Trabalho Prático 2 – Redes de Computadores

Vitor Rodarte Ricoy - 2019007112

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

1. Dificuldades

1.1. Escrita de código em C

Essa dificuldade se manteve nesse trabalho com um nível mais alto do que no primeiro trabalho prático. Isso aconteceu devido à necessidade de uma estrutura de dados mais complexa para implementar listas de pokémons, levando à criação de uma biblioteca de lista encadeada por mim. Essa dificuldade aconteceu principalmente devido ao processo de construção dessa estrutura, além de outros pontos do código que são naturalmente complexos em C, como a manipulação de strings e a sintaxe mais limitada da linguagem.

1.2. Entendimento do Enunciado

Acredito que esse trabalho teve uma complexidade muito mais alta do que o trabalho anterior e, por isso, tive dificuldade em entender corretamente o enunciado do trabalho. Isso aconteceu, ao meu ver, devido à complexidade do jogo definido, além da falta de detalhamento de comportamentos específicos de operações. A maior parte das dúvidas que tive tirei no fórum do trabalho, porém a espera por respostas levou a um tempo mais longo para desenvolver o trabalho. Além disso, mesmo após esclarecidas as dúvidas fiquei com a sensação de que não sei se implementei exatamente o que foi pedido.

1.3. Legibilidade e Organização do Código

Essa dificuldade também aconteceu no último trabalho. Ela ocorreu, em grande parte, pela dificuldade de compreender o código em C no geral, já que ele costuma ser bem verboso e pouco compreensível. Além disso, a utilização da biblioteca pthreads para implementar os servidores colaborou ainda mais para essa falta de legibilidade e organização. Apesar dessa dificuldade, no fim acredito que o código ficou com uma legibilidade e organização razoável.

2. Imprevistos

Tive diversos imprevistos no desenvolvimento. A começar pela implementação em si que comecei antes da aula de dúvidas utilizando quatro executáveis e tive que alterar devido à mudança no enunciado. Depois disso, desenvolvi o código e fiquei com algumas dúvidas, que esclareci no fórum, o que levou a mais mudanças consideráveis no código. Também tive diversos imprevistos de projeto da solução que levaram a mudanças radicais no código, consumindo um tempo considerável. Um exemplo dessas mudanças foi o uso de threads e o uso de uma lista encadeada para guardar as listas de atacantes, permitindo um número variável de elementos e operações mais simples.

3. Decisões de Implementação

A primeira decisão tomada foi definir uma thread para cada servidor, sendo que cada servidor possui uma porta e um socket próprio. Também foi decidido que os pokémons atacantes e os pokémons defensores são gerados aleatoriamente. Os atacantes são gerados a cada turno, sendo que existe 25% de probabilidade de que cada um dos 3 tipos de pokémons sejam gerados e 25% de probabilidade de não gerar um novo pokémon atacante. Já os pokémons defensores são gerados quando é recebida a mensagem de início de jogo, para permitir que a operação getdefenders seja chamada mais de uma vez sem alterar o estado do jogo. Além disso, os pokémons defensores possuem necessariamente posições distintas e, por isso, podem existir no máximo 5 vezes o número de coluna deles.

Devido à decisão de definir uma thread para cada servidor, utilizei um mutex para garantir a sincronização entre essas threads. Ao invés de identificar seções críticas no código e criar um mutex para cada uma delas, optei por criar um mutex que garante exclusão mútua de execução entre as threads, ou seja, apenas um servidor executa em um dado instante de tempo. Para alcançar esse objetivo o mutex é bloqueado sempre que uma thread sai de um recvfrom e é liberado sempre que a thread entra nessa operação.

Também decidi criar um novo comando identificado como restart. Esse comando é usado para sincronizar a necessidade de reiniciar o jogo, causada por um gameover com status igual a 1, entre todos

os servidores.

Uma decisão central tomada foi que toda mensagem enviada deve possuir uma resposta. Isso implica em diversos efeitos, sendo o primeiro deles a necessidade de verificar o timeout apenas do recvfrom, já que ele é suficiente para indicar perda, pois um sendto sempre será acompanhado de um recvfrom. Outra consequência dessa decisão é a necessidade de definir retornos de controle para algumas operações que não possuíam retorno. Essas operações são a quit, que retorna ‘exited’ e a restart que retorna ‘restarted’.

Outra decisão tomada foi a de implementar a comunicação via sockets entre dois ou mais servidores para duas operações: quit e gameover com status igual a 1. Na operação quit um servidor envia aos outros três o comando para que eles se encerrem. No gameover o servidor que gerou o erro envia aos outros servidores uma operação restart. Todas as comunicações via sockets entre servidores possui o mesmo algoritmo de retransmissão do cliente.

Outras decisões de funcionamento são: comandos start no meio de um jogo são tratados como mensagem inválida, levando a um gameover com status 1 e, consequentemente, um reinício do jogo; sempre que ocorre um gameover com status igual a 1 o jogo é reiniciado; mensagens que causam o reinício do jogo, ao serem retransmitidas, levam ao mesmo efeito prático, porém as estatísticas do primeiro gameover são perdidas na retransmissão; especificamente para o comando quit foi criada uma regra que tenta a retransmissão apenas 5 vezes, caso a transmissão falhe todas as vezes é considerado que o servidor foi encerrado; se o cliente receber pelo teclado uma mensagem de shot com um id de pokémon atacante invalido, o cliente imprime um erro sem enviar nada ao servidor, já que não é possível decidir qual o servidor que receberá a mensagem; foi considerada apenas a primeira palavra do comando para definir se uma mensagem é válida, ou seja, se o conteúdo da operação estiver errado ela será executada como se fosse válida; e as portas dos servidores são sequencias, ou seja, são x, x+1, x+2 e x+3, sendo x o número de porta informado pelo usuário.

4. Soluções Adotadas

Algumas observações gerais são que considerei que nenhuma mensagem ultrapassa 1024 bytes, portanto a declaração de arrays de caracteres seguem esse tamanho no geral. Também foi criada uma biblioteca de funções compartilhadas entre o servidor e o cliente, que possui uma função para encerrar o programa com erro e uma mensagem, uma função que retorna quantos ataques um pokémon precisa sofrer para ser destruído e também uma função que envia uma mensagem para um servidor e recebe uma resposta, considerando a possibilidade de retransmissão e um timeout de um segundos. Vale notar que o mecanismo de retransmissão foi implementado nessa função.

4.1. Cliente

O cliente implementa o protocolo pedido para execução do jogo. Inicialmente ele envia um start para todos os servidores, e em seguida um getdefenders para um servidor aleatório. Vale notar que para toda execução de um jogo, um servidor aleatório é escolhido como o servidor que receberá as operações que são enviadas a apenas um servidor. Após isso o cliente lê comandos do teclado e os envia ao servidores.

Caso o comando lido seja um getturns, o cliente envia uma cópia dele para cada servidor e imprime as respostas separadamente. Além disso, o cliente também salva a lista de pokémons atacantes retornada, que é usada para validar e direcionar os comandos shot ao servidor correto. Ao receber um comando shot o cliente valida se o id do pokémon atacante é válido e usa a linha desse pokémon para enviar a requisição shot para o servidor correto. Se o comando não for nenhum desses dois, ele é enviado para o servidor aleatório escolhido no início do jogo.

O cliente também trata respostas de gameover. Caso elas tenham status 1, o cliente inicia um novo jogo, enviando novamente um start para cada servidor, reiniciando o fluxo descrito anteriormente. Caso elas tenham o status 0, o cliente gera um comando quit, enviado para o servidor escolhido no início do jogo, encerrando os quatro servidores.

Finalmente, vale notar que o cliente possui 2 sockets, um IPv4 e outro IPv6, e 4 dados de conexões, um para cada servidor. Essas informações são usadas para conectar corretamente nos servidores.

4.2. Servidor

Os servidores funcionam com base em threads. Primeiramente são criadas quatro threads, acompanhadas de quatro sockets, um para cada servidor. Os servidores funcionam de forma bem similar entre si, variando apenas o socket, os dados de conexão e um identificador da linha que o servidor controla.

Após a inicialização das quatro threads, os servidores esperam pelo início de um jogo, respondendo uma mensagem de start e retornando gameover com status 1 para qualquer outra mensagem recebida. Eles também reiniciam todos os parâmetros de controle usados para representar o estado do jogo em um dado momento, assim como suas estatísticas. Após receber um start o servidor entra em um estado que ele espera receber ou outro start ou uma outra mensagem. Caso ele receba outro start ele o trata como retransmissão, caso ele receba outra mensagem ele a processa. Vale notar que após receber uma outra mensagem, qualquer start subsequente é tratado como mensagem inválida, o que leva ao reinício do jogo.

As mensagens regulares são tratadas conforme a especificação e as decisões citadas acima. Todas elas possuem uma estrutura parecida e consistem em alterar alguma informação de estado do jogo e retornar uma mensagem de resposta. Ao gerar um gameover com status 1, ou receber uma mensagem de reset, o servidor volta a esperar o start, reiniciando as variáveis de controle novamente. Ao receber um quit o servidor o replica e encerra sua execução.

Vale notar que no início do programa são instanciadas as listas, inicializado o mutex, inicializados os sockets e criadas as threads. No fim do programa, a execução das threads são sincronizadas, as listas limpas, os sockets fechados e o mutex é destruído.

4.3. Lista Encadeada

Foi implementada uma lista duplamente encadeada circular para salvar a lista de pokémons atacantes de cada servidor, além de salvar, no cliente, a lista de pokémons atacantes ativos em cada rodada. Essa lista possui as operações de inicializar a lista, de adicionar um novo elemento, de remover um elemento pelo seu id, de buscar um elemento por seu id, de buscar um elemento por sua posição, de atualizar um elemento da lista por seu id, de converter a lista encadeada para um vetor de C e de apagar todos os elementos da lista.

4.4. Mecanismo de Retransmissão

O mecanismo de retransmissão implementado é baseado em timeout e na decisão tomada, citada anteriormente, de que toda mensagem enviada deve ser respondida. Com isso, implementei a lógica do recvfrom dar um timeout caso ele espere por mais de 1 segundo, que é um tempo mais do que necessário para executar qualquer operação do servidor. Caso ocorra o timeout o sendto é executado novamente, seguido do recvfrom e essa lógica se repete. A única exceção desse funcionamento é o quit, como citado anteriormente, que permite no máximo 5 timeouts, devido à possibilidade do servidor ter sido encerrado.

O tempo de 1 segundo escolhido funcionou bem para todos os testes que fiz, pois ele não deu timeouts “falsos”, ou seja, as operações sempre levaram menos de 1 segundo para serem executadas e também 1 segundo é um tempo razoável para realizar as retransmissões caso se tenha um número até por volta de 10 retransmissões, o que acontece apenas em cenários muito extremos.

4.5. Cliente para Testes

Além do cliente descrito acima, que foi implementado no arquivo client.c, também implementei um cliente que foi utilizado exclusivamente para testes, em um arquivo test_client.c. Esse cliente segue os mesmos princípios do cliente principal, mas ao invés de ler as operações do teclado ele gera os comandos automaticamente, seguindo um algoritmo básico de um defensor atirar sempre no primeiro pokémon de ataque válido que ele encontrar. Vale notar que, devido à natureza desse cliente, cenários de erros, como um gameover prematuro ou uma resposta de ataque com erro não ocorrem nele, já que ele envia apenas operações válidas para o servidor.

5. Testes do Funcionamento

Para o teste não ocupar muito espaço no relatório testei o cenário de 2 colunas, 3 rodadas e 2 defensores. Vale notar que testei o código para diversos valores, entretanto a saída do programa para valores maiores ficam muito grandes. Para executar o teste usei um cliente implementado por mim no arquivo test_client.c, que joga o jogo de forma automatizada, para facilitar a interação com o servidor. Segue a saída da execução do teste:

```
> start
< game started: path 1
> start
< game started: path 2
> start
< game started: path 3
> start
< game started: path 4
> getdefenders
< defender [[1, 2], [1, 0]]
> getturn 0
< base 1
turn 0
fixedLocation 1

turn 0
fixedLocation 2

< base 2
turn 0
fixedLocation 1
1 Mewtwo 0

turn 0
fixedLocation 2

< base 3
turn 0
fixedLocation 1
2 Zubat 0

turn 0
fixedLocation 2

< base 4
turn 0
fixedLocation 1
3 Zubat 0

turn 0
fixedLocation 2

> getturn 1
< base 1
turn 1
fixedLocation 1

turn 1
fixedLocation 2

< base 2
turn 1
fixedLocation 1

turn 1
fixedLocation 2
1 Mewtwo 0

< base 3
turn 1
fixedLocation 1
4 Zubat 0

turn 1
fixedLocation 2
2 Zubat 0

< base 4
turn 1
fixedLocation 1
5 Zubat 0

turn 1
fixedLocation 2
3 Zubat 0

> shot 2 3 1
< shotresp 2 3 1 0
> getturn 2
< base 1
turn 2
fixedLocation 1
6 Mewtwo 0

turn 2
fixedLocation 2

< base 2
turn 2
fixedLocation 1
7 Zubat 0

turn 2
fixedLocation 2

< base 3
turn 2
fixedLocation 1
8 Mewtwo 0

turn 2
fixedLocation 2
4 Zubat 0

< base 4
turn 2
fixedLocation 1

turn 2
fixedLocation 2
5 Zubat 0

> shot 2 3 4
< shotresp 2 3 4 0
> getturn 3
< base 1
turn 3
fixedLocation 1
9 Zubat 0

turn 3
fixedLocation 2
6 Mewtwo 0

< base 2
turn 3
fixedLocation 1
10 Lugia 0

turn 3
fixedLocation 2
7 Zubat 0

< base 3
turn 3
fixedLocation 1
11 Zubat 0

turn 3
fixedLocation 2
8 Mewtwo 0

< base 4
turn 3
fixedLocation 1

turn 3
fixedLocation 2

> shot 2 3 7
< shotresp 2 3 7 0
> shot 2 1 6
< shotresp 2 1 6 0
> getturn 4
< gameover 0 1 3 0
> quit
< exited
```

Além disso, também executei o mesmo teste definindo perdas de pacotes, por meio do comando tc do linux. Esse teste foi útil para ver o funcionamento da retransmissão, assim como o atraso na execução gerado por ela. Para uma perda de 0% o programa executou em 0 segundos, como visto acima. Para 1% ele executou em 1 segundo, para 5% em 3 segundos, para 10% em 8 segundos, para 20% em 10 segundos e para 40% de perda o programa executou em 51 segundos. Com isso, podemos ver que a perda impacta fortemente na performance, mas está dentro de um limite tolerável, já que um cenário com 40% de perda é altamente improvável.