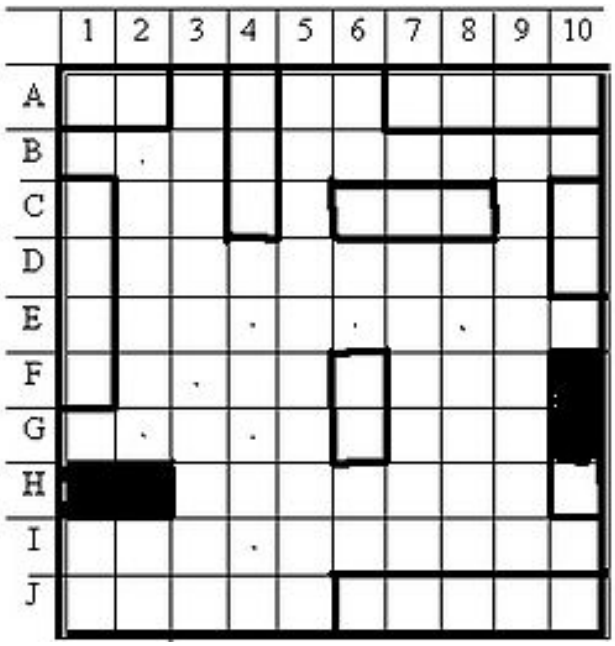
**Universidade de Itaúna**
**Curso de Ciência da Computação**
**Disciplina: Redes de Computadores I**
**Professor: Felipe Cunha**

# Trabalho Prático

**Trabalho em dupla**
**Valor do trabalho:** 20pts
**Entrega:** 30/11/2022

## 1 Batalha Naval

O principal objetivo do trabalho é praticar a programação com a biblioteca socket e usando o protocolo TCP. O trabalho consiste em desenvolver o jogo batalha naval para o paradigma cliente-servidor. Esta versão pode funcionar com apenas um cliente conectado. Você deverá criar um programa cliente e outro programa servidor. Batalha naval é um jogo de tabuleiro no qual o objetivo é afundar a frota de navios do inimigo. Inicialmente é definido um tabuleiro (matriz) de 10 x 10. Os quadrados do tabuleiro são identificados na horizontal por números e na vertical por letras. Os tabuleiros não são visíveis entre os jogadores. Os tipos de navios são, ocupando quadrados adjacentes na horizontal ou vertical: porta-aviões (cinco quadrados), navios-tanque (quatro quadrados), contratorpedeiros (três quadrados) e submarinos (dois quadrados). O número de navios são: 1, 2, 3 e 4 respectivamente. O jogo consiste em escolher um espaço do tabuleiro oponente e tentar acertar um dos navios da frota. O navio afunda quando todos quadrados forem adivinhados pelo oponente, quem perder todos os navios primeiro perde o jogo. Os programas irão funcionar utilizando o TCP.




O trabalho prático consistirá na implementação de dois programas, o cliente e o servidor. Ambos vão receber parâmetros de entrada como explicado a seguir:

```
cliente <ip/nome> <porta>

servidor <porta>
```

O primeiro programa, o cliente, irá conectar no servidor definido pelo endereço IP e porta passados por parâmetro. Já o servidor irá executar um serviço, que irá tratar uma conexão por vez. A porta a ser escutada é dada pelo parâmetro único do programa.

O cliente irá se conectar ao servidor e o jogo irá se iniciar. O cliente deve posicionar sua frota. Isso pode ser feito lendo um arquivo de entrada. O servidor pode posicionar sua frota escolhendo

aleatoriamente a posição inicial e a orientação (vertical, horizontal) dos navios. O cliente irá enviar a posição escolhida para o tiro, que pode ser lida do teclado. O servidor irá responder se acertou ou não e também a posição do seu tiro, e assim por diante até o jogo terminar. O servidor pode implementar a escolha da posição do tiro aleatoriamente, e se acertar, ele deve escolher um quadrado vizinho. No cliente, se pressionado a letra P, ele deve imprimir (em ASCII) seu tabuleiro e o tabuleiro com o que ele já aprendeu do servidor.

A documentação deve ser entregue em pdf junto ao código. Implementações modularizadas deverão mencionar quais funções são implementadas em cada módulo ou classe. A documentação deve conter os seguintes itens:

- Sumário do problema a ser tratado.
- Uma descrição sucinta dos algoritmos e dos tipos abstratos de dados, das principais funções, e procedimentos e as decisões de implementação.
- Decisões de implementação que porventura estejam omissos na especificação.
- Testes, mostrando que o programa está funcionando de acordo com a especificação, seguidos da sua análise. Print screens mostrando o correto funcionando do cliente e servidor e exemplos de testes executados.
- Conclusão e referências bibliográficas.

## 2 Sistema de Preços

O principal objetivo do trabalho é praticar a programação com a biblioteca Socket e utilizando o protocolo UCP. O trabalho consiste em desenvolver um sistema para enviar o preço de vários postos de combustíveis e requisitar o preço mais barato em uma dada região. Você deverá criar um programa cliente e outro programa servidor. Os programas irão funcionar utilizando o UDP.

O trabalho prático consistirá na implementação de dois programas, o cliente e o servidor. Ambos vão receber parâmetros de entrada como explicado a seguir:

```
cliente <ip/nome> <porta>

servidor <porta>
```

O primeiro programa, o cliente, irá conectar no servidor definido pelo endereço IP e porta passados por parâmetro. Já o servidor irá executar um serviço, que irá tratar uma comunicação por vez, mas que poderá comunicar com vários clientes. A porta a ser escutada é dada pelo parâmetro único do programa.

O cliente poderá enviar dois tipos de mensagens ao servidor: dados (D) e pesquisa (P). Como o UDP não garante a entrega de mensagens, o cliente deve implementar pelo menos uma retransmissão, caso a mensagem não seja recebida a primeira vez, para tentar garantir a entrega de mensagens. O servidor deve confirmar a recepção de mensagens. As mensagens de dados começam com a letra D seguida de um inteiro identificador da mensagem, um inteiro que indica o tipo de combustível ( 0 - diesel, 1 - álcool, 2- gasolina), um inteiro com o preço x 1000 (ex: R$3,299 fica 3299 ) e as coordenadas do posto de combustível (latitude e longitude). O servidor deverá confirmar a recepção da mensagem e adicionar a informação em um arquivo. Vários clientes podem enviar dados e mesmo com o término da comunicação entre um cliente e um servidor os dados enviados devem ser salvos no arquivo para consultas de outros clientes. As mensagens de pesquisa começam com a letra P seguida de um inteiro identificador da mensagem, um inteiro que indica o tipo de combustível ( 0 - diesel, 1 - álcool, 2- gasolina), um inteiro com o raio de busca e as coordenadas do centro de busca (latitude e longitude). O servidor deverá responder com o menor preço para aquele combustível para postos de combustível que estejam no centro de busca mais raio de busca. Para verificar o correto funcionamento dos programas, o cliente e o servidor devem imprimir na tela o conteúdo das mensagens que eles receberem.

A documentação deve ser entregue em pdf junto ao código. Implementações modularizadas deverão mencionar quais funções são implementadas em cada módulo ou classe. A documentação deve conter os seguintes itens:

- Sumário do problema a ser tratado.
- Uma descrição sucinta dos algoritmos e dos tipos abstratos de dados, das principais funções, e procedimentos e as decisões de implementação.
- Decisões de implementação que porventura estejam omissos na especificação.
- Testes, mostrando que o programa está funcionando de acordo com a especificação, seguidos da sua análise. Print screens mostrando o correto funcionando do cliente e servidor e exemplos de testes executados.
- Conclusão e referências bibliográficas.

## Tutorial apresentado pelo site: devmedia.com[1]

## Estrutura básica de uma aplicação de rede

Uma aplicação que utiliza sockets normalmente é composta por uma parte servidora e diversos clientes. Um cliente solicita determinado serviço ao servidor, o servidor processa a solicitação e devolve a informação ao cliente (ver Figura 1). Muitos serviços podem ser disponibilizados numa mesma máquina, sendo então diferenciados não só pelo endereço IP, mas também por um número de porta. Porém, o mais comum é termos uma máquina dedicada oferecendo apenas um ou dois serviços, evitando assim a concorrência.

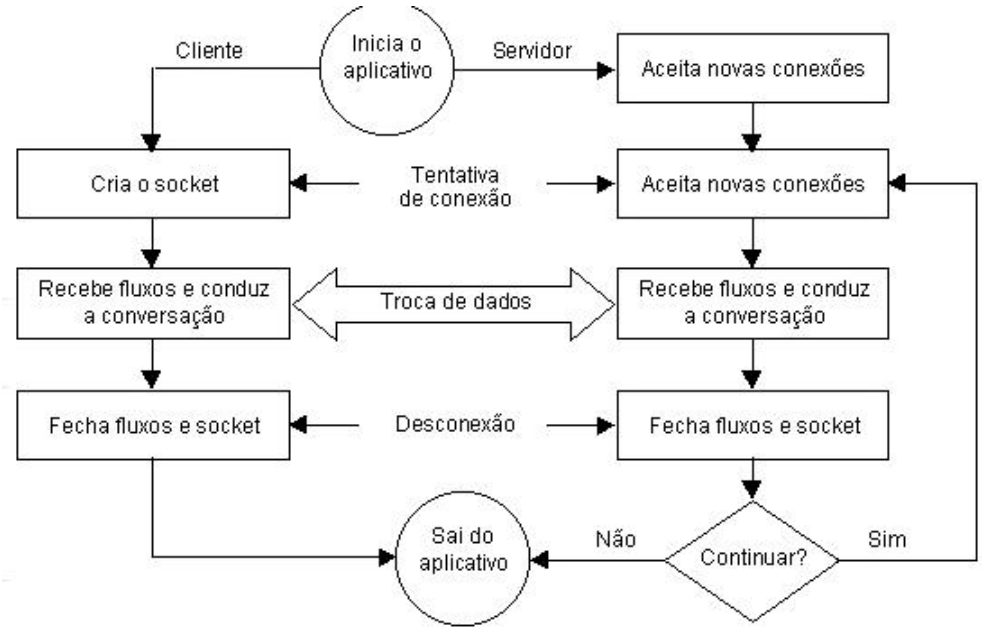



Figura 1: Fluxo de troca de dados com sockets.

Como primeiros passos na criação do servidor, é necessário importar o pacote java.net e em seguida instanciar um objeto do tipo ServerSocket, responsável por atender pedidos via rede e em determinada porta. Após receber uma conexão, um objeto do tipo Socket deve ser criado para manter a comunicação entre o cliente e o servidor.

Vejamos um exemplo. A seguinte linha cria o ServerSocket, que irá esperar conexões na porta 12345 (caso esta porta já esteja em uso, uma exceção será lançada):

```
1 ServerSocket server = new ServerSocket(12345);
```

Em seguida criamos um objeto Socket, o qual irá tratar da comunicação com o cliente, assim que um pedido de conexão chegar ao servidor e a conexão for aceita:

```
1 Socket client = server.accept();
```

Como vimos no exemplo, um socket servidor precisa definir o número da porta para receber conexões dos clientes. Este número pode variar entre 0 e 65535, porém, em nossas aplicações só devemos utilizar de 1024 em diante, pois as portas com números abaixo deste são reservados para o uso do sistema (por exemplo a porta 80 é usada pelo protocolo HTTP, 25 pelo SMTP, 110 pelo POP3, entre vários outros serviços).

## O protocolo TCP

Quando necessitamos de uma troca confiável de informações, isto é, quando é necessária a confirmação de recebimento da mensagem enviada, devemos utilizar o protocolo TCP (Transmission Control

[1] https://www.devmedia.com.br/java-sockets-criando-comunicacoes-em-java/9465

Protocol). Este protocolo estabelece uma conexão entre dois pontos interligados. Por exemplo, uma mensagem enviada de um host (o termo host representa uma máquina conectada na rede) a outro é confirmada pelo host receptor indicando o correto recebimento da mensagem. Uma mensagem pode ser enviada em vários pacotes, o TCP cuida para que os pacotes recebidos sejam remontados no host de destino na ordem correta (caso algum pacote não tenha sido recebido, o TCP requisita novamente este pacote). Somente após a montagem de todos os pacotes é que as informações ficam disponíveis para nossas aplicações. A programação do TCP com sockets utiliza streams, o que simplifica muito o processe de leitura e envio de dados pela rede.

Streams são objetos Java que permitem obter dados de qualquer fonte de entrada, seja o teclado, um arquivo ou até mesmo um fluxo de bytes recebidos pela rede (o que é o nosso caso). Isto torna a manipulação de dados da rede como se fossem arquivos, ao ler dados enviados é como se estivéssemos lendo um arquivo e ao enviar dados é como se estivéssemos gravando dados em um arquivo.

### Um primeiro servidor TCP

Vamos começar agora a trabalhar na prática com sockets. Primeiro vamos montar um servidor TCP que permite a seus clientes solicitarem a data e a hora atuais do servidor. A primeira versão deste servidor tem uma limitação (que mostraremos mais tarde como resolver): apenas um cliente pode ser atendido por vez.

Uma das características importantes do TCP é que os pedidos de conexões dos clientes vão sendo mantidos em uma fila pelo sistema operacional até que o servidor possa atendê-los. Isto evita que o cliente receba uma negação ao seu pedido, pois o servidor pode estar ocupado com outro processo e não conseguir atender o cliente naquele momento.

Cada sistema operacional pode manter em espera um número limitado de conexões até que sejam atendidas. Quando o sistema operacional recebe mais conexões que esse limite, as conexões mais antigas vão sendo descartadas.

Veja como funciona o nosso primeiro exemplo:

- Ao ser iniciado o servidor fica ouvindo na porta 12345 a espera de conexões de clientes;
- O cliente solicita uma conexão ao servidor;
- O servidor exibe uma mensagem na tela com o endereço IP do cliente conectado;
- O servidor aceita a conexão e envia um objeto Date ao cliente;
- O cliente recebe o objeto do servidor e faz o cast necessário, em seguida exibe na tela as informações de data;
- O servidor encerra a conexão.
- Na Algoritmo 1 é apresentado o código do nosso primeiro exemplo de servidor e na Algoritmo 2 é apresentado o código do cliente que utiliza o nosso servidor.

```
1  public class ServidorTCPBasico {
2    public static void main(String[] args) {
3      try {
4        // Instancia o ServerSocket ouvindo a porta 12345
5        ServerSocket servidor = new ServerSocket(12345);
6        System.out.println("Servidor ouvindo a porta 12345");
7        while(true) {
8          // o metodo accept() bloqueia a execucao ate que
9          // o servidor receba um pedido de conexao
10         Socket cliente = servidor.accept();
11         System.out.println("Cliente conectado: " + cliente.
     getInetAddress().getHostAddress());
12         ObjectOutputStream saida = new ObjectOutputStream(cliente.
     getOutputStream());
```

```
13          saida.flush();
14          saida.writeObject(new Date());
15          saida.close();
16          cliente.close();
17        }
18      }
19      catch(Exception e) {
20         System.out.println("Erro: " + e.getMessage());
21      }
22      finally {...}
23    }
24  }
```

Algoritmo 1: Código do servidor TCP básico.

```
1  public class ClienteTCPBasico {
2    public static void main(String[] args) {
3      try {
4        Socket cliente = new Socket("paulo",12345);
5        ObjectInputStream entrada = new ObjectInputStream(cliente.
     getInputStream());
6        Date data_atual = (Date)entrada.readObject();
7        JOptionPane.showMessageDialog(null,"Data recebida do servidor:" +
     data_atual.toString());
8        entrada.close();
9        System.out.println("Conexao encerrada");
10     }
11     catch(Exception e) {
12       System.out.println("Erro: " + e.getMessage());
13     }
14   }
15 }
```

Algoritmo 2: Código do cliente TCP básico.

## O protocolo UDP

Quando necessitamos de uma troca não confiável de informações podemos usar o protocolo UDP (User Datagram Protocol), pois este protocolo não garante a entrega dos pacotes (o UDP não espera uma mensagem de confirmação do host de destino). É de responsabilidade da aplicação receptora a remontagem dos pacotes na ordem correta e a solicitação de reenvio de pacotes que não foram recebidos. O UDP utiliza datagram sockets para a troca de mensagens. As principais aplicações do UDP são aplicações como transmissões de vídeo, skype, voip, etc... Para exemplificar imagine um serviço de voz sobre IP onde um pacote é perdido enquanto dois usuários conversam, não faz sentido reenviar o pacote pois o usuário da outra ponta precisaria saber que ainda faltam pacotes a receber. Veja uma simulação abaixo da conversa entre dois usuários:

- Erika: Olá, Paulo.
- Paulo: Olá, Erika.
- Erika: Como você está?
- Paulo: Tudo bem e vc? (este pacote foi perdido)

No exemplo acima o TCP enviaria novamente o pacote e o usuário da outra ponta deveria ficar esperando. Não faria nenhum sentido isso tratando-se de uma ligação telefônica. Este é um típico caso onde o UDP se aplica perfeitamente.

Os datagram sockets são mensagens que podem ser enviadas pela rede quando não existe a necessidade de confirmação de entrega, de tempo de entrega e nem mesmo garantia de conteúdo. Datagramas são úteis em aplicações que não necessitam do estabelecimento de uma conexão para o envio da mensagem. Um bom exemplo do seu uso é o envio de mensagens em broadcast para clientes de uma rede (o servidor pode enviar um datagrama para todos os clientes avisando que irá reiniciar, por exemplo). Em Java podemos trabalhar com datagramas utilizando as classes DatagramPacket e DatagramSocket do pacote java.net.

### Um servidor UDP básico

Antes de mostramos como criar um servidor TCP capaz de receber várias conexões simultâneas, vamos mostrar como criar um servidor UDP. Como vimos, o UDP envia os pacotes sem esperar por uma resposta do receptor. Este protocolo pode ser útil em situações como o envio de pacotes multimídia, por exemplo, ou um serviço de voz sobre ip, o que é muito comum.

Nosso servidor UDP envia mensagens para os clientes de uma determinada rede local. Perceba neste exemplo que no UDP o cliente também aguarda mensagens que poderão ser enviadas pelo servidor, ou seja, mantém um DatagramSocket em uma determinada porta. Por exemplo, o seguinte trecho cria o DatagramSocket que irá esperar mensagens na porta 12346.

```
DatagramSocket serverdgram = new DatagramSocket(12346);
```

Veja na Algoritmo 3 o remetente UDP exemplo e na Algoritmo 4 o código do receptor UDP, na Figura 4 você pode ver a mensagem sendo enviada pelo remetente. Na Figura 5 temos o receptor sendo inicializado para receber as mensagens e na Figura 6 a mensagem recebida pelo receptor. Como estamos usando UDP neste caso o remetente sempre vai mostrar a mensagem indicando que a mensagem foi enviada, porém não existe a confirmação de que a mensagem foi recebida.

```
public class RemetenteUDP {

  public static void main(String[] args) {

    if(args.length != 3) {
        System.out.println("Uso correto: <Nome da maquina> <Porta> <
   Mensagem>");
        System.exit(0);
    }

    try {
      //Primeiro argumento e o nome do host destino
      InetAddress addr = InetAddress.getByName(args[0]);
      int port = Integer.parseInt(args[1]);
      byte[] msg = args[2].getBytes();
      //Monta o pacote a ser enviado
      DatagramPacket pkg = new DatagramPacket(msg,msg.length, addr,
   port);
      // Cria o DatagramSocket que sera responsavel por enviar a
   mensagem
      DatagramSocket ds = new DatagramSocket();
      //Envia a mensagem
      ds.send(pkg);
      System.out.println("Mensagem enviada para: " + addr.
   getHostAddress() + "\n" +
        "Porta: " + port + "\n" + "Mensagem: " + args[2]);

      //Fecha o DatagramSocket
      ds.close();
    }
```

```
27
28      catch(IOException ioe) {...}
29    }
30  }
```

Algoritmo 3: Classe RemetenteUDP.

```
1  public class ReceptorUDP {
2
3    public static void main(String[] args) {
4        if(args.length != 1) {
5          System.out.println("Informe a porta a ser ouvida");
6          System.exit(0);
7        }
8
9        try {
10         //Converte o argumento recebido para inteiro (numero da porta)
11         int port = Integer.parseInt(args[0]);
12         //Cria o DatagramSocket para aguardar mensagens, neste momento
     o metodo fica bloqueando
13         //ate o recebimente de uma mensagem
14         DatagramSocket ds = new DatagramSocket(port);
15         System.out.println("Ouvindo a porta: " + port);
16         //Preparando o buffer de recebimento da mensagem
17         byte[] msg = new byte[256];
18         //Prepara o pacote de dados
19         DatagramPacket pkg = new DatagramPacket(msg, msg.length);
20         //Recebimento da mensagem
21         ds.receive(pkg);
22         JOptionPane.showMessageDialog(null,new String(pkg.getData()).
     trim(),
23           "Mensagem recebida",1);
24         ds.close();
25       }
26
27       catch(IOException ioe) {...}
28   }
29 }
```

Algoritmo 4: Classe ReceptorUDP.

## Referências


- https://www.devmedia.com.br/java-sockets-criando-comunicacoes-em-java/9465
- Java, guia do programador – Peter Jandl Junior, editora Novatec
- Java, como programar – 6ª Edição – H. M. Deitel e P. J. Deitel, editora Prentice Hall
- Java network programming – 3rd Edition – Elliotte Rusty Harold, editora O'REILLY