Comunicações por Computador
Relatório do Segundo Trabalho Prático

Bruno Pereira (A75135), Diogo Silva (A76407), and Maria Ana de Brito (A73580)

Mestrado Integrado em Engenharia Informática
Universidade do Minho, 2016/17

Abstract. Este relatório aborda o desenvolvimento da *Reverse Proxy* assim como todas as decisões tomadas e a sua justificação. Será ainda explicada a arquitetura do software e as suas funcionalidades. Por conseguinte, são ainda referidos pormenores que poderiam ter sido melhorados.

Keywords: proxy reversa, monitor udp, servidor tcp

1 Introdução

Este projeto convidou os alunos a desenvolver uma Proxy Reversa consistindo em duas fases:

- Inicialmente foi necessário desenvolver a Lógica de Monitorização assim como o Monitor UDP. A Lógica de Monitorização é responsável por efectuar pedidos probing e obter a consoante resposta do Monitor UDP. Assim, a Lógica de Monitorização mantém uma tabela sempre atualizada com a informação do Monitor UDP. Este, por sua vez, deve responder aos probings da Lógica de Monitorização assim como enviar periódicamente um probing sinalizando o estado do seu servidor TCP.
- Por fim, foi necessário terminar a implementação da Proxy Reversa. Esta consistiu no desenvolvimento de um novo módulo, a Lógica de Proxy responsável por aceitar pedidos dos vários clientes, escolher o melhor servidor TCP e intermediar essa conexão.

2 Arquitetura da solução implementada

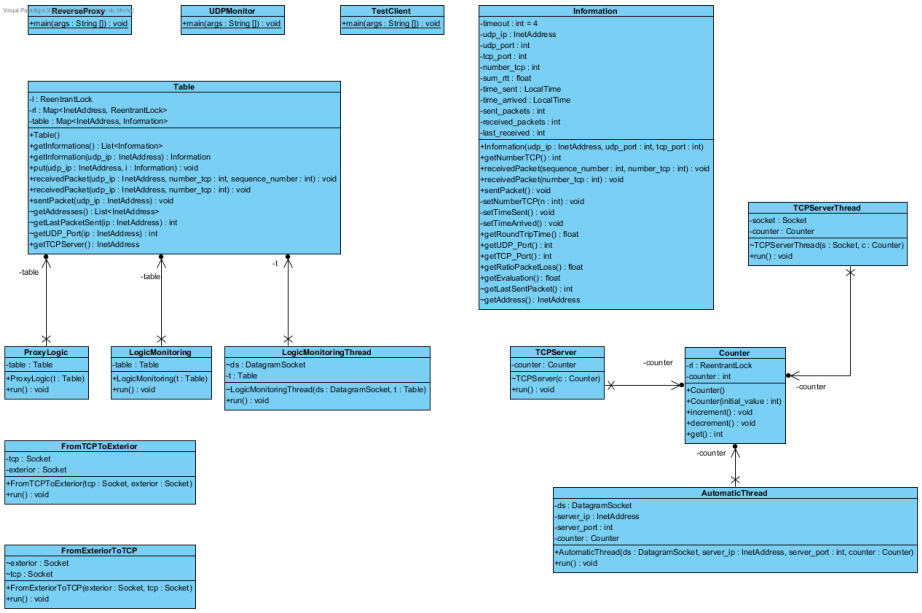

Fig. 1. Diagrama de Classes.

Como é possível observar, a *proxy* reversa apresenta três *main classes*: a *ReverseProxy*, *UDPMonitor* e *TestClient*. Estas terão o seu próprio contexto podendo correr em máquinas individualmente.

2.1 Proxy Reversa

A *proxy* reversa tem como responsabilidade iniciar a lógica de monitorização e a lógica de *proxy*. Além disso, a *ReverseProxy* cria um objeto do tipo *Table* que será passado como parâmetro à lógica de monitorização, *LogicMonitoring* assim como à lógica de *proxy*, *ProxyLogic*.

Por sua vez, a *ProxyLogic* ficará permanentemente ativa a aceitar pedidos vindos de vários clientes. Para cada pedido, determina o melhor servidor disponível criando assim um objeto do tipo *socket* com o qual o cliente se irá comunicar. Para tal ser possível, *ProxyLogic* encarrega-se de criar duas *threads*: uma que irá ler do cliente e escrever para o servidor TCP escolhido (*FromExteriorToTCP*) e outra que lê do servidor TCP e escreve para o cliente (*FromTCPToTExterior*). Estas permanecerão ativas até o cliente se desconectar.

Por fim, a *LogicMonitoring* encarrega-se de efetuar pedidos de *probing* aos monitores UDP enquanto armazena todos os dados provenientes numa tabela do tipo *Table*. Para tal ser possível, quando *LogicMonitoring* é inicializada, esta cria uma *thread* que percorrerá todos os monitores UDP ativos de 2 em 2 segundos e efetuará pedidos de *probing*. Por sua vez, a *LogicMonitoring* encarregar-se-à de receber as respostas dos monitores UDP. Estas respostas podem ser de três tipos: *init*, *automatic* ou *reply*. Assim, quando a resposta se trata do tipo:

- *init*: então trata-se de um novo monitor que acabou de se conectar pelo que uma nova entrada em *Table* será criada. Apesar de não ser necessária esta mensagem, é mais eficiente. Assim não é necessário verificar se o monitor UDP já se encontra em *Table* e assume-se que à partida será um novo monitor UDP.
- *reply*: então trata-se de uma resposta a um pedido de *probing* pelo que com esta mensagem vem nova informação sobre o estado do servidor TCP. Esta informação será adicionada à tabela.
- *automatic*: então é uma mensagem automatizada pelo monitor UDP pelo que será informação mais atualizada sobre o servidor TCP correspondente. Esta informação será adicionada à tabela tal como no caso de uma *reply*. Não obstante, o conteúdo será diferente tal como a natureza das mensagens.

2.2 Monitor UDP

O monitor UDP tem como responsabilidade monitorizar o seu servidor TCP assim como responder a *probing requests* por parte da lógica de monitorização. Assim quando *UDPMonitor* é criado, com ele também é inicializado o servidor TCP correspondente (a *thread TCPServer*) e uma *thread* responsável por enviar mensagens à lógica de monitorização sobre o número de conexões do seu servidor TCP de 1 em 1 segundo (*AutomaticThread*). Enquanto tudo isto funciona automaticamente, *UDPMonitor* agora apenas se encarrega de responder a *probing requests* sobre o número de conexões do seu servidor TCP.

2.3 Cliente

O cliente tem a possibilidade de fazer pedidos para descarregar um ficheiro localizado na máquina do servidor TCP. Assim, esse ficheiro deve ser dado como parâmetro tal como o endereço do servidor TCP. Por predefinição o conteúdo do ficheiro de texto é apresentado no ecrã, contudo, caso o utilizador deseje que *TestClient* guarde Todas as linhas de texto para um ficheiro basta apenas redirecionar para esse mesmo ficheiro.

3 Especificação do Protocolo de Monitorização

3.1 Primitivas de Comunicação

Para ser possível que o *UDPMonitor* comunique com o *LogicMonitoring* foram utilizados *DatagramSocket*, ou seja, pacotes UDP. Por outro lado, *TestClient* comunica-se com *ProxyLogic* através de *ServerSocket* ou seja, pacotes TCP sendo mais fiáveis pois numa conexão cliente-servidor nenhum pacote se pode perder.

3.2 Formato das Mensagens Protocolares (PDU)

Existem três tipos de *PDU* enviados pelo monitor UDP: um do tipo inicialização, outro do tipo resposta e por fim, um do tipo automático. Assim sendo, o tipo:

- de inicialização ou *init* apenas apresenta além do seu tipo, a porta onde o servidor TCP está alojado. Apesar de este pormenor não ser necessário, o grupo decidiu manter para atribuir alguma generalização ao programa. Ou seja, este *PDU* é do tipo: <tipo> <porta TCP>.
- de resposta ou *reply* apresenta além do tipo de mensagem, o número de conexões TCP do seu servidor e o número de sequência atual. Com número de sequência entende-se, o número do pacote enviado pela lógica de monitorização. Assim, o monitor UDP apenas repete o número de sequência que recebe para a lógica de monitorização perceber quantos pacotes foram perdidos, assim como o último entregue deduzindo quando a informação é mais recente. Ou seja, mesmo que a lógica de monitorização receba um pacote em atraso, além deste ser descartado, a informação sobre o número de conexões TCP não terá efeito pois trata-se de uma mensagem mais antiga. Logo, a mensagem é dada no seguinte formato: <tipo> <conexões TCP> <número de sequência>.
- automático ou *automatic* é enviado periodicamente à lógica de monitorização e apenas apresenta o número de conexões do seu servidor TCP: Ou seja, é no seguinte formato: <tipo> <conexões TCP>.

Por outro lado, a lógica de monitorização apresenta um PDU do tipo *probing request* sendo esse da forma: <número de sequência>.

3.3 Interações

Os vários *PDU* servem o propósito de dar a conhecer à lógica de monitorização o estado dos diversos servidores TCP. Assim, quem se encarrega de enviar o *PDU* do tipo de inicialização, resposta ou automático é o *UDPMonitor*. No caso do tipo automático, é a *thread* criada em *UDPMonitor*, a *AutomaticThread*. Todos estes *PDU* são enviados à *LogicMonitoring*. Por outro lado, a lógica de monitorização apenas envia *probing request* periodicamente a todos os monitores UDP.

Fora do contexto de monitorização, o *TestClient* comunica-se diretamente com *ProxyLogic*. Esta, por outro lado, irá dedicar um conjunto de *threads* a cada cliente. Estas *threads* serviram de intermediário entre o cliente e o servidor de *backend*.

4 Implementação

4.1 Detalhes e Parâmetros

Existem alguns detalhes que merecem ser salientados. Por exemplo, a função de escolha do melhor servidor TCP.

```
public float getEvaluation() {
    return (float) (getRatioPacketLoss() * 100 + getRoundTripTime() / 10000
    + getNumberTCP());
}
```

Aqui, o grupo tentou dar mais peso ao *round trip time* pois é um fator muito decisivo na qualidade da conexão. Assim, quanto maior a avaliação pior será a qualidade do servidor TCP. Por exemplo, para um rácio de pacotes perdidos de 50%, com 10 conexões TCP e 50ms de *round trip time* temos uma avaliação de 560. Por outro lado, para uma taxa de pacotes perdidos de 10% com um *round trip time* de 15ms e com 50 conexões TCP temos uma avaliação de aproximadamente 201. Assim, é possível observar o elevado desfasamento na avaliação consoante a qualidade do servidor TCP

Outro caso que pode ser discutido é o facto de a escolha do melhor servidor por parte da *Table* através do método *getTCPServer* obrigar a percorrer todos os servidores TCP. Outra forma de implementação poderia ser escolher o primeiro servidor TCP que respeitasse um critério. Por exemplo, ter uma pontuação inferior a 300. Mas isto, apesar de mais eficiente (na escolha) implicaria que o cliente teria de ficar à espera quando não houvesse nenhum servidor com uma pontuação inferior. E, além disso, sobrecarregaria sempre os servidores que se localizassem primeiro na tabela. Assim, o grupo decidiu que a tabela era toda percorrida, ou seja, todos os servidores TCP seriam testados e o melhor escolhido permitindo uma ocupação de recursos equilibrada.

4.2 Biblioteca de Funções

Aqui serão descritas as principais bibliotecas utilizadas.
java.net.InetAddress
java.time.LocalTime
java.net.DatagramPacket
java.net.DatagramSocket
java.util.concurrent.locks.ReentrantLock
java.io.BufferedReader
java.io.InputStreamReader
java.io.OutputStreamWriter
java.io.PrintWriter
java.net.Socket
java.net.ServerSocket;

5 Testes e Resultados

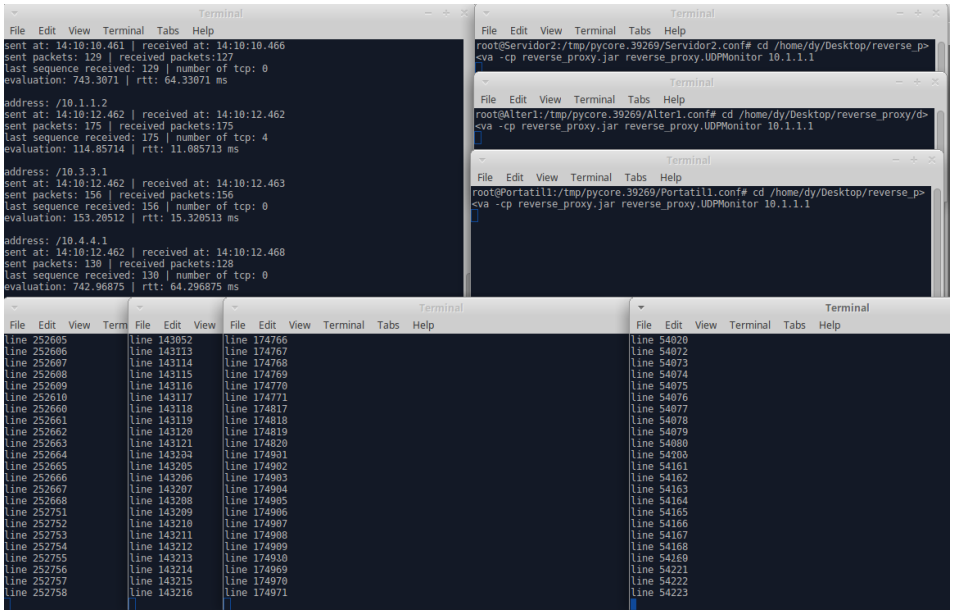

Fig. 2. Teste.

O teste representado apresenta três monitores UDP, e quatro clientes em máquinas diferentes. Neste teste foi utilizado um ficheiro com 500000 linhas. O mesmo ficheiro foi requerido pelos quatro clientes. Os números representados são as linhas lidas do ficheiro de texto, processadas pelo servidor. Como podemos observar, apesar do servidor TCP estar ocupado, não apresenta qualquer impacto na qualidade do mesmo. É possível visualizar o *print* do estado da tabela para os vários servidores TCP.

Pode-se verificar que o servidor com a pior avaliação encontra-se na máquina 10.4.4.1, isto deve-se ao facto de já ter ocorrido a perda de dois pacotes como se pode visualizar, e, além disso, esta máquina não se encontra na mesma rede que a *proxy* reversa e portanto o *round trip time* é superior. Aliás, a melhor máquina, como se pode visualizar, é a que se encontra precisamente na mesma rede que a *proxy* reversa. Além deste teste, o grupo fez outros com mais clientes e servidores e os resultados mantiveram-se os mesmos. Contudo, é possível verificar-se que a velocidade com que o ficheiro é descarregado torna-se cada vez menor sempre que um novo cliente se conecta.

6 Conclusão

De uma forma geral o grupo ficou satisfeito com o resultado obtido. O grupo sentiu inicialmente alguma dificuldade na utilização das primitivas de comunicação novas (isto é, pacotes datagrama) mas que com o desenvolvimento do programa foram notoriamente ultrapassadas. Apesar disso, atualmente o grupo considera já ter algum à vontade com a temática abordada através da realização deste projeto. A arquitetura do programa foi bem pensada e ponderada pelo grupo. Houve uma tentativa de generalização do mesmo para possível aperfeiçoamento no futuro (num cenário menos académico). Assim, o grupo considera ter atingido os objetivos fornecidos assim como demonstrar o conhecimento a ser avaliado.

References

1. Java™ Platform, Standard Edition 8 API Specification,
   url https://docs.oracle.com/javase/8/docs/api