Simulação de Pool de Impressão Dístribuida Utilizando Socket Exercício Computacional II - Sistemas Distribuídos

Rafael Gonçalves de Oliveira Viana[1]

[1]Sistemas de Informação – Universidade Federal do Mato Grosso do Sul (UFMS)
Caixa Postal 79400-000 – Coxim – MS – Brazil

rafael.viana@aluno.ufms.br
25 de outubro de 2017

Resumo. *Este documento relata como foi o desenvolvimento de uma Pool de Impressão na linguaguem Java versão 8, utilizando Threads e Sockets.*

1. Introdução

De acordo com [1], em um ambiente de trabalho, onde desejasse imprimir documentos enviados, a um curto período de tempo, é necessário a utilização de uma pool de impressão.

Uma pool de impressão permite que várias impressoras físicas possam ser controladas por uma única impressora lógica, sempre que um trabalho for enviado para a impressora lógica, está consultará o estado das impressoras físicas para verificar qual equipamento está livre no momento e enviará o trabalho para ela. Um exemplo de pool de impressão é demonstrada na figura 1.

```
ServerSocket servidor = new ServerSocket(this.porta);
Random r = new Random();
System.out.println("Entro: " + this.porta);
while (true) {
    if (r.nextInt( bound: 100) < 15) {
        String log = "Impressora " + this.porta + " está Off-line";
        Arquivos.CriarArquioX( nome: "logs_" + porta,  caminho: "./Logs", log);
        Thread.sleep( millis: r.nextInt( bound: 2000) + 1000);
        log = "Impressora " + this.porta + " está On-line";
        Arquivos.CriarArquioX( nome: "logs_" + porta,  caminho: "./Logs", log);
        System.out.println("Volto: " + this.porta);
    }
    Socket cliente = servidor.accept();
    PrintStream ps = new PrintStream(cliente.getOutputStream());
    String localPort = Port(cliente);
    String status;
    if (r.nextInt( bound: 100) < 15) {
        status = "  ERRO";
        ps.println("ERRO");
    } else {
        status = "  OK";
        ps.println("OK");
    }
    if (!localPort.equals("Estralho")) {
        String log = "Impressão: " + (cont++) + " Cliente " + localPort + " Data: " + LocalTime.now().toString() + status;
        Arquivos.CriarArquioX( nome: "logs_" + porta,  caminho: "./Logs", log);
    }
```

Figura 1. Proposta do exercício computacional.

2. Fundamentação Téorica

Nesta seção serão tratadas, soluçoes de problemas pertinentes que foram utilizadas neste documento.

2.1. Problema de Concorrência

O problema de concorrência existem em diversos cenários em que, um pedaço de código que definimos como crítico não pode ser executado por duas threads ou mais, ao mesmo tempo. Apenas uma thread por vez consegue entrar em alguma região crítica e realizar determina função.

O java realiza a syncronyzação das threasd

Para mais informações referente a sincronização de threads na linguagem java consultar a referência [2].

2.2. Socket

Segundo [3].[4]. Os sockets são compostos por um conjunto de primitivas do sistema operacional e foram originalmente desenvolvidos para o BSD Unix. Sockets são suportados em Java desde o JDK 1.0, para sua utilização devemos fazer uso das classes contidas no pacote java.net.2.

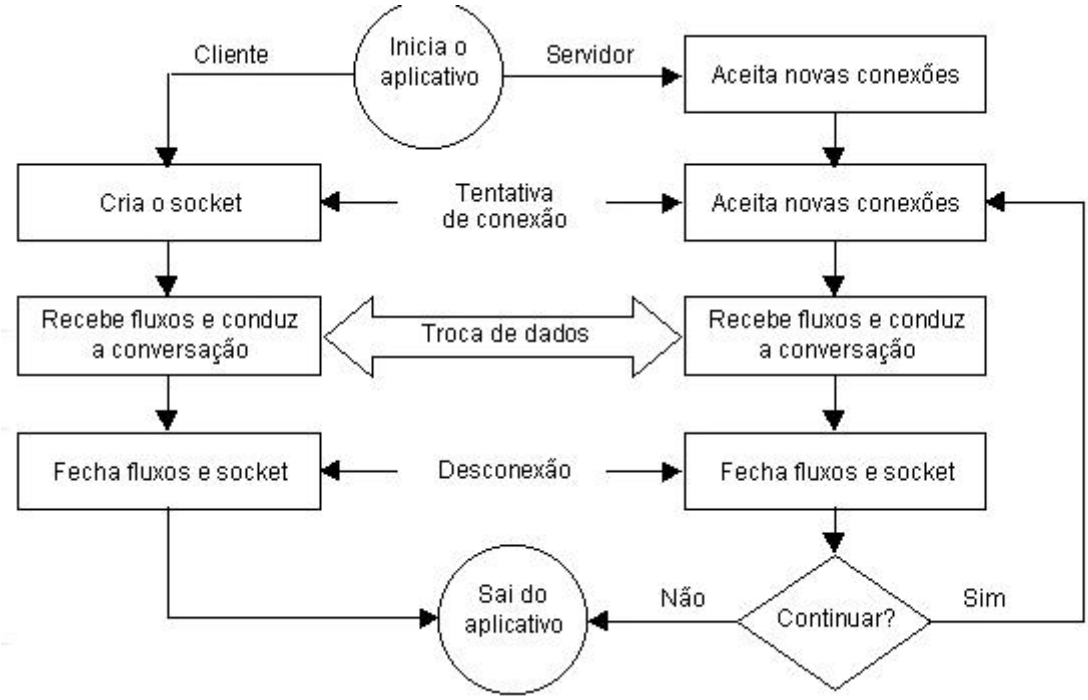

Figura 2. Fluxo de troca de dados com sockets.

2.3. Problema de Espera Ocupada

Para mais informações referente a espera ocupada de threads na linguagem java consultar as referências [4] [5].

3. Desenvolvimento

Para resolver o trabalho proposto, foi desenvolvido uma aplicação java que simula, uma pool de impressões que as ordena e envia para uma impressora. Foram criadas 3 classes Javas sendo elas: Cliente , Pool de Impressão e Impressora. As mesmas serão comentadas nas próximas seções.

3.1. Cliente

Essa classe se conecta a classe servidor, repassando informações via Socket, em uma via de mão dupla. Os dados serão aceitos na memória e posteriormente serão alocados

no pelo escalonador da pool de impressão, em alguma impressora disponível, onde mais tarde será notificado com o resultado da processamento da impressão.

O código responsável pela chamada de impressão no cliente e simples, podemos observar o método na figura 3.

```
public class Cliente {
    public static void main(String[] args)
            throws UnknownHostException, IOException, InterruptedException {
        new Cliente( host: "svgov.ddns.net",  porta: 3000).executa();
    }

    private String host;
    private int porta;

    public Cliente(String host, int porta) {
        this.host = host;
        this.porta = porta;
    }

    public void executa() throws UnknownHostException, IOException, InterruptedException {
        System.out.println("Bem vindo digite algo para ser impresso!");
        BufferedReader rf = new BufferedReader(new InputStreamReader(System.in));
        String mensagem = rf.readLine();
        Socket cliente = new Socket(this.host, this.porta);
        System.out.println("O cliente de numero "+ cliente.getLocalPort() + " se conectou ao servidor "+ cliente.getPort()+"!");

        Recebedor r = new Recebedor(cliente.getInputStream());
        Thread t1 = new Thread(r);
        t1.start();
        PrintStream saida = new PrintStream(cliente.getOutputStream());

        saida.println(mensagem);

        t1.join();
        saida.close();
        cliente.close();
    }
```

Figura 3. Código cliente, da pool de impressão.

3.2. Pool de Impressão - Servidor

Classe responsável pela comunicação dos Sockets (Cliente/Impressora), ela contém o escalonador de impressão, que é responsável por designar os serviços na pool de impressão.

O código do servidor é responsável por 3 threads sendo elas:

1. Aceitar conexões via Socket e adicionar documento para impressão.
2. Procurar impressoras disponíveis (escalonador de impressão)
3. Enviar documento para impressoras e receber resposta de confirmação das mesmas.

A figura 4 demonstra a função que inicializa as variáveis proposta no exercício computacional, além de iniciar as Threads do servidor. O responsável por determinar quais impressoras estão disponíveis, e alocar trabalho para as mesmas, é o escalonador de impressão que é desenvolvido na classe *Chegada de Documento*. O escalonador ao perceber que não tem tarefas a realizar vai dormir wait(), quando um processo chega para ser escalonado ele acorda e realiza o repasse para a primeira impressora disponível, e volta a dormi.

```
public void executa() throws IOException {
    BufferedReader rd = new BufferedReader(new InputStreamReader(System.in));

    System.out.println("#### BEM VINDO AO POOL DE IMPRESSÃO RAFAEL VIANA ####\n" +
            "Exemplos de valores de entrada\n"+
            "R = (0 a 100), 0 representando 0% e 100 represetando 100% de probabiliadde de erro de impressão.\n" +
            "M = (1 a N>1) Valor inteiro  do Tamanho Buffer do Servidor.\n" +
            "T = (1 a 3) Valor inteiro\n" +
            "Informe em sequência R M T , separados por espaço entre os valores!");

    String[] linha = rd.readLine().split( regex: " ");
    Buffer b = new Buffer(Integer.parseInt(linha[0]),Integer.parseInt(linha[1]),Integer.parseInt(linha[2]));
    System.out.println("Servidor Rodando na Porta "+ porta+ "!");
    b.addImpressora(new Impressora( ip: "sdufms2017.ddns.net", porta: 4200, name: "4200"));
    b.addImpressora(new Impressora( ip: "sdufms2017.ddns.net", porta: 4300, name: "4300"));
    SaidaDocumento escalonador = new SaidaDocumento(b);
    Thread esc = new Thread(escalonador);
    esc.start();
    ServerSocket servidor = new ServerSocket(this.porta);
    while (true) {
        Socket cliente = servidor.accept();
        ChegadaDocumento tc = new ChegadaDocumento(b, cliente, new Scanner(cliente.getInputStream()).nextLine(),
                new PrintStream(cliente.getOutputStream()));
        new Thread(tc).start();
    }
```

Figura 4.

O código completo, incluindo todas as classes como na figura 5, estão disponíveis no link https://github.com/rafaelgov95/SD/tree/master/PTJI/Projeto-PTJ

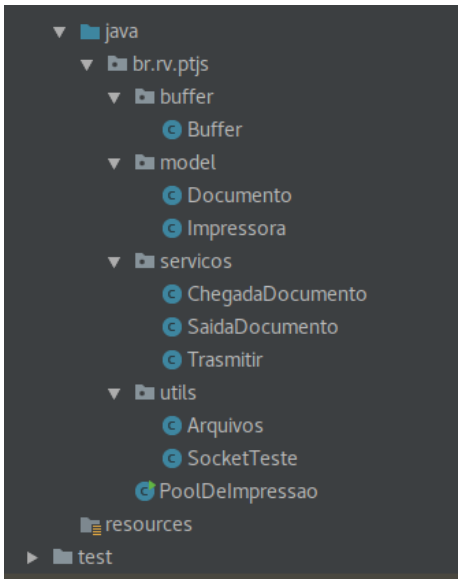

Figura 5. Classes do Servidor de pool de Impressão

3.3. Impressoras

Classe responsável pela impressão dos documentos contidos no buffer da Pool de Impressão 3.2.

Ela trabalha junto ao escalonador do servidor, que é responsável por notificar as impressoras cadastradas, que novas impressões devem ser executadas, vale destacar que o escalonador foi desenvolvido, de acordo com a boa prática de não deixar uma espera ocupada.

A figura 6, apresenta o método que realiza a criação das Threads. Elas executam em modo servidor aguardando o escalonador alocar alguma tarefa.

```
public void executa() throws IOException, InterruptedException {
    StartImpressora imp1 = new StartImpressora( porta: 4200);
    StartImpressora imp2 = new StartImpressora( porta: 4300);
    Thread t1 = new Thread(imp1);
    t1.start();
    Thread t2 = new Thread(imp2);
    t2.start();
    t1.join();
    t2.join();

}
```

Figura 6. Criação de duas impressoras pelo método StartImpressora.

Após a criação, as Threads que representam as impressoras, as mesmas ficam aguardando no while(true), que não gera espera ocupada, por causa do método de servidor, *servidor.accept*, como na figura 7.

```
ServerSocket servidor = new ServerSocket(this.porta);
Random r = new Random();
System.out.println("Entro: " + this.porta);
while (true) {
    if (r.nextInt( bound: 100) < 15) {
        String log = "Impressora " + this.porta + " está Off-line";
        Arquivos.CriarArquioX( nome: "logs_" + porta, caminho: "./Logs", log);
        Thread.sleep( millis: r.nextInt( bound: 2000) + 1000);
        log = "Impressora " + this.porta + " está On-line";
        Arquivos.CriarArquioX( nome: "logs_" + porta, caminho: "./Logs", log);
        System.out.println("Volto: " + this.porta);
    }
    Socket cliente = servidor.accept();
    PrintStream ps = new PrintStream(cliente.getOutputStream());
    String localPort = Port(cliente);
    String status;
    if (r.nextInt( bound: 100) < 15) {
        status = "  ERRO";
        ps.println("ERRO");
    } else {
        status = "  OK";
        ps.println("OK");
    }
    if (!localPort.equals("Estralho")) {
        String log = "Impressão: " + (cont++) + " Cliente " + localPort + " Data: " + LocalTime.now().toString() + status;
        Arquivos.CriarArquioX( nome: "logs_" + porta, caminho: "./Logs", log);
    }
```

Figura 7.

4. Comandos da Implementação

Nesta seção será apresnetado os comandos para poder utilizar o Pool de Impressão.

4.1. Cliente

4.2. Servidor

4.3. Impressoras

5. Testes e Resultados

Nesta seção será demonstrado os logs das impressoras e do servidor após a conexão de 4 clientes, com o servidor. Para realizar os testes foram utilizado um Raspberry Pi3 e uma Virtual Machine da Google Developers.

5.0.1. Distribuição de Recursos

1. Como servidor da pool de impressão ficou o Raspberry Pi3, demonstrando que a classe Servidor não precisa de um grande poder computacional para ser executada 3.2.
2. Como impressoras virtuais da pool de impressão, que se encontra no Raspberry Pi3, foi utilizado uma Virtual Machine da Google Developer, com o código de impressoras virtuais.3.3

5.0.2. Passos

1. Primeiramente a requisição de algum cliente se conecta ao pool de impressão, presente no Raspberry Pi3, por meio do Socket.
2. Ao receber a requisição de impressão, a pool adiciona o documento ao Buffer, e acorda o escalonador de impressão que envia a impressão para, alguma impressora disponível na virtual machine da Google developers.
3. Ao receber o documento a impressora responde um ”OK”para o pool, porém ela pode perder o documento e então envia uma mensagem de ERRO, o que provocaria o reenvio da mesma, até que consiga realizar a impressão.
4. Durante todos esses passos mensagens como ”Adicionada à Fila”ou mensagens de erros, são encaminhadas para os clientes em tempo real.

Agora que já entendemos como funciona a comunicação, podemos visualizar os relatórios, comecaremos pelo do escalonador, que após a execução dos 4 clientes é demonstrado na figura 8.

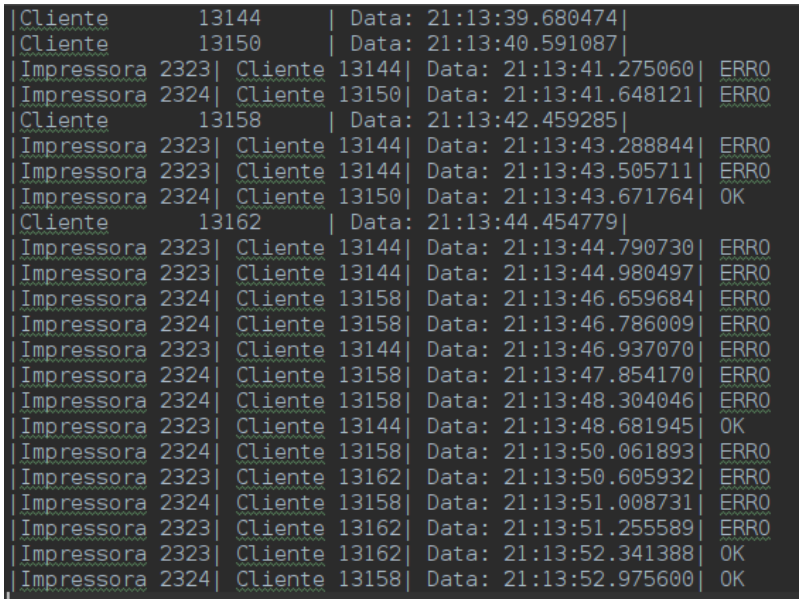

Figura 8. Relatório de impressões, registrado pelo escalonador presente no servidor.

O relatório da impressora 4200, pode ser observado na figura 9.

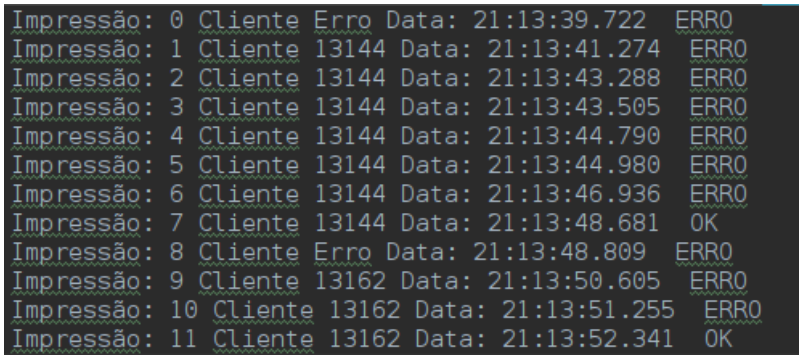

Figura 9. Relatório da impressora 4200, após execução dos 4 clientes.

O relatório da impressora 4300, pode ser observado na figura 10.

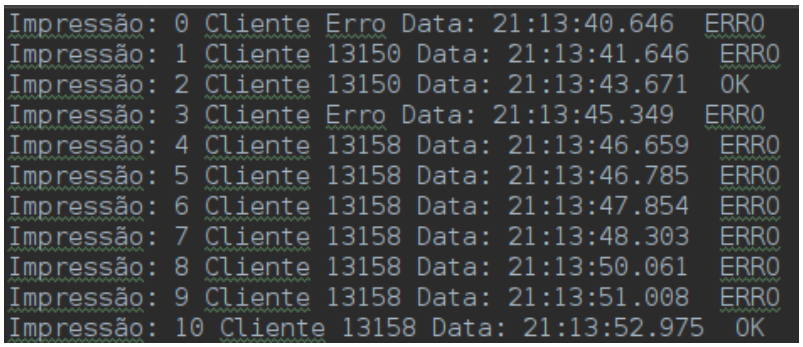

Figura 10. Relatório da impressora 4300, após execução dos 4 clientes.

O cliente após o envio do documento, aguarda a impressão do mesmo, enquanto isso recebe notificações a respeito de sua impressão em tempo real, podemos ver na figura 11, o resultado final de notificações do cliente.

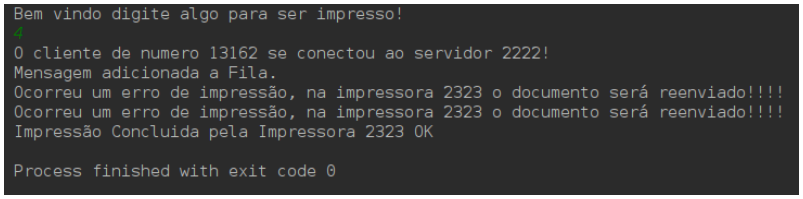

Figura 11. Notificações em tempo real de impressão para o Cliente.

6. Conclusão

Neste relatório foi demonstrado o desenvolvimento de uma pool de impressão, enfrentando problemáticas como: região crítica de memória compartilhada entre Threads, comunicação via Sockets geograficamente distantes , espera ocupada do escalonadores, e apontando suas soluções utilizando o *synchronized*, *SocketJava* e *Wait()/Notify* respectivamente.

Referências

[1] E. Software, "Pool de impressão – você sabe sua finalidade?." http://www.devmedia.com.br/java-sockets-criando-comunicacoes-em-java/9465, 2017. [Online; acesso em 22-Outubro-2017].

[2] Jacques, "Sincronização de Threads." http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/threads/sincronizacao.html, 2017. [Online; acesso em 22-Outubro-2017].

[3] Caelum, "Apêndice - Sockets." https://www.caelum.com.br/apostila-java-orientacao-objetos/apendice-sockets/, 2017. [Online; acesso em 18-Outubro-2017].

[4] Caelum, "Apêndice - Problemas com concorrência." https://www.caelum.com.br/apostila-java-orientacao-objetos/apendice-problemas-com-concorrencia/, 2017. [Online; acesso em 20-Outubro-2017].

[5] MC514–Sistemas, "Produtores e Consumidores." http://www.ic.unicamp.br/~islene/mc514/prod-cons/prod-cons.pdf, 2017. [Online; acesso em 22-Outubro-2017].