PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Sistema para coleta de dados de sensores em rede

Pedro Augusto Bruno Marinho de Souza

Projeto Final de Graduação

Centro Técnico Científico - CTC

Departamento de Informática

Curso de Graduação em Engenharia da Computação

Orientadora: Prof.ª Noemi Rodriguez

Rio de Janeiro

Dezembro de 2018

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Pedro Augusto Bruno Marinho de Souza

Sistema para coleta de dados de sensores em rede

Relatório de Projeto Final, apresentado como requisito parcial para obtenção do título de bacharel em Engenharia da Computação

Orientadora: Profª. Noemi Rodriguez

Rio de Janeiro

Dezembro de 2018

Resumo

De Souza, Pedro. Rodriguez, Noemi. Sistema para coleta de dados de sensores em rede. Rio de Janeiro, 2018. 24p. Relatório de Projeto Final II – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Nesse projeto implementamos um protocolo de rede para comunicação entre ESP8266 em uma área com baixa cobertura de Internet. Utilizamos o firmware Lua disponível para a plataforma ESP8266.

Palavras-chave

Programação orientada a eventos. Comunicação em linha. Autonomia de bateria. Criptografia. Nuvem.

Abstract

De Souza, Pedro. Rodriguez, Noemi. System for collecting sensors data in network. Rio de Janeiro, 2018. 24p. Report of Final Project II – Departament of Informatics, Pontifical Catholic University of Rio de Janeiro.

In this project, we implemented a network protocol for communication between ESP8266 in a area with low Internet coverage. We used the Lua firmware available for ESP8266 plataform

Keywords

Event driven programming. Line communication. Battery autonomy. Criptografy. Cloud.

Sumário

1.Introdução 5
1.1 Motivação 5
1.2 Definição do Problema 6
2.Proposta e Objetivos do trabalho 6
3.Tecnologias utilizadas 7
4.Metodologia 7
5.Atividades realizadas 8
5.1 Conhecimentos preliminares 8
5.2 Estudos realizados 8
5.3 Desenvolvimento 11
6. SDK ESP8266 15
6.1 Módulos 16
7. Implementação 17
7.1 Nó final 17
7.2 Nós centrais 19
7.3 Nó Inicial 21
8. Consumo 22
9. Considerações Finais 23

1.Introdução

Com o avanço da tecnologia, o uso de sistemas embarcados vem ganhando utilidade e importância no dia a dia. A Internet das Coisas (IoT), vem se tornando um importante campo da tecnologia no mundo moderno e os sistemas embarcados possuem grande importância nisso.

Nesse trabalho desenvolvemos uma aplicação para monitoramento de sensores utilizando dispositivos em uma área sem cobertura completa de Internet. Para fazer a informação chegar à internet, desenvolvemos um protocolo de comunicação onde partindo do princípio que apenas um desses dispositivos, chamado de nó inicial, tem cobertura de internet. Os outros nós se comunicam e enviam os dados para esse nó inicial que faz o envio de todos esses dados para o servidor.

1.1 Motivação

Na área da Internet das Coisas, uma atividade principal é fazer com que microcontroladores leiam valores de sensores e armazene esses valores em um servidor para, posteriormente, realizar uma análise em cima desses dados. Contudo, nem sempre esses microcontroladores tem acesso a internet para poder enviar esses dados para um servidor.

A motivação para realização deste trabalho consiste em explorar essa dificuldade de comunicação existente, as funcionalidades existentes nesses microcontroladores e implementar uma aplicação que resolva esse problema na comunicação em uma área sem cobertura total de internet.

Além disso, outra motivação importante foi a viabilidade de se manter essa rede através de uma bateria, com isso, exploramos as funcionalidades que o ambiente do ESP8266 nos fornece no quesito de economia de energia para maximizar a eficiências dessas baterias.

1.2 Definição do Problema

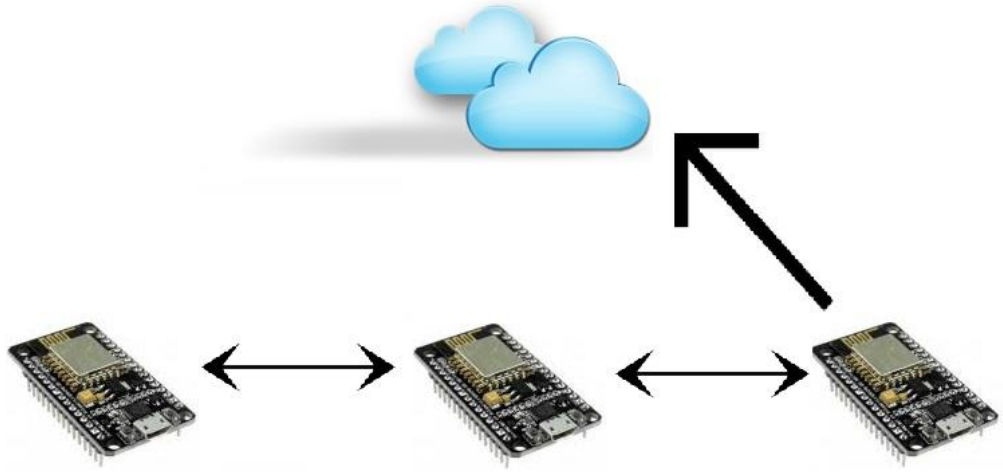

Figura 1- Esquema da Rede

Desenvolvemos neste trabalho uma aplicação onde, em uma rede de microcontroladores, apenas um terá acesso a internet para enviar os valores coletados pelos outros a um servidor na nuvem, conforme mostra a figura 1.

O desafio deste projeto consistiu em realizar uma comunicação confiável nesta rede e enviar os dados coletados dos sensores de umidade para este servidor. Os dados, posteriormente, poderão ser analisados em um aplicativo celular por algum usuário encarregado de realizar esta análise sobre estes dados.

2.Proposta e Objetivos do trabalho

Este trabalho consiste em desenvolver três partes principais, onde o desenvolvimento de cada uma será independente da outra.

Primeiramente, desenvolveremos uma aplicação que monitore a umidade, que será lida de sensores de umidade, através de uma rede de NodeMCUs[1], que são placas da família ESP8266[2], conectados a sensores de umidade. Nesta rede apenas um nó (que será chamado de nó principal), terá acesso a internet, com isso, ele será o encarregado de enviar os valores dos sensores lidos pelos outros nós ao servidor. Esse nó faz a função de um gateway, onde realiza a comunicação da nossa rede e de seu protocolo com a rede pública da internet.

[1] http://www.nodemcu.com/index_en.html
[2] https://en.wikipedia.org/wiki/ESP8266

Depois, desenvolveremos um servidor, que fornecerá uma API de comunicação para que o nó principal envie, através desta API, os valores ao servidor. Esses valores serão enviados para o servidor em intervalos de tempo pré-definidos. Esses valores serão enviados no formato JSON para o servidor.

Por fim, desenvolveremos um aplicativo mobile para usuários, que serão responsáveis por esse monitoramento, observarem o histórico das medições, utilizando um aplicativo celular que se comunicará com este servidor, através da mesma API mencionada anteriormente, e este fornecerá os dados para o celular dos usuários.

3.Tecnologias utilizadas

Dentre as diversas opções disponíveis no vasto mundo de sistemas embarcados, a placa que escolhemos para este projeto foi o NodeMCU, da família ESP8266. Para o servidor que armazenará os dados coletados, utilizamos o banco de dados MongoDB e NodeJS para desenvolvimento de uma API de comunicação com o aplicativo. O aplicativo de celular foi desenvolvido usando o framework Ionic.

Com relação ao desenvolvimento para o microcontrolador NodeMCU, escolhemos utilizar o SDK baseado na linguaguem de programação Lua. Inicialmente, tentamos desenvolver utilizando a biblioteca para ESP8266 da IDE do Arduino, baseada na linguagem de programação C, mas devido a dificuldades com o módulo HTTP para comunicação entre os nós, optamos pelo uso do firmware Lua.

4.Metodologia

O desenvolvimento do projeto foi dividido em duas partes: primeiro foi feito um estudo aprofundado do problema a ser atacado e, posteriormente, a implementação desse problema.

O desenvolvimento do projeto foi interativo: primeiro era desenvolvido um módulo, como, por exemplo, comunicação entre dois nós. A partir do momento que este módulo estivesse funcional, ele era unificado ao resto do projeto. Esse processo contribuiu para a melhor solução de bugs que poderiam acontecer em cada módulo, pois, como cada funcionalidade era desenvolvida separadamente

dentro de um módulo, foi possível encontrar aonde esses bugs ocorriam mais rapidamente.

5.Atividades realizadas

Esta seção apresenta, de forma geral, todas as atividades que foram executadas durante a realização deste projeto. Essas atividades são: estudos realizados, conhecimentos preliminares, testes efetuados e a metodologia utilizada no desenvolvimento do projeto.

5.1 Conhecimentos preliminares

O aluno já tinha conhecimento prévio no desenvolvimento de aplicativos em Ionic, devido à realização de estágio.

Com relação ao servidor, o aluno já possuía conhecimento em NodeJs e MongoDB, contudo, esse conhecimento foi adquirido ao longo de 2015, sendo necessário, além de um reforço, um entendimento de como integrar essas duas tecnologias. Com isso, foi empreendido um estudo inicial focado nessas duas tecnologias.

5.2 Estudos realizados

Aqui serão apresentados os estudos realizados para desenvolver o projeto. Entre esses estudos estão as linguagens que precisarem serem aprendidas, conhecimentos técnicos sobre o hardware, etc.

5.2.1 Modos de operação Wifi

A biblioteca para ESP8266 foi desenvolvida em cima do SDK do ESP8266. O NodeMCU pode operar em três modos Wifi:

- Soft Access Point (AP): um access point é um dispositivo que fornece, para outros dispositivos, uma rede Wifi onde eles podem se conectar. Na família ESP8266, um access point é denomidado soft access point e o máximo número de dispositivos conectados a ele é cinco.

- Station (STA): esse modo é usado para conectar o NodeMCU a uma rede Wifi estabelecida por um access point.
- "Multi Modo": o NodeMCU fornece um modo de operação onde ele pode ser configurado como station e access point ao mesmo tempo, cabendo ao desenvolvedor definir qual modo estará sendo usado em cada parte do código.

No projeto, optamos por utilizar os modos access point e station separadamente ao invés de usar apenas o “multi modo”. Essa escolha se deu pelo fato da placa se comportar melhor (menos bugs) dessa maneira.

5.2.2 Consumo de energia

Uma grande preocupação durante a realização do projeto foi com o consumo de energia de cada nó. Seria inviável manter cada nó em pleno funcionamento durante todo o tempo, visto que geraria uma ociosidade muito grande. Para resolver esse problema, utilizamos a solução de deixar cada nó em modo sleep e, somente quando for o momento de enviar/receber dados, fazer com que o nó acorde e realize as operações. As placas da família ESP8266 possuem três modos de sleep, são eles:

- Modem Sleep: somente o sinal de wifi é desligado. É útil quando não se deseja utilizar sinal wifi, mas é necessário continuar realizando outras operações, como controlar sensores e atuadores, emitir sinais, etc.
- Light Sleep: além do sinal wifi, o clock interno da CPU também é desligado e a CPU fica pendente (idle). Este modo é útil quando desejamos manter a CPU pendente, mas acordá-la a partir de um sinal (HIGH/LOW) para continuar a execução do programa.
- Deep Sleep: sinal wifi, clock interno e CPU são desligados. Apenas o RTC (para medida de tempo) é mantido ligado. Este é o modo que menos consome energia, visto que praticamente toda a placa fica desligado. É útil quando temos situação onde precisamos realizar uma operação, dormir por um determinado período de tempo (medido em microssegundos) e repetir esse processo indefinidamente. Neste modo, após realizar a operação e dormir pelo tempo necessário, a placa emite um sinal, através de uma interrupção, na porta GPIO16 e utilizamos esse sinal na porta RST (Reset) para reinicializar a placa.

Como, na nossa aplicação, durante o tempo dormindo não é realizada nenhuma atividade, optamos por utilizar o deep sleep para obter uma maior economia de bateria.

5.2.3 Rede Mesh

Uma rede mesh é uma topologia de rede onde os nós se conectam dinamicamente e de forma não hierárquica com outros nós para criar uma forma eficiente de roteamento de dados entre clientes. Por ter organização e configuração dinâmicas, a rede mesh tem menos trabalho de instalação. Para transmitir mensagens, a rede mesh utiliza uma técnica onde a mensagem é propagada em broadcast para os nós e, os que captarem essa mensagem, retransmitem para frente. Essa dinâmica faz com que se tenha vários caminhos possíveis dentro da rede e, com isso, tornando esta rede muito menos suscetível a falhas dos nós.

A biblioteca para ESP8266 na IDE do Arduino oferece um módulo para uso da rede mesh[3]. O algoritmo dessa biblioteca consiste em nomear o SSID do wifi de cada nó com um prefixo pré-definido concatenando com um id único para cada nó. Ao tentar enviar uma mensagem para a rede, o nó lê uma lista de pontos de acessos que sejam acessíveis a ele e tenta se conectar, um por vez, aos que tenham o prefixo conhecido.

Optamos por não utilizar esse módulo por usarmos o firmware Lua, que não possui esse módulo, e por ele não fazer distinção para o nó que está enviando, ou seja, ele manda a mensagem para o primeiro nó disponível que encontra independente de que posição esse nó ocupe na rede.

5.2.4 Criptografia

No desenvolvimento do projeto, utilizamos criptografia para criptografar as mensagens enviadas entre os nós para fornecer uma maior segurança para a aplicação. Existem dois tipos de algoritmos de criptografia, são eles:

- Chave Simétrica: é um método de criptografia onde uma mesma chave é usada para criptografar e decriptar uma mensagem, ou seja, uma

[3] https://github.com/esp8266/Arduino/tree/master/libraries/ESP8266WiFiMesh

mensagem criptografada com uma chave, só poderá ser decriptada pelo receptor, caso ele tenha essa chave. São algoritmos mais simples e, portanto, mais rápidos que os algoritmos de chave assimétrica. Entretanto, por utilizar somente uma única chave, uma grande desvantagem é que somente os responsáveis por enviar e receber mensagens devem possuir acesso à chave. Caso alguém mal-intencionado tenha acesso à chave, pode facilmente descriptografar a mensagem e ter acesso a dados que deveriam ser sigilosos.

- Chave Assimétrica: é um método de criptografia que utiliza duas chaves. Uma chave pública, que, como o nome diz, pode ser divulgada para todos, para criptografar e uma chave privada para descriptografar. Essas duas chaves são matematicamente relacionadas, conforme a lógica do algoritmo. Um emissor utiliza a chave pública do destinatário da mensagem para criptografar a mensagem. Uma vez criptografada, esta mensagem só pode ser descriptografada utilizando a chave privada do destinatário, ou seja, somente ele poderá descriptografar a mensagem. É um algoritmo mais seguro fazer uso de uma chave privada, contudo, por ser mais complexo, o processo de criptografia é muito mais lento que o de criptografica por chave simétrica.

No desenvolvimento desse projeto, utilizamos o algoritmo Advanced Encryption Standard(AES)[4], que é um algoritmo de chave simétrica e é fornecido no módulo crypto na biblioteca do ESP8266 em Lua.

O algoritmo AES não é o ideal pois se um nó for comprometido, todos serão uma vez que só uma chave é utilizada. Por outro lado, como é um algoritmo leve, não é gerada uma carga de processamento muito grande nos nós.

5.3 Desenvolvimento

Aqui será descrito, de forma resumida, tudo o que foi feito de implementação para a rede, para o servidor e para o aplicativo mobile.

5.3.1 Rede

O protocolo consiste em o nó final dar início à “cadeia de comunicação” passando dados para o nó seguinte. Esse nó pega os dados, inclui os dados lidos por ele e passa para o seguinte, assim por diante, supondo que estamos utilizando

[4] https://pt.wikipedia.org/wiki/Advanced_Encryption_Standard

uma topologia em linha. Nós desenvolvemos três códigos para a comunicação na rede de NodeMCU's, tendo sido utilizados três nós para os testes. Foram usados três nós para testes devido a limitação na quantidade de placas disponíveis, quantidade de portas USB no computador e quantidade de protoboards. Os códigos são:

- Nó Final: nó ao extremo da rede, só funcionará no modo Access Point (AP), enviando seu valor para o nó seguinte.
- Nó Inicial: nó que, por garantia, terá conexão com a internet. Receberá o valor lido pelos outros nós, adicionará o seu valor e enviará para o servidor.
- Nó Central: código usado pelos nós do meio da rede (os que não são o nó final e o nó inicial). Alternam entre modo AP, para receber valores dos nós anteriores, e modo estação, para enviar para o nó seguinte.

5.3.2 Banco de Dados

Nós criamos um banco de dados em MongoDB para armazenar os valores lidos dos sensores. Esse banco possui uma coleção chamada 'Value' onde são armazenados os dados lidos de cada sensor. Para essa coleção, são esperados documentos com os seguintes parâmetros:

- _id: campo criado automaticamente pelo MongoDB que serve como um identificador único para cada documento dentro de uma coleção.
- sensor: campo do tipo string, armazena o identificador do sensor que leu o valor do campo 'value'.
- value: campo do tipo numérico, armazena o valor lido pelo sensor ligado ao nó.
- time: campo do tipo numérico, armazena o timestamp de quando o valor do campo 'value' foi lido.

Na figura abaixo, podemos observar um exemplo de documento armazenado nessa coleção:

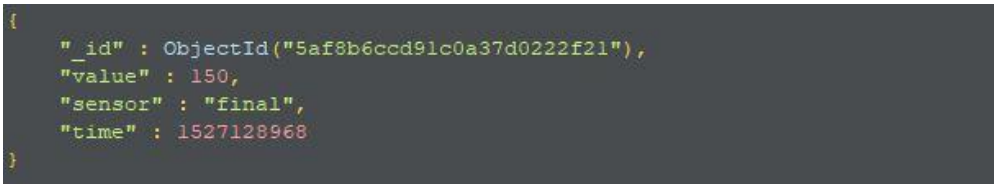

Figura 2 - Exemplo de documento armazenado no MongoDB

5.3.3 API

Para realizar a comunicação do nó principal com o banco de dados, nós criamos uma API em NodeJS. Essa API possui os seguintes endpoints:

- GET /values: retorna todos os dados da coleção 'Values' do banco, ou seja, retorna todos os valores lidos pelos sensores que estão armazenados no banco, assim como o sensor que leu e o timestamp da leitura.

Exemplo de retorno desta chamada:


```
{
    "data": [
        {
            "_id": "5af8b6ccd91c0a37d0222f21",
            "value": 150,
            "sensor": "final",
            "time": 1527128968
        },
        {
            "_id": "5af8b6ccd91c0a37d0222f22",
            "value": 200,
            "sensor": "inicial",
            "time": 1527128968
        },
        {
            "_id": "5af8b769860b4d056442ed12",
            "value": 150,
            "sensor": "final",
            "time": 1527128968
        },
        {
            "_id": "5af8b769860b4d056442ed13",
            "value": 200,
            "sensor": "inicial",
            "time": 1527128968
        }
    ]
}
```


Figura 3 - Exemplo de retorno da chamada GET /values

- GET /sensors: retorna uma lista com todos os sensores que tem dados armazenados no banco

Exemplo de retorno dessa chamada:


```
{
    "data": [
        "final",
        "inicial"
    ]
}
```


Figura 4 - Exemplo de retorno da chamada GET /sensors

- POST /values: envia para o servidor uma lista onde cada posição dessa lista é um objeto.

Exemplo de um body esperado por esse endpoint:

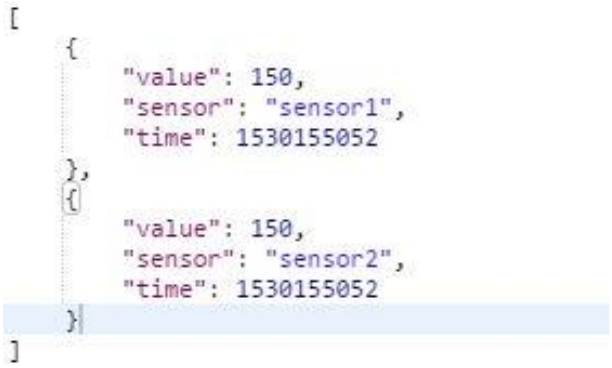

Figura 5 - Exemplo da chamada POST /values

5.3.4 Aplicativo

O aplicativo, desenvolvido utilizando o Ionic Framework, possui uma tela inicial onde há uma listagem dos dados (valor, sensor e data) e um filtro por sensor, onde o usuário pode fazer com que apareça somente os dados de determinado sensor na listagem.

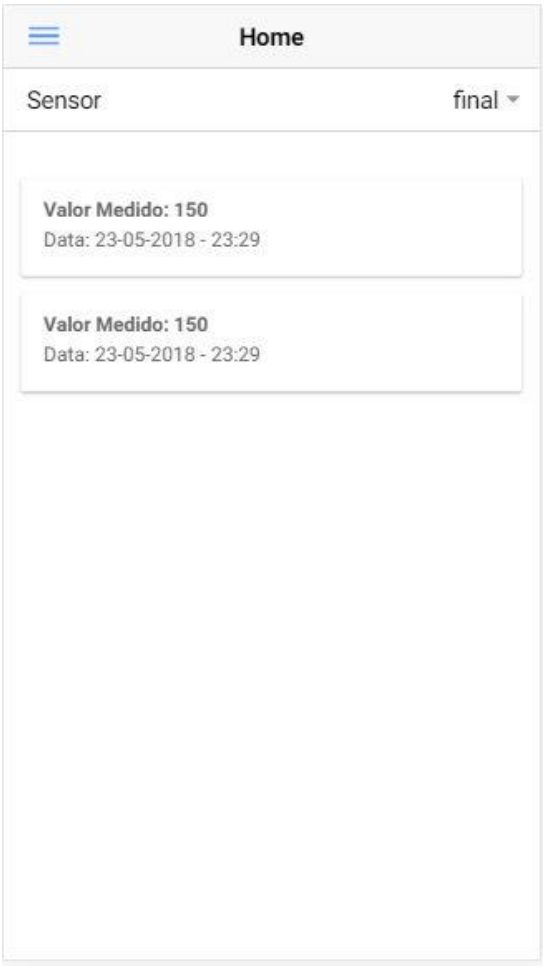

Figura 6 - Tela principal do Aplicativo

6. SDK ESP8266

Nesta seção serão apresentados com mais detalhes os módulos do SDK baseado em Lua para desenvolvimento nos microcontroladores da família ESP8266.

A programação em NodeMCU em Lua é feita de maneira assíncrona e orientada a eventos. Assim, quase todas as funções da biblioteca possuem um parâmetro para indicar callbacks, que serão executadas quando a função tiver sua execução terminada.

No exemplo abaixo, o método on() recebe dois parâmetros, uma string que representa qual evento será tratado e a função response_connection(), que será executada depois que o evento “receive” ocorrer.

```
-- Responde a requisição
local function response_connection(client, request)
  local end_connection =  function()
    print("requisicao")
    data = crypto.decrypt("AES-CBC", key, request):match("(.-)%z*$")
    client:close()
    config_client_mode()
  end
  client:send(buf, end_connection)
end

-- Trata a conexão
local function handle_connection(conn)
  print ("recebeu conexao")
  conn:on("receive", response_connection)
end
```

Figura 7 - Exemplo de callback

6.1 Módulos

A seguir falaremos dos módulos do SDK que utilizamos na implementação do projeto.

6.2.1 WiFi

Um dos módulos mais importantes do projeto. Aqui são disponibilizadas as funções para configurar o modo do ESP8266 (station ou access point), definir o SSID e senha de cada nó e obter o endereço IP de um nó estação na rede em que está conectado.

6.2.2 Net

Módulo com operações TCP e UDP que permite transformar um access point em um servidor e fazê-lo escutar por conexões em determinadas portas. Também permite que um nó em comportamento de station crie uma conexão soquete com determinada porta do servidor e envie uma mensagem para o servidor. Para que se crie uma comunicação soquete com o servidor, é necessário que o nó station esteja conectado no Wifi do servidor (access point).

6.2.3 HTTP

Módulo com operações HTTP usado para a comunicação do nó final com o servidor através da API. Fornece funções para execuções dos métodos GET, POST, PUT e DELETE. Para realizar a comunicação, basta enviar a URL de destino, o header, o dado a ser enviado e um callback que será executado quando a requisição for completada.

6.2.4 Crypto

Módulo que fornece várias funções para uso de algoritmos criptográficos. É possível utilizar encriptação/decriptação utilizando o algoritmo AES. Além disso, fornece funções de hash utilizando os algoritmos MD5, SHA1, SHA256, SHA384 e SHA512.

7. Implementação

A implementação da comunicação entre os nós foi dividida em 3 códigos: o código do nó final, os dos nós do meio e o do nó inicial. As características inerentes a esses nós foram apresentadas na seção 5.3.1.

7.1 Nó final

Para o nó final, o código começa com uma configuração para conectar ao próximo nó da rede. Esse código pode ser visto abaixo.

```
-- Configurando modo cliente
local wificonf = {
  -- verificar ssid e senha
  ssid = "middle_node1",
  pwd = "a12345678",
  -- callback que vai executar qdo ganhar IP na rede:
  got_ip_cb = function (iptable) print ("ip: ".. iptable.IP); send_message(); end,
  save = false
}

wifi.setmode(wifi.STATION)
wifi.sta.config(wificonf)

print("AWAKE")
```

Figura 8 - Código de configuração do nó final

Esse código executa a função wifi.setmode() para colocar esse nó no modo estação. Após isso, é chamada a função wifi.sta.config() para configurar a conexão. Para essa função é passado um objeto com parâmetros necessários, como o SSID e senha do nó seguinte e uma função de callback que será executada quando a conexão for estabelecida.

A função de callback passada, imprime no console, para fins de depuração, o IP adquirido por esse nó na rede e chama a função send_message(), que será responsável por efetivamente enviar os dados desejados para o próximo nó.

```
-- Envia mensagem para o próximo nó
local function send_message()
  cl=net.createConnection(net.TCP, 0)
  cl:on("receive", function(sck, c)
    print(c)
    if file.open("log.txt", "a+") then
      file.writeline("Registra tempo de fim do envio")
      file.close()
    end
  end)

  cl:on("disconnection", function(sck, c) print("vai dormir"); node.dsleep(10000000); end)

  cl:on("connection", function(sck, c)
    dataToSend = getValue()
    encryptedData = crypto.encrypt("AES-CBC", key, dataToSend)
    sck:send(encryptedData)
    if file.open("log.txt", "a+") then
      file.writeline("Registra tempo de inicio do envio")
      file.close()
    end
  end)
  cl:connect(80,"192.168.4.2")
end
```

Figura 9 - Função para enviar mensagem do nó final

A função send_message() cria um soquete TCP e se conecta ao servidor criado pelo nó seguinte utilizando o IP do nó e a porta 80. Além de se conectar, a função se registra a três eventos:

- Conexão (connection): esse evento dispara a callback que foi passada como parâmetro quando a conexão com o servidor criado pelo próximo nó é concluída. Nessa callback, fazemos a criptografia do dado a ser enviado, enviamos o dado e fazemos o registro do tempo de início desse envio.

- Recepção (receive): esse evento ocorre quando recebemos a resposta do envio dos dados. A callback registrada a esse evento imprime a resposta e registra o tempo em que essa resposta foi recebida, assim, temos registrados o tempo de quando enviamos e o tempo de quando obtemos a resposa.
- Desconexão (disconnection): esse evento ocorre quando o nó seguinte fecha a conexão soquete a partir do momento em que ele responde essa conexão. Quando esse evento ocorre, nosso nó final sabe que a comunicação foi concluída e seu dado foi enviado. Com isso, a callback registrada a esse evento dispara a função para colocar o nó em deep sleep.

7.2 Nós centrais

Assim como para o nó final, o código dos nós centrais começa executando uma configuração inicial onde ele define como sendo um ponto de acesso e define as configurações como SSID, senha, IP, máscara e gateway. Essa parte de código pode ser visto na figura abaixo.

```
-- Criando AP
wifi.setmode(wifi.SOFTAP)
-- SSID e Senha do AP
wifi.ap.config({ssid="middle_node1", pwd="a12345678"})
-- Definindo IP no modo AP
wifi.ap.setip({ip="192.168.4.2", netmask="255.255.255.0", gateway="192.168.4.2"})
```

Figura 10 - Configuração de Access Point do nó central

Após fazer essas configurações iniciais, criamos um servidor socket que passa a escutar conexões na porta definida. Quando criamos esse servidor, é passada uma função de callback que registra um novo callback para o evento de recepção (receive). Esse callback, ao ser executado, descripta a mensagem recebida, fecha essa conexão e chama uma função que irá configurar o nó para trabalhar, a partir desse momento, como nó cliente (estação). Na figura 11, podemos ver o código que realiza a criação do servidor.

```
-- Responde a requisição
local function response_connection(client, request)
  local end_connection =  function()
    print("requisicao")
    data = crypto.decrypt("AES-CBC", key, request):match("(.-)%z*$")
    client:close()
    config_client_mode()
  end
  client:send(buf, end_connection)
end

-- Trata a conexão
local function handle_connection(conn)
  print ("recebeu conexao")
  conn:on("receive", response_connection)
end

-- Espera conexão
local function initialize()
  local server = net.createServer(net.TCP)
  if server then
    server:listen(80, "192.168.4.2", handle_connection)
  end
end

initialize()
```

Figura 11 - Código responsável por criar o servidor nos nós centrais

A função de configurar o modo cliente executa as mesmas funções das configurações iniciais do nó final, conecta o nó ao nó seguinte através do SSID e senha e configura um callback que será chamado quando essa conexão for concluída. Através desse callback, são realizadas as mesmas funcionalidades já explicadas para o nó final, um soquete é criado e três eventos (conexão, recebimento e desconexão) são associados a call-backs, que enviam os dados, registram os tempos de requisição e fazem o nó entrar em modo deep sleep.

```lua
-- Configurando modo cliente
local function config_client_mode()
  local wificonf = {
    -- verificar ssid e senha
    ssid = "main_node",
    pwd = "a12345678",
    -- callback que vai executar qdo ganhar IP na rede:
    got_ip_cb = function (iptable) print ("ip: ".. iptable.IP); send_message(); end,
    save = false
  }
  wifi.setmode(wifi.STATION)
  wifi.sta.config(wificonf)
end
```

Figura 12 - Configuração do modo cliente no nó central

```lua
-- Envia mensagem para o próximo nó
local function send_message()
  cl=net.createConnection(net.TCP, 0)
  cl:on("receive", function(sck, c)
    print(c)
    if file.open("log.txt", "a+") then
      file.writeline("Registra tempo de fim do envio")
      file.close()
    end
  end)

  cl:on("disconnection", function(sck, c) print("vai dormir"); node.dsleep(10000000); end)

  cl:on("connection", function(sck, c)
    dataToSend = parseJSONArray(data, getValue())
    encryptedData = crypto.encrypt("AES-CBC", key, dataToSend)
    sck:send(encryptedData)
    if file.open("log.txt", "a+") then
      file.writeline("Registra tempo de inicio do envio")
      file.close()
    end
  end)
  cl:connect(80,"192.168.4.3")
end
```

Figura 13 - Função que envia mensagens no nó central

7.3 Nó Inicial

Para o nó inicial fizemos uma configuração inicial em que o nó é configurado como modo de access point e espera uma conexão do nó anterior. Após receber esta conexão, o nó muda para o modo estação (cliente). Contudo, como esse é o nó com acesso à internet, ao invés de se conectar a um outro nó, ele se conecta ao SSID da rede que tem acesso à internet. Após essa conexão ser estabelecida, é executado um callback que envia os dados recebidos do último nó e o dado coletado do próprio nó para o servidor atráves de uma API

utilizando a biblioteca HTTP do ESP8266. O código pode ser visto na figura abaixo:

```
-- Envia mensagem para o próximo nó
local function send_message()
  http.post("http://polar-dawn-30624.herokuapp.com/values", 'Content-Type: application/json\r\n', parseJSONArray(data, getValue()), cb_send_message)
end
```

Figura 14 - Envio dos dados para o servidor

8. Consumo

Durante o desenvolvimento medimos o consumo de corrente do microcontrolador. Foram observadas as seguintes medições aproximadas:

- Envio de dados (modo estação): 120 mA
- Recebimento de dados (modo ponto de acesso): 50 mA
- Deep Sleep: 10 uA
- Standby: 10 mA

Podemos perceber uma grande vantagem na utilização do deep sleep ao invés de simplesmente manter o nó acordado “para sempre”. Para efeito de comparação, considerando que utilizemos uma bateria de 2000 mAh para alimentar nosso microcontrolador e que o tempo que ele fica em modo estação é o mesmo tempo que ele fica em modo ponto de acesso, temos que no caso em que não utilizamos deep sleep e deixamos o nó acordado indefinidamente, essa bateria duraria aproximadamente 23,5 horas.

No caso em que utilizamos deep sleep, caso considerarmos que o nó fica acordado metade do tempo e dormindo na outra metade, temos que essa mesma bateria duraria 40 horas, ou seja, o dobro do tempo.

Contudo, as considerações que fizemos foram de forma a facilitar as contas. Podemos ver que apenas fazendo com que o nó fique metade do tempo dormindo, a eficiência com relação ao consumo de energia dobrou. Dependendo do tipo de aplicação e da necessidade de se manter os atualizados dados com alta (ou baixa) frequência, podemos ter aplicações onde o nó fica acordado por alguns segundos e dormindo por minutos, o que faria a bateria durar muito mais do que essas 47 horas.

Considerando uma aplicação que não opere o tempo todo utilizando o wifi, ou seja, fique a maior parte do tempo em standby e só utilize o wifi de forma rápida, temos que caso fique acordada indefinidamente, sua autonomia seria de aproximadamente 165 horas. Para essa mesma aplicação agora considerando que ela fique em deep sleep por 2 minutos e opere durante 30 segundos, sua autonomia aumentaria para aproximadamente 1000 horas. O mesmo caso, mas com tempo de sleep de 4 minutos, a autonomia praticamente dobraria, ou seja, 2000 horas.

9. Considerações Finais

Nesse trabalho produzimos três códigos em Lua que representam cada nó da nossa rede. Os esforços realizados se propuseram a validar o uso do ambiente ESP8266 para projetos que façam bastante uso da comunicação por wifi e sua viabilidade no quesito consumação de energia.

Com relação ao ambiente, observamos que, apesar de uma documentação fraca, principalmente para o ambiente utilizando o Arduino IDE, é possível encontrar exemplos de códigos e algumas explicações através da comunidade, contudo a documentação e o ambiente de desenvolvimento não estão tão maduros como no caso da placa Arduino, por exemplo.

No que diz respeito ao consumo, percebemos que há uma grande vantagem em se utilizar os modos de sleep, principalmente o deep sleep, contudo, se a aplicação requerer que o microcontrolador fique acordado por muito tempo utilizando wifi indefinidamente, ficaria inviável mantê-lo através de uma bateria. No nosso exemplo, verificamos que uma aplicação que requer que o ESP8266 fique acordado por metade do tempo, seria necessário trocar de bateria a cada dois dias. Para aplicações onde o tempo dormindo é muito maior que o tempo acordado, ficaria mais viável a utilização do ESP8266. Para aplicações onde o uso do wifi é limitado e fique sua boa parte do tempo acordado sem fazer uso do mesmo, a autonomia da bateria aumenta consideravelmente. Por fim, para uso em um projeto onde a placa será alimentada por algum computador ou outro tipo de fonte, o ESP8266 atenderia tranquilamente, pois não haveria problemas de autonomia de bateria.

Com relação a trabalhos futuros, seria interessante um projeto que estude e implemente detecção de falhas nos nós, ou seja, uma maneira de repor

um nó defeituoso na rede sem haver necessidade de se mexer, seja fisicamente ou através do código e sem reinicialização, dos demais nós. Outro bom projeto seria a implementação de um protocolo de segurança para criptografia de dados mais seguros, utilizando chave assimétrica, certificados, dentre outras funcionalidades.