Produtor e Consumidor Distribuído com *Buffer* Limitado

Braully Rocha da Silva[1]

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)

1. Introdução

Este trabalho apresenta uma proposta de implementação para o problem do Produtor e Consumidor Distribuído com *Buffer* Limitado.

2. Problema

Processos compartilham um *buffer* de tamanho fixo com N posições, processos que produzem mensagens e inserem no *buffer* são chamados de produtores, processo que retiram mensagens do *buffer* são chamados de consumidores.

"A relação produtor-consumidor ocorre em sistemas concorrentes e o problema se resume em administrar o *buffer* que tem tamanho limitado. Se o *buffer* está cheio, o produtor deve se bloquear, se o *buffer* está vazio, o consumidor deve se bloquear"[Toscani et al. 2003].

Dado a definição do problema, transportando para um cenário distribuído, onde os processos estarão distribuídos, adicionamos uma complexidade extra a esse tradicional problema de concorrência. Para tanto os processos precisam se comunicar e interagir com o *buffer* através da rede, em um tipíco sistema distribuído.

3. Arquitetura Utilizada

A arquitetura performada para o problema do produtor-consumidor distribuído é a Cliente-Servidor.

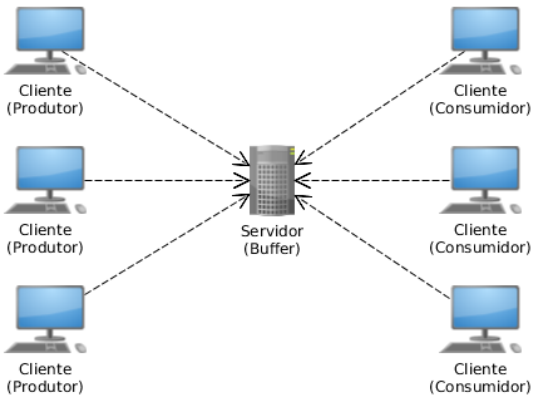

Figura 1. Arquitetura Cliente Servidor.

Conforme visto na figura 1 o *Buffer* Limitado será um processo servidor, ao passo que os Consumidores e os Produtores serão os clientes.

3.1. Consumidor

O Consumidor é criado por uma aplicação de configuração, neste momento o consumidor deve saber as informações de localização do *buffer* distribuído, para que este possa realizar as chamadas remotas. O Consumidor deve verificar se o *buffer* está vazio antes de tentar realizar uma retirada de dados, caso o *buffer* esteja vazio o consumidor irá se bloquear por um tempo e esperar que mais dados sejam produzidos. Caso o consumidor tente retirar dados do *buffer* vazio, este irá informar um erro de *BufferVazio*, para que o consumidor tente novamente mais tarde.

No pseudocódigo 1 é demonstrado o ciclo de trabalho do consumidor, enquanto um "fim" não for explicitamente sinalizado o consumidor irá constantemente verificar se o *buffer* remoto não está vazio, se tiver elementos ele irá tentar remover do *buffer*, se tiver sucesso ele irá aguardar um tempo aleatório entre 0 e `TEMPO_MAXIMO_ESPERA_ENTRE_CONSUMOS` segundos, caso não tenha elementos no *buffer* o consumidor irá esperar entre 0 e `TEMPO_MAXIMO_ESPERA_PRODUCAO` segundos para poder tentar novamente. Caso entre a verificação de existência de elementos no *buffer* e sua remoção o *buffer* tenha ficado vazio, o *buffer* poderá retornar uma exceção do tipo *BufferVazio*, neste caso o consumidor irá capturar o erro e aguardar o mesmo tempo que ele aguardaria se não existe elementos, tentando novamente após esse tempo.

Listing 1. Ciclo de Trabalho do Consumidor

```
while (!fim) {
    try {
        if (!referenciaRemotaBuffer.isVazio()) {
            Character removerDado = referenciaRemotaBuffer.removerDado(identificadorConsumidor);
            sleep(Math.random() * TEMPO_MAXIMO_ESPERA_ENTRE_CONSUMOS);
        } else {
            System.out.println(identificadorConsumidor + " buffer vazio, esperando produção");
            sleep(Math.random() * TEMPO_MAXIMO_ESPERA_PRODUCAO);
        }
    } catch (BufferVazioException e) {
        System.err.println("Tentando remover de buffer vazio");
        sleep(Math.random() * TEMPO_MAXIMO_ESPERA_PRODUCAO);
    } catch (Exception e) {
        System.err.println("Falha no consumidor: " + e.getLocalizedMessage());
        e.printStackTrace();
    }
}
```

3.2. Produtor

O Produtor assim como o consumidor é criado por uma aplicação de configuração, que informa uma referência para o *buffer* remoto, o produtor por sua vez produz informações de tempos em tempos e insere no *buffer*, consultando antes se esse não está cheio, caso esteja o *buffer* espera por um tempo até que alguma informação seja consumida e o *buffer* suporte novas informações. Caso o produtor tente inserir informações no *buffer* cheio, este irá informar um erro de *BufferCheio*, para que o produtor tente novamente mais tarde.

No pseudocódigo 1 é demonstrado o ciclo de trabalho do produtor, enquanto um "fim" não for explicitamente sinalizado o produtor irá verificar se o *buffer* remoto não está cheio, se tiver espaço livre ele irá produzir um dado e tentar inserir no *buffer*, se tiver sucesso ele irá aguardar um tempo aleatório entre 0 e `TEMPO_MAXIMO_ESPERA_ENTRE_PRODUCOES` segundos, caso o *buffer* esteja cheio o produtor irá dormir e esperar entre 0 e `TEMPO_MAXIMO_ESPERA_CONSUMO` segundos para poder tentar novamente. Caso entre a verificação de existência de espaço livre no *buffer* e sua produção o *buffer* tenha ficado cheio, o *buffer* poderá retornar uma exceção

do tipo *BufferCheio*, neste caso o produtor irá capturar o erro e aguardar o mesmo tempo que ele aguardaria se não existe espaços livres, tentando novamente após esse tempo.

Listing 2. Ciclo de Trabalho do Produtor

```
while (!fim) {
    Character dado = null;
    try {
        dado = RandomStringUtils.randomAlphabetic(1).charAt(0);
        if (!referenciaBufferRemoto.isCheio()) {
            referenciaBufferRemoto.inserirDado(dado, identificadorProdutor);
            sleep(Math.round(Math.random() * TEMPO_MAXIMO_ESPERA_ENTRE_PRODUCOES));
        } else {
            System.out.println("=" + identificadorProdutor + " buffer cheio, esperando consumo");
            sleep(Math.round(Math.random() * TEMPO_MAXIMO_ESPERA_CONSUMO));
        }
    } catch (BufferCheioException e) {
        System.err.println("Tentando produzir em buffer cheio");
        sleep(Math.round(Math.random() * TEMPO_MAXIMO_ESPERA_CONSUMO));
    } catch (Exception e) {
        System.err.println("Falha no produtor: " + e.getLocalizedMessage());
        e.printStackTrace();
    }
}
```

3.3. Buffer

O *buffer* distribuído deve disponibilizar para os processos clientes informações sobre sua situação e disponibilizar métodos para inserir e retirar informações.

Listing 3. Metodos do Buffer

```
    String situacaoBuffer();

    void inserirDado(char dado, String idProdutor);

    Character removerDado(String idConsumidor);

    Boolean isCheio();

    Boolean isVazio();
```

O pseudocódigo 3 apresenta a proposta deste trabalho para um *buffer* distribuído, os métodos *isCheio()* e *isVazio()* informam se o *Buffer* está cheio ou vazio, necessários para os produtores e consumidores informar sobre a situação do *buffer* antes de realizar uma operação. O método "*inserirDado*" permite ao produtor inserir dados no *buffer*, o parametro "idProdutor" é opcional e serve para o *buffer* conhecer o produtor da informação, ao passo que o método "*removerDados*" permite ao consumidor retirar informações do *buffer* e da mesma forma o parâmetro "idConsumidor" serve para o *buffer* conhecer o consumidor da informação.

Alem disso o *buffer* deve tratar a concorrência de operações de seus clientes, realizar operações de inserção e remoções atômicas, protegidas de acessos concorrentes de threads e processos, e o mais importante bloquear por um período de tempo consumidores quando o *buffer* estiver vazio e produtores quando o *buffer* estiver cheio, no ultimo caso informar um erro se a espera for muito longa,o para que não fiquem esperando indefinidamente.

Listing 4. Ciclo de Trabalho do Buffer Distribuído

```
public synchronized void inserirDado(char dado, String idProdutor) {
    while (tamanho >= TAMANHO_MAXIMO_BUFFER - 1) {
        try {
            System.out.println("Produtor tentando inserir em buffer cheio");
            System.out.println("Bloquear chamada temporariamente e esperar retiradas");
            this.wait(TEMPO_MAXIMO_ESPERA_CONSUMO);
        } catch (Exception ex) {
            System.err.println("Não ocorreu retirada durante a espera");
            throw new BufferCheioException();
        }
    }
    buffer[fim] = dado;
```

```
    System.out.println(">" + idProdutor + " inserindo "
                       + dado + "\tna posição " + fim + "\t");
    fim++;
    tamanho++;
    informarSitauacaoBuffer();
    /* Acordar chamadas de consumidores que possam estar aguardando (wait) */
    this.notifyAll();
}

public synchronized Character removerDado(String identificador) {
  Character retirado = null;
  while (tamanho <= 0) {
      try {
          System.out.println("Consumidor tentando remover de buffer vazio");
          System.out.println("Bloquear chamada  temporariamente e esperar produção");
          this.wait(TEMPO_MAXIMO_ESPERA_PRODUCAO);
      } catch (Exception e) {
          System.err.println("Não ocorreu produção durante a espera");
          throw new BufferVazioException();
      }
  }
  retirado = buffer[fim];
  tamanho--;
  System.out.println("<" + identificador + " retirando "
                     + retirado + "\tda posição " + fim + "\t");
  fim--;
  informarSitauacaoBuffer();
  /* Acordar chamadas de produtores que possam estar aguardando (wait) */
  this.notifyAll();
  return retirado;
}
```

O pseudocódigo 4 apresenta os métodos do *buffer* utilizado pelo produtor e consumidor, é possível verificar que ambos são protegidos de concorrência de threads (syncronized), e que existe uma determinada tolerância de produção em *buffer* cheio ou de consumo em *buffer* vazio. Antes de um erro ser informado, o *buffer* irá bloquear a chamada, esperando que alguma situação possa alterar o estado do *buffer* e permitir que a chamada se concretize sem erro, caso isso não ocorra dentro de um dado período de tempo um erro indicando a situação do *buffer* é repassado para o cliente.

4. Solução Implementada

A solução implementada consiste de uma aplicativo visual para instanciação dos elementos envolvidos. A execução da aplicação apresenta ma constante troca de mensagens entre o *buffer* e seus clientes produtores e consumidores. A figura 2 apresenta de forma geral a troca de mensagens entre os elementos envolvidos no problema.

4.1. Linguagem

A linguagem de programação utilizada foi a Linguagem Java, por possui varios recursos nativos que são interessantes ao problema apresentado. A vesrão e o Kit de Desenvolvimento especifico utilizado foi o JAVA SE JDK 7.

4.2. Java RMI

A abordagem distribuída utilizada é a cliente servidor com Objetos Distribídos, com o uso de Invocação Remota de Metódos, conhecido RMI. A implementação utilizada é a implementação nativa da linguagem Java, o Java RMI [Corporation a].

4.3. *Metódos Sincronizados*

A parte do sistema que realiza operações sobre os contadores do *buffer* e sua estrutura interna de armazenamento está protegida do acesso concorrente de threads, com o uso dos metódos sincronizados (Synchronized Methods), um recurso nativo da linguagem que facilita a programação concorrente com compartilhamento de recursos [Corporation b].

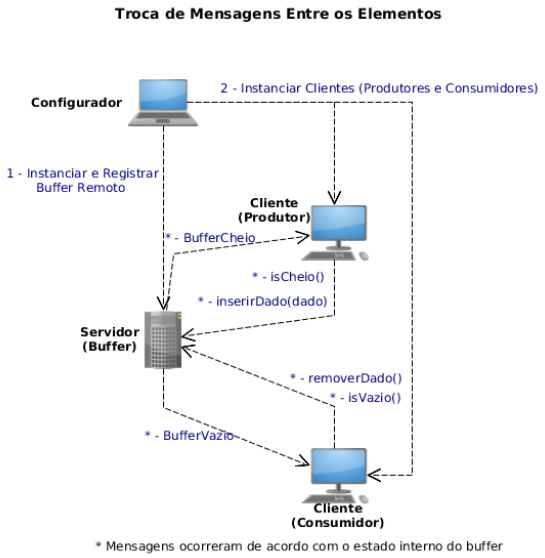

Figura 2. Troca de mensagens entre os elementos.

5. Testes Realizados

Para realizar os teste o ambiente precisa ser executado, uma copia da implementação ”produtor-consumidor-distribuido-1.0.jar“ deve ser feita em cada host que irá executar o projeto. Após isso o ”rmiregistry“ deve ser executado nos hosts que irão abrigar os objetos remotos, conforme comando na figura 3. Com o rmiregistry devidamente executado em 3 hosts. Deve se executar a aplicação de configuração conforme comando em figura 4. Com isso o ambiente está pronto para executar os testes.

```
$ rmiregistry -J-cp -J"produtor-consumidor-distribuido-1.0.jar"
```

Figura 3. Executando rmiregistry.

```
$ java -jar produtor-consumidor-distribuido-1.0.jar
```

Figura 4. Executando a aplicação de configuração.

5.1. Teste Simples

Em um teste simples, um *buffer*, um produtor e um consumidor devem ser instanciados para que possam, interagir entre si com pouca concorrência, apenas para verificar se o ambiente está corretamente em execução. Para instanciar o nome do host onde deseja-se instanciar deve ser preenchido, o elemento a ser instanciado (buffer, produtor ou consumidor) e em seguida clicar em instanciar, conforme figura 5

Na figura 6 é possível ver o que será mostrado no console da aplicação de configuração, durante a execução do teste simples, o produtor e consumidor inserindo e removendo informações no *buffer* praticamente na mesma cadência, dessa forma o *buffer* dificilmente ficará cheio ou vazio. Porém é possível constatar que o ambiente está operacional.

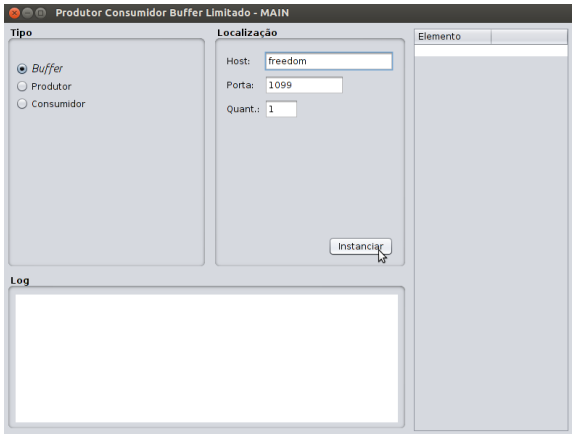

Figura 5. Tela de aplicação de configuração

```
buffer[16]       = A P N T O Y H C S h A q j A K W
>produtor/0 inserindo b na posição 16
buffer[17]       = A P N T O Y H C S h A q j A K W b
>produtor/0 inserindo K na posição 17
buffer[18]       = A P N T O Y H C S h A q j A K W b K
>produtor/0 inserindo p na posição 18
buffer[19]       = A P N T O Y H C S h A q j A K W b K p
>produtor/0 inserindo H na posição 19
buffer[20]       = A P N T O Y H C S h A q j A K W b K p H
<consumidor/0 retirando         da posição 20
buffer[19]       = A P N T O Y H C S h A q j A K W b K p
<consumidor/0 retirando H       da posição 19
buffer[18]       = A P N T O Y H C S h A q j A K W b K
<consumidor/0 retirando p       da posição 18
buffer[17]       = A P N T O Y H C S h A q j A K W b
<consumidor/0 retirando K       da posição 17
buffer[16]       = A P N T O Y H C S h A q j A K W
>produtor/0 inserindo y na posição 16
buffer[17]       = A P N T O Y H C S h A q j A K W y
>produtor/0 inserindo m na posição 17
buffer[18]       = A P N T O Y H C S h A q j A K W y m
<consumidor/0 retirando p       da posição 18
buffer[17]       = A P N T O Y H C S h A q j A K W y
```

Figura 6. Execução de teste simples

5.2. Teste de Consumo Intensivo

Um teste de consumo intensivo foi realizado, instanciando um *buffer*, um produtor e cinquenta consumidores, a concorrência no consumo é acirrada, na figura 7. Com cinquenta vezes mais consumidores que produtores, o *buffer* precisa tratar intensamente com a concorrência dos dados produzidos, bloqueando as chamadas de consumo e aguardando produções, que são imediatamente consumidas após a inserção no *buffer*, com esse teste é possível constatar que aplicação é tolerante a concorrência de consumo e evita falhas excessivas de *BufferVazio* nos clientes consumidores, através de uma espera das chamadas.

Na figura 8 é possível ver o que será mostrado no console da aplicação de configuração, durante a execução do teste de consumo intensivo, o produtor único inserindo informação no *buffer* e os consumidores tentando consumir, porem a grande maioria fica aguardando e a maior parte do tempo o *buffer* estará vazio.

5.3. Teste de Produção Intensiva

Um teste de produção intensiva foi realizado, instanciando um *buffer*, um consumidor e cinquenta produtores, em pouco tempo o *buffer* fica cheio e a concorrência na produção fica acirrada, na figura 9. Com cinquenta vezes mais produtores que consumidores, o *buffer* precisa tratar intensamente com a concorrência dos dados produzidos, bloqueando

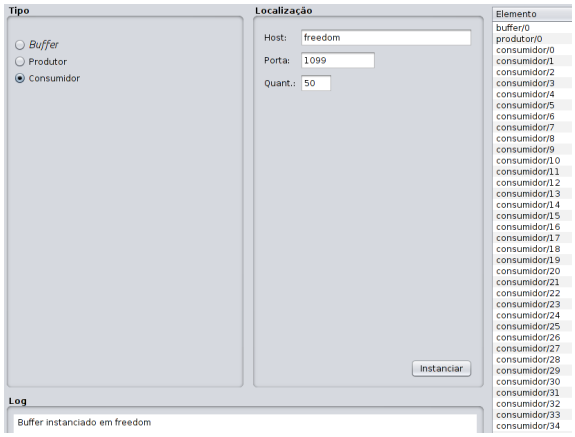

Figura 7. Instanciar um teste de consumo itensivo.

```
=consumidor/32 buffer vazio, esperando produção
=consumidor/42 buffer vazio, esperando produção
=consumidor/1 buffer vazio, esperando produção
=consumidor/11 buffer vazio, esperando produção
=consumidor/38 buffer vazio, esperando produção
>produtor/0 inserindo y na posição 0
buffer[1]        = y
<consumidor/8 retirando y       da posição 0
buffer[0]        =
=consumidor/27 buffer vazio, esperando produção
=consumidor/14 buffer vazio, esperando produção
=consumidor/42 buffer vazio, esperando produção
=consumidor/30 buffer vazio, esperando produção
=consumidor/38 buffer vazio, esperando produção
=consumidor/24 buffer vazio, esperando produção
=consumidor/39 buffer vazio, esperando produção
=consumidor/38 buffer vazio, esperando produção
=consumidor/2 buffer vazio, esperando produção
=consumidor/6 buffer vazio, esperando produção
=consumidor/4 buffer vazio, esperando produção
=consumidor/7 buffer vazio, esperando produção
=consumidor/41 buffer vazio, esperando produção
=consumidor/44 buffer vazio, esperando produção
```

Figura 8. Execução de um teste de consumo intensivo.

as chamadas de produção quando o *buffer* está cheio e aguardando novos consumos, com esse teste é possível constatar que aplicação é tolerante a concorrência de produção e evita o estouro do *buffer*, e evitando falhas excessivas de *BufferCheio* nos produtores, através de uma espera das chamadas.

Na figura 10 e 11 é possível ver o que será mostrado no console da aplicação de configuração, durante a execução do teste de produção intensiva, o único consumidor retira informações do *buffer* em uma cadência muito menor do que os cinquenta produtores inserem, na maior parte do tempo o *buffer* estará cheio, e os produtores estarão bloqueados aguardando liberação de espaço no *buffer*.

6. Conclusões

Mesmo utilizando um simples controle de concorrência é possível distribuír o problema do produtor e consumidor, sem que as aplicações clientes precisem tratar com grande complexidade de concorrência, uma implementação adequada do *buffer* com o correto controle de concorrência de acesso e sinalização de estado é capaz de produzir um resul-

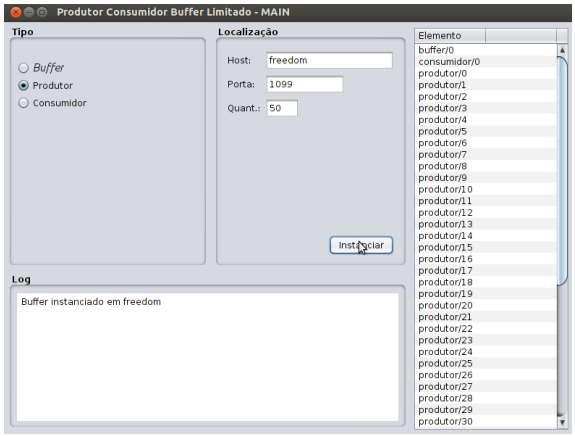

Figura 9. Instanciar um teste de produção intensiva

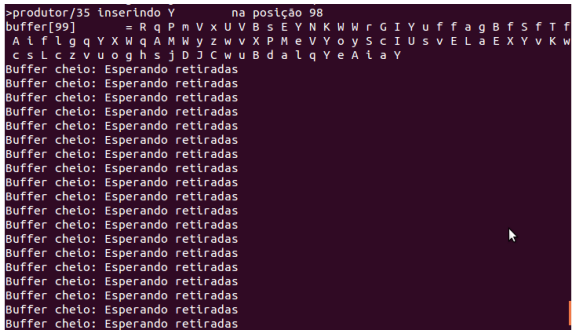

Figura 10. Execução do teste de produção intensiva - Parte 1.

tado satisfatório.

Referências

Corporation, O. Java remote method invocation api - java se documentation. `http://docs.oracle.com/javase/7/docs/technotes/guides/rmi`. Acessado: 2015-05-01.

Corporation, O. Synchronized methods - the java tutorials. `https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html`. Acessado: 2015-05-01.

Toscani, S., de Oliveira, R., and da Silva Carissimi, A. (2003). *Sistemas operacionais e programação concorrente*. Série Livros Didáticos. Sagra Luzzatto.

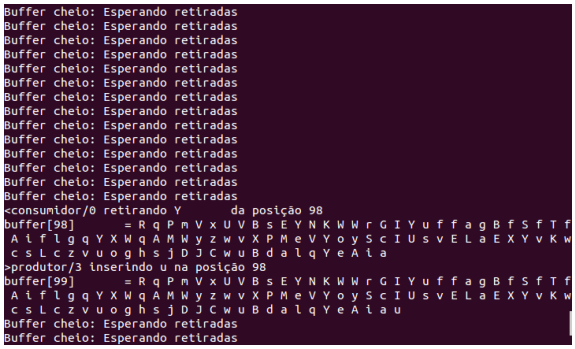

Figura 11. Execução do teste de produção intensiva - Parte 2.