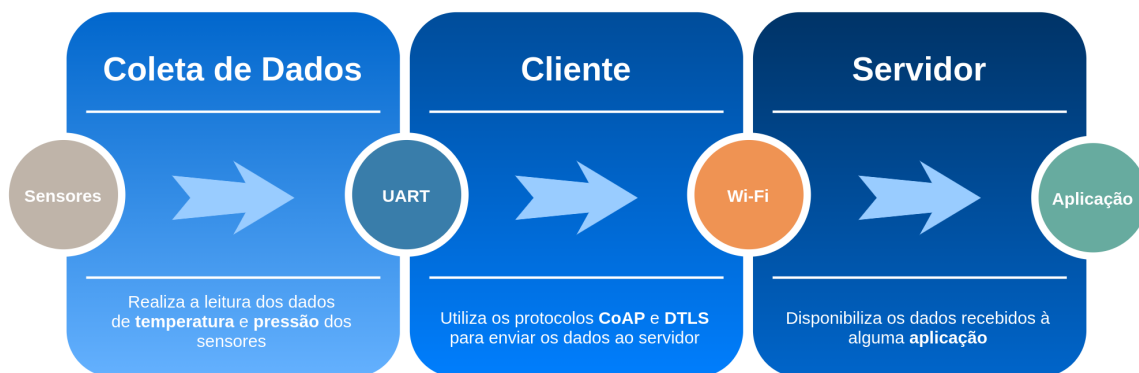


Fonte: Quatro Rodas (2018)

4.2 Modelo

O modelo deste trabalho consiste basicamente em três aplicações principais: uma aplicação para realizar a coleta de dados dos sensores; uma aplicação cliente, implementando os protocolos CoAP e DTLS; e uma aplicação para o servidor, que recebe os dados provenientes do cliente, e então permite o acesso destes dados à demais aplicações. A maneira em que estas aplicações foram organizadas é apresentada na Figura 10.

Figura 10 – Organização das aplicações do modelo

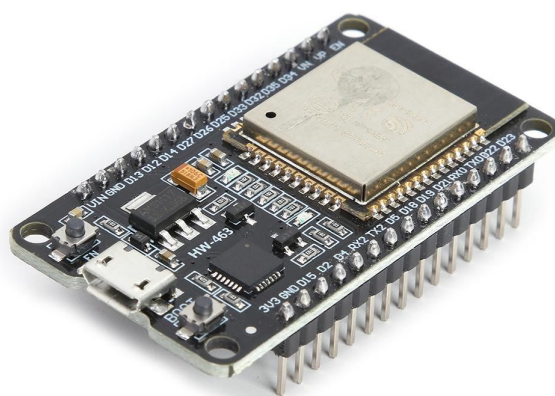


Fonte: Elaborado pelo autor (2020)

4.2.1 Aplicação: Coleta de dados

A coleta de dados dos sensores é realizada por uma aplicação separada, utilizando um dispositivo ESP32 WROOM-32 DevKit V1 e um sensor BMP280. Este sensor é capaz de obter informações de temperatura e pressão barométrica do ambiente. O dispositivo ESP32 WROOM-32 DevKit V1 é exibido na Figura 11.

Figura 11 – Dispositivo ESP32 WROOM-32 DevKit V1



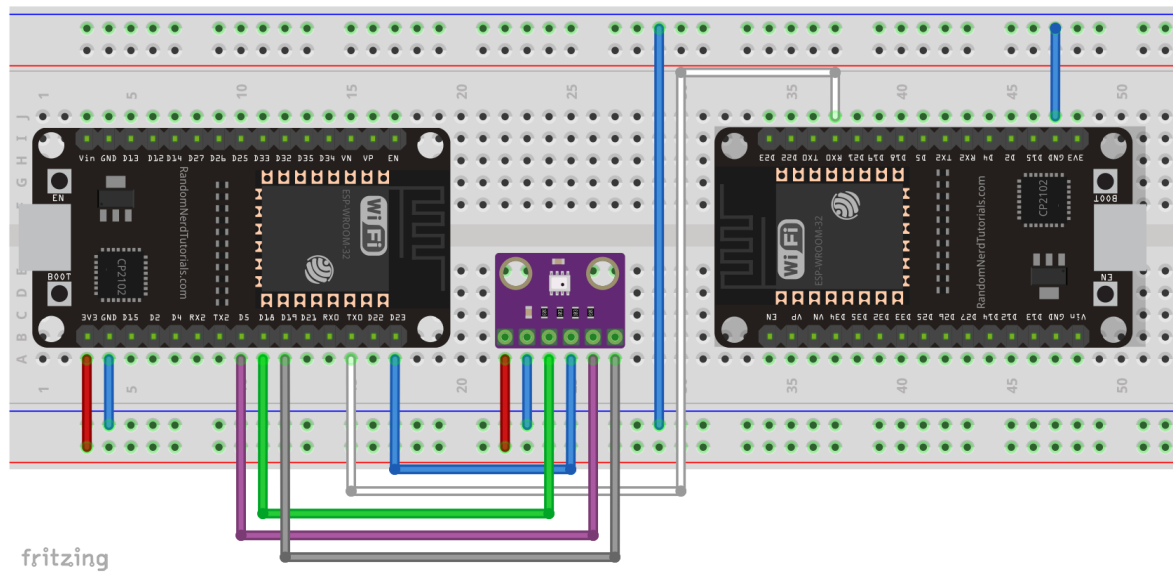
Fonte: https://img.cesdeals.com/products/3/4/1/341313/MAIN_341313-3.jpg

Esta aplicação utilizou o *framework* Arduino para oferecer compatibilidade com a biblioteca Adafruit_BMP280, responsável por ler os dados provenientes do sensor BMP280.

A aplicação consiste em um *loop* que faz a leitura dos dados do sensor em intervalos de tempo regulares. Assim que os dados são lidos, eles são redirecionados à aplicação cliente. Como ambas as aplicações estão em dispositivos distintos, foi estabelecido um canal de comunicação entre elas por meio do barramento UART. Deste modo, quando os dados são lidos do sensor, eles são imediatamente escritos no barramento UART, permitindo que a aplicação cliente possa lê-los.

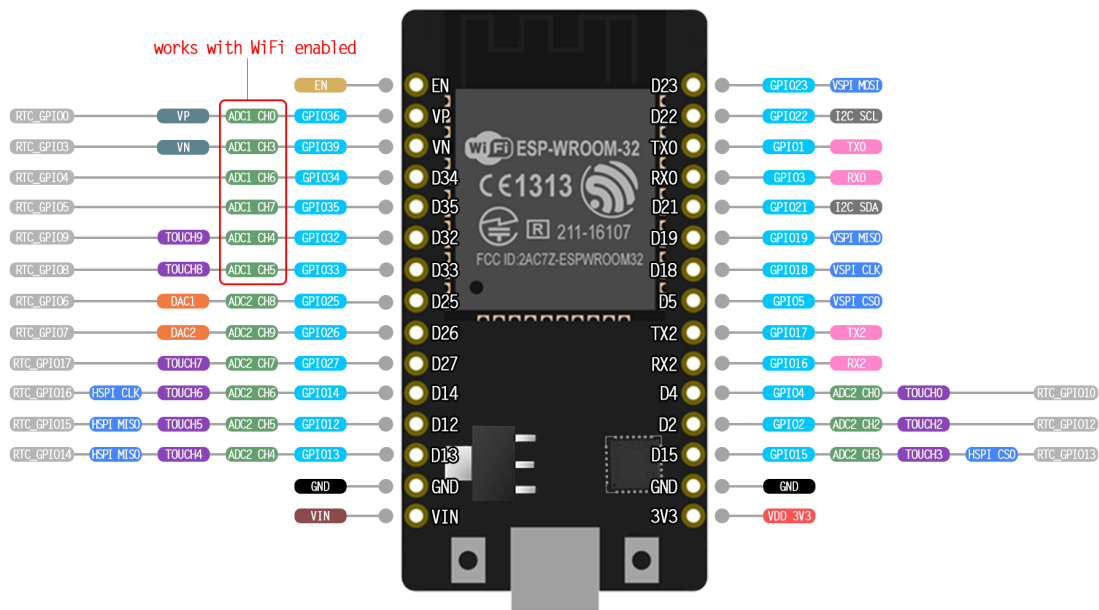
Para estabelecer a comunicação UART, o pino TX0 do dispositivo responsável pelo sensor foi conectado com o pino RX0 do dispositivo da aplicação cliente, estabelecendo assim um canal de comunicação *simplex*. Saídas TX permitem a escrita de dados seriais no barramento UART, enquanto que entradas RX permitem a leitura deste barramento. A Figura 12 apresenta o protótipo com a conexão de ambas as aplicações. Detalhes sobre a pinagem do ESP32 WROOM-32 DevKit V1 são apresentados na Figura 13.

Figura 12 – Representação do protótipo incluindo as aplicações de coleta de dados e cliente



Fonte: Elaborado pelo autor (2020)

Figura 13 – Detalhes da pinagem do ESP32 WROOM-32 DevKit V1



Fonte: Elaborado pelo autor (2020)

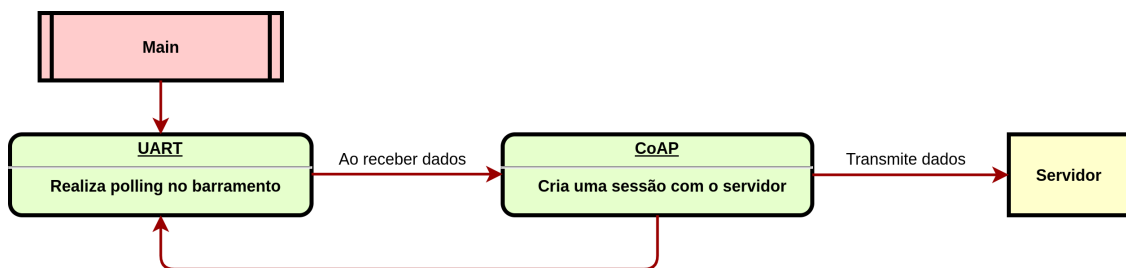
4.2.2 Aplicação: Cliente

Esta aplicação foi desenvolvida para implementar um cliente CoAP com DTLS, estabelecendo um canal de comunicação seguro com o servidor. Para isto, o exemplo `coap_client`, disponibilizado pelo *framework* ESP-IDF, foi utilizado como base. Esta aplicação utilizou somente um dispositivo ESP32 WROOM-32 DevKit V1. A transmissão

dos dados para o servidor é feita via rede *Wi-Fi*.

A Figura 14 apresenta o fluxo de execução da aplicação cliente. Ao iniciar a aplicação, o módulo UART realiza um *polling* no barramento em busca de novos dados. Quando a chegada de dados é detectada, o sistema cria uma nova *thread* para estabelecer uma sessão CoAP com o servidor. Com a sessão criada, os dados são transferidos para o servidor e a *thread* é finalizada. Enquanto isso, a *thread* principal continua a realizar o *polling* no barramento.

Figura 14 – Fluxo de execução da aplicação cliente



Fonte: Elaborado pelo autor (2020)

Este trabalho utilizou dois dispositivos ESP32 no protótipo por motivos de compatibilidade. O *framework* ESP-IDF foi o único *framework* que implementava um cliente CoAP com DTLS para dispositivos embarcados. Em contrapartida, não foram encontradas bibliotecas para lidar com o sensor BMP280 compatíveis com o ESP-IDF. Bibliotecas para estes fins geralmente são implementadas tendo o *framework* Arduino como plataforma base. Para contornar este problema, adotou-se uma arquitetura com dois dispositivos distintos: um dispositivo para a implementação do cliente CoAP, utilizando o *framework* ESP-IDF; e outro dispositivo para o gerenciamento dos sensores, utilizando bibliotecas Arduino. A comunicação entre ambos os dispositivos foi feita por meio do barramento UART.

4.2.3 Aplicação: Servidor

A aplicação do servidor foi desenvolvida tendo como base o exemplo `coap-server`, disponibilizado pela biblioteca `libcoap`, utilizando a linguagem de programação C. Este exemplo, contudo, não implementava um recurso com o método *POST*, necessário para o contexto deste trabalho. Para tanto, foi incluído um método *handler* para o recurso `/sensor`, criado para receber os dados provenientes da aplicação cliente. Este método *handler* foi associado ao método *POST*. Isto significa que, sempre que o servidor receber uma mensagem *POST* direcionada ao recurso `/sensor`, a função *handler* será executada.

Inicialmente, a aplicação do servidor foi executada em um *notebook*. Mas, para efeitos de testes, também foi utilizado em uma placa integrada Raspberry Pi 3. A ideia foi mensurar os tempos de processamento do servidor em dispositivos de diferentes capacidades.

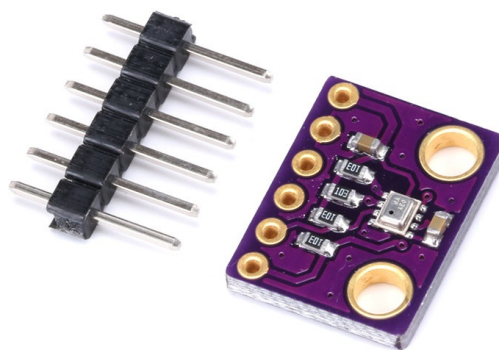
As seções seguintes irão apresentar com mais detalhes os equipamentos utilizados neste trabalho.

4.3 Equipamentos utilizados

4.3.1 Sensor BMP280

O sensor BMP280 é um sensor de temperatura e pressão barométrica especialmente projetado para aplicações móveis. Seu baixo consumo de energia permite a implementação em dispositivos alimentados por bateria (Bosch Sensortec, c2020). A comunicação entre o BMP280 e o microcontrolador pode ocorrer através da interface I2C (*Inter-Integrated Circuit*) ou SPI (*Serial Peripheral Interface*). Sua tensão de operação é 3.3V, podendo ser ligado diretamente em pequenas baterias de 3V. Ele é ideal para utilização em projetos de *drones*, estações meteorológicas, entre outros. O BMP280 pode ser visto na Figura 15.

Figura 15 – Sensor de temperatura e pressão barométrica BMP280



Fonte: <https://uploads.filipeflop.com/2017/07/1-16.jpg>

4.3.2 Dispositivos ESP32

O ESP32 é um microcontrolador de baixo custo e baixo consumo de energia, criado e desenvolvido pela Espressif Systems, sendo considerado o sucessor do microcontrolador ESP8266.

O ESP32 conta com dois processadores – sendo o principal deles um *Tensilica Xtensa LX6 dual-core* de 32 bits, operando em 160 MHz ou 240 Mhz, e um co-processador de baixa potência ULP (*Ultra Low Power*) – incluindo 520 KB de memória SRAM (*Static Random-Access Memory*), conectividade sem fio Wi-Fi (802.11 b/g/n) e Bluetooth. Além disso, ele possui módulos de *hardware* para aceleração de métodos criptográficos, como AES (*Advanced Encryption Standard*), SHA-2 (*Secure Hash Algorithm*), RSA, criptografia de curvas elípticas e gerador de números aleatórios. Ele também possui módulos para o gerenciamento de energia, em que pode passar a consumir até 5 uA em modo *deep sleep*, podendo ser acordado através de sinais de interrupções (Espressif Systems, c2020d).

Projetado para dispositivos móveis, eletrônicos vestíveis e aplicativos IoT, o ESP32 atinge um consumo de energia muito baixo devido à inclusão de vários modos de energia e escalonamento de energia dinâmico. Ele é capaz de funcionar de forma confiável em ambientes industriais, com uma temperatura de operação variando de -40 °C a +125 °C. Alimentado por circuitos de calibração avançados, o ESP32 pode remover dinamicamente as imperfeições do circuito externo e se adaptar às mudanças nas condições externas (Espressif Systems, c2020c).

Além de ser possível programá-lo utilizando o *framework* Arduino, o ESP32 também conta com um *framework* próprio, conhecido como ESP-IDF (Espressif Systems, c2020a). O ESP-IDF é o *framework* de desenvolvimento oficial para o ESP32 e ESP32-S Series SoCs. O ESP-IDF possibilita a programação em um nível muito mais próximo do *hardware*, permitindo o controle de componentes específicos com maior precisão. Este trabalho utilizou o *framework* Arduino para a implementação do gerenciamento de sensores e o *framework* ESP-IDF para a implementação do cliente CoAP DTLS.

4.3.3 Raspberry Pi 3 Model B

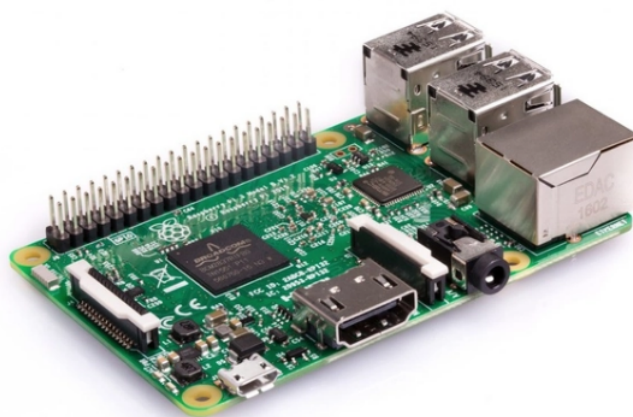
Raspberry Pi 3 é uma placa integrada que funciona como um minicomputador. Ela possui diversas interfaces presentes em um computador pessoal, como interfaces de áudio, vídeo, rede e entradas USB. Ela foi desenvolvida com a finalidade de facilitar o entendimento de programação para jovens e adultos. Por apresentar um consumo

de energia menor do que um computador normal, ela é muito utilizada como recurso computacional em diversos projetos.

As especificações do modelo Raspberry Pi 3 Model B, o qual foi utilizado neste trabalho, são descritas a seguir.

- *Quad Core* 1.2 GHz Broadcom BCM2837 de 64 *bits*
- 1 GB de memória RAM
- Wi-Fi e *Bluetooth Low Energy* (BLE)
- Ethernet
- 40 pinos GPIO
- 1 entrada Micro SD
- 1 entrada HDMI
- 1 entrada de áudio
- 4 portas USB 2.0
- Entrada Micro USB para fonte de alimentação de até 2.5 A

Figura 16 – Raspberry Pi 3 Model B



Fonte: Raspberry Pi Foundation (c2020)

4.4 Detalhes de implementação

Esta seção mostrará detalhes mais específicos sobre a implementação de cada módulo, explicando as funções mais relevantes para este trabalho.

4.4.1 Bibliotecas utilizadas

4.4.1.1 Adafruit BMP280

Adafruit BMP280 é uma biblioteca que possibilita o gerenciamento de diversas funções do sensor de BMP280, sendo compatível com um grande conjunto de microcontroladores. Esta biblioteca é mantida pela Adafruit e o seu código-fonte está disponível em seu repositório no GitHub:

https://github.com/adafruit/Adafruit_BMP280_Library

Esta biblioteca foi utilizada pela aplicação de coleta de dados do sensor, habilitando a leitura de informações do sensor BMP280.

4.4.1.2 ESP-IDF

ESP-IDF é o *framework* de desenvolvimento oficial para os dispositivos ESP32. Ele é disponibilizado para ambientes Linux, Windows e macOS, e seu código-fonte está disponível em seu repositório no GitHub:

<https://github.com/espressif/esp-idf>

Este repositório contém uma série de exemplos de aplicações envolvendo os recursos do ESP32, tais como Wi-Fi, Bluetooth, Ethernet, periféricos, protocolos, armazenamento, sistema, entre outros. Os exemplos estão organizados em categorias, e podem ser encontrados no diretório `examples`, localizado no diretório raiz do projeto.

Para a implementação de um cliente CoAP com DTLS, este trabalho utilizou como base o exemplo de cliente implementado pelo ESP-IDF, o qual pode ser encontrado no diretório `examples/protocols/coap_client`. Este exemplo utiliza biblioteca `libcoap` para a implementação do protocolo CoAP, adaptando-a para a arquitetura do ESP32.

Detalhes de instalação, configuração e utilização do ESP-IDF podem ser encontrados no Apêndice A deste trabalho.

4.4.1.3 libcoap

O libcoap é uma biblioteca escrita na linguagem C que implementa o protocolo CoAP, sendo definido pela RFC 7252 (Shelby; Hartke; Bormann, 2014) e projetado para dispositivos que possuem recursos limitados. O libcoap é uma biblioteca multiplataforma, sendo projetada para executar tanto em dispositivos embarcados como em sistemas computacionais de alto desempenho.

A biblioteca oferece as principais funcionalidades para o desenvolvimento de clientes e servidores CoAP, incluindo funcionalidades de *resource observation*, transferência em blocos, *FETCH/PATCH*, *No-Response* e TCP. Esta biblioteca foi projetada para suportar a segurança na camada de transporte utilizando *frameworks* tais como GnuTLS, OpenSSL, mbedTLS e tinydtls (libcoap, c2020).

A biblioteca também possui alguns exemplos de implementações, mostrando como suas funcionalidades podem ser utilizadas em diversas aplicações, incluindo a implementação de um cliente e um servidor, que traz várias funcionalidades *server-side* presentes no libcoap. O código-fonte do libcoap pode ser obtido em seu repositório no GitHub:

<https://github.com/obgm/libcoap>

Este trabalho utilizou como base a implementação do servidor disponibilizado pelo libcoap para o desenvolvimento da aplicação e condução dos experimentos. Para isto, o libcoap foi compilado em duas versões distintas: uma delas utilizando o *framework* OpenSSL e a outra utilizando o *framework* mbedTLS. A utilização de dois *frameworks* foi devido à incompatibilidade do OpenSSL com uma das *cipher suites* utilizadas nos experimentos (TLS_DHE_RSA_WITH_AES_256_GCM_SHA384, ver Tabela 5).

Detalhes de instalação, configuração e utilização do libcoap podem ser encontrados no Apêndice B deste trabalho.

4.4.2 Geração de certificados

Para a o processo de testes e validação do modelo, ambos os modos PSK (*Pre-Shared Key*) e PKI foram utilizados. No entanto, o modo PKI exige a utilização de certificados, tanto pela aplicação cliente quanto pelo servidor. Para tanto, houve a necessidade da geração de tais certificados.

Este trabalho utilizou um certificado autoassinado para servir como certificado de confiança e certificado do cliente e servidor, e também um par de chaves RSA de 2048 *bits*.

O certificado e o par de chaves foram gerados com o auxílio do *framework* OpenSSL por meio do seguinte comando:

```
$ openssl req -x509 -sha256 -nodes -newkey rsa:2048 -keyout keyfile.pem -out  
↪ certfile.pem
```

Os certificados utilizados pelas aplicações podem ser encontrados na Seção D.2, para a aplicação cliente, e na Seção E.2, para a aplicação do servidor, ambas disponíveis nos Apêndices deste trabalho.

4.4.3 Implementação da aplicação de coleta de dados

4.4.3.1 main.cpp

No início do arquivo temos a definição dos pinos que foram utilizados para conectar o sensor ao dispositivo (ver Figura 13).

A seguir, uma instância da classe `Adafruit_BMP280` é inicializada passando as definições dos pinos como parâmetros.

```
6 | #define BMP_SCK 18  
7 | #define BMP_MISO 19  
8 | #define BMP_MOSI 23  
9 | #define BMP_CS 5  
10 |  
11 | Adafruit_BMP280 bmp(BMP_CS, BMP_MOSI, BMP_MISO, BMP_SCK);
```

As linhas seguintes definem a estrutura das informações que serão passadas entre as aplicações (`struct` `Payload`).

Dentro da função `void setup` é verificado se o sistema está conseguindo se comunicar com o *hardware* do sensor BMP280. Se tudo estiver configurado corretamente, ele continuará a execução do código sem problemas. Caso contrário, ele ficará trancado no *loop*. Então, algumas configurações padrões são feitas no sensor.

```
25 | void setup()  
26 | {  
27 |     Serial.begin(115200);  
28 |     Serial.println(F("BMP280 test"));  
29 |  
30 |     if (!bmp.begin())
```

```

31     {
32         Serial.println(F("Could not find a valid BMP280 sensor, check wiring!"));
33         while (1);
34     }
35
36     /**
37      * Default settings from datasheet.
38      **/
39     bmp.setSampling(Adafruit_BMP280::MODE_NORMAL,      /* Operating Mode */
40                    Adafruit_BMP280::SAMPLING_X2,     /* Temp. oversampling */
41                    Adafruit_BMP280::SAMPLING_X16,    /* Pressure oversampling */
42                    Adafruit_BMP280::FILTER_X16,      /* Filtering. */
43                    Adafruit_BMP280::STANDBY_MS_500); /* Standby time. */
44 }

```

Por último, há a função `void loop`, que irá ler os dados do sensor e então escrevê-los na variável `payload`. A linha 53 irá escrever estes dados no barramento UART, e um `delay` de 5 segundos será aplicado.

```

46     void loop()
47     {
48         memcpy(payload.id, PAYLOAD_ID, sizeof(payload.id));
49         payload.temperature = bmp.readTemperature();
50         payload.pressure = bmp.readPressure();
51         payload.altitude = bmp.readAltitude();
52
53         Serial.write((const uint8_t *)&payload, sizeof(Payload));
54
55         delay(5000);
56     }

```

O código completo do arquivo `main.cpp` pode ser encontrado na Seção C.2, localizada nos Apêndices deste trabalho.

4.4.4 Implementação da aplicação cliente

A aplicação cliente foi desenvolvida tendo como base o exemplo `coap_client` disponibilizado pelo ESP-IDF, em que alguns ajustes foram efetuados para se encaixar no contexto deste trabalho.

A configuração inicial do projeto foi realizada de acordo com os passos descritos na Seção A.1. O projeto foi dividido em quatro componentes, cada um em um arquivo separado: `payload.h`, `coap.c`, `uart.c` e `main.c`.

4.4.4.1 `payload.h`

No arquivo `payload.h` temos a definição da estrutura dos dados que foi utilizada pelos módulos UART e CoAP. Esta estrutura foi definida da seguinte maneira:

```
8 | #define PAYLOAD_ID "PAYLOAD"
9 | typedef struct Payload {
10 |     char id[8];
11 |     float temperature;
12 |     float pressure;
13 |     float altitude;
14 | } Payload;
```

A linha 8 define o identificador da estrutura, `"PAYLOAD"`, utilizado para verificar se os dados recebidos estão na estrutura definida por `Payload`. O arquivo `payload.h` foi incluído em ambos os arquivos `coap.c` e `uart.c`.

O código completo do arquivo `payload.c` pode ser encontrado na Seção D.5, localizada nos Apêndices deste trabalho.

4.4.4.2 `coap.c`

O código presente no arquivo `coap.c` foi baseado em quase sua totalidade no exemplo `coap_client_example_main.c`, disponível no repositório do ESP-IDF, com exceção de algumas alterações que foram feitas para adaptar o cliente no contexto deste trabalho. Os respectivos ajustes serão detalhados a seguir.

No início do arquivo, na seção de variáveis globais, linha 77, uma variável do tipo `coap_string_t` foi incluída para armazenar os dados que são enviados ao servidor. `coap_string_t` é uma `struct` que possui dois atributos: `size_t length` e `uint8_t *s`, que armazenam o comprimento do *array* de *bytes* e um ponteiro para este *array*, respectivamente. Portanto, estas variáveis são inicializadas com tamanho igual à 0 e um ponteiro nulo.

```
77 | static coap_string_t payload = { 0, NULL };
```

A seguir temos a declaração do método `void coap_example_client`. Seu início foi alterado para levar em consideração os dados a serem enviados ao servidor. Ela recebe os dados lidos pelo módulo UART por meio do parâmetro `void *pvParameters`, enviado na criação da *thread* (mais detalhes na Seção 4.4.4.3).

Na linha 213 é inicializada a variável Payload `data`, e na linha 214 é realizada a cópia *byte a byte* dos dados recebidos em `*pvParameters` para a variável `data`. As linhas seguintes (216 à 232) têm como finalidade a formatação da `struct` Payload em uma *string* para enviá-la ao servidor.

```
211 static void coap_example_client(void *pvParameters)
212 {
213     Payload data;
214     memcpy(&data, pvParameters, sizeof(Payload));
215
216     size_t len = 0;
217     len = snprintf(
218         NULL, len,
219         "Temperature = %.2f °C\nPressure = %.2f Pa\nAltitude = %.2f m",
220         (double) data.temperature,
221         (double) data.pressure,
222         (double) data.altitude
223     );
224
225     char buffer[len+1];
226     snprintf(
227         buffer, len+1,
228         "Temperature = %.2f °C\nPressure = %.2f Pa\nAltitude = %.2f m",
229         (double) data.temperature,
230         (double) data.pressure,
231         (double) data.altitude
232     );
233     ...
```

Ainda no mesmo método, na linha 425, o método da requisição foi alterado de `COAP_REQUEST_GET` para `COAP_REQUEST_POST`, pois enviaremos uma mensagem *POST* para o servidor. Nas linhas 429 à 432, os dados da variável `buffer` e o seu comprimento são alocados na variável global `payload`, inicializada anteriormente. Na linha 433, o método `coap_add_data` é utilizado para inserir o `payload` na requisição. Para fins de visualização, imprimimos uma mensagem com os dados que foram enviados na requisição na linha 435.

```

421     ...
422     /* Set request options. */
423     request->type = COAP_MESSAGE_CON;
424     request->tid = coap_new_message_id(session);
425     request->code = COAP_REQUEST_POST;
426     coap_add_optlist_pdu(request, &optlist);
427
428     /* Set request payload. */
429     size_t length = sizeof(buffer);
430     payload.s = (unsigned char *) coap_malloc(length);
431     payload.length = length;
432     decode_segment((const uint8_t *) &buffer, length, payload.s);
433     coap_add_data(request, payload.length, payload.s);
434
435     printf("=== Sent ===\n%.*s\n\n", (int) length, buffer);
436     ...

```

Por fim, as chamadas de configuração utilizadas pelo exemplo são encapsuladas pelo método `void setup_coap`, que será chamado pelo método `start_listening` no arquivo `main.c` (ver Seção 4.4.4.4).

```

475     static void setup_coap()
476     {
477         ESP_ERROR_CHECK(nvs_flash_init());
478         ESP_ERROR_CHECK(esp_netif_init());
479         ESP_ERROR_CHECK(esp_event_loop_create_default());
480         ESP_ERROR_CHECK(example_connect());
481     }

```

De acordo com a documentação oficial do ESP-IDF, o método `nvs_flash_init()` tem como função inicializar a partição padrão NVS (*Non-Volatile Storage*), a qual foi projetada para armazenar pares de chave e valor na memória *flash* do ESP32 (Espressif Systems, c2020g). Em seguida, a chamada do método `esp_netif_init()` inicializa a pilha TCP/IP subjacente (Espressif Systems, c2020b).

Na sequência, ocorre a chamada do método `esp_event_loop_create_default()`. A biblioteca de *loop* de eventos permite que os componentes declarem eventos para os quais outros componentes podem registrar *handlers*, ou seja, funções que serão executadas quando estes eventos ocorrerem (Espressif Systems, c2020f).

Por fim, a chamada `example_connect()` vai realizar a configuração da conexão Wi-Fi e Ethernet, utilizando os valores especificados no menu de configuração do projeto (ver Seção A.2).

Todas as funções anteriores são executadas como parâmetro de uma outra função chamada `ESP_ERROR_CHECK`. Esta função, na verdade, é uma macro que funciona como um *assert* presente em outras linguagens, exceto pelo fato de que ela verifica um valor do tipo `esp_err_t` em vez de uma condição booleana. Se seu argumento não for igual à `ESP_OK`, então uma mensagem de erro é impressa no *console* e o método `abort()` é chamado (Espressif Systems, c2020e).

O código completo do arquivo `coap.c` pode ser encontrado na Seção D.6, localizada nos Apêndices deste trabalho.

4.4.4.3 `uart.c`

As funções responsáveis por lidar com a comunicação do barramento UART foram definidas no arquivo `uart.c`.

A definição dos pinos utilizados pelo UART é feita pelo menu de configuração do projeto (ver Seção A.2), e seus valores são atribuídos automaticamente às definições iniciais deste arquivo no processo de construção da aplicação.

O método `setup_uart` realiza a configuração inicial do barramento UART no dispositivo, configurando os pinos, *baudrate*, e alguns outros *bits* de configuração, que podem ser vistos no trecho de código a abaixo. Em seguida, algumas funções da biblioteca `uart.h` são executadas para instalar o *driver* da UART no dispositivo e realizar a configuração dos parâmetros e pinos.

```
33  static void setup_uart() {
34      /* Configure parameters of an UART driver,
35       * communication pins and install the driver */
36      uart_config_t uart_config = {
37          .baud_rate = ECHO_UART_BAUD_RATE,
38          .data_bits = UART_DATA_8_BITS,
39          .parity     = UART_PARITY_DISABLE,
40          .stop_bits = UART_STOP_BITS_1,
41          .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
42          .source_clk = UART_SCLK_APB,
43      };
44      int intr_alloc_flags = 0;
```

```

45
46 #if CONFIG_UART_ISR_IN_IRAM
47     intr_alloc_flags = ESP_INTR_FLAG_IRAM;
48 #endif
49
50     ESP_ERROR_CHECK(
51         uart_driver_install(
52             ECHO_UART_PORT_NUM,
53             BUF_SIZE * 2, 0, 0,
54             NULL, intr_alloc_flags
55         )
56     );
57     ESP_ERROR_CHECK(
58         uart_param_config(ECHO_UART_PORT_NUM, &uart_config)
59     );
60     ESP_ERROR_CHECK(
61         uart_set_pin(
62             ECHO_UART_PORT_NUM, ECHO_TEST_TXD,
63             ECHO_TEST_RXD, ECHO_TEST_RTS,
64             ECHO_TEST_CTS
65         )
66     );
67 }

```

A próxima função, `void listen_uart`, é encarregada de realizar o monitoramento contínuo do barramento UART em busca de dados. Quando houver novos dados, e se eles estiverem na estrutura definida por `Payload`, então eles são direcionados ao módulo CoAP para serem enviados ao servidor.

```

68 static void listen_uart(void *arg, void (*callback)(void *)) {
69     Payload payload;
70     while (1) {
71         int len = uart_read_bytes(
72             ECHO_UART_PORT_NUM,
73             &payload,
74             sizeof(Payload),
75             20 / portTICK_RATE_MS
76         );
77
78         if (memcmp(&payload.id, PAYLOAD_ID, sizeof(payload.id)) == 0
79             && len == sizeof(Payload)) {
80             xTaskCreate(callback, "task", 8*1024, &payload, 5, NULL);

```

```

81     }
82 }
83 }

```

Esta função possui dois parâmetros, sendo o primeiro deles, `void *arg`, os argumentos recebidos quando a *thread* atual foi criada, e o segundo parâmetro, `void (*callback) (void *)`, é um ponteiro para uma função de *callback*. Esta função de *callback* é passada pelo método `void start_listening`, definida no arquivo `main.c`, e neste caso é a função para executar uma instância de cliente CoAP. No entanto, para garantir o desacoplamento, este módulo não possui conhecimento desta função.

Na linha 71 é realizada uma tentativa de leitura no barramento UART utilizando a função `uart_read_bytes`. A quantidade de *bytes* lidos é inserida na variável `int len`. Na linha 78 é feita uma verificação para validar se os dados lidos correspondem com a estrutura definida anteriormente, `struct Payload`. Esta verificação é feita comparando o `payload.id` com a string `"PAYLOAD"`, definida por `#define PAYLOAD_ID`. Se o resultado for igual a 0, então ambos são iguais e uma nova *thread* é criada. A criação desta *thread* é realizada chamando o método `xTaskCreate`, passando como parâmetros a função de *callback*, um nome para a *thread*, a quantidade de palavras a serem alocadas para a sua pilha, os parâmetros iniciais, sua prioridade em relação à outras *threads* e uma função *handle*, que neste caso não houve necessidade de se utilizar.

Logo após a criação desta *thread*, o módulo CoAP irá se encarregar de enviar os dados passados por parâmetro para o servidor. O método `void listen_uart` continuará sua execução por meio do laço *while*, até que alguma interrupção cancele a execução da aplicação.

O código completo do arquivo `uart.c` pode ser encontrado na Seção D.7, localizada nos Apêndices deste trabalho.

4.4.4.4 main.c

O arquivo `main.c` contém as funções que serão executadas no início da aplicação.

O método `void start_listening` realiza as chamadas de configuração de ambos os módulos CoAP e UART, e então dá início ao *polling* no barramento UART por meio da função `listen_uart`, apresentada anteriormente. Caso algum erro ocorra durante a sua execução, a chamada `vTaskDelete(NULL)`, na linha 8, finaliza a *thread* da aplicação.

O método `void app_main` é o ponto de entrada da aplicação, iniciando a configuração de ambos os módulos e o processo de escuta no barramento UART, passando a função `coap_example_client` como *callback* a ser chamado sempre que novos dados forem lidos do barramento.

```
1  #include "uart.c"
2  #include "coap.c"
3
4  static void start_listening(void *arg) {
5      setup_coap();
6      setup_uart();
7      listen_uart(arg, coap_example_client);
8  }
9
10 void app_main(void) {
11     start_listening(NULL);
12 }
```

O código completo do arquivo `main.c` pode ser encontrado na Seção D.8, localizada nos Apêndices deste trabalho.

4.4.4.5 Kconfig.projbuild

O `Kconfig.projbuild` adiciona as opções de configuração para a conexão (Wi-Fi ou Ethernet), as configurações do módulo CoAP e as configurações do módulo UART, utilizadas ao longo deste trabalho. Para habilitar estas opções, basta adicionar este arquivo no mesmo diretório que os demais arquivos deste projeto, de acordo com a estrutura especificada na Seção D.1.

O arquivo `Kconfig.projbuild` pode ser encontrado na Seção D.9, localizada nos Apêndices deste trabalho.

4.4.4.6 Certificados

Os certificados utilizados no projeto cliente foram gerados de acordo com os passos apresentados na Seção 4.4.2 e foram separados em três arquivos – `coap_ca.pem`, `coap_client.crt` e `coap_client.key` – representando o certificado de confiança, o certificado do cliente e a chave privada, respectivamente.

É importante ressaltar que o certificado do cliente é autoassinado, sendo considerado suficiente para ser utilizado nos testes. O ideal seria envolver uma assinatura de autoridade certificadora juridicamente válida, mas o propósito neste trabalho não foi ter custos com este certificado. Logo, os certificados `coap_ca.pem` e `coap_client.crt` são idênticos. Para efeitos de organização, estes arquivos foram armazenados no diretório `certs`, na pasta raiz do projeto.

Os arquivos de certificados utilizados pelo cliente podem ser encontrados na Seção D.2, localizada nos Apêndices deste trabalho.

4.4.5 Implementação da aplicação do servidor

A aplicação do servidor foi desenvolvida utilizando o exemplo `coap-server` como base, disponibilizado pela biblioteca `libcoap`. Este exemplo, contudo, não oferecia um recurso associado à um método *POST*, necessário para o contexto deste trabalho. Para tanto, um novo recurso `/sensor` foi criado, adicionando-se uma função *handler* à ele. Isto significa que, ao receber uma mensagem *POST* proveniente de um cliente autenticado, esta função *handler* será executada.

A configuração inicial deste projeto foi realizada de acordo com os passos descritos na Seção B.1. No entanto, uma vez que as alterações descritas nesta seção foram adicionadas ao arquivo `coap-server`, localizado em `libcoap/examples`, houve a necessidade da execução do comando `make` no diretório raiz do projeto. Este comando faz a compilação de todos os arquivos que sofreram modificações desde a última vez que o comando foi utilizado.

4.4.5.1 `coap-server.c`

No arquivo `coap-server.c`, localizado no diretório `examples`, foi adicionado uma nova função chamada `void hnd_post_sensor`, definindo um *handler* para o recurso `/sensor`, sendo associado ao método *POST*. Esta função recebe alguns parâmetros referentes à requisição CoAP e tem como responsabilidade o tratamento de requisições *POST* para o recurso `/sensor`.

Para isto, ela realiza a leitura do *payload* da requisição por meio da função `coap_get_data`, na linha 245, copiando os dados para a variável `data`. Logo após, na linha 249, ele atribui à requisição o código `COAP_RESPONSE_CODE(200)`, representando um OK e indicando ao cliente que a requisição *POST* foi realizada com sucesso. Ela então adiciona a requisição e mais algumas configurações nas opções do protocolo, por meio da

função `coap_add_option` e, por fim, na linha 259, utiliza a função `coap_add_data` para adicionar os dados lidos à resposta que será enviada ao cliente.

```
234     static void hnd_post_sensor(coap_context_t *ctx UNUSED_PARAM,
235                               struct coap_resource_t *resource,
236                               coap_session_t *session UNUSED_PARAM,
237                               coap_pdu_t *request,
238                               coap_binary_t *token UNUSED_PARAM,
239                               coap_string_t *query UNUSED_PARAM,
240                               coap_pdu_t *response)
241     {
242         size_t size;
243         unsigned char *data;
244
245         (void) coap_get_data(request, &size, &data);
246
247         if (size > 0) {
248             unsigned char buf[3];
249             response->code = COAP_RESPONSE_CODE(200);
250             coap_add_option(
251                 response,
252                 COAP_OPTION_CONTENT_FORMAT,
253                 coap_encode_var_safe(
254                     buf, sizeof(buf),
255                     COAP_MEDIATYPE_TEXT_PLAIN
256                 ),
257                 buf
258             );
259             coap_add_data(response, size, (const uint8_t *) data);
260         }
261     }
```

Mais adiante, neste mesmo arquivo, há uma função chamada `void init_resources`. Esta função é responsável por inicializar os recursos do servidor CoAP. Com o objetivo de inicializar o recurso `/sensor`, as seguintes linhas foram inseridas dentro desta função:

```
973     /* define sensor resource */
974     r = coap_resource_init(coap_make_str_const("sensor"), resource_flags);
975     coap_register_handler(r, COAP_REQUEST_POST, hnd_post_sensor);
976     coap_add_resource(ctx, r);
```

A chamada à função `coap_resouse_init`, na linha 974, inicializa um recurso chamado "`sensor`", atribuindo algumas *flags* globais que são utilizadas pelo recurso. Estas *flags* foram definidas previamente pelo `libcoap`. Uma vez inicializado, este recurso é atribuído à variável `r`, que é um ponteiro para uma `struct coap_resource_t`, e é reutilizado para cada recurso adicionado ao servidor. Na linha 975, a função `hnd_post_sensor` é associada ao recurso `r` como *handler*, tendo como método de requisição o método *POST*. Finalmente, na linha 976, o recurso `r` é vinculado ao servidor CoAP.

Com exceção das alterações mostradas acima, o restante deste arquivo manteve-se inalterado. Demais arquivos de código e configurações deste projeto não sofreram alterações.

O código completo do arquivo `coap-server.c` pode ser encontrado na Seção E.3, localizada nos Apêndices deste trabalho.

4.5 Testes experimentais da aplicação

Para realizar a validação da aplicação, alguns testes iniciais foram realizados para confirmar o seu funcionamento. Para isto, foram feitos testes de comunicação entre o cliente e o servidor, verificando também os dados lidos pelo sensor.

O servidor CoAP foi executado, passando os certificados, com o seguinte comando:

```
$ ./coap-server -A 192.168.25.10 -c certs/certfile.pem -j certs/keyfile.pem -v9
```

Em que `192.168.25.10` é o endereço IP do *notebook*, onde o servidor está executando. O cliente deverá buscar por este IP na hora de enviar mensagens ao servidor. Além disso, a opção `-v9` foi adicionada ao comando para visualizarmos mensagens adicionais sobre o processamento do servidor.

Para configurar o endereço do servidor no cliente, foi necessário entrar no diretório raiz do projeto `coap-client`, e abrir o menu de configuração com o seguinte comando:

```
$ idf.py menuconfig
```

Ao acessar a opção **Example CoAP Client Configuration**, um novo menu é aberto. Nele, foi configurado o endereço `coaps://192.168.25.10:5684/sensor`. O resultado pode ser visto na Figura 17.

Figura 17 – Configuração do endereço do servidor CoAP

```
(Top) → Example CoAP Client Configuration
Espressif IoT Development Framework Configuration
(coaps://192.168.25.10:5684/sensor) Target Uri

[Space/Enter] Toggle/enter  [ESC] Leave menu          [S] Save
[O] Load                  [?] Symbol info          [/] Jump to symbol
[F] Toggle show-help mode  [C] Toggle show-name mode [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```

Fonte: Elaborado pelo autor (2020)

Além disso, também foi necessário configurar a *cipher suite* que seria utilizada na validação. Para isto, em **Component config** → **mbedTLS** → **TLS Key Exchange Methods**, habilitou-se somente a opção **Enable RSA-only based ciphersuite modes**, como pode ser visto na Figura 18. *Cipher suites* do modo PSK devem estar habilitadas, mesmo que estejamos usando PKI.

Após este processo de configuração, a aplicação foi gravada na memória *flash* do ESP32.

Figura 18 – Configuração de *cipher suites*

```
(Top) → Component config → mbedTLS → TLS Key Exchange Methods
Espressif IoT Development Framework Configuration
[*] Enable pre-shared-key ciphersuites
[*] Enable PSK based ciphersuite modes
[*] Enable DHE-PSK based ciphersuite modes
[*] Enable ECDHE-PSK based ciphersuite modes
[*] Enable RSA-PSK based ciphersuite modes
[*] Enable RSA-only based ciphersuite modes
[ ] Enable DHE-RSA based ciphersuite modes
[ ] Support Elliptic Curve based ciphersuites

[Space/Enter] Toggle/enter  [ESC] Leave menu          [S] Save
[O] Load                  [?] Symbol info          [/] Jump to symbol
[F] Toggle show-help mode  [C] Toggle show-name mode [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```

Fonte: Elaborado pelo autor (2020)

A monitoração do dispositivo cliente foi feita por meio do seguinte comando:

```
$ idf.py -p /dev/ttyUSB0 monitor
```

Ao energizar o dispositivo responsável pelos sensores, a seguinte mensagem foi exibida na saída do monitor do cliente:

```

1 | I (45768) CoAP_client: DNS lookup succeeded. IP=192.168.25.10
2 | === Sent ===
3 | Temperature = 28.44 °C
4 | Pressure = 100225.64 Pa
5 | Altitude = 91.93 m

```

A linha 1 indica que o cliente obteve sucesso ao buscar o IP do servidor, e as linhas seguintes apresentam os dados lidos pelo sensor e, na sequência, enviados ao servidor. Os dados compreendem amostras de temperatura, pressão barométrica e altitude coletados em uma região próxima ao nível do mar.

No terminal onde o servidor foi executado, obteve-se as seguintes mensagens:

```

1 | Oct 31 17:18:54.737 DEBG created UDP endpoint 192.168.25.10:5683
2 | Oct 31 17:18:54.737 DEBG created DTLS endpoint 192.168.25.10:5684
3 | Oct 31 17:18:54.737 DEBG created TCP endpoint 192.168.25.10:5683
4 | Oct 31 17:18:54.737 DEBG created TLS endpoint 192.168.25.10:5684
5 | Oct 31 17:19:05.121 DEBG (if3) DTLS: new incoming session
6 | Oct 31 17:19:05.121 DEBG (if3) DTLS: received 149 bytes
7 | Oct 31 17:19:05.121 DEBG (if3) DTLS: sent 60 bytes
8 | Oct 31 17:19:05.137 DEBG (if3) DTLS: received 181 bytes
9 | Oct 31 17:19:05.137 DEBG (if3) DTLS: SSL_accept:before SSL initialization
10 | Oct 31 17:19:05.137 DEBG (if3) DTLS: SSL_accept:before SSL initialization
11 | Oct 31 17:19:05.178 DEBG (if3) DTLS: Using PKI ciphers
12 | Oct 31 17:19:05.178 DEBG (if3) DTLS: SSL_accept:SSLv3/TLS read client hello
13 | Oct 31 17:19:05.178 DEBG (if3) DTLS: SSL_accept:SSLv3/TLS write server hello
14 | Oct 31 17:19:05.178 DEBG (if3) DTLS: SSL_accept:SSLv3/TLS write certificate
15 | Oct 31 17:19:05.178 DEBG (if3) DTLS: sent 1013 bytes
16 | Oct 31 17:19:05.178 DEBG (if3) DTLS: SSL_accept:SSLv3/TLS write server done
17 | Oct 31 17:19:05.247 DEBG (if3) DTLS: received 358 bytes
18 | Oct 31 17:19:05.247 DEBG (if3) DTLS: SSL_accept:SSLv3/TLS write server done
19 | Oct 31 17:19:05.252 DEBG (if3) DTLS: SSL_accept:SSLv3/TLS read client key exchange
20 | Oct 31 17:19:05.252 DEBG (if3) DTLS: SSL_accept:SSLv3/TLS read change cipher spec
21 | Oct 31 17:19:05.252 DEBG (if3) DTLS: SSL_accept:SSLv3/TLS read finished
22 | Oct 31 17:19:05.252 DEBG (if3) DTLS: SSL_accept:SSLv3/TLS write session ticket
23 | Oct 31 17:19:05.252 DEBG (if3) DTLS: SSL_accept:SSLv3/TLS write change cipher spec
24 | Oct 31 17:19:05.252 DEBG (if3) DTLS: sent 266 bytes
25 | Oct 31 17:19:05.253 DEBG (if3) DTLS: SSL_accept:SSLv3/TLS write finished
26 | Oct 31 17:19:05.253 CIPH (if3) DTLS: Using cipher: AES256-GCM-SHA384
27 | Oct 31 17:19:05.253 DEBG ***EVENT: 0x01de
28 | Oct 31 17:19:05.253 DEBG (if3) DTLS: session connected
29 | Oct 31 17:19:05.267 DEBG (if3) DTLS: received 132 bytes

```

```

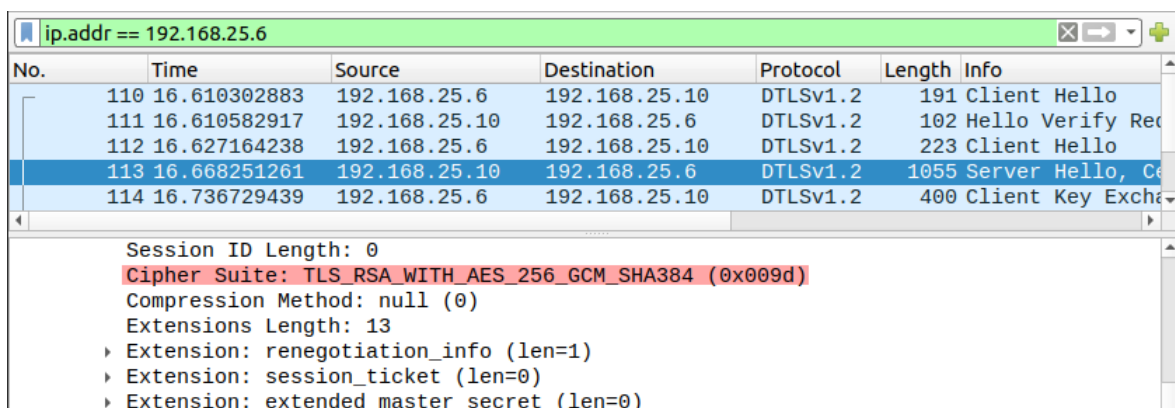
30 v:1 t:CON c:POST i:2d01 {} [ Uri-Path:sensor ] :: 'Temperature = 24.42
    ↳ \xC2\xBC\x0APressure = 102107.95 Pa\x0AAltitude = -64.98 m\x00'
31 Oct 31 17:19:05.267 DEBG call custom handler for resource 'sensor'
32 Oct 31 17:19:05.267 DEBG (if3) DTLS: sent 126 bytes
33 v:1 t:ACK c:2.01 i:2d01 {} [ Content-Format:text/plain ] :: 'Temperature = 24.42
    ↳ \xC2\xBC\x0APressure = 102107.95 Pa\x0AAltitude = -64.98 m\x00'

```

As linhas iniciais mostram que o servidor criou um *endpoint* para o protocolo DTLS no endereço 192.168.25.10 e porta 5684. A partir da linha 5, obteve-se informações da chegada de uma requisição proveniente do cliente, com o intuito de abertura de uma sessão DTLS. As linhas seguintes descrevem o *handshake* do DTLS.

A *cipher suite* utilizada na comunicação é indicada pela linha 26, AES256-GCM-SHA384, sendo este o nome utilizado pelo OpenSSL para a *cipher suite* `TLS_RSA_WITH_AES_256_GCM_SHA384` (Rudolph; Grundmann, c2019), que foi a *cipher suite* configurada no cliente. Para obtermos uma confirmação, os pacotes desta comunicação foram capturados e analisados pelo Wireshark. A Figura 19 mostra detalhes do pacote *Server Hello*, contendo a *cipher suite* acordada entre as partes.

Figura 19 – Pacote contendo a *cipher suite* utilizada na comunicação (Wireshark)



Fonte: Elaborado pelo autor (2020)

Após a linha 28, a sessão foi estabelecida e ambas as partes deram início a troca de dados, que pode ser visto nas linhas 30 e 33. A linha 31 informa que o recurso `/sensor` obteve um acesso e então executou a sua função *handler*.

De volta ao monitor do cliente, obtivemos a mensagem de resposta do servidor:

```

=== Received ===
Temperature = 28.44 °C
Pressure = 100225.64 Pa
Altitude = 91.93 m

```

Com este teste foi possível constatar que ambos os dispositivos estão conseguindo se comunicar, realizando a transmissão dos dados através de um meio de comunicação seguro. Outros testes foram realizados para as demais *cipher suites* do modo PKI e do modo PSK, mas estes testes não serão apresentados neste trabalho, pois seguiram o mesmo procedimento do teste acima, e resultados similares foram obtidos em todos os casos.

4.6 Testes experimentais e mensurações quantitativas

Esta seção teve como objetivo a realização de experimentos na aplicação desenvolvida neste trabalho, resultando em mensurações relativas ao tempo de execução total de cada *cipher suite* utilizada, mostrando também os tempos de execução do cliente e do servidor.

Além disso, experimentos adicionais foram feitos no lado do servidor a fim de analisar os tempos de processamento com dois dispositivos de capacidades diferentes: um *notebook Acer Aspire E15* e uma placa integrada *Raspberry Pi 3*. Detalhes sobre as especificações de cada um destes dispositivos são apresentados na Tabela 4.

Tabela 4 – Especificações dos dispositivos utilizados nos experimentos no lado do servidor

	Processador	Núcleos	Frequência	Arquitetura	RAM
<i>Notebook</i>	Intel Core i7-5500U	<i>Quad Core</i>	2.4 GHz	64 bits	8 GB
Raspberry Pi 3	Broadcom BCM2837	<i>Quad Core</i>	1.2 GHz	64 bits	1 GB

Fonte: Elaborado pelo autor (2020)

4.6.1 Descrição das *cipher suites*

As *cipher suites* utilizadas nos experimentos desta seção são listadas na Tabela 5.

Tabela 5 – *Cipher suites* analisadas

<i>Cipher suite</i>	Algoritmo
TLS_RSA_WITH_AES_256_GCM_SHA384	RSA
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	DHE
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	ECDHE
TLS_PSK_WITH_AES_256_GCM_SHA384	PSK

Fonte: Elaborado pelo autor (2020)

A primeira *cipher suite* analisada neste trabalho, **TLS_RSA_WITH_AES_256_GCM_SHA384**, utiliza o algoritmo RSA para realizar a troca de chaves, processo conhecido como *key exchange*. A autenticação também é feita utilizando o algoritmo RSA. Esta *cipher suite* utiliza o algoritmo AES para a criptografia simétrica, fazendo uso de chaves de 256 *bits* com o modo *Galois/Counter* (AES256 GCM). Além disso, as operações de *hash* são feitas com o algoritmo SHA384. Esta *cipher suite* foi incluída na RFC 5288 (Salowe; Choudhury; McGrew, 2008).

A segunda *cipher suite* da lista, **TLS_DHE_RSA_WITH_AES_256_GCM_SHA384**, faz uso do algoritmo DHE (*Diffie-Hellman Ephemeral*) para a troca de chaves, ao passo que a autenticação é performada pelo algoritmo RSA. Similar às outras *cipher suites*, o algoritmo AES 256 GCM é adotado para a criptografia simétrica e o SHA384 para a geração de *hashes* (Salowe; Choudhury; McGrew, 2008).

A terceira *cipher suite* da lista, **TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384**, utiliza o algoritmo ECDHE (*Elliptic Curve Diffie-Hellman Ephemeral*) para a troca de chaves. Este algoritmo trata-se de um protocolo para acordo de chaves que permite que dois participantes, cada um munido de um par de chaves pública/privada de curvas elípticas, possam estabelecer uma chave compartilhada secreta por meio de um canal inseguro. Este algoritmo é uma variante do algoritmo DHE, fazendo uso de chaves de curvas elípticas (Stallings, 2013). Além disso, os participantes podem gerar novos pares de chaves pública/privada para cada execução do algoritmo, também chamado de valores efêmeros (Herzog; Khazan, 2011). A autenticação é feita com o algoritmo RSA, enquanto que os processos de criptografia simétrica e *hash* são feitos pelos algoritmos AES256 GCM e SHA384, respectivamente. Esta *cipher suite* foi incluída na RFC 5289 (Rescorla, 2008).

Por fim, temos a *cipher suite* **TLS_PSK_WITH_AES_256_GCM_SHA384**. Esta *cipher suite* é utilizada no modo PSK, onde a chave é previamente compartilhada entre os participantes por meio de um canal seguro. O processo de troca de chaves neste modo trata-se de um acordo entre os participantes sobre qual chave, dentre um conjunto de chaves previamente compartilhadas, será utilizada na comunicação, visto que ambos os participantes podem ter chaves compartilhadas com outras entidades (Eronen; Tschofenig, 2005). Sistemas que utilizam este modo dependem de uma ou mais chaves para garantir a confidencialidade. Chaves suficientemente longas, escolhidas randomicamente, podem resistir a praticamente qualquer ataque de força bruta. Uma desvantagem deste modo é que ele não dispõe de *Perfect Forward Secrecy* (PFC), ou seja, se a chave secreta compartilhada for comprometida, um atacante poderá decifrar mensagens antigas que fizeram uso desta chave (Eronen; Tschofenig, 2005). Um exemplo de uso do PSK está em métodos

criptográficos utilizados em redes Wi-Fi, tais como WEP (*Wired Equivalent Privacy*) (Arash; Danesh; Samadi, 2009), WPA (*Wi-Fi Protected Access*), também chamado de WPA-PSK ou WPA2-PSK (Bersani; Tschofenig, 2007), e EAP (*Extensible Authentication Protocol*), também conhecido como EAP-PSK (Simon; Aboba; Eronen, 2008). Esta *cipher suite* foi incluída na RFC 5487 (Badra, 2009).

A próxima seção apresentará resultados quantitativos da utilização destas *cipher suites* na aplicação desenvolvida neste trabalho, considerando tempos de processamento obtidos nos lados do cliente e servidor.

4.6.2 Resultados referentes ao tempo de processamento

Os resultados apresentados nesta seção foram obtidos a partir da aplicação desenvolvida neste trabalho, calculando o tempo de processamento das mensagens trocadas de ambos os lados, alternando as *cipher suites* utilizadas.

As variáveis mensuradas consistem nos tempos de processamento do cliente e do servidor, tempos de rede e tempos totais, em que cada registro equivale à uma mensagem trocada entre os participantes. Os tempos de processamento do cliente e servidor e os tempos totais foram calculados a partir de arquivos de *logs* gerados pelas aplicações, enquanto que o tempo de rede foi obtido subtraindo os tempos de processamento do tempo total.

Para cada *cipher suite* foram coletadas aproximadamente 1000 amostras, onde registros *outliers* foram excluídos do conjunto de dados.

O objetivo desta seção foi apresentar uma comparação quantitativa da utilização de diferentes *cipher suites* em um ambiente típico IoT, utilizando um dispositivo restrito como cliente e dispositivos mais robustos atuando como servidores. Para destacar o impacto de cada *cipher suite* no lado do servidor, testes adicionais foram realizados utilizando dois dispositivos de diferentes capacidades atuando como servidor: uma Raspberry Pi 3 e um *notebook*.

As tabelas a seguir apresentam os dados referentes às *cipher suites* listadas na Tabela 5, em que o *notebook* atuou como servidor. Cada *cipher suite* será referenciada pelo seu algoritmo de *key exchange*. Estas tabelas apresentam dados estatísticos sobre as amostras coletadas, onde constam a média dos tempos de processamento, desvios-padrão dos valores e pontos de mínimo e máximo.

A seguir são apresentadas as tabelas com os resultados obtidos com a Raspberry

Tabela 6 – RSA (*notebook*): Tempos de execução (ms)

	Cliente	Servidor	Rede	Total
Média	19,49	119,54	16,85	155,89
Desvio padrão	0,53	50,68	32,82	66,57
Mínimo	18	82	5	82
Máximo	21	426	480	675

Fonte: Elaborado pelo autor (2020)

Tabela 7 – DHE (*notebook*): Tempos de execução (ms)

	Cliente	Servidor	Rede	Total
Média	19,91	487,83	11,21	518,41
Desvio padrão	0,42	21,41	13,36	27,79
Mínimo	19	455	5	480
Máximo	21	648	310	876

Fonte: Elaborado pelo autor (2020)

Tabela 8 – ECDHE (*notebook*): Tempos de execução (ms)

	Cliente	Servidor	Rede	Total
Média	20,37	540,96	1855,87	2417,22
Desvio padrão	0,83	285,75	716,54	588,16
Mínimo	19	3	784	898
Máximo	23	999	4008	4034

Fonte: Elaborado pelo autor (2020)

Tabela 9 – PSK (*notebook*): Tempos de execução (ms)

	Cliente	Servidor	Rede	Total
Média	11,21	71,57	16,13	98,92
Desvio padrão	0,47	56,22	25,41	71,41
Mínimo	10	40	5	57
Máximo	13	382	267	532

Fonte: Elaborado pelo autor (2020)

Pi 3 atuando como servidor. Demais componentes do ambiente permaneceram os mesmos ao longo dos dois experimentos.

A Figura 20 apresenta os tempos médios de processamento de cada *cipher suite* calculados no lado do cliente. Como pode ser observado, a *cipher suite* PSK apresentou o menor tempo médio de processamento dentre as demais *cipher suites* analisadas, atingindo

Tabela 10 – RSA (Raspberry Pi 3): Tempos de execução (ms)

	Cliente	Servidor	Rede	Total
Média	19,8	143,26	17,57	180,64
Desvio padrão	0,51	42,71	28,98	57,78
Mínimo	18	107	5	135
Máximo	21	407	524	947

Fonte: Elaborado pelo autor (2020)

Tabela 11 – DHE (Raspberry Pi 3): Tempos de execução (ms)

	Cliente	Servidor	Rede	Total
Média	20,64	459,57	905,4	1385,62
Desvio padrão	0,85	287,95	316,59	272,42
Mínimo	18	1	5	937
Máximo	23	999	1306	937

Fonte: Elaborado pelo autor (2020)

Tabela 12 – ECDHE (Raspberry Pi 3): Tempos de execução (ms)

	Cliente	Servidor	Rede	Total
Média	20,68	477,11	2289,38	2787,18
Desvio padrão	0,89	303,45	675,94	583,19
Mínimo	19	3	1003	1723
Máximo	23	999	4286	4610

Fonte: Elaborado pelo autor (2020)

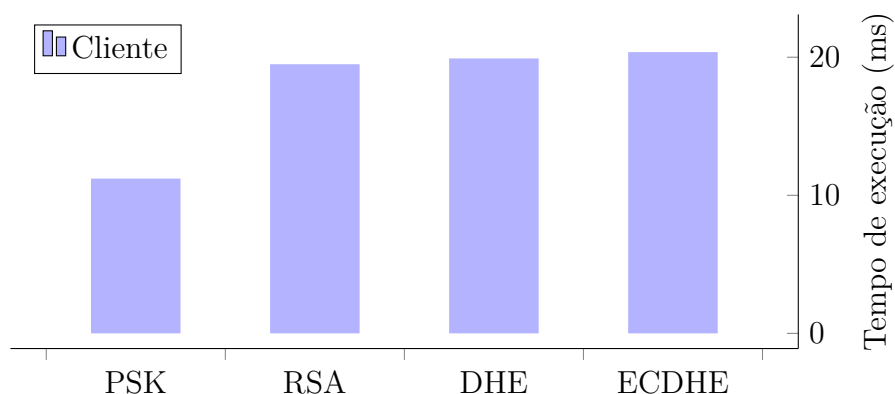
Tabela 13 – PSK (Raspberry Pi 3): Tempos de execução (ms)

	Cliente	Servidor	Rede	Total
Média	11,17	100,71	67,33	179,23
Desvio padrão	0,49	66,38	206,48	219,22
Mínimo	10	8	6	73
Máximo	13	444	1143	1475

Fonte: Elaborado pelo autor (2020)

praticamente a metade do tempo necessário por seus pares. Este fato é em parte explicado pelo uso de chaves simples para garantir a confidencialidade da comunicação, não sendo necessária a utilização de certificados para este propósito. As demais *cipher suites* não mostraram diferenças significativas entre seus tempos de processamento, permanecendo no patamar dos 20 ms médios por execução.

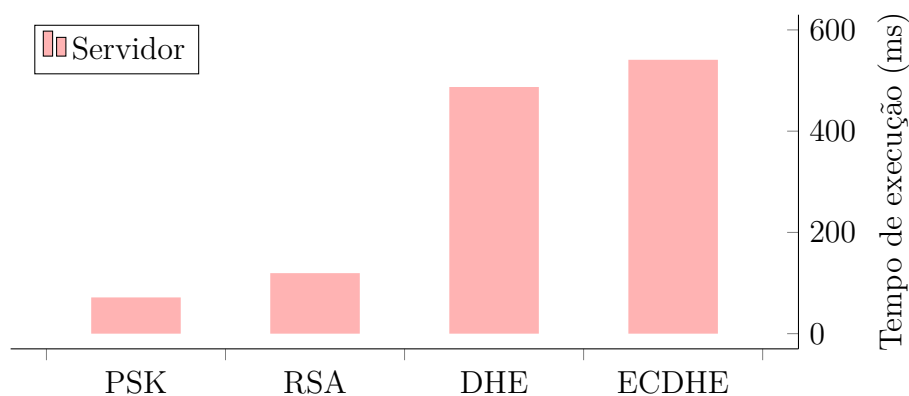
Figura 20 – Tempo médio de processamento de diferentes *cipher suites* no cliente



Fonte: Elaborado pelo autor (2020)

A Figura 21 apresenta os tempos médios de processamento das *cipher suites* no lado do servidor, utilizando o *notebook* como recurso computacional. Neste gráfico é possível observar que, assim como no lado do cliente, a utilização de PSK é a mais veloz dentre todas as outras *cipher suites*. Logo após vem o RSA, com tempos levemente maiores, exibindo um aumento de 40% em relação ao PSK. Por fim, temos os algoritmos DHE e ECDHE, com tempos de execução significativamente maiores, atingindo médias de 487,83 ms e 540,96 ms, respectivamente.

Figura 21 – Tempo médio de processamento de diferentes *cipher suites* no lado do servidor

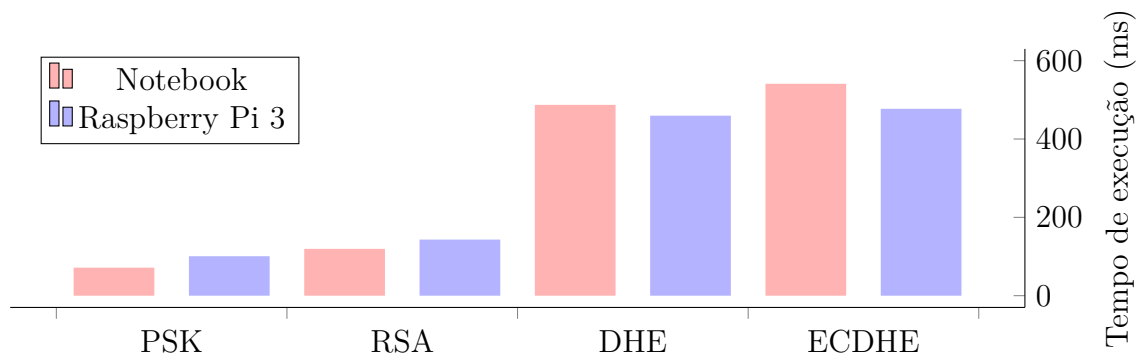


Fonte: Elaborado pelo autor (2020)

Os tempos médios de execução do lado do servidor, considerando sua execução em dois dispositivos com diferentes especificações (ver Tabela 4), é exibida na Figura 22. De modo geral, os tempos de processamento não mostraram diferenças significativas entre os

dois dispositivos. No entanto, a Raspberry obteve médias maiores para o PSK e o RSA, apresentando aumentos de 40,72% e 19,84%, respectivamente, em relação ao *notebook*. No caso dos algoritmos DHE e ECDHE o cenário foi o inverso, em que os tempos médios da Raspberry sofreram reduções de 5,68% e 11,80%, respectivamente, em relação aos tempos obtidos no *notebook* para ambos os algoritmos.

Figura 22 – Tempo médio de processamento de diferentes *cipher suites* no servidor



Fonte: Elaborado pelo autor (2020)

4.6.3 Resultados referentes ao consumo energético

Para mensurar o consumo de energia, experimentos foram realizados utilizando as *cipher suites* apresentadas nesta seção. Para isto, uma bateria com capacidade nominal de 3200 mAh foi utilizada como fonte de alimentação da aplicação. Os experimentos compreenderam a execução do processo de *handshake* do DTLS seguido de uma troca de mensagens CoAP até a drenagem completa da bateria.

O código-fonte da aplicação cliente foi levemente modificado, removendo o processo de escuta ativa no barramento UART com o objetivo de obter maior precisão na mensuração do consumo energético de cada *cipher suite*. Estas alterações removeram a espera entre os envios, de modo que o cliente possa iniciar um novo envio assim que o servidor envia a resposta da mensagem anterior.

A duração de cada teste (tempo decorrido entre o início do experimento até a drenagem total da bateria) e número de execuções (estabelecimento de uma sessão DTLS seguida do envio de uma mensagem) foram obtidos através de arquivos de *logs* gerados pela aplicação do servidor. O consumo médio por execução (mA) foi obtido dividindo-se a capacidade nominal da bateria pelo número de execuções. O consumo médio por hora

(mAh) foi obtido dividindo-se a capacidade nominal da bateria pela duração de cada experimento (em horas). Os resultados destes experimentos são apresentados na Tabela 14.

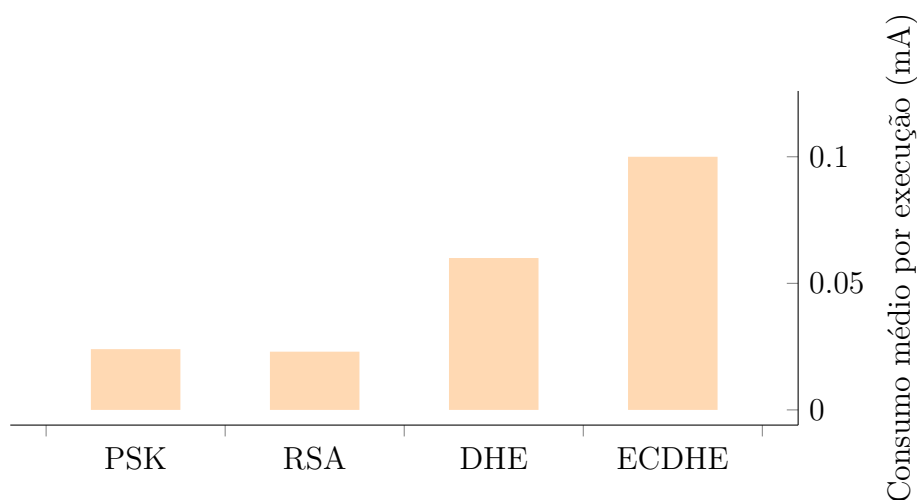
Tabela 14 – Consumo energético de diferentes *cipher suites*

<i>Cipher suite</i>	Duração do experimento	Número de execuções	Consumo médio por execução (mA)	Consumo médio por hora (mAh)
RSA	22h 59min 39s	141099	0,023	139,17
DHE	27h 04min 11s	53450	0,060	118,21
ECDHE	30h 21min 22s	31942	0,100	112,85
PSK	21h 07min 01s	135313	0,024	151,54

Fonte: Elaborado pelo autor (2020)

A Figura 23 apresenta um gráfico com os dados de consumo médio por execução de cada um dos algoritmos analisados.

Figura 23 – Consumo médio por execução (mA) de diferentes *cipher suites*



Fonte: Elaborado pelo autor (2020)

É importante destacar que, devido ao método utilizado para realizar a medição, os valores podem apresentar um certo grau de imprecisão e servem apenas para fins de comparação entre as *cipher suites*. Ainda assim, os resultados observados foram influenciados pelo consumo geral do dispositivo, possivelmente envolvendo outros módulos que consumiram parte da energia ao longo dos experimentos.

Os resultados indicaram que os algoritmos PSK e RSA para o processo de troca de chaves são os mais econômicos, consumindo aproximadamente 0,02 mA por execução. Por

outro lado, os algoritmos DHE e ECDHE apresentaram um maior consumo energético, obtendo valores de 0,055 mA e 0,1 mA por execução, resultados significativamente maiores que o de seus pares.

5 CONCLUSÃO

Este trabalho realizou pesquisas teóricas sobre dois dos principais protocolos utilizados em ambientes IoT, CoAP e MQTT, trazendo uma análise quantitativa acerca do desempenho do protocolo CoAP em ambientes reais, através de experimentações e trabalhos do estado da arte. Com base nestas pesquisas, ficou evidente a necessidade de protocolos mais leves para utilização em ambientes restritos, reduzindo *overhead* de processamento e consumo de energia. Além disso, estes protocolos devem ser utilizados em conjunto com protocolos de segurança, como o TLS, amplamente utilizado hoje em dia. No entanto, devido ao seu maior *overhead*, por utilizar pacotes TCP, o TLS não é adaptável a ambientes IoT, surgindo a necessidade de protocolos como o DTLS, que utilizam meios mais eficientes para a entrega de pacotes, como o UDP.

Uma aplicação IoT foi desenvolvida neste trabalho, utilizando implementações dos protocolos CoAP e DTLS em dispositivos ESP32. Esta aplicação teve como objetivo a validação de um ambiente utilizando estes dois protocolos no contexto de telemetria veicular para processamento de dados em tempo real. Testes foram realizados nesta aplicação para mensurar os tempos de processamento e o consumo energético de diferentes *cipher suites*, incluindo *cipher suites* que utilizam certificados (PKI) e *cipher suites* de senhas pré-compartilhadas (PSK). Os resultados destes experimentos mostraram que *cipher suites* que usam PSK são significativamente mais velozes e mais econômicas do que *cipher suites* que utilizam PKI.

Este trabalho forneceu detalhes de implementação das bibliotecas utilizadas pela aplicação, no intuito de facilitar a busca por informações de futuros trabalhos que foquem no desenvolvimento deste dois protocolos.

Trabalhos futuros podem realizar mensurações quantitativas do MQTT, ou até mesmo de variações mais eficientes que estão sendo propostas, como MQTT-SN, que faz uso do UDP em vez do TCP, e também do CoAP com DTLS e 6LoWPAN, para ganhos de eficiência.

REFERÊNCIAS

ARASH, Lashkari; DANESH, Mir; SAMADI, Behrang. A survey on wireless security protocols (wep, wpa and wpa2/802.11i). In: **2009 2nd IEEE International Conference on Computer Science and Information Technology**. [S.l.: s.n.], 2009. p. 48–52.

BADRA, Mohamad. **Pre-Shared Key Cipher Suites for TLS with SHA-256/384 and AES Galois Counter Mode**. 2009. Disponível em: <<https://tools.ietf.org/html/rfc5487>>. Acesso em: 29 de out. de 2020.

BERSANI, Florent; TSCHOFENIG, Hannes. **The EAP-PSK Protocol: A Pre-Shared Key Extensible Authentication Protocol (EAP) Method**. 2007. Disponível em: <<https://tools.ietf.org/html/rfc4764>>. Acesso em: 01 de nov. de 2020.

Bosch Sensortec. **Pressure Sensor BMP280**. c2020. Disponível em: <<https://www.bosch-sensortec.com/products/environmental-sensors/pressure-sensors/pressure-sensors-bmp280-1.html>>. Acesso em: 25 de nov. de 2020.

DIZDAREVIĆ, Jasenka; CARPIO, Francisco; JUKAN, Admela; MASIP-BRUIN, Xavi. A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration. **ACM Comput. Surv.**, Association for Computing Machinery, New York, NY, USA, v. 51, n. 6, jan. 2019. ISSN 0360-0300. Disponível em: <<https://doi.org/10.1145/3292674>>.

Eclipse Foundation. **Eclipse Californium™**. c2020. Disponível em: <<https://www.eclipse.org/californium/>>. Acesso em: 18 de out. de 2020.

ENDLER, Markus; SILVA, Anderson; CRUZ, Rafael A. M. S. An approach for secure edge computing in the internet of things. In: **2017 1st Cyber Security in Networking Conference (CSNet)**. [S.l.: s.n.], 2017. p. 1–8.

ERONEN, Pasi; TSCHOFENIG, Hannes. **Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)**. 2005. Disponível em: <<https://tools.ietf.org/html/rfc4279>>. Acesso em: 29 de out. de 2020.

Espressif Systems. **ESP-IDF Programming Guide**. c2020. Disponível em: <<https://www.espressif.com/en/products/socs/esp32>>. Acesso em: 25 de out. de 2020.

Espressif Systems. **ESP-NETIF**. c2020. Disponível em: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/esp_netif.html#esp-netif>. Acesso em: 26 de out. de 2020.

Espressif Systems. **ESP32**. c2020. Disponível em: <<https://docs.espressif.com/projects/esp-idf/en/latest/esp32>>. Acesso em: 25 de out. de 2020.

Espressif Systems. **ESP32-WROOM-32E ESP32-WROOM-32UE - Datasheet**. c2020. Disponível em: <https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf>. Acesso em: 01 de nov. de 2020.

Espressif Systems. **ESP_ERROR_CHECK**. c2020. Disponível em: <<https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/error-handling.html#esp-error-check-macro>>. Acesso em: 26 de out. de 2020.

Espressif Systems. **Event Loop Library**. c2020. Disponível em: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/esp_event.html#event-loop-library>. Acesso em: 26 de out. de 2020.

Espressif Systems. **NVS**. c2020. Disponível em: <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/nvs_flash.html#non-volatile-storage-library>. Acesso em: 26 de out. de 2020.

Espressif Systems. **Project Configuration**. c2020. Disponível em: <<https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/kconfig.html>>. Acesso em: 27 de out. de 2020.

FELTRIN, Lucas; TSOUKANERI, Galini; CONDOLUCI, Massimo; BURATTI, Chiara; MAHMOODI, Toktam; DOHLER, Mischa; VERDONE, Roberto. Narrowband iot: A survey on downlink and uplink perspectives. **IEEE Wireless Communications**, v. 26, n. 1, p. 78–86, 2019.

GLAROUDIS, Dimitrios; IOSSIFIDES, Athanasios; CHATZIMISIOS, Periklis. Survey, comparison and research challenges of iot application protocols for smart farming. **Computer Networks**, v. 168, p. 107037, 2020. ISSN 1389-1286. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S1389128619306942>>.

Grafana Labs. **Grafana**. c2020. Disponível em: <<https://grafana.com/>>. Acesso em: 02 de nov. de 2020.

HARDY, Alex; BERNICK, Jessalyn; CAPDEVILA, Nick; PERRY, Tristan. Electronic cvt - controls final design report. 2019. Disponível em: <<https://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1563context=mesp>>. Acesso em: 03 de nov. de 2020.

HARRINGTON, Timothy; BOWMAN, Douglas; JOHNSON, Walter; BENNER, Richard; CAPENER, Ronald; HAN, Huong. **Vehicle tag used for transmitting vehicle telemetry data**. [S.l.]: Google Patents, dez. 9 2004. US Patent App. 10/855,871.

HASSANALIERAGH, Moeen; PAGE, Alex; SOYATA, Tolga; SHARMA, Gaurav; AKTAS, Mehmet; MATEOS, Gonzalo; KANTARCI, Burak; ANDREESCU, Silvana. Health monitoring and management using internet-of-things (iot) sensing with cloud-based processing: Opportunities and challenges. In: **2015 IEEE International Conference on Services Computing**. [S.l.: s.n.], 2015. p. 285–292.

HERZOG, Jonathan; KHAZAN, Roger. **Use of Static-Static Elliptic Curve Diffie-Hellman Key Agreement in Cryptographic Message Syntax**. 2011. Disponível em: <<https://tools.ietf.org/html/rfc6278>>. Acesso em: 29 de out. de 2020.

IHS Markit. **The Internet of Things: a movement, not a market**. 2017. Disponível em: <https://cdn.ihs.com/www/pdf/IoT_ebook.pdf>. Acesso em: 26 de mar. de 2019.

KAEEWONGSRI, Kriddikorn; SILANON, Kittasil. Design and implement of a weather monitoring station using coap on nb-iot network. In: **2020 17th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)**. [S.l.: s.n.], 2020. p. 230–233.

Kaspersky Lab. **Kaspersky Lab detectou mais de 7.000 amostras de malware em dispositivos IoT desde o começo do ano**. 2017. Disponível em: <https://www.kaspersky.com.br/about/press-releases/2017_kaspersky-lab-detected-over-7000-samples-of-malware-devices-iot-since-the-start-of-year>. Acesso em: 26 de mar. de 2019.

KIM, Hokeun; LEE, Edward A. Authentication and authorization for the internet of things. **IT Professional**, v. 19, n. 5, p. 27–33, 2017. ISSN 1520-9202.

KOTHMAYR, Thomas; SCHMITT, Corinna; HU, Wen; BRÜNIG, Michael; CARLE, Georg. Dtls based security and two-way authentication for the internet of things. **Ad Hoc Networks**, v. 11, n. 8, p. 2710 – 2723, 2013. ISSN 1570-8705.

LARA, Evangelina; AGUILAR, Leocundo; SANCHEZ, Mauricio A.; GARCÍA, Jesús A. Lightweight authentication protocol for m2m communications of resource-constrained devices in industrial internet of things. **Sensors** **2020**, p. 468–473, 2020.

LI, Nan; LIU, Dongxi; NEPAL, Surya. Lightweight mutual authentication for iot and its applications. **IEEE Transactions on Sustainable Computing**, v. 2, n. 4, p. 359–370, 2017.

libcoap. **libcoap**. c2020. Disponível em: <<https://libcoap.net/>>. Acesso em: 25 de out. de 2020.

LIU, Jin; XIAO, Yang; CHEN, C.L. Philip. Internet of things' authentication and access control. **International Journal of Security and Networks**, v. 7, n. 4, p. 228–241, 2012. Disponível em: <<https://www.inderscienceonline.com/doi/abs/10.1504/IJSN.2012.053461>>.

Mosquitto. **MQTT man page**. c2020. Disponível em: <<https://mosquitto.org/man/mqtt-7.html>>. Acesso em: 05 de nov. de 2020.

NAIK, Nitin. Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. In: **2017 IEEE International Systems Engineering Symposium (ISSE)**. [S.l.: s.n.], 2017. p. 1–7.

NEETHIRAJAN, Suresh. Recent advances in wearable sensors for animal health management. **Sensing and Bio-Sensing Research**, v. 12, p. 15 – 29, 2017. ISSN 2214-1804.

Quatro Rodas. **Novo câmbio CVT garante economia e, finalmente, desempenho**. 2018. Disponível em: <<https://quatorrodas.abril.com.br/auto-servico/novo-cambio-cvt-garante-economia-e-finalmente-desempenho/>>. Acesso em: 03 de nov. de 2020.

RAHMAN, Reem Abdul; SHAH, Babar. Security analysis of iot protocols: A focus in coap. In: **2016 3rd MEC International Conference on Big Data and Smart City (ICBDSC)**. [S.l.: s.n.], 2016. p. 1–7.

Raspberry Pi Foundation. **Raspberry Pi 3 Model B**. c2020. Disponível em: <<https://www.raspberrypi.org/products/raspberry-pi-3-model-b>>. Acesso em: 04 de nov. de 2020.

RAY, Partha Pratim. A survey of iot cloud platforms. **Future Computing and Informatics Journal**, v. 1, n. 1, p. 35 – 46, 2016. ISSN 2314-7288. Disponível em: <<http://www.sciencedirect.com/science/article/pii/S2314728816300149>>.

RESCORLA, Eric. **TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM)**. 2008. Disponível em: <<https://tools.ietf.org/html/rfc5289>>. Acesso em: 29 de out. de 2020.

RESCORLA, Eric; MODADUGU, Nagendra. **Datagram Transport Layer Security**. 2006. Disponível em: <<https://tools.ietf.org/html/rfc4347>>. Acesso em: 03 de nov. de 2020.

RESCORLA, Eric; MODADUGU, Nagendra. **Datagram Transport Layer Security Version 1.2**. 2012. Disponível em: <<https://tools.ietf.org/html/rfc6347>>. Acesso em: 01 de nov. de 2020.

ROEPKE, Rene; THRAEM, Timo; WAGENER, Johannes; WIESMAIER, Alex. A survey on protocols securing the internet of things: Dtls, ipsec and ieee 802.11 i. 2016.

RUDOLPH, Hans Christian; GRUNDMANN, Nils. **Ciphersuite: TLS_RSA_WITH_AES_256_GCM_SHA384**. c2019. Disponível em: <https://ciphersuite.info/cs/TLS_RSA_WITH_AES_256_GCM_SHA384/>. Acesso em: 31 de out. de 2020.

SABRI, Challouf; KRIAA, Lobna; AZZOUZ, Saidane Leila. Comparison of iot constrained devices operating systems: A survey. In: **2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)**. [S.l.: s.n.], 2017. p. 369–375.

SAE Brasil. **Baja Nacional**. c2020. Disponível em: <<https://saebrasil.org.br/programas-estudantis/baja-sae-brasil/>>. Acesso em: 01 de nov. de 2020.

SAE International. **Baja SAE**. c2020. Disponível em: <<https://www.sae.org/attend/student-events/>>. Acesso em: 01 de nov. de 2020.

SALOWEY, Joseph; CHOUDHURY, Abhijit; MCGREW, David. **AES Galois Counter Mode (GCM) Cipher Suites for TLS**. 2008. Disponível em: <<https://tools.ietf.org/html/rfc5288>>. Acesso em: 29 de out. de 2020.

SANO, Alex. Uma análise da eficiência de uma transmissão cvt. 2013. Disponível em: <<https://repositorio.unesp.br/handle/11449/120947?locale-attribute=en>>. Acesso em: 03 de nov. de 2020.

SHELBY, Zach; HARTKE, Klaus; BORMANN, Carsten. **The Constrained Application Protocol (CoAP)**. 2014. Disponível em: <<https://tools.ietf.org/html/rfc7252>>. Acesso em: 01 de nov. de 2020.

SILVA, Daniel Alves da; TORRES, José Alberto Sousa; PINHEIRO, Alexandre; FILHO, Francisco L. de Caldas; MENDONÇA, Fabio L. L.; PRACIANO, Bruno J. G; KFOURI, Guilherme de Oliveira; SOUSA, Rafael T. de. Inference of driver behavior using correlated iot data from the vehicle telemetry and the driver mobile phone. In: **2019 Federated Conference on Computer Science and Information Systems (FedCSIS)**. [S.l.: s.n.], 2019. p. 487–491.

SIMON, Daniel; ABOBA, Bernard D.; ERONEN, Pasi. **Extensible Authentication Protocol (EAP) Key Management Framework**. 2008. Disponível em: <<https://tools.ietf.org/html/rfc5247>>. Acesso em: 01 de nov. de 2020.

STALLINGS, William. **Cryptography and Network Security: Principles and Practice**. 6th. ed. USA: Prentice Hall Press, 2013. ISBN 0133354695.

TANGANELLI, Giacomo; VALLATI, Carlo; MINGOZZI, Enzo. Coapthon: Easy development of coap-based iot applications with python. In: **2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)**. [S.l.: s.n.], 2015. p. 63–68.

THANGAVEL, Dinesh; MA, Xiaoping; VALERA, Alvin; TAN, Hwee-Xian; TAN, Colin Keng-Yan. Performance evaluation of mqtt and coap via a common middleware. In: **2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)**. [S.l.: s.n.], 2014. p. 1–6.

TIBURSKI, Ramão Tiago; MATOS, Everton de; HESSEL, Fabiano. Evaluating the dtls protocol from coap in fog-to-fog communications. In: **2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)**. [S.l.: s.n.], 2019. p. 90–905.

WESTPHALL, Johann; LOFFI, Leandro; WESTPHALL, Carla Merkle; MARTINA, Jean Everson. Coap + dtls: A comprehensive overview of cryptographic performance on an iot scenario. In: **2020 IEEE Sensors Applications Symposium (SAS)**. [S.l.: s.n.], 2020. p. 1–6.

ÇAVUŞOĞLU, Ünal; EBLEME, Mehmet; BAYILMIŞ, Cüneyt; KUCUK, Kerem. Coap and its performance evaluation. **Sakarya University Journal of Science**, p. 78–85, 02 2020.

Apêndices

A ESP-IDF

A.1 Instalação

Para a utilização desta aplicação exemplo, foi necessária a instalação da *toolchain* do ESP-IDF, a qual é uma ferramenta que auxilia no processo de configuração, gravação e monitoramento do dispositivo. Para isto, os seguintes passos foram executados em um terminal Linux:

Código-fonte

O código-fonte da *toolchain* está disponibilizada em seu repositório no GitHub, e foi obtido por meio dos seguintes comandos:

```
$ mkdir -p ~/esp  
$ cd ~/esp  
$ git clone --recursive https://github.com/espressif/esp-idf.git
```

Pré-requisitos

Para a utilização do ESP-IDF, é necessária a instalação de alguns pacotes que são pré-requisitos:

```
$ sudo apt-get install git wget flex bison gperf python python-pip python-setuptools  
↪ cmake ninja-build ccache libffi-dev libssl-dev dfu-util
```

Configurações pré-instalação

Algumas distribuições Linux podem exibir a mensagem de erro `Failed to open port /dev/ttyUSB0` ao fazer o *upload* da aplicação para o ESP32. Este problema ocorre porque o usuário atual do sistema não possui permissões para utilizar as portas USB através do ESP-IDF. Isto pode ser resolvido adicionando o usuário ao grupo `dialout`.

```
$ sudo usermod -a -G dialout $USER
```


Além disso, o ESP-IDF utiliza a versão Python 3 como interpretador. Algumas distribuições Ubuntu e Debian ainda configuram a versão Python 2.7 como interpretador padrão do sistema. Para adicionar a versão Python 3 como interpretador padrão, basta executar o seguinte comando:

```
$ sudo update-alternatives --install /usr/bin/python python /usr/bin/python3 10
```

Instalação

```
$ cd ~/esp/esp-idf
$ ./install.sh
```

Configuração das variáveis de ambiente

Para obtermos acesso ao arquivo executável do ESP-IDF, foi necessário configurar sua variável de ambiente. Para isso, ele disponibiliza um *script* que realiza este processo automaticamente.

```
$ . $HOME/esp/esp-idf/export.sh
```

Cópia do projeto exemplo

Neste ponto, copiamos o projeto exemplo `coap_client`, disponibilizado pelo ESP-IDF, para um diretório local.

```
$ cd ~/esp
$ cp -r $IDF_PATH/examples/protocols/coap_client .
```

Configuração do projeto exemplo

Por meio dos comandos abaixo, configuramos alguns atributos do projeto. Um menu interativo foi aberto no próprio terminal, permitindo a configuração de diversos atributos pertinentes ao projeto e à configuração do próprio dispositivo ESP32. A página inicial deste menu pode ser visto na ??.

```
$ cd ~/esp/coap_client
$ idf.py set-target esp32
$ idf.py menuconfig
```

Acessando a opção `Example CoAP Client Configuration`, obtemos algumas configurações específicas para o protocolo CoAP, como pode ser visto na Figura 26. Neste submenu

podemos alterar a URI (*Uniform Resource Identifier*) de destino do CoAP, assim como as informações de usuário e senha caso o modo PSK (*Public Shared Key*) esteja ativado.

Construção da aplicação

Para iniciar o processo de *build* da aplicação, utilizou-se o seguinte comando:

```
| $ idf.py build
```

Gravação da aplicação no dispositivo

Com a aplicação compilada, realizou-se o processo de *upload* da aplicação para a memória *flash* do dispositivo. Para isso, o seguinte comando foi utilizado, informando-se a porta USB do dispositivo (`/dev/ttyUSB0`, por exemplo), e o *baudrate* (taxa de transmissão), o qual é um atributo opcional.

```
| $ idf.py -p PORT [-b BAUD] flash
```

Um exemplo deste comando utilizando todas as opções é mostrado a seguir:

```
| $ idf.py -p /dev/ttyUSB0 -b 115200 flash
```

Monitoramento da saída do dispositivo

Por fim, para habilitar o monitoramento da porta serial do dispositivo, o seguinte comando foi executado, novamente, informando a porta onde o dispositivo se encontrava:

```
| $ idf.py -p PORT monitor
```

Um exemplo deste comando utilizando todas as opções é dado a seguir:

```
| $ idf.py -p /dev/ttyUSB0 monitor
```

Ao término destes passos, a comunicação serial entre o dispositivo e o terminal foi estabelecida, e as saídas da aplicação puderam ser visualizadas diretamente pelo terminal.

Esta sequência de passos foi obtida por meio da documentação do ESP-IDF. Demais informações podem ser encontradas em <https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started>.

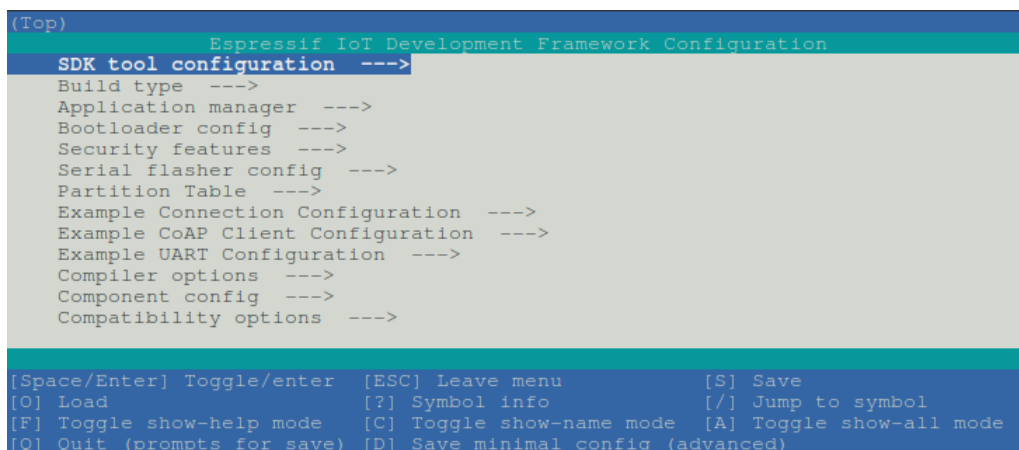
A.2 Menu de configuração

O ESP-IDF oferece um método fácil de criar configurações personalizadas através do `kconfiglib`, extensão do sistema `Kconfig` baseada em Python, que provê mecanismos de configuração de projeto em tempo de compilação. Estes arquivos são baseados em opções de tipos primitivos, tais como `integer`, `string` e `boolean`. Também é possível especificar dependências entre as opções, além de valores *default*, agrupamentos, entre outros (Espressif Systems, c2020h).

Para ter acesso à este menu, o ESP-IDF fornece o comando abaixo, e o resultado de sua execução pode ser observado na Figura 24.

```
$ idf.py menuconfig
```

Figura 24 – Menu de configuração do projeto exemplo `coap_client`



```
(Top)
Espressif IoT Development Framework Configuration
SDK tool configuration --->
Build type --->
Application manager --->
Bootloader config --->
Security features --->
Serial flasher config --->
Partition Table --->
Example Connection Configuration --->
Example CoAP Client Configuration --->
Example UART Configuration --->
Compiler options --->
Component config --->
Compatibility options --->

[Space/Enter] Toggle/enter  [ESC] Leave menu          [S] Save
[O] Load                    [?] Symbol info          [/] Jump to symbol
[F] Toggle show-help mode  [C] Toggle show-name mode [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```

Fonte: Elaborado pelo autor (2020)

A criação de novas opções é possível por meio do arquivo `Kconfig.projbuild`. Ao atualizar o menu de configuração pela primeira vez os valores são armazenados no arquivo `sdkconfig`.

Nas subseções seguintes serão apresentadas as configurações pertinentes à este trabalho. Todas estas opções estão disponíveis no menu inicial e devem ser obrigatoriamente configuradas.

A.2.1 Configuração do módulo Wi-Fi

Na opção **Example Connection Configuration** podemos configurar os atributos da conexão de rede do dispositivo ESP32. Neste submenu podemos escolher entre as

interfaces de rede Wi-Fi ou Ethernet, assim como configurar o SSID e a senha caso a interface escolhida seja o Wi-Fi. Mais detalhes podem ser vistos na Figura 25.

Figura 25 – Submenu para configurações de rede

```
(Top) → Example Connection Configuration
      Espressif IoT Development Framework Configuration
[*] connect using WiFi interface
(myssid) WiFi SSID
(mypassword) WiFi Password
[ ] connect using Ethernet interface
[*] Obtain IPv6 address
      Preferred IPv6 Type (Local Link Address) --->

[Space/Enter] Toggle/enter  [ESC] Leave menu          [S] Save
[O] Load                  [?] Symbol info          [/] Jump to symbol
[F] Toggle show-help mode  [C] Toggle show-name mode [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```

Fonte: Elaborado pelo autor (2020)

A.2.2 Configuração do módulo CoAP

O exemplo `coap_client` traz opções adicionais para a configuração de atributos relacionados ao protocolo CoAP, que podem ser acessados por meio da opção **Example CoAP Client Configuration**. O resultado pode ser visto na Figura 26.

Figura 26 – Submenu para configurações do protocolo CoAP

```
(Top) → Example CoAP Client Configuration
      Espressif IoT Development Framework Configuration
(coaps://californium.eclipse.org) Target Uri
(sesame) Preshared Key (PSK) to used in the connection to the CoAP server
(password) PSK Client identity (username)

[Space/Enter] Toggle/enter  [ESC] Leave menu          [S] Save
[O] Load                  [?] Symbol info          [/] Jump to symbol
[F] Toggle show-help mode  [C] Toggle show-name mode [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```

Fonte: Elaborado pelo autor (2020)

Neste submenu devemos obrigatoriamente configurar o endereço onde o servidor CoAP está localizado. Caso opte-se por uma comunicação segura, é necessário utilizar o prefixo `coaps` no lugar de `coap`, assim como informar o usuário e senha caso o método de segurança utilizado seja o PSK.

Para a utilização do CoAP com certificados (PKI), é necessário marcar a opção **PKI Certificates**. Para isso, basta voltar ao menu inicial e adentrar as opções **Component config** → **CoAP Configuration** → **CoAP Encryption method**.

O submenu para a escolha entre os modos PKS e PKI do CoAP é exibido na Figura 27.

Figura 27 – Submenu para configurações do modo de segurança do CoAP

```
(Top) → Component config → CoAP Configuration → CoAP Encryption method
Espressif IoT Development Framework Configuration
( ) Pre-Shared Keys
(X) PKI Certificates

[Space/Enter] Toggle/enter  [ESC] Leave menu          [S] Save
[O] Load                    [?] Symbol info          [/] Jump to symbol
[F] Toggle show-help mode   [C] Toggle show-name mode [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```

Fonte: Elaborado pelo autor (2020)

A.2.3 Configuração do módulo UART

Além disso, o este trabalho adicionou novas opções para a configuração dos parâmetros do módulo UART, as quais podem ser acessadas por meio da opção **Example UART Configuration**. Estas opções são apresentadas na Figura 28. Estas opções representam, respectivamente, o número da porta UART utilizada (este trabalho utilizou o pino RX0 para a leitura, então esta opção recebe o valor 0, ver Figura 13), o *baudrate* da comunicação, os pinos TX e RX utilizados (ver Figura 13), e o tamanho do *buffer* utilizado para armazenar os dados da UART.

Figura 28 – Submenu para configurações do módulo UART

```
(Top) → Example UART Configuration
      Espressif IoT Development Framework Configuration
(0) UART port number
(115200) UART communication speed
(3) UART RXD pin number
(4) UART TXD pin number
(2048) UART echo example task stack size

[Space/Enter] Toggle/enter  [ESC] Leave menu          [S] Save
[O] Load                [?] Symbol info          [/] Jump to symbol
[F] Toggle show-help mode  [C] Toggle show-name mode [A] Toggle show-all mode
[Q] Quit (prompts for save) [D] Save minimal config (advanced)
```

Fonte: Elaborado pelo autor (2020)

B libcoap

B.1 Instalação

Para realizar o *download* de seu código-fonte e configurá-lo, foram utilizados os passos abaixo:

```
$ git clone https://github.com/obgm/libcoap --recursive
$ ./autogen.sh
$ ./configure --enable-dtls --with-openssl --disable-doxygen --disable-manpages
$ make
$ cd examples
```

B.2 Utilização

No diretório `examples`, dois arquivos executáveis foram gerados: `coap-client` e `coap-server`. Podemos executar um servidor com o seguinte comando:

```
$ ./coap-server -A 127.0.0.1 -k password
```

A opção `-A` recebe o endereço e a opção `-k` recebe a senha configurada para o modo PSK. Um outro modo de criação de servidor seria passando certificados para ele, o que pode ser feito por meio do seguinte comando:

```
$ ./coap-server -A 127.0.0.1 -k password -C ca_cert.pem -c certfile.pem -j
↔ keyfile.pem
```

Nesta segunda situação, a opção `-C` recebe o certificado assinado por uma autoridade certificadora, CA (*Certificate Authority*), a opção `-c` recebe o certificado do servidor e a opção `-k` recebe o arquivo com a chave privada do servidor. A opção `-C` torna-se desnecessária caso o certificado do servidor seja autoassinado. Para configurar o servidor em ambos os modos, PSK e PKI, basta configurar a chave pré-compartilhada em conjunto com os certificados, como mostrado no comando acima.

Para enviarmos uma mensagem para o servidor utilizando uma chave pré-compartilhada, utilizamos o seguinte comando:

```
| $ ./coap-client coaps://127.0.0.1 -k password -u user
```

Como dito anteriormente, a opção `-k` recebe a senha pré-compartilhada, e a opção `-u` recebe o nome do usuário a ser autenticado. O comando acima envia uma mensagem *GET* para o servidor, pois é o método padrão do libcoap caso nenhuma outra opção de método seja definida. Para enviarmos um *POST*, podemos fazer:

```
| $ ./coap-client coaps://127.0.0.1 -k password -m post -e hello
```

Em que a opção `-m` recebe o nome do método e a opção `-e` recebe o seu *payload*, ou seja, o texto a ser enviado.

O envio de mensagens utilizando PKI são realizadas de maneira similar ao comando apresentado para o caso do servidor, sendo necessário fornecer as opções `-C`, `-c` e `-k` para configurar os certificados.

C Aplicação 1: uart-sender

C.1 Estrutura do projeto

```
uart-sender/  
  src/  
    main.cpp
```

C.2 main.cpp

```
1  #include <Arduino.h>  
2  #include <Adafruit_BMP280.h>  
3  #include <Wire.h>  
4  #include <SPI.h>  
5  
6  #define BMP_SCK 18  
7  #define BMP_MISO 19  
8  #define BMP_MOSI 23  
9  #define BMP_CS 5  
10  
11  Adafruit_BMP280 bmp(BMP_CS, BMP_MOSI, BMP_MISO, BMP_SCK);  
12  
13  typedef struct Payload  
14  {  
15      char id[8];  
16      float temperature;  
17      float pressure;  
18      float altitude;  
19  } Payload;  
20  
21  #define PAYLOAD_ID "PAYLOAD"  
22  
23  Payload payload;  
24
```

```

25 void setup()
26 {
27     Serial.begin(115200);
28     Serial.println(F("BMP280 test"));
29
30     if (!bmp.begin())
31     {
32         Serial.println(F("Could not find a valid BMP280 sensor, check wiring!"));
33         while (1)
34             ;
35     }
36
37     /**
38      * Default settings from datasheet.
39      */
40     bmp.setSampling(Adafruit_BMP280::MODE_NORMAL, /* Operating Mode */
41                    Adafruit_BMP280::SAMPLING_X2, /* Temp. oversampling */
42                    Adafruit_BMP280::SAMPLING_X16, /* Pressure oversampling */
43                    Adafruit_BMP280::FILTER_X16, /* Filtering. */
44                    Adafruit_BMP280::STANDBY_MS_500); /* Standby time. */
45 }
46
47 void loop()
48 {
49     memcpy(payload.id, PAYLOAD_ID, sizeof(payload.id));
50     payload.temperature = bmp.readTemperature();
51     payload.pressure = bmp.readPressure();
52     payload.altitude = bmp.readAltitude();
53
54     Serial.write((const uint8_t *)&payload, sizeof(Payload));
55
56     delay(10000);
57 }

```

D Aplicação 2: coap-client

D.1 Estrutura do projeto

```
coap-client/  
  main/  
    certs/  
      coap_ca.pem  
      coap_client.crt  
      coap_client.key  
    coap.c  
    Kconfig.projbuild  
    main.c  
    payload.h  
    uart.c  
    main.cpp
```

D.2 Certificados

D.2.1 coap_ca.pem

```
-----BEGIN CERTIFICATE-----  
MIIDazCCA10gAwIBAgIUeCIA1sxlL1b7PfbQ6k/GwodV0eQwDQYJKoZIhvcNAQEL  
BQAwRTElMAkGA1UEBhMCQVUxEzARBgNVBAGMClNvbWUtU3RhdGUxITAfBgNVBAoM  
GEludGVybmV0IFdpZGdpdHMgUHR5IEExOZDAeFw0yMDEwMDEwMDEwMDEwMDEw  
MjYxOTM1NDRAeUxkZGdpdHMgUHR5IEExOZDAeFw0yMDEwMDEwMDEwMDEwMDEw  
HwYdVQKDBhJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGQwggEiMA0GCSqGSIb3DQEB  
AQUAA4IBDwAwggEKAoIBAQR4CSDyEkjkfAyoIO9wFd8z8MjsRaCJkhPqwFB52d  
/k9dBC1IM0yY79ETf9tpayP2VTGdq29yjOPe8TKi/deFPG9RJCtw5c1sZebdzg/  
7N8WQ31vg0s8H4hSjQB7bKfQTL3cC3DNQPW3ZCeHPGFtoCr5Gn2byiNoIt9SB+Nd  
FzS/OGrGFDWehk4mZW4+PmLzdV1wtEv1GmqQ5QKeLDDesk7uGgYnWUycf+qgSWlM  
NHgJCB493brQndMfmu/1eXXvLTE19nfkuYPBW78gHOYb8stjc/loAyMmqr6vao+  
k0vWE+MKr7tWOKQF85r3Q5dHg/a2CiUMy1idMX9PkR3vAgMBAAGjUzBRMBOGA1Ud  
DgQWBBS9xY9cgtftUHfsGw2h1Nqz8ShLvTafBgNVHSMEGDAWgBS9xY9cgtftUHfs
```

```
Gw2h1Nqz8ShLvTAPBgNVHRMBAf8EBTADAQH/MAOGCSqGSIb3DQEBCwUAA4IBAQDW
6Hw2Sqe5ZEVYQ4h9IpW2Ky4uqJsdZrsMMRdICf4/t0Ya0q54topul/OioBIt05U
00A/yUGRg9Ybmzcc2elz1L6spo0bQ7jfn40k5P7/9VSotWzcXv3e0vxSn7ZsibhB
Cszo04pUtzzPrnvkAr/vzuj6rrGvDKv/HGylBjf3JNFiarDsXenwgiOpNVPRpaBq
cAz+aB/OCmINQ4pc5fNTRW7FcgGu0AdAWEV8M1C7hCP7DIMbajwRF0nN3AEQ1jFo
9I+u0BJCBudE7k0f3kt3TaOmtZHCuW08HNoE4JAAtTF5t9ZI29VHU8mNBFaqXdxfe
MwASeY525NyM/TmfVr0A
-----END CERTIFICATE-----
```

D.3 coap_client.crt

```
-----BEGIN CERTIFICATE-----
MIIDazCCA10gAwIBAgIUeCIA1sxlL1b7PfbQ6k/GwodV0eQwDQYJKoZIhvcNAQEL
BQAwrTElMAkGA1UEBhMCVUxEzARBgNVBAMcL1NvbWUtU3RhdGUxITAfBgNVBAoM
GEludGVybWV0IFdpZGdpdHMgUHR5IEExOZDAeFw0yMDEwMDEwMDEwMDEwMDEwMDEw
MjYxOTM1NDRAmEUxCzAJBgNVBAYTAkFVMRMwEQYDVRQQIDApTb211LVNOYXR1MSEw
HwYDVQQKBHJbnR1cm5ldCBXaWRnaXRzIFB0eSBMdGQwggEiMAOGCSqGSIb3DQEB
AQUAA4IBDwAwggEKAoIBAQR4CSDyEkjkfAyoIO9wFd8z8MjsRaCJkhPqwFB52d
/k9dBC1IM0yY79ETf9tpayP2VTGdq29yj0Pe8TKi/deFPG9RJCtw5c1sZebdzgL/
7N8WQ31vg0s8H4hsjQB7bKfQTL3cC3DNQPW3ZCeHPGFtoCr5Gn2byiNoIt9SB+Nd
FzS/0GrGFDWehk4mZW4+PmLzdV1wtEvlGmqQ5QKelDDesk7uGgYnWUycf+qgSWlM
NHgJCBR493brQndMfmU/1eXXvLTEl9nfkuYPBW78gHOYb8stjc/loAyMmqr6vao+
k0vWE+MKr7tWOKQF85r3Q5dHg/a2CiUMy1idMX9PkR3vAgMBAAGjUzBRMBOGA1Ud
DgQWBBS9xY9cgtftUHfsGw2h1Nqz8ShLvTAPBgNVHSMEGDAWgBS9xY9cgtftUHfs
Gw2h1Nqz8ShLvTAPBgNVHRMBAf8EBTADAQH/MAOGCSqGSIb3DQEBCwUAA4IBAQDW
6Hw2Sqe5ZEVYQ4h9IpW2Ky4uqJsdZrsMMRdICf4/t0Ya0q54topul/OioBIt05U
00A/yUGRg9Ybmzcc2elz1L6spo0bQ7jfn40k5P7/9VSotWzcXv3e0vxSn7ZsibhB
Cszo04pUtzzPrnvkAr/vzuj6rrGvDKv/HGylBjf3JNFiarDsXenwgiOpNVPRpaBq
cAz+aB/OCmINQ4pc5fNTRW7FcgGu0AdAWEV8M1C7hCP7DIMbajwRF0nN3AEQ1jFo
9I+u0BJCBudE7k0f3kt3TaOmtZHCuW08HNoE4JAAtTF5t9ZI29VHU8mNBFaqXdxfe
MwASeY525NyM/TmfVr0A
-----END CERTIFICATE-----
```

D.4 coap_client.key

```
-----BEGIN PRIVATE KEY-----
MIIEwAIBADANBgkqhkiG9w0BAQEFAASCBKowggSmAgEAAoIBAQR4CSDyEkjkfA
yoIO9wFd8z8MjsRaCJkhPqwFB52d/k9dBC1IM0yY79ETf9tpayP2VTGdq29yj0Pe
8TKi/deFPG9RJCtw5c1sZebdzgL/7N8WQ31vg0s8H4hsjQB7bKfQTL3cC3DNQPW3
ZCeHPGFtoCr5Gn2byiNoIt9SB+NdFzS/0GrGFDWehk4mZW4+PmLzdV1wtEvlGmqQ
5QKelDDesk7uGgYnWUycf+qgSWlMNHgJCBR493brQndMfmU/1eXXvLTEl9nfkuYP
```

```

BW78gHOYb8stjc/loAyMmqr6vao+k0vwE+MKr7tW0KQF85r3Q5dHg/a2CiUMy1id
MX9PkR3vAgMBAACgEBAIUfhcjhXSIWHZ5SR7mUWhgNCErfeYZ9clvfKpyx8ld0
WD9FueW0BeDBBCz/bvs1Q152U+Lbye51QDFSvztKCaVt7IrDN7SbjlThaDgwfPIw
R6R5iDTg4moQcqv10s29tJFnGxndR0Jh0JzAHgCErud+RdUsdBI3WpHXnSPtzQnX
hCj8J18iSJS0nWiKXFHYGTjtthmnhH+HE9IbpBrKwVR0QuusAVerxJESha/8PnG+
Swsx0HfEX+EBE2bNpYfarGTu56bjqkTIse2gmt+bPuMqha54EQxxsSU/Wh4IPZtR
lsuHCs55bLBfnlPjjJZ2wQG85rGVSiuZhbQz7N0e1ECgYEA7I8vN8IHs6+rpfpU
u7QqGLDUsxbl6vKTmilBbL7h+N5+0wtXbxV3KAoP4XNAZJp8mNu3hoV0fhhMYSzp
07RvV9bv9QKbFuNFHCgA6XSu3LcY7SiBjYr9zMwFdj5NROXWBiMLGRLf4vjVactP
+iG6BNVX9H657G7m6aunvDltQ1cCgYEA6g2o+JwZm6Kg5JftTjKzsAPryXiJuVyp
1i3AktgLJCMBGo+9pHcRwbSAP+vg9JaJvf6v93IT9WJrlKNwcYpwwv0qWuZChV
igt0SbjzU/6b0ELck6iUZDlv4A/8E1EVcj4j000DsibHr7rNhDNaaTciHg0kk1Lj
otG88BX3sikCgYEA0nKjQAhmj61Fl/g273HdTW/rGZ07P74kH656HMNIbV3xd9EM
LK+8/Kr06/N7IsTo+ZfmEk9/v96KX48KqiNINq3pdV+nF1qCfGT8orQCaCqfiORQ
1NoE5e/PIB1W5IQ1XepJEjpfY02b9m1ALjdY5LnjcIhY4QTcep8SLvorwCsCgYEA
xNVGFAXbN1r2aigmpwwt6Ekiy02109JeViyygusmvBhrtGLL/uMv6LJu5NLvNWL
gZgspWzTx+fySMsecia/wRY7vluVpJ+TCGwHmPRUln/Z87Q3Imq0mPEA1/M5b91R
6ui84lsuBwW8C1pS+eniD/gQ1iMEJe2giM5QUax3ybkCgYEAyNjgV7wuFZASf/M9
ohxwvqUKB+0P8wZ3VkusK5jNxC/6+XmPuH6PXM4uUJvIYionf1Hqa0UaHTbIUrJd
adfSsPpMgVqYXBSL3ceLi0B5NAu7/PYf1Y7T2jPlJmhn+N338Mv2YT+Zjnn/nds
bSTFBux8z0fL6UNiqMJ0bNU/0iU=
-----END PRIVATE KEY-----

```

D.5 payload.h

```

1  #ifndef COAP_CLIENT_PAYLOAD_H
2  #define COAP_CLIENT_PAYLOAD_H
3
4  #define PAYLOAD_ID "PAYLOAD"
5
6  typedef struct Payload {
7      char id[8];
8      float temperature;
9      float pressure;
10     float altitude;
11 } Payload;
12
13 #endif //COAP_CLIENT_PAYLOAD_H

```

D.6 coap.c

```
1  #include <string.h>
2  #include <stdio.h>
3  #include <sys/socket.h>
4  #include <netdb.h>
5  #include <sys/param.h>
6
7  #include "freertos/FreeRTOS.h"
8  #include "freertos/task.h"
9  #include "freertos/event_groups.h"
10
11 #include "esp_log.h"
12 #include "esp_wifi.h"
13 #include "esp_event.h"
14
15 #include "nvs_flash.h"
16
17 #include "protocol_examples_common.h"
18
19 /* Needed until coap_dtls.h becomes a part of libcoap proper */
20 #include "libcoap.h"
21 #include "coap_dtls.h"
22
23 #include "coap.h"
24 #include "payload.h"
25
26 #define COAP_DEFAULT_TIME_SEC 5
27
28 /* The examples use simple Pre-Shared-Key configuration that you can set via
29    'make menuconfig'.
30
31    If you'd rather not, just change the below entries to strings with
32    the config you want - ie #define EXAMPLE_COAP_PSK_KEY
33    ↪ "some-agreed-preshared-key"
34
35    Note: PSK will only be used if the URI is prefixed with coaps://
36    instead of coap:// and the PSK must be one that the server supports
37    (potentially associated with the IDENTITY)
38    */
39 #define EXAMPLE_COAP_PSK_KEY CONFIG_EXAMPLE_COAP_PSK_KEY
40 #define EXAMPLE_COAP_PSK_IDENTITY CONFIG_EXAMPLE_COAP_PSK_IDENTITY
```

```

40
41  /* The examples use uri Logging Level that
42  you can set via 'make menuconfig'.
43
44  If you'd rather not, just change the below entry to a value
45  that is between 0 and 7 with
46  the config you want - ie #define EXAMPLE_COAP_LOG_DEFAULT_LEVEL 7
47  */
48  #define EXAMPLE_COAP_LOG_DEFAULT_LEVEL CONFIG_COAP_LOG_DEFAULT_LEVEL
49
50  /* The examples use uri "coap://californium.eclipse.org" that
51  you can set via the project configuration (idf.py menuconfig)
52
53  If you'd rather not, just change the below entries to strings with
54  the config you want - ie #define COAP_DEFAULT_DEMO_URI
55  ↪ "coaps://californium.eclipse.org"
56  */
57
58  #define COAP_DEFAULT_DEMO_URI CONFIG_EXAMPLE_TARGET_DOMAIN_URI
59
60  const static char *TAG = "CoAP_client";
61
62  static int resp_wait = 1;
63  static coap_optlist_t *optlist = NULL;
64  static coap_string_t payload = { 0, NULL };      /* optional payload to send */
65  static int wait_ms;
66
67  #ifdef CONFIG_COAP_MBEDTLS_PKI
68  /* CA cert, taken from coap_ca.pem
69  Client cert, taken from coap_client.crt
70  Client key, taken from coap_client.key
71
72  The PEM, CRT and KEY file are examples taken from the wpa2 enterprise
73  example.
74
75  To embed it in the app binary, the PEM, CRT and KEY file is named
76  in the component.mk COMPONENT_EMBED_TXTFILES variable.
77  */
78  extern uint8_t ca_pem_start[] asm("_binary_coap_ca_pem_start");
79  extern uint8_t ca_pem_end[] asm("_binary_coap_ca_pem_end");
80  extern uint8_t client_cert_start[] asm("_binary_coap_client_cert_start");
81  extern uint8_t client_cert_end[] asm("_binary_coap_client_cert_end");
82  extern uint8_t client_key_start[] asm("_binary_coap_client_key_start");
83  extern uint8_t client_key_end[] asm("_binary_coap_client_key_end");

```

```

82  #endif /* CONFIG_COAP_MBEDTLS_PKI */
83
84  /**
85    * Calculates decimal value from hexadecimal ASCII character given in
86    * @p c. The caller must ensure that @p c actually represents a valid
87    * hexadecimal character, e.g. with isxdigit(3).
88    *
89    * @hideinitializer
90    */
91  #define hexchar_to_dec(c) ((c) & 0x40 ? ((c) & 0x0F) + 9 : ((c) & 0x0F))
92
93  static void decode_segment(const uint8_t *seg, size_t length, unsigned char *buf) {
94      while (length--) {
95          if (*seg == '%') {
96              *buf = (hexchar_to_dec(seg[1]) << 4) + hexchar_to_dec(seg[2]);
97              seg += 2; length -= 2;
98          } else {
99              *buf = *seg;
100         }
101         ++buf; ++seg;
102     }
103 }
104
105 static void message_handler(coap_context_t *ctx, coap_session_t *session,
106                             coap_pdu_t *sent, coap_pdu_t *received,
107                             const coap_tid_t id) {
108     unsigned char *data = NULL;
109     size_t data_len;
110     coap_pdu_t *pdu = NULL;
111     coap_opt_t *block_opt;
112     coap_opt_iterator_t opt_iter;
113     unsigned char buf[4];
114     coap_optlist_t *option;
115     coap_tid_t tid;
116
117     if (COAP_RESPONSE_CLASS(received->code) == 2) {
118         /* Need to see if blocked response */
119         block_opt = coap_check_option(received, COAP_OPTION_BLOCK2, &opt_iter);
120         if (block_opt) {
121             uint16_t blktype = opt_iter.type;
122
123             if (COAP_OPT_BLOCK_MORE(block_opt)) {
124                 /* more bit is set */

```



```

125
126     /* create pdu with request for next block */
127     pdu = coap_new_pdu(session);
128     if (!pdu) {
129         ESP_LOGE(TAG, "coap_new_pdu() failed");
130         goto clean_up;
131     }
132     pdu->type = COAP_MESSAGE_CON;
133     pdu->tid = coap_new_message_id(session);
134     pdu->code = COAP_REQUEST_GET;
135
136     /* add URI components from optlist */
137     for (option = optlist; option; option = option->next) {
138         switch (option->number) {
139             case COAP_OPTION_URI_HOST :
140             case COAP_OPTION_URI_PORT :
141             case COAP_OPTION_URI_PATH :
142             case COAP_OPTION_URI_QUERY :
143                 coap_add_option(pdu, option->number, option->length,
144                               option->data);
145                 break;
146             default:;      /* skip other options */
147         }
148     }
149
150     /* finally add updated block option from response, clear M bit */
151     /* blocknr = (blocknr & 0xfffffff7) + 0x10; */
152     coap_add_option(pdu,
153                   blktype,
154                   coap_encode_var_safe(buf, sizeof(buf),
155                                       ((coap_opt_block_num(block_opt)
156                                        ↪ + 1) << 4) |
157                                       COAP_OPT_BLOCK_SZX(block_opt)),
158                                       ↪ buf);
159
160     tid = coap_send(session, pdu);
161
162     if (tid != COAP_INVALID_TID) {
163         resp_wait = 1;
164         wait_ms = COAP_DEFAULT_TIME_SEC * 1000;
165         return;
166     }
167 }

```

```

166         printf("\n");
167     } else {
168         if (coap_get_data(received, &data_len, &data)) {
169             printf("=== Received ===\n%. *s\n\n", (int)data_len, data);
170         }
171     }
172 }
173 clean_up:
174     resp_wait = 0;
175 }
176
177 #ifdef CONFIG_COAP_MBEDTLS_PKI
178
179 static int
180 verify_cn_callback(const char *cn,
181                   const uint8_t *asn1_public_cert,
182                   size_t asn1_length,
183                   coap_session_t *session,
184                   unsigned depth,
185                   int validated,
186                   void *arg
187                  )
188 {
189     coap_log(LOG_INFO, "CN '%s' presented by server (%s)\n",
190             cn, depth ? "CA" : "Certificate");
191     return 1;
192 }
193 #endif /* CONFIG_COAP_MBEDTLS_PKI */
194
195 static void coap_example_client(void *pvParameters)
196 {
197     Payload data;
198     memcpy(&data, pvParameters, sizeof(Payload));
199
200     size_t len = 0;
201     len = snprintf(NULL, len, "Temperature = %.2f °C\nPressure = %.2f Pa\nAltitude =
↵ %.2f m", (double) data.temperature, (double) data.pressure, (double)
↵ data.altitude);
202
203     char buffer[len+1];
204     snprintf(buffer, len+1, "Temperature = %.2f °C\nPressure = %.2f Pa\nAltitude =
↵ %.2f m", (double) data.temperature, (double) data.pressure, (double)
↵ data.altitude);

```

```

205
206     struct hostent *hp;
207     coap_address_t dst_addr;
208     static coap_uri_t uri;
209     const char *server_uri = COAP_DEFAULT_DEMO_URI;
210     char *phostname = NULL;
211
212     coap_set_log_level(EXAMPLE_COAP_LOG_DEFAULT_LEVEL);
213
214     while (1) {
215 #define BUFSIZE 40
216         unsigned char _buf[BUFSIZE];
217         unsigned char *buf;
218         size_t buflen;
219         int res;
220         coap_context_t *ctx = NULL;
221         coap_session_t *session = NULL;
222         coap_pdu_t *request = NULL;
223
224         optlist = NULL;
225         if (coap_split_uri((const uint8_t *) server_uri, strlen(server_uri), &uri)
226             ↪ == -1) {
227             ESP_LOGE(TAG, "CoAP server uri error");
228             break;
229         }
230         if (uri.scheme == COAP_URI_SCHEME_COAPS && !coap_dtls_is_supported()) {
231             ESP_LOGE(TAG, "MbedTLS (D)TLS Client Mode not configured");
232             break;
233         }
234         if (uri.scheme == COAP_URI_SCHEME_COAPS_TCP && !coap_tls_is_supported()) {
235             ESP_LOGE(TAG, "CoAP server uri coaps+tcp:// scheme is not supported");
236             break;
237         }
238
239         phostname = (char *) calloc(1, uri.host.length + 1);
240         if (phostname == NULL) {
241             ESP_LOGE(TAG, "calloc failed");
242             break;
243         }
244
245         memcpy(phostname, uri.host.s, uri.host.length);
246         hp = gethostbyname(phostname);

```

```

247     free(phostname);
248
249     if (hp == NULL) {
250         ESP_LOGE(TAG, "DNS lookup failed");
251         vTaskDelay(1000 / portTICK_PERIOD_MS);
252         free(phostname);
253         continue;
254     }
255     char tmpbuf[INET6_ADDRSTRLEN];
256     coap_address_init(&dst_addr);
257     switch (hp->h_addrtype) {
258         case AF_INET:
259             dst_addr.addr.sin.sin_family = AF_INET;
260             dst_addr.addr.sin.sin_port = htons(uri.port);
261             memcpy(&dst_addr.addr.sin.sin_addr, hp->h_addr,
262                 ↪ sizeof(dst_addr.addr.sin.sin_addr));
263             inet_ntop(AF_INET, &dst_addr.addr.sin.sin_addr, tmpbuf,
264                 ↪ sizeof(tmpbuf));
265             ESP_LOGI(TAG, "DNS lookup succeeded. IP=%s", tmpbuf);
266             break;
267         case AF_INET6:
268             dst_addr.addr.sin6.sin6_family = AF_INET6;
269             dst_addr.addr.sin6.sin6_port = htons(uri.port);
270             memcpy(&dst_addr.addr.sin6.sin6_addr, hp->h_addr,
271                 ↪ sizeof(dst_addr.addr.sin6.sin6_addr));
272             inet_ntop(AF_INET6, &dst_addr.addr.sin6.sin6_addr, tmpbuf,
273                 ↪ sizeof(tmpbuf));
274             ESP_LOGI(TAG, "DNS lookup succeeded. IP=%s", tmpbuf);
275             break;
276         default:
277             ESP_LOGE(TAG, "DNS lookup response failed");
278             goto clean_up;
279     }
280
281     if (uri.path.length) {
282         buflen = BUFSIZE;
283         buf = _buf;
284         res = coap_split_path(uri.path.s, uri.path.length, buf, &buflen);
285
286         while (res--) {
287             coap_insert_optlist(&optlist,
288                               coap_new_optlist(COAP_OPTION_URI_PATH,
289                                                 coap_opt_length(buf),

```

```

286                                     coap_opt_value(buf));
287
288             buf += coap_opt_size(buf);
289     }
290 }
291
292     if (uri.query.length) {
293         buflen = BUFSIZE;
294         buf = _buf;
295         res = coap_split_query(uri.query.s, uri.query.length, buf, &buflen);
296
297         while (res--) {
298             coap_insert_optlist(&optlist,
299                                 coap_new_optlist(COAP_OPTION_URI_QUERY,
300                                                    coap_opt_length(buf),
301                                                    coap_opt_value(buf)));
302
303             buf += coap_opt_size(buf);
304         }
305     }
306
307     ctx = coap_new_context(NULL);
308     if (!ctx) {
309         ESP_LOGE(TAG, "coap_new_context() failed");
310         goto clean_up;
311     }
312
313     /*
314      * Note that if the URI starts with just coap:// (not coaps://) the
315      * session will still be plain text.
316      *
317      * coaps+tcp:// is NOT supported by the libcoap->mbedtls interface
318      * so COAP_URI_SCHEME_COAPS_TCP will have failed in a test above,
319      * but the code is left in for completeness.
320      */
321     if (uri.scheme == COAP_URI_SCHEME_COAPS || uri.scheme ==
322         ⇨ COAP_URI_SCHEME_COAPS_TCP) {
323 #ifndef CONFIG_MBEDTLS_TLS_CLIENT
324         ESP_LOGE(TAG, "MbedTLS (D)TLS Client Mode not configured");
325         goto clean_up;
326 #endif /* CONFIG_MBEDTLS_TLS_CLIENT */
327 #ifdef CONFIG_COAP_MBEDTLS_PSK
328         session = coap_new_client_session_psk(ctx, NULL, &dst_addr,

```

```

328                                     uri.scheme ==
329                                     ↪ COAP_URI_SCHEME_COAPS ?
330                                     ↪ COAP_PROTO_DTLS :
331                                     ↪ COAP_PROTO_TLS,
332                                     EXAMPLE_COAP_PSK_IDENTITY,
333                                     (const uint8_t
334                                     ↪ *)EXAMPLE_COAP_PSK_KEY,
335                                     sizeof(EXAMPLE_COAP_PSK_KEY) - 1);
336 #endif /* CONFIG_COAP_MBEDTLS_PSK */
337
338 #ifdef CONFIG_COAP_MBEDTLS_PKI
339     unsigned int ca_pem_bytes = ca_pem_end - ca_pem_start;
340     unsigned int client_cert_bytes = client_cert_end - client_cert_start;
341     unsigned int client_key_bytes = client_key_end - client_key_start;
342     coap_dtls_pki_t dtls_pki;
343     static char client_sni[256];
344
345     memset (&dtls_pki, 0, sizeof(dtls_pki));
346     dtls_pki.version = COAP_DTLS_PKI_SETUP_VERSION;
347     if (ca_pem_bytes) {
348         /*
349          * Add in additional certificate checking.
350          * This list of enabled can be tuned for the specific
351          * requirements - see 'man coap_encryption'.
352          *
353          * Note: A list of root ca file can be setup separately using
354          * coap_context_set_pki_root_cas(), but the below is used to
355          * define what checking actually takes place.
356          */
357         dtls_pki.verify_peer_cert      = 1;
358         dtls_pki.require_peer_cert     = 1;
359         dtls_pki.allow_self_signed     = 1;
360         dtls_pki.allow_expired_certs   = 1;
361         dtls_pki.cert_chain_validation = 1;
362         dtls_pki.cert_chain_verify_depth = 2;
363         dtls_pki.check_cert_revocation = 1;
364         dtls_pki.allow_no_crl          = 1;
365         dtls_pki.allow_expired_crl     = 1;
366         dtls_pki.allow_bad_md_hash     = 1;
367         dtls_pki.allow_short_rsa_length = 1;
368         dtls_pki.validate_cn_call_back  = verify_cn_callback;
369         dtls_pki.cn_call_back_arg       = NULL;
370         dtls_pki.validate_sni_call_back = NULL;

```

```

367         dtls_pki.sni_call_back_arg      = NULL;
368         memset(client_sni, 0, sizeof(client_sni));
369         if (uri.host.length) {
370             memcpy(client_sni, uri.host.s, MIN(uri.host.length,
371                 ↪ sizeof(client_sni)));
372         } else {
373             memcpy(client_sni, "localhost", 9);
374         }
375         dtls_pki.client_sni = client_sni;
376     }
377     dtls_pki.pki_key.key_type = COAP_PKI_KEY_PEM_BUF;
378     dtls_pki.pki_key.key.pem_buf.public_cert = client_cert_start;
379     dtls_pki.pki_key.key.pem_buf.public_cert_len = client_cert_bytes;
380     dtls_pki.pki_key.key.pem_buf.private_key = client_key_start;
381     dtls_pki.pki_key.key.pem_buf.private_key_len = client_key_bytes;
382     dtls_pki.pki_key.key.pem_buf.ca_cert = ca_pem_start;
383     dtls_pki.pki_key.key.pem_buf.ca_cert_len = ca_pem_bytes;
384
385     session = coap_new_client_session_pki(ctx, NULL, &dst_addr,
386                                         uri.scheme ==
387                                         ↪ COAP_URI_SCHEME_COAPS ?
388                                         ↪ COAP_PROTO_DTLS :
389                                         ↪ COAP_PROTO_TLS,
390                                         &dtls_pki);
391
392     #endif /* CONFIG_COAP_MBEDTLS_PKI */
393     } else {
394         session = coap_new_client_session(ctx, NULL, &dst_addr,
395                                         uri.scheme == COAP_URI_SCHEME_COAP_TCP
396                                         ↪ ? COAP_PROTO_TCP :
397                                         ↪ COAP_PROTO_UDP);
398     }
399     if (!session) {
400         ESP_LOGE(TAG, "coap_new_client_session() failed");
401         goto clean_up;
402     }
403
404     coap_register_response_handler(ctx, message_handler);
405
406     request = coap_new_pdu(session);
407     if (!request) {
408         ESP_LOGE(TAG, "coap_new_pdu() failed");
409         goto clean_up;
410     }

```

```

405
406     /* Set request options. */
407     request->type = COAP_MESSAGE_CON;
408     request->tid = coap_new_message_id(session);
409     request->code = COAP_REQUEST_POST;
410     coap_add_optlist_pdu(request, &optlist);
411
412     /* Set request payload. */
413     size_t length = sizeof(buffer);
414     payload.s = (unsigned char *) coap_malloc(length);
415     payload.length = length;
416     decode_segment((const uint8_t *) &buffer, length, payload.s);
417     coap_add_data(request, payload.length, payload.s);
418
419     printf("=== Sent ===\n%.*s\n\n", (int) length, buffer);
420
421     resp_wait = 1;
422     coap_send(session, request);
423
424     wait_ms = COAP_DEFAULT_TIME_SEC * 1000;
425
426     while (resp_wait) {
427         int result = coap_run_once(ctx, wait_ms > 1000 ? 1000 : wait_ms);
428         if (result >= 0) {
429             if (result >= wait_ms) {
430                 ESP_LOGE(TAG, "select timeout");
431                 break;
432             } else {
433                 wait_ms -= result;
434             }
435         }
436     }
437     clean_up:
438     if (optlist) {
439         coap_delete_optlist(optlist);
440         optlist = NULL;
441     }
442     if (session) {
443         coap_session_release(session);
444     }
445     if (ctx) {
446         coap_free_context(ctx);
447     }

```



```

448     coap_cleanup();
449     /*
450      * change the following line to something like sleep(2)
451      * if you want the request to continually be sent
452      */
453     break;
454 }
455
456 vTaskDelete(NULL);
457 }
458
459 static void setup_coap()
460 {
461     ESP_ERROR_CHECK( nvs_flash_init() );
462     ESP_ERROR_CHECK(esp_netif_init());
463     ESP_ERROR_CHECK(esp_event_loop_create_default());
464
465     /* This helper function configures Wi-Fi or Ethernet, as selected in
466      ↪ menuconfig.
467      * Read "Establishing Wi-Fi or Ethernet Connection" section in
468      * examples/protocols/README.md for more information about this function.
469      */
470     ESP_ERROR_CHECK(example_connect());
471 }

```

D.7 uart.c

```

1  #include <string.h>
2  #include "payload.h"
3  #include "driver/uart.h"
4  #include "driver/gpio.h"
5
6  /**
7   * This is an example which echos any data it receives on configured UART back to
8   ↪ the sender,
9   * with hardware flow control turned off. It does not use UART driver event queue.
10  *
11  * - Port: configured UART
12  * - Receive (Rx) buffer: on
13  * - Transmit (Tx) buffer: off
14  * - Flow control: off
15  * - Event queue: off

```

```

15  * - Pin assignment: see defines below (See Kconfig)
16  */
17
18  #define ECHO_TEST_TXD (CONFIG_EXAMPLE_UART_TXD)
19  #define ECHO_TEST_RXD (CONFIG_EXAMPLE_UART_RXD)
20  #define ECHO_TEST_RTS (UART_PIN_NO_CHANGE)
21  #define ECHO_TEST_CTS (UART_PIN_NO_CHANGE)
22
23  #define ECHO_UART_PORT_NUM      (CONFIG_EXAMPLE_UART_PORT_NUM)
24  #define ECHO_UART_BAUD_RATE    (CONFIG_EXAMPLE_UART_BAUD_RATE)
25  #define ECHO_TASK_STACK_SIZE   (CONFIG_EXAMPLE_TASK_STACK_SIZE)
26
27  #define BUF_SIZE (1024)
28
29  #define PAYLOAD_ID "PAYLOAD"
30
31  typedef struct Payload {
32      char id[8];
33      float temperature;
34      float pressure;
35      float altitude;
36  } Payload;
37
38  static void setup_uart() {
39      /* Configure parameters of an UART driver,
40       * communication pins and install the driver */
41      uart_config_t uart_config = {
42          .baud_rate = ECHO_UART_BAUD_RATE,
43          .data_bits = UART_DATA_8_BITS,
44          .parity     = UART_PARITY_DISABLE,
45          .stop_bits = UART_STOP_BITS_1,
46          .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
47          .source_clk = UART_SCLK_APB,
48      };
49      int intr_alloc_flags = 0;
50
51      #if CONFIG_UART_ISR_IN_IRAM
52          intr_alloc_flags = ESP_INTR_FLAG_IRAM;
53      #endif
54
55      ESP_ERROR_CHECK(uart_driver_install(ECHO_UART_PORT_NUM, BUF_SIZE * 2, 0, 0,
56          ↪ NULL, intr_alloc_flags));
57      ESP_ERROR_CHECK(uart_param_config(ECHO_UART_PORT_NUM, &uart_config));

```

```

57     ESP_ERROR_CHECK(uart_set_pin(ECHO_UART_PORT_NUM, ECHO_TEST_TXD, ECHO_TEST_RXD,
    ↪     ECHO_TEST_RTS, ECHO_TEST_CTS));
58 }
59
60 static void listen_uart(void *arg, void (*callback)(void *)) {
61     Payload payload;
62     while (1) {
63         int len = uart_read_bytes(ECHO_UART_PORT_NUM, &payload, sizeof(Payload), 20
    ↪         / portTICK_RATE_MS);
64
65         if (memcmp(&payload.id, PAYLOAD_ID, sizeof(payload.id)) == 0 && len ==
    ↪         sizeof(Payload)) {
66             xTaskCreate(callback, "callback_task", 8 * 1024, &payload, 5, NULL);
67         }
68     }
69 }

```

D.8 main.c

```

1     #include "uart.c"
2     #include "coap.c"
3
4     static void start_listening(void *arg) {
5         setup_coap();
6         setup_uart();
7         listen_uart(arg, coap_example_client);
8         vTaskDelete(NULL);
9     }
10
11 void app_main(void) {
12     xTaskCreate(start_listening, "coap", 8 * 1024, NULL, 5, NULL);
13 }

```

D.9 Kconfig.projbuild

```

1     menu "Example CoAP Client Configuration"
2
3         config EXAMPLE_TARGET_DOMAIN_URI
4             string "Target Uri"
5             default "coaps://californium.eclipse.org"
6             help

```

```

7         Target uri for the example to use. Use coaps:// prefix for encrypted
8         ↪ traffic
9
10        using Pre-Shared Key (PSK) or Public Key Infrastructure (PKI).
11
12    config EXAMPLE_COAP_PSK_KEY
13        string "Preshared Key (PSK) to used in the connection to the CoAP server"
14        depends on COAP_MBEDTLS_PSK
15        default "sesame"
16        help
17            The Preshared Key to use to encrypt the communicatons. The same key must
18            ↪ be
19            used at both ends of the CoAP connection, and the CoaP client must
20            ↪ request
21            an URI prefixed with coaps:// instead of coap:// for DTLS to be used.
22
23    config EXAMPLE_COAP_PSK_IDENTITY
24        string "PSK Client identity (username)"
25        depends on COAP_MBEDTLS_PSK
26        default "password"
27        help
28            The identity (or username) to use to identify to the CoAP server which
29            PSK key to use.
30
31    endmenu
32
33    menu "Example UART Configuration"
34
35        config EXAMPLE_UART_PORT_NUM
36            int "UART port number"
37            range 0 2 if IDF_TARGET_ESP32
38            range 0 1 if IDF_TARGET_ESP32S2
39            default 2 if IDF_TARGET_ESP32
40            default 1 if IDF_TARGET_ESP32S2
41            help
42                UART communication port number for the example.
43                See UART documentation for available port numbers.
44
45        config EXAMPLE_UART_BAUD_RATE
46            int "UART communication speed"
47            range 1200 115200
48            default 115200
49            help
50                UART communication speed for Modbus example.

```

```
47
48     config EXAMPLE_UART_RXD
49         int "UART RXD pin number"
50         range 0 34 if IDF_TARGET_ESP32
51         range 0 46 if IDF_TARGET_ESP32S2
52         default 5
53         help
54             GPIO number for UART RX pin. See UART documentation for more information
55             about available pin numbers for UART.
56
57     config EXAMPLE_UART_TXD
58         int "UART TXD pin number"
59         range 0 34 if IDF_TARGET_ESP32
60         range 0 46 if IDF_TARGET_ESP32S2
61         default 4
62         help
63             GPIO number for UART TX pin. See UART documentation for more information
64             about available pin numbers for UART.
65
66     config EXAMPLE_TASK_STACK_SIZE
67         int "UART echo example task stack size"
68         range 1024 16384
69         default 2048
70         help
71             Defines stack size for UART echo example. Insufficient stack size can
72             ↪ cause crash.
73 endmenu
```

E Aplicação 3: coap-server

E.1 Estrutura do projeto

```
libcoap/  
  examples/  
    certs/  
      certfile.pem  
      keyfile.pem  
      coap-server.c
```

E.2 Certificados

E.2.1 certfile.pem

```
-----BEGIN CERTIFICATE-----  
MIIDazCCA10gAwIBAgIUeCIA1sxlL1b7PfbQ6k/GwodV0eQwDQYJKoZIhvcNAQEL  
BQAwRTELMAkGA1UEBhMCVUxEzARBgNVBAgMC1NvbWUtU3RhdGUxITAfBgNVBAoM  
GEludGVybmVOIFdpZGdpdHMgUHR5IEExOZDAeFw0yMDEwMjE1NDRaFw0yMDEw  
MjYxOTM1NDRaMEUxCzAJBgNVBAYTAkFVMRMwEQYDVRQQIDApTb211LVNOYXR1MSEw  
HwYDVQQKBhJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGQwggEiMAOGCSqGSIb3DQEB  
AQUAA4IBDwAwggEKAoIBAQR4CSDyEkjfkAyoI09wFd8z8MjsRaCJkhPqwFB52d  
/k9dBC1IM0yY79ETf9tpayP2VTGdq29yj0Pe8TKi/deFPG9RJCtw5c1sZebdzg/  
7N8WQ31vg0s8H4hSjQB7bKfQTL3c3DNQPW3ZCeHPGFtoCr5Gn2byiNoIt9SB+Nd  
FzS/0GrGFDWehk4mZW4+PmLzdV1wtEv1GmqQ5QKe1DDesk7uGgYnWUycf+qgSWlM  
NHgJCBR493brQndMfmu/1eXXvLTEL9nfkuYPBW78gH0Yb8stjc/loAyMmqr6vao+  
k0vve+MKr7tWOKQF85r3Q5dHg/a2CiUMy1idMX9PkR3vAgMBAAGjUzBRMBOGA1Ud  
DgQWBBS9xY9cgtftUHfsGw2h1Nqz8ShLvTafBgNVHSMEGDAWgBS9xY9cgtftUHfs  
Gw2h1Nqz8ShLvTAPBgNVHRMBAf8EBTADAQH/MAOGCSqGSIb3DQEBwUAA4IBAQDW  
6Hw2Sqe5ZEVYQ4h9IpW2KyU4uqJsdZrsMMRdICf4/t0Ya0q54topul/OioBIt05U  
00A/yUGRg9Ybmzcc2elz1L6spo0bQ7jfn40k5P7/9VSotWzcxv3e0vxSn7ZsibhB  
Cszo04pUtzzPrnvkAr/vzuj6rrGvDKv/HGylBjf3JNFiarDsXenwgi0pNVPRpaBq  
cAz+aB/OCmINQ4pc5fNTRW7FcgGu0AdAWEV8M1C7hCP7DIMbajwRF0nN3AEQ1jFo  
9I+u0BJCBudE7k0f3kt3Ta0mtZHCuW08HNoE4JAAtTF5t9ZI29VHU8mNBFaqXdxfe
```

```
MwASeY525NyM/TmfVr0A
-----END CERTIFICATE-----
```

E.2.2 keyfile.pem

```
-----BEGIN PRIVATE KEY-----
MIIeWAIBADANBgkqhkiG9w0BAQEFAASCBKowggSmAgEAAoIBAQQDYR4CSDyEkjKfA
yoI09wFd8z8MjsRaCJkhPqwFB52d/k9dBC1IM0yY79ETf9tpayP2VTGdq29yj0Pe
8TKi/deFPG9RJCtw5c1sZebdzgL/7N8WQ31vg0s8H4hSjQB7bKfQTL3cC3DNQPW3
ZCeHPGFtoCr5Gn2byiNoIt9SB+NdFzS/0GrGFDWehk4mZW4+PmLzdV1wtEv1GmqQ
5QKe1DDesk7uGgYnWUycf+qgSWlMNHgJCB493brQndMfmu/1eXXvLTE19nfkuYP
BW78gHOYb8stjc/loAyMmqr6vao+k0vwE+MKr7tW0KQF85r3Q5dHg/a2CiUMy1id
MX9PkR3vAgMBAAECCgEBAIUfhcjhXSIWHZ5SR7mUWhgNCErfeYZ9c1vfKpyx8ld0
WD9FueW0BeDBBCz/bvs1Q152U+Lbye51QDFSvztKCaVt7IrDN7Sbj1ThaDgwfPIw
R6R5iDTg4moQcqV10s29tJFnGxndROJh0JzAHgCERud+RdUsdBI3WpHXnSPtzQnX
hCj8J18iSJSOnWiKXFHYGTjtmtmhnH+HE9IbpBrKwVROQuusAVerxJESha/8PnG+
Swsx0HfEX+EBe2bNpYfarGTu56bjqkTIse2gmt+bPuMqha54EQxxsSU/Wh4IPZtR
lsuHCs55bLBfnilPjjJZ2wQG85rGVSiuZhbQz7N0e1ECgYEA7I8vN8IHs6+rpfpU
u7QqGLDUxsbL6vKTmilBbL7h+N5+0wtXbxV3KAoP4XNAZJp8mNu3hoV0fhhMYszp
07RvV9bv9QKbFuNFHCgA6XSu3LcY7SiBjYr9zMwFdj5NROXWBiMLGRLf4vjVactP
+iG6BNVX9H657G7m6aunvDltQ1cCgYEA6g2o+JwZm6Kg5JftTjKzsAPryXiJuVyp
1i3AktgLJCMB0Go+9pHcRwbSAP+vg9JaJvf6v93IT9WJrlKNwcYpfwWv0qWuZChV
igt0SbjzU/6b0ELck6iUZDlv4A/8E1EVcj4j000DsibHr7rNhDNaatciHg0kk1Lj
otG88BX3sikCgYEA0nKjQAhmj61Fl/g273HdTW/rGZ07P74kH656HMNIbv3xd9EM
LK+8/Kr06/N7IsTo+ZfmEk9/v96KX48KqinINq3pdV+nF1qCfGT8orQCaCqfiORQ
1NoE5e/PIB1W5IQ1XepJEjpfY02b9m1ALjdY5LnjcIhY4QTcep8SLvorwCsCgYEA
xNVGfAXbN1r2aigmpwvWt6Ekiy02109JeViyygusmvBhrtGLL/uMv6LJu5NLvNWL
gZgspwzTx+fySMsecia/wRY7vluVpJ+TCGwHmPRUln/Z87Q3Imq0mPEA1/M5b91R
6ui84lsuBwW8C1pS+eniD/gQ1iMEJe2giM5QUax3ybkCgYEAyNJgV7wuFZASf/M9
ohxwvqUKB+0P8wZ3VkusK5jNxC/+XmPuH6PXM4uUJvIYionf1Hqa0UaHTbIUrJd
adfSsPpMgVqYXBSL3ceLi0B5NAu7/PYf1Y7T2jP1JJmhn+N338Mv2YT+Zjnn/nds
bSTFBux8z0fL6UNiqMJ0bNU/0iU=
-----END PRIVATE KEY-----
```

E.3 coap-server.c

```
232 ...
233 /* define post handling for sensor data */
234 static void
235 hnd_post_sensor(coap_context_t *ctx UNUSED_PARAM,
236                 struct coap_resource_t *resource,
```

```

237         coap_session_t *session UNUSED_PARAM,
238         coap_pdu_t *request,
239         coap_binary_t *token UNUSED_PARAM,
240         coap_string_t *query UNUSED_PARAM,
241         coap_pdu_t *response) {
242
243     size_t size;
244     unsigned char *data;
245
246     (void) coap_get_data(request, &size, &data);
247
248     if (size > 0) {
249         unsigned char buf[3];
250         response->code = COAP_RESPONSE_CODE(201);
251         coap_add_option(response,
252                         COAP_OPTION_CONTENT_FORMAT,
253                         coap_encode_var_safe(buf, sizeof(buf),
254                                             ↪ COAP_MEDIATYPE_TEXT_PLAIN),
255                         buf);
256         coap_add_data(response, size, (const uint8_t *) data);
257         printf("Counter = %d\n@\n", counter++);
258     }
259     ...

```



```

962     ...
963     static void
964     init_resources(coap_context_t *ctx) {
965         coap_resource_t *r;
966
967         r = coap_resource_init(NULL, 0);
968         coap_register_handler(r, COAP_REQUEST_GET, hnd_get_index);
969
970         coap_add_attr(r, coap_make_str_const("ct"), coap_make_str_const("0"), 0);
971         coap_add_attr(r, coap_make_str_const("title"), coap_make_str_const("\General
972         ↪ Info\"), 0);
973         coap_add_resource(ctx, r);
974
975         /* define sensor resource */
976         r = coap_resource_init(coap_make_str_const("sensor"), resource_flags);
977         coap_register_handler(r, COAP_REQUEST_POST, hnd_post_sensor);
978         coap_add_resource(ctx, r);

```



```

979     /* store clock base to use in /time */
980     my_clock_base = clock_offset;
981
982     r = coap_resource_init(coap_make_str_const("time"), resource_flags);
983     coap_register_handler(r, COAP_REQUEST_GET, hnd_get_time);
984     coap_register_handler(r, COAP_REQUEST_PUT, hnd_put_time);
985     coap_register_handler(r, COAP_REQUEST_DELETE, hnd_delete_time);
986     coap_resource_set_get_observable(r, 1);
987
988     coap_add_attr(r, coap_make_str_const("ct"), coap_make_str_const("0"), 0);
989     coap_add_attr(r, coap_make_str_const("title"), coap_make_str_const("\\"Internal
↪ Clock\\""), 0);
990     coap_add_attr(r, coap_make_str_const("rt"), coap_make_str_const("\\"ticks\\""), 0);
991     coap_add_attr(r, coap_make_str_const("if"), coap_make_str_const("\\"clock\\""), 0);
992
993     coap_add_resource(ctx, r);
994     time_resource = r;
995
996     if (support_dynamic > 0) {
997         /* Create a resource to handle PUTs to unknown URIs */
998         r = coap_resource_unknown_init(hnd_unknown_put);
999         coap_add_resource(ctx, r);
1000     }
1001     #ifndef WITHOUT_ASYNC
1002     r = coap_resource_init(coap_make_str_const("async"), 0);
1003     coap_register_handler(r, COAP_REQUEST_GET, hnd_get_async);
1004
1005     coap_add_attr(r, coap_make_str_const("ct"), coap_make_str_const("0"), 0);
1006     coap_add_resource(ctx, r);
1007     #endif /* WITHOUT_ASYNC */
1008     r = coap_resource_init(coap_make_str_const("example_data"), 0);
1009     coap_register_handler(r, COAP_REQUEST_GET, hnd_get_example_data);
1010     coap_register_handler(r, COAP_REQUEST_PUT, hnd_put_example_data);
1011     coap_resource_set_get_observable(r, 1);
1012
1013     coap_add_attr(r, coap_make_str_const("ct"), coap_make_str_const("0"), 0);
1014     coap_add_attr(r, coap_make_str_const("title"), coap_make_str_const("\\"Example
↪ Data\\""), 0);
1015     coap_add_resource(ctx, r);
1016 }
1017 ...

```

F Artigo

Esta seção contém o artigo resultante deste trabalho no formato da SBC (Sociedade Brasileira de Computação).

Desenvolvimento de uma aplicação IoT utilizando CoAP e DTLS para telemetria veicular

Lukas Derner Grüdtner¹

¹Departamento de Informática e Estatística (INE)
Universidade Federal de Santa Catarina (UFSC)
Florianópolis, SC – Brasil

lukas.grudtner@grad.ufsc.br

Abstract. *The large number of networked devices has brought problems due to the scalability of systems and the data traffic generated. However, another problem that requires attention is the implementation of security mechanisms to guarantee the privacy of the users of these systems. Due to their constrained characteristics, IoT devices require special mechanisms for data protection. This work aimed to carry out a study on the two main protocols proposed to address the communication problem in IoT environments, CoAP and MQTT. For the validation of the model, an IoT application was implemented in the context of vehicular telemetry, using the CoAP protocol with DTLS to communicate between devices and ensure security in the exchange of messages.*

Resumo. *A grande quantidade de dispositivos em rede tem trazido problemas devido a escalabilidade de sistemas e tráfego de dados gerado. No entanto, outro problema que requer atenção é a implantação de mecanismos de segurança para garantir a privacidade dos usuários destes sistemas. Devido aos recursos restritivos, os dispositivos IoT necessitam de mecanismos especiais para a proteção dos dados. Este trabalho teve como objetivo realizar um estudo sobre os dois principais protocolos propostos para tratar o problema de comunicação em ambientes IoT, CoAP e MQTT. Para a validação do modelo, foi implementada uma aplicação IoT no contexto de telemetria veicular, fazendo uso do protocolo CoAP com DTLS para fazer a comunicação entre os dispositivos e garantir a segurança na troca de mensagens.*

1. Introdução

Internet of Things (IoT) é um paradigma relativamente recente que tem tido um enorme crescimento nos últimos anos. Ambientes que integram redes IoT têm por objetivo a coleta de dados através de pequenos sensores acoplados em objetos físicos. Estes sensores são geridos por um pequeno microcontrolador, que tem sob sua supervisão um conjunto de sensores e atuadores. Ao receber os dados, este microcontrolador os envia pela rede, com destino à bancos de dados remotos, a fim de armazená-los. Estes microcontroladores trabalham em baixas frequências e geralmente possuem apenas alguns KiB de memória, o que restringe seu uso à aplicações que não demandam alto processamento. Além disso, os microcontroladores também são capazes de processar comandos através de extensões chamadas de atuadores.

Na indústria automotiva, a telemetria veicular tem sido extensamente utilizada para realizar o monitoramento de diversos aspectos de um veículo, com a finalidade de

efetuar análises em busca de pontos de desperdício, possíveis otimizações e também aumentar a segurança para seus usuários. Estas informações geralmente dizem respeito ao comportamento do veículo em estado de funcionamento, tais como velocidade, aceleração, rotação do motor, pressão dos pneus, geoposicionamento, consumo de combustível, entre diversas outras informações que são possíveis de se coletar através de sensores [Harrington et al. 2004].

Este trabalho teve como objetivo geral o desenvolvimento de uma aplicação IoT, fazendo uso dos protocolos CoAP (*Constrained Application Protocol*) e DTLS (*Data-gram Transport Layer Protocol*) para estabelecer um canal de comunicação seguro entre cliente e servidor. Um microcontrolador ESP32 foi adotado como cliente, o qual faz a coleta de dados do ambiente por meio de sensores e os envia até um servidor utilizando ambos os protocolos para garantir confidencialidade e baixa sobrecarga na transmissão. Por intermédio desta aplicação, experimentos foram realizados com a finalidade de mensurar e comparar algumas das principais *cipher suites* utilizadas para prover segurança à comunicação.

O restante deste artigo está organizado da seguinte maneira: a seção 3 descreve os trabalhos correlatos da literatura; a seção 2 apresenta os conceitos básicos utilizados neste trabalho; na seção 4 é descrito o desenvolvimento da aplicação proposta; e, por fim, na seção 5 são apresentadas as conclusões.

2. Conceitos básicos

2.1. Internet of Things

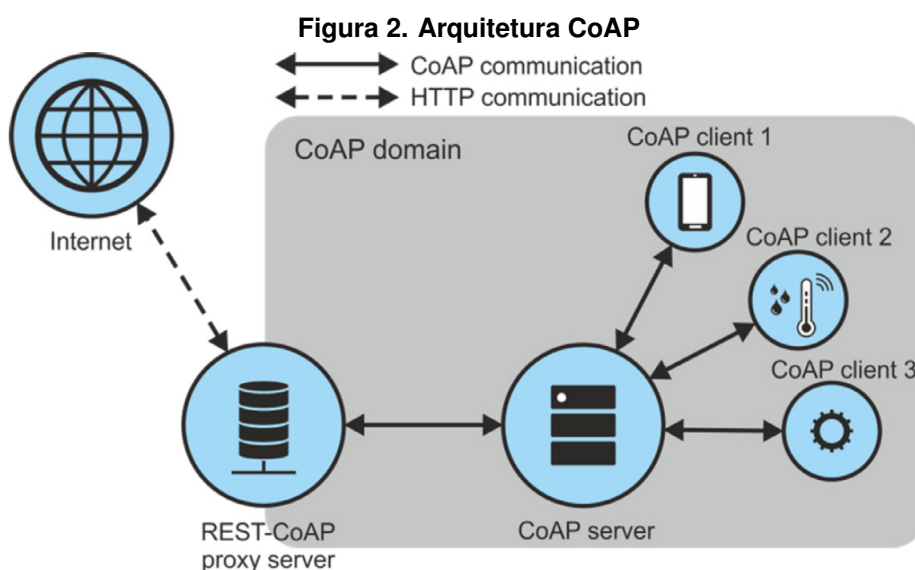
Internet of Things (IoT) é um conceito que abrange toda e qualquer tipo de rede capaz de conectar objetos físicos entre si e que, além disso, permite obter e manipular dados gerados por estes objetos. IoT utiliza a Internet como sistema de comunicação para estabelecer uma interação inteligente entre pessoas e objetos do ambiente [Ray 2016]. São características de dispositivos IoT uma reduzida capacidade de processamento e memória, devido principalmente ao seu tamanho, e a capacidade de acesso à redes sem fio.

Ambientes IoT habilitam uma representação virtual de objetos físicos, permitindo-lhes trocar informações contextuais entre si para coordenar ações, obtendo respostas mais rápidas e precisas em relação às mudanças de seus ambientes e, deste modo, tornando-se aptos a utilizar seus recursos de forma mais eficiente [Lara et al. 2020]. Dispositivos IoT são muito mais limitados em termos de recursos do que outras máquinas conectadas à Internet, especialmente em termos de memória. Estes dispositivos normalmente dispõem de alguns KiB de memória [Sabri et al. 2017].

2.2. DTLS

O DTLS é um protocolo responsável por prover segurança em comunicações baseadas em datagramas. Ele foi introduzido por Nagendra Modadugu e Eric Rescorla em 2014 e foi padronizado na RFC 4347 (*Request for Comments*) [Rescorla and Modadugu 2006]. O DTLS utiliza o UDP (*User Datagram Protocol*) como camada de transporte e é fortemente baseado no TLS (*Transport Layer Security*), que por sua vez utiliza o TCP (*Transmission Control Protocol*). Ele reutiliza a infraestrutura de protocolo presente no TLS e provê algumas funcionalidades adicionais para habilitar suporte ao UDP [Roepke et al. 2016].

Este protocolo é fortemente baseado no protocolo HTTP (*Hyper Text Transfer Protocol*), utilizando também a arquitetura REST (*Representational State Transfer*), que usa os métodos *GET*, *POST*, *PUT* e *DELETE* para manter uma interface uniforme e padronizada. Estes métodos servem para buscar, inserir, atualizar e deletar dados de aplicação em servidores remotos. Devido à estas características, o CoAP é facilmente conversível para o HTTP, permitindo que *gateways* intermediários façam a conversão entre ambos os protocolos, conforme a necessidade. A Figura 2 ilustra a arquitetura do CoAP.



Fonte: [Glaroudis et al. 2020]

2.4. MQTT

O MQTT é um protocolo de aplicação construído na arquitetura *publish/subscribe*, projetado para comunicações M2M em redes com recursos limitados. Ele é um protocolo binário e normalmente requer um cabeçalho fixo de 2 bytes, com *payload* máximo de até 256MB [Naik 2017]. O MQTT utiliza o TCP como camada de transporte e, consequentemente, utiliza o TLS para prover segurança.

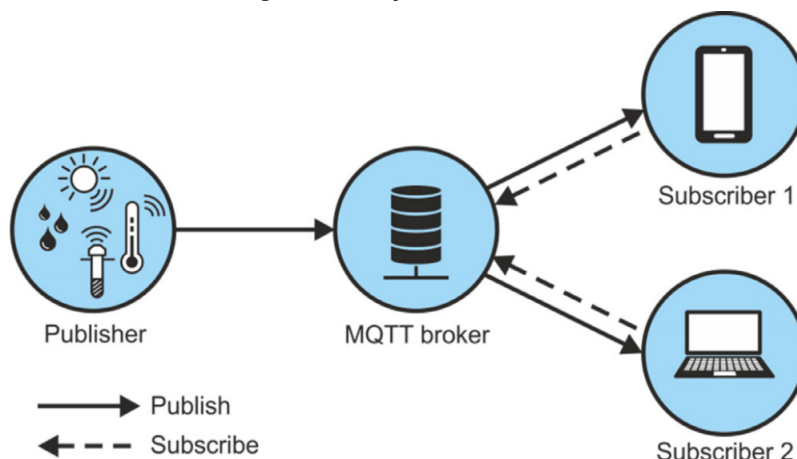
A arquitetura do protocolo MQTT é composta por dois tipos de entidades: cliente e servidor (*broker*). A comunicação ocorre entre cliente e *broker*. O papel do *broker* é realizar o gerenciamento das mensagens entre clientes.

Diferentemente do modelo *request/response*, muito utilizado em arquiteturas cliente/servidor, o modelo *publish/subscribe* trabalha de maneira um pouco diferente. Para enviar dados, um dispositivo cliente deve enviar uma mensagem *publish* para o *broker*, informando o tópico da mensagem e seu conteúdo. O tópico da mensagem é uma *string* que referencia um determinado canal de mensagens. O sistema de tópicos utiliza um mecanismo de hierarquia, onde *strings* separadas por barras (/) indicam os níveis e subníveis do tópico [Dizdarević et al. 2019].

Caso algum dispositivo cliente queira receber dados de um tópico específico, ele deve enviar uma mensagem *subscribe* ao *broker*, informando o tópico desejado. Neste

caso, sempre que um cliente enviar dados referentes à este tópico, o *broker* encaminhará a mensagem imediatamente para o dispositivo que realizou o *subscribe*. Por utilizar o TCP, a comunicação entre cliente e *broker* é orientada a conexão. A Figura 3 ilustra arquitetura do protocolo MQTT.

Figura 3. Arquitetura MQTT



Fonte: GLAROUDIS2020

3. Trabalhos correlatos

No trabalho de [Westphall et al. 2020], os autores desenvolveram uma aplicação IoT para a área de agricultura com a finalidade de mensurar a quantidade de pesticida aplicado em plantações, e também coletar informações sobre a temperatura em pontos específicos do terreno, utilizando sensores de temperatura e GPS (*Global Positioning System*). Esta aplicação faz parte de um ambiente *fog* e *cloud*, e utilizou os protocolos CoAP e DTLS.

No trabalho de [Kaewwongsri and Silanon 2020], os autores descreveram a implementação de um protótipo de uma estação climática, a qual é capaz de monitorar e coletar dados climáticos em tempo real. Esta estação utilizou uma placa Arduino e alguns sensores para mensurar variáveis como temperatura, umidade, velocidade e direção do vento, gás ozônio, pressão atmosférica e dados de chuva.

Em seu trabalho, [Çavuşoğlu et al. 2020] avaliaram o desempenho do protocolo CoAP, analisando parâmetros como tempos de atraso, taxas de vazão e consumo de energia por mensagem. Estes critérios foram analisados utilizando os métodos *GET*, *POST* e *PUT* do protocolo CoAP.

Em seu trabalho, [Tiburski et al. 2019] avaliaram o protocolo CoAP com DTLS em comunicações *fog-to-fog* operando em RANs (*Radio Access Networks*) com diferentes taxas de perda de pacotes e latência. Seus experimentos levaram em conta parâmetros de desempenho, *overhead* e problemas de *handshake*.

4. Desenvolvimento

4.1. Contexto

Baja SAE é uma competição conduzida pela *Society of Automotive Engineers International* (SAE *International*), na qual equipes de universidades de todo o mundo projetam

e constroem pequenos carros *off-road*, em que todos os carros possuem motores com as mesmas especificações. O objetivo destas competições é projetar, construir e conduzir veículos *off-road* que possam suportar os elementos mais adversos de terrenos acidentados [SAE International 2020].

Durante estas competições, os veículos construídos por estas equipes passam por diversos testes de segurança e provas dinâmicas para avaliar aspectos dos veículos como suspensão, tração, aceleração e retomada, entre outros. No fim da competição há uma prova de resistência, conhecida como Enduro, que possui quatro horas de duração.

Durante estas provas de resistência, é muito comum o superaquecimento do câmbio CVT (*Continuously Variable Transmission*), presente em alguns veículos. Dentro deste câmbio existe uma correia que tende a sofrer danos quando a temperatura ultrapassa um certo limiar [Sano 2013], [Hardy et al. 2019]. O monitoramento da temperatura interna do câmbio CVT em tempo real é importante para evitar complicações durante a corrida, alertando o piloto e a equipe antecipadamente para efetuar a troca da correia.

4.2. Modelo

O modelo deste trabalho consiste basicamente em três aplicações principais: uma aplicação para realizar a coleta de dados dos sensores; uma aplicação cliente, implementando um cliente CoAP e DTLS; e uma aplicação para o servidor, que recebe os dados provenientes do cliente, e então permite o acesso destes dados à demais aplicações. A maneira em que estas aplicações foram organizadas é apresentada na Figura 4.



Fonte: Elaborado pelo autor (2020)

A coleta de dados dos sensores é realizada por uma aplicação separada, utilizando um dispositivo ESP32 WROOM-32 DevKit V1 e um sensor BMP280. Este sensor é capaz de obter informações de temperatura e pressão do ambiente. Esta aplicação utilizou o *framework* Arduino para oferecer compatibilidade com a biblioteca Adafruit_BMP280, responsável por ler os dados provenientes do sensor BMP280.

O fluxo de execução desta aplicação consiste em um *loop* que faz a leitura dos dados do sensor em intervalos de tempo regulares. Assim que os dados são lidos, eles são redirecionados à aplicação cliente. Como ambas as aplicações estão em dispositivos distintos, foi estabelecido um canal de comunicação entre elas por meio do barramento UART (*Universal Asynchronous Receiver Transmitter*). Deste modo, quando os dados

são lidos do sensor, eles são imediatamente escritos no barramento UART, permitindo que a aplicação cliente possa lê-los.

A aplicação cliente implementou um cliente CoAP com DTLS, estabelecendo um canal de comunicação seguro com o servidor. Para isto, o exemplo `coap_client`, disponibilizado pelo *framework* ESP-IDF, foi utilizado como base. Esta aplicação utilizou somente um dispositivo ESP32 WROOM-32 DevKit V1. A transmissão dos dados para o servidor é feita via rede *Wi-Fi*.

A aplicação do servidor foi desenvolvida tendo como base o exemplo `coap-server`, disponibilizado pela biblioteca `libcoap`, utilizando a linguagem de programação C. Este exemplo, contudo, não implementava um recurso com o método *POST*, necessário para o contexto deste trabalho. Para tanto, foi incluído um método *handler* para o recurso `/sensor`, criado para receber os dados provenientes da aplicação cliente. Este método *handler* foi associado ao método *POST*. Isto significa que, sempre que o servidor receber uma mensagem *POST* direcionada ao recurso `/sensor`, a função *handler* será executada.

4.3. Resultados referentes ao tempo de processamento

O objetivo desta seção foi apresentar uma comparação quantitativa da utilização de diferentes *cipher suites* em um ambiente típico IoT, utilizando um dispositivo restrito como cliente e dispositivos mais robustos atuando como servidores. Para destacar o impacto de cada *cipher suite* no lado do servidor, testes adicionais foram realizados utilizando dois dispositivos de diferentes capacidades atuando como servidor: uma Raspberry Pi 3 e um *notebook*.

Os resultados apresentados nesta seção foram obtidos a partir da aplicação desenvolvida neste trabalho, calculando o tempo de processamento das mensagens trocadas de ambos os lados, alternando as *cipher suites* utilizadas. A lista de *cipher suites* é apresentada na Tabela 1, em que a primeira coluna indica o nome da *cipher suite* e a segunda coluna indica o algoritmo utilizado para o processo de troca de chaves.

Tabela 1. Cipher suites analisadas

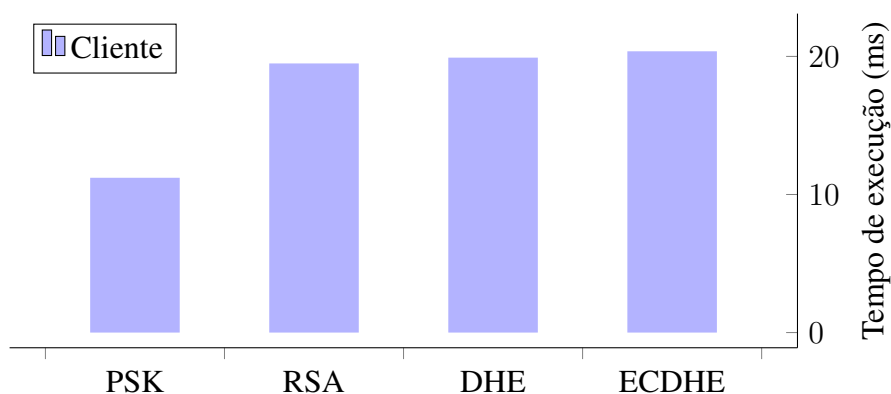
<i>Cipher suite</i>	Algoritmo
TLS_RSA_WITH_AES_256_GCM_SHA384	RSA
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	DHE
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	ECDHE
TLS_PSK_WITH_AES_256_GCM_SHA384	PSK

Fonte: Elaborado pelo autor (2020)

As variáveis mensuradas consistem nos tempos de processamento do cliente e do servidor, tempos de rede e tempos totais, em que cada registro equivale à uma mensagem trocada entre os participantes. Os tempos de processamento do cliente e servidor e os tempos totais foram calculados a partir de arquivos de *logs* gerados pelas aplicações, enquanto que o tempo de rede foi obtido subtraindo os tempos de processamento do tempo total. Para cada *cipher suite* foram coletadas aproximadamente 1000 amostras, onde registros *outliers* foram excluídos do conjunto de dados.

A Figura 5 apresenta os tempos médios de processamento de cada *cipher suite* calculados no lado do cliente. Como pode ser observado, a *cipher suite* PSK (*Pre-Shared Key*) apresentou o menor tempo médio de processamento dentre as demais *cipher suites* analisadas, atingindo praticamente a metade do tempo necessário por seus pares. Este fato é em parte explicado pelo uso de chaves simples para garantir a confidencialidade da comunicação, não sendo necessária a utilização de certificados para este propósito. As demais *cipher suites* não mostraram diferenças significativas entre seus tempos de processamento, permanecendo no patamar dos 20 ms médios por execução.

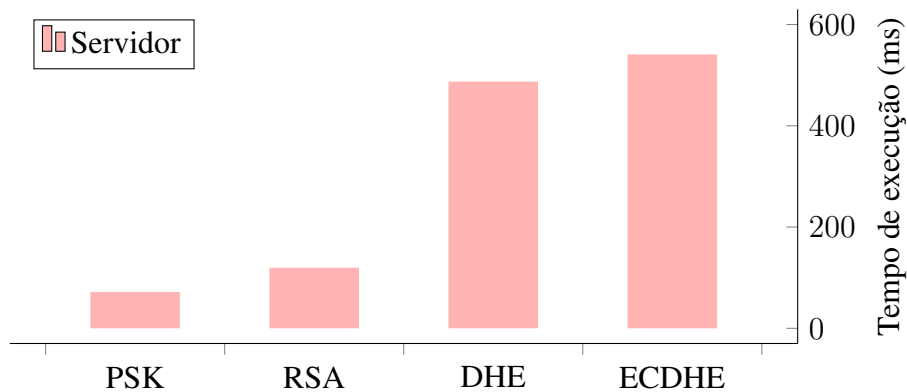
Figura 5. Tempo médio de processamento de diferentes *cipher suites* no cliente



Fonte: Elaborado pelo autor (2020)

A Figura 6 apresenta os tempos médios de processamento das *cipher suites* no lado do servidor, utilizando o *notebook* como recurso computacional. Neste gráfico é possível observar que, assim como no lado do cliente, a utilização de PSK é a mais veloz dentre todas as outras *cipher suites*. Logo após vem o RSA (*Rivest-Shamir-Adleman*), com tempos levemente maiores, exibindo um aumento de 40% em relação ao PSK. Por fim, temos os algoritmos DHE (*Diffie-Hellman Ephemeral*) e ECDHE (*Elliptic Curve Diffie-Hellman Ephemeral*), com tempos de execução significativamente maiores, atingindo médias de 487,83 ms e 540,96 ms, respectivamente.

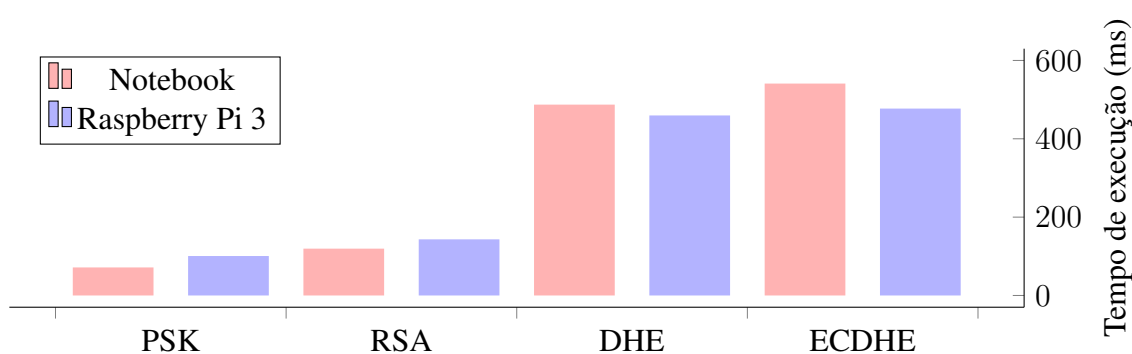
Figura 6. Tempo médio de processamento de diferentes *cipher suites* no lado do servidor



Fonte: Elaborado pelo autor (2020)

Os tempos médios de execução do lado do servidor, considerando sua execução em dois dispositivos com diferentes especificações, é exibida na Figura 7. De modo geral, os tempos de processamento não mostraram diferenças significativas entre os dois dispositivos. No entanto, a Raspberry obteve médias maiores para o PSK e o RSA, apresentando aumentos de 40,72% e 19,84%, respectivamente, em relação ao *notebook*. No caso dos algoritmos DHE e ECDHE o cenário foi o inverso, em que os tempos médios da Raspberry sofreram reduções de 5,68% e 11,80%, respectivamente, em relação aos tempos obtidos no *notebook* para ambos os algoritmos.

Figura 7. Tempo médio de processamento de diferentes *cipher suites* no servidor



Fonte: Elaborado pelo autor (2020)

4.4. Resultados referentes ao consumo energético

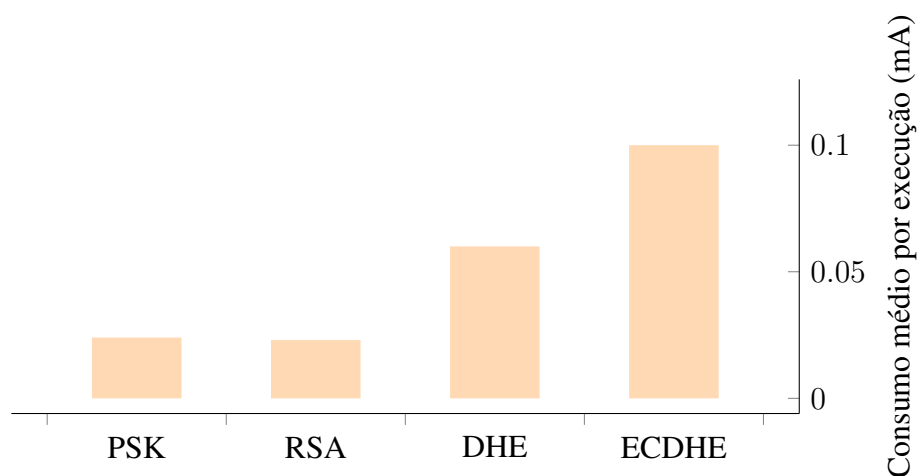
Para mensurar o consumo de energia, experimentos foram realizados utilizando as *cipher suites* apresentadas nesta seção. Para isto, uma bateria com capacidade nominal de 3200 mAh foi utilizada como fonte de alimentação da aplicação. Os experimentos compreenderam a execução do processo de *handshake* do DTLS seguido de uma troca de mensagens CoAP até a drenagem completa da bateria.

O código-fonte da aplicação cliente foi levemente modificado, removendo o processo de escuta ativa no barramento UART com o objetivo de obter maior precisão na mensuração do consumo energético de cada *cipher suite*. Estas alterações removeram a espera entre os envios, de modo que o cliente possa iniciar um novo envio assim que o servidor envia a resposta da mensagem anterior.

A duração de cada teste (tempo decorrido entre o início do experimento até a drenagem total da bateria) e número de execuções (estabelecimento de uma sessão DTLS seguida do envio de uma mensagem) foram obtidos através de arquivos de *logs* gerados pela aplicação do servidor. O consumo médio por execução (mA) foi obtido dividindo-se a capacidade nominal da bateria pelo número de execuções. O consumo médio por hora (mAh) foi obtido dividindo-se a capacidade nominal da bateria pela duração de cada experimento (em horas). A Figura 8 apresenta um gráfico com os dados de consumo médio por execução de cada um dos algoritmos analisados.

É importante destacar que, devido ao método utilizado para realizar a medição, os valores podem apresentar um certo grau de imprecisão e servem apenas para fins de

Figura 8. Consumo médio por execução (mA) de diferentes *cipher suites*



Fonte: Elaborado pelo autor (2020)

comparação entre as *cipher suites*. Ainda assim, os resultados observados foram influenciados pelo consumo geral do dispositivo, possivelmente envolvendo outros módulos que consumiram parte da energia ao longo dos experimentos.

Os resultados indicaram que os algoritmos PSK e RSA para o processo de troca de chaves são os mais econômicos, consumindo aproximadamente 0,02 mA por execução. Por outro lado, os algoritmos DHE e ECDHE apresentaram um maior consumo energético, obtendo valores de 0,055 mA e 0,1 mA por execução, resultados significativamente maiores que o de seus pares.

5. Conclusão

Este trabalho realizou pesquisas teóricas sobre dois dos principais protocolos utilizados em ambientes IoT, CoAP e MQTT, trazendo uma análise quantitativa acerca do desempenho do protocolo CoAP em ambientes reais, através de experimentações e trabalhos do estado da arte. Com base nestas pesquisas, ficou evidente a necessidade de protocolos mais leves para utilização em ambientes restritos, reduzindo *overhead* de processamento e consumo de energia. Além disso, estes protocolos devem ser utilizados em conjunto com protocolos de segurança, como o TLS, amplamente utilizado hoje em dia. No entanto, devido ao seu maior *overhead*, por utilizar pacotes TCP, o TLS não é adaptável a ambientes IoT, surgindo a necessidade de protocolos como o DTLS, que utilizam meios mais eficientes para a entrega de pacotes, como o UDP.

Uma aplicação IoT foi desenvolvida neste trabalho, utilizando implementações dos protocolos CoAP e DTLS em dispositivos ESP32. Esta aplicação teve como objetivo a validação de um ambiente utilizando estes dois protocolos no contexto de telemetria veicular para processamento de dados em tempo real. Testes foram realizados nesta aplicação para mensurar os tempos de processamento e o consumo energético de diferentes *cipher suites*, incluindo *cipher suites* que utilizam certificados (PKI) e *cipher suites* de senhas pré-compartilhadas (PSK). Os resultados destes experimentos mostraram que *cipher suites* que usam PSK são significativamente mais rápidas e mais econômicas do que *cipher suites* que utilizam PKI.

Este trabalho forneceu detalhes de implementação das bibliotecas utilizadas pela aplicação, no intuito de facilitar a busca por informações de futuros trabalhos que foquem no desenvolvimento deste dois protocolos.

Trabalhos futuros podem realizar mensurações quantitativas do MQTT, ou até mesmo de variações mais eficientes que estão sendo propostas, como MQTT-SN, que faz uso do UDP em vez do TCP, e também do CoAP com DTLS e 6LoWPAN, para ganhos de eficiência.

Referências

- Dizdarević, J., Carpio, F., Jukan, A., and Masip-Bruin, X. (2019). A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration. *ACM Comput. Surv.*, 51(6).
- Glaroudis, D., Iossifides, A., and Chatzimisios, P. (2020). Survey, comparison and research challenges of iot application protocols for smart farming. *Computer Networks*, 168:107037.
- Hardy, A., Bernick, J., Capdevila, N., and Perry, T. (2019). Electronic cvt - controls final design report.
- Harrington, T., Bowman, D., Johnson, W., Benner, R., Capener, R., and Han, H. (2004). Vehicle tag used for transmitting vehicle telemetry data. US Patent App. 10/855,871.
- Kaewwongsri, K. and Silanon, K. (2020). Design and implement of a weather monitoring station using coap on nb-iot network. In *2020 17th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, pages 230–233.
- Kothmayr, T., Schmitt, C., Hu, W., Brünig, M., and Carle, G. (2013). Dtls based security and two-way authentication for the internet of things. *Ad Hoc Networks*, 11(8):2710 – 2723.
- Lara, E., Aguilar, L., Sanchez, M. A., and García, J. A. (2020). Lightweight authentication protocol for m2m communications of resource-constrained devices in industrial internet of things. *Sensors 2020*, pages 468–473.
- Naik, N. (2017). Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. In *2017 IEEE International Systems Engineering Symposium (ISSE)*, pages 1–7.
- Rahman, R. A. and Shah, B. (2016). Security analysis of iot protocols: A focus in coap. In *2016 3rd MEC International Conference on Big Data and Smart City (ICBDSC)*, pages 1–7.
- Ray, P. P. (2016). A survey of iot cloud platforms. *Future Computing and Informatics Journal*, 1(1):35 – 46.
- Rescorla, E. and Modadugu, N. (2006). Datagram transport layer security.
- Rescorla, E. and Modadugu, N. (2012). Datagram transport layer security version 1.2.
- Roepke, R., Thraem, T., Wagener, J., and Wiesmaier, A. (2016). A survey on protocols securing the internet of things: Dtls, ipsec and iee 802.11 i.

- Sabri, C., Kriaa, L., and Azzouz, S. L. (2017). Comparison of iot constrained devices operating systems: A survey. In *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*, pages 369–375.
- SAE International (c2020). Baja sae.
- Sano, A. (2013). Uma análise da eficiência de uma transmissão cvt.
- Shelby, Z., Hartke, K., and Bormann, C. (2014). The constrained application protocol (coap).
- Thangavel, D., Ma, X., Valera, A., Tan, H.-X., and Tan, C. K.-Y. (2014). Performance evaluation of mqtt and coap via a common middleware. In *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pages 1–6.
- Tiburski, R. T., de Matos, E., and Hessel, F. (2019). Evaluating the dtls protocol from coap in fog-to-fog communications. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 90–905.
- Westphall, J., Loffi, L., Westphall, C. M., and Martina, J. E. (2020). Coap + dtls: A comprehensive overview of cryptographic performance on an iot scenario. In *2020 IEEE Sensors Applications Symposium (SAS)*, pages 1–6.
- Çavuşoğlu, , EBLEME, M., Bayilmiş, C., and Kucuk, K. (2020). Coap and its performance evaluation. *Sakarya University Journal of Science*, pages 78–85.