

Propostas de melhorias para o MiniEdit

Gabriela de Sousa Peres – 200037684
Depart. De Engenharia Elétrica,
Faculdade de Tecnologia, Universidade
de Brasília - Brasília, Brasil
200037684@aluno.unb.br

Henrique V. R. de Oliveira - 180064649
Depart. De Engenharia Elétrica,
Faculdade de Tecnologia, Universidade
de Brasília - Brasília, Brasil
180064649@aluno.unb.br

Henrique Rossi N. Pfitscher - 160008069
Depart. De Engenharia Elétrica,
Faculdade de Tecnologia, Universidade
de Brasília - Brasília, Brasil
160008069@aluno.unb.br

Resumo — Este documento trata sobre as melhorias implementadas no MiniEdit, desde comandos simples e básicos até os mais diversos. O MiniEdit é essencial para criação de topologias em redes definidas por software.

Palavras Chave - MiniEdit; programa; comando; topologia.

Abstract - This document covers the improvements implemented in MiniEdit, from simple and basic commands to the most diverse ones. MiniEdit is essential for creating topologies in software-defined networks.

Keywords - MiniEdit; program; command; topology.

I. INTRODUÇÃO

A crescente adoção do paradigma de Redes Definidas por Software (SDN) tem impulsionado a necessidade de ferramentas de simulação mais robustas, que possam atender tanto ao ensino quanto à pesquisa acadêmica. O Mininet, juntamente com sua interface gráfica Miniedit, é amplamente utilizado para a criação e emulação de redes SDN em ambiente controlado.

No entanto, apesar de sua facilidade de uso e ampla aceitação na comunidade acadêmica, a ferramenta carece de funcionalidades modernas que possam expandir suas capacidades e torná-la mais alinhada às demandas atuais de estudo e pesquisa. Este projeto tem como proposta aprimorar o Miniedit, adicionando recursos que facilitem o entendimento e o desenvolvimento de trabalhos acadêmicos nas disciplinas relacionadas a SDN. A implementação de novas funcionalidades permitirá aos estudantes e pesquisadores uma experiência mais completa na análise e gestão de redes simuladas, ampliando as possibilidades de experimentação e aprendizado.

II. OBJETIVO

A. Objetivo Geral

Aprimorar a ferramenta Miniedit para proporcionar um ambiente mais completo e didático, facilitando o aprendizado

e a elaboração de trabalhos acadêmicos em disciplinas que envolvem Redes Definidas por Software.

B. Objetivos Específicos

1. Desenvolver templates pré-definidos para topologias ordinárias de redes;
2. Criar um painel de gerenciamento com comandos básicos para facilitar a administração da rede, incluindo ferramentas como iperf e tcpdump.
3. Implementar um painel de visualização de fluxos de dados SDN através de dumpflow;
4. Adicionar funcionalidades de bloqueio de rotas e reconciliação automática;
5. Desenvolver um modelo para detecção de ciclos em redes e a implementação de algoritmos de roteamento baseados em vetor de distância (DV) ou estado de enlace (LS);
6. Verificar se o endereço IPv4 atribuído ao host pela janela do miniedit é válido;
7. Integrar ferramentas avançadas de captura e análise de pacotes, como Wireshark e tcpdump.

III. METODOLOGIA

O desenvolvimento das melhorias para o Miniedit será conduzido utilizando a linguagem de programação Python, em alinhamento com a estrutura original da ferramenta. O ambiente de testes e desenvolvimento será baseado no Mininet, que opera em sistemas Linux. A metodologia adotada será estruturada conforme descrito a seguir:

- **Análise e Planejamento:** Levantamento detalhado dos requisitos e funcionalidades desejadas, com base nos objetivos específicos do projeto.
- **Desenvolvimento Modular:** Cada funcionalidade será implementada de forma independente, garantindo a segurança e estabilidade de cada novo recurso antes da integração com o restante do sistema.
- **Testes Unitários e Integração:** Cada módulo será testado isoladamente para verificar sua funcionalidade, seguido de

testes de integração para garantir o correto funcionamento em conjunto com o Miniedit.

- Documentação e Validação: Produção de documentação detalhada para auxiliar estudantes e pesquisadores no uso das novas funcionalidades, além da validação do projeto por meio de testes em cenários acadêmicos.

Os códigos foram acrescentados a um diretório de desenvolvimento hospedado na plataforma Github, acessíveis por meio do link: <https://github.com/G4briel4Sous4/Miniedit-2.0.git>

IV. PROCEDIMENTOS EXPERIMENTAIS

A. Templates pré-definidos

Para o desenvolvimento de *templates* pré-definidos para as topologias foi necessário criar uma função dentro da classe Mininet() para que essa modalidade funcionasse. A função chamada de `load_template()` contém os seguintes argumentos: Selecciona os arquivos.py para serem escolhidos pelo usuário e serem carregados. Limpa a topologia assim que o mininet é aberto. Atualiza a interface gráfica de acordo com o template carregado e por fim adiciona os nós e os links.

As dificuldades encontradas no meio do percurso foram as substituições dos nós do grafo pelos componentes reais do MiniEdit. Essa troca não foi possível realizar devido a erros estruturais no próprio programa do miniedit.py. Entretanto, a representação gráfica pode ser visualizada da seguinte maneira:

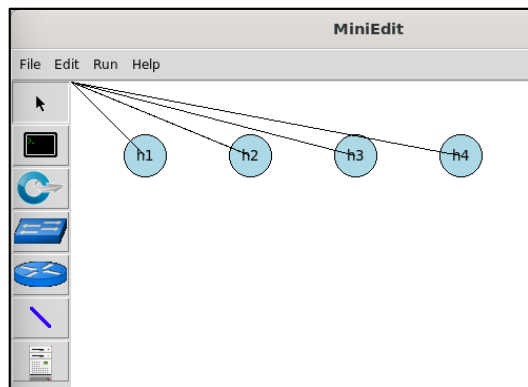


Figura 1. Topologia apresentada de forma visual.

Essa é uma forma de visualizar os programas carregados. Apesar de não ser como o esperado, a alegria é de tomar conta devido ao aprendizado e tamanho aprofundamento na disciplina.

B. Implementação IPERF e TCPCDump

Partindo para o segundo ponto, a implementação dos botões Ping, IPerf e TCPCDump, ao longo da jornada obteve erro em alguns momentos. O TCPCDump, claro, foi o mais

simples, devido a análise ser exclusivamente em um único host, já os outros pode-se observar pela imagem abaixo.

```

Topologia limpa,
*** Adicionando controladora
Unable to contact the remote controller at 127.0.0.1:6633
*** Adicionando switches
*** Adicionando hosts
*** Criando links
Template carregado com sucesso: ex2.py
Executando Ping de h1 para h2...
ping: None; Temporary failure in name resolution

Executando Ping de h3 para h4...
ping: None; Temporary failure in name resolution

Executando Ping de h2 para h3...
ping: None; Temporary failure in name resolution

Executando TCPCDump no host h4...
TCPCDump iniciado no host h4 com PID [1] 12299
12299

Executando Iperf entre h1 e h2...
error: Temporary failure in name resolution
    
```

Figura 2. Topologia apresentada de forma visual.

Com alguns ajustes foi possível obter êxito para o comando TCPCDump, como mostra a seguir.

```

Executando TCPCDump no host h4...
mininet@mininet-vms:~$ ps aux | grep tcpdump
tcpdump 12299 0.0 0.3 10900 5400 pts/7 S+ 11:49 0:00 tcpdump -i an
12299
Executando Iperf entre h1 e h2...
mininet@mininet-vms:~$ sudo tcpdump -i any
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked v1), capture size 262144 byt
es
11:58:33.224546 IP mininet-vm.ntp > prod-ntp-4.ntp4.ps5.canonical.com.ntp: NTPv4
Client, length 40
11:58:33.228955 IP localhost.57685 > localhost.domain: 1988* [Iau] PTR? 57.190.1
25.185.in-addr.arpa. (56)
11:58:33.230415 IP mininet-vm.41820 > b5d58408.virtua.com.br.domain: 29084* [Iau]
PTR? 57.159.125.185.in-addr.arpa. (56)
11:58:33.408445 IP b5d58408.virtua.com.br.domain > mininet-vm.41820: 29084 2/0/1
PTR prod-ntp-4.ntp4.ps5.canonical.com., PTR prod-ntp-4.ntp1.ps5.canonical.com.
(133)
11:58:33.409026 IP localhost.domain > localhost.57685: 1988 2/0/1 PTR prod-ntp-4
.ntp4.ps5.canonical.com., PTR prod-ntp-4.ntp1.ps5.canonical.com. (133)
11:58:33.443733 IP localhost.50142 > localhost.domain: 41467* [Iau] PTR? 53.0.0.
127.in-addr.arpa. (52)
11:58:36.225461 IP localhost.41383 > localhost.domain: 15226* [Iau] #? 1.ubuntu.
pool.ntp.org. (50)
    
```

Figura 3. Análise do arquivo gerado pelo comando TCPCDump.

Já para os comando IPerf e Ping com algumas modificações na construção da função dentro do miniedit foi possível obter êxito, como mostra a figura a seguir.

```

mininet@mininet-vms:~$ new_miniedit
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data:
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.79 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.800 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.805 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.840 ms
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time
201 ms min/avg/max/mdev = 0.486/0.807/0.914/0.100 ms

Executando TCPCDump no host h2...
TCPCDump iniciado no host h2 com PID [1] 14643
14643

Executando Iperf entre h1 e h2...
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: 415 KByte (default)

[ 3] local 10.0.0.1 port 37694 connected with 10.0.0.2 port
5001

[ ID] Interval      Transfer
Bandwidth

OK
    
```

Figura 4. Saída dos comando Ping e IPerf em conformidade.

O maior dos problemas veio em seguida. Em uma outra ocasião de teste das topologias, a máquina Virtual da aluna Gabriela encerrou-se sozinha por falta de espaço na memória RAM e não conseguiu implementar pequenas melhorias que restavam. São elas: a implementação visual para melhor entendimento e a padronização das topologias, ou seja, dos arquivos python.

Entretanto, apesar do ocorrido o projeto foi recuperado e compartilhado com os colegas e deram andamento no projeto.

C. Painel DumpFlow (visualização de dados SDN)

Com o fim de criar um painel *DumpFlow* para visualizar os fluxos presentes em cada um dos *switches* utilizados, acrescentou-se o seguinte código ao Miniedit.py.

```
def dump_flows(self):
    "Dump flow tables for all switches."
    if self.net is None:
        showerror(title="Error", message="Network is not running.")
        return

    self.dump_flows_panel.delete('1.0', END)
    for switch in self.net.switches:
        self.dump_flows_panel.insert(END, f'Flows for switch {switch.name}:\n')
        flows = switch.dpctl1('dump-flows')
        self.dump_flows_panel.insert(END, flows + '\n')
```

Código 1. Implementação painel DumpFlows

A tela demonstrativa destes dados aparece junto aos elementos do Miniedit desde sua execução, como pode ser visto na Fig. 5.

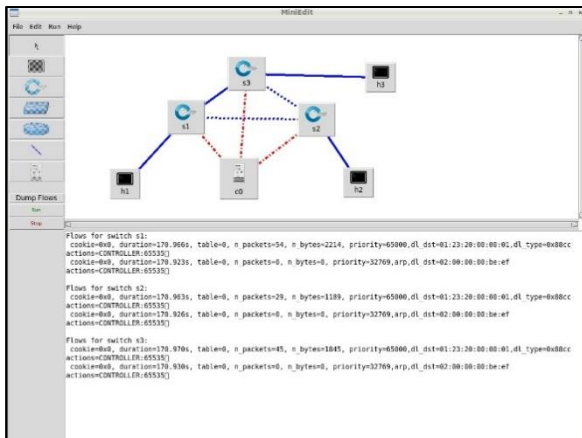


Figura 5. Painel Dumpflow dentro do Miniedit.

É notável na execução do mesmo os fluxos separados por aparelhos *switches* demonstrando o destino de cada fluxo registrado com os aparelhos *hosts* específicos.

D. Bloqueios de rotas e reconciliação

Para a derrubada de rotas e reconciliação, tal função já é implementada no Miniedit através das seguintes funções *link_up* e *link_down*, vistos abaixo:

```
def linkUp( self ):
    if ( self.selection is None or self.net is None):
        return
    link = self.selection
    linkDetail = self.links[link]
    src = linkDetail['src']
    dst = linkDetail['dst']
    srcName, dstName = src[ 'text' ], dst[ 'text' ]
    self.net.configLinkStatus(srcName, dstName, 'up')
    self.canvas.itemconfig(link, dash=())

def linkDown( self ):
    if ( self.selection is None or self.net is None):
        return
    link = self.selection
    linkDetail = self.links[link]
    src = linkDetail['src']
    dst = linkDetail['dst']
    srcName, dstName = src[ 'text' ], dst[ 'text' ]
    self.net.configLinkStatus(srcName, dstName, 'down')
    self.canvas.itemconfig(link, dash=(4, 4))
```

Código 2. Implementação função de derrubada e reconciliação de rotas

A utilização destas funções é possível através do seguinte passo a passo:

- 1) Clicar com o botão direito do mouse sobre um *link* (desde que não seja link com controladora);
- 2) Aparecerá a opção “Link Up” ou “Link Down”;
- 3) Clique na opção desejada;
- 4) O link caído ficará com o tracejado no lugar de uma conexão de linha sólida.
- 5) Para recuperar volte ao passo 1.

Na Fig. 6 é possível notar que os *links* entre s1 e s2 e entre s2 e s3 foram derrubados. Logo a conexão que uma vez era plena entre todos os *hosts* da rede ficou comprometida, permitindo somente a conexão entre h1 e h3.

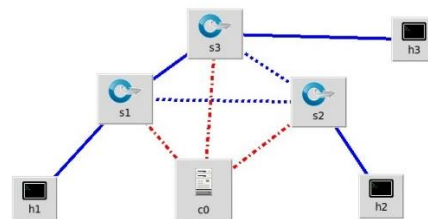


Figura 6. Derrubada de conexões no Miniedit.

Tais funções são fundamentais nos testes de adaptação a desconexão e redundâncias da rede simulada.

E. Verificação do endereço IPv4

Para verificar se o endereço IP atribuído ao host através do menu do miniedit é um endereço IPv4 válido, foi utilizada a biblioteca *ipaddress*, que é uma biblioteca em python usada para manipular endereços de IPv4 e IPv6; ela fornece recursos para criar, manipular e operar em endereços IPv4 e IPv6. Foram utilizadas, então, no código, a função *ip_address* da biblioteca *ipaddress*, que converte a string inserida pelo usuário que contém o endereço IP em um objeto IP correspondente (IPv4 ou IPv6), e a função *type*, que retorna o tipo do argumento que foi passado para ela.

A Fig. 7 mostra a implementação dessa verificação; nela, é possível ver que a lógica se baseia em um *try except*, no qual o código tenta transformar a string de entrada em um endereço IP por meio da função *ip_address* e, caso não seja possível realizar essa conversão, ou seja, caso a string não tenha o formato de um endereço IPv4 ou IPv6, o código entra no *except*, *printa* a mensagem “invalid address” no terminal e atribui ao host o endereço IP original; caso o código consiga realizar a conversão da *string* em um objeto do tipo *IPv4Address* ou *IPv6Address*, então é atribuído à variável *ip_type* o tipo do endereço IP da string convertida e, caso o tipo seja *IPv4Address*, o endereço é atribuído ao host; caso contrário, o código *printa* no terminal a mensagem “invalid address” e é atribuído ao host o endereço IP original.

```

if len(hostBox.result['ip']) > 0:
    try:
        ip_type = type(ip_address(hostBox.result['ip']))
        if ip_type is IPv4Address: newHostOpts['ip'] = hostBox.result['ip']
        else: print('invalid address')
    except: print('invalid address')
    
```

Figura 7 - Verificação de endereço IPv4

A Fig. 8 mostra um endereço de IP inválido inserido no campo *IP Address* do host *h1* e a Fig. 9 mostra a saída dessa atribuição de IP (houve falha na atribuição de IP, pois não era um IP válido). Já a Fig. 10 mostra uma atribuição de IP válido e a Fig. 11 mostra a resposta a essa atribuição (o endereço IPv4 foi atribuído ao host *h1* com sucesso).

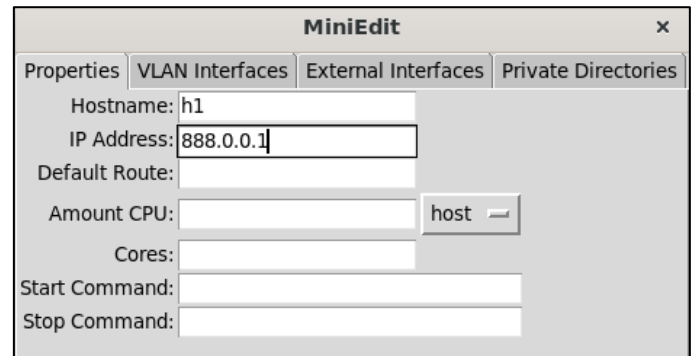


Figura 8. Endereço de IP inválido

```

topo=None
invalid address
New host details for h1 = {'nodeName': 1, 'sched': 'host', 'hostname': 'h1'}
    
```

Figura 9. Resposta à atribuição de IP inválido

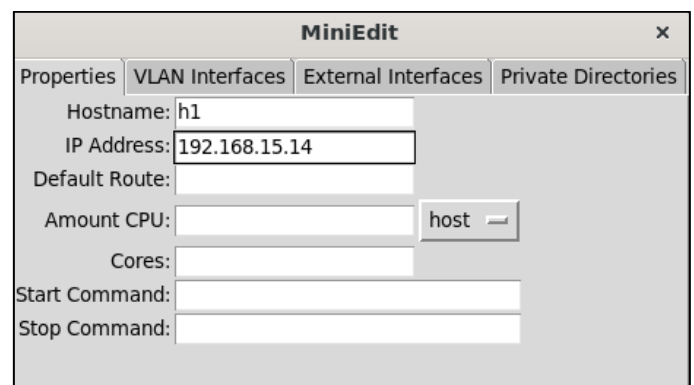


Figura 10. Endereço de IP válido.

```

New host details for h1 = {'nodeName': 1, 'sched': 'host', 'hostname': 'h1', 'ip': '192.168.15.14'}
    
```

Figura 11. Resposta à atribuição de IP válido.

F. Integração da ferramenta Wireshark

Para realizar a integração do Wireshark ao miniedit, foram criadas duas funções: a função *linkWireshark* e a função *showWiresharkMenu*. A função *linkWireshark* executa o comando que abre o wireshark a partir do terminal do nó selecionado e inicia a captura de pacotes no wireshark. Já a função *showWiresharkMenu* cria uma opção na interface gráfica do miniedit de iniciar a captura de pacotes no wireshark de um dos nós de um determinado enlace.

As Fig. 12 e 13 mostram os códigos das funções *linkWireshark* e *showWiresharkMenu*, respectivamente. Como se pode ver na figura 10, a função *linkWireshark* recebe como parâmetros *nodeName* e *nodeIntf*, sendo *nodeName* o nome do nó (host/switch) em que será realizada a captura dos pacotes, e *nodeIntf* a interface de rede do nó que será monitorada pelo Wireshark. A função então executa o Wireshark no nó especificado através de *self.net.get(nodeName).cmd("sudo wireshark -i " + nodeIntf + " -k &")*; o

`self.net.get(nodeName).cmd(...)` chama o método `cmd()` do Mininet para rodar um comando dentro do nó especificado e o comando `sudo wireshark -i <nodeIntf> -k &` abre o Wireshark, selecionando a interface `nodeIntf` para captura, com a opção `-k`, que faz com que a captura comece automaticamente.

A Fig. 13 mostra a definição da função `showWiresharkMenu`. Essa função exibe um menu para capturar tráfego de rede com o Wireshark, permitindo escolher entre duas interfaces conectadas. Como se pode ver na figura, primeiro a função verifica se `self.selection` e `self.net` estão definidos e, caso `self.selection` (um link selecionado) ou `self.net` (a rede simulada) seja `None`, a função retorna sem fazer nada. Em seguida, o código obtém os detalhes do link selecionado, através de `link = self.selection` e de `linkDetail = self.links[link]`, e, desses detalhes, são extraídos o nome do nó de origem e o nome do nó de destino do link, através de `src = linkDetail['src']`, `dst = linkDetail['dest']` e `srcName`, `dstName = src['text']`, `dst['text']`. Depois, são obtidas as interfaces conectadas ao enlace através de `intf = self.net.get(srcName).connectionsTo(self.net.get(dstName))`, que retorna uma lista de tuplas com as interfaces conectadas entre `srcName` e `dstName`, e `src_intf = intf[0][0]` e `dst_intf = intf[0][1]`, que são a interface do nó de origem e a interface do nó de destino, respectivamente. Em seguida, o código limpa o menu existente por meio de `self.linkWiresharkPopup.delete(0, 'end')`, que remove todos os itens do menu para atualizar a lista. Por último, o código adiciona opções no menu para capturar tráfego nas interfaces dos nós do enlace através de `self.linkWiresharkPopup.add_command(...)`, com o usuário podendo escolher entre realizar a captura na interface do nó de origem e na interface do nó de destino.

```
def linkWireshark( self, nodeName, nodeIntf ):
    if ( self.selection is None or
        self.net is None):
        return
    self.net.get(nodeName).cmd("sudo wireshark -i " + nodeIntf + " -k &")
```

Figura 12. Função `linkWireshark`

```
def showWiresharkMenu( self ):
    if self.selection is None or self.net is None:
        return

    link = self.selection
    linkDetail = self.links[link]

    src = linkDetail['src']
    dst = linkDetail['dest']
    srcName, dstName = src['text'], dst['text']
    intf = self.net.get(srcName).connectionsTo(self.net.get(dstName))
    src_intf = intf[0][0]
    dst_intf = intf[0][1]

    self.linkWiresharkPopup.delete(0, 'end')

    self.linkWiresharkPopup.add_command(label=str(src_intf), font=self.font, command=lambda: self.linkWireshark(srcName, str(src_intf)))
    self.linkWiresharkPopup.add_command(label=str(dst_intf), font=self.font, command=lambda: self.linkWireshark(dstName, str(dst_intf)))
```

Figura 13. Função `showWiresharkMenu`

A figura 14 mostra a opção de iniciar a captura de pacotes pelo wireshark em um enlace de uma topologia, ao apertar com o

botão direito nesse enlace; na figura, percebe-se que o usuário pode escolher entre realizar a captura na interface do nó de origem do enlace ou realizar a captura na interface do nó de destino. Ao clicar em alguma das duas opções, o miniedit abre o wireshark e inicia a captura dos pacotes na interface escolhida.

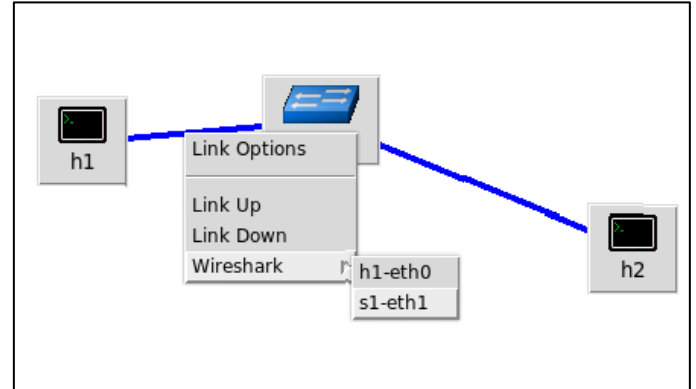


Figura 14. Opções de captura de pacote

G. Detecção de ciclos e implementação de spanning tree

Para implementar a detecção de ciclos e o algoritmo de *spanning tree* no código, foi utilizada a biblioteca *Networkx*, que é uma biblioteca python utilizada para a manipulação de grafos; através dela, é possível detectar se um determinado grafo possui ciclos e determinar a árvore geradora mínima desse grafo, através do uso do algoritmo *Kruskal*. Para poder realizar a manipulação de uma determinada topologia como se fosse um grafo, primeiro foram definidos, dentro da classe *Miniedit*, os atributos `self.G`, `self.cycleDetect`, `self.g` e `self.allowSpn`, sendo `self.G` um grafo vazio criado através do `nx.Graph()`, `self.g` e `self.cycleDetect` duas listas vazias e `self.allowSpn` uma string vazia, como mostra a figura 15. O grafo `self.G` representa a topologia da rede e nele serão adicionados todos os nós e enlaces da topologia montada, formando um grafo que representa a topologia; a lista `self.g` servirá como uma cópia do `self.G` para a realização do cálculo e implementação de uma nova árvore geradora após uma queda de link ou após uma subida de link; a lista `self.cycleDetect` servirá para a detecção de ciclos na topologia, sendo adicionados a ela todos os ciclos presentes na topologia; o `allowSpn` será utilizado para habilitar a execução da *spanning tree* na topologia.

```
self.G = nx.Graph()
self.cycleDetect = []
self.g = []
self.allowSpn = ''
```

Figura 15. Definição dos atributos `self.G`, `self.cycleDetect` e `self.g`

Após a definição desses atributos na classe *Miniedit*, foi implementado, dentro do método *start*, o detector de ciclos, como mostra a figura 16. O detector de ciclos consiste, basicamente, em detectar todos os ciclos simples presentes no grafo *self.G* por meio da função *nx.simple_cycles()* do *Networkx*, na conversão do resultado da função *nx.simple_cycles()* para uma lista de ciclos, por meio da função *list()*, e no armazenamento da lista de ciclos na variável *self.cycleDetect*. Caso nenhum ciclo seja encontrado, *self.cycleDetect* continuará sendo uma lista vazia; caso a topologia tenha ciclo, o programa *printa* no terminal a mensagem “Há ciclos na topologia” e, em seguida, a lista de todos os ciclos que há na topologia. Após a detecção de ciclos, é atribuída a *self.g* uma cópia do grafo *self.G*, que será usada futuramente para o cálculo e implementação da *spanning tree* no caso de quedas ou de subidas de link.

```
self.cycleDetect = list(nx.simple_cycles(self.G))
if self.cycleDetect:
    print("Há ciclos na topologia:")
    print(self.cycleDetect)
self.g = self.G.copy()
```

Figura 16. Detector de ciclos

Após a implementação da detecção de ciclos, foi implementado a lógica para aplicar o *Spanning Tree* usando o algoritmo de *Kruskal* (através da função *nx.minimum_spanning_tree(self.G)*). A figura 17 mostra essa implementação. A implementação inicia com o *if self.appPrefs['startSpn'] == '1' and self.cycleDetect:*, que verifica se a opção de implementar o *spanning tree* está habilitada e se há ciclos detectados na topologia; se essas duas condições forem verdadeiras, o código prossegue para o cálculo e aplicação do algoritmo de *spanning tree* na topologia. Em seguida, a linha *spn_tree = sorted(nx.minimum_spanning_tree(self.G).edges(data=True))* calcula a árvore geradora mínima da topologia usando o algoritmo de *Kruskal* e atribui à variável *spn_tree* a lista contendo todos os enlaces que fazem parte da árvore geradora mínima calculada. Depois, a linha *e_list = [e for e in self.G.edges]* cria uma lista contendo todas as arestas da topologia atual. A linha *print(sorted(nx.minimum_spanning_tree(self.G).edges(data=True)))* imprime no console a lista das arestas que pertencem à árvore geradora mínima. Em seguida, no loop *for e in e_list:*, o programa percorre todas as arestas *e* da topologia e, para cada aresta (*e[0]*, *e[1]*), ele verifica se a mesma está presente na árvore geradora mínima (*if (e[0] in s) and (e[1] in s)*): se ela estiver, *inSpnTree = 1*, e, caso ela não esteja, *inSpnTree* continua 0. Ainda para a aresta *e*, é verificado se *inSpnTree == 0* (se ela não faz parte da árvore mínima), e, caso seja satisfeita essa comparação, o link correspondente a essa aresta é desativado por meio de *self.net.configLinkStatus(e[0], e[1],*

'down'), o que garante que apenas os links essenciais para manter a topologia conectada permaneçam ativos.

```
if self.appPrefs['startSpn'] == '1' and self.cycleDetect:
    self.allowSpn = '1'
    spn_tree = sorted(nx.minimum_spanning_tree(self.G).edges(data=True))
    e_list = [e for e in self.G.edges]
    print(sorted(nx.minimum_spanning_tree(self.G).edges(data=True)))
    for e in e_list:
        inSpnTree = 0
        for s in spn_tree:
            if (e[0] in s) and (e[1] in s):
                inSpnTree = 1
        if inSpnTree == 0 and not(e[0] in self.controllers or e[1] in self.controllers):
            self.net.configLinkStatus(e[0], e[1], 'down')
```

Figura 17. Implementação da *spanning tree* pelo algoritmo *Kruskal*

Após a implementação da *spanning tree* no estado inicial da simulação, foi implementado o código que recalcula a *spanning tree*, para caso o usuário realize uma queda de link na topologia simulada; a figura 18 mostra essa implementação, dentro do método *linkDown*. Como se pode ver no código, o programa primeiro verifica se há ciclos detectados na topologia e se o *Spanning Tree* está ativado (*if self.cycleDetect and self.allowSpn == '1'*). Então, o programa tenta realizar a remoção do enlace que sofreu o *link down* da cópia do grafo *self.g* (*try: self.g.remove_edge(srcName, dstName)*); caso esse enlace exista no *self.g*, a remoção dele do grafo consegue ser realizada e, caso contrário, o programa sai do *try* e entra no *except*, que não faz nada. Caso o programa entre no *try*, a árvore geradora mínima é recalculada e todos os enlaces que não fazem parte dela são bloqueados (*link down*), enquanto que aqueles que estão nela são ativados (*link up*). A mesma lógica é implementada para o caso em que o usuário realize um *link up* na topologia, porém com a adição do enlace ao *self.g* e sem a necessidade de um *try except*, já que a adição de um enlace já existente em um grafo não gera erro. A figura 19 mostra a implementação dessa lógica para o caso de *link up*.

```
if self.cycleDetect and self.allowSpn == '1':
    try:
        self.g.remove_edge(srcName, dstName)
        spn_tree = sorted(nx.minimum_spanning_tree(self.g).edges(data=True))
        print(spn_tree)
        e_list = [e for e in self.g.edges]
        for e in e_list:
            inSpnTree = 0
            for s in spn_tree:
                if (e[0] in s) and (e[1] in s):
                    inSpnTree = 1
                    self.net.configLinkStatus(e[0], e[1], 'up')
            if inSpnTree == 0 and not(e[0] in self.controllers or e[1] in self.controllers):
                self.net.configLinkStatus(e[0], e[1], 'down')
    except:
        pass
```

Figura 18. *Spanning tree* para o caso de *link down*

```

if self.cycleDetect and self.allowSpn == '1':
    self.g.add_edge(srcName, dstName)
    spn_tree = sorted(nx.minimum_spanning_tree(self.g.edges(data=True)))
    print(spn_tree)
    e_list = [e for e in self.g.edges]
    for e in e_list:
        inSpnTree = 0
        for s in spn_tree:
            if (e[0] in s) and (e[1] in s):
                inSpnTree = 1
                self.net.configLinkStatus(e[0], e[1], 'up')
        if inSpnTree == 0 and not(e[0] in self.controllers or e[1] in self.controllers): self.net.configLinkStatus(e[0], e[1], 'down')

```

Figura 19. Spanning tree para o caso de link up

Depois de implementar a detecção de ciclos e o algoritmo *kruskal* de *spanning tree*, foram realizados testes para verificar o funcionamento dessas implementações. A figura 20 mostra o teste de detecção de ciclos e geração de *spanning tree*; nesse teste, foi montada uma topologia simples formada por dois hosts e três switches, sendo que os três switches formam um ciclo. Ao rodar essa topologia, a saída do programa mostrou que há um ciclo na topologia, indicou qual que é esse ciclo e montou a árvore geradora dessa topologia. Foi realizado, então, um ping do host h1 para o h2, como mostra a figura 21; o ping foi realizado sem perdas de pacote, indicando que o algoritmo de *spanning tree* foi implementado com sucesso.

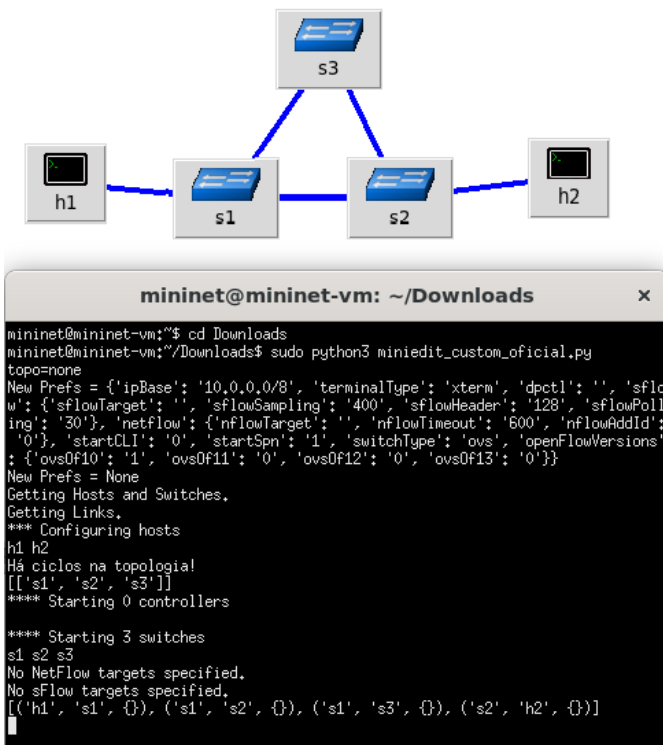


Figura 20. Detecção de ciclo na topologia e geração da sua *spanning tree*

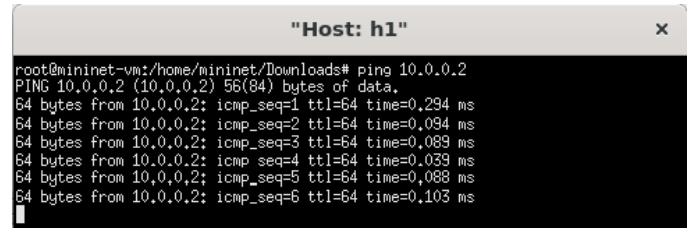


Figura 21. Ping realizado com sucesso

Em seguida, foi feito um *link down* no enlace entre o s1 e o s2, para testar a implementação do algoritmo para quedas de enlace. A figura 22 mostra esse teste; nela, nota-se que a árvore geradora mínima foi recalculada e printada no terminal. Foi realizado novamente um ping entre o h1 e o h2, como mostra a figura 23. O ping foi realizado com sucesso, mostrando que o programa conseguiu, de fato, recalculando a árvore mínima e redefinir a rota.

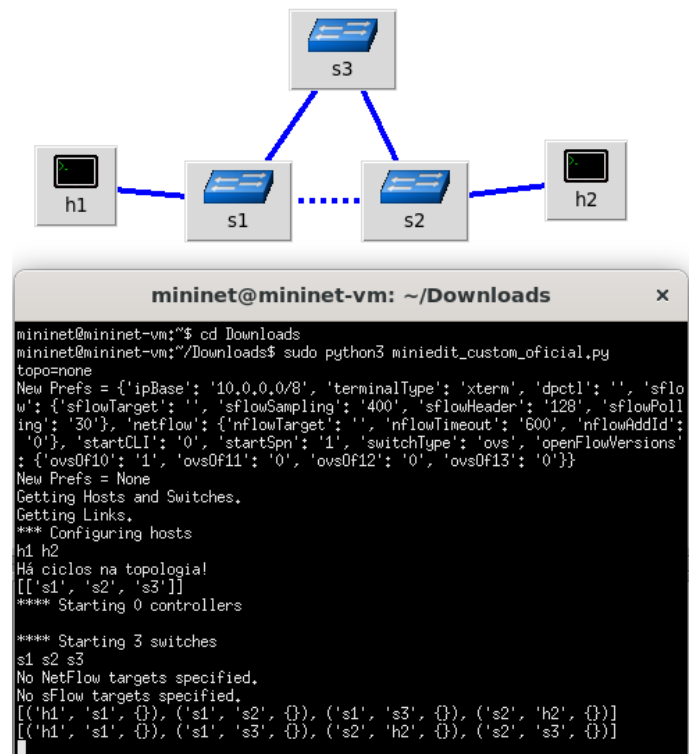


Figura 22. Árvore geradora mínima recalculada após queda de enlace

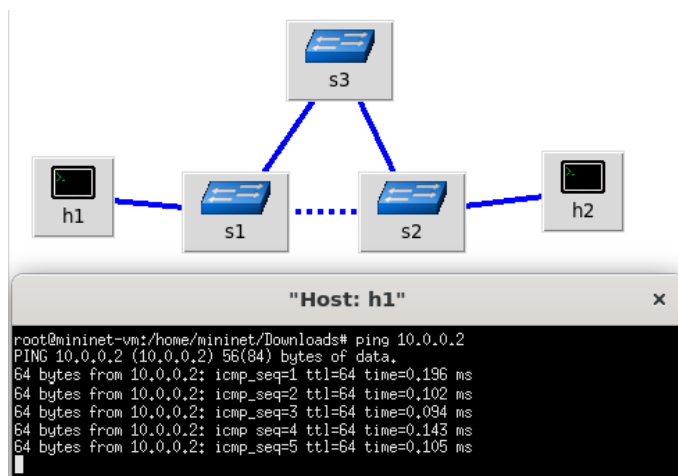


Figura 23. Ping feito com sucesso após a queda de enlace

V. REVISÃO BIBLIOGRÁFICA

A. Redes Definidas por Software (SDN)

Redes Definidas por Software (SDN) são uma abordagem inovadora para o gerenciamento e controle de redes de computadores. Diferente das redes tradicionais, nas quais o plano de controle e o plano de dados estão integrados em cada dispositivo de rede, as SDNs promovem a separação desses planos. O controle centralizado da rede é realizado por um controlador SDN, que toma decisões de roteamento e gerencia o tráfego de forma programável e dinâmica.[1] Os principais benefícios das SDNs incluem maior flexibilidade, facilidade de automação e melhor otimização de recursos.

Essa arquitetura permite que administradores de rede implementem políticas de encaminhamento personalizadas e utilizem protocolos como OpenFlow, que facilita a comunicação entre o controlador e os dispositivos de rede.[2]

B. Interfaces Southbound e Northbound em SDN

A arquitetura das Redes Definidas por Software (SDN) baseia-se na separação entre o plano de controle e o plano de dados, possibilitando a administração centralizada dos dispositivos de rede. Para que essa comunicação ocorra de maneira eficiente, dois tipos principais de interfaces são utilizados:

- Southbound Interface (SBI) – Responsável pela comunicação entre o controlador SDN e os dispositivos de rede.
- Northbound Interface (NBI) – Conecta o controlador SDN com aplicações e serviços que demandam inteligência de rede.

1) Southbound Interface (SBI)

Comunicação com os Dispositivos de Rede A Southbound Interface (SBI) permite que o controlador SDN envie instruções diretamente aos switches, roteadores e outros elementos da rede, ditando como os pacotes devem ser encaminhados e manipulados. Essa interface é fundamental

para garantir que as decisões de controle centralizado sejam implementadas no plano de dados. O protocolo OpenFlow é o exemplo mais notável de uma Southbound Interface, sendo um dos primeiros padrões a permitir que controladores SDN interajam com switches de maneira programável. A principal vantagem das SBIs é permitir que os dispositivos de rede sejam gerenciados de forma homogênea, independentemente do fabricante, desde que suportem o protocolo utilizado pelo controlador SDN.

2) Northbound Interface (NBI):

A Southbound Interface foca na comunicação com dispositivos de rede, a Northbound Interface (NBI) estabelece uma ponte entre o controlador SDN e as aplicações de alto nível, permitindo que desenvolvedores criem softwares que interajam diretamente com a infraestrutura da rede. As NBIs geralmente são baseadas em APIs RESTful, facilitando a integração com sistemas externos. Isso possibilita a criação de aplicações inteligentes, como:

- Balanceadores de carga dinâmicos – Ajustam o tráfego com base no consumo de largura de banda.
- Monitoramento de tráfego em tempo real – Ferramentas que analisam métricas de rede e geram alertas.
- Firewalls definidos por software – Implementam políticas de segurança de maneira centralizada e adaptável.
- Gerenciamento de Qualidade de Serviço (QoS) – Permite priorizar certos tipos de tráfego, como videoconferências ou VoIP.

3) Integração entre Southbound e Northbound Interfaces:

A comunicação entre a Southbound Interface e a Northbound Interface acontece dentro do controlador SDN, que atua como intermediário entre os dispositivos de rede e as aplicações externas. O controlador interpreta as solicitações das aplicações e traduz essas regras em comandos que os switches conseguem entender. A Fig. 14., a seguir, ilustra a relação entre essas interfaces dentro da arquitetura SDN:

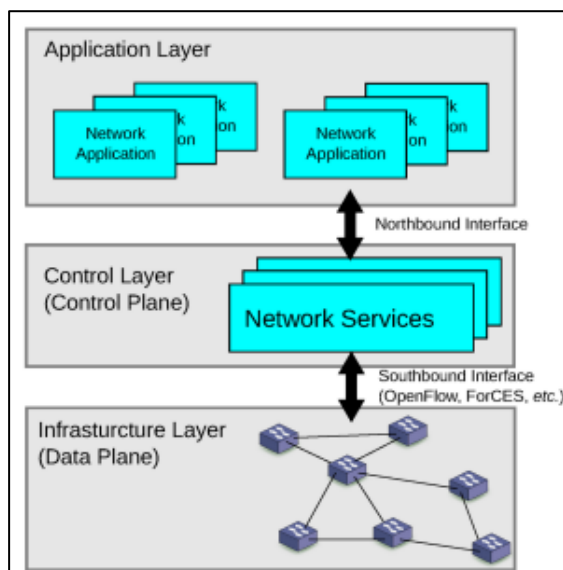


Figura 14. Função *showWiresharkMenu*

C. Mininet

O Mininet é uma ferramenta de emulação de redes amplamente utilizada para testar e desenvolver soluções SDN. Ele permite a criação de redes virtuais completas em um único sistema, incluindo switches, hosts e controladores SDN, possibilitando a execução de experimentos realistas sem a necessidade de hardware dedicado.[4] Entre as vantagens do Mininet, destacam-se sua facilidade de uso, compatibilidade com diversos controladores SDN e suporte a diferentes protocolos de comunicação. Ele é amplamente utilizado tanto no meio acadêmico quanto na indústria para o desenvolvimento e validação de novas tecnologias em redes programáveis.[5]

D. Spanning Tree

O Spanning Tree Protocol (STP) é um protocolo de rede fundamental para evitar loops em topologias de comutação Ethernet, garantindo caminhos redundantes sem causar tempestades de broadcast. Ele opera criando uma árvore lógica dentro de uma rede de switches, desativando seletivamente links redundantes para evitar ciclos. O STP utiliza o algoritmo do Protocolo de Eleição da Raiz, onde um switch raiz é escolhido com base na menor Bridge ID, e os demais switches calculam os caminhos mais curtos para essa raiz. Se um enlace ativo falhar, o protocolo pode reativar um link previamente bloqueado para restaurar a conectividade. Suas variantes, como RSTP (Rapid Spanning Tree Protocol) e MSTP (Multiple Spanning Tree Protocol), trazem melhorias na convergência e suporte a múltiplas VLANs, demonstrado na Fig. 15.

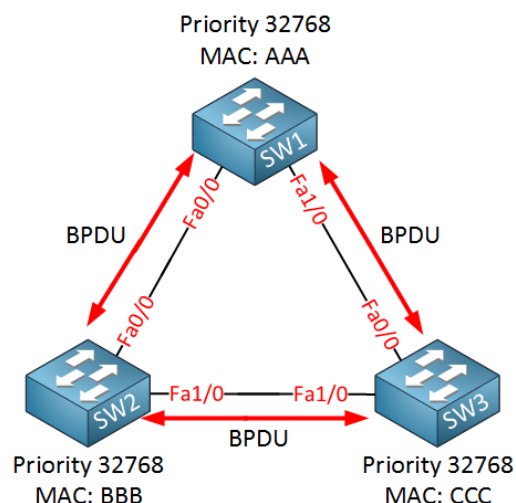


Figura 15. Demonstração Spanning Tree

No contexto de Redes Definidas por Software (SDN), o uso do STP se torna menos necessário, pois a inteligência da rede é centralizada no controlador SDN. Em SDN, a topologia pode ser gerenciada dinamicamente com caminhos otimizados e sem necessidade de protocolos tradicionais de prevenção de loops. Protocolos como OpenFlow permitem ao controlador SDN determinar regras de encaminhamento diretamente, evitando a necessidade de bloqueio de portas, como ocorre no STP. No entanto, em ambientes híbridos, onde switches tradicionais e SDN coexistem, pode ser necessário integrar STP para garantir compatibilidade e evitar loops em partes legadas da infraestrutura de rede.

E. Algoritmo de Kruskal

O algoritmo de Kruskal é um dos principais algoritmos utilizados para encontrar a Árvore Geradora Mínima (*MST – Minimum Spanning Tree*) de um grafo ponderado, garantindo que todos os nós sejam conectados com o menor custo possível e sem formar ciclos. Ele opera de forma gulosa (*greedy*), selecionando sempre a aresta de menor peso disponível e adicionando-a ao conjunto de arestas da árvore, desde que isso não crie um ciclo. A implementação típica do algoritmo utiliza a estrutura Union-Find (*Disjoint Set Union – DSU*) para verificar se a adição de uma aresta conecta dois componentes separados, garantindo assim que o resultado seja uma árvore válida.

Na área de redes, o algoritmo de Kruskal pode ser aplicado para otimizar a construção de topologias eficientes, como em redes de telecomunicações e roteamento de pacotes, reduzindo custos de infraestrutura. Em Redes Definidas por Software (SDN), ele pode ser usado pelo controlador para calcular caminhos eficientes, minimizando a latência e o consumo de banda ao estruturar a rede de forma hierárquica. Além disso, em ambientes onde a conectividade física precisa ser otimizada, como em malhas ópticas e redes de data centers, o Kruskal auxilia no planejamento de conexões com menor custo e maior eficiência energética.

VI. CONCLUSÕES

Este trabalho teve grande relevância para os estudantes envolvidos, proporcionando a oportunidade de aprimorar os conhecimentos em redes definidas por software (SDN) por meio da utilização prática de uma ferramenta gráfica, da identificação de lacunas de conhecimento e de monitoramento, e, por fim, da implementação de soluções.

Com base no embasamento teórico, foi possível compreender os indicadores necessários para o acompanhamento e gerenciamento de uma SDN. O objetivo geral de incorporar novas funcionalidades úteis ao contexto do Miniedit foi alcançado com sucesso, resultando em ferramentas valiosas para análise e gerenciamento de redes, como validação de IP, analisador de pacotes, derrubada de rotas, e comandos comuns como *ping* e *iperf*.

Dessa forma, o trabalho contribuiu significativamente para o desenvolvimento de habilidades práticas e teóricas, além de oferecer ferramentas que podem ser aplicadas em cenários reais de gerenciamento de redes.

REFERÊNCIAS BIBLIOGRÁFICA

- [1] L. A. J. Costa and M. Silva, “Redes definidas por software: uma abordagem sistêmica para o desenvolvimento de pesquisas em redes de computadores,” ResearchGate, 2014. [Online]. Available: <https://www.researchgate.net/publication/260346033> Redes Definidas por Software uma abordagem sistêmica para o desenvolvimento de pesquisas em Redes de Computadores
- [2] G. de Teleinformática e Automação UFRJ, “Redes definidas por software (sdn),” 2018. [Online]. Available: <https://www.gta.ufrj.br/ensino/eel878/redes1-2018-1/trabalhos-v1/sdn/>
- [3] W. Braun and M. Menth, “Software-defined networking using openflow: Protocols, applications and architectural design choices,” Future Internet, vol. 6, no. 2, pp. 302–336, 2014. [Online]. Available: <https://www.mdpi.com/1999-5903/6/2/302>
- [4] M. Team, “Mininet overview,” 2023. [Online]. Available: <http://mininet.org/overview/>
- [5] R. Tolentino and M. Almeida, “Análise da ferramenta mininet como ambiente de experimentação,” Simpósio de Pesquisa em Computação - SPC 2018, 2018. [Online]. Available: [https://spc.unifesspa.edu.br/images/SPC 2018/ Tolentino.pdf](https://spc.unifesspa.edu.br/images/SPC%202018/Tolentino.pdf)