

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Instituto de Informática
Disciplina INF01002 - Protocolos de Comunicação

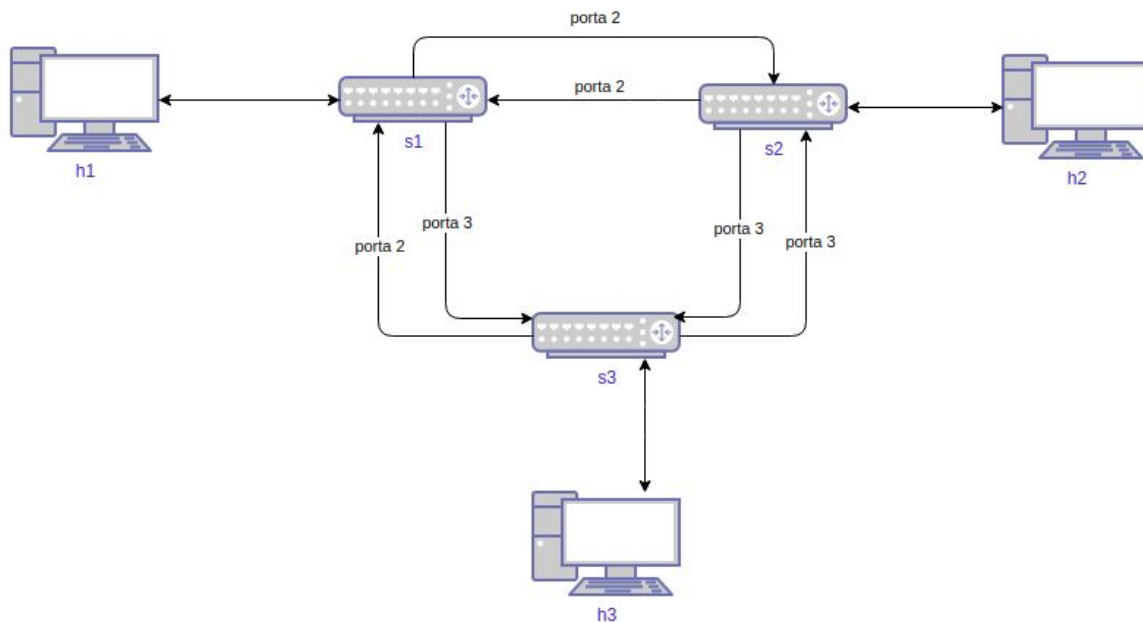
Alunos: André Dexheimer Carneiro
Rubens Ideron dos Santos

Matrícula: 00243653
00243658

Trabalho Final Parte 1 - Documentação

1. Lógica de Funcionamento do Mecanismo

Tomamos como base o exemplo de aplicação P4 (basic.p4) juntamente com a infraestrutura (Mininet, aplicações de transmissão e recebimento Python) fornecidas pelo professor, no qual a seguinte topologia de rede é adotada:



Todos os roteadores s1, s2 e s3 rodam o programa basic.p4. As aplicações de transmissão e recebimento Python rodam nos nodos clientes h1, h2 e h3.

Isto posto, modificamos o arquivo basic.p4 para incluir um mecanismo de *In-Network Telemetry* que funciona com cabeçalhos entre a camada de rede e de transporte. Tal mecanismo funciona da seguinte forma:

- Para cada pacote que entra na rede, a flag “Evil Bit” (unused) do cabeçalho IPv4 é marcada como 1 (normalmente ela é 0) e após o

cabeçalho IP é inserido um cabeçalho intPai e um cabeçalho intFilho com as informações do roteador corrente.

- Se a flag “Evil Bit” já for 1, ou seja, se já existem cabeçalhos int nesse pacote, o cabeçalho intPai e o(s) cabeçalho(s) intFilho presentes no pacote são parseados. Finalmente um novo cabeçalho intFilho é adicionado no **início** da lista de intFilhos e o campo de total de filhos do cabeçalho intPai é atualizado.

O cabeçalho intPai contém dois campos de 32 bits cada: um deles relativo ao tamanho de cada filho e outro relativo à quantidade total de filhos. Já o cabeçalho intFilho possui 4 campos: o campo relativo à identificação do roteador (32 bits), *timestamp* (48 bits), porta de entrada (9 bits) e porta de saída (9 bits). Como a soma dos tamanhos dos 4 campos não é múltiplo de 8, foi necessário também adicionar um “campo” de padding de 6 bits.

Importante salientar que a struct headers responsável por conter os pacotes parseados e que posteriormente serão emitidos pelo roteador tem um vetor de cabeçalhos intFilho de tamanho 20, portanto o número máximo de hops onde o *In-Network Telemetry* ainda funciona é 20.

Note que os cabeçalhos int adicionados a cada roteador não são eliminados uma vez que o pacote deixa a rede (e a flag “Evil Bit” também não é retornada a zero). Isto implica que a aplicação de recebimento Python saiba lidar com os cabeçalhos int adicionados ao pacote.

A aplicação de transmissão Python foi modificada levemente para enviar uma mensagem padrão a um endereço de IP fixo no caso em que foi chamada se parâmetros adicionais na linha de comando, para facilitar os testes. Quando definido, ela envia a mensagem informada como parâmetro ao IP informado como parâmetro através do protocolo TCP na porta 1234.

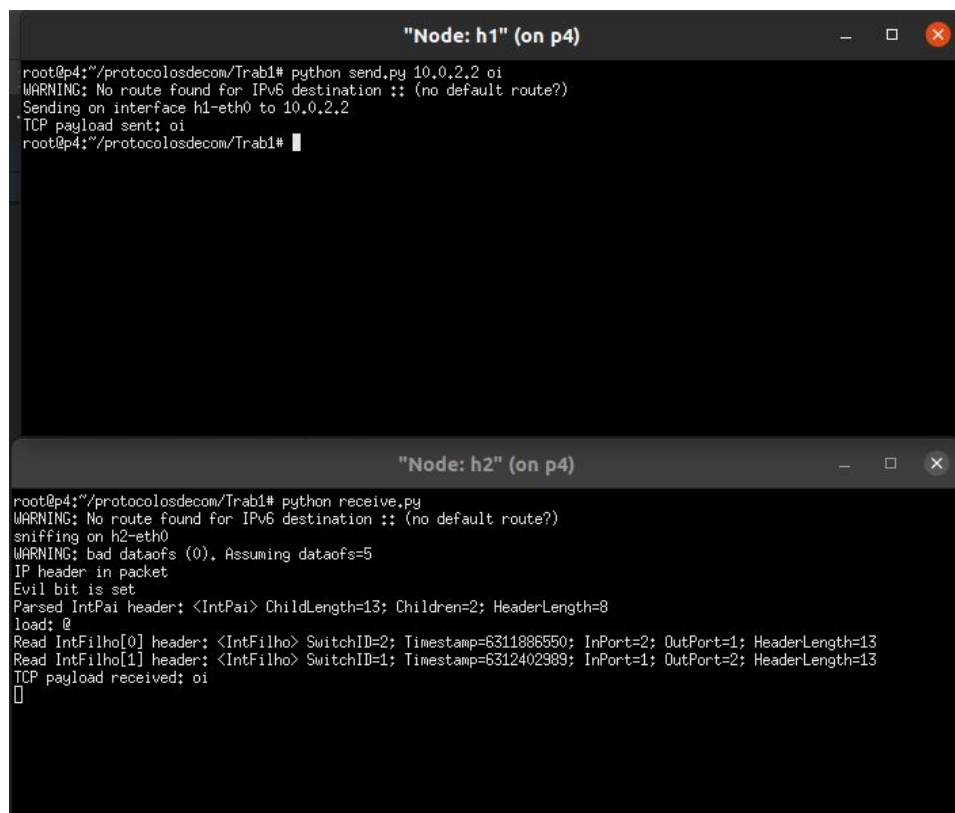
Já a aplicação de recebimento Python foi profundamente modificada. A função `handle_pkt`, que inicialmente apenas recebia o pacote e exibia o conteúdo do segmento TCP, testa se a flag “Evil Bit” está setada como 1 e caso positivo o conteúdo do pacote IP é lido como um pacote intPai (e suas informações são impressas no terminal) e o conteúdo deste por sua vez é lido como uma sequência finita de intFilhos (cujas informações são impressas no terminal). O número de intFilhos a serem lidos é informado pelo respectivo campo do cabeçalho intPai. Finalmente o cabeçalho TCP é “pulado” e o conteúdo do segmento TCP é mostrado no terminal.

2. Experimentos

Para validar a implementação, conduzimos testes com todos os possíveis pares *sender-receiver* verificando os seguintes itens:

- **Rota correta de roteamento**, o que pode ser observado pelos IDs dos switches percorridos e número de cabeçalhos INT filho anexados ao pacote.
- **Portas de entrada e saída**, devem estar de acordo com os parâmetros dos arquivos *s<id>-runtime.json*.
- **Timestamps**, que é um inteiro representando o tempo em microssegundos no momento em que o pacote chega ao *Ingress* e inicializado em 0 quando os switches são criados. Portanto podemos verificar se a parte mais significativa dos valores obtidos são iguais, dada a baixa latência do caso de teste utilizado, e se a ordem dos valores é equivalente à topologia da rede.
- **Integridade dos dados dos cabeçalhos e payload**, que é validado no momento em que conseguimos receber e parsear os dados no destinatário de forma dinâmica, usando parâmetros como o número e o tamanho de filhos determinado no cabeçalho INT pai, e mantendo o conteúdo original do pacote, nesse caso um cabeçalho TCP carregando uma string conhecida no payload.

Abaixo, um *snapshot* da execução do envio a partir do host H1 tendo como destinatário o host H2 como exemplo. O destinatário exibe as informações do cabeçalho INT pai (tamanho do cabeçalho filho e quantidade de filhos) e, em seguida, a pilha de cabeçalhos INT filho contendo suas respectivas informações (SwitchID, Timestamp, InPort, OutPort) bem com o tamanho do cabeçalho (HeaderLength), fixo em 13 bytes, seguido do payload do TCP.



```
"Node: h1" (on p4)
root@p4:~/protocolosdecom/Trab1# python send.py 10.0.2.2 oi
WARNING: No route found for IPv6 destination :: (no default route?)
Sending on interface h1-eth0 to 10.0.2.2
TCP payload sent: oi
root@p4:~/protocolosdecom/Trab1#

"Node: h2" (on p4)
root@p4:~/protocolosdecom/Trab1# python receive.py
WARNING: No route found for IPv6 destination :: (no default route?)
sniffing on h2-eth0
WARNING: bad dataofs (0). Assuming dataofs=5
IP header in packet
Evil bit is set
Parsed IntPai header: <IntPai> ChildLength=13; Children=2; HeaderLength=8
load: @
Read IntFilho[0] header: <IntFilho> SwitchID=2; Timestamp=6311886550; InPort=2; OutPort=1; HeaderLength=13
Read IntFilho[1] header: <IntFilho> SwitchID=1; Timestamp=6312402989; InPort=1; OutPort=2; HeaderLength=13
TCP payload received: oi
[]
```

Os *snapshots* das execuções de todos os pares se encontram na pasta *Trab1/test_evidence*, nomeadas de acordo com o padrão *<sender>-<receiver>.png*

3. Problemas encontrados durante a implementação

- Como sinalizar a presença dos cabeçalhos INT.

Pensamos em algumas soluções para isto, até mesmo em um campo chamado “next_protocol” no intPai. Colocamos a dúvida no Microsoft Teams que foi respondida pelo monitor da disciplina, Lucas. Resolvemos utilizar a solução apontada pelo monitor, ou seja, utilizar uma flag não utilizada no IPv4 popularmente conhecida como “Evil Bit”. O IPv4 possui 3 flags e o “Evil Bit” é a mais significativa delas, portanto se o valor do campo flag for maior ou igual a 4, o “Evil Bit” está setado e o pacote contém um cabeçalho intPai.

- Usar um vetor de cabeçalhos (exemplo do source routing na pasta da aula prática)

Discutimos um pouco sobre se era certo utilizar um vetor de cabeçalhos na struct headers. Tínhamos medo que a fase de emit incluísse os cabeçalhos não setados na struct, mas os outros exemplos da pasta oferecida pelo professor resolveram nossas dúvidas.

- Obtenção do timestamp (exemplo no git do p4lang)

Demorou um pouco, mas pesquisando no google descobrimos como obter o timestamp e o que ele significa no contexto de switches P4.

- ID único do switch (Lucas e arquivos s<id>-runtime.json de cada switch)

Lucas nos deu duas sugestões a respeito deste problema: utilizar fontes P4 distintos para cada switch (que implica em várias modificações em coisas que não entendemos) ou alterar os dados do json de cada switch (que fizemos). Adicionamos um campo “SwitchID” em cada entrada da tabela LPM nos arquivos s<id>-runtime.json.

- Deparse do cabeçalho no receive.

Insistimos bastante em utilizar o Scapy conforme sua documentação, principalmente seguindo o que explicam sobre novos protocolos. Após não funcionar desta forma, utilizamos classes Python para definir o formato dos cabeçalhos IntPai e IntFilho (int_headers.py).

Todos os campos devem ser convertidos para hexadecimal e posteriormente para inteiro. Como os campos sobre as portas de entrada e saída são de 9 bits, tivemos que implementar uma função de conversão de hexadecimal para int customizada para este uso (hexToInt da biblioteca utils.py).

4. Passos da Implementação dos Requisitos Funcionais

Primeiramente criamos os cabeçalhos intPai e intFilho no basic.p4 com seus respectivos campos e os adicionamos à struct headers (o intFilho é um vetor de 20 posições nessa struct).

Após isto, adicionamos dois estados no parser: um para o intPai e outro para o intFilho. O estado parse_intPai extrai um cabeçalho de intPai, salva a quantidade de filhos em uma variável meta.nRemaining e chama por default o estado parse_intFilho. Meta é uma struct global de dados úteis para o processamento do pacote. Esta variável então serve de contador no estado parse_intFilho, o qual diminui o contador em uma unidade, extrai o cabeçalho intFilho e chama ele mesmo por default, a menos que o contador esteja em zero. Além disso modificamos o estado parse_ipv4 para chamar o estado parse_intPai caso a flag “Evil Bit” esteja setada.

No ingress processing, criamos duas novas ações: new_intPai e new_intFilho. Na ação new_intPai, o tamanho do filho é fixo em 13 e a quantidade de filhos é 0. Na ação new_intFilho, incrementa-se em uma unidade o campo relativo à quantidade de filhos no intPai. Por enquanto na ação relativa à criação do intFilho deixamos todos os campos receberem dados hardcoded, para primeiramente testar o que já foi implementado no parser e para testar o que será implementado no receive.py. Na seção apply, implementamos o seguinte: se o “Evil Bit” estiver ativado e o intPai for válido, cria-se um novo intFilho, testa-se se o ipv4 é válido e aplica-se o encaminhamento através do método LPM (longest prefix match). Caso intPai não seja válido, o pacote é descartado e caso o “Evil Bit” não esteja ativo, ativa-se a flag “Evil Bit”, cria-se um intPai e um intFilho e aplica-se o encaminhamento LPM.

Egress e Checksum não foram alterados e finalmente colocamos intPai e intFilho na emissão do deparser.

Prosseguimos para a edição do `receive.py` e criação do `int_headers.py` e `utils.py` como mencionado na seção anterior deste documento. Esta aplicação TCP recebe um pacote IPv4, testa se o “Evil Bit” está ativo informando no terminal, lê um cabeçalho `intPai` do payload do IPv4, e lê `n` `intFilhos` onde `n` é o número informado no campo quantidade de filhos do `intPai`. Para cada `intFilho` parseado, a informação dos campos é exibida no terminal. Finalmente o conteúdo do payload TCP é exibido no terminal.

Após testar o que já foi implementado, prosseguimos alterando os arquivos `s<id>-runtime.json` onde `<id>` é um dos switches (no caso da topologia de rede abordada: 1, 2 ou 3) para adicionar o campo “SwitchID” em cada entrada da tabela LPM. Este campo assume valores distintos conforme o switch. Após isto, retiramos a atribuição hardcoded dos campos `intFilho` no `ingress processing` e atribuímos os valores corretos (porta de entrada, porta de saída, timestamp e `IdSwitch`).