Programação com Threads

Lucas B. Lopes de Moura, Vinicius S. Mendes

[1]Instituto de Ciências Exatas e Informatica
Pontifícia Universidade Catolica de Minas Gerais (PUCMG)
Caixa Postal 1.686 – 30535.901 – Belo Horizonte – MG – Brasil

[2]Departamento de Ciência da Computacao
Belo Horizonte, BR.

lblmoura@sga.pucminas.br, vinicius.mendes.1248940@sga.pucminas.br

1. Video e Repositorio

Criamos um vídeo para demonstrar o funcionamento do programa, explicar como o código foi realizado e os principais pontos de entendimento do sistema. O vídeo pode ser encontrado no seguinte link: https://www.youtube.com/watch?v=eV8tSaqTmbE e o repositório com o código fonte pode ser encontrado em https://github.com/lucasbottrel/RPG

2. Codigo Fonte

O Código fonte do programa é dividido em três arquivos principais, responsáveis principalmente por realizar a comunicação entre cliente e servidor e realizar a lógica de funcionamento do jogo de RPG.

2.1. Servidor

A classe servidor é responsável por abrir a comunicação com os potenciais clientes através de uma porta, receber mensagens dos clientes (de todas as threads inicializadas, distinguindo-as) e reencaminhar as informações recebidas para todos os outros clientes logados no servidor. Como porta padrão, foi utilizada a porta 4000. Através do método **clientConnectionLoop()**, o servidor espera a todo tempo por novas conexões, criando um **ClientSocket** (explicado na próxima seção) para cada cliente conectado. Com a conexão estabelecida, o servidor cria uma ***Thread*** para aquele cliente, a fim de evitar que um cliente atrapalhe o outro ao realizar chamadas ao servidor.

```
1   /**
2      * Inicia o loop infinito de espera por conexoes dos clientes
3      *
4      * @throws IOException quando um erro de I/O (Input/Output) ocorrer
5      */
6     private void clientConnectionLoop() throws IOException {
7         try {
8             while (true) {
9                 System.out.println("\n(?) Aguardando conexao de novo
    jogador ...");
10
11                final ClientSocket clientSocket;
12                try {
13                    clientSocket = new ClientSocket(serverSocket.accept
    ());
```

```
                    System.out.println("\n(!) Jogador " + clientSocket.
    getRemoteSocketAddress() + " conectado!");
                }catch(SocketException e){
                    System.err.println("Erro ao aceitar conexao do
    cliente. O servidor possivelmente esta sobrecarregado:");
                    System.err.println(e.getMessage());
                    continue;
                }

                /*
                Cria uma nova Thread para permitir que o servidor nao
    fique bloqueado enquanto atende as requisicoes de um unico cliente.
                */
                try {
                    new Thread(() -> clientMessageLoop(clientSocket)).
    start();
                    clientSocketList.add(clientSocket);
                }catch(OutOfMemoryError ex){
                    System.err.println(
                            "Nao foi possivel criar thread para novo
    cliente. O servidor possivelmente esta sobrecarregdo. Conexao sera
    fechada: ");
                    System.err.println(ex.getMessage());
                    clientSocket.close();
                }
            }
        } finally{
            /*Se sair do laco de repeticao por algum erro, exibe uma
    mensagem
            indicando que o servidor finalizou e fecha o socket do
    servidor.*/
            stop();
        }
    }

```

Com o acesso feito ao servidor, para cada ***Thread*** é iniciado tambem um *loop* que informa a conexão realizada ao servidor para aquele cliente, assim como registra as mensagens enviadas por ele. Por fim, o servidor encaminha todas ações realizadas por outros clientes ao servidor, utilizando o método **sendMessageToAll()**.

```
/**
 * Encaminha uma mensagem recebida de um determinado cliente
 * para todos os outros clientes conectados.
 *
 * @param sender cliente que enviou a mensagem
 * @param msg mensagem recebida.
 */
private void sendMsgToAll(final ClientSocket sender, final String msg)
    {

    final Iterator<ClientSocket> iterator = clientSocketList.iterator()
    ;
    /*Percorre a lista usando o iterator enquanto existir um proxima
    elemento (hasNext)
    para processar, ou seja, enquanto nao percorrer a lista inteira.*/
```

```
    while (iterator.hasNext()) {
        //Obtem o elemento atual da lista para ser processado.
        final ClientSocket client = iterator.next();
        /*Verifica se o elemento atual da lista (cliente) nao e o cliente que enviou a mensagem.
        Se nao for, encaminha a mensagem pra tal cliente.*/

            if(client.sendMsg(msg)){}
            else iterator.remove();
    }
}
```

2.2. Client Socket

O arquivo ***Client Socket*** realiza a comunicação entre cliente e servidor. Na verdade se trata de um intermediador entre essa conversa, tornando possível a conexão Cliente-Servidor e Servidor-Cliente, assim como é responsável por fechar o *Socket* do cliente quando há desconexão.

```
/**
* Envia uma mensagem e nao espera por uma resposta.
* @param msg mensagem a ser enviada
* @return true se o socket ainda estava aberto e a mensagem foi enviada, false caso contrario
*/
public boolean sendMsg(String msg) {
   out.println(msg);

   return !out.checkError();
}

/**
* Obtem uma mensagem de resposta.
* @return a mensagem obtida ou null se ocorreu erro ao obter a resposta
*/
public String getMessage() {
   try {
       return in.readLine();

   } catch (IOException e) {
       return null;
   }
}

/**
* Fecha a conexao do socket e os objetos usados para enviar e receber mensagens.
*/
@Override
public void close() {
   try {
       in.close();
       out.close();
       socket.close();

   } catch(IOException e){
```

```
        System.err.println("Erro ao fechar socket: " + e.getMessage());
    }
}
```

2.3. Cliente

A classe ***Cliente*** é responsável por realizar toda a lógica do RPG no contexto do cliente, incluindo selecionar os personagens, usar habilidades e conduzir a batalha entre jogador e o inimigo. Nessa classe, cabe destacar que existe um objeto da classe **ClientSocket** para cada cliente instanciado no programa principal.

```
    /**
     * Objeto que armazena alguns dados do cliente (como o login)
     */
    private ClientSocket clientSocket;

    /**
     * Executa a aplicacao cliente.
     * Pode-se executar quantas instancias desta classe desejar.
     * Isto permite ter varios clientes conectados e interagindo
     * por meio do servidor.
     *
     * @param args parametros de linha de comando (nao usados para esta
    aplicacao)
     */
    public static void main(String[] args) {
        try {
            Cliente client = new Cliente();
            client.start();
        } catch (IOException e) {
            System.out.println("Erro ao conectar ao servidor: " + e.
    getMessage());
        }
    }
```

Cada cliente possui uma ***Thread*** associada, de tal forma que o servidor consiga-o identificar de forma separada e trata-lo individualmente no servidor e, consequentemente, no contexto do RPG.

```
    /**
     * Inicia o cliente, conectando ao servidor e entrando no loop de
    envio e recebimento de mensagens.
     * @throws IOException quando um erro de I/O (Input/Output) ocorrer
     */
    private void start() throws IOException {
        final Socket socket = new Socket(SERVER_ADDRESS, Servidor.PORT);
        clientSocket = new ClientSocket(socket);
        System.out.println("\n(!) Jogador conectado ao servidor no
    endereCo " + SERVER_ADDRESS + " e porta " + Servidor.PORT + ".\n");

        login();

        new Thread(this).start();
        messageLoop();
    }
```

O código roda em dois *loops*. A função **messageLoop()** é basicamente o RPG, com as opções de ação de cada jogador, de acordo com o personagem selecionado. Nessa função, dependendo das escolhas do jogador, são enviados mensagens para o servidor (basicamente as escolhas refletem no dano causado ao BOSS, magias e efeitos individuais). Já o loop da função **run()** roda para receber as mensagens vindas do servidor e orienta o jogador sobre o que está acontecendo no contexto geral do jogo e das ações de outros jogadores.

```
1
2  /**
3       * Aguarda mensagens do servidor enquanto o socket nao for fechado
       e o cliente nao receber uma mensagem null.
4       */
5      @Override
6      public void run() {
7          String msg;
8          while((msg = clientSocket.getMessage())!=null) {
9              System.out.println(msg);
10
11             if(msg.equals("Obrigado por Jogar! Voce sera DESCONECTADO."
       )) System.exit(0);
12         }
13     }
```

3. Protocolo de Transmissão

Com base na captura realizada no WireShark ficou constatado que o protocolo de transmissão de dados utilizado na reprodução do vídeo no *YouTube* foi o **QUIC** (*Quick UDP Internet Protocol*), visto que o navegador utilizado para busca e reprodução do vídeo foi o Google Chrome, no sistema operacional Windows 11.

3.1. QUIC

Atualmente, temos alguns protocolos de transmissão de multimídias, de modo que cada tipo de protocolo é utilizado dependendo de determinada circunstância, isto é, o protocolo a ser selecionado pode variar de acordo com o tipo de conexão que se deseja estabelecer. Nesse contexto, surgiu o **QUIC**, desenvolvido pela Google, normalmente utilizado para *streamings* de mídia, jogos e serviços VoIP (Voz sobre IP), com objetivo de pegar os melhores conceitos de TCP (*Transmission Control Protocol*) e UDP (*User Datagram Protocol*), como a segurança que está atrelada ao TCP e a velocidade que está vinculada ao UDP, ainda que as conexões estejam atreladas ao **TLS** (*Transport Layer Security*), permitindo a comunicação criptografada entre um domínio de site e um navegador.

Desse modo, realizando uma análise superficial, é possível vermos que no protocolo TCP a transmissão de dados ocorre por meio da conexão em três vias ou *three-handshake* que faz com que tenhamos a confirmação de envio de dados entre o remetente e o destinatário, permitindo uma maior segurança no envio dos pacotes e garantindo que todos vão chegar ao destino. Porém esse método se torna pouco ágil devido a confirmação de envio dos pacotes.

Nesse contexto, o QUIC aproveita o conceito de segurança no envio e o navegador pode começar a trocar pacotes com o servidor através da conexão inicial realizada,

de forma a otimizar o tempo gasto de conexão. Além disso, o QUIC implementa novos recursos em sua arquitetura que vão auxiliar na consolidação de um protocolo mais eficaz na transmissão de multimídias, ajustando o controle de congestionamento e a retransmissão automática, com o objetivo de torná-lo mais confiável que o UDP, que por sua vez manda os pacotes ao destinatário sem confirmação na entrega de pacotes, porém de forma rápida.

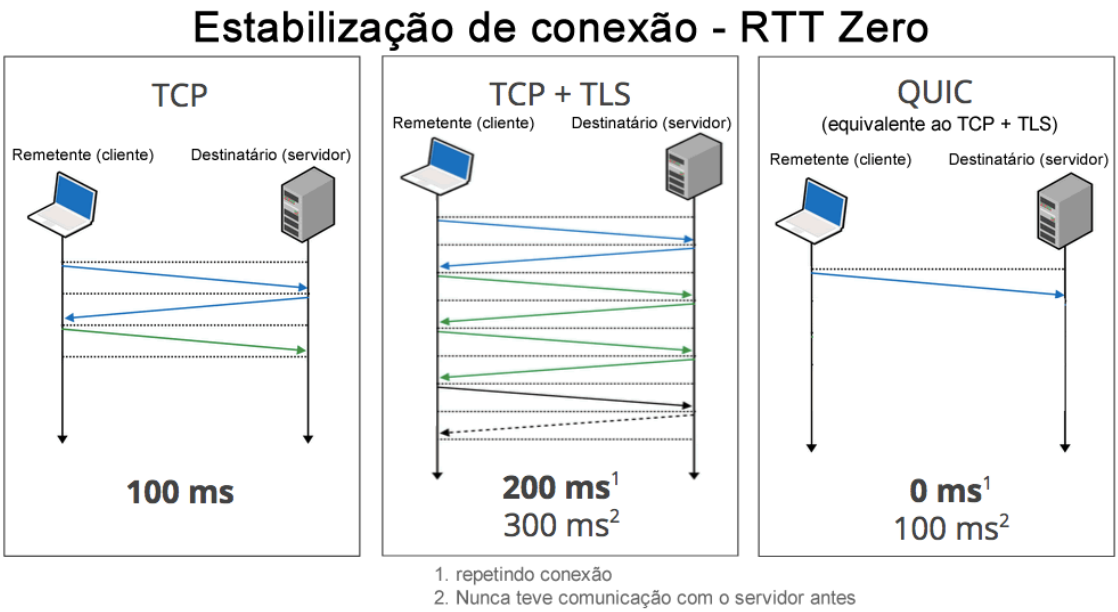

Figure 1. Funcionamento do QUIC. Fonte: www.dicasparacomputador.com

Assim como o HTTP/2, um avanço que foi impulsionado pelo **SPDY** (protocolo de rede deprecado desenvolvido principalmente pela Google para transporte de dados pela internet), o HTTP/3 é desenvolvido novamente sobre essas conquistas e possibilitará ao QUIC ser mais viável e poderoso.

Embora o HTTP/2 tenha nos dado multiplexação, ele é limitado pelo TCP. É possível usar uma conexão única de TCP para múltiplas correntes em conjunto para transferir dados, mas quando uma delas sofre a perda de um pacote, a conexão inteira (e todas as suas correntes) é mantida refém, até que TCP retransmita o pacote perdido.

O QUIC não é limitado por isso. Com QUIC desenvolvido sobre um protocolo UDP sem conexão, o conceito de conexão não carrega as limitações de TCP e falhas que ocorram em uma corrente não têm influência sobre o restante.

Com um foco nas correntes UDP, QUIC alcança multiplexação sem rodar sobre uma conexão TCP e desenvolve sua conexão em um nível mais alto. Novas correntes dentro de conexões QUIC não são forçadas a aguardar até que as outras sejam finalizadas e se beneficiam ao eliminar a sobrecarga do handshake do TCP, o que reduz a latência.

Embora o QUIC acabe com os recursos de confiabilidade do TCP, ele compensa com a camada UDP, oferecendo retransmissão de pacotes. Entre Google Chrome, YouTube, Gmail, pesquisas no Google e outros serviços, o Google foi capaz de implementar o QUIC em uma boa parte da Internet. Os engenheiros do Google alegam que, em

2017, 7% do tráfego na Internet já era conduzido por QUIC.

Por fim, de acordo com a empresa Google, o protocolo de transporte para internet QUIC apresenta bons resultados em comparação ao TCP e UDP, visto que segundo seus testes de desempenho o QUIC mostra 10% de redução no tempo de carregamento de páginas e, para o caso de de transmissão de vídeos houve redução de 30% nos *rebuffers*. Dessa forma, com base no exposto acima, é possível vermos que a captura do WireShark realizada pega pacotes com o protocolo QUIC, pelo fato do navegador utilizado (Google Chrome) utilizar o qual, uma vez que pertence ao mesmo desenvolvedor e apresenta melhor desempenho na transmissão de pacotes mutimídia.

Referências

TANENBAUM, A. S. **Redes de Computadores**. 5ª Ed., Pearson, 2011.

HOSTMIDIA. **O que é HTTP/3 e QUIC?**.Disponível em: https:www.hostmidia.com.br/blog/o-que-e-http3-e-quic. Acesso em: 12 setembro 2022.

KINSTA. **O Que é HTTP/3 – A Verdade Sobre o Novo Protocolo Baseado em UDP**. 6, Janeiro de 2022. Disponível em ¡https://kinsta.com/pt/blog/http3/¿. Acesso em: 12, Setembro de 2022.

Como e por que chegamos ao QUIC e ao HTTP/3. **Apiki**. 07, Agosto e 2020. Disponível em https://blog.apiki.com/como-e-por-que-chegamos-ao-quic-e-ao-http-3/. Acesso em: 12, Setembro de 2022.