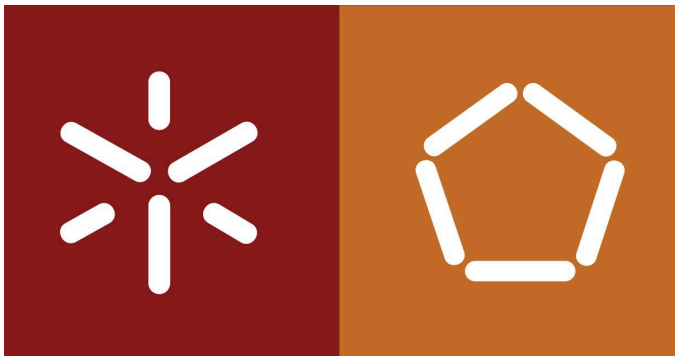

Data Plane Programming

Grupo nº11,

Alexandre Eduardo Vieira Martins PG53604

Índice

Introdução

As **Redes Definidas por Software (SDN – *Software-Defined Networking*)** introduzem uma arquitetura inovadora que separa logicamente o **plano de controlo** do **plano de dados**, permitindo uma gestão centralizada e programável da rede através de software. Esta separação funcional é fundamental para uma maior flexibilidade e automação. O **plano de dados** (data plane) é responsável pela execução das ações sobre os pacotes — como encaminhar, descartar ou modificar — enquanto o **plano de controlo** (control plane) define as políticas e toma decisões sobre como esses pacotes devem ser tratados e encaminhados ao longo da rede.

Para permitir maior flexibilidade na definição do comportamento da rede, surgiu a linguagem **P4**. O P4 permite programar o comportamento do plano de dados de dispositivos de rede, como switches e routers, de forma independente dos protocolos subjacentes. Assim, é possível adaptar a rede a novas necessidades sem depender das limitações do hardware ou firmware dos fabricantes.

Neste contexto, os Label Switching Protocols desempenham um papel importante, especialmente em redes de grande escala. Protocolos como o **MPLS (Multiprotocol Label Switching)** utilizam labels para encaminhar pacotes de forma eficiente, evitando a análise detalhada do cabeçalho IP em cada salto. Estes rótulos são atribuídos e geridos pelo plano de controlo, mas utilizados pelo plano de dados para realizar o encaminhamento de forma rápida e simplificada.

A topologia utilizada foi implementada em Mininet e inclui múltiplos túneis entre dois domínios de rede. A Tabela 1 apresentada em seguida, mostra as ligações, interfaces, endereços IP e MAC dos dispositivos envolvidos, sendo essencial para a configuração e testes realizados.

Host	Interface	IP	MAC
h1	h1-eth0	10.0.1.1/24	00:04:00:00:00:01
h2	h2-eth0	10.0.1.2/24	00:04:00:00:00:02
h3	h3-eth0	10.0.1.3/24	00:04:00:00:00:03
h4	h4-eth0	10.0.8.1/24	00:04:00:00:00:04

Link	Interface	MAC
s1-h1	s1-eth1	00\:aa\:bb:00:01:01
s1-h2	s1-eth2	00\:aa\:bb:00:01:02
s1-r1	s1-eth3	00\:aa\:bb:00:01:01
s1-h3	s1-eth4	00\:aa\:bb:00:01:03

Router	Port	Interface	MAC
r1	1	r1-eth1 (to s1)	aa:00:00:00:01:01
r1	2	r1-eth2 (to r2)	aa:00:00:00:01:02
r1	3	r1-eth3 (to r6)	aa:00:00:00:01:03
r2	1	r2-eth1 (to r1)	aa:00:00:00:02:01
r2	2	r2-eth2 (to r3)	aa:00:00:00:02:02
r3	1	r3-eth1 (to r2)	aa:00:00:00:03:01
r3	2	r3-eth2 (to r4)	aa:00:00:00:03:02
r4	1	r4-eth1 (to r3)	aa:00:00:00:04:01
r4	2	r4-eth2 (to h4)	aa:00:00:00:04:02
r4	3	r4-eth3 (to r5)	aa:00:00:00:04:03
r5	1	r5-eth1 (to r6)	aa:00:00:00:05:01
r5	2	r5-eth2 (to r4)	aa:00:00:00:05:02
r6	1	r6-eth1 (to r1)	aa:00:00:00:06:01
r6	2	r6-eth2 (to r5)	aa:00:00:00:06:02

Implementação

MSLP

Para a implementação do mslp decidimos implementar uma solução similar à sugerida no enunciado do trabalho.
Começamos então por definir o header da seguinte forma

```
header mslp_label_t {
    bit<15> label;
    bit<1>  s;
}
```

O header contém 2 campos, a label e o um bit a indicar se esta label trata-se do final da stack, a label ocupa 15 bits pois o p4 só permite a definição de headers com tamanho múltiplo de 8.

Router de entrada do túnel

R1 Ingress

Após isto adicionamos a stack de labels à struct de headers através de um array de tamanho 3. A razão que a stack é de 3 é porque esse é o número de labels que existem no túnel. Ou seja no Ingress do router r1 vamos ter

```
action setTunnel(bit<15> labelr4, bit<15> labelr3, bit<15> labelr2) {
        hdr.mslp_stack[0].setValid();

        hdr.mslp_stack[0].label = labelr2;
        hdr.mslp_stack[0].s = 0;

        hdr.mslp_stack[1].setValid();

        hdr.mslp_stack[1].label = labelr3;
        hdr.mslp_stack[1].s = 0;

        hdr.mslp_stack[2].setValid();

        hdr.mslp_stack[2].label = labelr4;
        hdr.mslp_stack[2].s = 1;  // Bottom of stack

        meta.needs_tunnel = 1;
    }
```

Uma linha do flow file de r1 seria do tipo

```
table_add mslpTunnel setTunnel 10.0.8.0/24 => 3020 2020 1020
```

Nesta linha indicamos as labels do túnel necessárias e o ip de destino, é com este que decidimos se vamos passar à criação do túnel ou não. Decidimos por fazer com que todo o tráfego orientado para h4 seja enviado pelo túnel, por isso a decisão de efetuar a criação do túnel ou fazer o encaminhamento normalmente é com base no destino.

R1 Egress e Deparser

No egress a única tarefa necessária é a mudança do protocolo para MSLP no caso do túnel tenha sido criado, fazemos isso a partir de

```
if (meta.needs_tunnel == 1) {
            hdr.ethernet.etherType = TYPE_MSLP;
        }
```

O campo meta.needs_tunnel é alterado na setTunnel no ingress.

A diferença do Deparser é só a emissão da stack MSLP

```
apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.mslp_stack);
        packet.emit(hdr.ipv4);
    }
```

Routers Intermédios

O papel destes routers trata-se de receber a stack, com 2 ou 3 elementos, processar o elemento no topo da stack, eliminá-lo e dar shift à stack.

Parser

O parsing é feito da seguinte forma

```
state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_MSLP: parse_mslp_label;
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }
```

Extrai o pacote ethernet e faz a chamada da parse_mslp_label, que vai extrair a stack MSLP e a parse_ipv4, que vai extrair o pacote ipv4.

A parse_mslp_label vai proceder à extração sequencial da stack MSLP. Para isso começa a ler a stack extraindo o topo da mesma, verifica se este é o último elemento da stack através do atributo s do header do MSLP, caso não seja continua este processo até chegar ao fim da stack.

```
state parse_mslp_label {
        transition parse_mslp_0;
    }
```

```
state parse_mslp_0 {
        packet.extract(hdr.mslp_stack[0]);
        transition select(hdr.mslp_stack[0].s) {
            1: parse_ipv4;
            0: parse_mslp_1;
        }
    }

    state parse_mslp_1 {
        packet.extract(hdr.mslp_stack[1]);
        transition select(hdr.mslp_stack[1].s) {
            1: parse_ipv4;
            0: parse_mslp_2;
        }
    }

    state parse_mslp_2 {
        packet.extract(hdr.mslp_stack[2]);
        transition parse_ipv4;
    }
```

Ingress

O Ingress neste routers é dividido em dois tipos baseado na posição do nodo no túnel

```
table mslpTunnel {
        key = {
            hdr.mslp_stack[0].label: exact;
        }
        actions = {
            popFwdLast;
            popFwdShift;
            drop;
        }
        size = 256;
        default_action = drop;
    }
```

Existindo só dois nodos intermédios, o primeiro deles irá ter que fazer o shift da stack toda, enquanto que o outro só precisa de dar shift a um elemento e eliminar o outro da stack.

Isso é feito a partir das seguintes actions

```
action popFwdLast(bit<9> port, macAddr_t nextHop) {
        hdr.mslp_stack[0].label = hdr.mslp_stack[1].label;
        hdr.mslp_stack[0].s = hdr.mslp_stack[1].s;
        hdr.mslp_stack[1].setInvalid();

        standard_metadata.egress_spec = port;
        meta.nextHopMac = nextHop;
    }

    action popFwdShift(bit<9> port, macAddr_t nextHop) {
        hdr.mslp_stack[0].label = hdr.mslp_stack[1].label;
        hdr.mslp_stack[0].s = hdr.mslp_stack[1].s;

        hdr.mslp_stack[1].label = hdr.mslp_stack[2].label;
        hdr.mslp_stack[1].s = hdr.mslp_stack[2].s;

        hdr.mslp_stack[2].setInvalid();

        standard_metadata.egress_spec = port;
        meta.nextHopMac = nextHop;
    }
```

Router Final

O router final tem o papel do desencapsulamento do protocolo MSLP. O Parsing desde irá ser igual ao do router intermédios, após isso, efetua logo o desencapsulamento e procede para fazer as tarefas restantes, neste caso este nodo é também uma firewall.

No Ingress é verificado se o pacote veio pelo túnel através da label, caso isto seja verdade um atributo nos metadados é alterado para que no Egress seja efetuado o desencapsulamento.

Egress

```
if (meta.needs_decap == 1) {
            hdr.mslp_stack[0].setInvalid();

            hdr.ethernet.etherType = TYPE_IPV4;
        }
```

Aqui a única tarefa efetuada é o desencapsulamento, ou seja, indica-mos que a stack é inválida e mudamos o protocolo para IPV4.

Firewall

Para a implementação da firewall, decidimos utilizar uma abordagem semelhante à usada no exercício da firewall no repositório *p4lang/tutorials*. Ou seja, optámos por utilizar um **Bloom filter**. Para tal, começámos por criar duas funções de *hash* para auxiliar na verificação de se um pacote externo pertence a uma ligação previamente estabelecida. A firewall permite também que qualquer pacote proveniente da rede interna saia livremente, registando a ligação no **Bloom filter**.

Implementação da Firewall

1. Parser para ligações TCP e UDP

Começámos por definir o parser de forma a suportar ligações tanto **TCP** como **UDP**. O parser inicia com a extração do cabeçalho Ethernet e, conforme o campo *etherType*, decide se deve continuar para o processamento de pacotes **IPv4**. Uma vez identificado um pacote **IPv4**, verifica-se o campo *protocol* para decidir entre **TCP**, **UDP** ou outra opção não tratada. Para os protocolos **TCP** e **UDP**, os respetivos cabeçalhos são extraídos:

```
    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }
    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol) {
            TYPE_TCP: parse_tcp;
            TYPE_UDP: parse_udp;
            default: accept;
        }
    }
    state parse_tcp{
        packet.extract(hdr.tcp);
        transition accept;
    }
    state parse_udp{
        packet.extract(hdr.udp);
        transition accept;
    }
```

Figura 1 - Firewall Parser

2. *Deparser* com suporte aos protocolos pretendidos

O *deparser* foi implementado de forma a reemitir os cabeçalhos dos protocolos que pretendemos tratar. Independentemente de o pacote ser **TCP** ou **UDP**, os cabeçalhos correspondentes são emitidos, o que garante a integridade da estrutura dos pacotes processados:

```
control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.tcp);
        packet.emit(hdr.udp);
    }
}
```

Figura 2 - Firewall Deparser

3. Cálculo de funções de dispersão (hashes)

Para implementar o Bloom filter, criámos uma ação **compute_hashes** que gera duas posições de registo com base nas informações do fluxo (endereços IP, portas e protocolo). Utilizámos os algoritmos **crc16** e **crc32** para garantir uma distribuição eficiente no filtro de Bloom:

```
compute_hashes(ip4Addr_t ipAddr1, ip4Addr_t ipAddr2, bit<16> port1, bit<16> port2, bit<8> protocol)
{
       hash(reg_pos_one, HashAlgorithm.crc16, (bit<32>)0, {ipAddr1,
                                                           ipAddr2,
                                                           port1,
                                                           port2,
                                                           protocol},
                                                           (bit<32>)BLOOM_FILTER_ENTRIES);

       hash(reg_pos_two, HashAlgorithm.crc32, (bit<32>)0, {ipAddr1,
                                                           ipAddr2,
                                                           port1,
                                                           port2,
                                                           protocol},
                                                           (bit<32>)BLOOM_FILTER_ENTRIES);
}
```

Figura 3 - Firewall *compute_hashes*

4. Lógica de processamento no ingress

A função *apply* no ingress pipeline é responsável por aplicar a lógica da firewall. Esta começa por validar se o cabeçalho **IPv4** está presente e, em seguida, aplica a tabela *ipv4Lpm* para garantir que o destino está na rede. Se o pacote for **TCP** ou **UDP**, então a lógica divide-se conforme a direção do tráfego.

4.1 Pacotes provenientes do interior

Se o pacote entrar pela porta associada à rede interna (neste caso, porta 2), calcula-se o *hash* da ligação com os dados do pacote (endereços e portas de origem e destino), e os dois bits correspondentes são escritos nos filtros de Bloom. Isto regista a existência da ligação:

```
if(standard_metadata.ingress_port == 2) {
    if(hdr.ipv4.protocol == TYPE_TCP) {
        compute_hashes(hdr.ipv4.srcAddr, hdr.ipv4.dstAddr,
hdr.tcp.srcPort, hdr.tcp.dstPort, hdr.ipv4.protocol);
    } else if (hdr.ipv4.protocol == TYPE_UDP) {
        compute_hashes(hdr.ipv4.srcAddr, hdr.ipv4.dstAddr,
hdr.udp.srcPort, hdr.udp.dstPort, hdr.ipv4.protocol);
    }
    bloom_filter_1.write(reg_pos_one, 1);
    bloom_filter_2.write(reg_pos_two, 1);
}
```

Figura 4 - Tratamento para pacotes quem vem de dentro da rede na firewall

4.2 Pacotes provenientes do exterior

Se o pacote entrar por outra porta (isto é, da rede externa), o sistema calcula os hashes invertendo origem e destino (pois trata-se da resposta ao tráfego previamente registado). Em seguida, lê os dois **Bloom filters** para verificar se ambos os bits estão definidos. Se não estiverem, o pacote é descartado:

```
else {
    if(hdr.ipv4.protocol == TYPE_TCP) {
        compute_hashes(hdr.ipv4.dstAddr, hdr.ipv4.srcAddr,
hdr.tcp.dstPort, hdr.tcp.srcPort, hdr.ipv4.protocol);
    } else if(hdr.ipv4.protocol == TYPE_UDP) {
        compute_hashes(hdr.ipv4.dstAddr, hdr.ipv4.srcAddr,
hdr.udp.dstPort, hdr.udp.srcPort, hdr.ipv4.protocol);
    }
    bloom_filter_1.read(reg_val_one, reg_pos_one);
    bloom_filter_2.read(reg_val_two, reg_pos_two);

    if(reg_val_one != 1 || reg_val_two != 1) {
        drop(); return;
    }
}
```

Figura 5 - Tratamento para pacotes quem vem de fora da rede na firewall

4.3 Verificação da tabela MAC interna

Após a verificação e aceitação do pacote, seja ele interno ou externo, aplica-se a tabela *internalMacLookup*, para determinar a porta de saída com base no endereço MAC de destino.

Testes e Resultados

O tráfego parece estar quase todo a funcionar, no entanto existe algum problema no encaminhamento do router r1, infelizmente não consegui descobrir o problema.
Apesar disto vou apresentar alguns exemplos no wireshark para provar que pelo menos funciona parcialmente, e indicar claramente que a causa está algures em r1.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	DigitalEquip_00:01:…	DigitalEquip_00:02:…	0x88b5	104	Local Experimental Ethertype 1
2	0.003841362	10.0.8.1	10.0.1.1	ICMP	98	Echo (ping) reply id=0xeea8, seq=1/256, ttl=61
3	1.020917645	DigitalEquip_00:01:…	DigitalEquip_00:02:…	0x88b5	104	Local Experimental Ethertype 1
4	1.024775960	10.0.8.1	10.0.1.1	ICMP	98	Echo (ping) reply id=0xeea8, seq=2/512, ttl=61
5	2.038620528	DigitalEquip_00:01:…	DigitalEquip_00:02:…	0x88b5	104	Local Experimental Ethertype 1
6	2.042746969	10.0.8.1	10.0.1.1	ICMP	98	Echo (ping) reply id=0xeea8, seq=3/768, ttl=61
7	3.062012173	DigitalEquip_00:01:…	DigitalEquip_00:02:…	0x88b5	104	Local Experimental Ethertype 1
8	3.065662524	10.0.8.1	10.0.1.1	ICMP	98	Echo (ping) reply id=0xeea8, seq=4/1024, ttl=61
9	4.090203849	DigitalEquip_00:01:…	DigitalEquip_00:02:…	0x88b5	104	Local Experimental Ethertype 1
10	4.095885173	10.0.8.1	10.0.1.1	ICMP	98	Echo (ping) reply id=0xeea8, seq=5/1280, ttl=61

Figura 6 - Wireshark do router 1 interface 2

Como se pode ver na figura 6 o router 1 envia corretamente o ping pelo túnel, isto pode se verificar a partir do protocolo, que é o que eu indicamos da seguinte forma,

```
const bit<16> TYPE_MSLP = 0x88B5;
```

e vemos também os pacotes com protocolo ICMP, que representam a resposta ao ping. Ou seja, o tráfego até este ponto encontra-se totalmente funcional.
A partir deste nodo no entanto a resposta não é encaminhada por r1 como podemos ver na figura 7.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.1.1	10.0.8.1	ICMP	98	Echo (ping) request id=0xeea8, seq=1/256, ttl=64 (no response found!)
2	1.020035661	10.0.1.1	10.0.8.1	ICMP	98	Echo (ping) request id=0xeea8, seq=2/512, ttl=64 (no response found!)
3	2.038411973	10.0.1.1	10.0.8.1	ICMP	98	Echo (ping) request id=0xeea8, seq=3/768, ttl=64 (no response found!)
4	3.061445220	10.0.1.1	10.0.8.1	ICMP	98	Echo (ping) request id=0xeea8, seq=4/1024, ttl=64 (no response found!)
5	4.089991085	10.0.1.1	10.0.8.1	ICMP	98	Echo (ping) request id=0xeea8, seq=5/1280, ttl=64 (no response found!)

Figura 7 - Wireshark do switch 1 interface 3

Conclusão

Apesar de não ter conseguido os resultados esperados considero que os meus conhecimentos de p4 evoluíram muito graças a este trabalho, e considero que com melhor planeamento, conseguiria realizar um trabalho muito superior numa futura nova tentativa.

Bibliografia

https://github.com/p4lang/tutorials