Emulação de uma topologia real para teste de algoritmos distribuídos TCP

Afonso Bonito, Bruno Rafael, Tiago Campos
FEUP

***Resumo*—No desenvolvimento de algoritmos para sistemas distribuídos existe, tipicamente, disparidade entre o ambiente de desenvolvimento e o ambiente real onde o algoritmo irá ser utilizado. Esta disparidade deve-se sobretudo ao facto de o teste do algoritmo distribuído ser efetuado na máquina local ou mesmo na rede local, onde as condições da rede são muito melhores do que numa situação real. Neste trabalho pretende-se apresentar e descrever uma forma de solucionar esta disparidade, com base na implementação de uma API TCP que injete, internamente ao seu funcionamento, condições de uma rede real. Por fim, serão discutidos resultados de teste da API implementada, e destacadas as principais noções quanto ao estado da implementação final.**

***Palavras-chave*—sistemas distribuídos, TCP, emulação de rede, algoritmos distribuídos**

I. Introdução

A. Descrição do problema

A forma mais conveniente de testar um algoritmo distribuído, quase sempre implementado integralmente em *software*, é executar o algoritmo na máquina local, não existindo assim o problema associado à transferência do mesmo para o *hardware* onde irá ser executado.

Porém, o teste de algoritmo na máquina local, ou mesmo em diferentes máquinas da rede local, incorre em fortes limitações em termos da verificação do mesmo, dado que esta apresenta habitualmente características muito favoráveis em termos de atraso de rede, *jitter* e perda de pacotes. Por tal, o comportamento do algoritmo não é comprovado em condições de rede desfavoráveis, podendo falhar sob estas condições.

Procura-se então uma solução que permita combinar a conveniência do teste local com a robustez do teste sob condições reais, no contexto de algoritmos distribuídos implementados com recurso ao protocolo TCP.

B. Objetivos

Pretende-se desenvolver uma camada de *software* transparente, onde uma aplicação TCP pense que está a utilizar a API proveniente da linguagem de programação utilizada, mas onde na realidade está a utilizar uma API que exporta as mesmas funções, sendo nestas injetadas as não idealidades anteriormente referidas.

Parte-se de um trabalho anteriormente desenvolvido [1], onde se procurou resolver o mesmo problema para algoritmos UDP. O objetivo consiste na adaptação da solução UDP para uma solução TCP, também como na melhoria da solução UDP nos casos relevantes para ambos os protocolos. A solução mais apelativa será proceder à implementação TCP sobre a anterior implementação UDP, herdando assim as características de emulação de atrasos e roteamento físico. Como tal, pretende-se implementar o algoritmo TCP baseado em troca de mensagens UDP.

Para além disto é também de interesse desenvolver uma aplicação TCP que utilize a API desenvolvida e que permita explorar uma quantidade vasta de funções da API, de modo a comprovar o seu bom funcionamento.

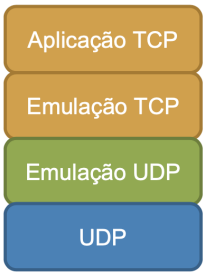

Figura 1. Pilha protocolar proposta

A figura 1 apresenta a pilha protocolar proposta, tendo nas duas camadas superiores a aplicação e a API a desenvolver, seguidas da emulação UDP desenvolvida no trabalho anterior [1]. Por fim, tem-se a API UDP fornecida pela linguagem de programação.

Os principais aspetos a implementar são:

- Máquina de estados TCP
- *Streaming* de dados
- Retransmissão de pacotes perdidos
- Reordenação de pacotes

De onde se destaca a máquina de estados TCP (figura 2) por ser a base de todo o funcionamento de um *socket*, definindo a mudança do estado da conexão do mesmo face à receção e envio de pacotes de controlo e sendo também a base das funções da API `accept()`, `connect()` e `close()`.

Destaca-se também a mudança de semântica de comunicação baseada em mensagens para comunicação baseada em *streaming* de dados, onde as mensagens podem ser segmentadas em pacotes que operam independentemente. Dada a natureza fiável do protocolo TCP, é também importante implementar deteção de pacotes perdidos através de *timeouts*, mecanismos de retransmissão dos mesmos, e reordenação de pacotes recebidos fora de ordem no recetor.

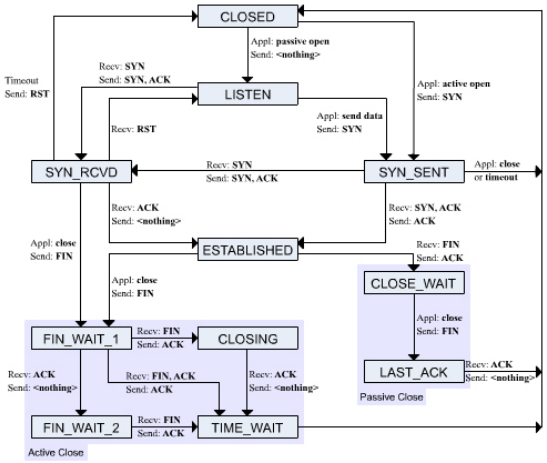

Figura 2. Máquina de estados TCP [2]

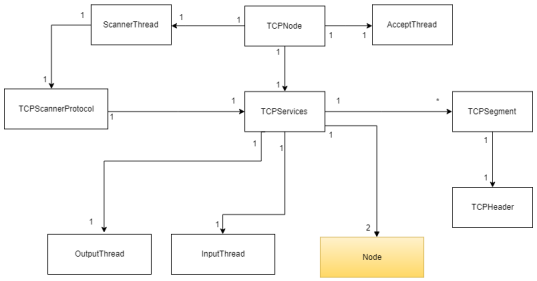

Figura 3. Diagrama UML de classes da solução

II. Implementação

A. Arquitetura

A figura 3 mostra a relação entre as diferentes classes implementadas na solução:

- **TCPNode**: Corresponde a um nó da rede TCP. É nesta classe que é executada a aplicação. Cada TCPNode tem um identificador, que diz respeito a um incremento a um *firstPort* fixo, de modo a garantir que cada nó da rede tenha um porto único no *localhost*.
- **TCPServices**: Implementa métodos semelhantes aos de Socket e ServerSocket da API de Java.
- **Node**: Classe já existente no projeto anterior [1], correspondente a um nó da rede UDP, implementando já o envio e receção de datagramas UDP. Foram efetuadas alterações, nomeadamente a emulação de perda de pacotes e a implementação de um temporizador na receção de datagramas, utilizado para os *timeouts* de retransmissão.
- **AcceptThread**: Corre o método `accept()` de TCPServices. Deste modo, cada nó está constantemente à espera de estabelecer uma ligação com um *initiator*.
- **ScannerThread**: Lê os *inputs* do utilizador na linha de comandos.
- **TCPScannerProtocol**: Classe onde são definidos os comandos que poderão ser executados pelo utilizador.
- **OutputThread**: Responsável por traduzir a semântica de *streaming* característica do TCP para a transmissão e eventual retransmissão de datagramas UDP.
- **InputThread**: Trata da receção, confirmação (envio de ACK's) e reordenação de datagramas UDP e na conversão dos mesmos para a semântica de *streaming*.
- **TCPHeader**: Permite à aplicação abstrair-se da estrutura do cabeçalho TCP.
- **TCPSegment**: Contém um TCPHeader e um vetor de *bytes*, correspondente aos dados.

B. Funcionalidades implementadas

1) TCPServices: Os principais métodos implementados na classe TCPServices são:

- `connect(int id)`: Utilizado para o estabelecimento de uma ligação por parte do nó que se pretende conectar. Diferente do método de Socket no facto de o argumento ser o identificador do TCPNode ao qual se pretende conectar, em vez do endereço IP e porto.
- `accept()`: Aceita a ligação de qualquer nó que se queira conectar.
- `getOutputStream()`: Retorna uma OutputStream na qual a aplicação pode escrever mensagens, sendo que estas serão lidas na InputStream do nó ao qual está conectada.
- `getInputStream()`: Retorna uma InputStream da qual a aplicação pode ler mensagens que tenham sido escritas na OutputStream do nó ao qual está conectada.
- `close()`: Termina uma conexão previamente estabelecida.

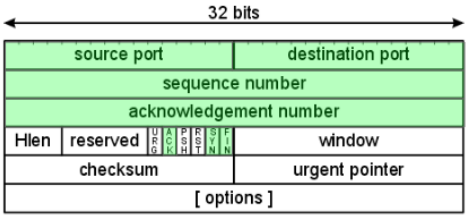

Figura 4. Cabeçalho TCP [3]

2) Cabeçalho TCP: A figura 4 representa o cabeçalho TCP. Realçados a verde estão os campos que foram implementados nesta solução. Note-se que *source port* e *destination port* dizem respeito aos portos TCP emulados (dados pelo identificador do TCPNode), e não aos portos utilizados pelos DatagramSocket's em que realmente são enviados e recebidos os dados. Os outros campos implementados (*sequence number*, *acknowledgement number*, e as *flags* ACK, SYN e FIN) são aqueles que permitem garantir as características desejadas para o protocolo TCP: o tipo de comunicação baseado em conexão, a fiabilidade e a ordenação.

3) Perda de pacotes: A emulação de perda de pacotes foi implementada na camada da Emulação UDP (figura 1). Sempre que um datagrama deve ser enviado, existe uma probabilidade de que este não seja, de facto. O valor dessa probabilidade é configurável pelo utilizador.

4) Timeouts: Também na Emulação UDP, foi implementado um temporizador que é despoletado por iniciativa da camada superior e ativa uma *flag*, ao fim de um determinado período de tempo sem receber um datagrama. Este temporizador é utilizado como *timeout*, sendo que, de um modo geral, é iniciado após uma transmissão, e caso este termine antes da receção da respetiva confirmação, procede-se à retransmissão.

C. Estabelecimento e fecho de conexão

Segundo o protocolo TCP, o estabelecimento e fecho de conexão é efetuado com um *3-way handshake*, representado na figura 5, estando também de acordo com a máquina de estados da figura 2.

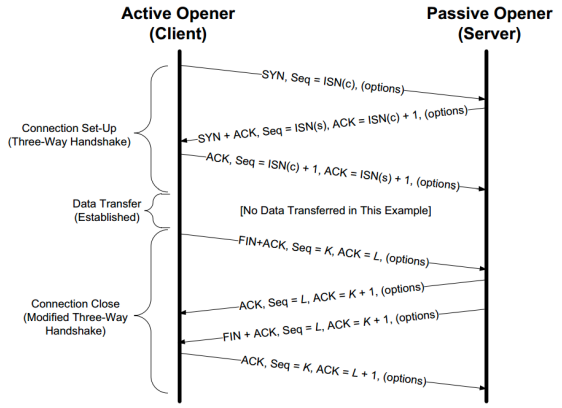

Figura 5. Estabelecimento e fecho de conexão TCP [4]

Em ambos os casos, existe um agente ativo que inicia o *handshake*, e um agente passivo que o aceita (ou rejeita), estando os pacotes de estabelecimento de conexão associados à *flag* SYN do cabeçalho TCP, e o fecho de conexão associado à *flag* FIN. A única diferença entre ambos os *handshakes* está no último pacote de fecho de conexão, que não tem um ACK associado porque o nó passivo já fechou a conexão. Nesse caso, o nó ativo assume que a conexão foi fechada quando ocorre um *timeout* após enviar o seu último pacote de FIN.

Caso um nó se tente conectar a um nó inativo, ou que já esteja conectado, os seus pacotes de SYN serão ignorados. Neste caso, este terá de desistir da tentativa de conexão. Isto é conseguido através de um contador que, após um determinado número de *timeouts*, termina a execução do `connect()`, sinalizando um erro no estabelecimento da ligação. Este mecanismo é também utilizado durante a conexão entre dois nós, servindo para a terminar no caso de um destes se desativar.

D. Escrita e leitura de dados

O principal problema relativamente à escrita e leitura de dados entre vários nós consistiu em emular o comportamento dos métodos `getOutputStream()` e `getInputStream()` da API de Java. A solução encontrada foi o uso de *Piped Streams*, que permitem a comunicação por *streaming* entre duas *threads* distintas.

Deste modo, quando a aplicação invoca o método `getOutputStream()`, na realidade é-lhe retornada uma PipedOutputStream, que está ligada a uma PipedInputStream da OutputThread. Esta *thread* divide os dados lidos em vários segmentos TCP, e encarrega-se da transmissão e eventual retransmissão dos mesmos, através de um objeto da classe Node. Estes segmentos são enviados para um Node de outra aplicação, instanciado na respetiva InputThread. Esta trata de reordenar os segmentos, enviar as respetivas confirmações e escrever os dados recebidos na PipedOutputStream, que está ligada à PipedInputStream da aplicação (retornada pelo método `getInputStream()`), podendo esta ler a mensagem através da função `read()`. Esta interação está representada na figura 6 e é totalmente transparente à aplicação.

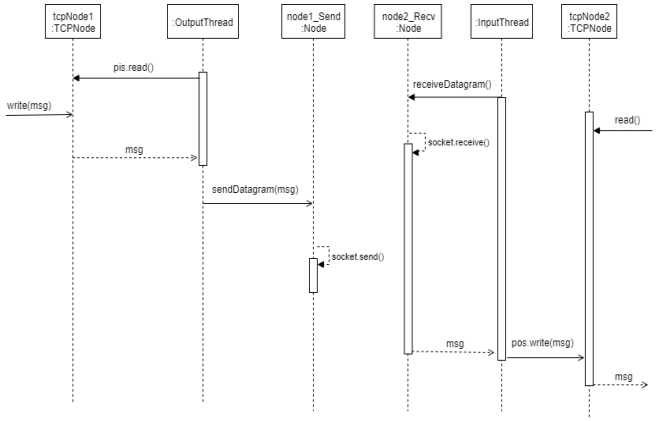

Figura 6. Diagrama UML de sequência da escrita e leitura de dados

A comunicação bidirecional é conseguida através do uso de duas instâncias de Node por cada TCPNode. Assim, um TCPNode com identificador i, instanciará TCPServices com um Node 2i, usado pela OutputThread e um Node 2i+1, usado pela InputThread. Deste modo, se este TCPNode se conectar a um outro, de identificador j, teremos as seguintes ligações: Node 2j $\leftrightarrow$ Node 2i+1 e Node 2j+1 $\leftrightarrow$ Node 2i, permitindo que a escrita e leitura entre cada par de nós TCP seja bidirecional.

E. Aplicação

A aplicação permite a instanciação individual de nós através da janela de comandos, podendo o utilizador especificar o seu identificador e a probabilidade de perda de pacotes. Qualquer nó, ao ser instanciado, aceita pedidos de conexão. Os comandos possíveis são:

- **connect i**: Tenta estabelecer uma conexão com o nó i.
- **write s**: Envia a linha ”s”ao nó ao qual está conectado.
- **read**: Lê uma linha da sua InputStream.
- **close**: Termina a conexão atual.

Além da aplicação principal, foi desenvolvido um programa para efetuar medições temporais no envio de dados. Este programa estabelece a ligação com outro nó, envia um determinado número de pacotes (com uma determinada probabilidade de perda dos mesmos), e mede o tempo que demora a receber todas as confirmações, tendo sido usado para validar a manipulabilidade das condições da rede emulada, e retirar conclusões face à mesma, sendo estas exploradas na secção seguinte.

III. Resultados

Considera-se que foi implementado com sucesso um protocolo funcional, emulando atrasos e perdas de pacotes implementadas na camada Emulação UDP da pilha protocolar proposta e que acontecem com naturalidade numa rede real, demonstrando, também, como os pacotes são retransmitidos em caso de falha.

Finda a implementação desta emulação do protocolo TCP, foram recolhidas estatísticas relativas à influência da percentagem de perda de pacotes e do tempo limite de espera para retransmissão (*timeout*) caso essa perda ocorra. Fazendo variar esses valores, para diferentes tamanhos dos dados enviados, extraíram-se os resultados apresentados nas figuras 7 e 8.

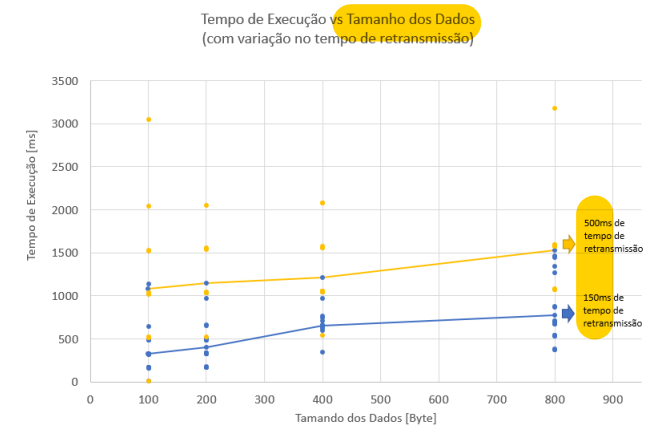

Figura 7. Influência do tempo para retransmissão

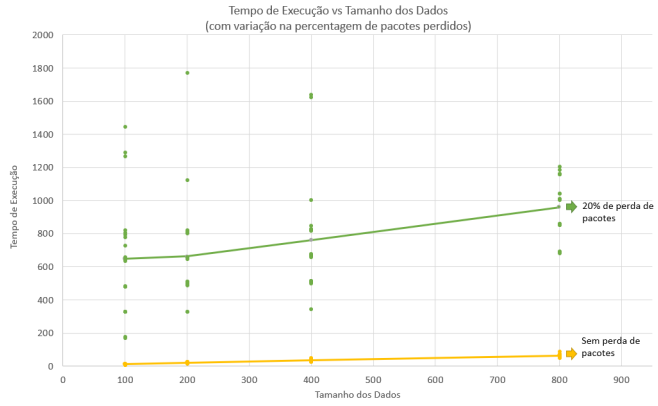

Figura 8. Influência da percentagem de perda de pacotes

Destes gráficos é possível concluir, primeiramente, que aumentando o tamanho dos dados enviados, o tempo de execução da aplicação também aumenta de forma aproximadamente linear, que pode ser explicado pela necessidade de enviar e receber mais pacotes, tanto de dados como de confirmação da sua receção.

Quanto ao tempo limite para retransmissão usado (figura 7) é possível concluir que demonstra também uma relação direta com o tempo de execução. Isso deve-se ao facto de um maior tempo de espera pela mensagem de confirmação implicar que a mensagem demora mais tempo a ser recebida no caso de a falha se verificar. O elevado *jitter* observado deve-se à elevada probabilidade de perda de pacotes (10%). É de notar que um tempo para retransmissão muito baixo também acarreta os seus problemas, na medida em que se houver atrasos de rede maiores do que o mesmo, continuará a haver envios com a mesma informação até que a mesma seja dada como recebida. Por esse motivo, os valores dos *timeouts* utilizados foram calculados em função dos atrasos de rede injetados na camada de Emulação UDP, de modo a minimizar o tempo de execução.

Quanto à percentagem de pacotes perdidos (figura 8), o aumento desta faz também aumentar o tempo de execução. Este é o resultado esperado devido à necessidade de retransmissão sempre que um pacote é perdido, fazendo aumentar esse tempo.

IV. Análise Crítica

A. Limitações e sugestões de melhoria

Existem alguns pontos onde a API desenvolvida pode ser melhorada, nomeadamente:

- Suporte para múltiplas conexões em simultâneo
- Implementação mais completa do cabeçalho TCP
- Utilização de um único Node por cada TCPNode
- Emular outros tipos de falhas de rede

Atualmente, o estabelecimento de múltiplas conexões em simultâneo pelo mesmo nó é dificultado pelo facto de que a cada instanciação da classe TCPServices esteja associada uma só ligação, sendo o TCPServices incapaz de aceitar mais ligações enquanto uma já está ativa, proveniente do facto de a classe TCPServices implementar a API de ambas as classes Socket e ServerSocket. A decomposição do TCPServices no equivalente a estas duas classes permite resolver o problema.

Como mencionado anteriormente, a implementação do cabeçalho TCP foi parcial e não suporta alguns campos como *window* e *checksum*. A implementação de SEQ assume que cada mensagem começa em SEQ = 0, que não desrespeita o protocolo TCP mas que poderá ser implementado de forma mais realista.

A utilização de dois Nodes por cada TCPNode provém da dificuldade em separar, nos buffers de receção, pacotes de controlo de pacotes de dados. Por exemplo, no caso de envio e receção em simultâneo, os pacotes de ACK em resposta às transmissões seriam confundidos com os pacotes de dados recebidos. Uma implementação com apenas um Node é possível, se for efetuado um processamento mais complexo dos pacotes recebidos.

Em redes reais é possível encontrar diferentes tipos de falhas com diferentes causas. Neste trabalho foi aproveitada a implementação previamente desenvolvida de atrasos de rede, sobre a qual se implementou também a possibilidade de perda de pacotes (com base numa função aleatória) de forma a testar a capacidade do protocolo de reenviar informação perdida. A

emulação de falhas tal como a perda de pacotes em *burst* ajudaria a perceber como o protocolo se comportaria perante estes casos.

B. Áreas não exploradas

Não foi explorado o comportamento do sistema com um número elevado de TCPNodes na rede, dado que as ligações entre nós TCP são ponto-a-ponto e neste momento não suportam conexões em simultâneo. Dado que o número de nós na rede UDP altera o roteamento físico emulado, pode ser interessante explorar o comportamento do sistema nestas condições.

Não foram exploradas as limitações da emulação em termos de utilização de recursos (CPU e RAM) da máquina utilizada, sendo que o trabalho anterior [1] já as explorava, embora a implementação da emulação TCP provavelmente incorra num *overhead* significativo.

Os testes desenvolvidos na camada de aplicação representam casos básicos onde um nó envia dados e o outro os recebe, sem preocupação com comunicação "*back-and-forth*" entre os nós (embora o envio de pacotes de ACK exiba este comportamento). O teste de um algoritmo distribuído complexo nesta rede apresentará resultados mais interessantes.

C. Distribuição de tarefas

O trabalho foi realizado em simultâneo por todos os elementos da equipa, de modo a promover a discussão de todos os aspetos a serem implementados. Não houve, portanto, uma discriminação explícita das tarefas a ser efetuadas por cada membro.

Tendo isso em conta, a distribuição do trabalho, em percentagem, por cada elemento da equipa é:

- Afonso Bonito: 33,(3)%
- Bruno Rafael: 33,(3)%
- Tiago Campos: 33,(3)%

Referências

[1] E. Rodrigues, D. Fonseca, R. Teixeira, "Emulação de uma topologia real para teste de sistemas distribuídos", SDIS, Janeiro de 2018.

[2] https://www.researchgate.net/figure/TCP-Finite-State-Machine_fig1_260186294, acedido 16 de Janeiro de 2019.

[3] http://telescript.denayer.wenk.be/~hcr/cn/idoceo/tcp_header.html, acedido 14 de Janeiro de 2019.

[4] https://notes.shichao.io/tcpv1/ch13/, acedido em 13 de Janeiro de 2019.