

Comunicações por Computador - Trabalho Prático N<sup>o</sup>2  
Proxy TCP reverso com monitorização proativa  
Mestrado Integrado em Engenharia Informática  
Universidade do Minho

Ana Isabel Castro  
(a55522)

Joana Miguel  
(a57127)

Lúcia Abreu  
(a71634)

28 de Maio de 2017

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Arquitetura da solução</b>	<b>2</b>
<b>3</b>	<b>Especificação do protocolo de monitorização</b>	<b>4</b>
3.1	Funcionamento geral . . . . .	4
3.2	Primitivas de comunicação . . . . .	4
3.3	Formato das mensagens protocolares . . . . .	5
3.4	Interações . . . . .	6
<b>4</b>	<b>Implementação</b>	<b>8</b>
4.1	Monitorização . . . . .	8
4.1.1	Servidor de back-end . . . . .	8
4.1.2	Servidor proxy . . . . .	9
4.2	Re-encaminhamento de tráfego . . . . .	12
4.2.1	Considerações adicionais . . . . .	14
4.3	Considerações adicionais globais . . . . .	14
<b>5</b>	<b>Testes e resultados</b>	<b>15</b>
5.1	Inicialização . . . . .	16
5.1.1	Back-end . . . . .	16
5.1.2	Reverse Proxy . . . . .	17
5.2	Teste 1 . . . . .	17
5.3	Teste 2 . . . . .	18
<b>6</b>	<b>Conclusões e trabalho futuro</b>	<b>22</b>

# Lista de Figuras

2.1	Arquitetura geral da solução a implementar . . . . .	2
3.1	Pacote geral . . . . .	5
3.2	Pacote de registo . . . . .	6
3.3	Pacote de <i>probe request</i> . . . . .	6
3.4	Pacote de <i>probe response</i> . . . . .	6
3.5	Camada do protocolo de monitorização e suas interações . . . . .	7
4.1	Esquema geral da implementação do Monitor UDP e a troca de mensagens com o <i>Reverse Proxy</i> . . . . .	8
4.2	Mecanismo de monitorização do servidor de <i>proxy</i> reverso . . . . .	10
4.3	Troca de pacotes de monitorização entre o servidor de <i>proxy</i> reverso e um agente de monitorização . . . . .	11
4.4	Fórmula de classificação de um servidor de <i>back-end</i> de acordo com as suas métricas	13
4.5	Mecanismo de reencaminhamento do servidor de <i>proxy</i> reverso . . . . .	14
5.1	Topologia de rede dentro do CORE usada para os testes. . . . .	15
5.2	Inicialização do servidor HTTP e Monitor UDP no Servidor1. . . . .	16
5.3	Inicialização do servidor HTTP e Monitor UDP no Servidor2. . . . .	17
5.4	Inicialização do Reverse Proxy no Servidor2. . . . .	17
5.5	Leitura do <code>index.html</code> diretamente ao Servidor1. . . . .	18
5.6	Leitura do <code>index.html</code> indiretamente através do Reverse Proxy no Servidor3. . . . .	18
5.7	Leitura consecutiva do ficheiro <code>index.html</code> através do <i>Reverse Proxy</i> . . . . .	19
5.8	Conteúdo dos ficheiros <code>index.html</code> lidos pelo cliente 10 vezes. . . . .	19
5.9	Output do <i>MonitorUDP</i> no Servidor2 onde informa que alterou as estatísticas enviadas ao <i>Reverse Proxy</i> . . . . .	20
5.10	Output do <i>Reverse Proxy</i> onde mostra o <i>score</i> dado a cada back-end aquando a chegada de um pedido de um cliente. . . . .	20
5.11	Conteúdo dos ficheiros <code>index.html</code> lidos pelo cliente 10 vezes, após o Servidor2 reportar estatísticas com resultados inflacionados. . . . .	21

## Resumo

Em serviços *web* é muito comum utilizar uma abordagem de *Proxy TCP reverso*. Esta abordagem é usada quando existem muitos serviços e um único servidor não é suficiente para conseguir responder a todos os clientes que chegam. Nestas situações existe uma *pool* de  $N$  servidores *back-end* gerida pelo servidor *Proxy TCP reverso* que mantém informação recolhida dinamicamente de cada um destes servidores. Através desta informação o servidor redirecionada para um *back-end* disponível o pedido de um cliente.

O principal objetivo deste trabalho consistiu em desenhar e implementar um serviço simples de *proxy reverso TCP* em duas fases distintas. Na primeira fase desenhou-se e implementou-se um protocolo de monitorização sobre UDP de forma a obter e manter atualizada uma tabela, com informação dos servidores *back-end*, recolhida pelo servidor *Proxy TCP reverso*. Na segunda fase, implementou-se o servidor *proxy TCP* genérico. Este servidor fica à escuta na porta 80 e redireciona automaticamente cada conexão TCP na porta 80 que recebe para um dos servidores disponíveis com base numa métrica calculada a partir da informação mantida.

# Capítulo 1

## Introdução

O segundo trabalho da unidade curricular de *comunicações por computador* consiste em analisar, explorar e implementar uma abordagem muito comum em serviços *web*, um *Proxy TCP* reverso com monitorização proativa.

Quando se tem muitos serviços, um único servidor pode não ser suficiente para dar resposta a todos os pedidos que chegam de clientes. Nestas situações pode ser essencial ter uma *pool* de  $N$  servidores. Para fazer esta gestão utiliza-se um servidor principal, servidor *front-end*, que é o único ponto de entrada para todos os clientes. Este servidor tem definido um nome e um endereço de *IP* únicos, bem conhecidos. A tarefa essencial do servidor *front-end*, normalmente designado por *Proxy Reverso*, consiste em receber e atender as conexões de clientes e encaminhá-las para um dos  $N$  servidores *back-end* disponíveis.

A escolha de um dos servidores *back-end* disponíveis, pode ser cega e ter em conta, por exemplo, um algoritmo de *Round-Robin* que apenas faz uma distribuição equitativa das conexões que recebe pelos  $N$  servidores *back-end*. No entanto, recolhendo informação do estado dos servidores *back-end* e da rede, é possível calcular uma métrica dinâmica bem definida e redirecionar em função a este cálculo o que permite uma escolha mais ponderada.

O objetivo central deste trabalho consiste em desenhar e implementar um serviço simples utilizando a abordagem de um *Proxy Reverso TCP* e todo processo associado, em duas fases distintas:

1. **Monitorização:** Protocolo de monitorização da *pool* de servidores *back-end* para recolha de informação atualizada via *UDP*. Esta informação é recolhida e mantida numa tabela pelo servidor *proxy reverso* sendo obtida/calculada com base em parâmetros/dados dinâmicos.
2. **Proxy TCP reverso:** Implementação de um servidor *proxy TCP* genérico que fica à escuta na porta 80 e recebe conexões de clientes. Para cada conexão de cliente estabelecida:
  - determina um dos servidores *back-end* disponíveis tendo em consideração as medidas mantidas na tabela com o estado de cada servidor *back-end*;
  - intermedeia a conexão *TCP* entre o cliente e o servidor *back-end* selecionado.

# Capítulo 2

## Arquitetura da solução

O objetivo central do trabalho consiste em desenhar e implementar um serviço simples de *proxy reverse TCP*. Na Figura 2.1 encontra-se representada a arquitetura da solução a implementar.

Pool Servidores *Back-end*

Clientes

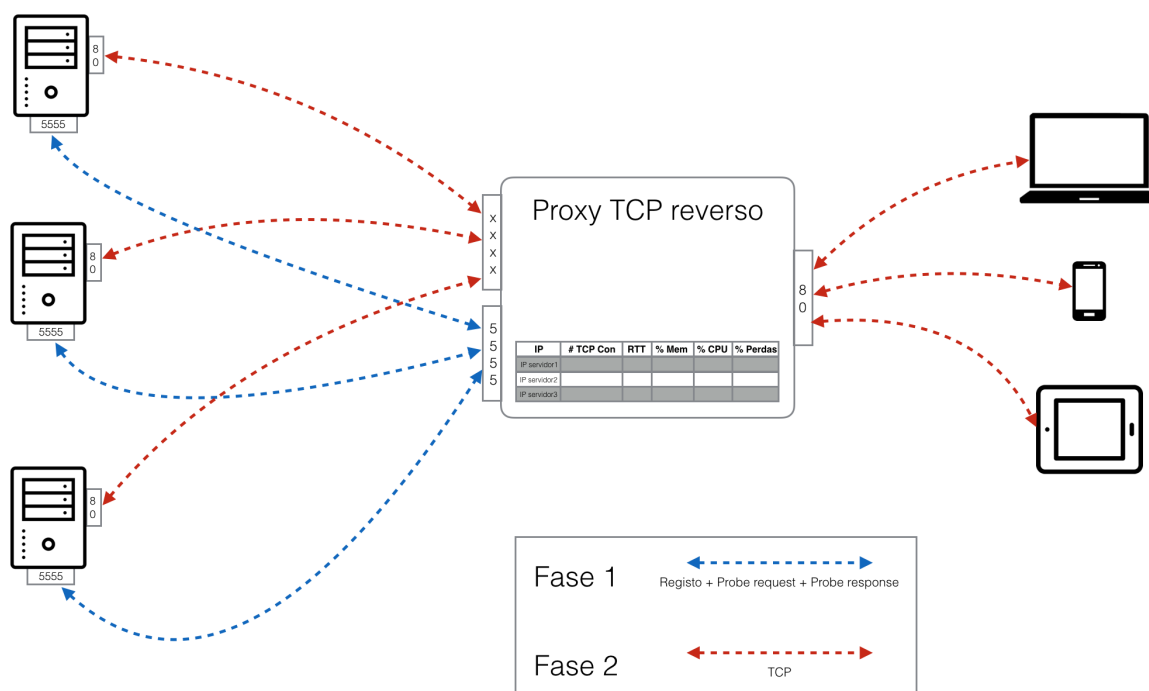


Figura 2.1: Arquitetura geral da solução a implementar

Observando a Figura 2.1, identifica-se que no sistema existem dois componentes essenciais:

1. **Agente de monitorização do estado dos servidores *back-end*:** a operar exclusivamente sobre UDP na porta 5555, comunica com o servidor *reverse proxy* enviando mensagens de monitorização.
2. **Servidor reverse proxy:** Mantém a informação de estado sobre cada servidor *back-end* e é capaz de intermediar as conexões TCP.

Na arquitetura da solução, na Figura 2.1, observa-se que o sistema pode ser dividido em duas fases distintas em que os componentes principais identificados intervêm.

**Fase 1** A primeira fase centra-se em desenvolver um de protocolo monitorização da *pool* de servidores *back-end* disponíveis.

Cada servidor de *back-end* tem um *MonitorUDP*, ou seja, um agente de monitorização que utiliza o protocolo de aplicação e comunica exclusivamente pela porta 5555 sob protocolo UDP.

O *ReverseProxy* é responsável por manter uma tabela de estado, dialogando em UDP com os agentes *MonitorUDP*. Esta tabela mantém a informação do estado de cada servidor *back-end* e é atualizada regularmente pelo agente de monitorização UDP. Na tabela de dados para cada servidor *back-end*, identificado pelo seu IP, encontra-se a seguinte informação:

- round-trip time (RTT) estimado entre o back-end e o servidor *proxy* reverso
- taxa de pacotes perdidos
- número de conexões TCP
- percentagem de uso CPU
- percentagem de uso memória

Os servidores *back-end* e *servidor proxy* comunicam através do protocolo implementado e trocam mensagens entre si:

- ***mensagem de registo***: O *MonitorUDP* envia, de x em x segundos, uma mensagem de registo a informar o servidor principal de que se encontra disponível. O *ReverseProxy* recebe a mensagem de registo e regista na tabela uma entrada associada ao IP do servidor *back-end* que a enviou.
- ***probe request***: É um pedido de monitorização. O servidor de *front-end* tem a responsabilidade de manter os dados atuais, e por isso faz um *pooling* periódico a cada servidor *back-end* que se manifestou disponível.
- ***probe response***: Quando o servidor *back-end* recebe uma mensagem de pedido de monitorização (*probe request*) deve responder com um *probe response* com informação sobre o seu estado. O *ReverseProxy* recebe a mensagem de *probe response* e atualiza na tabela a informação correspondente.

**Fase 2** A segunda fase consiste em implementar o servidor *proxy TCP* que atende pedidos TCP na porta 80 e intermedeia as conexões de clientes.

A tabela que o *reverse proxy* mantém, com dados de cada servidor da *pool* de servidores *back-end*, é utilizada para determinar e escolher o melhor servidor de *back-end* naquele instante, abrindo uma nova conexão TCP com a porta 80 desse servidor. Após isto, o servidor *proxy TCP* funciona como intermediário entre as duas conexões:

- Conexão TCP servidor *proxy TCP* → servidor *back-end* selecionado**
  - Recebe do *socket* TCP do cliente
  - Envia para o *socket* TCP do servidor *back-end*
- Conexão TCP servidor *proxy TCP* → cliente**
  - Recebe no *socket* TCP do servidor
  - Envia para o cliente

# Capítulo 3

## Especificação do protocolo de monitorização

### 3.1 Funcionamento geral

O protocolo de monitorização é um protocolo que opera em UDP e oferece um tipo de *socket*: `SocketMonitorizacao`. O *socket*, que funciona sobre um *socket UDP*, serve como veículo de transporte para os pacotes de monitorização. Estes pacotes de monitorização (`PacoteMonitorizacao`), que encapsulam um `DatagramPacket` (pacote UDP), têm associado:

- um IP de envio do pacote
- a porta de destino do pacote
- o pacote de dados de monitorização

O protocolo, como é comum num protocolo de comunicação, permite o envio e receção de dados, de seguida é descrito o comportamento do protocolo nas diferentes situações.

- **Envio de dados:**

O envio de dados do protocolo de monitorização é similar ao envio de dados do UDP. É extraído do pacote de monitorização a ser enviado, o `DatagramPacket` associado. Este pacote UDP já tem configurado o IP, a porta associada e os dados - o pacote de dados de monitorização. Este pacote UDP é depois enviado através do *socket* UDP.

- **Receção de dados:**

A receção de dados é feita sob a receção do protocolo UDP. É efetuada a receção do `DatagramPacket` e é feita a leitura e conversão para um `PacoteMonitorizacao`. Significando que é identificado e reconstruído o pacote de dados de monitorização. Este pacote é depois devolvido como valor de retorno.

### 3.2 Primitivas de comunicação

Para haver a troca de dados de comunicação, não é necessário que exista uma ligação prévia. Todos os pacotes são trocados apoiados no protocolo UDP e portanto não existe a noção de



ligação. Para que exista a troca de dados, alguém deverá estar à espera de receber um pacote numa porta específica, tal é feito utilizando o *socket* disponibilizado, `SocketMonitorizacao`.

Este `SocketMonitorizacao` funciona para os dois lados da comunicação, para quem quer enviar os dados e para quem quer receber. Assim, este *socket* disponibiliza as seguintes primitivas de comunicação:

- `new SocketMonitorizacao()`: cria um *socket* de monitorização sem porta associada, provavelmente este *socket* será utilizado para receber pacotes uma vez que não tem porta associada para ficar à escuta.
- `new SocketMonitorizacao(porta)`: cria um *socket* de monitorização com uma porta associada. Este *socket* pode ser utilizado para ficar à escuta da receção de pacotes de monitorização.
- `enviar(pacoteMonitorizacao)`: envia um pacote de monitorização para o ip e porta associados ao pacote
- `receber()`: fica à escuta para a receção de um pacote de monitorização. Este pacote quando recebido é retornado pela primitiva.

### 3.3 Formato das mensagens protocolares

Na conceção do protocolo, a definição dos pacotes de dados de monitorização foi um dos primeiros focos. Houve um levantamento dos diferentes tipos de pacotes que poderiam ser trocados e foram identificados os seguintes tipos de pacotes:

- **Registo**: Registo de um agente de monitorização
- ***Probe request***: Pedido de informações sobre a monitorização
- ***Probe response***: Resposta com informações da monitorização

Uma vez que os pacotes a serem trocados são sobretudo de informação, o tamanho do pacote de dados de monitorização definido foi igual para todos.

Todos os pacotes têm o tipo que, tal como esperado, permite distinguir o pacote diferente recebido. A estrutura do pacote geral é a seguinte:



Figura 3.1: Pacote geral

O **pacote de registo**, para além do tipo, contém também a informação de quanto em quanto tempo é que o monitor espera enviar mensagens de registo. Assim, a parte da informação do pacote definido foi a seguinte:

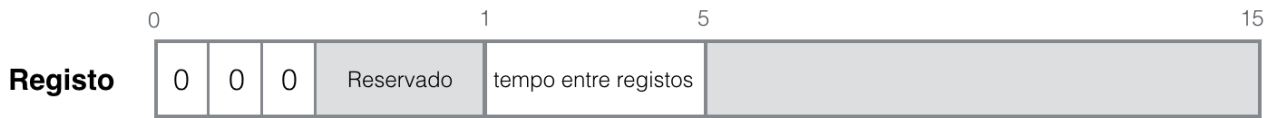


Figura 3.2: Pacote de registo

O pacote de *probe request*, para além do tipo, tem um número de sequência associado, este número de sequência serve para mais tarde poder ser confirmado e fornecer a informação da percentagem de perdas. A estrutura da informação presente no pacote de *probe request* foi definida como:

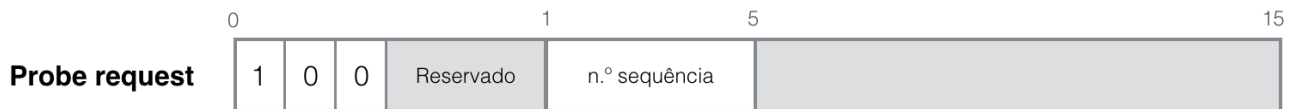


Figura 3.3: Pacote de *probe request*

O pacote de resposta (*probe response*) contém as informações de monitorização definidas no agente. Contém também o número de sequência do pedido de informações, confirmando assim a sua receção. Desta forma, é possível montar um controlo de informações de monitorização que permita construir um estado para cada um dos agentes de monitorização. A estrutura de informação do pacote de *probe response* foi definida como:

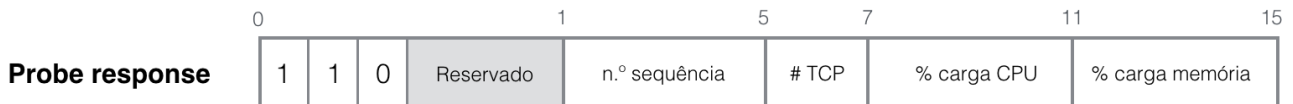


Figura 3.4: Pacote de *probe response*

### 3.4 Interações

Como indicado anteriormente, o protocolo definido opera sobre o protocolo UDP e está entre a camada da aplicação e a camada de transporte, oferecendo uma abstração da transferência de pacotes de monitorização. Assim, é natural criar uma nova camada tal como nas camadas do modelo OSI, isolando responsabilidades e funcionalidades.

Mesmo que seja necessário trocar o protocolo de transporte sobre o qual opera o protocolo de monitorização a sua substituição vai ser independente para as aplicações.

A Figura 3.5 contém a ilustração desta nova camada e das suas interações.

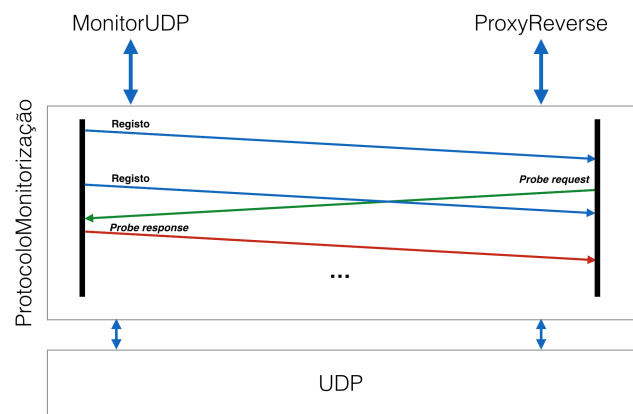


Figura 3.5: Camada do protocolo de monitorização e suas interações

# Capítulo 4

## Implementação

### 4.1 Monitorização

#### 4.1.1 Servidor de back-end

O *monitor UDP* é um agente de monitorização do estado de um servidor de *back-end*. Isto é, cada servidor de *back-end* terá um monitor UDP, que irá monitorizar o seu estado da máquina, recolhendo métricas e enviando-as ao servidor *Reverse Proxy*, tal como esquematizado na Figura 4.1.

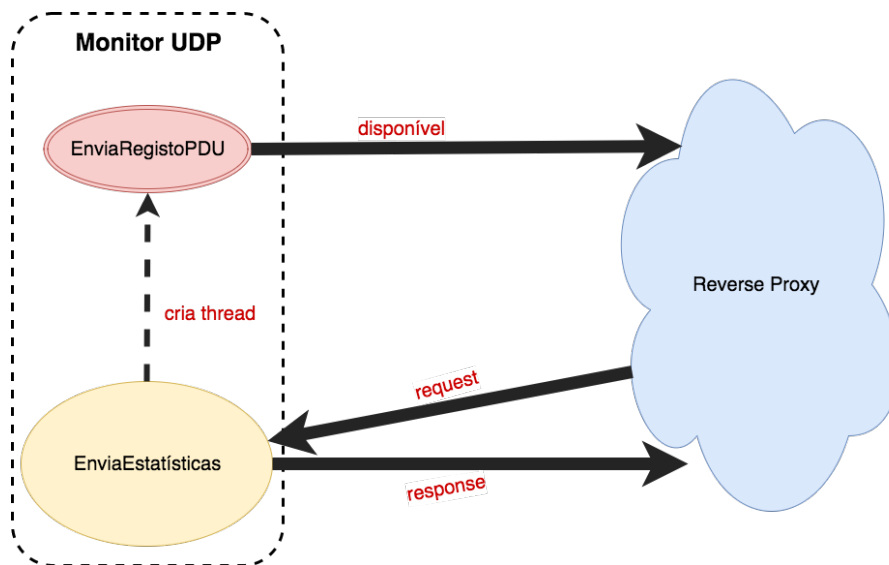


Figura 4.1: Esquema geral da implementação do Monitor UDP e a troca de mensagens com o *Reverse Proxy*.

Como já referido, cada servidor de *back-end* terá um monitor UDP que irá receber como parâmetro na inicialização o intervalo de tempo em segundos que irá enviar o PDU de registo ao *Reverse Proxy*. Quando o monitor entra em execução cria uma *thread* que envia de x em x segundos, isto é, envia em intervalos de tempo recebido como parâmetro pelo programa, o PDU registo que indica ao *Reverse Proxy* que este se encontra disponível para lhe enviar informações sobre o estado do servidor de *back-end* que está a monitorizar.

Paralelamente a isso, o monitor UDP fica à espera de receber um *PDU request* do *Reverse Proxy*. Este servidor de *front-end* faz um pedido ao monitor de UDP de um dado servidor de

*back-end*, para este lhe enviar a informação recolhida sobre o estado da máquina, de modo ao *Reverse Proxy* atualizar a sua tabela de estados, para mais tarde balancear e distribuir a carga de trabalho pelos servidores de *back-end*.

Quando o monitor UDP recebe um *PDU request* terá que proceder ao cálculo de diversas métricas de monitorização, que depois enviará ao servidor *Reverse Proxy* num *PDU response*.

Nesse *PDU response*, para além de enviar o mesmo número de sequência do *PDU request*, irá enviar as seguintes métricas:

- Número de conexões TCP ativas na máquina
- % Carga de CPU desde o último PDU request recebido
- % Memória (RAM) ocupada

**Número de Conexões TCP** O número de conexões ativas calculou-se recorrendo ao comando UNIX `netstat`. O `netstat` é um utilitário de rede que pode ser usado para verificar conexões de TCP ativas. No monitor UDP, executou-se o seguinte comando:

```
netstat -nt | grep -i ESTABLISHED | wc -l
```

O `netstat -nt` devolve as atuais conexões TCP do servidor de *back-end* e depois filtra-se este output através do comando `grep -i ESTABLISHED` de modo a ficar somente com as conexões TCP ativas, e por fim, calcula-se o número total de resultados (linhas) com o comando `wc -l`.

**Carga de CPU e Memória Ocupada** Para a obtenção da percentagem de carga de CPU e da percentagem de memória ocupada recorreu-se à biblioteca *JavaSysMon* que disponibiliza métodos que permitem obter estas métricas.

A percentagem de carga de CPU será uma média entre o último instante que se obteve a essa informação até ao instante atual. Quanto à memória ocupada, como o *JavaSysMon* apenas fornece o tamanho absoluto de memória RAM disponível e memória RAM total para um dado instante, calcula-se a percentagem de memória ocupada através da fórmula:

$$\%MEM_{ocupada} = 100 - \left( \frac{MEM_{livre}}{MEM_{total}} \times 100 \right)$$

### 4.1.2 Servidor proxy

O servidor de proxy está dividido em duas partes, a parte relativa à monitorização e a parte de reencaminhamento de tráfego de TCP.

Relativamente à monitorização, o servidor de proxy tem um agente de monitorização (**Receptor**) que está sempre pronto a receber os pacotes enviados pelos agentes de monitorização dos servidores de *back-end*. Este agente está à escuta na porta 5555 e é ele que trata os pacotes recebidos, atualizando o estado de cada um dos servidores de *back-end* envolvidos.

O servidor de proxy contém uma tabela (**EstadoGlobalServidores**) com a informação do estado de cada um dos servidores de *back-end* que comunicou o Registo com o servidor de proxy. Esta informação é atualizada a cada pacote recebido ou com a ausência de respostas esperadas. Na Figura 4.2 encontra-se uma ilustração desta tabela e da sua informação bem como a o funcionamento geral do mecanismo de monitorização do servidor de *proxy* reverso.

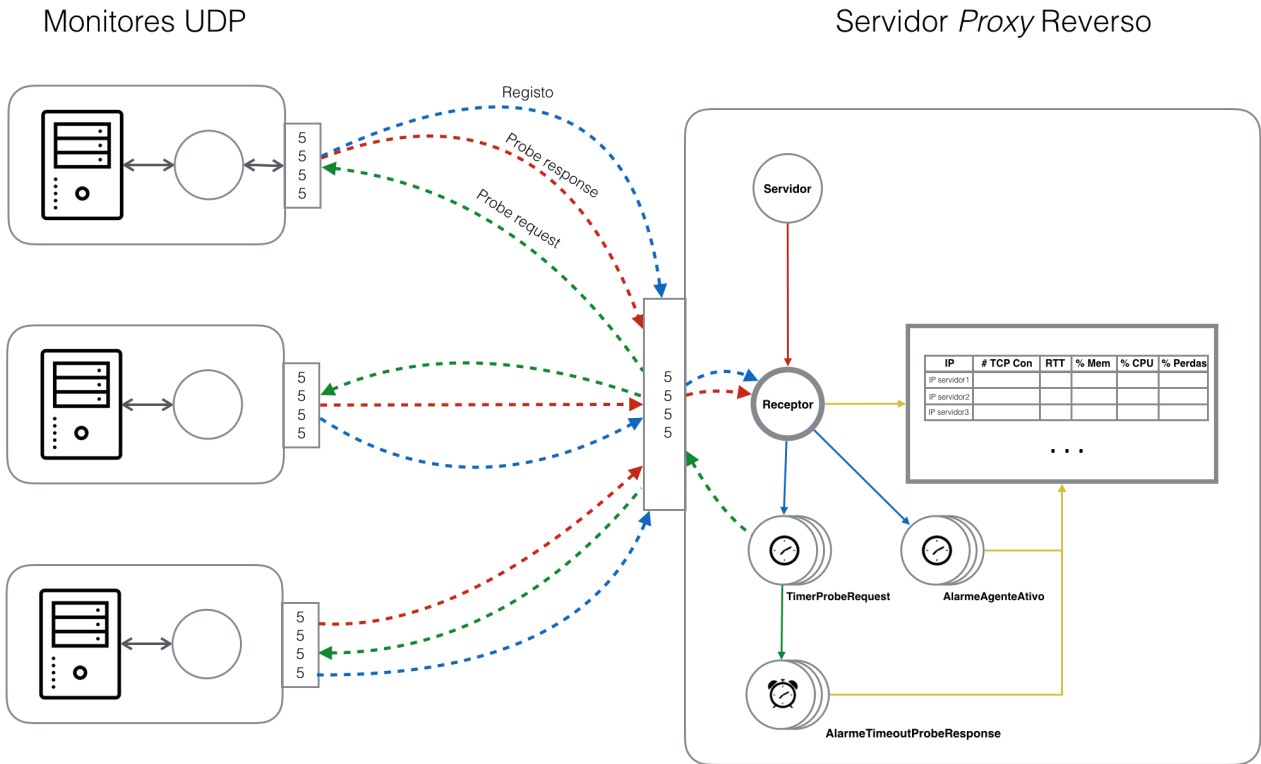


Figura 4.2: Mecanismo de monitorização do servidor de *proxy* reverso

O funcionamento agente de monitorização do servidor de proxy para os diferentes pacotes recebidos é explicado de seguida.

### Pacote de registo

Quando é recebido um pacote de registo é feita uma consulta do estado dos servidores de *back-end*. Se já existir um estado associado ao servidor de *back-end* em questão, é atualizado o tempo entre registos que o agente de monitorização do servidor de *back-end* comunicou. Caso ainda não exista então é criado um estado para o servidor de *back-end* (o IP é dado pelo pacote UDP recebido).

Para além de inicializar o estado é também despoletado um mecanismo para recorrentemente (com um tempo definido no servidor de proxy), enviar pedidos de informação de monitorização (*probe request*) para o servidor de *back-end*. Este mecanismo é feito numa nova thread (**TimerProbeRequest**). Para cada um dos *probe request* enviados, é acionado um alarme (**AlarmeTimeoutProbeRequest**) para garantir que, caso uma resposta não chegue num tempo pré-definido (também definido a nível do servidor proxy), é considerada uma falha de pacote no estado do servidor de *back-end*. No estado do servidor de *back-end* ficam guardados os pacotes enviados e ainda não confirmados bem como a sua data de envio.

É por fim registado um mecanismo para verificar que o servidor de *back-end* ainda se encontra ativo. Este mecanismo (**AlarmeAgenteAtivo**) corre periodicamente com um tempo pré-definido no servidor de proxy, verifica que o servidor foi trocando mensagens desde a última verificação, marcando o servidor como inactivo caso tal não tenha ocorrido.

## Pacote de probe response

Quando é recebido um pacote de resposta de informação (*probe response*), é atualizado o estado do servidor. Cada pacote de *probe response* contém o número de sequência do *probe request* associado. Esta atualização pode significar duas coisas:

- Caso o pacote chegue dentro do *timeout* definido para a chegada do *probe response*, o pacote ainda vai estar na lista dos pacotes por confirmar e então é atualizada a informação do servidor bem como é contabilizado como um pacote confirmado.
- Caso o pacote chegue fora do *timeout* definido, ele não vai estar na lista dos pacotes por confirmar e então a informação que o pacote traz é descartada (uma vez que poderá já estar desatualizada). Não é contabilizada uma perda já que o mecanismo de *timeout* (*AlarmeTimeoutProbeRequest*) já o assinalou.

## Pacote de probe request

Não é esperado que o servidor de proxy receba este tipo de pacotes mas na eventualidade de um pacote destes chegar ele é simplesmente descartado.

## Fluxo de monitorização

A Figura 4.3 demonstra o fluxo do mecanismo de monitorização do servidor *proxy* reverso para um qualquer servidor de *back-end* e demonstra as diferentes trocas de pacotes e ações que acontecem.

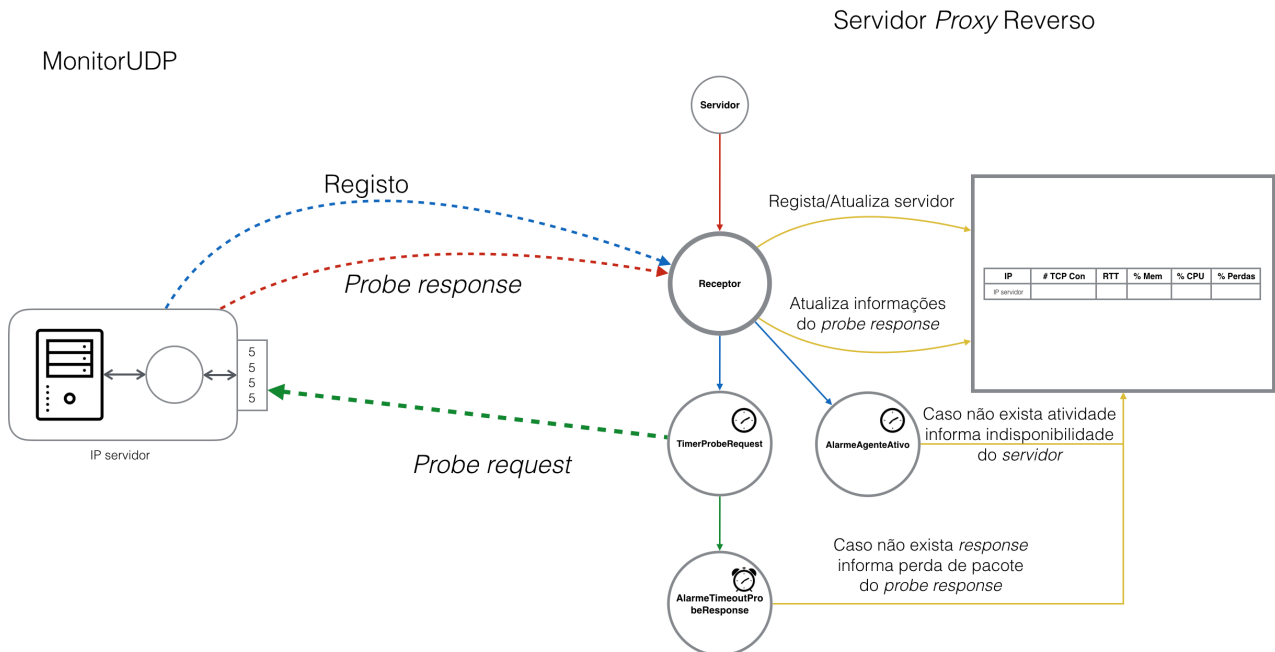


Figura 4.3: Troca de pacotes de monitorização entre o servidor de *proxy* reverso e um agente de monitorização

## Parametrização

A parametrização do funcionamento do agente de monitorização do servidor de *back-end* foi feita com os seguintes valores:

- **timeOutProbeRequest**: 500 milisegundos - Tempo que se espera para receber um *probe request*, caso não seja recebido, é considerada uma perda;
- **intervaloEntreEnviosProbeRequest** 5000 milisegundos - Tempo periódico que define de quanto em quanto tempo é que é enviado um *probe request* para o servidor *back-end*;
- **tempoTimeOutServidorAtivo** 10000 milisegundos - Tempo periódico que caracteriza que quanto em quanto tempo se deve verificar se o servidor teve alguma atividade. Caso não tenha existido, o servidor é considerado inativo;

## 4.2 Re-encaminhamento de tráfego

A implementação da funcionalidade de reencaminhamento do tráfego TCP teve preocupações com a eficiência de transmissão. O servidor de *proxy*, está sempre à escuta do estabelecimento de conexões TCP na porta por defeito (80). Quando é estabelecida uma conexão TCP, é criada uma thread (*TrataClienteTCP*) que tratará este cliente enquanto a conexão estiver ativa.

A thread *TrataClienteTCP* recebe o estado dos vários servidores de *back-end* e pede-lhe o IP do servidor de *back-end* para tratar o seu *request*. O cálculo de qual o servidor de *back-end* que deve tratar o próximo pedido encontra-se explicado a seguir.

### Cálculo do servidor de *back-end* a tratar o próximo pedido

O estado de cada um dos servidores de *back-end* é mantido numa espécie de tabela. Como referido anteriormente, para cada um dos servidores é guardada informação como: a percentagem da carga de CPU, percentagem da carga de memória, número de conexões TCP, percentagem de pacotes perdidos e tempo médio de resposta (RTT).

Para conseguir pesar as informações de monitorização, desenvolveu-se uma fórmula que combina estes factores e produz um valor numérico para cada um dos servidores. Quanto menor este valor, melhor será a avaliação do servidor. A fórmula desenvolvida encontra-se definida na Figura 4.4. Para cada um dos parâmetros é definido um valor ideal e depois é encontrada a razão entre o valor ideal e o valor atual. É de notar que o peso dado a cada um dos componentes varia, sendo que os mais importantes são o tempo do RTT, a percentagem de perdas de pacotes e o número de conexões TCP (30%). Caso a percentagem de CPU, memória, ou percentagem de pacotes perdidos atinjam um *threshold* de 70%, a partir do qual se espera que a performance do servidor de *back-end* se degrade bastante, o valor devolvido é enorme (simulando o infinito) para não levar à sua escolha. Caso este *threshold* não se verifique, a percentagem de carga de CPU e memória não tem uma contribuição tão relevante para o cálculo.



```

if (percentagemCargaCPU >= 0.77 ||
    percentagemCargaMemoria >= 0.77 ||
    percentagemPerdas >= 0.7) {

    valor ← 100;
}

else {
    rttlIdeal ← 20;
    nrConexoesTCPIdeal ← 10;
    percentagemPerdasIdeal ← 0.01;
    percentagemCargaCPUIdeal ← 0.1;
    percentagemCargaMemorialdeal ← 0.1;

    valor ← 0.3 x (sampleRTT / rttlIdeal) +
        0.3 x (nrConexoesTCP / nrConexoesTCPIdeal) +
        0.3 x (percentagemPerdas / percentagemPerdasIdeal) +
        0.05 x (percentagemCargaCPU / percentagemCargaCPUIdeal) +
        0.05 x (percentagemCargaMemoria / percentagemCargaMemorialdeal);
}

```

Figura 4.4: Fórmula de classificação de um servidor de *back-end* de acordo com as suas métricas

Com esta classificação dos diferentes servidores de *back-end*, é feita uma ordenação ascendente, onde é escolhido o primeiro, ou seja, o que, em princípio, tem melhores condições para responder ao pedido TCP.

## Funcionamento

Após a escolha do servidor para responder ao pedido, e como forma de aumentar a performance da conexão, a leitura dos dados do cliente e envio para o servidor de *back-end* bem como a leitura da resposta do servidor de *back-end* para o cliente, é feita em paralelo com a utilização de uma thread extra. A thread principal trata de ler do cliente e enviar para o servidor, e a thread auxiliar faz o processo inverso. A Figura 4.5 representa o diagrama dos diferentes fluxos deste processo de reencaminhamento.

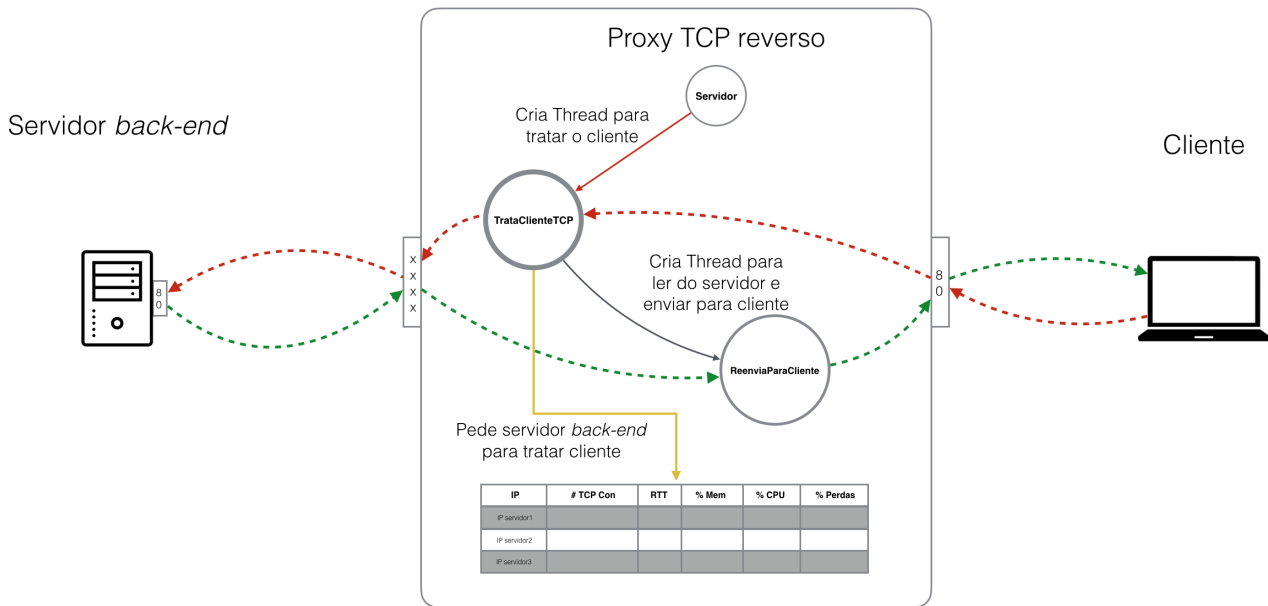


Figura 4.5: Mecanismo de reencaminhamento do servidor de *proxy* reverso

#### 4.2.1 Considerações adicionais

Houve a investigação da possibilidade de utilização de cache para a resposta aos pedidos TCP, contudo, esta hipótese foi descartada devido às seguintes motivações:

- Paralelização: a leitura dos dados do cliente e escrita para o servidor de *back-end*, bem como a leitura dos dados do servidor de *back-end* e escrita para o cliente é feita em paralelo. Para implementar uma cache obrigaria que fossem primeiro lidos todos os dados do cliente e depois houvesse uma procura na cache para ver se o pedido lá se encontrava. Assim, perder-se-ia eficiência e a menos que os dados a serem respondidos pelo servidor fossem de um tamanho bastante considerável e que os dados do cliente tivessem pouco tamanho, o que compensaria estar à espera da resposta do servidor.
- Conteúdo dinâmico: não é garantido que a resposta do servidor de *back-end* a um exacto pedido TCP seja exactamente igual. Uma alteração no contexto do servidor de *back-end* pode levar a que a resposta seja diferente e a implementação de cache poderia alterar esse comportamento. Mais uma vez, para a implementação de cache ter-se-ia que confirmar que esse ponto não seria um problema.

Assim, para manter um mecanismo de proxy genérico, não foi implementado um mecanismo de cache.

### 4.3 Considerações adicionais globais

As informações guardadas no servidor proxy, encontram-se acessível em múltiplas *threads* e pode sofrer de alterações em qualquer um dos instantes. Houve uma grande preocupação em manter o acesso exclusivo a estas informações, controlando a concorrência a estes recursos. A informação do estado dos servidores de *back-end* foi o recurso mais crítico uma vez que tanto a monitorização como o reencaminhamento de tráfego TCP fazem uso desse estado.

# Capítulo 5

## Testes e resultados

Tal como sugerido no enunciado do trabalho, foi usado a aplicação Core para testar o trabalho com a topologia CC-Topo-2017.imn, como se mostra na Figura 5.1.

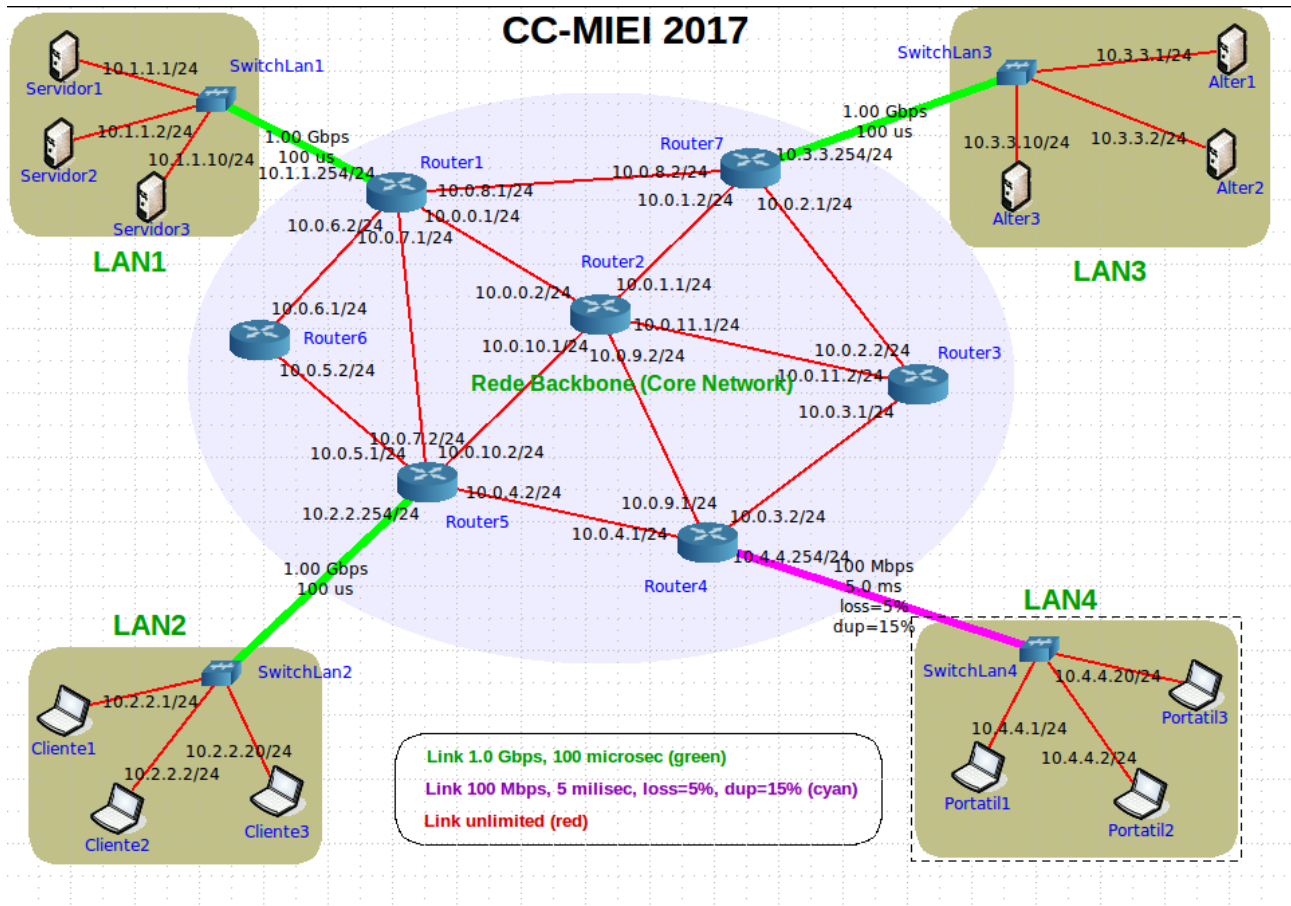


Figura 5.1: Topologia de rede dentro do CORE usada para os testes.

O *Servidor1* e o *Servidor2* serão as máquinas de *back-end* onde correrá o servidor HTTP *mini-httpd* e o programa do Monitor UDP.

O *Servidor3* será onde se vai executar o *Reserve Proxy*, que encaminhará pedidos HTTP na porta 80, para uma das duas máquinas a correr (*Servidor1* ou *Servidor2*).

## 5.1 Inicialização

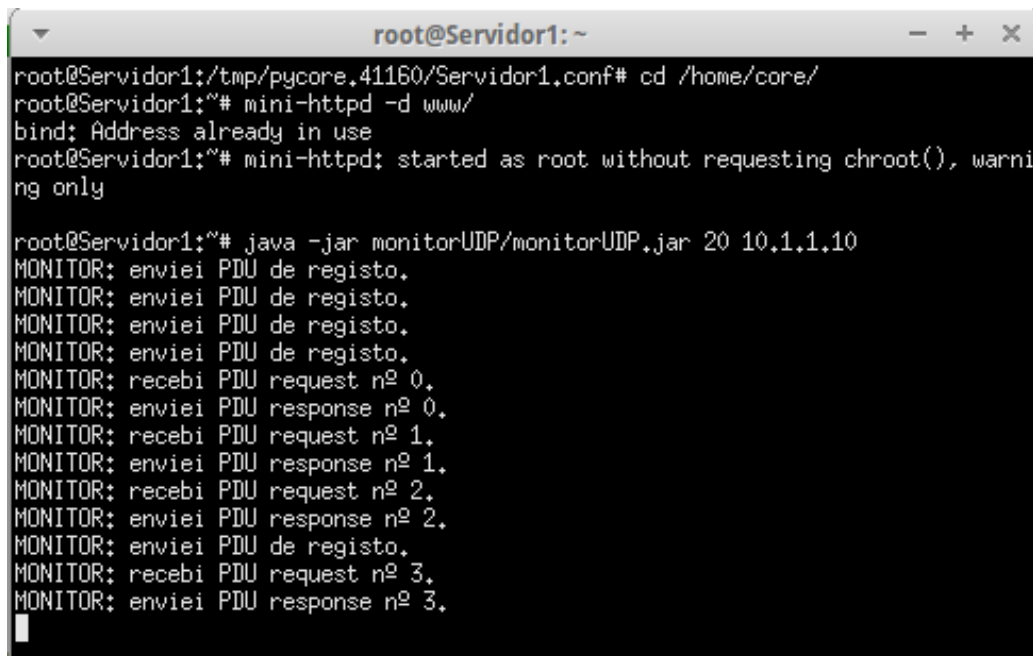
Inicialmente, executa-se o core com a topologia `CC-Topo-2017.imn` e inicia-se a sessão. Após estarem a correr todas as máquinas da topologia, inicia-se uma sessão de `bash` com todas as máquinas da LAN1: Servidor1, Servidor2, Servidor3.

### 5.1.1 Back-end

Nas máquinas Servidor1 e Servidor2 inicia-se o servidor HTTP, neste caso o *mini-httpd*, usando a pasta `www` criada na *home* da máquina, como raiz do site: `mini-httpd -d www`.

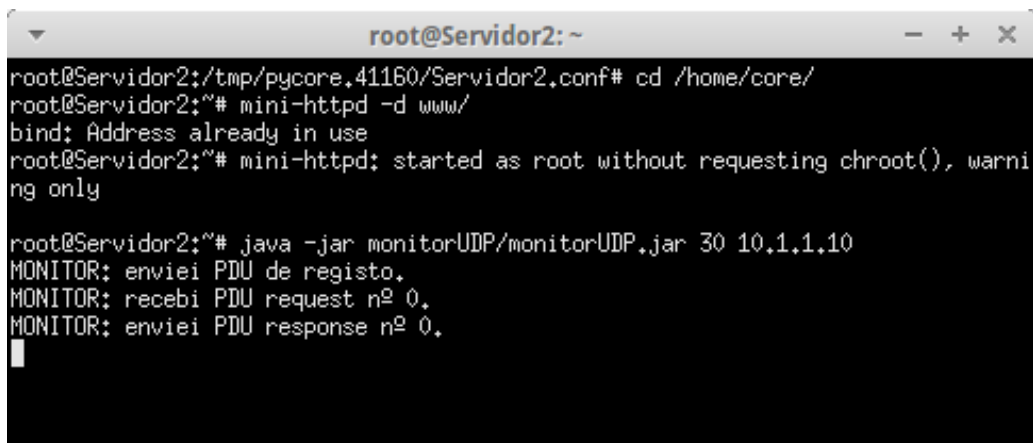
Após ligar o servidor HTTP, podemos começar a executar o monitor UDP, especificando quanto será o intervalo de tempo de mensagens Registro e o IP do Reserve Proxy: `java -jar monitorUDP 20 10.1.1.10`.

Pode-se confirmar este passo nas Figuras 5.2 e 5.3.



```
root@Servidor1:~  
root@Servidor1:/tmp/pycore.41160/Servidor1.conf# cd /home/core/  
root@Servidor1:~# mini-httpd -d www/  
bind: Address already in use  
root@Servidor1:~# mini-httpd: started as root without requesting chroot(), warni  
ng only  
  
root@Servidor1:~# java -jar monitorUDP/monitorUDP.jar 20 10.1.1.10  
MONITOR: enviei PDU de registo.  
MONITOR: enviei PDU de registo.  
MONITOR: enviei PDU de registo.  
MONITOR: enviei PDU de registo.  
MONITOR: recebi PDU request nº 0.  
MONITOR: enviei PDU response nº 0.  
MONITOR: recebi PDU request nº 1.  
MONITOR: enviei PDU response nº 1.  
MONITOR: recebi PDU request nº 2.  
MONITOR: enviei PDU response nº 2.  
MONITOR: enviei PDU de registo.  
MONITOR: recebi PDU request nº 3.  
MONITOR: enviei PDU response nº 3.  
█
```

Figura 5.2: Inicialização do servidor HTTP e Monitor UDP no Servidor1.



```
root@Servidor2: ~
root@Servidor2:/tmp/pycore.41160/Servidor2.conf# cd /home/core/
root@Servidor2:~# mini-httpd -d www/
bind: Address already in use
root@Servidor2:~# mini-httpd: started as root without requesting chroot(), warning only

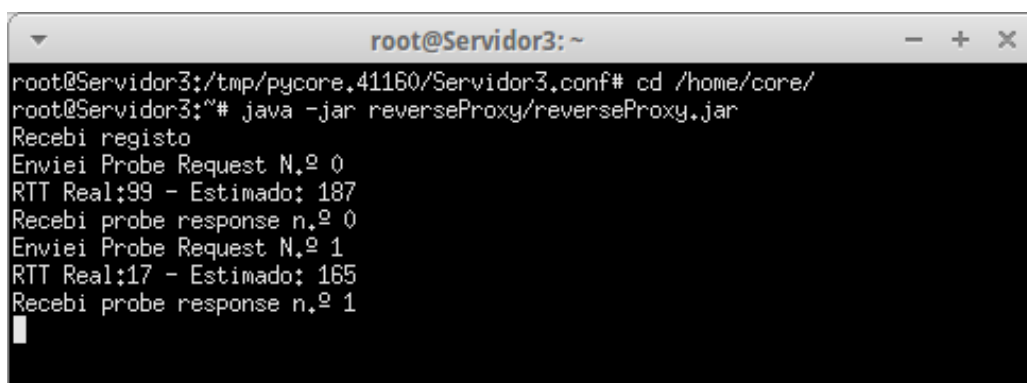
root@Servidor2:~# java -jar monitorUDP/monitorUDP.jar 30 10.1.1.10
MONITOR: enviei PDU de registo.
MONITOR: recebi PDU request nº 0.
MONITOR: enviei PDU response nº 0.
```

Figura 5.3: Inicialização do servidor HTTP e Monitor UDP no Servidor2.

### 5.1.2 Reverse Proxy

No Servidor3, basta iniciar o programa do Reverse Proxy, que este passa a receber as mensagens dos Monitores UDP a correr nas outras máquinas e passa também a aceitar pedidos HTTP na porta 80. Para tal, corre-se o comando: `java -jar reserveProxy.jar`.

Pode-se confirmar este passo na Figura 5.4.



```
root@Servidor3: ~
root@Servidor3:/tmp/pycore.41160/Servidor3.conf# cd /home/core/
root@Servidor3:~# java -jar reverseProxy/reverseProxy.jar
Recebi registo
Enviei Probe Request N.º 0
RTT Real:99 - Estimado: 187
Recebi probe response n.º 0
Enviei Probe Request N.º 1
RTT Real:17 - Estimado: 165
Recebi probe response n.º 1
```

Figura 5.4: Inicialização do Reverse Proxy no Servidor2.

## 5.2 Teste 1

Agora que se tem 2 servidores de *back-end* e o *Reserve Proxy* a correr, apenas falta testar se realmente se consegue ler o `index.html` presente na pasta `www`. Para tal, iniciou-se uma sessão de bash na máquina *Cliente1*.

Primeiro foi testado se um dos servidores de *back-end* servia o `index.html` caso fosse contactado diretamente. Na Figura 5.5 mostra-se que de facto, o *Cliente1* consegue ler o ficheiro a partido do servidor HTTP a correr no Servidor1, através do comando `wget`.

```
root@Cliente1: /tmp/pycore.41160/Cliente1.conf
root@Cliente1:/tmp/pycore.41160/Cliente1.conf# wget 10.1.1.1
--2017-05-22 17:42:07-- http://10.1.1.1/
Connecting to 10.1.1.1:80... connected.
HTTP request sent, awaiting response... 200 Ok
Length: 30 [text/html]
Saving to: `index.html'

100%[=====] 30          --.-K/s   in 0s

2017-05-22 17:42:07 (66.8 KB/s) - `index.html' saved [30/30]

root@Cliente1:/tmp/pycore.41160/Cliente1.conf# cat index.html
<h1> hello world CC TP2 </h1>
root@Cliente1:/tmp/pycore.41160/Cliente1.conf#
```

Figura 5.5: Leitura do index.html diretamente ao Servidor1.

Estando confirmado que os servidores HTTP no Servidor1 e Servidor2 estão a executar corretamente, resta agora tentar ler o mesmo ficheiro `index.html`, mas agora indiretamente através do Reverse Proxy que está a correr na máquina Servidor3. Como se pode ver pela Figura 5.6, o pedido foi executado com sucesso e o conteúdo do ficheiro lido de facto reflete o ficheiro original presente na pasta `www`.

```
root@Cliente1: /tmp/pycore.41160/Cliente1.conf
root@Cliente1:/tmp/pycore.41160/Cliente1.conf# wget 10.1.1.10
--2017-05-22 17:42:44-- http://10.1.1.10/
Connecting to 10.1.1.10:80... connected.
HTTP request sent, awaiting response... 200 Ok
Length: 30 [text/html]
Saving to: `index.html'

100%[=====] 30          --.-K/s   in 0s

2017-05-22 17:42:44 (5.64 MB/s) - `index.html' saved [30/30]

root@Cliente1:/tmp/pycore.41160/Cliente1.conf# cat index.html
<h1> hello world CC TP2 </h1>
root@Cliente1:/tmp/pycore.41160/Cliente1.conf#
```

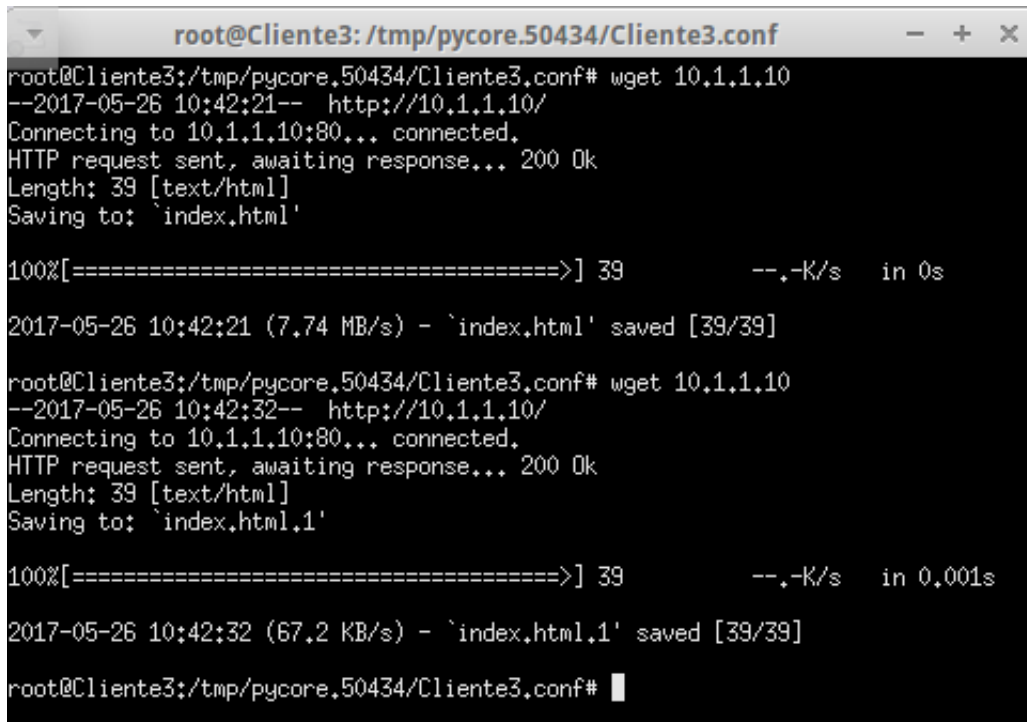
Figura 5.6: Leitura do index.html indiretamente através do Reverse Proxy no Servidor3.

## 5.3 Teste 2

Tal como foi sugerido na aula final teorico-prática onde se apresentou este trabalho, foi realizado um teste adicional onde é manipulado um dos servidores de *back-end* de forma a reportar ao *reverse proxy* estatísticas da máquina com valores muito altos, de forma a influenciar o ranking dos servidores de *back-end* no *reverse proxy*.

O teste consiste em ir buscar o `index.html` 10 vezes através do *reverse proxy* (ver Figura

5.7) e ver qual o ficheiro fornecido, isto porque os servidores de *back-end* estão a servir ficheiros `index.html` diferentes, que especificam no conteúdo quem foi o servidor que atendeu um pedido.



```
root@Cliente3: /tmp/pycore.50434/Cliente3.conf
root@Cliente3:/tmp/pycore.50434/Cliente3.conf# wget 10.1.1.10
--2017-05-26 10:42:21-- http://10.1.1.10/
Connecting to 10.1.1.10:80... connected.
HTTP request sent, awaiting response... 200 Ok
Length: 39 [text/html]
Saving to: `index.html'

100%[=====] 39          --.-K/s   in 0s

2017-05-26 10:42:21 (7.74 MB/s) - `index.html' saved [39/39]

root@Cliente3:/tmp/pycore.50434/Cliente3.conf# wget 10.1.1.10
--2017-05-26 10:42:32-- http://10.1.1.10/
Connecting to 10.1.1.10:80... connected.
HTTP request sent, awaiting response... 200 Ok
Length: 39 [text/html]
Saving to: `index.html.1'

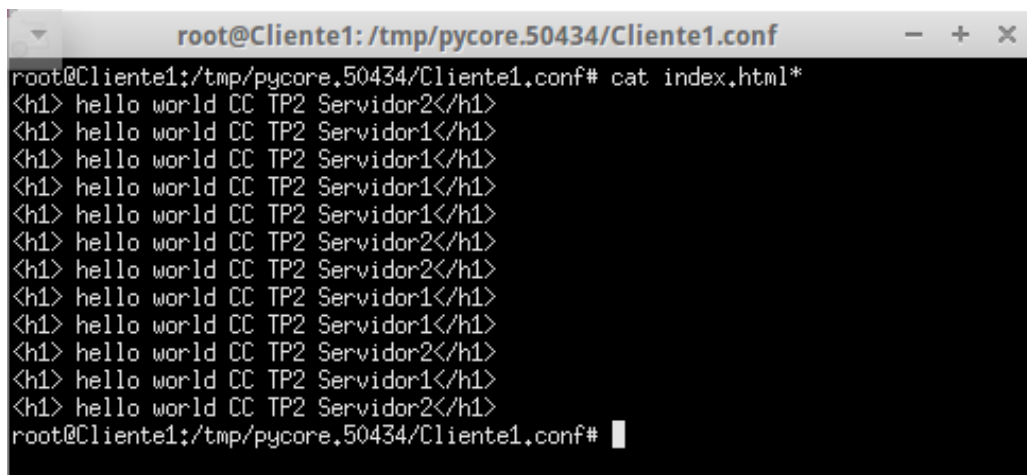
100%[=====] 39          --.-K/s   in 0.001s

2017-05-26 10:42:32 (67.2 KB/s) - `index.html.1' saved [39/39]

root@Cliente3:/tmp/pycore.50434/Cliente3.conf#
```

Figura 5.7: Leitura consecutiva do ficheiro `index.html` através do *Reverse Proxy*.

Na Figura 5.8 mostra-se um exemplo do resultado deste teste em condições normais, onde os pedidos são espalhados pelos servidores de *back-end*.



```
root@Cliente1: /tmp/pycore.50434/Cliente1.conf
root@Cliente1:/tmp/pycore.50434/Cliente1.conf# cat index.html*
<h1> hello world CC TP2 Servidor2</h1>
<h1> hello world CC TP2 Servidor1</h1>
<h1> hello world CC TP2 Servidor1</h1>
<h1> hello world CC TP2 Servidor1</h1>
<h1> hello world CC TP2 Servidor1</h1>
<h1> hello world CC TP2 Servidor2</h1>
<h1> hello world CC TP2 Servidor2</h1>
<h1> hello world CC TP2 Servidor1</h1>
<h1> hello world CC TP2 Servidor1</h1>
<h1> hello world CC TP2 Servidor2</h1>
<h1> hello world CC TP2 Servidor1</h1>
<h1> hello world CC TP2 Servidor2</h1>
root@Cliente1:/tmp/pycore.50434/Cliente1.conf#
```

Figura 5.8: Conteúdo dos ficheiros `index.html` lidos pelo cliente 10 vezes.

Para demonstrar a relevância das estatísticas enviadas pelo *monitorUDP* para a escolha do servidor de *back-end*, o Servidor2 executou uma versão modificada o *monitorUDP*, de maneira a que o CPU e a RAM estejam sempre a 100% de utilização e o número de conexões TCP também seja sempre 100.

A Figura 5.9 mostra o Servidor2 a correr o programa alterado.

```

root@Servidor2: ~
MONITOR: recebi PDU request nº 2.
IP:Servidor2 -> CPU e RAM para 100% e TCP com 100 ligações.
MONITOR: enviei PDU response nº 2.
MONITOR: enviei PDU de registo.
MONITOR: recebi PDU request nº 3.
IP:Servidor2 -> CPU e RAM para 100% e TCP com 100 ligações.
MONITOR: enviei PDU response nº 3.
MONITOR: recebi PDU request nº 4.
IP:Servidor2 -> CPU e RAM para 100% e TCP com 100 ligações.
MONITOR: enviei PDU response nº 4.
MONITOR: recebi PDU request nº 5.
IP:Servidor2 -> CPU e RAM para 100% e TCP com 100 ligações.
MONITOR: enviei PDU response nº 5.
MONITOR: recebi PDU request nº 6.
IP:Servidor2 -> CPU e RAM para 100% e TCP com 100 ligações.
MONITOR: enviei PDU response nº 6.
MONITOR: enviei PDU de registo.
MONITOR: recebi PDU request nº 7.

```

Figura 5.9: Output do *MonitorUDP* no Servidor2 onde informa que alterou as estatísticas enviadas ao *Reverse Proxy*.

A Figura 5.10 mostra o *score* dado a cada *back-end* pelo *reverse proxy* quando chega um novo pedido. Como é possível ver, o *score* do Servidor2 é muito pior que o Servidor1 devido à versão modificada do *monitorUDP* a correr no Servidor2.

```

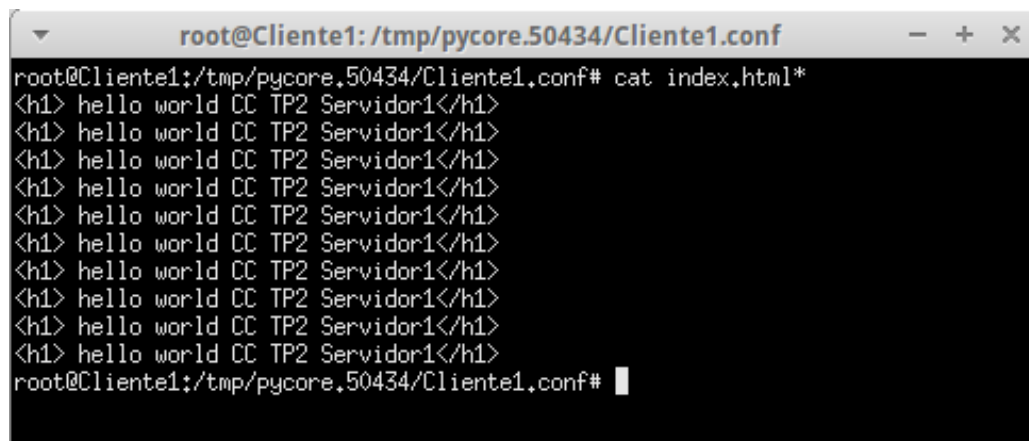
root@Servidor3: ~
:: [10.1.1.1] RTT Real:11 - Estimado: 169
>> [10.1.1.1] PDU Probe response - Nr Seq: 1
IP:10.1.1.1 Score:0.695 RTT:11 TCP:0 CPU:17 RAM:89
IP:10.1.1.2 Score:4.06 RTT:4 TCP:100 CPU:100 RAM:100
>> [10.1.1.2] PDU Registo
<< [10.1.1.2] PDU Probe request - Nr Seq: 3
:: [10.1.1.2] RTT Real:3 - Estimado: 130
>> [10.1.1.2] PDU Probe response - Nr Seq: 3
<< [10.1.1.1] PDU Probe request - Nr Seq: 2
:: [10.1.1.1] RTT Real:8 - Estimado: 148
>> [10.1.1.1] PDU Probe response - Nr Seq: 2
IP:10.1.1.1 Score:0.6 RTT:8 TCP:0 CPU:7 RAM:89
IP:10.1.1.2 Score:4.045 RTT:3 TCP:100 CPU:100 RAM:100
<< [10.1.1.2] PDU Probe request - Nr Seq: 4
:: [10.1.1.2] RTT Real:2 - Estimado: 114
>> [10.1.1.2] PDU Probe response - Nr Seq: 4
<< [10.1.1.1] PDU Probe request - Nr Seq: 3
>> [10.1.1.1] PDU Registo
:: [10.1.1.1] RTT Real:11 - Estimado: 130
>> [10.1.1.1] PDU Probe response - Nr Seq: 3
<< [10.1.1.2] PDU Probe request - Nr Seq: 5

```

Figura 5.10: Output do *Reverse Proxy* onde mostra o *score* dado a cada *back-end* aquando a chegada de um pedido de um cliente.

Como tal, a Figura 5.11 mostra o resultado disto para um cliente, que passa a ser servido apenas pelo Servidor1, embora o Servidor2 esteja ativo. Dado que o *reverse proxy* calcula sempre um *score* para cada pedido e o Servidor2 tem sempre um *score* muito alto (o que é mau neste contexto), então o Servidor1 é sempre o escolhido.



A terminal window titled 'root@Cliente1: /tmp/pycore.50434/Cliente1.conf' with standard window controls. The terminal shows the command 'cat index.html\*' being executed, resulting in ten identical lines of HTML code: '<h1> hello world CC TP2 Servidor1</h1>'. The prompt 'root@Cliente1:/tmp/pycore.50434/Cliente1.conf#' is visible at the bottom.

```
root@Cliente1: /tmp/pycore.50434/Cliente1.conf
root@Cliente1:/tmp/pycore.50434/Cliente1.conf# cat index.html*
<h1> hello world CC TP2 Servidor1</h1>
<h1> hello world CC TP2 Servidor1</h1>
<h1> hello world CC TP2 Servidor1</h1>
<h1> hello world CC TP2 Servidor1</h1>
<h1> hello world CC TP2 Servidor1</h1>
<h1> hello world CC TP2 Servidor1</h1>
<h1> hello world CC TP2 Servidor1</h1>
<h1> hello world CC TP2 Servidor1</h1>
<h1> hello world CC TP2 Servidor1</h1>
<h1> hello world CC TP2 Servidor1</h1>
root@Cliente1:/tmp/pycore.50434/Cliente1.conf#
```

Figura 5.11: Conteúdo dos ficheiros `index.html` lidos pelo cliente 10 vezes, após o Servidor2 reportar estatísticas com resultados inflacionados.

Concluí-se então que tanto o *Monitor UDP* como o *Reverse Proxy* funcionam corretamente, tal como especificado nos capítulos anteriores.

# Capítulo 6

## Conclusões e trabalho futuro

O trabalho consistiu em analisar, explorar e implementar uma abordagem muito comum em serviços *web*, um *Proxy TCP* reverso com monitorização proativa, o que consistiu em duas fases distintas.

Na fase 1 implementou-se o protocolo de monitorização da *pool* de servidores *back-end* em que o objetivo central consiste na recolha de informação atualizada via *UDP*. O servidor *proxy reverso* recolhe esta informação, que é obtida/calculada com base em parâmetros/dados dinâmicos, e mantém-na numa tabela atualizada (uma entrada por cada servidor *back-end*, identificado pelo respetivo endereço *IP*). Selecionaram-se os seguintes parâmetros dinâmicos a monitorizar para cada um dos *servidores back-end*:

- **Round-trip time (RTT):** estimado entre o *back-end* e o servidor *proxy*
- **Taxa de pacotes perdidos:** perdas de pacotes *UDP*
- **Conexões TCP:** número total de conexões *TCP* no momento da monitorização
- **Uso CPU:** percentagem de uso de *CPU* no momento de monitorização
- **Uso memória:** percentagem de uso memória no momento de monitorização

É essencial que a informação da tabela seja baseada em parâmetros dinâmicos e que esteja o mais atualizada possível, pois, só assim o servidor *proxy* poderá selecionar adequadamente o servidor *back-end* (fase 2).

O protocolo de comunicação definido permite o envio e receção de pacotes. Estes pacotes podem ser:

- **Registo:** Pacote que permite registar um agente com o servidor central
- **Probe request:** Pacote para pedir informações a um agente
- **Probe response:** Pacote de resposta com informações de monitorização

Para representar estes pacotes no transporte que é feito na rede, definiu-se o formato que estes teriam. Houve uma preocupação com o tamanho que estes pacotes poderiam ter e orientou-se o seu tamanho ao byte. No byte em que se encontra o tipo do pacote apenas 3 bits foram utilizados, havendo espaço reservado para futuras utilizações. Os restantes dados são específicos de cada pacote e em cada um deles, os bytes têm um significado associado.

Na fase 2 implementou-se a componente do servidor *proxy TCP* que fica à escuta na porta 80 e recebe conexões de clientes. O servidor *proxy*, para cada conexão de cliente estabelecida:

- **Seleciona um dos servidores *back-end* disponíveis:** Para tal, utiliza uma métrica que considera as medidas monitorizadas na fase 1. A escolha das medidas que intervêm no cálculo da métrica e, consequentemente, na escolha do servidor *back-end* foi uma decisão que se centrou em ter os parâmetros que melhor representam a disponibilidade do *back-end* de receber um pedido naquele momento. A escolha destas medidas foi um *tradeoff* entre a melhor forma de representar a disponibilidade do *back-end* e, também, não o ocupar computacionalmente de mais com a monitorização.

Outra decisão importante foi definir a métrica, ou seja, a fórmula que para cada *back-end*, considerando os parâmetros definidos, os traduz num único valor comparável entre os vários servidores. Compreendeu-se que os parâmetros que mais afetam a disponibilidade de um servidor são: RTT, número de conexões TCP e a percentagem de perdas e, por isso, deu-se um maior peso no cálculo a estas medidas. No entanto, quando a percentagem de CPU, percentagem de memória, ou a percentagem de perda ultrapassa ultrapassa os 70% estas medidas têm um maior impacto no desempenho de um servidor e, como tal, refletiu-se esta influência no cálculo da métrica de decisão.

- **Intermedeia a conexão TCP:** Entre o cliente e o servidor *back-end* selecionado. Assim, o servidor *proxy* faz a *ponte* entre os dois servindo apenas como um intermediário que reencaminha pedidos e respostas. Desta forma, este servidor fica menos sobrecarregado do que se tivesse de responder a todos os pedidos e faz apenas a gestão dos servidores *back-end* considerando medidas dinâmicas que lhe permitem tomar uma melhor decisão.

Como retrospectiva de alguns aspectos analisados ao longo do trabalho, identificam-se alguns pontos que poderiam sofrer uma investigação mais aprofundada, para melhor perceber a extensão e impacto das decisões tomadas:

- **Parâmetros a monitorizar**

Tal como foi descrito neste capítulo uma das decisões do trabalho foi selecionar parâmetros que melhor representem dinamicamente a disponibilidade de cada servidor *back-end* de responder a pedidos. No entanto, esta escolha podia ser cega e ter em conta, por exemplo, um algoritmo de *Round-Robin* que apenas faz uma distribuição equitativa das conexões que recebe pelos N servidores *back-end*. Tentou-se fazer melhor e selecionar parâmetros dinâmicos e representativos. Ainda assim, para se conseguir confirmar o conjunto dos melhores parâmetros seria necessário fazer estudos e testes para compreender realmente como e quanto estes parâmetros influenciam a capacidade de resposta de um servidor *back-end*. Podendo assim compreender se se chegou aos melhores parâmetros a utilizar ou se alguma destas medidas poderia ser, eventualmente, descartada ou se outra medida poderia ser incluída.

- **Métrica para selecionar um servidor *back-end***

O conjunto dos melhores parâmetros é utilizado na fórmula da métrica que permite traduzir todas as medidas num só valor para cada servidor *back-end*. A formulação desta métrica poderia considerar um peso equitativo para cada métrica, dando igual relevância a todas as medidas. No entanto, também aqui, se tentou fazer melhor considerando diferentes pesos e diferentes situações. Mesmo assim, esta fórmula poderia ser alvo de vários testes e estudos para se confirmar ou obter a melhor forma de calcular e seleccionar um dos servidores.

- **Cache no mecanismo de reencaminhamento**

No mecanismo de reencaminhamento do tráfego TCP, investigou-se a possibilidade de utilização de cache para resposta aos pedidos dos clientes. Contudo, nesta iteração, para manter uma aplicação genérica de proxy reverso TCP, acabou por se deixar este mecanismo de cache de lado. Alguns dos fatores que tornariam esta solução não genérica com a introdução do mecanismo de cache são:

- Perda de performance na paralelização de leitura e escrita entre cliente e servidor (seria necessário ler o pedido completo do cliente para verificar se o resultado estava em cache).
- Possibilidade de estar a alterar a lógica de resposta de conteúdo dos servidores de *back-end* com a cache. Não é garantido que dois pedidos iguais devolvam exactamente os mesmos dados (dados de sessão por exemplo em HTTP).