MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

1º Ten HUGO LEAL BRITO SALES
1º Ten JOÃO PEDRO DA COSTA RAMALHO DOS SANTOS

IMPLEMENTAÇÃO DE ALGORITMO PARA CÁLCULO DE ROTAS IP PARA O RDS-DEFESA

Rio de Janeiro
2019

INSTITUTO MILITAR DE ENGENHARIA

1º Ten HUGO LEAL BRITO SALES
1º Ten JOÃO PEDRO DA COSTA RAMALHO DOS SANTOS

IMPLEMENTAÇÃO DE ALGORITMO PARA CÁLCULO DE ROTAS IP PARA O RDS-DEFESA

Projeto de Fim de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: Frederico Tosta de Oliveira - M.Sc.
Co-Orientador: Prof. Leandro de Mattos Ferreira - M.Sc.

Rio de Janeiro
2019

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80 - Praia Vermelha
Rio de Janeiro - RJ CEP 22290-270

Sales, Hugo Leal Brito
Implementação de algoritmo para cálculo de rotas IP para o RDS-Defesa / Hugo Leal Brito Sales, João Pedro da Costa Ramalho dos Santos, orientado por Frederico Tosta de Oliveira e Leandro de Mattos Ferreira - Rio de Janeiro: Instituto Militar de Engenharia, 2019.

42p.: il.

Projeto de Fim de Curso (graduação) - Instituto Militar de Engenharia, Rio de Janeiro, 2019.

1. Curso de Graduação em Engenharia de Computação - projeto de fim de curso. 1. Cálculo de rotas IP. 2. Rádio definido por software. I. de Oliveira, Frederico Tosta . II. Ferreira, Leandro de Mattos . III. Título. IV. Instituto Militar de Engenharia.

INSTITUTO MILITAR DE ENGENHARIA

1 Ten HUGO LEAL BRITO SALES
1 Ten JOÃO PEDRO DA COSTA RAMALHO DOS SANTOS

IMPLEMENTAÇÃO DE ALGORITMO PARA CÁLCULO DE ROTAS IP PARA O RDS-DEFESA

Projeto de Fim de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: Frederico Tosta de Oliveira - M.Sc.

Co-Orientador: Prof. Leandro de Mattos Ferreira - M.Sc.

Aprovado em 03 de Outubro de 2019 pela seguinte Banca Examinadora:

Frederico Tosta de Oliveira - M.Sc. do IME - Presidente

Prof. Leandro de Mattos Ferreira - M.Sc. do IME

Prof. Marcelo Figueira de Vasconcelos - D.Sc. do IME

Prof. Ronaldo Ribeiro Goldschmidt - D.Sc. do IME

Rio de Janeiro
2019

Ao Instituto Militar de Engenharia, por toda nossa formação técnica e aos nossos familiares, por todo o apoio sempre.

AGRADECIMENTOS

Agradeço imensamente minha família por todo apoio e base desde sempre, e aos amigos por todas as experiências e convivência. Sou também extremamente grato ao IME e meus mestres pela formação acadêmica e pessoal. Por último, porém não menos especial, sou muito grato ao meu amigo João Pedro pelo companheirismo e amizade ao longo desses anos: que os laços dessa amizade se fortaleçam cada vez mais e essa parceria continue firme como sempre foi.

Hugo Leal Brito Sales

Agradeço, primeiramente, a Deus por tudo que Ele me proporcionou e proporcina até hoje. Agradeço à minha família por toda educação, amor e dedicação a mim concedidos: vocês são a base de tudo. Ao Instituto Militar de Engenharia e a todos os meus professores, agradeço pela excelente formação pessoal e profissional. E por fim, agradeço ao meu grande amigo Hugo Leal pela amizade e pelo comprometimento ao longo de todos os anos, especialmente durante o decorrer deste Projeto de Final de Curso.

João Pedro da Costa Ramalho dos Santos

“Não existe caminho para a felicidade. A felicidade é o caminho. ”

BUDA

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

CTEx	Centro Tecnológico do Exército
ETH-HMI	Ethernet - Human Machine Interface
ETH-MP	Ethernet - Módulo de Processamento
FO	Forma de Onda
IP	Internet Protocol
ISO	International Organization for Standardization
LAN	Local Area Network
OSI	Open System Interconnection
RDS	Rádio Definido por Software
RF	Rádio-frequência
TAP	Terminal Access Point
SCA	Software Communications Architeture
XML	eXtensible Markup Language

RESUMO

As comunicações são de extrema importância para a garantia da eficiência das Forças Armadas de um país. Elas permitem a comunicabilidade entre os diversos escalões da Força: desde os mais baixos e responsáveis pela operacionalidade até os mais altos e encarregados da coordenação e controle dos seus subordinados. É, portanto, essencial garantir sua integridade e bom funcionamento. O Exército Brasileiro está constantemente sendo modernizado e o uso de Rádio Definido por Software é um dos avanços que, junto ao software Planejador de Missões, abre portas para uma comunicação mais segura, tecnológica e eficiente. O objetivo do presente trabalho é contribuir para o projeto RDS-Defesa através da implementação de um algoritmo para calcular rotas IP estáticas, a fim de criar a tabela de rotas de cada rádio presente num esquema multirede desenhado no Planejador de Missões. Após geradas as tabelas, elas serão configuradas em cada rádio e a comunicação estará garantida de acordo com o esquema projetado.

ABSTRACT

The communications have utmost importance for ensuring the efficiency of a country's Armed Forces. They allow communicability between the various levels of the Force: from the lowest and more operational to the highest and responsible for the coordination and control of their subordinates. Therefore, it is essential to ensure its integrity and proper functioning. The Brazilian Army is constantly being modernized and the use of Software Defined Radio is one of the advances that, together with the Mission Planner software, opens doors for safer, more technological and efficient communication. The objective of this project is to contribute to the RDS-Defesa project by implementing an algorithm to calculate static IP routes in order to create the routing table of each radio present in a scheme created in the Mission Planner. After the tables are generated, they will be configured on each radio and the communication will be guaranteed according to the scheme designed.

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

Vive-se atualmente uma era em que o mundo está completamente conectado, no qual é possível se comunicar com pessoas do outro lado do planeta em questão de segundos. No entanto, essa agilidade nem sempre esteve presente no cotidiano das pessoas. Apesar de parecer impossível viver sem se comunicar à distância, era assim que os seres humanos viviam até o século XIX, quando se deu as invenções do rádio e do telefone (CALABRE, 2002).

No âmbito das Forças Armadas, essas invenções tiveram grande impacto nas guerras e nas operações militares, pois possibilitaram comunicação à distância entre tropas e uma atuação mais ampla do Comando e Controle.

Quando o rádio chegou ao Brasil na década de 1920 (SAMPAIO, 1984), os militares foram os primeiros a utilizar o equipamento e o fazem até os dias atuais. Com o avanço da tecnologia, a era dos rádios analógicos e digitais está cada vez mais chegando ao fim, e visando acompanhar esse progresso, as Forças Armadas estão desenvolvendo soluções tecnológicas em radiocomunicação, sendo o Rádio Definido por Software (RDS) uma vertente desse avanço. O projeto denominado RDS-Defesa, de responsabilidade do Exército Brasileiro, visa promover o aumento da interoperabilidade, reduzir os custos de desenvolvimento e possibilitar a agregação de novos serviços sem substituir o hardware, apenas por meio de atualização do software (DE PAIVA JUNIOR et al., 2012).

Atualmente, o RDS do Exército Brasileiro é configurado manualmente e permite a criação de redes IP. O roteamento dos pacotes é do tipo estático e é missão do oficial especializado em comunicações criar todas as configurações do rádio, incluindo os endereços IP de suas diversas interfaces e as tabelas de roteamento. Com o objetivo de simplificar a fase de preparação das redes rádio, automatizar a configuração dos equipamentos e reduzir a possibilidade de erro, foi criado o software Planejador de Missões (PRADO FILHO et al., 2017).

Esse programa permite que o usuário desenhe um esquema de rádios e redes e, após terminado, o Planejador gera arquivos XML de configuração de cada rádio presente no esquema. Porém, as tabelas de roteamento não são criadas e o usuário deve calculá-las manualmente. O desenvolvimento de um programa que calcule as tabelas é, portanto,

imprescindível para tornar a configuração ainda mais rápida, eficiente e menos suscetível a erros.

1.2 OBJETIVOS

O presente projeto tem como objetivo implementar um programa capaz de fornecer a tabela de rotas para cada RDS presente num esquema multiredes desenhado no software Planejador de Missões. Utilizando os arquivos XML de configuração de cada rádio, é montada uma estrutura de dados para representar a rede e o algoritmo é executado para gerar a tabela de rotas para cada RDS. Assim, é possível configurá-los e deixá-los prontos para a operação.

1.3 JUSTIFICATIVA

O presente trabalho é imprescindível para tornar mais rápida, automática e menos sujeita a erros a configuração dos Rádios Definidos por Software do Exército Brasileiro. Atualmente, utiliza-se o software Planejador de Missões para planejar uma rede de rádios e gerar os arquivos XML de configuração de cada rádio da rede. Porém, como o software não cria as tabelas de rotas, é necessário calculá-las manualmente para em seguida inserir cada uma no arquivo de configuração do rádio correspondente.

Quanto maior a rede, mais demorado e custoso é o cálculo das tabelas e muito tempo acaba sendo investido. Dessa forma, tornar o processo automático tem como objetivo acelerar a configuração dos rádios, evitando erros humanos durante o cálculo das tabelas e inserção das rotas nos respectivos rádios.

1.4 METODOLOGIA

A metodologia foi dividida em 4 fases, progredindo desde a análise de requisitos junto à equipe do projeto até a conclusão do programa desenvolvido.

Na fase 1 foi realizada uma visita ao CTEx para levantar dos requisitos do projeto. Nesta primeira fase, foi identificado o que o Planejador de Missões é capaz de fazer e também suas limitações. A equipe que trabalha no projeto apontou como principal limitação o fato de as tabelas de rotas não serem geradas pelo software e o presente trabalho foi proposto como forma de solucionar este problema. Os integrantes da equipe ainda sugeriram possíveis abordagens para a solução do problema.

Na fase 2 foi definida a abordagem a ser seguida e realizado o estudo acerca da fundamentação teórica do projeto, que trata dos algoritmos de estruturas de dados e da teoria de roteamento estático de pacotes. Foi definido que a idéia geral da solução é desenvolver um programa na linguagem Python que lê os arquivos XML de cada rádio de um esquema gerado pelo Planejador, identifica as redes diretamente conectadas a cada rádio e, dessa forma, monta uma estrutura de dados que representa a rede. Em seguida, é executado um algoritmo capaz de calcular os caminhos de custo mínimo e, baseado neste resultado, a tabela de rotas é montada.

Na fase 3 ocorreu o desenvolvimento do programa de acordo com as características definidas na fase anterior.

Na fase 4 foram realizados testes para verificar o correto funcionamento do algoritmo, além de incrementar a funcionalidade de escrever no XML as tabelas de rotas geradas, deixando o arquivo pronto para configurar o RDS.

1.5 ESTRUTURA

O texto do projeto possui a estrutura de capítulos de acordo com o especificado abaixo:

- Capítulo 2: apresenta a fundamentação teórica necessária para o entendimento do projeto, do problema e de sua solução.
- Capítulo 3: ilustra o problema, mostrando uma possível situação, as informações detidas e o objetivo a ser alcançado na situação em questão.
- Capítulo 4: utilizando dos conhecimentos abordados no capítulo 2, propõe uma solução para o problema em questão.
- Capítulo 5: conclusão do projeto.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo aborda a teoria necessária para entender o problema e a solução que será proposta.

2.1 REDES

Esta seção foi escrita baseada na seção *Reference Models* em Wetherall e Tanenbaum (2011b).

Com o objetivo de facilitar o processo de padronização e obter interconectividade entre máquinas de diferentes fabricantes, a ISO, acrônimo do inglês *International Organization for Standardization*, aprovou, no início da década de 1980, um modelo de arquitetura para sistemas abertos, visando permitir a comunicação entre máquinas heterogêneas e definindo diretivas genéricas para a construção de redes de computadores independente da tecnologia de implementação. Segundo Wetherall e Tanenbaum (2011a), esse modelo não é uma arquitetura de redes, pois não especifica os serviços e protocolos exatos que devem ser usados em cada camada. Ele apenas informa o que cada camada deve fazer.

O modelo foi denominado *Open Systems Interconnection* e é comumente chamado de OSI, servindo de base para a implementação de qualquer tipo de rede, seja de curta, média ou longa distância.

O modelo é composto por sete camadas e cada uma delas realiza determinadas funções. As camadas são: aplicação, apresentação, sessão, transporte, rede, dados e física.

O roteamento de pacotes é uma operação da terceira camada do modelo OSI (camada de rede) e designa o processo de reencaminhamento de pacotes através das redes. O processo se baseia no endereço IP destino e na máscara de rede do mesmo e é feita por dispositivos chamados de roteadores.

Este processo pressupõe que em cada roteador exista uma tabela de roteamento. Esta tabela deve conter uma entrada para cada rede destino existente, associada ao seu respectivo próximo salto. Ou seja, para qual endereço IP o pacote deve ser encaminhado para chegar na rede destino.

O roteamento dos pacotes pode ser configurado para ser estático ou dinâmico. O caso estático é recomendado para redes com um número limitado de roteadores, onde a estrutura da rede é simples. O administrador do sistema constrói cada tabela de roteamento e

a introduz em seu respectivo roteador. Para isso ele necessita ter conhecimento de toda a infraestrutura da rede. As tabelas de roteamento estático não se ajustam automaticamente a alterações da rede e, por isso, essa configuração deve ser utilizada em redes nas quais as rotas não sofram mudanças. Caso haja alguma modificação, as tabelas devem ser refeitas e os roteadoes, reconfigurados.e

Ao contrário do estático, o roteamento dinâmico é recomendado para redes complexas, de grande dimensão, com redundâncias de caminhos e cuja topologia pode sofrer alterações com frequência. Como o nome sugere, a tabela de roteamento é construída dinamicamente e recorre a protocolos de roteamento para conseguir as informações necessárias. Esses protocolos propagam as alterações através da rede para que os roteadores se mantenham atualizados. Uma vantagem do roteamento dinâmico é que os protocolos de roteamento podem resolver situações complexas de roteamento mais rápida e eficientemente que o administrador do sistema. Porém, o uso desses protocolos resulta numa maior sobrecarga dos roteadores e da rede.

2.2 RÁDIO DEFINIDO POR SOFTWARE

O RDS é uma das linhas de pesquisa mais recentes a respeito de rádios, visto que os benefícios de se ter um rádio cujas configurações são flexíveis e alteradas via software são bem atraentes, deixando os rádios analógicos e digitais comuns cada vez mais ultrapassados. Segundo Reis et al. (2012), define-se RDS como um rádio no qual algumas ou todas as funções da camada física são definidas por software. Isto significa que as características de operação do rádio podem ser modificadas em tempo de execução. Pode-se, por exemplo, mudar o padrão de comunicação e realizar ajustes às variações do canal.

Podemos citar, ainda de acordo com Reis et al. (2012), algumas vantagens do uso do RDS:

- São mais robustos a variações de temperatura e envelhecimento de seus componentes, pois o processamento é transferido para o domínio digital, deixando de ter seu desempenho atrelado à precisão dos componentes analógicos do rádio;
- Pode-se utilizar uma arquitetura comum para os rádios de forma a comoditizar a tecnologia (DE PAIVA JUNIOR et al., 2012), facilitando suporte e apoio logístico;
- Possui uma maior facilidade para desenvolver ferramentas de simulação e correção de erros, visto que seu desenvolvimento é em ambiente de software;

- Facilidade de manutenção e operação dos rádios, pois com essa arquitetura as modificações podem ser feitas sem que a infra-estrutura seja alterada.

O RDS projetado pelo Exército utiliza um padrão de *framework* de aplicações denominado *Software Communications Architeture*, também conhecido pela sigla SCA. O SCA foi desenvolvido pelo Departamento de Defesa dos Estados Unidos através do programa *Joint Tactical Radio System* e é um padrão bastante adotado para a construção de rádios militares, segundo de Paiva Junior et al. (2012).

Os rádios possuem sistema operacional Linux, o qual se conecta ao *device* de rede no interior do *framework* SCA por intermédio de uma comunicação IP virtual. Essa comunicação recebe dois endereços IP virtuais, denominados *Terminal Access Point* (TAP). Um deles é designado para a interface Linux e é o endereço pelo qual o usuário do rádio vai ser encontrado na rede do *preset*. O outro IP virtual é designado para o lado SCA, que é a entrada do *device* de rede.

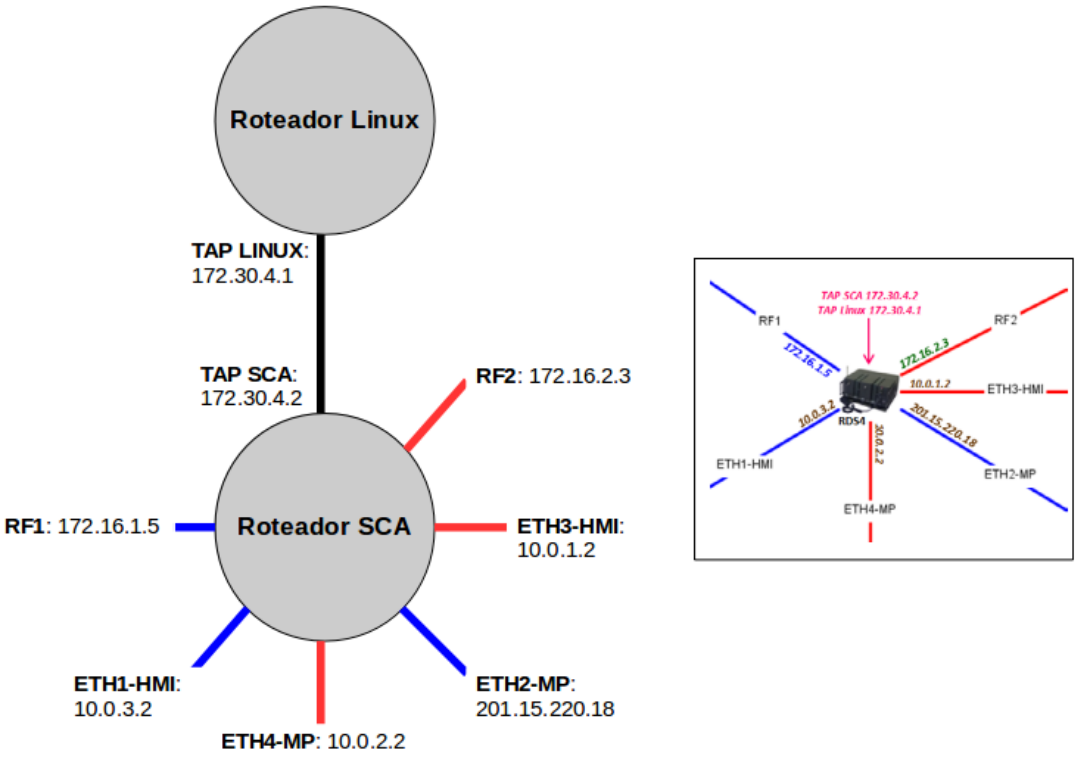

FIG. 2.1: Lógica interna ao rádio

Como mostra a figura 2.1, o rádio possui duas interfaces de rede sem fio (interfaces RF), cujas formas de onda são implementadas pelo próprio SCA, e quatro conexões Ethernet. Em geral, as interfaces RF são utilizadas para conectar os rádios entre si e as Ethernet, para conectar os rádios às redes LAN.

FIG. 2.2: RDS veicular desenvolvido pelo CTEx

2.3 XML

Segundo a W3C (2008), linguagem de marcação é um agregado de códigos que podem ser aplicados a dados ou textos para serem lidos por computadores ou pessoas. A linguagem XML, do inglês *eXtensible Markup Language*, é uma linguagem de marcação recomendada pela *World Wide Web Consortium* para a criação de documentos com dados organizados hierarquicamente, tais como textos, banco de dados ou desenhos vetoriais. O texto é estruturado em tags, como mostra o código 2.1.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<filmes>
    <filme id="1">
        <titulo>O XML veste prada</titulo>
        <genero>Aventura</genero>
        <genero>Documentario</genero>
        <elenco>
            <ator>Mark UPlanguage</ator>
            <ator>Mary well-Formed</ator>
            <ator>Sedna D. Atabase</ator>
        </elenco>
    </filme>
</filmes>
```

Listing 2.1: Exemplo de código XML

2.4 SOFTWARE PLANEJADOR DE MISSÕES

O Planejador de Missões é um software que permite a configuração, de maneira gráfica, de uma topologia de diversas redes e dos rádios RDS que as compõem. Na atual versão

do programa, dentro de um mesmo arquivo é possível criar várias abas denominadas de *presets*, como mostra a figura 2.3. Os *presets* são independentes entre si e em cada um deles o usuário cria uma topologia diferente contendo rádios RDS, redes LAN e redes FO (redes rádio). O mesmo RDS pode estar presente em mais de um *preset* e, neste caso, o rádio é configurado de uma forma distinta para cada *preset* em que estiver presente.

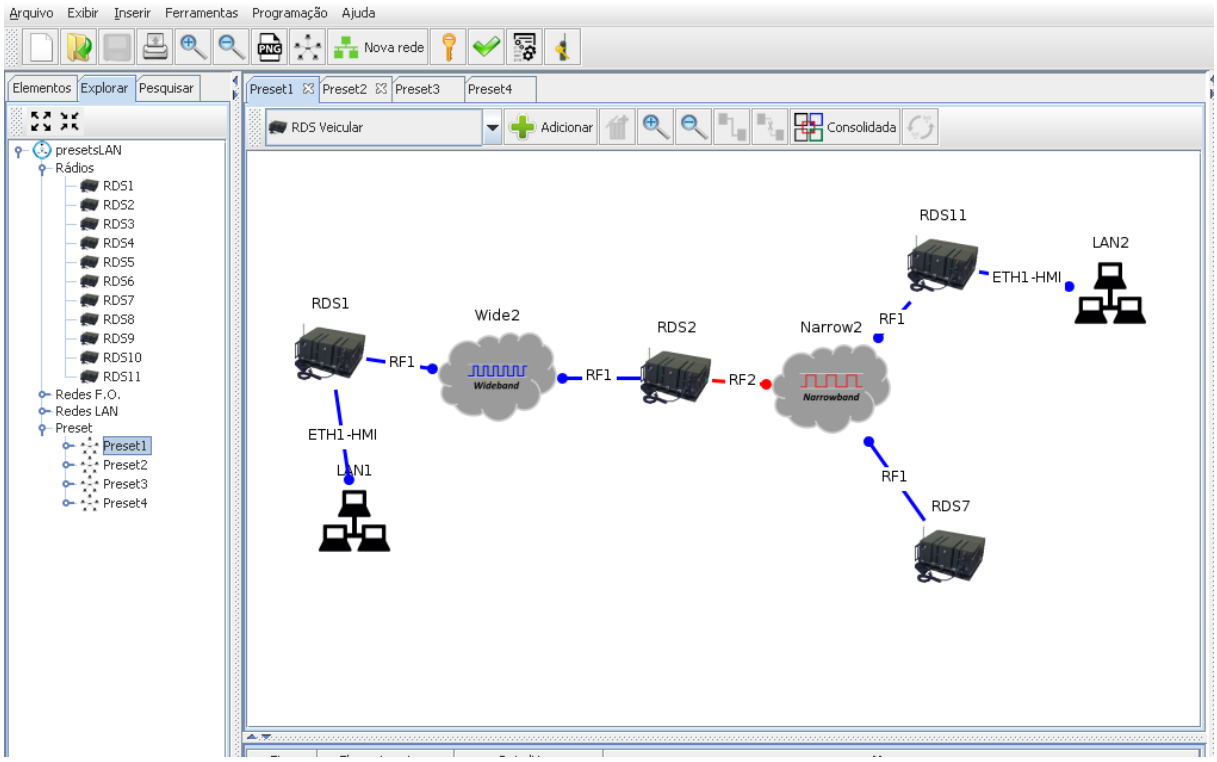

FIG. 2.3: Exemplo de esquema com múltiplos *presets*

Nas figuras 2.4, 2.5, 2.6 e 2.7 é possível ver a configuração de cada um dos 4 *presets* do esquema da figura 2.3.

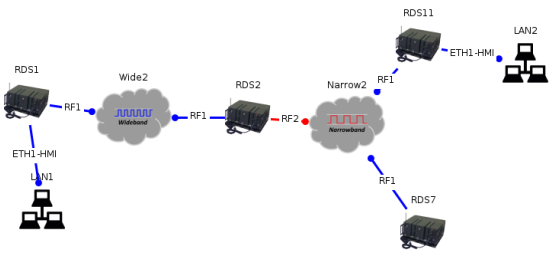

FIG. 2.4: *Preset* 1

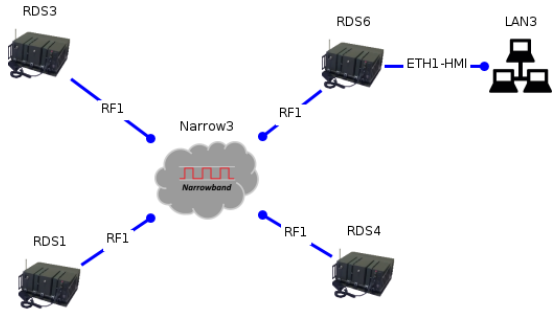

FIG. 2.5: *Preset* 2

Quando um rádio ou uma rede são inseridos em um *preset*, determinadas configurações desses elementos são preenchidas automaticamente pelo Planejador. O usuário pode mudar tais informações, se necessário. É importante salientar que ele deve atentar para

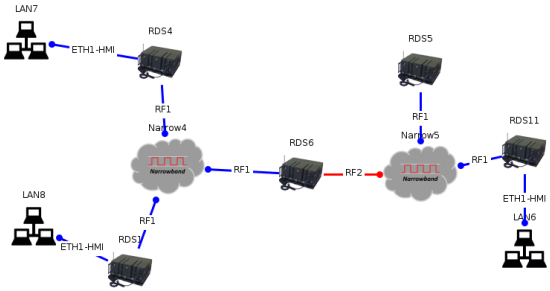

FIG. 2.6: *Preset* 3

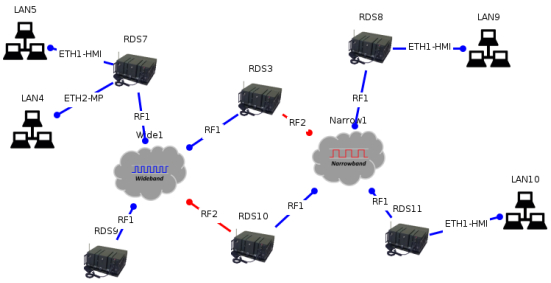

FIG. 2.7: *Preset* 4

que o endereçamento IP e o roteamento continuem coerentes apenas dentro de um mesmo *preset*, sem interferir ou ser interferido pelos demais. Afinal, não há comunicação entre rádios de *presets* diferentes. De forma mais prática, cada *preset* representa uma operação militar e, portanto, todas as informações dos RDS presentes nele têm que ser coerentes.

Quando é inserido num *preset*, o rádio automaticamente ganha um número inteiro X como identificador único, iniciando em 1 e incrementado em uma unidade a cada rádio inserido. Tal identificador é utilizado para a criação dos endereços IP do RDS. Quando o rádio é inserido, sua interface Linux da TAP recebe o endereço 172.30.X.2. O lado SCA da TAP recebe o endereço 172.30.X.1.

Ao ser criada, uma rede FO ganha um endereço classe C na faixa 172.16.X.0/24, sendo X um inteiro a partir de 1 e incrementado a cada nova rede FO inserida. O endereço do *gateway* é assumido como 172.16.X.1. Além disso, para cada rádio que for conectado, é atribuído um endereço IP a partir de 172.16.X.2, inclusive, a uma de suas interfaces RF.

Na criação de LAN, a rede recebe o endereço de classe C no formato 10.0.X.0/24, no qual X é um inteiro que começa em 0 e é incrementado a cada LAN inserida. O *gateway* da rede recebe o endereço 10.0.X.1 e cada rádio conectado recebe, em uma de suas interfaces Ethernet, um endereço IP sequencial a partir de 10.0.X.2.

Após um esquema ser montado, o Planejador de Missões permite a geração dos arquivos XML de configuração de cada RDS que, em seguida, serão transferidos para os respectivos rádios.

Na figura 2.8 é possível ver um esquema montado no Planejador de Missões, com as configurações de um dos rádios do *preset* sendo exibidas na parte lateral.

2.5 GRAFOS

Grafos são muito utilizados em problemas de redes, de roteamento e de cálculo de caminhos, visto que exprimem uma forma fácil e eficiente de representar várias dessas situações,

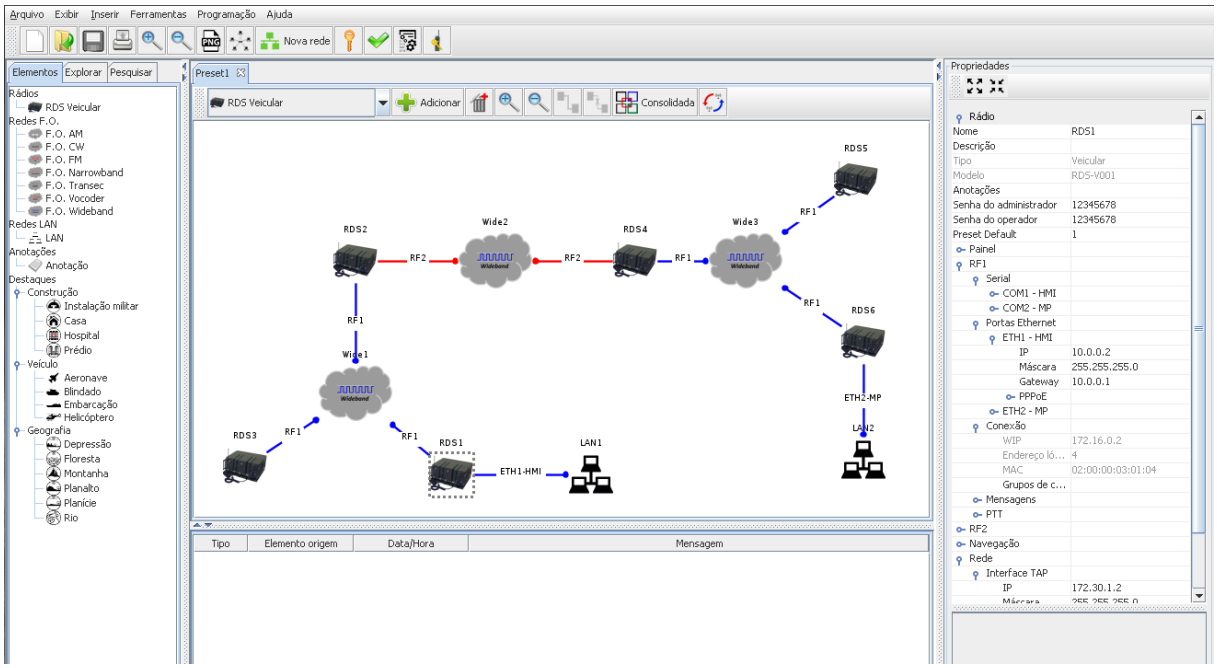

FIG. 2.8: Exemplo de esquema no Planejador de Missões

além de já existirem diversos algoritmos desenvolvidos para esse tipo de estrutura.

Segundo o autor Drozdek (2002), um grafo G = (V,E) é definido a partir de um conjunto não vazio V de vértices e um conjunto E de arestas, que pode ou não ser vazio. Cada aresta é um conjunto de dois vértices de V. Além disso, pode-se atribuir um número às arestas, o qual pode ser chamado de peso, custo, distância ou comprimento, de acordo com o contexto.

Outro detalhe importante é que as arestas podem ter direção e, quando isso acontece, o grafo é dito direcionado. Isto é, se um grafo for direcionado, uma aresta que liga os vértices A e B não necessariamente liga o vértice B ao vértice A. Se as arestas de um grafo não têm direção, o grafo é dito bidirecionado.

Na figura 2.9, temos o grafo que representa o esquema do Planejador de Missões mostrado na figura 2.8.

2.6 ALGORITMO DE DIJKSTRA

O algoritmo de Dijkstra, muito utilizado em cálculo de rotas e algoritmos de roteamento, é uma excelente ferramenta para auxiliar no problema do caminho mínimo. Ele considera os pesos das arestas do grafo e, para um determinado vértice, fornece uma árvore de caminhos mínimos com raiz neste vértice.

Dado um grafo G(V,E) orientado, com pesos positivos em suas arestas e um vértice

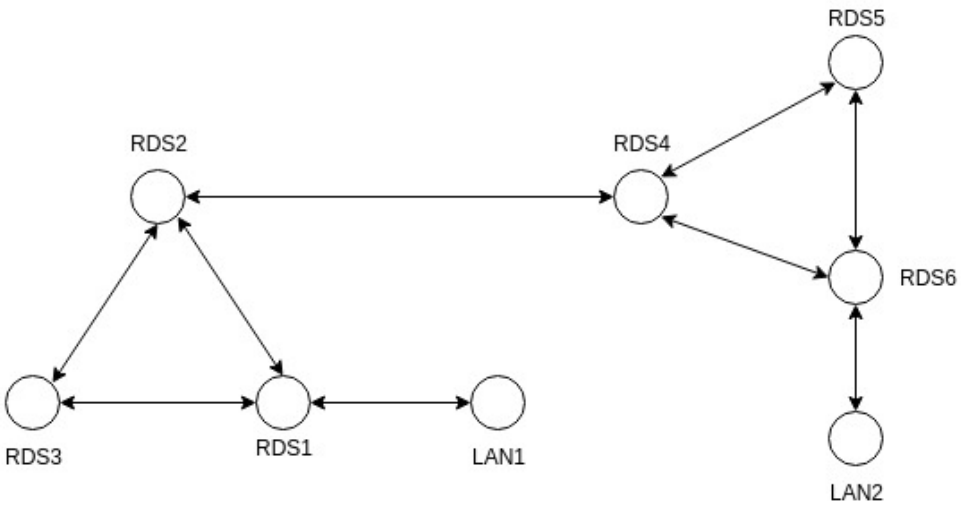

FIG. 2.9: Exemplo de grafo

v de G. A idéia da implementação do algoritmo de Dijkstra é o seguinte:

a) Atribua valor zero à estimativa do custo mínimo do vértice v (raiz da busca) e infinito às demais estimativas;

b) Atribua um valor qualquer aos precedentes (o precedente de um vértice t é o vértice que precede t no caminho de custo mínimo de v para t);

c) Enquanto houver um vértice aberto:

- Seja k um vértice ainda aberto cuja estimativa seja a menor dentre todos os vértices abertos;
- Feche o vértice k;
- Para todo vértice j ainda aberto que seja sucessor de k faça:
  - Some a estimativa do vértice k com o custo da aresta que une k a j;
  - Caso esta soma seja melhor que a estimativa anterior para o vértice j, substitua-a e anote k como precedente de j.

3 ILUSTRAÇÃO DO PROBLEMA

Este capítulo será destinado à ilustração do problema. Serão apresentadas todas as informações que se possui para um determinado esquema e o resultado que deseja-se alcançar com elas.

Seja o esquema da figura 3.1 um esquema montado no Planejador de Missões para o qual deseja-se calcular as tabelas de rotas de todos os rádios presentes nele. Neste caso, o esquema só possui um *preset*, que é mostrado na figura 3.1. Sabendo que todos os arquivos XML de configuração dos rádios têm a mesma estrutura, será utilizado o XML do RDS1 para ilustrar o problema.

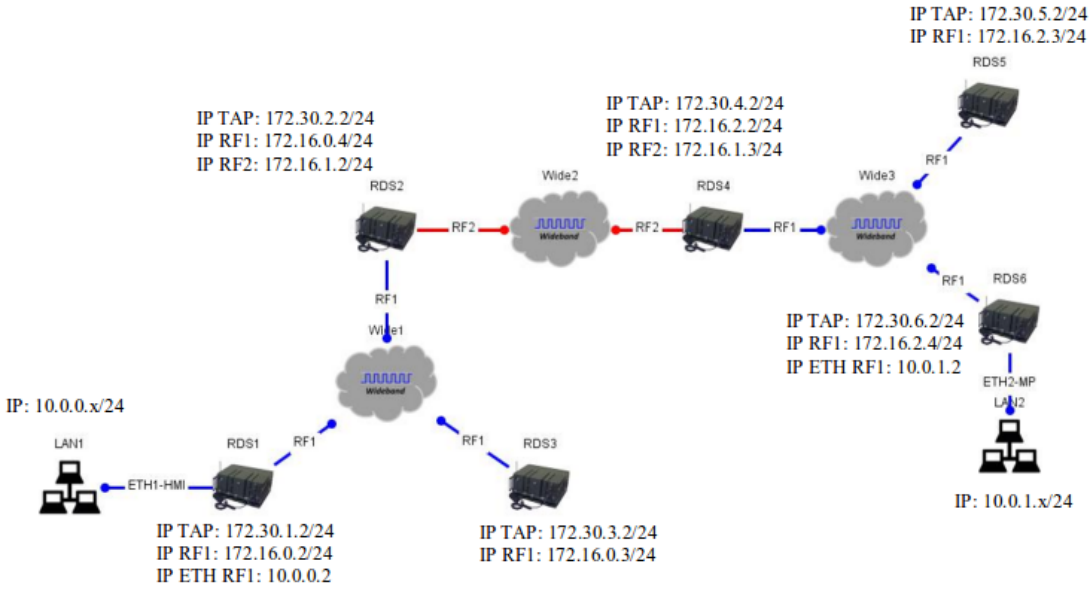

FIG. 3.1: Esquema exemplo

Conforme mostrado na figura 3.2, o XML de um rádio possui seu nome (RDS1), o endereço IP (172.30.1.2) e máscara (255.255.255.0) da sua interface TAP Linux e uma tag *<rf>* para cada interface RF usada.

Cada tag *<rf>* existente é identificada com um número inteiro. Dentro da tag *<rf>*, têm-se o endereço IP e a máscara desta interface, além dos endereços IP das interfaces RF dos demais rádios que estiverem conectados na mesma rede, como se pode observar na figura 3.3. E ainda, como mostra a figura 3.4, dentro da tag *<rf>* estão os endereços IP da TAP Linux dos rádios que estiverem conectados à mesma rede rádio, neste caso da forma de onda Wideband, que a interface RF do rádio do XML.

```
<radio xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="Radio.xsd">
  <versaoConfigurador>1.0-RC1</versaoConfigurador>
  <nome>RDS1</nome>
  <tipo>Veicular</tipo>
  <modelo>RDS-V001</modelo>
  <descricao/>
  <painel>...</painel>
  <usuario>...</usuario>
  <presetDefault>1</presetDefault>
  <presetAtual>1</presetAtual>
  <preset>
    <id>1</id>
    <descricao/>
    <rede>
      <ipTAPLinux>
        <nome>TAP1</nome>
        <id>9</id>
        <idConexaoSCA>DCETap1</idConexaoSCA>
        <IP>
          <enderecoIP>172.30.1.2</enderecoIP>
          <mascara>255.255.255.0</mascara>
          <gateway>172.30.1.1</gateway>
        </IP>
        <macAddress>02:00:00:00:01:71</macAddress>
      </ipTAPLinux>
      <roteamento>...</roteamento>
      <defaultGateway>0.0.0.0</defaultGateway>
    </rede>
    <rf>...</rf>
  </preset>
</radio>
```

FIG. 3.2: XML do RDS1

```
<IPSNDCF>
  <wIP>
    <id>11</id>
    <idConexaoSCA>
      <arp>SNDCF_ARP</arp>
      <ip>SNDCF_IP</ip>
    </idConexaoSCA>
    <IP>
      <enderecoIP>172.16.0.2</enderecoIP>
      <mascara>255.255.255.0</mascara>
      <gateway>172.16.0.1</gateway>
    </IP>
    <macAddress>02:00:00:03:01:04</macAddress>
  </wIP>
  <staticArpTable>
    <entrada>
      <macAddress>02:00:00:03:01:06</macAddress>
      <enderecoIP>172.16.0.4</enderecoIP>
    </entrada>
    <entrada>
      <macAddress>02:00:00:03:01:05</macAddress>
      <enderecoIP>172.16.0.3</enderecoIP>
    </entrada>
  </staticArpTable>
  <staticMulticastTable> </staticMulticastTable>
</IPSNDCF>
</wideband>
</formadeonda>
```

FIG. 3.3: Parte do trecho do XML contido na tag relativa à interface RF1 do RDS1

Como mencionado no capítulo 2, cada rádio tem duas interfaces RF. Ou seja, ele pode se conectar a duas redes rádio diferentes, cada uma com uma forma de onda, distintas ou não. As interfaces RF são identificadas pelos números 1 ou 2, resultando nas interfaces RF1 e RF2. Associada a cada RF há duas interfaces Ethernet, a Eth-MP e a Eth-HMI.

FIG. 3.4: Trecho do XML do RDS1 relativo à TAP dos vizinhos

Essas interfaces herdam o identificador da RF a qual correspondem. Em outras palavras, a Eth1-MP é a interface Eth-MP da RF1 e a Eth1-HMI é a interface Eth-HMI da RF1. O mesmo vale para as interfaces Ethernet da RF2.

Dessa forma, como visto na figura 3.5, o XML possui duas tags *<ethernet>* dentro de cada tag *<rf>*, uma para cada interface Ethernet associada à RF. Dentro da tag referente à interface Ethernet, estão contidos o endereço IP desta interface e a máscara da rede LAN a qual está conectada, desta forma é possível deduzir o endereço de rede da rede local em questão.

Com acesso a todas essas informações e utilizando ainda os arquivos XML dos demais RDS do esquema, é possível montar a tabela de rotas de qualquer um dos rádios. Na figura 3.6, está ilustrada a tabela de rotas que deseja-se alcançar para o RDS1 do esquema mostrado na figura 3.1.

FIG. 3.5: Trecho do XML do RDS1 relativo à interface Ethernet

Rota	Destino	Próximo salto	Interface	Custo do próximo salto
1	172.30.2.0/24	172.16.0.4	172.16.0.2	1
2	172.30.3.0/24	172.16.0.3	172.16.0.2	1
3	172.30.4.0/24	172.16.0.4	172.16.0.2	1
4	172.30.5.0/24	172.16.0.4	172.16.0.2	1
5	172.30.6.0/24	172.16.0.4	172.16.0.2	1
6	10.0.1.0/24	172.16.0.4	172.16.0.2	1
7	10.0.0.0/24	10.0.0.2	10.0.0.2	1
8	172.16.0.0/24	172.16.0.2	172.16.0.2	1
9	0.0.0.0/24	10.0.0.2	10.0.0.2	1

FIG. 3.6: Tabela de rotas do RDS1

4 SOLUÇÃO PROPOSTA

Para propor a solução do problema, é importante lembrar que os RDS dispõem de diversas interfaces as quais, quando são utilizadas, possuem endereços IP que pertencem a redes diferentes. Além disso, eles possuem tabelas de rotas e realizam o roteamento de pacotes entre as redes. Em outras palavras, além de serem estações terminais, que são materializadas pelos endereços IP das TAP, os rádios também funcionam como roteadores na camada de rede. Quando conecta duas redes diferentes, por exemplo, o rádio assume a função que um roteador assumiria numa rede IP comum.

Sabendo disso e considerando que as redes montadas normalmente são pequenas e não sofrem muitas alterações, será utilizado o roteamento estático. Outras razões para essa escolha é o fato de que são sabidas todas as informações necessárias sobre a rede através dos arquivos exportados pelo Planejador de Missões e, no caso do RDS, a taxa de comunicação de dados dos enlaces rádio é muito baixa, não sendo recomendada a utilização de um esquema de roteamento dinâmico por causa da grande troca de dados exigida pelo mesmo.

O problema a ser resolvido é, então, a partir de todos os endereços IP de usuários dos rádios (IP da TAP Linux) e de todas as redes presentes no preset, calcular as tabelas de rotas estáticas de cada RDS. Assim, será possibilitada a comunicação entre quaisquer endereços IP de todo o *preset*, independente de quantas redes o mesmo possua.

A implementação da solução foi feita na linguagem Python 3, pois é uma linguagem orientada a objeto e facilita a abordagem do problema através da criação de classes. Além disso, possui uma grande variedade de bibliotecas já desenvolvidas que são bastante úteis. A biblioteca *ipaddress*, por exemplo, permite a criação, a manipulação e a realização de operações com endereços IPv4, IPv6, interfaces e redes IP, o que foi essencial para a condução da solução do problema.

4.1 LEITURA DOS ARQUIVOS XML

Dado um esquema no software Planejador de Missões, é possível extrair as informações da rede através dos arquivos XML gerados para cada rádio. Os RDS precisam ser configurados antes das operações e são esses arquivos que contêm informações sobre as redes e rotas que deverão ser inseridos neles para realizar a configuração. Apesar de não conterem

as tabelas de rotas, esses arquivos contêm várias informações sobre o rádio e as redes nas quais ele está presente. Tais informações são úteis para montar o grafo que representa a rede de todos os *presets* e seus elementos.

Do arquivo XML de cada rádio, são extraídas informações acerca do RDS em questão, tais como nome, os *presets* em que está presente e endereço IP da sua interface TAP Linux. Também são coletados dados acerca de sua vizinhança, isto é, endereços das redes LAN às quais está diretamente conectado e endereços TAP dos rádios conectados às mesmas redes que ele. Esses dados são armazenados em atributos de objetos criados e variáveis globais, que serão mostrados na seção seguinte.

Para manipular os arquivos XML foi utilizado o módulo *xml.etree.ElementTree* da *Python Standard Library*, que implementa um simples e eficiente API para análise, criação e manipulação de arquivos XML. Este módulo interpreta o XML como uma árvore e possui duas classes: *ElementTree*, que representa o arquivo inteiro como uma árvore, e *Element*, que representa um nó simples na árvore. Utilizando essas duas classes é possível fazer operações de leitura e escrita em todo o XML. Além disso, como todos os arquivos gerados pelo Planejador de Missões possuem mesmo padrão na formatação das tags, é possível utilizar apenas um script que realize a leitura de todos, independente da quantidade de rádios ou do tamanho da rede.

4.2 CRIAÇÃO DA ESTRUTURA DE DADOS

Em geral, grafos são utilizados para representar redes porque terminais e roteadores constituem os vértices e as conexões entre eles são representadas pelas arestas, sejam uni ou bidirecionais. De fato, podemos ver nas figuras 4.1 e 4.2 um exemplo em que um esquema ilustrado no Planejador de Missões é representado por um grafo.

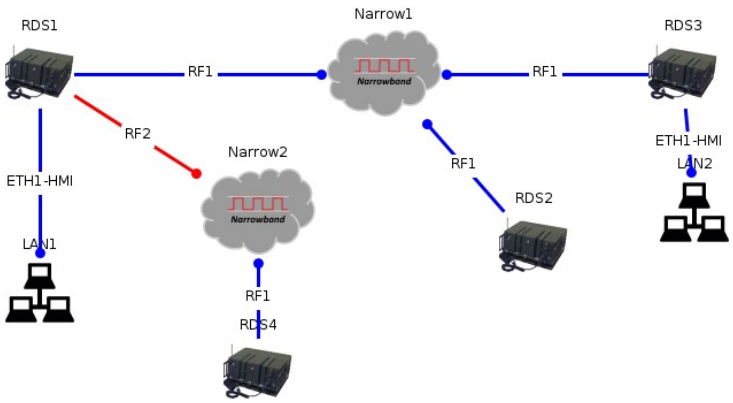

FIG. 4.1: Exemplo de esquema ilustrado no Planejador de Missões

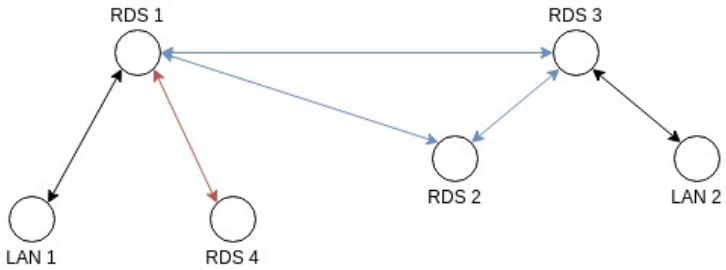

FIG. 4.2: Esquema da figura 4.1 representado por um grafo

Para a construção do grafo, foi criada uma classe específica para armazenar as informações necessárias e executar o algoritmo de Dijkstra posteriormente. No trecho de código 4.1, está mostrada a parte que diz respeito aos grafos.

Na composição da classe criada, têm-se: um conjunto de vértices ou *nodes* que será composto por objetos do tipo RDS e LAN; um dicionário de arestas ou *edges* que representarão as conexões entre os vértices e um atributo que armazena as distâncias ou pesos das arestas, que variam conforme o tipo de conexão (*Narrowband, Wideband ou LAN*), sendo esses valores variáveis globais definidas no início do código.

```
class Graph(object):
    def __init__(self):
        self.nodes = set()
        self.edges = defaultdict(list)
        self.distances = {}

    def add_node(self, node):
        self.nodes.add(node)

    def add_edge(self, node1, node2, distance):
        self.edges[node1].append(node2)
        self.edges[node2].append(node1)
        self.distances[(node1, node2)] = distance
        self.distances[(node2, node1)] = distance
```

Listing 4.1: Classe Graph

4.3 CRIAÇÃO DOS VÉRTICES

Para modelar o problema de forma mais congruente, foram criadas as classes RDS e LAN para compor os vértices do grafo, cada uma com os atributos correspondentes e úteis para a resolução. A classe RDS foi definida como mostra o código 4.2.

```
class RDS(object):
    def __init__(self, name, ip_tap_linux, interface_tap_linux,
   radio_xml):
        self.name = name
        self.ip_tap_linux = IPv4Address(ip_tap_linux)
        aux = IPv4Interface(interface_tap_linux)
        self.destination_network = aux.network
        self.routing_table = None
        self.lan_neighbors = list()
        self.intermediate_networks = list()
        self.xml = radio_xml
```

Listing 4.2: Classe RDS

Os rádios possuem atributos que são características individuais, tais como nome, endereço IP da TAP Linux e a máscara correspondente, que serão utilizados para calcular a rede TAP individual. Essa rede é armazenada na variável *destination.network*. Além disso, possuem o atributo *xml*, que armazena o arquivo XML correspondente ao rádio, e o atributo nomeado *routing_table*, o qual é inicializado vazio mas posteriormente é preenchido com a tabela de rotas calculada no *preset*.

Os atributos *lan_neighbors* e *intermediate_networks* dizem respeito à vizinhança do rádio. Todos são listas que armazenam, respectivamente, os endereços IP das redes LAN vizinhas e informações acerca dos IPs das redes intermediárias.

Todos esses dados preenchidos na criação de um objeto RDS são necessários para a construção do grafo e posterior cálculo da tabela de rotas.

Já a classe LAN é definida de forma mais simples, visto que possui menor complexidade e menos atributos. Basicamente, ela foi criada para seus vértices serem diferentes do tipo RDS. Ela foi definida como segue no código 4.3.

```
class LAN(object):
    def __init__(self, network_lan):
        self.destination_network = network_lan
```

Listing 4.3: Classe LAN

O único atributo que ela possui é o *destination.network*, também presente no RDS e que diz respeito ao seu endereço de rede. Este atributo será necessário para criar as arestas do grafo e montar a tabela de rotas posteriormente.

4.4 CRIAÇÃO DAS ARESTAS

Como já foi dito anteriormente na seção 4.2, as arestas representam as conexões entre dois elementos da rede, sejam eles RDS ou redes LAN. Dessa forma, todos que estiverem conectados no esquema devem possuir arestas para representar essa ligação no grafo. A seguir, será mostrado como é o processo de criação das arestas a partir das informações obtidas do XML.

Utilizando o esquema da figura 4.3 como exemplo, note que os IPs das interfaces estão ressaltados, pois eles são utilizados para preencher as matrizes *self.intermediate_networks*. Cada linha dessa matriz faz a correspondência entre os IPs da interface com os vizinhos conectados, sendo a primeira coluna o IP do próprio rádio, a segunda o IP da interface do vizinho, a terceira o custo da ligação e a quarta o próprio objeto RDS ou LAN vizinho. Para ilustrar, nas tabelas 4.1 e 4.2 estão representadas as matrizes dos RDS1 e RDS2, respectivamente, na forma como elas são inicializadas.

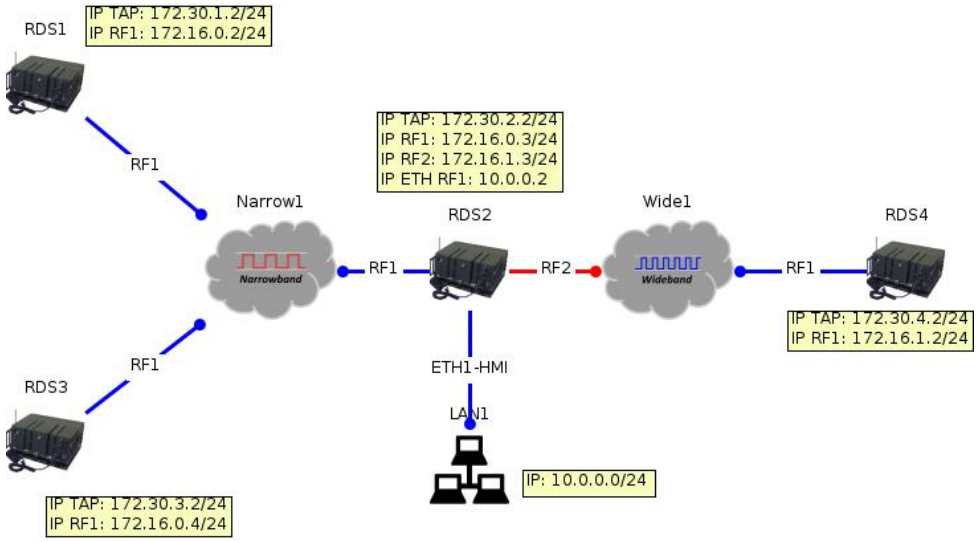

FIG. 4.3: Ilustração de esquema gerado no Planejador de Missões

$$\begin{pmatrix} 172.16.0.2 & 172.16.0.3 & narrowCost & null \\ 172.16.0.2 & 172.16.0.4 & narrowCost & null \end{pmatrix}$$

TAB. 4.1: *self.intermediate_networks* do RDS1

$$\begin{pmatrix} 172.16.0.3 & 172.16.0.2 & narrowCost & null \\ 172.16.0.3 & 172.16.0.4 & narrowCost & null \\ 172.16.1.3 & 172.16.1.2 & wideCost & null \\ 10.0.0.2 & 0 & lanCost & null \end{pmatrix}$$

TAB. 4.2: *self.intermediate_networks* do RDS 2

Existem dois tipos de arestas no problema em questão: as que representam conexões entre os rádios (sem fio) e as que representam conexões entre os rádios e as LANs (*ethernet*). Para a criação das arestas do primeiro caso, é preciso comparar as matrizes *self.intermediate_networks* de todos os rádios dois a dois de forma a descobrir se deve existir uma aresta entre eles ou não. Essa conferência é explicada a seguir.

Seja A a matriz do RDS1 e B a matriz do RDS2. Caso as igualdades $a_{i1} = b_{j2}$ e $a_{i2} = b_{j1}$ ocorram, com i e j índices quaisquer das matrizes, os rádios estão conectados. Assim, cria-se uma aresta entre seus vértices correspondentes no grafo com custo a_{i3} ou b_{j3}, e adiciona-se o objeto RDS2 em a_{i4} e o RDS1 em b_{j4}. Dessa forma, cada rádio terá uma matriz do tipo [*IP_próprio, IP_vizinho, Custo, Objeto_vizinho*].

As matrizes *self.intermediate_networks* dos RDS1 e RDS2 ficarão, então, como mostram as tabelas 4.3 e 4.4.

$$\begin{pmatrix} 172.16.0.2 & 172.16.0.3 & narrowCost & RDS2 \\ 172.16.0.2 & 172.16.0.4 & narrowCost & RDS3 \end{pmatrix}$$

TAB. 4.3: *self.intermediate_networks* do RDS 1

$$\begin{pmatrix} 172.16.0.3 & 172.16.0.2 & narrowCost & RDS1 \\ 172.16.0.3 & 172.16.0.4 & narrowCost & RDS3 \\ 172.16.1.3 & 172.16.1.2 & wideCost & RDS4 \\ 10.0.0.2 & 0 & lanCost & null \end{pmatrix}$$

TAB. 4.4: *self.intermediate_networks* do RDS 2

Após este procedimento, é necessário criar as arestas do segundo caso, entre os rádios e as LANS. Para esta situação, não é necessário fazer comparação entre matrizes *self.intermediate_networks*, apenas uma verificação na própria matriz do rádio basta, conforme será explicado a seguir.

Para cada rádio presente no grafo, todas as linhas da matriz *self.intermediate_networks* são varridas em busca das quais os termos da segunda coluna são *0*, o que significa que trata-se de uma LAN. Com essa condição satisfeita, o valor *0* é substituido pelo IP da interface LAN e é necessário fazer a correspondência desse IP com o objeto LAN em questão. Para isso, utiliza-se o atributo *self.lan_neighbors* do rádio, que armazena as LANs diretamente conectados a ele, para fazer a verificação. Quando os IPs de rede correspondem com o atributo *self.destination_network* de algum objeto LAN, uma aresta

é criada entre o RDS e a LAN com custo *lanCost* e o objeto é adicionado na quarta coluna da matriz na linha correspondente, como está mostrado na tabela 4.5.

$$\begin{pmatrix} 172.16.0.3 & 172.16.0.2 & narrowCost & RDS1 \\ 172.16.0.3 & 172.16.0.4 & narrowCost & RDS3 \\ 172.16.1.3 & 172.16.1.2 & wideCost & RDS4 \\ 10.0.0.2 & 10.0.0.2 & lanCost & LAN1 \end{pmatrix}$$

TAB. 4.5: *self.intermediate_networks* do RDS 2

Todo o procedimento descrito acima é realizado pelo método *check_connections()*, da classe *Graph*.

4.5 ALGORITMO DE DIJKSTRA

Com o grafo já montado, isto é, seus vértices e arestas bem determinados, é possível executar o algoritmo de Dijkstra para obter os caminhos mínimos para cada RDS e montar, assim, suas tabelas de rotas. O algoritmo utilizado está descrito no código 4.4 e seu funcionamento foi explicado no capítulo 2.

```
def dijkstra(graph, initial):
    visited = {initial: 0}
    path = {}

    nodes = set(graph.nodes)

    while nodes:
        min_node = None
        for node in nodes:
            if node in visited:
                if min_node is None:
                    min_node = node
                elif visited[node] < visited[min_node]:
                    min_node = node
        if min_node is None:
            break

        nodes.remove(min_node)
        current_weight = visited[min_node]

        for edge in graph.edges[min_node]:
```

```
            try:
                weight = current_weight + graph.distances[(min_node,
    edge)]
            except:
                continue
            if edge not in visited or weight < visited[edge]:
                visited[edge] = weight
                path[edge] = min_node

    return visited, path
```

Listing 4.4: Algoritmo de Dijkstra

O algoritmo deve ser executado uma vez para cada RDS da rede. Através da função *shortest_path_all()*, apresentada no código 4.5, são obtidos todos os destinos e os respectivos próximos saltos a partir do rádio em questão. Porém, o retorno da função é uma tabela com objetos RDS ou LAN, sendo necessário transformá-los em redes e endereços IP para constituir a tabela de rotas propriamente dita. O retorno do método *shortest_path_all()* é uma matriz de duas colunas, onde a primeira representa o objeto destino e a segunda representa o objeto para o qual o pacote deverá realizar o próximo salto. Nas tabelas 4.6 e 4.7 estão mostrados os retornos do método para o exemplo trabalhado na imagem 4.3.

```
def shortest_path_all(graph, origin):
    visited, paths = dijkstra(graph, origin)
    objects_table =list()
    i=0
    for destination in graph.nodes:
        if destination != origin:
            full_path = deque()
            _destination = paths[destination]

            while _destination != origin:
                full_path.appendleft(_destination)
                _destination = paths[_destination]

            full_path.append(destination)
            objects_table.append([destination, full_path[0]])
            i = i + 1
    return objects_table
```

Listing 4.5: Método *shortest_path_all()*

$$\begin{pmatrix} RDS2 & RDS2 \\ RDS3 & RDS3 \\ LAN1 & RDS2 \\ RDS4 & RDS2 \end{pmatrix}$$

TAB. 4.6: Retorno do método *shortest_path_all()* no RDS1

$$\begin{pmatrix} RDS1 & RDS1 \\ RDS3 & RDS3 \\ LAN1 & LAN1 \\ RDS4 & RDS4 \end{pmatrix}$$

TAB. 4.7: Retorno do método *shortest_path_all()* no RDS2

4.6 CRIAÇÃO DA TABELA DE ROTAS

A tabela de rotas é constituída basicamente por quatro colunas: [*rede destino, próximo salto, interface de saída, custo*]. Como mostrado nas tabelas 4.6 e 4.7, a partir do retorno da função *shortest_path_all()* basta substituir os objetos pelos seus endereços IP correspondentes. Isto é, endereço IP da rede destino na primeira coluna e endereço IP do próximo salto na segunda. Em seguida, basta adicionar à tabela as colunas referentes à interface de saída e ao custo.

Esse processo é feito pela função *routing_table_gen()*, que é um método da classe *RDS*. Na prática, os possíveis destinos para os rádios são apenas as redes TAP de cada rádio e as redes LAN. Assim, como cada objeto do grafo possui o atributo *destination_network*, a primeira coluna da tabela de rotas é obtida apenas substituindo os objetos pelos seus respectivos atributos *destination_network*.

Para preencher a segunda e a terceira colunas da tabela de rotas, o procedimento é realizado da seguinte maneira: seja RDSx um objeto da segunda coluna da matriz *objects_table* e A a matriz *intermediate_networks* do rádio para o qual está se gerando a tabela de rotas. Se ocorrer a correspondência a_{i4} = RDSx, então a coluna de próximos saltos deverá ser preenchida pelo a_{i2} e a coluna de interface de saída pelo a_{i1}. Além disso, adiciona-se na quarta coluna o custo da ligação que corresponde ao elemento a_{i3} da matriz *intermediate_networks*. Nas tabelas 4.8 e 4.9 estão mostradas as tabelas de rotas para o exemplo trabalhado na figura 4.3.

172.30.2.0/24	172.16.0.3	172.16.0.2	*narrowCost*
172.30.3.0/24	172.16.0.4	172.16.0.2	*narrowCost*
10.0.0.0/24	172.16.0.3	172.16.0.2	*narrowCost*
172.30.4.0/24	172.16.0.3	172.16.0.2	*narrowCost*
0.0.0.0/24	172.16.0.3	172.16.0.2	*narrowCost*

TAB. 4.8: Tabela de rotas do RDS 1

172.30.1.0/24	172.16.0.2	172.16.0.3	*narrowCost*
172.30.3.0/24	172.16.0.4	172.16.0.3	*narrowCost*
172.30.4.0/24	172.16.1.2	172.16.1.3	*wideCost*
10.0.0.0/24	10.0.0.2	10.0.0.2	*lanCost*
0.0.0.0/24	10.0.0.2	10.0.0.2	*lanCost*

TAB. 4.9: Tabela de rotas do RDS 2

Outra funcionalidade realizada pelo método *routing_table_gen()* é a de criação da rota *default*, que precisa existir caso a rede LAN 10.0.0.0 esteja presente no *preset*. Essa rota *default* é a EB Net. Essa funcionalidade foi requisitada pela equipe do projeto RDS-Defesa. Após este procedimento, a tabela de rotas estará com todas as informações necessárias para ser escrita no arquivo de configuração do RDS e é armazenada no atributo *self.routing_table*. A escrita é realizada pelo método *write_xml()* e é explicada na seção seguinte.

4.7 ESCRITA DA TABELA DE ROTAS

Após a tabela ser gerada, é preciso inseri-la no arquivo XML do respectivo rádio e, por isso, foi criado o método *write_xml()* na classe *RDS*. Para cada *preset* no qual um rádio está presente, uma tabela de rotas é gerada. Portanto, nos arquivos de rádios presentes em mais de um *preset*, existe mais de uma tag *<preset>*, como mostrado na figura 4.4, e mais de uma escrita precisa ser realizada.

Dessa forma, o método *write_xml()* recebe como parâmetro o identificador do *preset* que está sendo analisado para realizar a escrita da tabela de rotas dentro da tag do respectivo *preset*. A figura 4.5 mostra um trecho do arquivo XML antes da escrita enquanto a figura 4.6 mostra o mesmo trecho de código após a escrita pelo método *write_xml()*. Para cada linha da tabela de rotas, uma tag *<rota>* é criada. A estrutura da tag *<rota>* foi definida por integrantes do projeto RDS-Defesa e é mostrada no trecho de código 4.6.

```
<radio xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="Radio.xsd">
  <versaoConfigurador>1.0-RC1</versaoConfigurador>
  <nome>RDS1</nome>
  <tipo>Veicular</tipo>
  <modelo>RDS-V001</modelo>
  <descricao/>
  <painel>
    <luz>70</luz>
    <volume>70</volume>
    <alerta>true</alerta>
  </painel>
  <usuario>
    <senhaAdmin>12345678</senhaAdmin>
    <senhaOperador>12345678</senhaOperador>
  </usuario>
  <presetDefault>1</presetDefault>
  <presetAtual>1</presetAtual>
  <preset>...</preset>
  <preset>...</preset>
  <preset>...</preset>
</radio>
```

FIG. 4.4: Trecho de XML de um rádio presente em 3 presets

No caso mostrado anteriormente, a tabela de rotas só contém uma rota e, por isso, só uma tag <*rota*> foi criada.

```
<rota>
    <apelido></apelido>
    <destino>
        <IPRedeDestino></IPRedeDestino>
        <mascara></mascara>
        <nextHop></nextHop>
        <interface></interface>
        <custo></custo>
    </destino>
</rota>
```

Listing 4.6: Estrutura da tag <rota>

```
<preset>
  <id>2</id>
  <descricao/>
  <rede>
    <ipTAPLinux>
      <nome>TAP1</nome>
      <id>9</id>
      <idConexaoSCA>DCETap1</idConexaoSCA>
      <IP>
        <enderecoIP>172.30.1.2</enderecoIP>
        <mascara>255.255.255.0</mascara>
        <gateway>172.30.1.1</gateway>
      </IP>
      <macAddress>02:00:00:00:01:71</macAddress>
    </ipTAPLinux>
    <roteamento>
      <rip_ativado>false</rip_ativado>
      <maxSaltos>10</maxSaltos>
    </roteamento>
```

FIG. 4.5: Tag <*roteamento*> antes da escrita

FIG. 4.6: Tag *<roteamento>* após a escrita

Dessa forma, o método *write_xml()* verifica todas as tags *<preset>* do arquivo até achar a correspondente ao *preset* analisado e, então, cria uma tag *<rota>* e todas as suas filhas para cada entrada da tabela de rotas. Além disso, preenche o conteúdo das tags com as informações contidas na tabela de rotas.

No trecho de código 4.7 vemos um trecho do método *write_xml()* que mostra como as tags são criadas e preenchidas com seus respectivos valores retirados da tabela de rotas, que é armazenada no atributo *routing_table* do objeto RDS.

```
    ip = destino.makeelement('IPRedeDestino', atrib)
    destino.append(ip)
    ip_str = str(self.routing_table[i][0].network_address)
    ip.text = ip_str

    gateway = destino.makeelement('nextHop', atrib)
    destino.append(gateway)
    gateway_str = str(self.routing_table[i][1])
    gateway.text = gateway_str
```

Listing 4.7: Trecho do *write_xml()*

4.8 ABORDAGEM POR *PRESETS*

Para esquemas com múltiplos *presets*, é necessário tratá-los de forma separada. Dessa forma, primeiramente é preciso saber quantos *presets* existem ao todo. Sabendo que os

presets possuem como identificador um número natural que se inicia em 1 e é acrescido de uma unidade a cada novo *preset* criado, foi elaborada a função *get_presets_number()*, apresentada no trecho de código 4.8. Essa função recebe como parâmetro a lista com os arquivos XML de todos os rádios, lê o identificador de todos os *presets* existentes e define o maior deles como o número total de *presets*. Definida a quantidade total de *presets*, utiliza-se uma estrutura de repetição da linguagem Python e a lógica do programa é executada para cada identificador e, portanto, os *presets* são tratados de forma independente.

```
def get_presets_number(files):
    presets_number = 0
    for xml in files:
        root = ET.parse(xml).getroot()
        for preset_id in root.findall("preset/id"):
            if int(preset_id.text) > presets_number:
                presets_number = int(preset_id.text)
    return presets_number
```

Listing 4.8: Função *get_presets_number()*

Em outras palavras, toda a lógica do programa é executada dentro de uma estrutura de repetição que tem como parâmetro o identificador do *preset* analisado. Assim, todos os demais *presets* são ignorados e o problema é resolvido apenas para o *preset* em questão, desde a captação das informações, criação de um novo grafo e objetos RDS e LAN, cálculo e montagem das tabelas de rotas e escrita no arquivo de configuração dos rádios. Após resolvido para determinado *preset*, o laço repetitivo é reiniciado e a resolução do problema começa totalmente do início para o *preset* seguinte.

4.9 RENOMEANDO ARQUIVOS XML

Todos os arquivos XML gerados pelo Planejador recebem o mesmo nome, o que torna a identificação e associação do arquivo com o rádio mais difícil. Dessa forma, a função *rename_xml()* foi criada com o objetivo de renomear os arquivos XML para os nomes dos respectivos rádios. Como mostrado no código 4.9, a função recebe uma lista com todos os arquivos XML existentes, recupera o nome de cada rádio na tag <*nome*> do próprio arquivo e renomeia o XML.

```
def rename_xml(files):
    for xml in files:
        root = ET.parse(xml).getroot()
        xml_new = root[1].text + ".xml"
```

```
        os.rename(xml, xml_new)
```

Listing 4.9: Método *rename_xml()*

5 CONCLUSÃO

Ao longo da realização deste projeto, foi possível confirmar a relevância do cálculo da tabela de rotas para uma rede de rádios, pois a má realização desta tarefa pode levar toda uma operação ao fracasso. Desta forma, a automatização desse processo é de suma importância para acelerar o processo e diminuir bastante a possibilidade de erros.

O objetivo final do trabalho foi implementar um programa para ler os arquivos de configuração dos rádios definidos por software de um determinado esquema, montar a tabela de rotas de cada RDS e realizar a escrita dessa tabela no arquivo XML do respectivo rádio.

A entrega final foi aprovada por integrantes da equipe de engenheiros do projeto RDS-Defesa e traz como principal benefício a geração da tabela de rotas de forma automática. Esse benefício se torna muito evidente para esquemas com grande quantidade de rádios, como é comum de acontecer em operações militares e até em Organizações Militares de grande porte. O produto final ainda realiza a escrita da tabela de rotas, deixando os arquivos prontos para serem utilizados na configuração dos rádios. Dessa forma, o único esforço do usuário é executar um programa Python, tornando a configuração dos rádios mais rápida e eficiente.

A principal sugestão para trabalhos futuros e que a lógica do cálculo das tabelas de rotas seja inserida no código fonte do próprio Planejador de Missões. Dessa forma, quando os arquivos de configuração forem gerados, a tabela de rotas já estaria presente neles e, assim, os XML já estariam prontos para o objetivo final. Outra funcionalidade que pode ser implementada futuramente é a de rotas redundantes. Em outras palavras, para um mesmo destino final, o programa geraria mais de uma rota possível e aumentaria, assim, a robustez da rede de rádios, característica importante nas operações militares.

6 REFERÊNCIAS BIBLIOGRÁFICAS

CALABRE, L. A era do rádio. In: _____. **Introdução**. [S.l.]: Zahar, 2002.

DE PAIVA JUNIOR, N. M.; MARQUES, E. C.; DA SILVA, F. A. B. ; OTHERS. Introdução ao desnevolvimento de rádios definidos por software para aplicações de defesa. **XXX Simpósio Brasileiro de Telecomunicações**, v. 1, p. 1, 2012. Disponível em: <http://sbrt.org.br/sbrt2012/publicacoes/99644_1.pdf>. Acesso em: 13 de mai. de 2019.

DROZDEK, A. **Estrutura de Dados e Algoritmos em C++**. [S.l.]: Pioneira Thomson Learning, 2002.

PRADO FILHO, H. V.; GALDINO, J. F. ; MOURA, D. F. C. Pesquisa e desenvolvimento de produtos de defesa: Reflexões e fatos sobre o projeto rádio definido por software do ministério da defesa à luz do modelo de inovação em tríplice hélice. **RMCT**, v. 30, p. 6, 2017. Disponível em: <https://www.researchgate.net/profile/Aderson_Passos/publication/321310069_Special_Edition_on_Innovation_Management_Military_Journal_of_Science_and_Technology/links/5a1bee5b4585155c26ae105c/Special-Edition-on-Innovation-Management-Military-Journal-of-Science-and-Technology.pdfpage=7>. Acesso em: 14 de mai. de 2019.

REIS, A. L. G.; BARROS, A. F.; LENZI, K. G.; MELONI, L. G. P. ; BARBIN, S. E. Introduction to the software-defined radio approach. **IEEE Latin America Transactions**, v. 10, n. 1, p. 1156–1161, 2012.

SAMPAIO, M. F. **História do rádio e da televisão no Brasil e no mundo: memórias de um pioneiro**. [S.l.]: Achiamé, 1984.

W3C. Extensible Markup Language. Disponível em: <https://www.w3.org/TR/REC-xml/>. Acesso em: 26 maio. de 2019.

WETHERALL, J.; TANENBAUM, A. Redes de computadores. 5ª edição. **Rio de Janeiro: Editora Campus**, v. 5, p. 77, 2011.

WETHERALL, J.; TANENBAUM, A. Reference models. In: _____. **Redes de Computadores. 5ª edição**. Rio de Janeiro: Editora Campus, 2011. p. 41 – 53.