Trabalho Prático 2 - Proxy TCP reverso com monitorização proactiva

António Silva, Gonçalo Pereira and José Silva

University of Minho, Department of Informatics, 4710-057 Braga, Portugal
e-mail: {a73827,a74413,a75280}@alunos.uminho.pt

1 Introdução

O presente relatório pretende explicar a estratégia adotada para a resolução do Trabalho Prático 2 para a disciplina de Comunicações por Computador. Este trabalho tem como objetivo implementar um servidor *proxy* reverso de tal forma a que responda a todos os pedidos feitos pelos clientes encaminhando-os para outro servidor de *backend* que faça o respetivo atendimento.

2 Arquitetura Implementada

A nossa arquitetura corresponde à descrita no enunciado deste trabalho: Dois programas, um para o sistema correspondente ao `Reverse Proxy` e outro para servidores `Backend`. Ambos os programas contêm uma camada de comunicação `UDP` e, no caso do `Proxy`, uma camada adicional de `TCP` de modo a poder receber os *requests* dos clientes, transmiti-los à aplicação `TCP` que resolve *requests* do lado do `Backend` e efetuar o correspondente percurso inverso.

3 Especificação do protocolo de monitorização

A monitorização dos servidores é feita através da camada UDP pelo que, do lado do *backend*, são guardados dados sobre o estado atual do servidor num pacote que é enviado e recebido do lado do *proxy*. O pacote é depois processado pela *thread* que monitoriza o servidor em questão, atualizando o estado do servidor guardado na tabela.

3.1 Primitivas de comunicação

A comunicação entre servidores (entre *proxy* e servidores de *backend*) é realizada enviando pacotes do tipo `UDP` pela porta **5555** contendo as mensagens protocolares.

3.2 Formato das mensagens protocolares (PDU)

São enviados dois tipos de pacotes pelo *backend*, que, neste caso, caracterizamos por dois tipos, sequencial (de registo) e *polling* (respetivamente 0 e 1):

- No caso do tipo 0, é enviado um pacote periódico consoante o número de sequência (começa em 0 até receber o primeiro pedido *poll* por parte do *proxy*), sendo que depois é incrementado unitariamente entre 1 e 100, no período de *ping* que é indicado inicialmente, reiniciando reinicia após chegar a 100 (volta a 1);
- No caso do tipo 1, o envio é apenas efetuado após a receção de um *poll* por parte do *proxy*, contendo o `CPU Load` ao invés do número de sequência.

O tipo dos pacotes especificados anteriormente é dado pelo seguinte:

```
Tipo de pacote 0 (registo periódico)
-Se SEQNUM=0
TIPO | SEQNUM | IP | BENCHMARK
-Se SEQNUM!=0
TIPO | SEQNUM | IP

Tipo de pacote 1 (depois de feito poll pelo proxy)
TIPO | CPULoad | IP
```

No caso do *proxy*, há apenas um tipo de pacote (de *poll*) que contém um único *byte*.

3.3 Interações

Inicialmente, é estabelecida a ligação entre *backend* e *proxy*, sendo que o *backend* começa uma contagem sequencial de números de sequência pelo número zero, incrementando mal a ligação é estabelecida. Após tal ligação, é constantemente enviado, pelo lado do *proxy*, um pedido de *polling*, cuja resposta servirá para atualizar o *load* do `CPU` da entrada na tabela do servidor em questão, juntamente com o `RTT` e, na eventualidade de não receber resposta dentro do tempo, incrementará o número de pacotes perdidos.

Já pelo lado do *backend* é enviado um número de sequência incrementado no período de tempo fornecido como parâmetro, juntamente com a resposta ao *poll*, caso o tenha recebido (em redes com perdas significativas, poderá nem sequer responder ao pedido de *polling*, levando, eventualmente, e no caso de várias perdas seguidas, ao *time out* e remoção do servidor em questão da tabela de melhor servidor).

4 Implementação

4.1 Backend Side

UDP Monitor

O programa do *backend* possui uma *main* que trata de do envio dos pacotes com o formato correto dependendo do contexto. A mesma é informada por outra *thread*(Watcher), através de um objeto partilhado, sobre *polls* vindos do *proxy*.

A *main* começa por fazer um *benchmark*/teste ao CPU para de seguida enviar continuamente os pacotes de registo e estabelecimento de ligação até recebida a confirmação (primeiro *poll*) por parte do *proxy*.

Após a confirmação, caso não sejam recebidos *polls*, a *main* envia continuamente os pacotes de registo, incrementando o número de sequência até 100 (depois faz *reset* para 1). Quando recebe um *poll*, é obtido o *CPU load* através de uma chamada ao sistema operativo e envia o pacote de tipo 1.

4.2 Proxy Side

UDP Monitor

– Estruturas de Dados

A tabela com os estados dos servidores foi implementada numa `ConcurrentSkipListSet`, pelo que os acessos à mesma podem ser feitos por várias *threads* ao mesmo tempo e é-nos possível utilizar um comparador "*custom*"para efetuar a ordenação.

O comparador ordena as entradas na tabela segundo a seguinte cadeia de comparações:

1. **Validade das entradas na tabela:** Se uma das entradas for inválida (o *backend* correspondente não respondeu ao último *poll*) e a outra for válida, o segundo servidor é escolhido.
2. ***Ratio* de pacotes perdidos/tempo decorrido desde início da ligação:** Pacotes perdidos são contados através do número de sequência dos pacotes de registo e, para ser uma comparação justa, são divididos pelo tempo decorrido. Assim, se um servidor s1 possui 5 pacotes perdidos e está há dois minutos ligado e um servidor s2 possui 2 pacotes perdidos mas está ligado há 10 segundos, o servidor s1 será o escolhido.
3. **CPU Load:** Se um servidor s1 possuir `CPU load` superior a 80% e outro servidor s2 possuir inferior a 80%, é automaticamente o s2 escolhido. A escolha do valor advém de termos reparado que a partir de 80% de *load*, a performance do `CPU` e responsividade geral do sistema diminui drasticamente.
4. **Relação velocidade/número de *requests*:** Se um servidor s1 possuir um resultado de *benchmark* de 60 e estiver a atender 3 *requests* e um s2 com resultado de *benchmark* de 30 a atender 0 *requests*, o escolhido será o s2, mesmo sendo mais lento.
5. **RTTs:** O servidor com menor `RTT` é escolhido.

A comparação seguinte na cadeia só é efetuada caso as anteriores resultarem num empate.

Os pacotes recebidos são guardados em `ArrayBlockingQueues` (uma para cada tipo de pacote) que por sua vez são mapeadas em `HashMaps`, pelo que obtemos a correspondência <IP do servidor *backend*, Fila de pacotes de tipo X>. As `ArrayBlockingQueues` são bastante úteis quando há uma *thread* que faz *input* de objetos e outras *threads* que ficam à espera que haja uma inserção.

– Funcionamento

A *main* começa por executar todas as *threads* auxiliares como a `Monitor`, que faz monitorização dos pacotes de tipo 0 e regista perdas de pacotes (lê da `ArrayBlockingQueue` dos pacotes de tipo 0 referida acima) e a `StatusManager`, que faz monitorização dos pacotes de tipo 1 (lê da `ArrayBlockingQueue` dos pacotes de tipo 1 referida acima), trata de enviar *polls* periodicamente e regista todos os parâmetros contidos no estado do servidor (por ex. `CPU load`, *timeouts*, etc).

Durante o resto do processo, a *main* encontra-se num ciclo onde recebe pacotes e guarda-os nas estruturas corretas, até ser detetado um input por parte do utilizador

TCP layer

A camada `TCP` é invocada pela *main* no início da sua execução e limita-se a esperar por conexões `TCP` na porta 80, criando *threads* TCPConnection que lêem o input vindo do cliente, escolhem o melhor servidor (primeira entrada válida na tabela) ou esperam que existam servidores válidos. De seguida, envia o input do cliente ao servidor *backend* escolhido e espera pela resposta. Quando a recebe, envia de volta ao cliente sem fazer modificações.

5 Testes e resultados

De modo a testar o funcionamento da rede em CORE, optamos pela seguinte topologia, realizada em ambiente CORE:

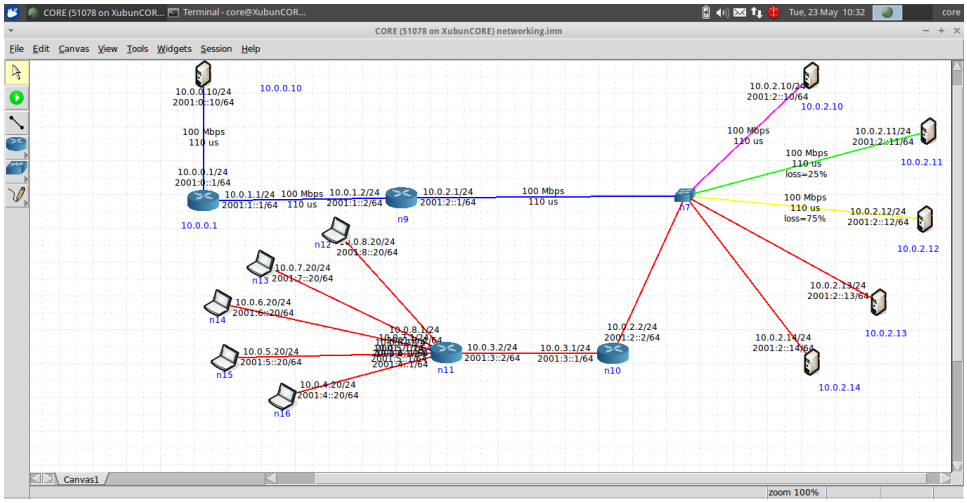

Figura 1. Topologia da Rede em CORE.

Após inicialização da rede, abrimos três janelas da bash: uma para um *proxy* e as restantes para *backends*. De seguida, inicializamos o *proxy* através da execução do jar.

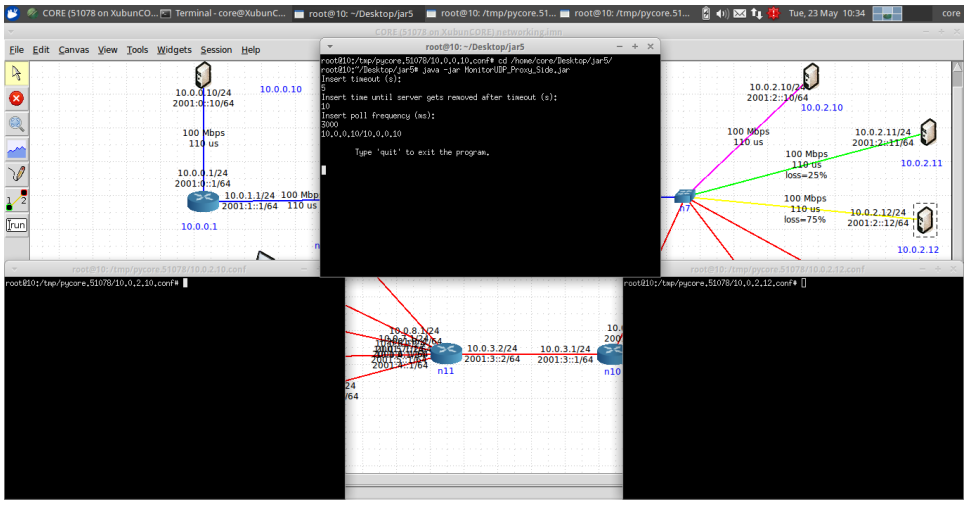

Figura 2. Inicialização do *proxy*.

Imediatamente após a execução do *proxy*, efetuamos a inicialização dos *backends*, executando, tal como no passo anterior, os jar em cada um deles. Inicialmente executamos apenas num deles, para demonstrar, de seguida, a atualização da tabela de melhor servidor.

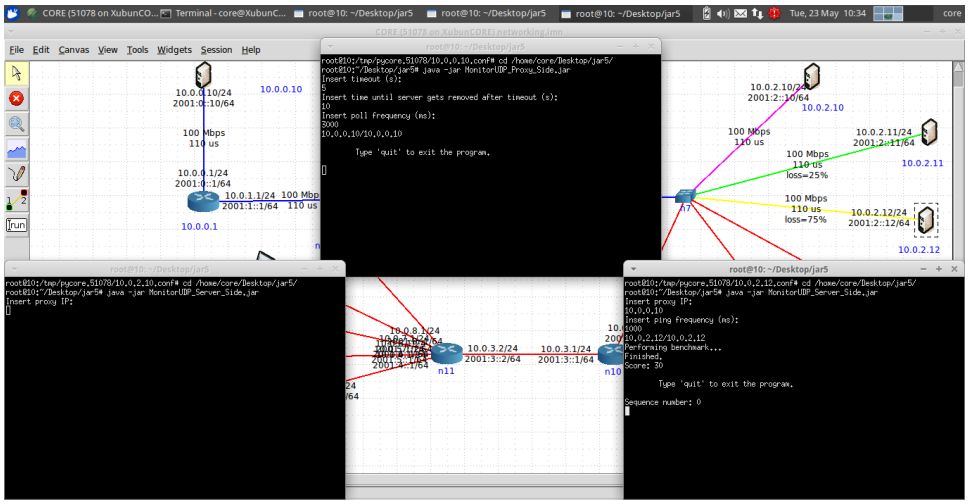

Figura 3. Inicialização do *backend*.

Estando já o processo a decorrer com normalidade, é feita uma atualização na tabela do *proxy*, indicando qual o melhor servidor a utilizar (sendo que, neste caso, como só um deles está a funcionar, apenas mostrará esse servidor).

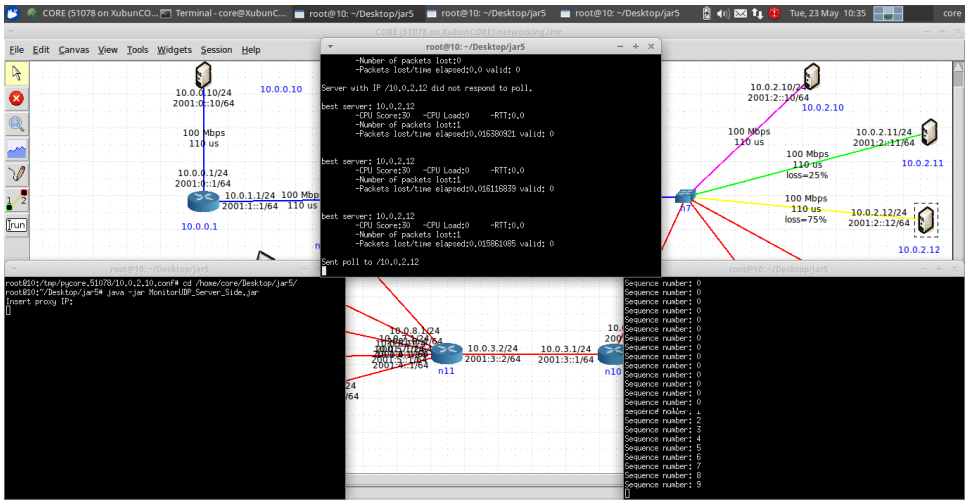

Figura 4. Determinação e indicação do melhor servidor.

De modo a elucidar a atualização do melhor servidor, e como o servidor anterior teria uma ligação mais fraca (isto é, o primeiro servidor a ser executado teria perdas de ligação bastante mais significativas, incitando a perda de pacotes), executamos um outro servidor numa rede sem quaisquer perdas, o que fez com que o melhor servidor fosse atualizado para este último adicionado:

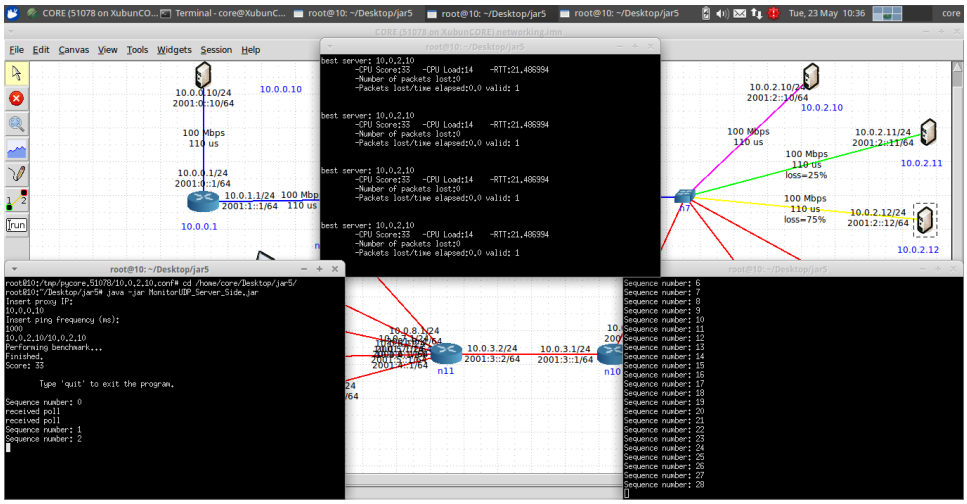

Figura 5. Atualização da tabela de melhor servidor.

Por fim, e de modo a demonstrar que estes estariam a funcionar de forma concorrente e a receber os respetivos pacotes sem que lhes houvesse intrusão/intermissão, testamos o envio de uma mensagem `telnet`, de um dos servidores para o *proxy*, resultando na seguinte situação, em que o *proxy* recebe, efetivamente, a mensagem:

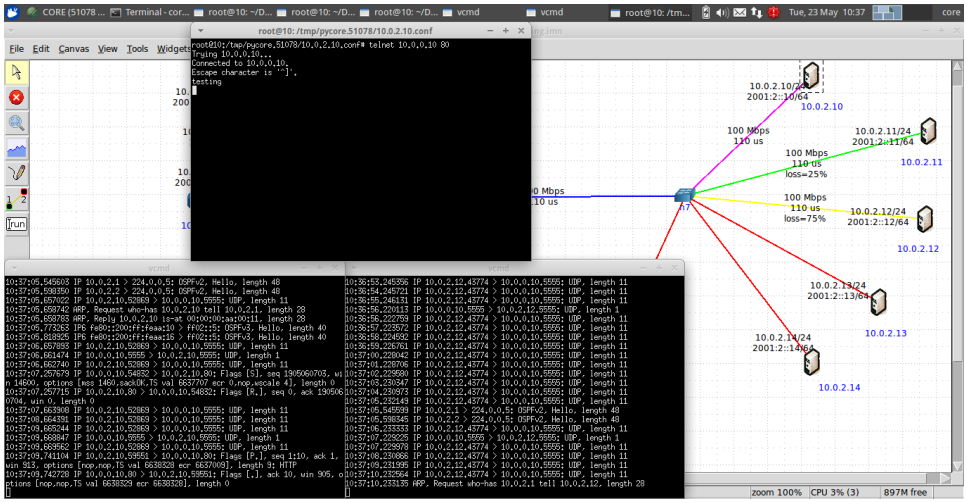

Figura 6. Exemplo de mensagem enviada por `TELNET`.

De seguida, e de modo a testar o servidor `mini-httpd`, fizemos com que tal servidor fosse executado num dos *backends*, do seguinte modo:

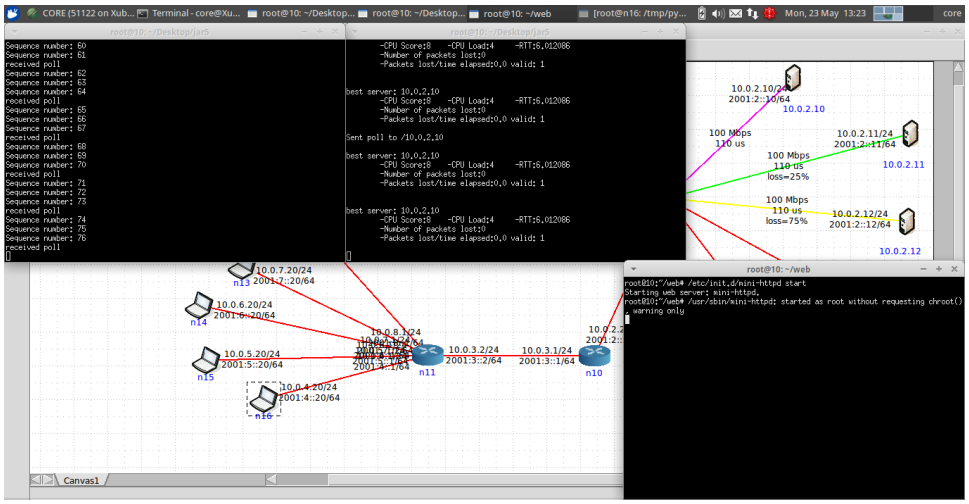

Figura 7. Inicialização do servidor `mini-httpd`.

Por fim, bastou-nos apenas, através de um dos clientes, tentar aceder ao servidor de `http`, enviando um pedido `WGET` para o proxy, e recebendo a página `index.html` guardada como default no servidor que estaria a executar o `mini-http`:

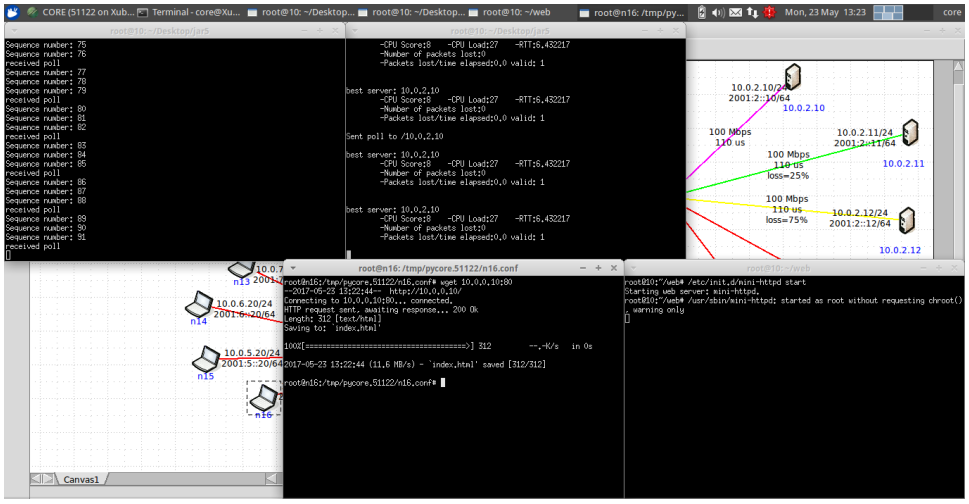

Figura 8. Pedido `WGET` para o *backend*.

6 Conclusões e trabalho futuro

Para um trabalho futuro uma opção seria termos forma de não fornecer parâmetros iniciais ao *proxy*, optando por uma arquitetura dinâmica (porém não 100% funcional para ser demonstrada numa rede de testes de topologia Core como verificada na aula, visto que seria mais difícil emular a perda de pacotes).

Concluímos também que deveríamos ter adaptado um critério de ordenação e comparação na tabela de *backends*, de modo a que fosse caracterizado por uma heurística. Deste modo, em vez de priorizar certas características, dar-se-ia um "peso"a cada uma das características (isto é, na situação atual, um *backend* com hardware bastante superior a outro, se estiver em situação de ocupação extrema, é "descartado"pelo mais inferior, que não está ocupado, visto que definimos a ocupação como sendo mais prioritária do que o `CPU Load`).