Manipulação de pacotes com P4

Trabalho Prático 2 - Redes Definidas por Software

15 de Junho de 2024

Guilherme Sampaio	**Miguel Gomes**	**Rodrigo Pereira**
PG53851	PG54153	PG54198

1. Introdução

Este relatório dá continuidade ao projeto desenvolvido na Etapa 1 — Firewall, e propõe a implementação de respostas `ICMP` nos *routers* `P4` utilizados na topologia. Além disso, é também pedido o desenvolvimento de um balanceador de carga, assim como a implementação de `NAT` (*Network Address Translation*). Deste modo, no final deste projeto teremos uma rede funcional com *firewall* L3/L4 com noção de estado, respostas ICMP, *load-balancing* e `NAT`.

No relatório, a Seção 2 descreve as alterações feitas ao cenário comparativamente ao anterior, a Secção X, detalha a implementação e análise experimental do requisito relativo a respostas `ICMP` e, do mesmo modo, a Seção Y trata do *load-balancing* e `NAT`. Finalmente, a Secção Z descreve como configurar, compilar e executar o projeto, seguido da Secção O com a conclusão e trabalho futuro.

2. Cenário de Rede & Topologia

O cenário utilizado na Etapa 1 é fixo, não sendo alterado de modo algum nesta etapa do trabalho. Do mesmo modo, a topologia também não sofreu qualquer alteração, uma vez que é totalmente dependente do cenário.

Relativamente às regras utilizadas na *firewall*, estas também se mantém inalteradas.

3. Implementação de Respostas `ICMP`

O *Internet Control Message Protocol* (`ICMP`) é um protocolo fundamental no funcionamento das redes de computadores, operando na camada de rede do modelo `OSI`. Este protocolo é utilizado primariamente para enviar mensagens de erro e informações operacionais, especialmente no diagnóstico e solução de problemas de conectividade.

No protocolo `IPv4`, `ICMP` é identificado pelo valor hexadecimal `0x01` no campo *Protocol* do cabeçalho `IPv4`. Um pacote `ICMP` é composto por um cabeçalho de 8 octetos, seguido de uma secção dedicada a dados de tamanho variável. Os primeiros quatro octetos do cabeçalho contém um tamanho e formato fixo, enquanto que o formato dos restantes é dependente do tipo e código do pacote `ICMP`.

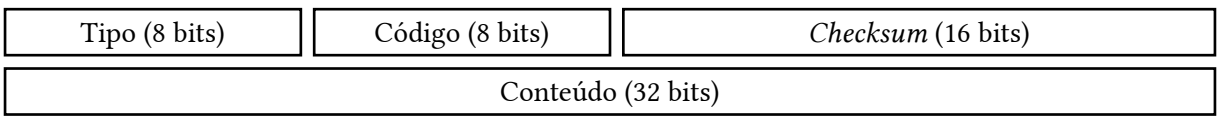

Figura 1: Cabeçalho de um pacote `ICMP`.

A Figura 1 ilustra a composição do cabeçalho de um pacote `ICMP`. O campo *Tipo* indica o tipo de mensagem de controlo, neste trabalho apenas utilizamos os tipos 0 — *Echo Reply* e 8 — *Echo Request*. Já o campo *Código* representa um valor específico ao tipo de mensagem de controlo, no caso 0 para ambos tipos.

Na receção de um pacote `ICMP`, podemos agir de dois modos. Caso o *Tipo*, no cabeçalho `ICMP`, seja igual a *Echo Request (8)* e o endereço de destino especificado no cabeçalho do pacote `IPv4` seja igual

ao endereço do nó recetor, então respondemos ao pedido com uma mensagem ICMP de tipo *Echo Reply (0)*. Caso contrário, damos continuidade ao encaminhamento do pacote até que chegue ao destino.

Este comportamento é facilmente atingido em P4 através do uso de uma tabela com correspondência exata e verificação do *Tipo*, na etapa de *Ingress* da *pipeline* do P4.

```
action reply_to_icmp(){
    ip4Addr_t dst = hdr.ipv4.dstAddr;
    hdr.ipv4.dstAddr = hdr.ipv4.srcAddr;
    hdr.ipv4.srcAddr = dst;
    hdr.icmp.type = 0;

    meta.AlreadyTranslated = 1;
}
```

```
table self_icmp {
  key = { hdr.ipv4.dstAddr : exact; }
  actions = {
      reply_to_icmp;
      NoAction;
  }
  default_action = NoAction;
}
```

Listing 1: Implementação de ICMP em P4.

No Listing 1, observamos o comportamento descrito acima. Na eventualidade de `hdr.ipv4.dstAddr` ser uma correspondência exata com o endereço do nó recetor, será aplicada a ação `reply_to_icmp` que altera o cabeçalho do pacote IPv4 de modo a responder à origem, e define o *Tipo* do cabeçalho ICMP como *Echo Reply (0)*.

Em termos de fluxo, existem algumas verificações extra antes de aplicar a tabela `self_icmp`.

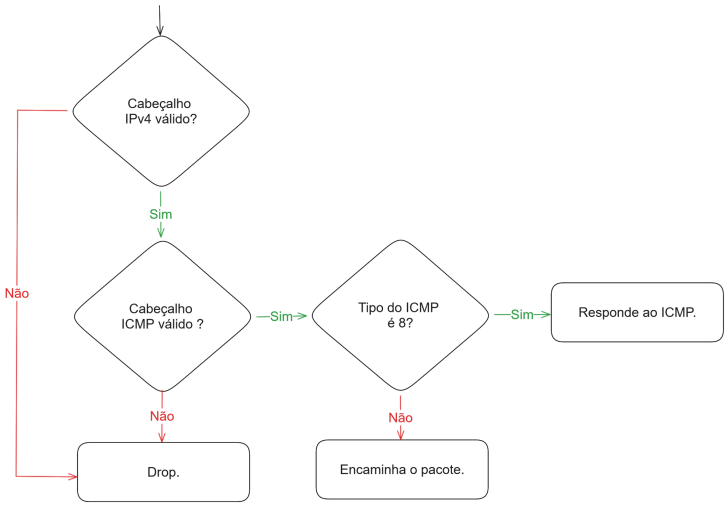

Figura 2: Mecanismos de resposta a pedidos ICMP.

A Figura 2 ilustra as verificações realizadas de modo a lidar com pacotes ICMP. Inicialmente, dá-se a validação dos cabeçalhos IPv4 e ICMP, que levam ao *drop* do pacote caso falhe, a seguir verifica-se o tipo de mensagem ICMP, tomando a ação correta consoante.

3.1. Análise Experimental

Mais uma vez, os testes relativos a funcionalidades adicionadas são todos executados durante a inicialização da aplicação. No caso, o objetivo é verificar o funcionamento da aplicação `ping` dos *hosts* de cada LAN para os respetivos *routers*. Apenas testamos os *pings* dentro de cada LAN devido às regras definidas nas *firewalls*. Além disso, queremos verificar o funcionamento de *pings* para *routers*, uma vez que os *hosts* por serem máquina *Linux* sabem lidar nativamente com pacotes ICMP.

Os testes são feitos através do comando `ping <IP> -c 5 | grep received`, é feita a procura pela palavra *received* para garantir o sucesso da execução.

Figura 3: Mecanismos de resposta a pedidos ICMP.

A Figura Figura 3 demonstra o resultado dos testes executados, garantindo o funcionamento correto da implementação de respostas a mensagens `ICMP`.

4. *Network Address Translation* (NAT)

O propósito de *Network Address Translation* é permitir que múltiplos dispositivos numa rede local privada compartilhem um único endereço `IP` público para aceder à internet. Esta técnica não só economiza endereços `IP`, como também melhora a segurança da rede ao mascarar os endereços `IP` internos dos dispositivos, dificultando a identificação e o acesso direto a esses dispositivos por parte de fontes externas. Além disso, o `NAT` facilita a gestão e modificação da rede interna sem a necessidade de alterar configurações externas, proporcionando flexibilidade e eficiência na administração da rede.

A implementação de `NAT` no projeto extende-se pelas fases de *ingress* e *egress* da *pipeline* de `P4`, sendo as explicações neste capítulo dividido nessa mesma forma.

4.1. Fase de *Egress*

Começar pela fase de *egress* pode parecer um pouco contraintuitivo, uma vez que decorre posteriormente à fase de *ingress*. Porém, há pontos necessários de interiorizar antes de passar ao *ingress*.

As alterações realizadas neste projeto têm como base todo o código desenvolvido na Etapa 1.

Fundamentalmente, a tradução de endereços públicos num endereço privado é feita com base em regras definidas no fichero de configuração, com a definição dos campos `port` e `localPort` nos *routers*.

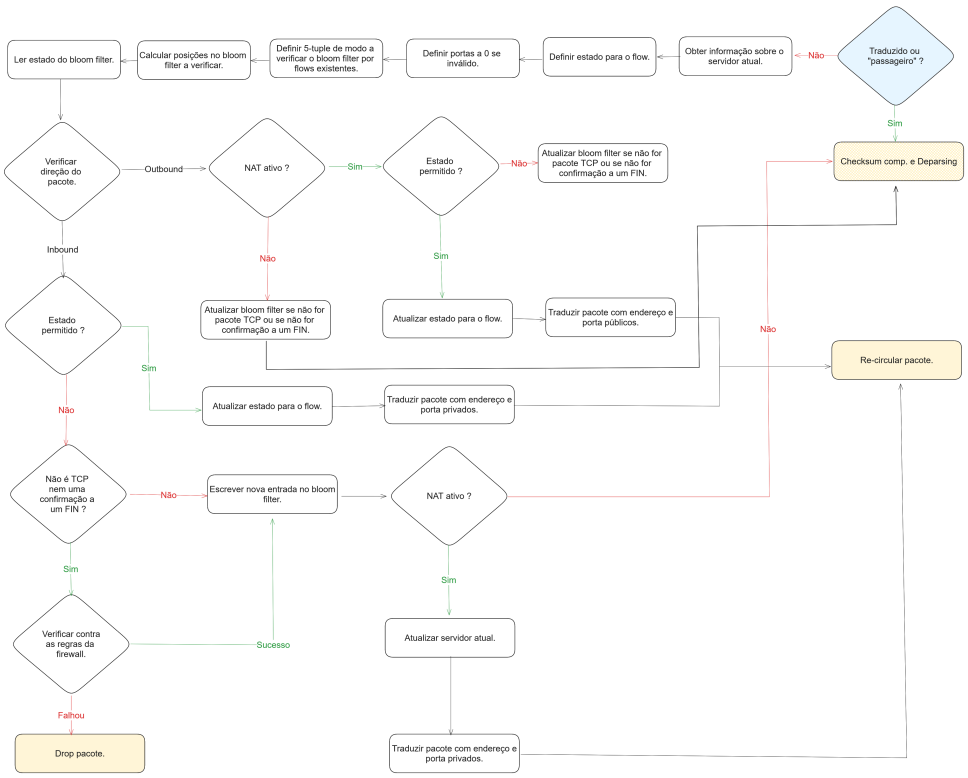

Figura 4: Fluxo na etapa de *egress*.

A Figura 4 ilustra o diagrama de fluxo com todos os passos que decorrem durante a fase de *egress* relativamente aos mecanismos de *network address translation* e *load-balancing*.

Em primeiro lugar é feita a verificação de se o pacote já foi previamente traduzido, ou é um pacote "*passageiro*". Um *router* define um pacote como sendo "*passageiro*" quando o pacote não se dirige, ou origina na rede que este controla, quando age como um *router* intermediário:

1. **Sim, já foi traduzido ou é "*passageiro*"**, então continua para as etapas de *checksum* e *deparsing*, uma vez que o NAT e o *load-balancer* já fizeram a sua "*magia*";
2. **Não**, então continuamos com o processamento do pacote.

De seguida, continuando do ponto 2, dá-se a definição do novo estado que este pacote definirá no router, é feita esta verificação uma vez que diferentes pacotes em diferentes contextos têm de ser tratados diferentemente. Existem três estados possíveis, representados pelos números entre 0 e 2:

- **0** — Representa um estado de *block*, indicando que a conexão levou um *reset*. Este é o estado escolhido quando a *flag* `RST` do cabeçalho `TCP` é válida ou quando se trata de uma resposta `ICMP`.
- **1** — Representa um estado de conexão *ongoing*. Este estado é escolhido quando as *flags* `RST` e `FIN` não estão presentes no cabeçalho `TCP`. Indica que uma conexão está ativa e a decorrer.
- **2** — Representa um estado acabado, indicando que a conexão foi terminada naturalmente. Este estado é escolhido quando a *flag* `FIN` está presente no cabeçalho `TCP`, e é válida.

Com o estado do pacote definido, vem a verificação da direção do pacote, que pode ser *inbound*, se estiver a entrar na rede local, ou *outbound*, se estiver a sair da rede local. E é neste passo que, pela primeira vez, o fluxo de processamento diverge.

1. **O pacote é *outbound* !**

Primeiro dá-se a verificação do estado do pacote, uma vez que apenas devemos atualizar os registos do *bloom filter* se o estado for 1 — *Ongoing*, ou se o pedido recebido não for um pacote `TCP` ou se não for uma confirmação(`ACK`) a um estado `FIN`. Caso contrário, os registos do *bloom filter* mantêm-se inalterados.

A seguir dá-se a tradução do endereço. Uma vez que se trata de um pacote que se dirige para fora da rede, temos de traduzir o endereço local (privado) num endereço público.

A tradução do endereço e portas é bastante simples uma vez que não é necessário Load Balancer para definir o endereço privado a usar na tradução(corresponde ao endereço originador privado na rede). Analogamente, o processo de escolha de porta privada e publica tambem é bastante simples visto que tem que se usar a porta de destino já definida no pacote para ambas. A necessidade de descobrir os endereços e portas a usar para este fluxo de pacotes deixa de existir caso já exista a entrada correspondente ao *five tuple* deste pacote no *bloom filter*. Caso essa entrada tenha estado 1 — *Ongoing*, usa-se os campos (endereços e portas definidos nessa entrada).

2. **O pacote é *inbound* !**

   De forma análoga à anterior, é realizada a verificação do estado do pacote, funcionando exatamente da mesma maneira caso o estado do pacote seja 1 — *Ongoing*. Porém, caso o pacote não seja `TCP` ou não seja uma confirmação a um `FIN` é feita uma filtragem pelas regras da *firewall*. Caso contrário (o pacote é `TCP` ou confirmação a um `FIN`), é criada uma entrada no *bloom filter*, o *host* escolhido pelo *load-balancer* é atualizado e o endereço público é traduzido em privado.

   Neste caso a tradução é feita com base no estado do pacote, porque se o pacote pertencer a uma sessão já existente e *ongoing*, registada no *bloom filter*, tem de ser encaminhado para o *host* responsável pela sessão. Caso contrário, é o *load-balancer* que decide o próximo *host* que vai receber o pacote.

Finalmente, um passo comum a ambos ramos, é feita a re-circulação do pacote, isto é, o pacote é enviado de novo para o início do *pipeline*. Desta vez, após o processamento na fase de *ingress*, vai poder saltar o processamento no *egress* (como descrito acima), podendo ser diretamente encaminhado.

4.2. Fase de *Ingress*

Tendo em conta a etapa de re-circulação aplicada no *egress*, torna-se mais fácil compreender o *ingress*.

A Figura Figura 5 ilustra o fluxo tomado por um pacote quando chega à etapa de *ingress*. Em primeiro lugar, verifica-se se o pacote já foi previamente traduzido (`meta.AlreadyTranslated = 1`), se o pacote já tiver sido traduzido, então está preparado, com os endereços corretos, para ser encaminhado. O pacote também será encaminhado diretamente se for considerado como "*passageiro*".

Caso contrário (o pacote é *inbound* ou *outbound*), é verificado o protocolo e, se for um pacote `ICMP`, o pacote é tratado como descrito na Seção 3, caso contrário segue para a etapa de *egress* para ser devidadmente processado (*load-balanced* e traduzido).

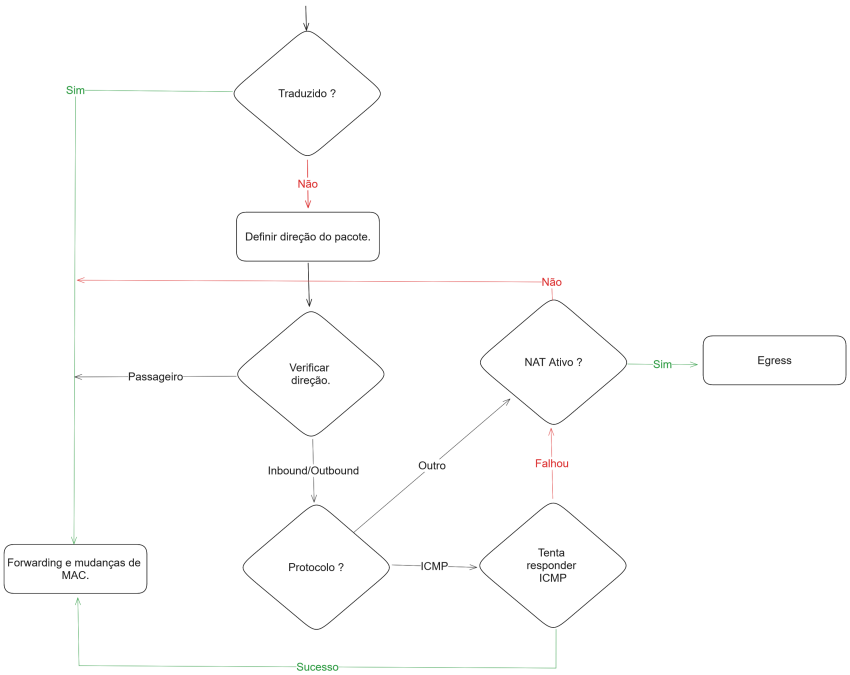

Figura 5: Fluxo na etapa de *ingress*.

5. *Load-Balancing*

O mecanismo de *load-balancing*, que decorre apenas na fase de *egress*, permite uma distribuição de carga pelos *hosts* de uma rede conforme os seus pesos (definidos também no ficheiro de configuração). O peso define quantos pacotes devem ser enviados para um *host*, antes de o trocar. Por exemplo, um *host* com peso de 6, irá receber 6 pacotes seguidos, a receção de um sétimo pacote será feita pelo *host* definido a seguir nas regras da tabela ServerLookup.

Em termos de código `P4` definimos uma variável com o identificador do servidor atual, pronto a receber os pacotes:

```
action                                              table ServerLookup {
setCurrentServer(ip4Addr_t privAddr, ip4Addr_t          key = { meta.cServerID : exact; }
publicAddr, bit<8> nextSID) {                           actions = {
    meta.cServer.privateAddr = privAddr;                    setCurrentServer;
    meta.cServer.publicAddr = publicAddr;                   NoAction;
    meta.cServer.nextServerID = nextSID;                }
}                                                    }
```

Listing 2: Mecanismo de *load-balancing* em `P4`.

Pelo Listing 2, a tabela `ServerLookup` é utilizada para mapear o `ID` do servidor (`meta.cServerID`) aos endereços `IP` público e privado e ao próximo `ID` de servidor. Utiliza a ação `setCurrentServer` que define os endereços do servidor e o próximo `ID` de servidor.

```
register<bit<8>>(1) currentServerID;
```

Listing 3: Servidor escolhido.

O registo `currentServerID` (Listing 3) armazena o `ID` do servidor atual, que é lido e armazenado em `meta.cServerID`. A tabela `ServerLookup` usa `meta.cServerID` como chave para encontrar e aplicar a ação `setCurrentServer`, que define os endereços `IP` público e privado do servidor (`meta.cServer.publicAddr` e `meta.cServer.privateAddr`) e o próximo `ID` de servidor (`meta.cServer.nextServerID`). Estes endereços são então utilizados na tradução de pacotes e na atualização dos *bloom filters*. Este mecanismo permite que o sistema selecione dinamicamente os endereços

IP corretos para a tradução de pacotes com base no `ID` do servidor atual, garantindo que os pacotes sejam encaminhados corretamente através da rede.

Isto apenas funciona, pois o comportamento do *router* é *round-robin*. De modo a garantir que os pesos de cada *host* são cumpridos, são necessárias tantas regras relativas ao *host* quanto o seu peso, o que pode não ser o método mais eficiente.

6. Conclusão e Trabalho Futuro

Dando por concluída esta etapa do trabalho, o grupo considera ter cumprido todos os objetivos propostos em ambos exercícios, implementação de respostas `ICMP` e implementação de `NAT` e *load-balancing*.

A implementação de respostas `ICMP` foi relativamente simples de se atingir, porém, a introdução de `NAT` e *load-balancing* levou a grandes tempos de *debug* e, pior ainda, imensas reestruturações de arquitetura. A definição da arquitetura para esta fase foi, sem dúvida, a tarefa que mais tempo consumiu.

O grupo considera as explicações apresentadas gerais e capazes de dar ao leitor uma noção base sobre o funcionamento do projeto. Porém, detalhes mais específicos relativos à implementação não foram incluídos de modo a melhorar a percetibilidade do relatório.

Relativamente a trabalho futuro, consideramos a introdução de testes automáticos relativos ao `NAT` e *load-balancing* um aspeto importante e necessário, de modo a igualar às fases anteriores em termos de testes.