Trabalho de Conclusão de Curso
Homepage: https://www.upf.br/Iceg/curso/ciencia-da-computacao

ORIGINAL PAPER

Configuração Remota Para a Plataforma Embarcada do Protegemed

Kelvin M. Klann[1] and Marcelo Trindade Rebonatto[1]

[1]Curso de Ciência da Computação , Universidade de Passo Fundo

*106881,rebonatto@upf.br

Abstract

Protegemed is a project to detect microshocks caused by electro medical equipment during surgeries. Shocks as small as 50 milliamperes may result in a cardiac arrest. An embedded device, called "the module", is used for monitoring the outlets of the surgery room. The device constantly gathers data about the electric currents of these outlets and sends it to a remote server through the network. If anything out of the ordinary is detected, the relevant staff is alerted, which can then perform the necessary actions. The surgery room is a controlled environment, but configuring the module requires physical access to it, since the communication happens in a one-way fashion (i.e.: from the module to the server). To end this predicament, a way of configuring the device remotely is proposed, through a new application-layer protocol. The protocol is used over a WebSocket connection and defines the format and semantics of the messages to be exchanged, both of which are based on HTTP. The new protocol is then implemented and used in Protegemed to perform bidirectional communication between a web browser and the embedded device through a WebSocket server. The protocol is defined, implemented and then validated. The same is done for the control and the configuration commands. Afterwards, the device is configured and controlled remotely using the protocol. The protocol itself is defined in a general way, enabling it to be used for message exchanging in other embedded projects.

Keywords: embedded; networking; protegemed; protocol; websocket

Resumo

O Protegemed é um projeto para detectar microchoques causados por equipamentos eletromédicos durante procedimentos cirúrgicos. Choques de apenas 50 miliamperes podem ocasionar uma parada cardíaca. Um dispositivo embarcado, denominado "módulo", é utilizado para monitorar as tomadas de uma sala de cirurgia. O dispositivo coleta dados em tempo real sobre as correntes elétricas das tomadas utilizadas e os envia para um servidor remoto. Caso um valor fora do limite configurado seja detectado, os funcionários responsáveis são alertados, para que possam tomar as medidas cabíveis. A sala de cirurgia é um ambiente controlado, mas, para configurar o dispositivo, é necessário ter acesso físico a ele. Para resolver isso, é proposta uma forma de configurar o dispositivo remotamente, através de um novo protocolo de rede na camada de aplicação. O protocolo é utilizado sobre uma conexão WebSocket e define o formato e a semântica das mensagens a serem enviadas, sendo que ambos são baseados no protocolo HTTP. O protocolo é implementado e utilizado no Protegemed para realizar uma comunicação bi-direcional entre um navegador web e o dispositivo embarcado através de um servidor WebSocket. O protocolo é definido, implementado e depois validado. O mesmo é feito para os comandos de configuração e de controle. Em seguida, o dispositivo é configurado e controlado remotamente. O protocolo em si foi definido de forma genérica, possibilitando que ele seja utilizado para a troca de mensagens em outros projetos de computação embarcada.

Palavras-Chave: embarcado; protegemed; protocolo; rede; websocket

1 Introdução

A plataforma Protegemed busca detectar, em equipamentos eletromédicos (EEMs), variações de energia elétrica que podem por em risco a vida de pacientes durante procedimentos cirúrgicos. Esse risco é inerente ao uso dos EEMs, sendo que uma corrente de apenas 50 μA atravessando o coração pode causar uma parada cardíaca, por exemplo. A detecção é feita utilizando um dispositivo de hardware embarcado nas tomadas elétricas de uma sala de cirurgia. A plataforma possui sensores que medem certos valores da corrente elétrica e os envia para um servidor. O servidor analisa os dados recebidos e alerta em caso de riscos (Spalding et al.; 2009) (Rebonatto; 2015).

Cada EEM possui um perfil de uso de energia diferente, que deve ser considerado ao medir a corrente elétrica. Porém, no momento, o perfil utilizado em cada tomada é fixo para todos os equipamentos conectados, dificultando a medição precisa no uso de diferentes aparelhos nas mesmas tomadas. Nesse caso, é necessário reconfigurar manualmente. Com isso, busca-se possibilitar a medição da corrente, independente da troca de aparelhos. Dessa forma, a plataforma deve detectar cada aparelho conectado e adequar os limites de acordo com cada aparelho.

Atualmente, a comunicação entre a plataforma embarcada e o servidor é feita, de modo geral, de forma unidirecional, enviando informações sobre os EEMs para o servidor. Esse trabalho visa possibilitar a configuração e controle remotos do componente de hardware da plataforma embarcada, de forma que um cliente possa realizar isso através de um navegador web.

Primeiramente, será feita uma breve revisão bibliográfica dos protocolos de rede que já são utilizados no projeto Protegemed. Após, será proposto um novo protocolo na camada de aplicação, que irá definir o formato das mensagens enviadas entre a plataforma embarcada e o servidor. Em seguida, será decorrido sobre os detalhes da implementação. Por fim, serão analisados os resultados.

2 Protegemed

O projeto Protegemed surgiu em 2009, em uma parceria entre os cursos da UPF de Ciência da Computação e de Física, realizada pelos professores Marcelo Trindade Rebonatto e Luiz Eduardo Schardong Spalding, dos respectivos cursos. Inicialmente, era utilizado um computador de mesa convencional dentro do centro cirúrgico, conforme a figura 1. O computador se comunicava através de uma interface serial com um dispositivo que media a corrente elétrica das tomadas da sala, alertando em caso de risco. (Schmitz; 2017) Sobre o estado atual da plataforma:

> Atualmente, o Protegemed encontra-se embarcado no interior do painel de gases da sala 1 do Centro Cirúrgico do Hospital São Vicente de Paulo (HSVP) de Passo Fundo, RS. Neste painel são instaladas 6 tomadas a serem monitoradas, cada tomada necessita de dois canais analógicos, um para o sinal de corrente diferencial e outro para a corrente de alimentação, totalizando 12 canais analógicos por painel (dos Santos; 2017).

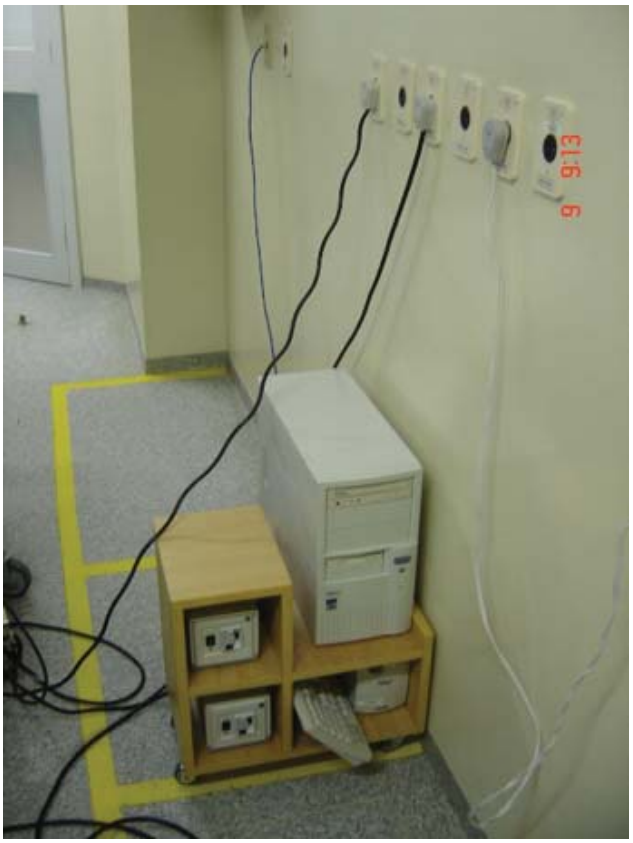
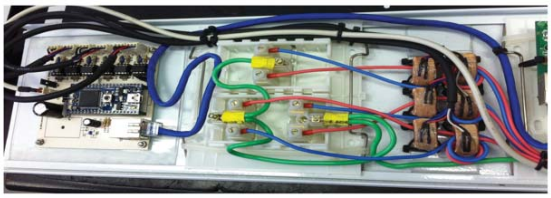


Figure 1: Primeira versão do Protegemed dentro de uma sala de cirurgia (2009)

(a)



(b)

Figure 2: Módulo embarcado dentro do painel de gases de uma sala de cirurgia (2015)

O Protegemed consiste em três unidades principais. A primeira é o módulo, composto por uma plataforma embarcada ligada a sensores que monitoram a corrente elétrica. O módulo analisa os dados coletados e os envia ao servidor. A segunda é o servidor, que recebe os dados do módulo e os armazena em um banco de dados. A terceira é a interface web, denominada “software de apoio”,

Table 1: Detalhes dos microcontroladores utilizados no Protegemed

Marca	Modelo	Clock da CPU	Memória RAM	Memória Flash	ADCs	Ano	Observações
NXP	LPC1768	96 MHz	32 KB*	512 KB	6	2012	*Existem versões com 64 KB
TI	TM4C1294-XL	120 MHz	256 KB	1024 KB	18	2017	

que permite manipular dados no banco, além de apresentar as informações coletadas pelo módulo através de gráficos (Schmitz; 2017). Existe uma quarta unidade que é o servidor WebSocket, que foi implementado na linguagem PHP utilizando a biblioteca Ratchet. Esse servidor (chamado a partir daqui de "servidor Ratchet" ou apenas de "Ratchet") permite a comunicação bidirecional entre o software de apoio e o módulo. O foco principal deste trabalho será em cima do servidor Ratchet. Uma visão geral da comunicação entre as unidades pode ser vista na figura 3.

No momento, existem duas versões em funcionamento em plataformas embarcadas. Uma utiliza o SoC NXP LPC1768 (NXP; 2016) (Schmitz; 2017, p. 26-27), que possui 32 KB de RAM e é o microcontrolador que se encontra no painel de gases mencionado anteriormente, ilustrado na figura 2. O seu firmware consiste na plataforma Mbed da ARM (ARM; n.d.), que foi utilizada como base para a implementação deste trabalho. O dispositivo é capaz de operar 6 canais analógicos ao mesmo tempo. Cada tomada utiliza 2 canais analógicos e no painel de gases da sala de cirurgia constam 6 tomadas. Como não é suportada a leitura de 12 canais analógicos simultaneamente em tempo real, são necessários 2 desse SoC para atender todas as tomadas. Para superar essa limitação (entre outras), uma outra versão utiliza o SoC TI TM4C (Tex; 2016), que pode ler até 18 canais analógicos ao mesmo tempo e possui 256 KB de memória principal. Esses parâmetros, entre outros, constam na tabela 1. (dos Santos; 2017) O acesso aos dispositivos é difícil, pois o centro cirúrgico é controlado. Sendo assim, seria vantajoso possibilitar a configuração remotamente. Para isso, será utilizado um protocolo de rede como meio de comunicação.

Tanto para a configuração, quanto para o controle remoto, será utilizada como cliente a interface Web existente do Protegemed, implementada em PHP utilizando o framework CodeIgniter. A interface é utilizada atualmente para a análise dos dados coletados da plataforma embarcada (Trentin and Rebonatto; 2011) (Perego and Rebonatto; 2014).

Na figura 3, é apresentado um diagrama da comunicação atual entre os diferentes agentes envolvidos, incluindo a direção da comunicação e os protocolos utilizados. Primeiramente, cada um dos componentes embarcados lê os valores das tomadas. Após um determinado número de leituras, o componente estabelece uma conexão HTTP com o servidor principal e envia os dados através dessa conexão. O servidor armazena os dados recebidos em um banco de dados. O usuário final, através de um navegador web, estabelece uma conexão HTTP com o servidor principal e acessa a interface web do software de apoio. Conforme a figura 4, na página de monitoração, o usuário pode solicitar os dados lidos em um determinado período. Após a requisição, o servidor busca os dados do banco e os retorna ao usuário solicitante.

Existe também um segundo fluxo, em que o servidor principal estabelece uma conexão WebSocket com um servidor secundário, que estabelece uma conexão WebSocket com cada componente embarcado (uma conexão para cada componente). A partir dessas conexões é que será definido um protocolo de rede na camada de aplicação para que o usuário final possa, através de uma página no software de apoio, controlar os microcontroladores, passando pelas conexões dos servidores principal e secundário. Ambos os servidores e o banco de dados executam dentro de uma máquina virtual em um computador dentro do hospital, utilizando o sistema operacional Linux.

3 Protocolos

3.1 HTTP

O HTTP é um protocolo de rede na camada de aplicação que utiliza o modelo requisição-resposta. O protocolo possibilita a comunicação entre sistemas de informação distribuídos e baseados em hipertexto. A semântica do protocolo é extensível e as mensagens são auto-descritivas. (Fielding and Reschke; 2014a).

O protocolo é sem estado (*stateless*). Isso implica (entre outros quesitos) em que todos os dados necessários para a operação do protocolo devem estar contidos em cada mensagem (i.e.: em cada requisição e em sua respectiva resposta).

O fluxo básico é o seguinte: Um cliente envia uma requisição a um servidor. O servidor recebe a requisição, processa e retorna uma resposta ao cliente. Baseado nas informações da resposta, o cliente pode decidir enviar uma nova requisição ou simplesmente considerar o fluxo como finalizado.

3.1.1 Formato

Conforme a figura 5, a requisição utiliza o seguinte formato: Na primeira linha ("*request-line*"), deve ser especificado o método HTTP a ser executado, o recurso sobre o qual o método deve ser aplicado e a versão do protocolo utilizada pelo cliente. Após, devem seguir todos os cabeçalhos (um por linha), uma linha em branco e o corpo da mensagem, respectivamente. O recurso identifica uma entidade remota, representado por um *Uniform Resource Identifier* (URI), que é o "alvo" do método. Exemplos: O caminho de um arquivo no servidor ou a interface de um comando. Os cabeçalhos ("*headers*") são opcionais, com exceção do cabeçalho **Host** no HTTP 1.1, utilizado para identificar o destino da mensagem. O corpo da mensagem ("*body*") é opcional.

Os métodos são *case-sensitive* e são escritos com letras maiúsculas. Os métodos definidos no HTTP relevantes a esse trabalho são os seguintes, em tradução livre de (Fielding and Reschke; 2014b, seção

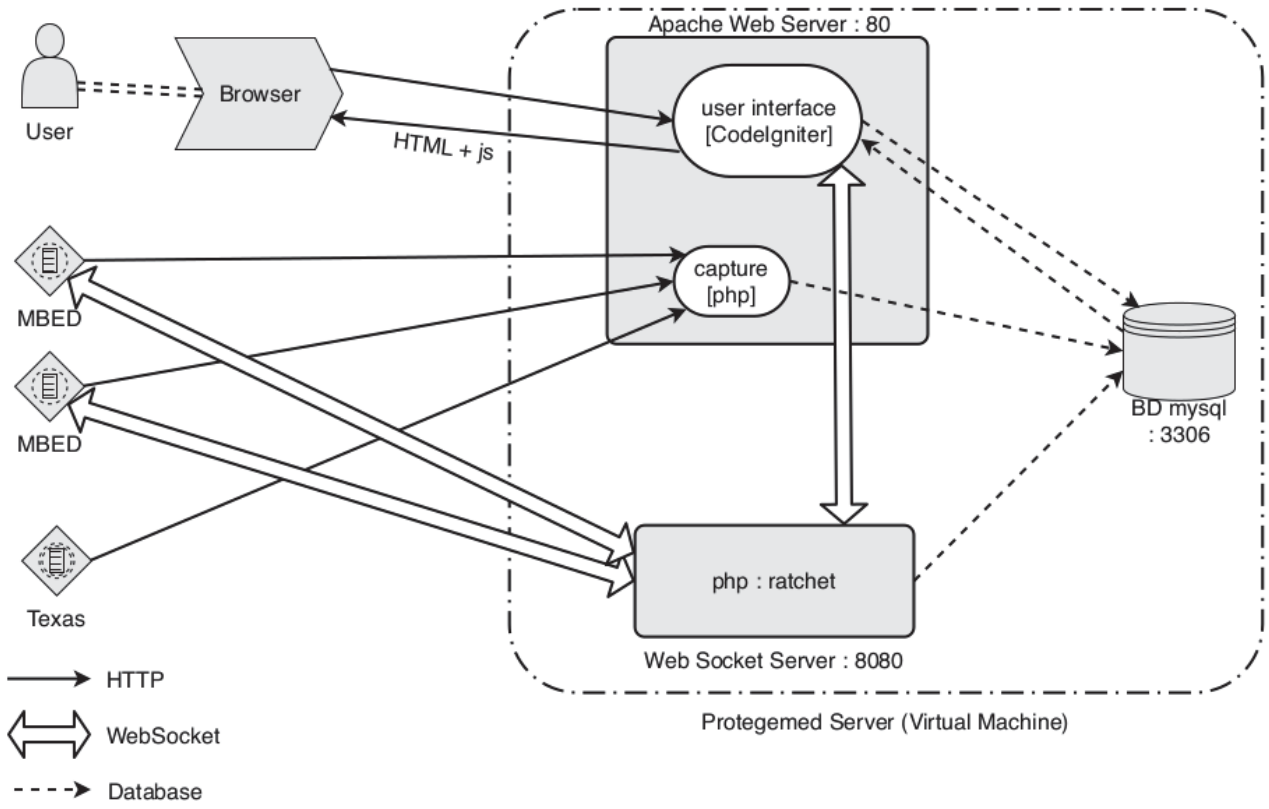


Figure 3: Comunicação entre os diferentes programas do Protegemed

Figure 4: Detalhes dos últimas formas de onda capturas no software de apoio

```
1 METODO RECURSO PROTOCOLO/VERSAO
2 CABECALHO_1: VALOR_1
3 CABECALHO_2: VALOR_2
4 [...]
5 CABECALHO_N: VALOR_N
6
7 BODY
```

(a) Formato de uma requisição HTTP

```
1 PROTOCOLO/VERSAO ESTADO DESCRICAO
2 CABECALHO_1: VALOR_1
3 CABECALHO_2: VALOR_2
4 [...]
5 CABECALHO_N: VALOR_N
6
7 BODY
```

(b) Formato de uma resposta HTTP

Figure 5: Formato das mensagens do HTTP

4.3):

- GET: Transfere a representação atual do recurso alvo.
- HEAD: Igual ao GET, porém, apenas transfere a linha de estado e os cabeçalhos.
- POST: Realiza um processamento específico ao recurso especificado na requisição.
- OPTIONS: Descreve as opções de comunicação para o recurso alvo.

Na resposta, a primeira linha ("*status-line*") deve conter a versão do HTTP utilizada no servidor, o código do estado da requisição ("*status code*") e a descrição do código, respectivamente. Da mesma forma que a requisição, todos os cabeçalhos devem constar imediatamente após a primeira linha. Os cabeçalhos e o corpo da resposta são opcionais.

O código de estado é um número inteiro que determina o resultado da requisição e a descrição é uma frase correspondente ao número. O primeiro dígito do código indica a sua classe, conforme a lista seguinte:

- 1xx: Informativo
- 2xx: Sucesso
- 3xx: Redirecionamento
- 4xx: Erro do Cliente
- 5xx: Erro no Servidor

Um código de 100 a 199 indica que a resposta foi recebida e que o processamento da requisição está em andamento. É uma resposta intermediária, enviada antes de finalizar o processamento solicitado na requisição e enviar uma resposta final. Mensagens dessa classe podem ser ignoradas pelo cliente. Um código de 200 a 299 indica que a requisição foi recebida, entendida e processada com sucesso. Um código de 300 a 399 indica que é necessária uma ação do cliente para poder cumprir o processamento da requisição. Um código de 400 a 499 indica que há um problema na requisição. Por exemplo, a presença de um erro sintático ou semântico. Um código de 500 e 599 indica que houve um erro interno no servidor que o impediu de completar a requisição. Um cliente deve no mínimo entender as classes de cada estado, mesmo que não entenda o código em específico. (Fielding and Reschke; 2014b, p. 46). A figura 6 contém um exemplo de uma troca de mensagens.

```
Client request:

  GET /hello.txt HTTP/1.1
  User-Agent: curl/7.16.3 libcurl/7.16.3
  Host: www.example.com
  Accept-Language: en, mi

Server response:

  HTTP/1.1 200 OK
  Date: Mon, 27 Jul 2009 12:28:53 GMT
  Server: Apache
  Last-Modified: Wed, 22 Jul 2009 19:15:56 GMT
  ETag: "34aa387-d-1568eb00"
  Accept-Ranges: bytes
  Content-Length: 51
  Vary: Accept-Encoding
  Content-Type: text/plain

  Hello World! My payload includes a trailing
  CRLF.
```

Figure 6: Uma requisição HTTP e sua respectiva resposta (Fielding and Reschke; 2014a, p. 7).

Através dos cabeçalhos, é possível especificar informações relacionadas à conexão, o tipo de conteúdo enviado, o tipo de conteúdo aceito, informações sobre segurança e sobre *caching*, entre outros. Através do *header* **Content-Type**, é possível especificar na requisição o tipo de conteúdo que está sendo enviado no corpo mensagem. Também é possível utilizar o cabeçalho **Accept** para especificar o tipo de conteúdo que o cliente aceita no corpo da resposta. Por exemplo: texto, JSON e XML. Na requisição, o tipo de conteúdo enviado pode ser diferente do tipo de conteúdo solicitado, conforme ilustrado na figura 7. Nesse exemplo, o cliente solicita a criação de um usuário e envia os atributos do usuário em um formato de texto simples ("*plain text*"). Na mesma requisição, solicita que o servidor responda no formato JSON. A resposta do servidor contém a entidade criada, no formato solicitado pelo cliente.

Tanto na requisição quanto na resposta, cada linha que precede o corpo da mensagem deve ser finalizada pela sequência de caracteres **CR LF**. Ou seja, 0x0d e 0x0a, respectivamente, na representação hexadecimal do padrão de codificação *American Standard Code for Information Interchange* (ASCII) (Fielding and Reschke; 2014a, seção 3).

```
POST /api/create-user HTTP/1.1
Host: http://example.com
Accept: application/json
Content-Type: text/plain

name=Bob
email=bob@bobindustries.com
```

(a) Cliente solicitando a criação de um novo usuário, no formato de texto simples

```
HTTP/1.1 201 Created
Content-Type: application/json
Location: /api/user/2

{
    "id" : 2,
    "email" : "bob@bobindustries.com"
    "name" : "Bob",
}
```

(b) Servidor respondendo com a entidade criada, representada no formato JSON

Figure 7: Troca de mensagens HTTP, utilizando diferentes tipos de conteúdo

3.2 WebSocket

O WebSocket é um protocolo de rede que permite a comunicação bidirecional entre um cliente e um host remoto. Consiste em um handshake seguido de frames de mensagens, realizados em cima do protocolo TCP (Fette and Melnikov; 2011).

Comparado ao HTTP, possui um *overhead* menor na comunicação bidirecional, pois ele opera sobre uma única conexão TCP (Schmitz; 2017). Isso é possível pois, no WebSocket, o servidor pode enviar mensagens diretamente ao cliente, sem precisar que este envie uma mensagem inicial (como uma requisição, no caso do HTTP, por exemplo). Em outras palavras, permite que a comunicação seja full-duplex, ao invés de somente half-duplex (Tanenbaum and Wetherall; 2011, p. 97). Essa característica possibilita que o *overhead* do protocolo seja menor, o que é vantajoso para sistemas embarcados, visto que seus recursos normalmente são mais limitados, comparados aos de um computador convencional (cf. seção 2). O protocolo pode ser utilizado como uma forma de comunicação de baixa latência.

O protocolo de Schmitz (chamado a partir daqui de "S17") define um formato de comunicação entre um cliente e um módulo e é executado sobre o protocolo WebSocket. A figura 8 mostra uma interface para o uso do protocolo, contendo um formulário para o envio de comandos e um console para mostrar as mensagens enviadas e recebidas. O formato do protocolo consiste no delimitador `#*`, seguido do comando a ser enviado, seguido do delimitador `*#`. Após, deve ser informado o ip do módulo e, opcionalmente, argumentos para o comando, separados pelo delimitador `:`. Os comandos suportados originalmente são os seguintes:

- test: Envia um comando de teste
- reset: Reinicia o módulo
- capture: Envia ao servidor uma captura dos valores do canal da tomada informado
- setLimit: Configura o valor do limite do canal informado
- setLimitStandBy: Configura o valor do limite de

Table 2: Comandos originais utilizados através do protocolo S17

Comando	Parâmetro 1	Parâmetro 2	Parâmetro 3	Exemplo
test	-	-	-	test
reset	-	-	-	reset
capture	Id da tomada	Canal (fase ou fuga)	-	capture:1:p
setLimit	Id da tomada	Canal (fase ou fuga)	Valor	setLimit:1:p:2
setStandByLimit	Id da tomada	Canal (fase ou fuga)	Valor	setStandByLimit:1:d:0.5

Figure 8: Comandos sendo enviados ao Ratchet na página index_test.php original

standby do canal informado

Conforme a tabela 2, Os comandos **test** e **reset** não necessitam de argumentos. Para o comando **capture**, é necessário informar o id da tomada e se deve ser utilizado o canal de fase ou de fuga. Para os comandos **setLimit** e **setLimitStandBy**, é necessário informar os mesmos valores do comando **capture** e também o valor do limite a ser configurado. No módulo, a *Thread* de WebSocket, ao receber uma mensagem, identifica o comando recebido, o executa e, opcionalmente, envia uma resposta de volta.

4 Definição do protocolo

Para permitir a comunicação bidirecional através da rede entre a plataforma embarcada e o servidor, é necessário definir o protocolo sobre o qual a comunicação irá ocorrer. Devido aos recursos limitados do hardware do componente embarcado, foi visto que o protocolo WebSocket seria um candidato viável, devido ao baixo custo de memória e processamento (Schmitz; 2017).

Diferentemente do HTTP, o WebSocket não define um padrão de *marshalling* (ex.: JSON; BSON) nem de formato das mensagens (ex.: cabeçalhos e corpo) (Fette and Melnikov; 2011). Além disso, um formato de requisição-resposta padronizado poderia facilitar a compreensão da implementação. Com isso, foi decidido criar um protocolo executado sobre uma conexão WebSocket e que utiliza parte do formato e da semântica do protocolo HTTP para a serialização das mensagens. O protocolo de Schmitz poderia ter sido expandido para suportar comandos de configuração. Entretanto, foi decidido utilizar um formato já padronizado, pois há planos de se utilizar o protocolo definido como a forma padrão de comunicação através da rede em outros componentes dentro do Protegemed, incluindo novos componentes a serem desenvolvidos. O suporte a meta-dados e a diferentes formatos de *marshalling*, através dos cabeçalhos, facilita a aplicação do protocolo em diferentes casos de uso. Por exemplo, caso seja necessário futuramente enviar dados estruturados no formato JSON. Uma outra possível vantagem seria o reuso das ferramentas voltadas ao HTTP, sem precisar de grandes modificações. Por exemplo, ferramentas que realizam requisições e permitem a visualização e análise dessas requisições e suas respectivas respostas.

4.1 Formato

O formato das mensagens é essencialmente equivalente ao do HTTP, com algumas exceções. Primeiramente, na primeira linha da requisição e da resposta, no lugar de "HTTP", é utilizado "HOWS". Foi escolhido utilizar algo diferente pois o protocolo proposto não é 100% compatível com o HTTP. Um exemplo de uma troca de mensagens está presente na figura 9. Os métodos do HTTP suportados no HOWS são equivalentes aos métodos descritos na seção 3.1.1.

Na figura 9, é realizada uma requisição do tipo HEAD para o recurso “/user/bob”, indicando, através do cabeçalho “Accept”, que o formato JSON é aceito na resposta. Na primeira linha da resposta, é indicado que a requisição foi processada com sucesso, através do código de estado 200. Na linha seguinte, através do cabeçalho “Content-Type”, é indicado que o corpo da mensagem teria incluído um conteúdo no formato JSON. A requisição termina nessa linha, ou seja, a resposta não inclui uma linha em branco nem um corpo, devido ao método da requisição ser do tipo HEAD. Na figura 10, é realizada uma requisição do tipo HEAD para o recurso “/helloworld”. Na primeira linha da resposta, é indicado que o recurso não foi encontrado, através do código de estado 404. A resposta termina nessa linha. Na figura 11, é realizada uma requisição OPTIONS para o recurso “/config”. Na primeira linha da resposta, o código de estado 200 é indica que a requisição foi processada com sucesso. Na linha seguinte, através do cabeçalho “Allow”, está presente uma lista de métodos válidos para o recurso. A lista contém os métodos GET, HEAD, OPTIONS e POST.

A semântica dos códigos de estado deve ser considerada idêntica à semântica de Fielding and Reschke (2014b). Os códigos de estado do HTTP suportados pelo HOWS são os seguintes:

- 100 Informational
- 200 OK
- 204 No Content
- 300 Multiple Choices
- 400 Bad Request
- 404 Not Found
- 500 Internal Server Error
- 505 Protocol Version Not Supported

Conforme (Fielding and Reschke; 2014b), para que a implementação de um cliente seja considerada

```
1  Client request:
2
3    HEAD /user/bob HOWS/1.0
4    Host: 192.168.17.101
5    Accept: application/json
6
7  Server response:
8
9    HOWS/1.0 200 OK
10   Content-Type: application/json
```

Figure 9: Requisição dos cabeçalhos de um recurso.

```
1  Client request:
2
3    HEAD /helloworld HOWS/1.0
4    Host: 192.168.17.101
5
6  Server response:
7
8    HOWS/1.0 404 Not Found
```

Figure 10: Requisição dos cabeçalhos de um recurso. O recurso não foi encontrado pelo servidor

compatível com o HTTP, não é necessário que nela seja considerado o significado de cada código de estado obtido. Basta que seja entendida a classe de cada código. Considerando o caso de uso do protocolo no projeto, os códigos de estado genéricos (i.e.: x00), juntamente com o código 204 são suficientes. Os outros códigos foram incluídos para facilitar a depuração. A análise de códigos que seriam incompatíveis com o HOWS está fora do escopo deste trabalho. O suporte aos outros códigos do HTTP fica a critério da implementação. Um registro dos códigos conhecidos é mantido em Internet Assigned Numbers Authority (n.d.).

As requisições também podem ser utilizadas para solicitar a execução de comandos no dispositivo remoto. Para solicitar a execução de um comando, deve ser utilizado o método POST. Quando a resposta contém um corpo, é recomendado o uso do cabeçalho **Content-Length** para indicar, em octetos, o seu tamanho. (Fielding and Reschke; 2014a, p. 29). Caso o comando não necessite de argumentos, recomenda-se utilizar um body vazio, juntamente o cabeçalho "Content-Length: 0". Por exemplo, na figura 12, é solicitado, através do recurso "/cmd/reset", que o dispositivo seja reiniciado. Mesmo sem enviar nada no body, foi escolhido utilizar o POST, visto que os comandos podem modificar o recurso remoto (e.g.: reiniciar o dispositivo). Isso significa que a requisição não é idempotente, tornando semanticamente inválido o uso do GET para essa

```
1  Client request:
2
3    OPTIONS /config HOWS/1.0
4    Host: 192.168.17.101
5
6  Server response:
7
8    HOWS/1.0 200 OK
9    Allow: GET, HEAD, OPTIONS, POST
10   Content-Length: 0
```

Figure 11: Requisição de quais métodos são aceitos para um determinado recurso

```
1  Client request:
2
3    POST /cmd/reset HOWS/1.0
4    Host: 192.168.17.101
5    Content-Length: 0
6
7  Server response:
8
9    HOWS/1.0 202 Accepted
```

Figure 12: Solicitação para que o microcontrolador seja reinicializado

situação.

4.2 Compatibilidade

Mesmo buscando ser um *subset* relativamente compatível do HTTP, é possível que certas funcionalidades acabem operando de forma diferente. Os seguintes requisitos do HTTP foram identificados como possíveis causas de incompatibilidade:

- O cabeçalho **Host** deve estar presente em todas as requisições. (Fielding and Reschke; 2014a, p. 43)
- Os cabeçalhos **Host** e **Connection** devem ser implementados. (Fielding and Reschke; 2014a, p. 13).

Os cabeçalhos relacionados à conexão não são considerados, pois ela já é gerenciada através do protocolo WebSocket. Visando manter a implementação simples, a possibilidade de se guardar as respostas em *cache* é desconsiderada. Isso inclui os cabeçalhos e métodos. Será considerado apenas os métodos cuja resposta **não** pode ser mantida em cache, como, por exemplo, o método POST (Fielding and Reschke; 2014b, p. 23). Não foi validada a compatibilidade do protocolo Host na forma que foi implementada no HOWS.

5 HOWS no Protegemed

O uso do protocolo na plataforma Protegemed será na leitura e na escrita das configurações de cada microcontrolador, além do envio de comandos. A leitura e os comandos devem utilizar somente a primeira linha da requisição e nada no body. A requisição de escrita deve informar o novo valor na mensagem.

O caminho "/config" é reservado na URI para o mapa de configuração do Protegemed. Cada chave de configuração é acessada através do recurso "/config". Para buscar o valor atual de um recurso, é utilizada uma requisição do tipo GET para o caminho do recurso e um body vazio, conforme a figura 13.

Para gravar um novo valor em uma chave de configuração, deve ser realizada uma requisição POST contendo o novo valor no body. No caso do JSON, o body da requisição deve conter um mapa da chave e o valor que se deseja gravar. O servidor deve retornar o novo valor do recurso, da mesma forma que ocorre com uma requisição GET. No caso do JSON, o body deve conter o mesmo recurso (especificado na url) mapeado ao novo valor. Na figura 14, o cliente envia uma requisição no formato JSON solicitando que o valor do atributo "serverIp" seja atualizado. Tanto

```
Client request:

  GET /config/serverIp HOWS/1.0
  Host: 192.168.17.101

Server response:

  HOWS/1.0 200 OK

  192.168.103.101
```

Figure 13: Requisição de leitura do valor corrente do IP do microcontrolador (no formato texto)

```
Client request:

  POST /config/serverIp HOWS/1.0
  Host: 192.168.17.101
  Accept: application/json
  Content-Type: application/json

  { "serverIp" : "196.168.103.102" }

Server response:

  HOWS/1.0 200 OK
  Content-Type: application/json

  { "serverIp" : "192.168.103.102" }
```

Figure 14: Requisição de escrita de um novo valor de IP para o microcontrolador (no formato JSON)

na leitura quanto na escrita, pode ser utilizado texto normal (plain-text) ou JSON.

O caminho "/cmd" é reservado na URI para comandos. Por exemplo, para realizar a captura do valor da corrente de fase ou da corrente de fuga, pode ser enviada uma requisição do tipo POST para o recurso "/cmd/capture". Na figura 16, a tomada a ser lida e o tipo de dado a ser lido são enviados no corpo da requisição. O servidor retorna o valor capturado do canal da tomada em um objeto JSON.

6 Implementação do HOWS

Serão detalhadas as adições e alterações realizadas no Protegemed, separadas por componente (servidor Ratchet, Mbed e software de apoio). Foi buscado um equilíbrio entre a reutilização das estruturas pré-existentes (para agilizar o desenvolvimento) e a produção de novas estruturas de forma organizada e de acordo com o que havia sido especificado. Para fins de referência futura, a implementação foi realizada na seguinte ordem:

i. Estruturas de dados e parsers (PHP)

```
Client request:

  POST /cmd/capture:1:p HOWS/1.0
  Host: 192.168.17.101

Server response:

  HOWS/1.0 200 OK
```

Figure 15: Requisição de leitura do valor da corrente de fase de uma determinada tomada (no formato de texto simples)

```
Client request:

  POST /cmd/capture HOWS/1.0
  Host: 192.168.17.101
  Accept: application/json
  Content-Type: application/json

  {
      "outlet" : 1,
      "purpose" : "phase"
  }

Server response:

  HOWS/1.0 200 OK
  Host: 192.168.17.101
  Content-Type: application/json

  {
      "result" : 0.2
  }
```

Figure 16: Requisição de leitura do valor da corrente de fase de uma determinada tomada (no formato JSON)

ii. Estruturas de dados e parsers (C++/desktop)
iii. Porte da implementação em C++ do desktop para o Mbed
iv. Página de testes e roteamento das mensagens no servidor Ratchet
v. Ampliação dos comandos suportados e adição das configurações no Mbed
vi. Criação da nova página de comunicação no software de apoio

6.1 Ratchet

O Ratchet é o componente que realiza o roteamento entre o software de apoio e cada componente embarcado. Cada nodo é responsável por iniciar uma conexão WebSocket com o servidor. Ao receber uma mensagem de um nodo, o servidor busca o destino na mensagem e a encaminha para o destinatário encontrado. Note que a comunicação entre os diferentes componentes do Protegemed é sempre realizada através do Ratchet, pois é nele que está implementado esse processo de análise do destino, além de um pré-processamento, dependendo do conteúdo da mensagem.

6.1.1 Parsers

Primeiramente, foram implementadas as classes Message, Request e Response em PHP. Message é uma estrutura de dados para uma mensagem genérica (requisição ou resposta), que contém as variáveis **line** e **body**. Request extende a classe Message e possui as variáveis **method** e **resource**. Response extende a classe Message e utiliza as variáveis **request** e **statusCode**. Cada classe possui um parser, sendo eles o MessageParser, o RequestParser e o ResponseParser.

Cada parser possui um método chamado **parse**, que recebe uma string por argumento e retorna um objeto respectivo à sua classe (Message, Request ou Response). O método parse da classe MessageParser ("MessageParser::parse") recebe uma string cujo conteúdo deve ser ou uma requisição ou uma resposta. Após, lê essa string e copia a linha, os cabecalhos e o body da mensagem para variáveis separadas. Por fim, cria um objeto Message contendo essas

variáveis e o retorna. O método RequestParser::parse recebe uma requisição em formato de string e a envia para o método MessageParser::parse para obter um objeto Message. A *request line* é lida desse objeto e, separado por espaços, são copiados o método, o recurso e a token (que contém o nome e a versão do protocolo) para variáveis separadas. Em seguida, um objeto Request é criado baseado nessas variáveis e no body do objeto Message. O ResponseParser funciona similarmente, porém, os valores copiados (também separados por espaço) da primeira linha do objeto Message são do código de estado e a mensagem de estado. Ao invés de um objeto Request, é criado um objeto Response, baseado nessas variáveis e também no body do objeto Message. Cada classe base possui também o método toString, que retorna cada campo separado por CRLF, independente do formato utilizado durante o *parsing*. Essa normalização também é feita na implementação em C++.

Para cada um dos *parsers*, foi criado um arquivo de testes para validar a implementação. Em cada arquivo foram testados oito tipos de requisições:

- Com a linha, os headers e o body (LF)
- Somente a linha e os headers (LF)
- Somente a linha e o body (LF)
- Somente a linha (LF)
- Com a linha, os headers e o body (CRLF)
- Somente a linha e os headers (CRLF)
- Somente a linha e o body (CRLF)
- Somente a linha (CRLF)

Os testes ajudaram a encontrar um bug no MessageParser no segundo caso, em que estava sendo criado um objeto inválido quando a requisição não possuía um body.

No arquivo *ws-comunicacao.php*, é definida a classe Comunicacao, que implementa a interface MessageComponentInterface do Ratchet. A classe possui uma única variável, denominada **$clientes**, que é uma lista das conexões ativas. Quando uma nova conexão WebSocket é estabelecida, o método onOpen é executado. O método recebe a conexão por argumento e a adiciona na lista **$clientes**. Quando uma conexão WebSocket é desfeita, o método onClose é executado. O método recebe a conexão por argumento e a remove da lista **$clientes**. Essas conexões são objetos que implementam a interface ConnectionInterface e armazenam diversas informações, dentre elas, o endereço IP remoto e o id da connexão, denominados **remoteAddress** e **resourceId**, respectivamente.

Quando uma mensagem é recebida, o método onMessage é executado. O método recebe por argumento a conexão de quem enviou e a mensagem recebida. Na versão original, o IP é separado dos argumentos e eles são gravados em variáveis separadas. Primeiramente, dependendo do comando, o Ratchet realiza um pré-processamento, como, por exemplo, buscar uma informação no banco de dados e adicioná-la na mensagem. Após, encaminha o comando e os argumentos para o IP em questão no seguinte formato: "comando:argumentos". Note que ao receber a mensagem, o destinatário não possui a informação de quem a enviou, apenas o IP do servidor Ratchet. Dessa forma, não é possível enviar uma resposta diretamente ao remetente. Isso ocorre pois, na implementação do onMessage, não é realizado um mapeamento entre remetente e destinatário. Ou seja, a troca de mensagens ocorre (fora as conexões mantidas na lista **$clientes**) de forma sem estado ("*stateless*"); cada mensagem enviada ao servidor Ratchet deve conter o seu destinatário.

Para possibilitar o envio de uma resposta ao destinatário no HOWS, primeiramente foi pensado em utilizar um cabeçalho extra na mensagem, "Host-From". Dessa forma, ao receber a requisição, o Ratchet adicionaria o IP do remetente nesse cabeçalho antes de encaminhá-la. O destinatário seria responsável por adicionar o valor do cabeçalho "Host-From" da requisição no cabeçalho "Host" da resposta. Todavia, caso houvessem múltiplos clientes com o mesmo IP conectados ao servidor Ratchet e um deles enviasse uma requisição, todos eles receberiam a resposta. Com isso, foi decidido utilizar o id da conexão para identificar o remetente, através do cabeçalho "From-Id". A requisição seria enviada normalmente com o cabeçalho "Host" ao servidor Ratchet, que seria responsável por adicionar o id da conexão do remetente no cabeçalho "From-Id" e encaminhá-la ao destinatário. O destinatário seria responsável por adicionar o valor do cabeçalho "From-Id" da requisição em um novo cabeçalho "To-Id" na resposta. O servidor Ratchet, então, encaminharia a resposta à conexão que possui o id desse cabeçalho. Porém, na implementação no mbed, como os cabeçalhos da requisição são copiados diretamente para a resposta, e para evitar o desenvolvimento e a validação de um parser de cabeçalhos no mbed, foi decidido, por fim, utilizar um único cabeçalho a mais, denominado "Request-Connection-Id". O processo seria equivalente ao anterior, porém o cabeçalho pode ser copiado da requisição para a resposta, pois mantém o mesmo valor semântico.

O fluxo final para a troca de mensagens no HOWS, de acordo com a figura 17. é o seguinte: Ao receber uma mensagem, o servidor Ratchet verifica a existência do cabeçalho "Request-Connection-Id". Caso exista, para cada conexão em **$clientes**, verifica se o id da conexão é igual ao valor do cabeçalho. Caso seja, envia a mensagem na conexão. Caso o cabeçalho não esteja presente, procura o cabeçalho "Host". Caso não exista, retorna um erro ao cliente. Caso exista, para cada conexão em **$clientes**, verifica se o IP da conexão é igual ao valor do cabeçalho. Caso seja, remove o cabeçalho "Host" da mensagem e adiciona o cabeçalho "Request-Connection-Id" contendo o id da conexão encontrada. Após, encaminha a mensagem na conexão encontrada. Caso não exista, retorna um erro ao cliente.

6.2 Mbed

Foram utilizadas duas interfaces para a comunicação com o microcontrolador LPC1768. A primeira é a entrada Ethernet, utilizada para toda a comunicação através da rede. A segunda é a entrada USB, para gerenciar o sistema de arquivos e para realizar a comunicação serial. A comunicação serial foi realizada através do programa **picocom** e de forma unidirecional, visualizando apenas a saída do módulo.

Durante a inicialização do módulo, são registrados os estados de certos objetos, como, por exemplo,

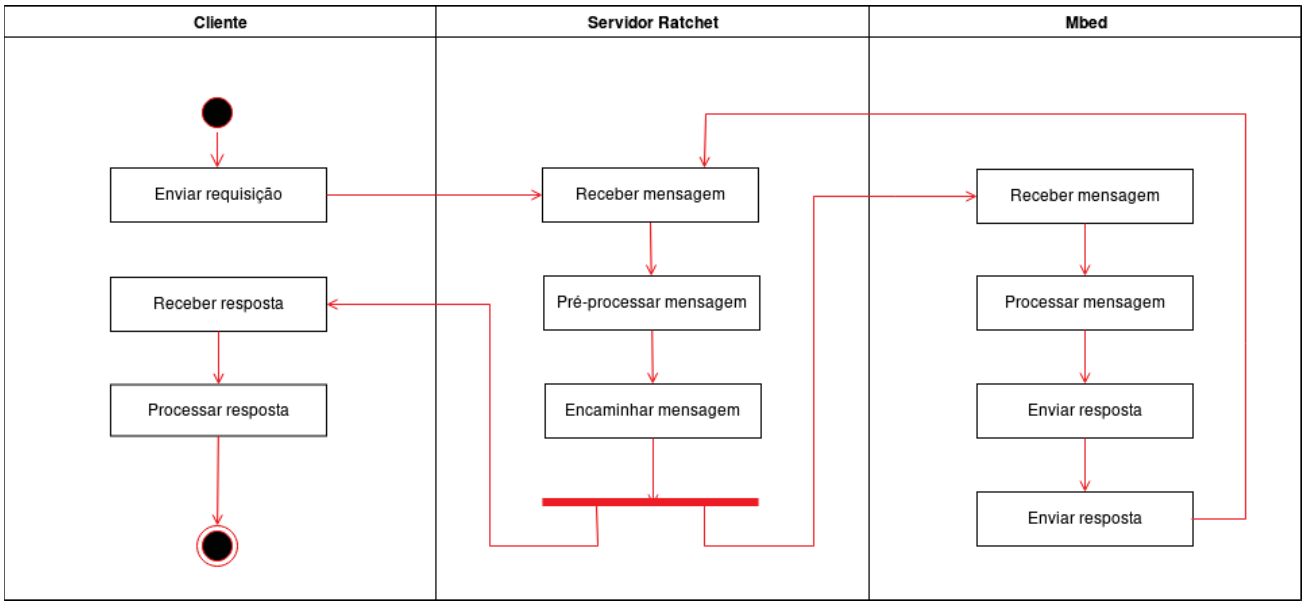


Figure 17: Fluxo da troca de mensagens através do servidor Ratchet

o objeto **Settings**, que é a interface principal para gerenciar as configurações internas. Conforme a figura 18, na saída, são registrados os valores das configurações de rede, como, por exemplo, a **networkAddress** e **networkMask**, que definem o IP do módulo e a máscara de rede, respectivamente. Também são mostrados os valores de configuração de cada canal, como, por exemplo, **purpose** e **limit**, que definem qual é o tipo do canal (se é de fase ou de fuga) e qual é o valor máximo considerado dentro do aceitável, respectivamente. Também é exibido o estado de cada compartimento de memória RAM, mostrando a quantidade que está sendo utilizada e o tamanho limite de cada um.

O desenvolvimento no Mbed foi realizado na IDE online da ARM, conforme a figura 19. Nessa plataforma, é possível criar e editar arquivos, projetos e bibliotecas. Também é possível compilar o projeto e baixar o arquivo gerado para ser gravado no microcontrolador. O gerenciamento de dependências é realizado automaticamente pela IDE, sendo necessário apenas criar os diretórios das bibliotecas e os arquivos com as extensões .h, .c e .cpp. O arquivo gerado possui a extensão **.bin** e é um firmware que contém também o sistema operacional do microcontrolador. Ao baixar o firmware, no LPC1768, é necessário montar o sistema de arquivos e copiar o arquivo diretamente para lá. Ao ser iniciado (ou reiniciado), o sistema operacional verifica se existe um arquivo **.bin** na raíz do sistema de arquivos. Caso exista, esse arquivo é carregado em memória e executado. Caso existam múltiplos arquivos **.bin** na raíz, o arquivo com a data de modificação mais recente é selecionado. O sistema de arquivos utiliza o formato FAT32 e é gravado diretamente no dispositivo de bloco ("*block device*"), ou seja, não são utilizadas partições. Para atualizar o firmware é necessário montar o sistema de arquivos do Mbed, compilar o programa, baixar o arquivo gerado para o sistema montado e reiniciar o dispositivo. Para automatizar parte da atualização, foram criados quatro *shell scripts*:

- mbed-mount: Monta o sistema de arquivos
- mbed-umount: Ejeta o sistema de arquivos
- mbed-update: Copia o firmware para a raíz do Mbed
- mbed-doupdate: Executa mbed-mount, mbed-update e mbed-umount, respectivamente.

Os comandos mbed-mount e mbed-umount encontram o dispositivo de bloco baseado em sua estiqueta ("*device label*"). O mbed-update copia o firmware de um diretório configurável para a raíz do sistema de arquios do módulo. Dessa forma, é necessário apenas baixar um arquivo através do navegador e executar um comando. Caso o primeiro passo também seja automatizado, o processo inteiro (incluindo a compilação) poderia depender de somente um comando.

Conforme a tabela 3, foi utilizada a versão C++98 da linguagem (mais especificamente, a versão ISO/IEC 14882:1998). Porém, não estão disponíveis todas as funcionalidades da linguagem. Por exemplo, não é possível utilizar a *Standard Template Library* (STL), que possui funções para auxiliar, por exemplo, na manipulação de strings (utilizando objetos do tipo std::string). Também não é possível utilizar funções virtuais. Basicamente, foram utilizadas as mesmas funções e funcionalidades que também estão disponíveis na linguagem C. Porém, foram utilizadas classes e objetos no lugar de structs e funções de inicialização/término.

Primeiramente, os *parsers* foram desenvolvidos em um computador convencional. A estrutura de classes foi baseada na versão equivalente em PHP. O mesmo foi realizado para os testes de validação dos *parsers*.

O MessageParser foi a parte mais complexa de se implementar em C++, especialmente porque muito teve de ser re-implementado. Isso ocorreu devido à classe equivalente em PHP utilizar expressões regulares (através das funções preg_match e preg_split) para simplificar a implementação e este recurso não está disponível na plataforma. Nativamente, isso só está disponível em

Table 3: Ferramentas e configurações utilizadas durante o desenvolvimento na linguagem C++

Tipo	Nome	Versão	Ambiente	Observações/Parâmetros
Linguagem	C++	199711 (C++98)	Desktop e Mbed	Sem STL, exceções nem funções virtuais
Compilador	g++	8.2.1	Desktop	-Wall -Wextra -Wpedantic -g -MMD \ -std=c++98 -fno-exceptions
Compilador	armcc	5060300	Mbed	-
Biblioteca	mbed	2.0.164	Mbed	-
Biblioteca	mbed-rtos	125:5713cbbdb706	Mbed	-
Biblioteca	libbsd	0.9.1	Desktop	strdup e strsep
Emulador de Terminal	picocom	3.1	Desktop	-s 9600 –imap lfcrlf
Análise dinâmica	valgrind	3.14.0	Desktop	–leak-check=yes

```
    logMarksInterval    =   3600
    logLevel            =   90
CHANNEL PARAMETERS
    channels            =   6
    samples             =   256
    CHANNEL PURPOSE GAIN      OFFSET LIMIT     STBYLIMIT
     0      p        120.0000 2091      0.1000 2048.0000
     1      d          1.0000 1213   2048.0000 2048.0000
     2      p          1.0000 2104   2048.0000 2048.0000
     3      d          1.0000 2089   2048.0000 2048.0000
     4      p          1.0000 2088   2051.0000 2048.0000
     5      d          1.0000 2083   2048.0000 2048.0000
CURRENT NETWORK CONFIG
    networkAddress      =   192.168.17.101
    networkMask         =   255.255.255.0
    networkGateway      =   192.168.17.1
    networkDhcp         =   false
LOG : 2015-01-01 04:38:29 UTC : 00001 :  Started Protegemed !
Thread WebSocket Iniciada
/src/prtgmdNetwork/prtgmdWebSocket/Websocket.cpp(471): error(0):

Pos-Init

Main RAM: Load$RW_IRAM1$Base 1d6d0
IRAM1    Base 100000c8  Length 1b28     Limit 10001bf0
IRAM1$RO Base 100000c8  Length 0000     Limit 100000c8
IRAM1$RW Base 100000c8  Length 1b28     Limit 10001bf0
```

Figure 18: Saída do microcontrolador durante a comunicação serial

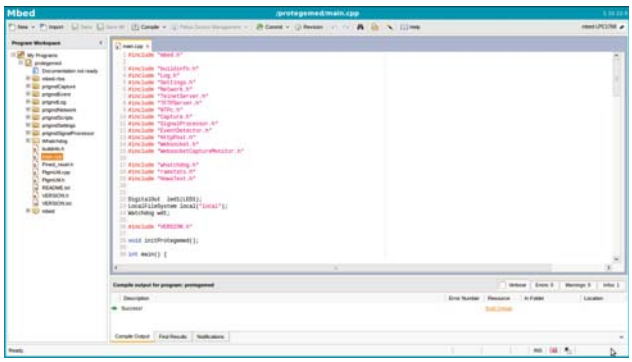


Figure 19: Projeto do protegemed aberto na plataforma de desenvolvimento online da ARM (ide.mbed.com)

compiladores que suportam a versão C++2011 (mais especificamente, ISO/IEC 14882:2011) e é necessário ter acesso à STL, que não está disponível no Mbed. A implementação dos *parsers* é, em boa parte, baseada na manipulação de strings. Sem acesso à STL, foi necessário implementar manualmente a comparação, extração, cópia e concatenação entre vetores de caracteres ("*C-strings*").

Certas funções da biblioteca de manipulação de strings padrão (*string.h*), tal como strtok, não validam os parâmetors de entrada. Isso pode resultar em um *buffer overrun* caso o tamanho do *buffer* de entrada seja maior do que o *buffer* de saída, por exemplo. Para evitar essa complicação, foram utilizadas as funções **strdup** e **strsep** da biblioteca de strings implementada pelo projeto OpenBSD. As funções estão disponíveis para o Linux através da biblioteca *libbsd*. A primeira função copia o conteúdo de uma string constante para uma variável. A outra permite copiar partes de uma string, baseado em um delimitador.

Durante a validação da implementação em C++, foram detectados problemas quando as mensagens não possuíam um corpo, sendo esse o mesmo problema que havia sido encontrado na implementação em PHP. Isso ocorreu mesmo o desenvolvimento em C++ ter sido baseado na versão corrigida do código em PHP. Esses erros foram corrigidos.

Após a implementação ser validada no desktop, ela foi portada para o Mbed. No caso das funções de string do projeto OpenBSD, os arquivos *strdup.c* e *strsep.c* foram copiados diretamente do projeto em questão. Foram copiadas as porções relevantes do arquivo *string.h*. Após alterações em algumas linhas, não havia mais nenhum erro causado por esses arquivos. Os arquivos do HOWS foram copiados para a plataforma e, seguido de algumas alterações para consertar certas incompatibilidades, o projeto foi compilado e o teste dos *parsers* foi executado com sucesso.

Na implementação dentro do Mbed, a função principal, *main*, é responsável por definir e gerenciar diferentes fluxos de execução através de *Threads*. Entre as *Threads* relevantes a esse trabalho, consta uma *Thread* para o envio dos dados coletados através do HTTP ("HttpPost::HttpPost_Thread"), e outra para a comunicação dos comandos através de WebSocket ("Websocket::Websocket_Thread"). Nessa última, eram realizados o *parse* e a execução de todos os comandos através do protocolo S17. Essa parte foi movida para uma classe separada, denominada CommandHandler. Isso permitiu reutilizar o código do S17, manter a compatibilidade com o protocolo HOWS e manter ambos protocolos

funcionando em paralelo.

Foi criada uma classe HowsHandler, contendo uma função *parse*, para executar uma configuração (utilizando a classe Settings) ou um comando (utilizando a classe CommandHandler), dependendo do conteúdo da mensagem. No início da função principal da *Thread*, foi adicionada uma verificação para testar se a mensagem que está vindo é do tipo HOWS. Caso seja uma resposta, ignorar. Caso seja uma requisição, enviar para o HowsHandler e retornar o resultado. Após estar funcional, os comandos foram expandidos, conforme a figura 4. A validação foi realizada da seguinte forma: buscar cada um valores configuráveis. Em seguida, gravar um novo valor em cada chave, buscá-los novamente e conferir se as alterações foram refletidas. O dispositivo foi reiniciado para garantir que os valores seriam restaurados. Posteriormente, o mesmo procedimento de buscar, gravar e buscar foi realizado. Entretanto, antes de reiniciar o Mbed, foi utilizado o comando write-file para registrar os novos valores no arquivo de configuração. Com a reinicialização, foi verificado no arquivo de configuração do Mbed que os valores haviam sido persistidos.

As funções de validação dos *parsers* foram adicionadas na função *main*, para serem executadas durante a inicialização do Mbed. Dessa forma, qualquer erro na validação causa o dispositivo a falhar o quanto antes, agilizando na descoberta de problemas. Após realizar uma certa quantia de requisições, o dispositivo Mbed era reiniciado automaticamente. Foi levantada a hipótese de que isso era causado por vazamentos de memória ("*memory leaks*"), que são causados quando uma parte da memória é alocada dinamicamente sem ser liberada posteriormente. Para verificar isso, foi decidido utilizar a ferramenta Valgrind, que, entre outras funcionalidades, permite analisar a ocorrência desses vazamentos (através do argumento "–leak-check=yes"). O programa de testes do HOWS foi executado através do Valgrind, que apontou a classe e a função onde o problema ocorria, conforme a figura 20. Duas variáveis, *line* e *header*, ocupando 64 bytes cada, estavam sendo alocadas sem que houvessem as suas respectivas liberações. Considerando oito testes de cada um dos dois *parsers* implementado, um vazamento de 1024 bytes incorria da inicialização do dispositivo, devido à execução das funções de validação. Após corrigir o código, compilar e testar novamente, segundo a ferramenta, nenhum *leak* é possível. Essa mensagem está visível na figura 21. No total, o tempo entre baixar o Valgrind, executá-lo, analisar a saída e corrigir o erro levou menos de trinta minutos. É possível que tempo necessário para encontrar os *leaks* verificando manualmente teria sido muito superior.

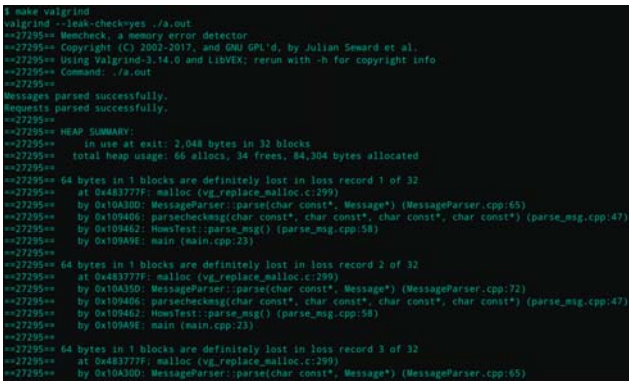


Figure 20: Execução através Valgrind do programa com vazamento de memória

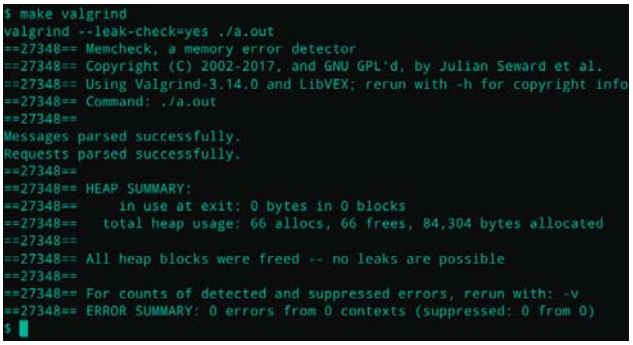


Figure 21: Execução através Valgrind do programa corrigido

6.3 Interface Web

No servidor Ratchet, foi utilizada a página index_test.php para estudar a comunicação entre o Ratchet e o mbed. Como era necessário reiniciar o mbed frequentemente (para atualizar o firmware), a tela foi alterada para deixar o comando de reset preenchido por padrão no campo. Além disso, foi adicionado um comando em JavaScript para o navegador tentar realizar uma conexão WebSocket automaticamente ao carregar a a página. Também foi alterada a fonte do log para monospace, para ficar visualmente mais similar ao código e facilitar a leitura.

Após realizadas as melhorias na interface de teste, foram implementados os formulários de comunicação utilizando o HOWS, conforme a figura 22. O formulário "Controle" permite enviar os comandos test, reset, print-config, read-config e write-config ao Mbed. O formulário "Configuração" permite buscar e gravar as configurações de gerais do módulo. No formulário de "Captura", é possível executar o comando "capture" no módulo. No próximo formulário, denominado "Limites", é possível enviar os comandos setGain, setOffset, setLimit e setStandByLimit para configurar os respectivos valores no módulo. Os detalhes dos comandos e configurações estão presentes na tabela 4 e na tabela 5, respectivamente.

A interface de teste foi, então, portada para o software de apoio. Foram copiados os arquivos MVC do módulo de comunicação já existente, removidas as partes que não seriam utilizadas e incluídos os campos da interface de teste. A página foi alterada para considerar os campos globais "Módulo" e "Tomada". Ao selecionar um módulo, os ids das tomadas são buscados no banco de dados e preenchidos no campo das tomadas. Os valores de ambos os campos servem para todos os formulários. Com isso, o campo de seleção de tomada foi removido de cada formulário.

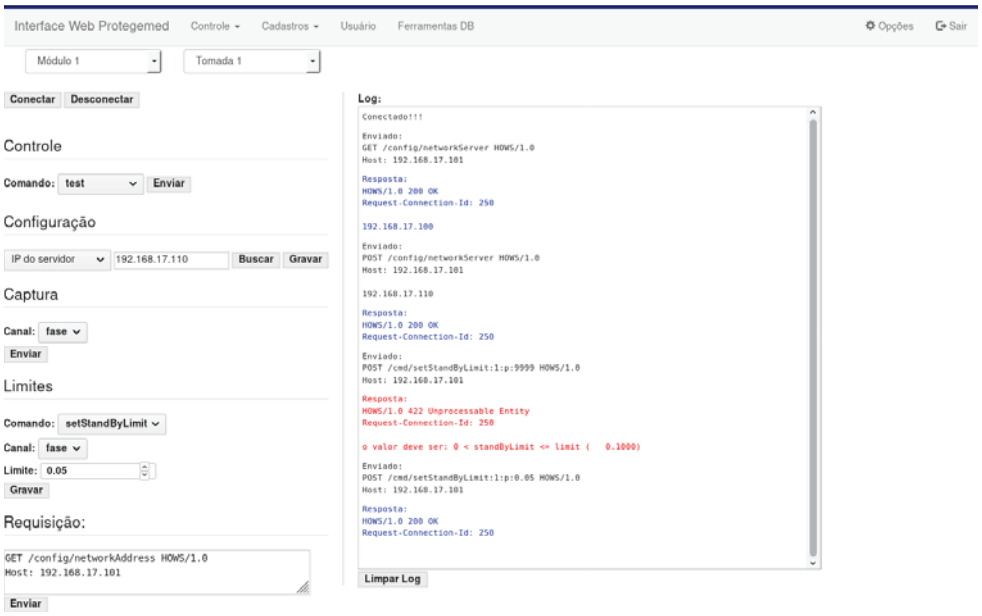


Figure 22: Página "Comunicação2" na interface web do software de apoio

7 Comandos implementados

Após a comunicação através do HOWS estar funcional, foram expandidos os comandos suportados, conforme a tabela 4. Os comandos **setGain** e **setOffset** são utilizados similarmente aos comandos **setLimit** e **setStandByLimit** originais. Com essas adições, é possível manipular todas as configurações implementadas no Mbed que são relacionadas às tomadas . O comando **read-config** chama a função Settings::readFile, que lê o arquivo de configuração gravado no sistema de arquivos. É o mesmo comando que é executado na inicialização do Mbed para que os valores padrões sejam utilizados. Certas vezes, ao ser executado, o comando causa o Mbed a ser reiniciado. Devido a isso, o comando foi desativado por padrão na interface web do software de apoio. Não se sabe o motivo exato do erro; pode ser que esteja relacionado à implementação nativa do sistema de arquivos do Mbed. O comando **write-config**, por outro lado, funciona corretamente. Ao ser executado, o comando chama a função Settings::writeFile, que grava, no arquivo de configuração, todos os valores que são configuráveis através do objeto Settings. Esse comando é necessário para que a troca dos valores das configurações seja persistida após o dispositivo ser reinicializado. O comando **print-config** chama as funções Settings::showParameters e Network::showConfig, que são responsáveis por mostrar, na saída serial, o valor de cada configuração gerais (do objeto Settings) e de rede (do objeto Network), respectivamente. Esses comandos são executados na inicialização, conforme a figura 18.

8 Considerações Finais

Nesse trabalho, foi realizada uma revisão do Protegemed, dos protocolos de rede utilizados e foi projetado um novo protocolo na camada de aplicação chamado HOWS. O protocolo foi implementado tanto no módulo embarcado quanto no servidor Ratchet. Seu funcionamento foi testado e validado, com testes de configuração e controle do módulo embarcado de forma remota. Um formulário para uso das funcionalidades no HOWS com o Ratchet foi disponibilizado ao usuário. Por fim, foram realizadas melhorias de "qualidade de vida" e foram expandidos as configurações e comandos suportados.

Os testes unitários foram integrais para encontrar *bugs* rapidamente e garantir o funcionamento correto da aplicação dados diferentes formatos de entrada. Mesmo tendo um parser pronto e validado na linguagem PHP, foram necessárias alterações não-triviais na hora de portar para a linguagem C++. O tempo de implementação no total foi subestimado, especialmente a parte de portar da linguagem PHP para a linguagem C++. Teria sido válido dedicar menos tempo para a implementação em PHP, mesmo que isso tivesse resultado em uma API mais complexa e/ou menos elegante.

O trabalho envolveu o conjunto de ferramentas utilizadas em uma aplicação web convencional, sendo elas o banco de dados, o servidor e a interface. Além disso, foi realizado o gerenciamento de memória manual e a portabilidade na parte de computação embarcada. No aspecto de redes de computadores, foi realizada uma revisão de protocolos pré-existentes, assim como a definição de um novo protocolo. Foi abrangido desde uma camada próxima ao hardware, utilizando C++, até partes em um nível mais alto, através do PHP e JavaScript, sendo todas as camadas interligadas através da rede. A possibilidade de atuar em todas essas diferentes áreas da computação foi o principal motivador pessoal para a realização deste trabalho. O protocolo HOWS foi definido de forma genérica, podendo ser utilizado em outros projetos de computação embarcada. Por exemplo, para configurar a velocidade de rotação do motor de um robô, ou para controlar um dispositivo da categoria de Internet das Coisas ("*IoT*").

Foi sugerida a implementação de diversas melhorias. Dentre elas, a de converter a forma que são especificados os argumentos dos comandos

Table 4: Comandos que podem ser executados através do HOWS

Comando	Parâmetro1	Parâmetro2	Parâmetro 3	O comando é novo no Mbed?
test	-	-	-	Não
reset	-	-	-	Não
print-config	-	-	-	Sim
read-config	-	-	-	Sim
write-config	-	-	-	Sim
capture	Id da tomada	Canal (fase ou fuga)	-	Não
setLimit	Id da tomada	Canal (fase ou fuga)	Valor	Não
setStandByLimit	Id da tomada	Canal (fase ou fuga)	Valor	Não
setGain	Id da tomada	Canal (fase ou fuga)	Valor	Sim
setOffset	Id da tomada	Canal (fase ou fuga)	Valor	Sim

Table 5: Configurações no Mbed que podem ser manipuladas através do HOWS

Chave	Parâmetro	A configuração é nova no Mbed?
networkServer	Valor	Sim
networkAddress	Valor	Sim
networkMask	Valor	Sim
networkGateway	Valor	Sim
eventLimit	Valor	Sim
outlet0	Valor	Sim
outlet1	Valor	Sim
outlet2	Valor	Sim

no formato em texto para uma forma mais similar ao formato em JSON. Ou seja, Ao invés de passar os argumentos separados pelo delimitador :, seria utilizado um mapa de chave-valor no corpo da mensagem, similar ao da figura 16. Poderia ser implementado utilizando o caractere = para separar a chave do valor e uma quebra de linha para separar cada um dos argumentos, similar à figura 23. Uma outra melhoria, dependente da primeira, seria a de converter os comandos setGain, setOffset, setLimit e setStandByLimit para chaves de configuração equivalentes. Ou seja, adicionar pseudo-chaves de configuração, tais como gain, offset, limit e standByLimit. A requisição de configuração seria recebida pelo Mbed e, internamente, os comandos equivalentes seriam executados. Dessa forma, os seus respectivos valores poderiam ser configurados similarmente às chaves já implementadas (e.g.: networkAddress). Também foi sugerido unificar o formulário de captura com o formulário de comando. Similarmente, foi sugerido unificar o formulário de configuração dos limites com o formulário de configuração principal. Dessa forma, por padrão, os campos extras (e.g.: tipo de canal) seriam escondidos por padrão e só apareceriam após a opção relevante (e.g.: "capture" ou "setLimit") ser selecionada. Isso foi proposto visando reduzir a quantidade de formulários e de espaço vertical utilizados na página. Ainda no software de apoio, poderiam ser validados os campos no envio de cada formulário, evitando que sejam passados valores inválidos ao Mbed. Também seria possível garantir a validação de cada argumento ao processar os comandos no Mbed, buscando impedir a tentativa de gravar, por exemplo, um texto a uma chave que seja do tipo inteiro.

Existem outros trabalhos no projeto Protegemed que estão sendo ou serão desenvolvidos. Dentre eles, a utilização da tecnologia RFID para identificar automaticamente o equipamento conectado na tomada e configurar a leitura dos valores da tomada (i.e.: ganho, offset, limite e limite de standby) de

```
1  Client request:
2
3    POST /config/gain HOWS/1.0
4    Host: 192.168.17.101
5
6    outlet=1
7    purpose=p
8    value=1
9
10 Server response:
11
12   HOWS/1.0 200 OK
```

Figure 23: Requisição de configuração do valor de ganho de uma determinada tomada (no formato de texto simples)

acordo com o equipamento. Foi proposta a utilização de um chip dentro da tomada do equipamento que, ao conectar o aparelho na tomada, enviaria um sinal para uma antena localizada dentro do módulo. Está em andamento um trabalho sendo desenvolvido por outro aluno de graduação, Gabriel Cena Kressin, para realizar a monitoração do módulo embarcado. Por exemplo, caso a temperatura do módulo esteja fora do aceitável, ou algo no módulo encontre-se em um estado inválido, o sistema poderia enviar um comando para o módulo ser reiniciado. Foi cogitado que esse sistema utilize o protocolo HOWS para realizar essa troca de mensagens. Por fim, busca-se implementar uma forma de atualizar remotamente o firmware do módulo utilizando o protocolo de rede TFTP (Sollins; 1992). No momento, a atualização funciona localmente (dos Santos; 2017).

Agradecimentos

À minha família pelo apoio, por prezar pelo ensino e por ter insistido na finalização do curso.

Ao meu orientador Marcelo Trindade Rebonatto pela presença e pela liderança ao longo deste trabalho. Foi crucial para o entendimento dos detalhes do

projeto e do Protegemed como um todo.

Ao meu gestor Tiago Moraes Ferreira pela compreensão nos diversos dias de minha ausência do projeto, que foram essenciais para que as implementações deste trabalho fossem materializadas.

Aos colegas da JOIN pela compreensão quando não pude ir às jantas e às reuniões.

References

ARM (n.d.). *Mbed OS 2 Handbook*. https://os.mbed.com/handbook/mbed-SDK.

dos Santos, J. C. (2017). *Estudo e implementação para atualização de hardware e firmware do protegemed*, Pós graduação em Computação aplicada, Instituto de Ciências Exatas e Geociências da Universidade de Passo Fundo.

Fette, I. and Melnikov, A. (2011). The websocket protocol, *RFC 6455*, RFC Editor. http://www.rfc-editor.org/rfc/rfc6455.txt.

Fielding, R. and Reschke, J. (2014a). Hypertext transfer protocol (http/1.1): Message syntax and routing, *RFC 7230*, RFC Editor. http://www.rfc-editor.org/rfc/rfc7230.txt.

Fielding, R. and Reschke, J. (2014b). Hypertext transfer protocol (http/1.1): Semantics and content, *RFC 7231*, RFC Editor. http://www.rfc-editor.org/rfc/rfc7231.txt.

Internet Assigned Numbers Authority (n.d.). *Hypertext Transfer Protocol (HTTP) Status Code Registry*, Internet Assigned Numbers Authority. https://www.iana.org/assignments/http-status-codes/http-status-codes.xhtml.

NXP (2016). *UM10360 LPC176x/5x User manual*. Rev. 4.1. https://www.nxp.com/docs/en/user-guide/UM10360.pdf.

Perego, M. and Rebonatto, M. T. (2014). Interface web para monitoramento de salas de cirurgia com uso do sistema protegemed.

Rebonatto, M. T. (2015). *Métodos para análise de correntes elétricas de equipamentos eletromédicos em procedimentos cirúrgicos e detecção de periculosidade aos pacientes*, PhD thesis.

Schmitz, M. A. (2017). *Comunicação bidirecional para plataforma embarcada do protegemed*, Pós graduação em Computação aplicada, Instituto de Ciências Exatas e Geociências da Universidade de Passo Fundo.

Sollins, K. (1992). The tftp protocol (revision 2), *STD 33*, RFC Editor. http://www.rfc-editor.org/rfc/rfc1350.txt.

Spalding, L. E. S., Carpes Jr., W. P. and Batistela, N. J. (2009). A method to detect the microshock risk during a surgical procedure, *IEEE Transactions on Instrumentation and Measurement* **58**: 2335 – 2342. https://ieeexplore.ieee.org/document/4797883.

Tanenbaum, A. S. and Wetherall, D. (2011). *Computer Networks*, 5th ed edn, Pearson Prentice Hall, Boston. OCLC: ocn660087726.

Tex (2016). *Tiva™C Series TM4C1294 Connected LaunchPad Evaluation Kit EK-TM4C1294XL*. Rev. C. http://www.ti.com/lit/ug/spmu365c/spmu365c.pdf.

Trentin, E. S. and Rebonatto, M. T. (2011). Interface web para monitoramento de salas de cirurgia com uso do sistema protegemed.