Implementação do Protocolo RDT

uma abordagem em C para a implementação da confiabilidade no UDP

M. R. B. Antonio[a], J. P. A. Evaristo[a], M. P. de Oliveira[a]

[a] *Univerisade Federal de São Paulo, campus São José dos Campos*

May, 2022

Abstract

Este relatório tem como objetivo documentar e explicar a metodologia utilizada por nosso grupo na implementação do protocolo RDT de bit alternante proposto por Kurose. O relatório irá conter trechos explicados de nossos códigos de cliente e servidor, tal como a parte experimental no CORE, onde emulamos uma topologia de rede e realizamos testes com diferentes configurações para constatar a perda de pacotes e a solução proposta pelo protocolo.

1. Servidor

No início da implementação do código do nosso servidor, começamos com procedimentos básicos que também serão usados na criação do cliente. Nós incluímos os cabeçalhos necessários para o uso da API de sockets, nesse caso a ¡sys/socket.h¿, ¡netinet/in.h¿ e ¡arpa/inet.h¿. Além de definirmos algumas variáveis globais úteis, também definimos a estrutura do nosso pacote, que conterá o número de sequência, o ack, o prorio conteúdo em si (payload), a soma de verificação e o tamanho do pacote.

1.1. Criando um pacote

Em nossa função para criar pacotes, antes de tudo, zeramos a área do pacote através da função memset, para evitar erros desnecessários. Após isso, passamos os parâmetros necessários definidos em nossa estrutura. O conteúdo do payload é copiado do input com os dados através da função memcpy. No final da função, o pacote preenchido é retornado.

1.2. Checksum

Essa pequena função gera números aleatórios como conteúdo dos pacotes, a partir de um número de requisição feita pelo cliente. A função sprintf converte esses inteiros para um char que será o conteúdo dos pacotes, e a memcpy copia esses valores para dentro do buffer.

1.3. Payload

Essa pequena função gera números aleatórios como conteúdo dos pacotes, a partir de um número de requisição feita pelo cliente. A função sprintf converte esses inteiros para um char que será o conteúdo dos pacotes, e a memcpy copia esses valores para dentro do buffer.

1.4. Número de sequência

A lógica para definição dos números de sequência são muito simples: como estamos trabalhando num sistema de bit alternante, caso o número de sequência atual for 0, o próximo será 1 e vice versa.

1.5. Enviando o pacote

No início da nossa função de envio, definimos uma tentativa de timer para o caso de perda de requisição. Caso o servidor não consiga enviar a requisição ou a requisição se perca, após o estouro do timeout a conexão é encerrada. Essa funcionalidade do nosso código ainda não está 100% funcional, ela poderá e deverá ser otimizada para uma tentativa de reenvio da requisição. Após isso, são declaradas variáveis que serão úteis na execução do código. Nossa lógica de envio é baseada em dois laços de repetição. O maior, mostrado acima, se baseia na seguinte lógica: enquanto ainda tiver requisições a serem processadas (definido pelo cliente) e o timeout não tiver estourado, faça tal ação. No caso, uma mensagem vazia é gerada e a função para gerar o payload é chamada. o checksum desse conteúdo é calculado, e o pacote é criado. O número de sequência e o ack são passados através da variável nextseqnum, que é iniciada globalmente com o valor de 0. A segunda parte do código é composta pelo segundo laço de repetição da função de envio. Ela possui a seguinte lógica: Caso algo der errado, o pacote será reenviado para o cliente até todos os parâmetros estarem satisfeitos. São checados os seguintes casos: Caso a função sendto falhe ao enviar o pacote, o pacote é reenviado. Esse erro é a nível operacional e é bem raro de acontecer. Caso o ack não seja recebido, obviamente o pacote deve ser reenviado. Caso o ack recebido esteja errado, também deve ser reenviado. No final desse laço, o

timeout é incrementado e, caso estoure o máximo tolerado, a aplicação para. Caso o pacote esteja correto, o programa sai do laço menor e entra no maior até que as requisições sejam atendidas. No final, os parâmetros para o próximo pacote são criados. Note que o timeout é zerado e o i é incrementado no começo do código, tendo em em vista que a próxima requisição só é processada quando ocorre a passagem pelo laço maior.

1.6. Recebendo a requisição do clinte

A nossa função de recebimento configura um timeout de tolerância de recebimento de requisição. A requisição é recebida através da função recvfrom. O número de requisições é obtido através do buffer. Caso o recebimento esteja de acordo e o buffer não esteja vazio, a rdtsend é chamada. Através de testes, O servidor constatou que recebeu dados de um socket de datagrama com um número de bytes maior que zero, pois a função recvfrom retornou valor diferente de -1, mas o valor do parâmetro buffer não foi alterado nesse processo, indicando um erro no recebimento da requisição. Como printamos o valor da requisição no momento do seu envio, o valor do parâmetro buffer deveria ser alterado caso a função recvfrom retornasse algo diferente de -1. Dessa forma, caso o timeout estoure e o servidor não consiga receber nenhuma requisição mesmo com o cliente tentando enviá-la, a conexão é fechada.

1.7. Configurando a conexão na main

Em nossa main, usamos diversos mecanismos para usar a API de sockets. primeiramente, criamos um socket utilizando a função de mesmo nome. Isso retorna um identificador de socket que pode ser usado para se comunicar com o cliente. Após isso, nós configuramos o endereço do servidor através dos parâmetros presentes na estrutura sockaddrin, em que explicitamos a família, o endereço e a porta. Em seguida, vinculamos o socket a uma porta específica usando a função bind(). Isso permitirá que o servidor receba conexões de clientes na porta especificada. Caso os procedimentos ocorram de acordo, o servidor envia o pacote através da função rdtrecv que foi criada anteriormente.

2. Cliente

A função sendack, como o nome já remete, tem a função de enviar o ack de recebimento do pacote do cliente para o servidor. Possui funcionamento semelhante ao rdtsend ao utilizar a função sendto, entretanto, tem o diferencial de usar a função sprintf para armazenar o número de sequência dentro do buffer definido no começo da função, buffer esse que é enviado para o servidor.

2.1. Recebendo o Pacote

No início da nossa função de recebimento de pacotes, é definido um timeout de tolerância de recebimento. A lógica é semelhante à apresentada no servidor, dois laços de repetição. Enquanto houver requisições feitas, o laço maior continuará, e enquanto o pacote recebido for o errado, o laço menor continuará. Na função são checados casos de erro, como: Pacote não foi recebido, checksum está errado e número de sequência está errado. Caso algum desses casos ocorra, o ack enviado é o ack do último pacote recebido corretamente. Caso o pacote atual esteja correto, o ack enviado é o ack desse próprio pacote.

2.2. Envio de Requisição

A função rdtsend faz o papel de enviar as requisições iniciais para o servidor. A requisição é enviada através da sendto, e caso o envio tenha sucesso, logo em seguida a rdtrecv é chamada para fazer o recebimento do pacote e seguir o procedimento descrito acima. A lógica é semelhante a função rdtrecv do servidor, porém os papéis estão trocados.

2.3. Configurando a conexão na main

A main do código do cliente é semelhante a main do código do servidor. O socket é criado e os endereços são configurados. A diferença é que não é necessário o uso da função bind e o endereço do servidor e o número de requisições são passados como argumentos.

3. Testes e Discussão

Para a execução dos testes, estaremos rodando os códigos numa topologia básica criada no emulador CORE:

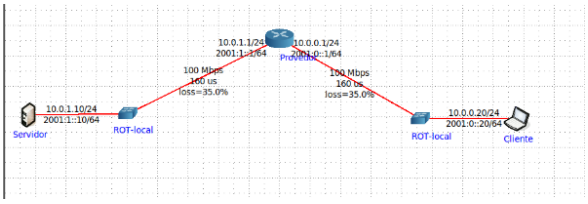

Figure 1: Topologia de rede criada no emulador CORE.

O uso dos nossos programas são feitos da seguinte forma: ./servidor porta ./cliente ipservidor porta requisições Nos testes, iremos utilizar as seguintes abordagens:

- Caso 1: baixa taxa de perda (5.0%) por parte do cliente e poucos pacotes a serem enviados (5);

- Caso 2: baixa taxa de perda (5.0%) por parte do cliente e muitos pacotes a serem enviados (50);
- Caso 3: alta taxa de perda (35%) por parte do cliente e poucos pacotes a serem enviados (5);
- Caso 4: baixa taxa de perda (35%) por parte do cliente e muitos pacotes a serem enviados (50);

Nos testes mais avançados, colocamos taxa de perda em ambos os lados do cliente e servidor:

- Caso 5: baixa taxa de perda (5.0%) por ambas as partes e poucos pacotes a serem enviados (5);
- Caso 6: baixa taxa de perda (5.0%) por ambas as partes e muitos pacotes a serem enviados (50);
- Caso 7: baixa taxa de perda (35%) por ambas as partes e alguns pacotes a serem enviados (15);

Em todas as entradas foi escolhido um valor de 160 ms de delay para a realização dos testes.

3.1. Caso 1

O que podemos perceber nesse teste: o servidor teve que reenviar o segundo pacote uma vez, já que o ack não foi recebido ao realizar o envio do pacote pela primeira vez. Se checarmos o recebimento do cliente, notamos que o referido pacote já tinha sido recebido e, além disso, não acusou nenhum erro de falha no pacote, então o pacote estava correto. O que pode ter acontecido: o ack que o cliente enviou se perdeu, ou não foi recebido por estouro do timeout. Após receber o mesmo pacote novamente, o cliente acusa o erro de número de sequência incorreta e o ack do último pacote é enviado, possibilitando que as demais requisições fossem recebidas, e o processo ocorreu normalmente, o esperado para uma taxa de perda baixa.

3.2. Caso 2

O caso 2 é bem parecido com o caso 1. Durante as 50 requisições, apenas 2 pacotes tiveram que ser reenviados, por conta do ack não ser recebido pelo servidor. É praticamente a mesma taxa de perda configurada se analisarmos com o total de pacotes.

```
root@n3:/tmp/pycore.40001/n3.conf# ./servidor 8080
Aguardando conexão...
Enviando pacotes...

Enviado pacote 0
 Conteúdo: 3
 Ack enviado: 0
  0 timeouts
Ack recebido: 0

Enviado pacote 1
 Conteúdo: 1
 Ack enviado: 1
  0 timeouts
O ack não foi recebido. Reenviando último pacote...

Enviado pacote 1
 Conteúdo: 1
 Ack enviado: 1
  1 timeouts
Ack recebido: 1

Enviado pacote 0
 Conteúdo: 3
 Ack enviado: 0
  0 timeouts
Ack recebido: 0

Enviado pacote 1
 Conteúdo: 1
 Ack enviado: 1
  0 timeouts
Ack recebido: 1

Enviado pacote 0
 Conteúdo: 2
 Ack enviado: 0
  0 timeouts
Ack recebido: 0
```

Figure 2: Caso 1: Servidor enviando 5 pacotes.

3.3. Caso 3

Durante esse teste, com a alta taxa de perda foi possível perceber uma limitação da nossa implementação: caso o último pacote chegue correto para o cliente, mas o ack se perca a caminho do servidor, o servidor tentará reenviar o pacote até que ocorra o estouro do timeout. Isso acontece pois, para o cliente, todos os pacotes necessários já chegaram de maneira correta, enquanto o servidor ainda não tem essa confirmação pela falta do ack. Como constatado no CORE, o cliente recebeu corretamente todos os 5 pacotes. Entretanto, o ack do último pacote se perdeu e acabou gerando a problemática que foi citada acima.

3.4. Caso 4

Esse teste evidencia a ineficiência do protocolo RDT 3.0, como foi discutido em sala de aula. Para uma taxa de 35% de perda, foram necessários vinte e três (23) reenvios de pacote por perda de ack. Para o servidor, esse é o pior caso possível, visto que o cliente já recebeu o pacote correto mas o servidor ainda trata de realizar o reenvio por não ter a confirmação necessária de que os pacotes chegaram corretamente.

```
 Conteúdo: 2
 Ack enviado: 0
  0 timeouts
0 ack não foi recebido. Reenviando último pacote...

Enviado pacote 0
 Conteúdo: 2
 Ack enviado: 0
  1 timeouts
0 ack não foi recebido. Reenviando último pacote...

Enviado pacote 0
 Conteúdo: 2
 Ack enviado: 0
  2 timeouts
0 ack não foi recebido. Reenviando último pacote...

Enviado pacote 0
 Conteúdo: 2
 Ack enviado: 0
  3 timeouts
0 ack não foi recebido. Reenviando último pacote...

Enviado pacote 0
 Conteúdo: 2
 Ack enviado: 0
  4 timeouts
0 ack não foi recebido. Reenviando último pacote...
```

Figure 3: Caso 3: Servidor reenviando o pacote até o estouro do timeout.

3.5. Caso 5

Esse caso veio somente para mostrar que não é porque tem uma taxa de perda configurada que necessariamente ocorrerá alguma perda, ainda mais numa amostra pequena como essa. Mesmo com os 2 lados com enlaces de 5% configurados, não houve nenhuma perda, todos os 5 pacotes foram recebidos sem a necessidade de nenhum reenvio.

3.6. Caso 6

Aqui, por termos uma amostragem maior, ocorrem erros e perdas mais interessantes para a análise. No caso acima, ocorre uma perda que ainda não tinha ocorrido: o servidor envia o pacote corretamente, porém o pacote se perde e o servidor tem de enviar o pacote novamente. Podemos notar essa diferença com os erros anteriores a partir do momento que não há pacote duplicados. O pacote com conteúdo 15, apesar de ter sido enviado 2 vezes pelo servidor, é recebido apenas 1 vez pelo cliente, devido ao primeiro envio ter sido perdido. Vale ressaltar também que a maioria dos erros encontrados são de perda e erro de sequência. Como estamos num ambiente de emulação, é muito improvável que aconteça um erro a nível operacional como as funções de envio e recebimento, ou um erro no checksum.

3.7. Caso 7

Esse teste final demonstra a dificuldade de se estabelecer uma conexão cliente-servidor quando os dois lados estão com uma alta taxa de perda de pacotes. Como evidenciado no emulador CORE, o servidor chegou a reenviar o mesmo pacote 6

```
Enviado pacote 0
 Conteúdo: 15
 Ack enviado: 0
  0 timeouts
0 ack não foi recebido. Reenviando último pacote...

Enviado pacote 0
 Conteúdo: 15
 Ack enviado: 0
  1 timeouts
Ack recebido: 0

Enviado pacote 1
 Conteúdo: 29
 Ack enviado: 1
  0 timeouts
Ack recebido: 1
```

Figure 4: Caso 6: Servidor enviando os pacotes.

```
Pacotes recebidos: 2

Pacote recebido com sucesso
O contéudo do pacote é: 15
O número de sequência do pacote é: 0, e o ack é: 0
Número de sequência Correta.
nextack = 1

Pacotes recebidos: 3

Pacote recebido com sucesso
O contéudo do pacote é: 29
O número de sequência do pacote é: 1, e o ack é: 1
Número de sequência Correta.
nextack = 0
```

Figure 5: Caso 6: Cliente recebendo os pacotes

vezes, devido às sucessivas perdas, tanto do pacote como do ack. A perda bilateral afetou tanto o desempenho da troca de requisições que o servidor estourou o timeout máximo, tentando reenviar o pacote 8 vezes, e fechou a conexão. As requisições ainda não tinham sido terminadas, então o cliente ficou com a conexão aberta também até o seu timeout estourar, configurando uma situação catastrófica a nível cliente-servidor, e evidenciando uma possível limitação de nossa implementação.

3.8. Perda no primeiro pacote

Essa situação específica ocorre quando a requisição que o cliente envia para o servidor é perdida. A criação e o envio dos pacotes não consegue ser estabelecida, ocasionando no erro e encerramento do programa após o estouro do timeout.

4. Conclusão

O protocolo RDT proposto por Kurose na literatura é interessante para uma abordagem didática, entretanto apresenta um desempenho não satisfatório, já que há casos de difícil tratamento e muitas perdas sequenciais, o que impacta negativamente na latência do sistema de rede, como visto em aula.