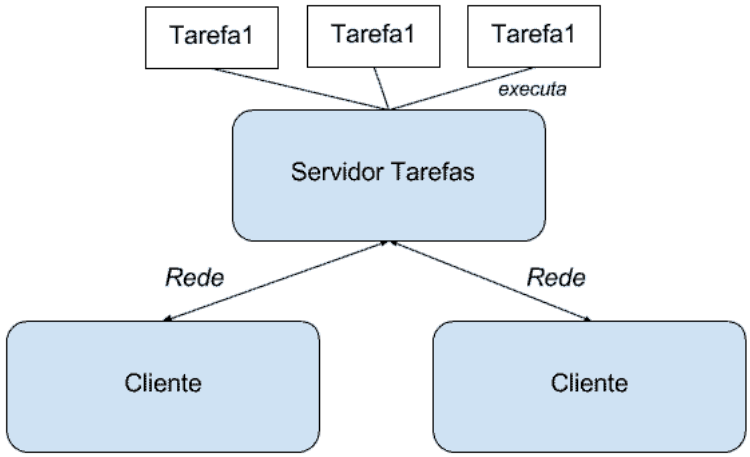
04

# O projeto Servidor de tarefas

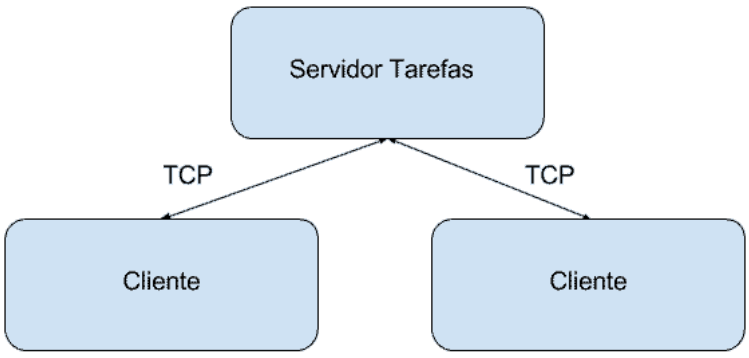
Para continuarmos a aprender mais sobre as threads da JVM, escolhemos um projeto prático onde introduzimos passo a passo novos recursos sobre threads. O objetivo é criar um servidor onde podemos submeter tarefas a executar. O servidor pode ou não confirmar o recebimento das tarefas e, claro, deve executá-las em paralelo.




## Socket e TCP/IP

Por conta da necessidade de dois computadores se comunicarem, surgiram diversos protocolos que permitissem tal troca de informação. O protocolo que vamos usar aqui é o **TCP** (***T***ransmission ***C***ontrol ***P***rotocol*).

Através do **TCP**, é possível criar um fluxo entre dois ou mais computadores - como é mostrado no diagrama abaixo:




É possível conectar mais de um cliente ao mesmo servidor, como é o caso de diversos banco de dados, servidores web, servidores de e-mail ou ftp, etc.

Ao escrever um programa em Java que se comunique com outra aplicação, não é necessário se preocupar com um nível tão baixo quanto o protocolo. As classes que trabalham com eles já foram disponibilizadas para serem usadas por nós no pacote `java.net`.

A vantagem de se usar o TCP, em vez de criar nosso próprio protocolo de bytes, é que o TCP vai garantir a entrega dos pacotes que transferimos e criar um protocolo base para isto é algo bem complicado.

Por outro lado, o TCP não é um *protocolo de aplicação* e sim de *transporte*. Isso significa que não é preciso se preocupar em como os dados serão transmitidos. O TCP garante que os dados serão transmitidos de maneira confiável, mas não se preocupa com o significado desses dados.

Isso é tarefa do protocolo de aplicação. Através dele, dependendo do protocolo, como o HTTP ou o FTP, podemos então definir que queremos acessar um arquivo no servidor, enviar parâmetros de pesquisa ou submeter dados de um formulário.

Em outras palavras, o TCP garante que os dados são transmitidos e o protocolo de aplicação define o significado desses dados.

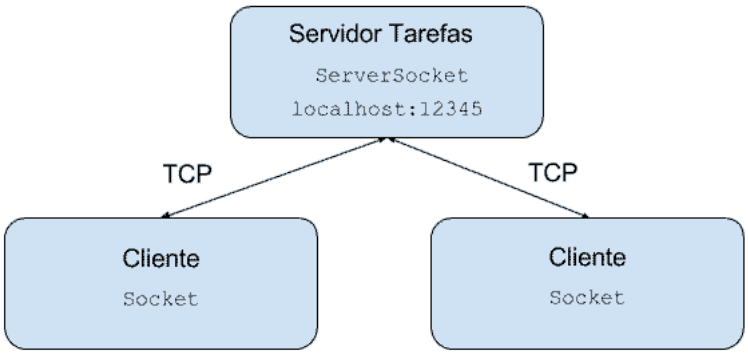
## Abrindo Portas

Nosso objetivo é estabelecer uma conexão e já mencionamos que diversos clientes podem se conectar a um só servidor. Cada cliente vai manter uma conexão com o servidor, mas como o servidor saberá distinguir entre os clientes?

Assim como existe o **IP** para identificar uma máquina, a **porta** é a solução para identificar diversos clientes *em uma máquina*. Esta porta é um número de 2 bytes, **varia de 0 a 65535**. Se todas as portas de uma máquina estiverem ocupadas, não é possível se conectar a ela enquanto nenhuma for liberada. Então, além do IP, também é preciso saber a porta!

## O que é um Socket?

Já sabemos que vamos utilizar o TCP e que precisamos do IP da máquina servidora e a porta. Todos esses detalhes do protocolo são abstraídos no mundo Java através de um **socket**. Um socket é o *ponto-final de um fluxo de comunicação* entre duas aplicações, através de uma rede. É exatamente isso que estamos procurando!




Vamos primeiro implementar o servidor, usando as classes do pacote `java.net`. O primeiro passo é criar o `ServerSocket`. Ao criar o `ServerSocket`, precisamos definir a porta. Há algumas portas já pré-definidas no sistema operacional. Por exemplo, a porta 22 é reservada para o SSH, 20 para o FTP, 80 para o HTTP etc. Normalmente, escolhendo um porta maior do que 1023, não devemos entrar em conflito com portas já pré-definidas.

Então vamos criar o projeto **servidor-tarefas**. No nosso exemplo vamos usar a porta ***12345***:

```
package br.com.alura.servidor;

import java.io.IOException;
import java.net.ServerSocket;

public class ServidorTarefas {

    public static void main(String[] args) throws IOException {
        System.out.println("---- Iniciando Servidor ----");
        ServerSocket servidor = new ServerSocket(12345);
```

```
    }
}
```

Criamos apenas um objeto `ServerSocket`, ainda não podemos aceitar uma conexão. Para tal, devemos chamar o método `accept`:

```
public class ServidorTarefas {

    public static void main(String[] args) throws Exception {
        System.out.println("---- Iniciando Servidor ----");
        ServerSocket servidor = new ServerSocket(12345);
        Socket socket = servidor.accept();
    }
}
```

O método `accept` é bloqueante e trava a thread principal. Ou seja, ao rodar, a thread `main` fica parada até receber uma conexão através de um cliente. Repare também que o retorno desse método é um `Socket`, que abstrai os detalhes da conexão.

## Criando o cliente

Já podemos pensar no cliente. Nele vamos usar a mesma classe `Socket`, só que agora criando através do construtor, passando o IP da máquina e a porta:

```
package br.com.alura.cliente;

import java.net.Socket;

public class ClienteTarefas {

    public static void main(String[] args) throws Exception {
        Socket socket = new Socket("localhost", 12345);
        System.out.println("Conexão Estabelecida");
        socket.close();
    }
}
```

Com o nosso servidor rodando, já podemos estabelecer uma conexão. A nossa classe `ClienteTarefas` deve rodar sem problema e imprimir a mensagem.

Após rodar o cliente percebemos que a máquina virtual do cliente terminou. Não há surpresa aqui, no entanto o nosso servidor também parou de rodar. O nosso servidor só aceita um cliente! Claro que isso precisa mudar...

## Aceitando vários clientes

O problema é que o servidor precisa chamar o método `accept` para cada cliente, e no nosso código chamamos apenas uma vez `accept`. Como solução, vamos colocar o método dentro de um laço infinito. Na classe `ServidorTarefas`:

```
public static void main(String[] args) throws Exception {

    System.out.println("---- Iniciando Servidor ----");
    ServerSocket servidor = new ServerSocket(12345);

    while (true) {
        Socket socket = servidor.accept();
        System.out.println("Aceitando novo cliente na porta " + socket.getPort());
    }
}
```

Agora já podemos testar e estabelecer conexões através de vários clientes. No nosso caso, vamos criar apenas dois clientes. Repare abaixo uma possível saída no console do Eclipse:

```
---- Iniciando Servidor ----
Aceitando novo cliente na porta 61430
Aceitando novo cliente na porta 61432
```

O interessante é que a porta muda para cada cliente. Enquanto usamos a porta **12345** para criar a conexão inicial, toda comunicação a partir desse momento é feita com uma porta dedicada pra cada cliente.

## Cada cliente, uma nova thread

Vamos tentar simular uma execução mais pesada no lado do servidor. A ideia é que o cliente submeta tarefas e algumas delas podem ser mais pesadas. Para simular isso, vamos parar execução por 20 segundos através do método `Thread.sleep(20000)` :

```
public class ServidorTarefas {

    public static void main(String[] args) throws Exception {
        System.out.println("---- Iniciando Servidor ----");
        ServerSocket servidor = new ServerSocket(12345);

        while (true) {
            Socket socket = servidor.accept();

            System.out.println("Aceitando novo cliente na porta " + socket.getPort());
            Thread.sleep(20000);
        }
    }
}
```

E para travar o nosso cliente, vamos simular uma leitura do teclado:

```
public class ClienteTarefas {

    public static void main(String[] args) throws Exception {
        Socket socket = new Socket("localhost", 12345);
        System.out.println("Conexão Estabelecida");

        Scanner teclado = new Scanner(System.in);
```

```
            teclado.nextLine();

            socket.close();
        }
    }
```

Novamente, vamos estabelecer duas conexões. Ao rodar, percebemos que o primeiro cliente trava a execução do segundo. Isto acontece pois todos os clientes estão sendo executados na thread principal.

Bom, isso é fácil de resolver, basta criar uma nova thread para cada cliente no lado do servidor. Através dessa thread analisaremos os dados enviados pelo cliente e distribuiremos a tarefa. Então, nada mais justo do que chamar essa thread de `distribuirTarefas`, logo a sua classe será `DistribuirTarefas`, com a interface `Runnable`.

```
public class DistribuirTarefas implements Runnable {

    private Socket socket;

    public DistribuirTarefas(Socket socket) {
        this.socket = socket;
    }

    @Override
    public void run() {
        System.out.println("Distribuindo as tarefas para o cliente " +  socket);

        try {
            Thread.sleep(20000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

}
```

E o nosso servidor fica assim:

```
public class ServidorTarefas {

    public static void main(String[] args) throws Exception {

        System.out.println("---- Iniciando Servidor ----");
        ServerSocket servidor = new ServerSocket(12345);

        while (true) {
            Socket socket = servidor.accept();
            System.out.println("Aceitando novo cliente na porta " + socket.getPort());
            DistribuirTarefas distribuirTarefas = new DistribuirTarefas(socket);
            new Thread(distribuirTarefas).start();
        }
    }
}
```

Agora já podemos testar novamente o nosso servidor, junto com um ou mais clientes. Como resultado, o nosso servidor não trava mais e aceita vários clientes. No próximo capítulo vamos receber comandos pelo cliente e pensar sobre a quantidade de clientes que o servidor pode atender!

## O que aprendemos?

- Trabalhar com `Socket` e `ServerSocket` para criar uma conexão TCP
  - o servidor precisa *aceitar* novos clientes
  - a comunicação é estabelecida através de uma porta inicial, mas após isso cada cliente usa a sua própria porta
- No lado do servidor devemos usar para cada cliente uma nova thread
  - isso é porque o método `accept` da classe `ServerSocket` é *bloqueante*