

Aplicação Servidor e forma de atendimento de pedidos:

- só depois de completar o pedido de um cliente é que passa ao seguinte: **iterativo**
- o atendimento de vários pedidos acontece ao mesmo tempo: **concorrente**

exercício 1

1.a) Obtenha o código para esta atividade, num dos modos:

- [projeto NetBeans](#)
- [projeto Eclipse](#)

1.b.1-

Usando [3 terminais](#), execute o servidor Echo simples, (classe `so2.EchoServer`) ...

1.b.2

e execute ao mesmo tempo dois pedidos (corra 2 clientes, classe `so2.EchoClient`).

Deve observar, pelo output nas várias janelas.

1.c.1

Torne mais clara a solicitação simultânea dos clientes, **descomentando** a linha do `sleep()`. Isto vai garantir um atendimento mais demorado...

1.c.2

Volte a executar ambos os clientes.

Deverá ser evidente que o segundo pedido só é processado após ter sido completado o atendimento do primeiro pedido.

2- transformar um Servidor Iterativo num Servidor Concorrente**2.a) - delegar o tratamento do pedido para um novo objecto**

Altere um pouco a estrutura do servidor, editando o código em `EchoServerConcorrente.java`, de modo a que o atendimento do pedido seja escondido numa nova classe: `AtendedorDePedidos` (esta classe já foi enviada no mesmo código).

Depois do `accept()`, deve criar uma **instância** desta classe (passando como argumento o socket de dados) e invocar um método `atendePedido()`.

Compile e teste esta nova versão (executar `so2.EchoServerConcorrente`).

2.b) - associar uma thread ao atendimento do pedido

Atualize o seu servidor para que passe a ser concorrente.

Basta tornar o tipo `AtendedorDePedidos` numa subclasse de `Thread`, adicionar um método `run()` e invocar o método `start()`.

Teste esse comportamento com vários pedidos em simultâneo (e com tempo de espera para atrasar o atendimento e facilitar a observação).

[\[video de apoio\]](#)

exercício 3

Para permitir a administração remota do serviço, é comum usar-se um canal diferente do usado para atender os clientes comuns. Usar canais diferentes permite ter regras de acesso específicas em cada porto.

Ainda com a programação explícita de Threads,

adicione ao servidor do exercício anterior uma nova classe Thread, que será responsável pela administração do serviço.

3.1- Para já, vamos suportar apenas uma operação, que será interromper o serviço.

Ajuste o código, para ter esta *inner class*.

```
class ServiceManager extends Thread {
    ServerSocket serviceSocket;
    ServerSocket adminSocket;

    public ServiceManager(ServerSocket s) {
        // exercício
    }
    public void run() {
        // exercício
    }

    public void adminService() {
        try {
            // ... // aceitar ligações TCP num porto (anterior +1) via adminSocket
            // se... parar o serviço: fechar o socket serviceSocket
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Deve também ativar uma instância desta thread antes de iniciar o ciclo de atendimento do serviço.

Se tiver só uma operação (desligar), basta haver uma ligação neste novo porto, que pode simular com:

\$ telnet localhost 9001

3.2- Estude o impacto da abordagem anterior no caso de haver naquele momento alguns pedidos em atendimento. Serão atendidos ou interrompidos abruptamente?

Simule, se necessário acrescentando espera adicional (sleep).

Se for necessário, coordene algumas operações entre as threads, com join().

Note: Em soluções futuras, a maior parte destes procedimentos são tratados pelos *containers*.



[[video de apoio](#)]

exercício 4

Com base nos exemplos anteriores,

altere o servidor da atividade 3, sobre veículos, para atender pedidos em paralelo, tornando-o concorrente.

Acrescente `Thread.sleep(2000)` a meio do atendimento do pedido, para ver a diferença de iterativo vs concorrente.

 [Contactar suporte do site](#) 

Nome de utilizador: [Rodrigo Alves](#) ([Sair](#))

[Resumo da retenção de dados](#)

[Obter a Aplicação móvel](#)

Fornecido por [Moodle](#)