UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E APLICADAS - ICEA

GILDO TIAGO AZEVEDO

TRABALHO PRÁTICO 02

Implementação do Algoritmo de Berkeley e Algoritmo de Eleição

João Monlevade - MG

2021

GILDO TIAGO AZEVEDO

TRABALHO PRÁTICO 02

Implementação do Algoritmo de Berkeley e Algoritmo de Eleição

Trabalho apresentado como requisito parcial para aprovação na disciplina CSI302 - Sistemas Distribuídos pela Universidade Federal de Ouro Preto - campus João Monlevade.

Prof.: D.r Theo Silva Lins

João Monlevade - MG

2021

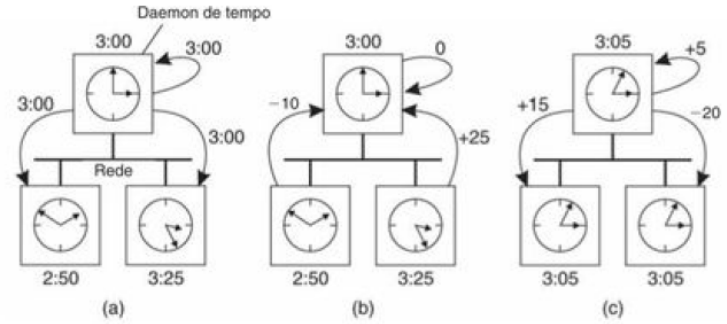
DESENVOLVIMENTO

Conforme proposto pelo enunciado do trabalho prático 02 da disciplina Sistemas Distribuídos, o objetivo é implementar o algoritmo de Berkeley e um algoritmo de eleição de líder, e para isto foi utilizado a linguagem JAVA. Para este trabalho, foi implementado um exemplo simples para demonstrar o algoritmo de Berkeley utilizando 4 clientes em uma estrutura RMI, e um algoritmo de anel utilizando uma estrutura LinkedList e multi thread para execução contínua de processos.

O código fonte implementado pode ser acessado/clonado no github pelo link: https://github.com/Azevor/Atividades_CSI302.git.

Algoritmo de Berkeley

O algoritmo de Berkeley foi desenvolvido com base na estrutura Java RMI, onde cada cliente, ao ser executado, entra no grupo de sincronização do relógio definido pelo servidor remoto. Seguindo a linha de raciocínio do algoritmo de Berkeley (figura 1) [TANENBAUM, pg. 146], sendo demonstrado em uma implementação simples, inicialmente houve uma dificuldade em conectar mais de um servidor para que cada um recebesse resposta (ajuste do relógio) individualmente. Este problema foi resolvido através da adaptação de código onde uma estrutura remota realiza chamada intermediária do servidor aos diversos clientes por meio de um método remoto [2].



Inicialmente, o método remoto que será executado pelos clientes para atualização dos relógios são declarados na interface ServicoListener que estende de Remote:

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

/**
 * Interface remota que permite chamada de método que atualiza os relógios dos clientes.
 * @author Gildo Tiago
 *
 */
public interface ServicoListener extends Remote {
    void atualizarRelogio(int resultado) throws RemoteException;
}
```

Para execução do lado servidor, foi configurado 4 relógios, para efeito de demonstração, e seus métodos são declarados na interface remota a seguir:

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

/**
 * Interface remota para comunicação dos clientes com o servidor remoto.
 * @author Gildo Tiago
 *
 */
public interface Servico extends Remote {
    void addListener(ServicoListener listener, int valorRelogio) throws RemoteException;
    void setRelogio1() throws RemoteException;
    void setRelogio2() throws RemoteException;
    void setRelogio3() throws RemoteException;
    void setRelogio4() throws RemoteException;
}
```

Sendo esta interface implementada pela classe do lado servidor:

```java
import java.rmi.RemoteException;
import java.text.DecimalFormat;
import java.util.HashMap;

/**
 * Implementação do objeto remoto que recebe a comunicação dos clientes.
 * @author Gildo Tiago
 *
 */
public class ImplementacaoServico implements Servico {
    /*
     * HashMap que adiciona como chave a interface cliente (proxy) e
     * armazena sua hora como respectivo valor.
     */
    private HashMap<ServicoListener, Integer> listeners = new HashMap<>();

    /* Definição de alguns valores para teste de sincronização dos relógios. */
    private int horas = 3;
    private int minutos = 15;
    private int valorRelogioLocal = (horas * 60 + minutos);
    private DecimalFormat df = new DecimalFormat("00"); // Exibir dois dígitos da hora.
    private int mediaDiferencas = 0; // Armazenar a média das diferenças de horário servidor-clientes.

    /* Verificar se armazenou a hora dos respectivos clientes. */
    private boolean setouRelogio1;
    private boolean setouRelogio2;
    private boolean setouRelogio3;
    private boolean setouRelogio4;
```

No código acima, a estrutura de HashMap armazena todos os clientes utilizando seus respectivos proxies como chave e seu relógio como valor do hash. Em seguida é atribuído um valor arbitrário para seu horário local (servidor). Uma variável booleana é criada para cada cliente configurado para verificar se houve recebimento do seu respectivo horário local (cliente).

```java
/* Método remoto que recebe o proxy cliente conectado e sua respectiva hora. */
@Override
public void addListener(ServicoListener listener, int valorRelogio) throws RemoteException {
    listeners.put(listener, valorRelogio);
    // Exibindo a hora local do servidor.
    System.out.println("Hora atual do servidor: " + df.format(valorRelogioLocal/60) + ":"
                    + df.format(valorRelogioLocal%60));
}

/*
 * Configuração feita para receber 4 clientes, neste trabalho prático.
 */

/* Método remoto que o cliente utiliza para setar (confirmar informação de hora) */
@Override
public void setRelogio1() throws RemoteException { // Cliente A
    setouRelogio1 = true;
    verifica();
}
```

Logo em seguida o método remoto implementado pelo servidor insere a chave hash e seu respectivo valor (hora local do cliente) em uma estrutura, e o servidor informa seu horário local para cada cliente conectado. A seguir, a implementação dos métodos remotos que confirmam o recebimento do horário local de cada cliente (método invocado pelo cliente).

```java
/*
 * Configuração feita para receber 4 clientes, neste trabalho prático.
 */

/* Método remoto que o cliente utiliza para setar (confirmar informação de hora) */
@Override
public void setRelogio1() throws RemoteException { // Cliente A
    setouRelogio1 = true;
    verifica();
}

@Override
public void setRelogio2() throws RemoteException { // Cliente B
    setouRelogio2 = true;
    verifica();
}

@Override
public void setRelogio3() throws RemoteException { // Cliente C
    setouRelogio3 = true;
    verifica();
}

@Override
public void setRelogio4() throws RemoteException { // Cliente D
    setouRelogio4 = true;
    verifica();
}
```

O método a seguir calcula a diferença de horário entre o servidor e os clientes conectados nele:

```java
    /**
     * Calcular diferença de horario entre o servidor e os clientes.
     * @param HashMap<Cliente, Horario> : clientes
     * @return HashMap<Cliente, Diferença_Horario> : diferencas
     */
    private HashMap<ServicoListener, Integer> sincronizar(HashMap<ServicoListener, Integer> clientes) {
        HashMap<ServicoListener, Integer> diferencas = new HashMap<>();
        int totalDiferencas = 0;
        /*
         * De acordo com o algoritmo de Berkeley, para cada horario do cliente,
         * será calculado a diferença entre o servidor e o cliente.
         */
        for (ServicoListener keyListener : clientes.keySet()) {
            diferencas.put(keyListener, clientes.get(keyListener)-valorRelogioLocal);
            totalDiferencas += clientes.get(keyListener)-valorRelogioLocal;
        }
        // Cálculo da média das diferenças de horário do cliente.
        mediaDiferencas = totalDiferencas/(1+diferencas.size()); // Soma-se 1 para contar com o servidor na média.
        return diferencas;
    }
```

Esta diferença é armazenada em uma estrutura de hash local, similar a hash dos horários dos clientes, entretanto este armazena a diferença entre o relógio cliente e o relógio do servidor, e esta diferença é acumulada em uma variável para posterior cálculo da média, que é calculada e armazenada numa variável global “mediaDiferencas”.

```java
    /**
     * Método local que verifica se todos os clientes conectados informaram suas respectivas horas.
     */
    private void verifica() {
        // Caso o servidor esteja ciente de todos os horários, realiza-se a sincronização dos horários.
        if (setouRelogio1 && setouRelogio2 && setouRelogio3 && setouRelogio4) {
            HashMap<ServicoListener, Integer> diferencas = sincronizar(listeners);
            System.out.println("Média calculada = " + mediaDiferencas + " minutos");
            valorRelogioLocal += mediaDiferencas;
            // Relógio do servidor recebe a média das diferenças entre os relógios e exibe seu novo horário.
            System.out.println("Ajuste do servidor: " + df.format(valorRelogioLocal/60) + ":"
                               + df.format(valorRelogioLocal%60));

            // Todo cliente conectado receberá a diferença, pra mais ou pra menos, de ajuste do relógio.
            for (ServicoListener keyListener : listeners.keySet()) {
                int diferenca = mediaDiferencas-diferencas.get(keyListener);
                try {
                    keyListener.atualizarRelogio(diferenca);
                } catch (RemoteException e) {
                    e.printStackTrace();
                }
            }
            setouRelogio1 = false;
            setouRelogio2 = false;
            setouRelogio3 = false;
        }
    }
}
```

Por fim o servidor faz o ajuste de seu relógio local adicionando ao seu horário a média das diferenças calculada, verifica todos os relógios dos clientes, enviando para cada um deles o resultado, que é a quantidade em minutos para ajuste dos respectivos horários, através do método “atualizarRelogio(diferenca)”. O cálculo destes valores é feito subtraindo-se a média das diferenças da diferenca do respectivo cliente, em minutos.

O servidor está implementado conforme segue:

```java
import java.rmi.registry.LocateRegistry;

/**
 * Definição do servidor, aberto para conexão de clientes.
 * @author Gildo Tiago
 *
 */
public class Servidor {

    public static void main(String args[]) {
        try {
            String bind = "BerkeleyRMI"; // Nome do servico a ser conectado pelo cliente.
            int porta = 8242;

            Servico servico = new ImplementacaoServico();
            Servico servicoDistribuido = (Servico) UnicastRemoteObject.exportObject(servico, 0);

            Registry registry = LocateRegistry.createRegistry(porta); // Servico de registro de diretório.
            registry.bind(bind, servicoDistribuido); // Nomeação para os clientes encontrarem os servicos.
            System.out.printf("Servico disponivel: %s%n", bind);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Na classe Servidor o nome do serviço é registrado como “BerkeleyRMI” juntamente com a porta 8242, aguardando a conexão com qualquer cliente RMI que acesse este diretório nomeado.

A classe cliente foi implementada separadamente, para simular diferentes acessos ao servidor, invés de criar várias instâncias de um mesmo objeto cliente.

```java
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.text.DecimalFormat;

import Servidor.Servico;
import Servidor.ServicoListener;

/**
 * Implementação da classe cliente para acessar servicos do servidor.
 * @author Gildo Tiago
 *
 */
public class ClienteA implements ServicoListener {
    /*
     * Definição de alguns valores para testar a sincronização dos relógios.
    */
    private static final int horas = 3;
    private static final int minutos = 20;
    private static final int valorRelogioLocal = (horas * 60 + minutos);
    private static DecimalFormat df = new DecimalFormat("00");
```

O “ClienteA” implementa a interface “ServiçoListener” apresentado anteriormente, como forma de “intermediar” o acesso ao servidor remoto, permitindo que o servidor acesse o método que irá notificar o cliente sobre o valor do ajuste a ser executado em seu horário local.

```java
    /**
     * Método remoto acessado pelo servidor para receber o ajuste do relógio local.
     */
    @Override
    public void atualizarRelogio(int resultado) throws RemoteException {
        // Ao receber a quantidade, em minutos, este valor é adicionado ao relógio local, pra mais ou pra menos.
        System.out.println("Servidor retornou para o Cliente A: " + resultado + " minutos");
        int ajusteRelogioLocal = valorRelogioLocal+resultado;
        // Exibe a hora ajustada na tela.
        System.out.println("Ajuste do Cliente A: " + df.format(ajusteRelogioLocal/60) + ":"
                           + df.format(ajusteRelogioLocal%60));
    }
```

Ao ser executado, o método “atualizarRelogio(int resultado)” recebe a quantidade, em minutos, do ajuste a ser feito no relógio do cliente, sendo este negativo ou positivo, atrasando ou adiantando seu relógio local.

```java
    public static void main(String[] args) {
        try {
            String nomeServico = "BerkeleyRMI"; // Nome do servico registrado no servidor.
            int porta = 8242;

            ServicoListener clienteA = new ClienteA();
            ServicoListener clienteAdistribuido = (ServicoListener) UnicastRemoteObject.exportObject(clienteA, 0);

            // Requisição de registro pelo cliente para acessar o servico do servidor.
            Registry registry = LocateRegistry.getRegistry(porta);
            Servico servicoRemoto = (Servico) registry.lookup(nomeServico);

            // Acesso ao método remoto que adiciona o proxy cliente no servidor, e a hora local.
            servicoRemoto.addListener(clienteAdistribuido, valorRelogioLocal);

            System.out.println("Relogio do Cliente A: " + df.format(horas) + ":" + df.format(minutos));
            servicoRemoto.setRelogio1();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Finalmente o método Main, ao ser executado fará a chamada remota pelo serviço e porta registrados pelo bind no lado servidor através do lookup, enviando sua instância (proxy) de ServiçoListener e seu respectivo horário local para o servidor. Os demais clientes foram implementados no mesmo formato, diferindo somente sua identificação (nome da classe e demais variáveis):

```java
public class ClienteB implements ServicoListener {

    /*
     * Os comentários realizados na classe Cliente B são análogos nesta.
     */

    private static final int horas = 3;
    private static final int minutos = 5;
    private static final int valorRelogioLocal = (horas * 60 + minutos);
    private static DecimalFormat df = new DecimalFormat("00");

    public static void main(String[] args) {
        try {
            String nomeServico = "BerkeleyRMI";
            int porta = 8242;

            ServicoListener clienteB = new ClienteB();
            ServicoListener clienteBdistribuido = (ServicoListener) UnicastRemoteObject.exportObject(clienteB, 0);

            Registry registry = LocateRegistry.getRegistry(porta);
            Servico servicoRemoto = (Servico) registry.lookup(nomeServico);

            servicoRemoto.addListener(clienteBdistribuido, valorRelogioLocal);

            System.out.println("Relogio do Cliente B: " + df.format(horas) + ":" + df.format(minutos));
            servicoRemoto.setRelogio2();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```java
public class ClienteC implements ServicoListener {

    /*
     * Os comentários realizados na classe Cliente C são análogos nesta.
     */

    private static final int horas = 3;
    private static final int minutos = 25;
    private static final int valorRelogioLocal = (horas * 60 + minutos);
    private static DecimalFormat df = new DecimalFormat("00");

    public static void main(String[] args) {
        try {
            String nomeServico = "BerkeleyRMI";
            int porta = 8242;

            ServicoListener clienteC = new ClienteC();
            ServicoListener clienteCdistribuido = (ServicoListener) UnicastRemoteObject.exportObject(clienteC, 0);

            Registry registry = LocateRegistry.getRegistry(porta);
            Servico servicoRemoto = (Servico) registry.lookup(nomeServico);

            servicoRemoto.addListener(clienteCdistribuido, valorRelogioLocal);

            System.out.println("Relogio do Cliente C: " + df.format(horas) + ":" + df.format(minutos));
            servicoRemoto.setRelogio3();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

```java
public class ClienteD implements ServicoListener {

    /*
     * Os comentários realizados na classe Cliente D são análogos nesta.
     */

    private static final int horas = 2;
    private static final int minutos = 50;
    private static final int valorRelogioLocal = (horas * 60 + minutos);
    private static DecimalFormat df = new DecimalFormat("00");

    public static void main(String[] args) {
        try {
            String nomeServico = "BerkeleyRMI";
            int porta = 8242;

            ServicoListener clienteD = new ClienteD();
            ServicoListener clienteDdistribuido = (ServicoListener) UnicastRemoteObject.exportObject(clienteD, 0);

            Registry registry = LocateRegistry.getRegistry(porta);
            Servico servicoRemoto = (Servico) registry.lookup(nomeServico);

            servicoRemoto.addListener(clienteDdistribuido, valorRelogioLocal);

            System.out.println("Relogio do Cliente D: " + df.format(horas) + ":" + df.format(minutos));
            servicoRemoto.setRelogio4();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Todas as classes clientes acima implementam o método “atualizarRelógio(int resultado)” como o “ClienteA”.

A execução do código se dá da seguinte forma:

Hora do servidor: 03:15 = 195 minutos → 0 para ele mesmo

Hora do cliente A: 03:20 = 200 minutos → +5 para o servidor

Hora do cliente B: 03:05 = 185 minutos → -10 para o servidor

Hora do cliente C: 03:25 = 205 minutos → +10 para o servidor

Hora do cliente D: 02:50 = 170 minutos → -25 para o servidor

Média das diferenças = (0+5-10+10-25)/5 = -4

* **Nova hora do servidor**: 195+(-4) = 191 = 03:11

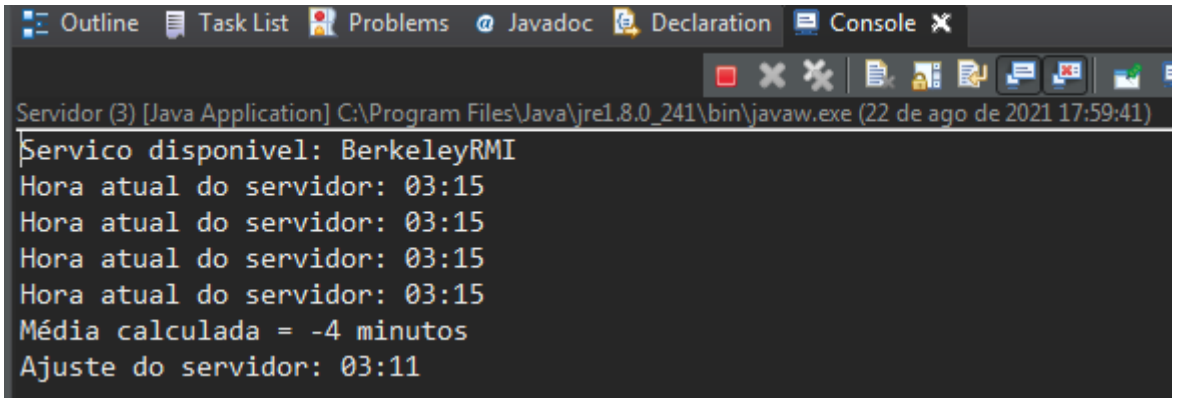
* **Cliente A recebe (-4-5):** 200+(-9) = 191 → nova hora = 03:11

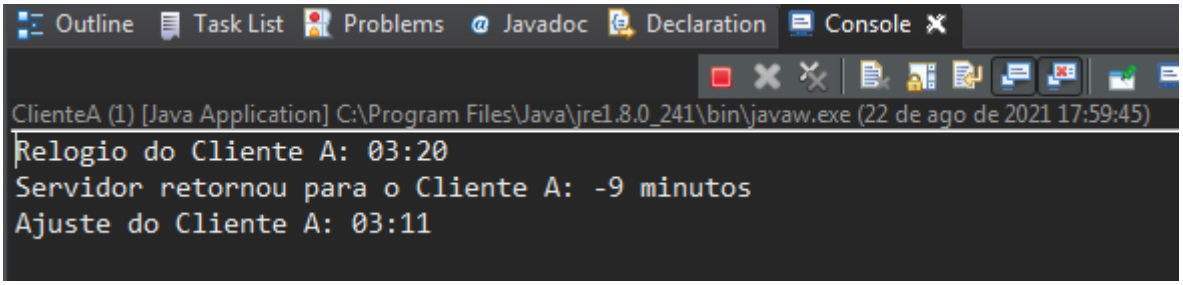
* **Cliente B recebe (-4-(-10))**: 185+(6) = 191 → nova hora = 03:11

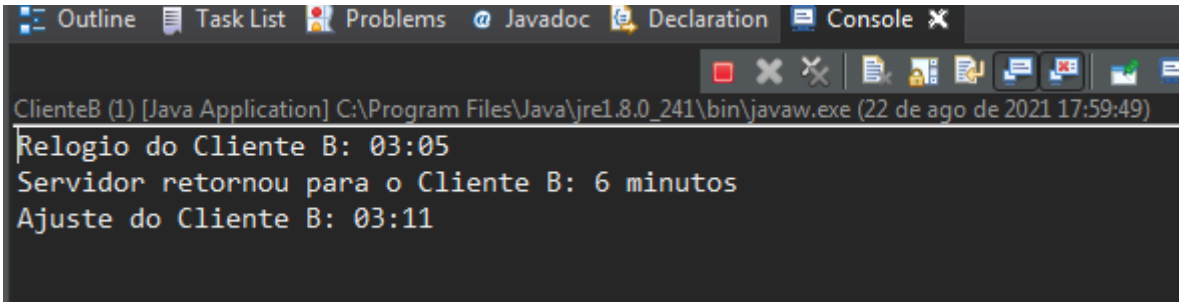
* **Cliente C recebe (-4-10)**: 205+(-14) = 191 → nova hora = 03:11

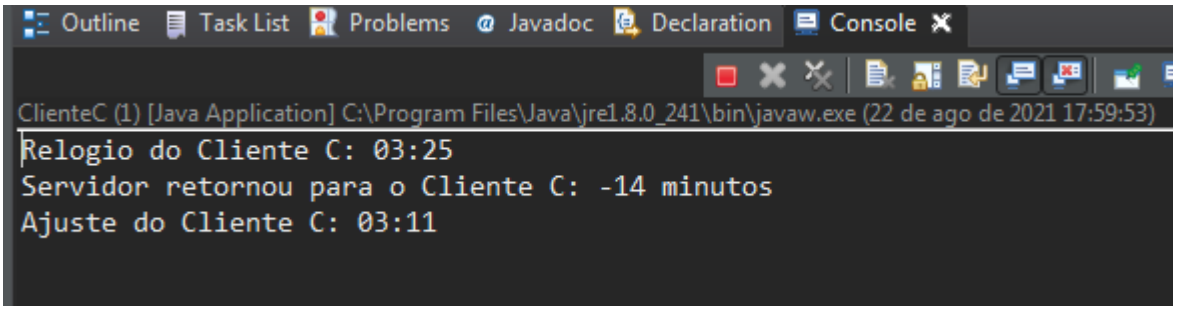
* **Cliente D recebe (-4-(-25))**: 170+(21) = 191 → nova hora = 03:11

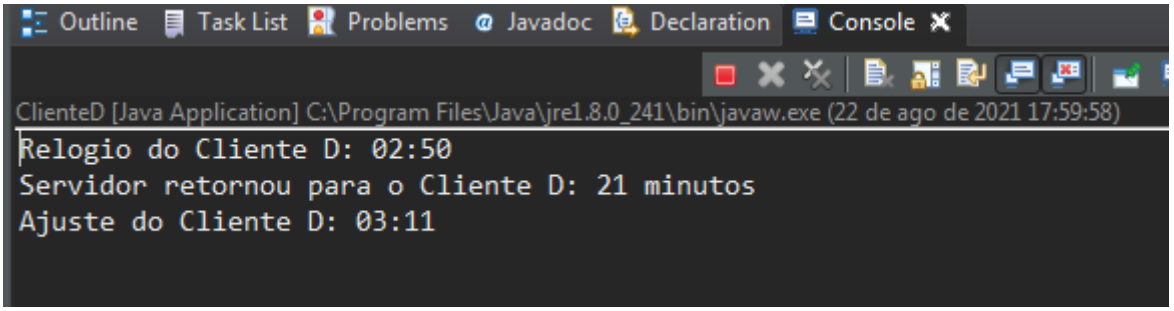
Resultados (servidor e respectivos clientes):







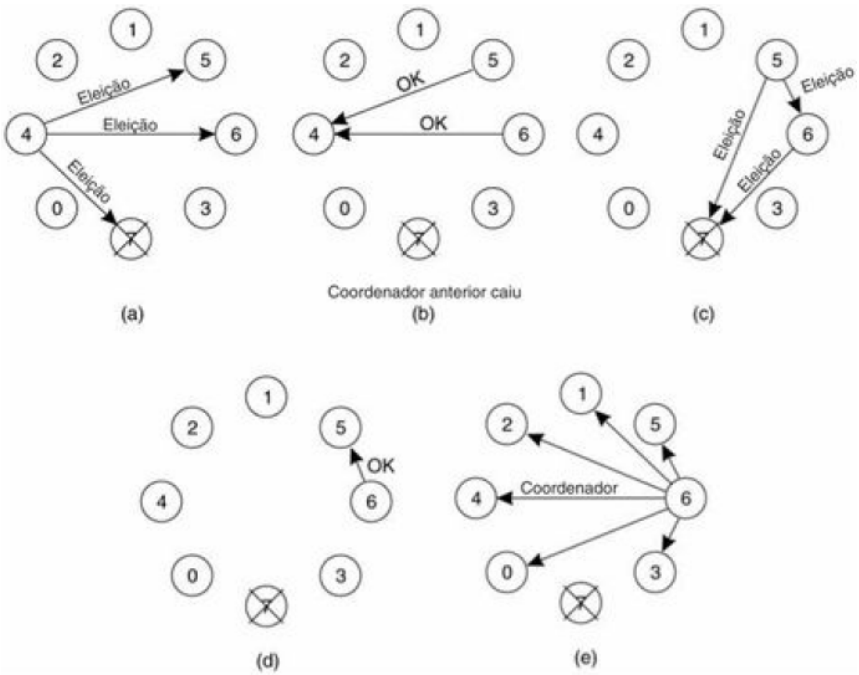




A execução demonstrou o correto funcionamento do algoritmo, ajustando os relógios dos clientes e do servidor de acordo com a diferença calculada entre os horários.

Algoritmo de Eleição

O algoritmo representado na implantação representa o algoritmo do anel, figura 2 [TANENBAUM, pg. 160].



Cada nó representa um processo no programa implementado, e cada processo recebe uma ID. O primeiro processo que entra na requisição é automaticamente marcado como coordenador. Caso uma requisição seja feita por um processo e não haja um coordenador ativo, uma nova eleição é executada e o processo de maior ID é selecionado como coordenador.

```java
import java.util.LinkedList;

/**
 * Gerenciar a atividade do processo identificado pelo id.
 *
 */
public class Processo {
    private int mPid;
    private boolean ehCoordenador;

    public Processo(int pPid, boolean pEhCoordenador) {
        setPid(pPid);
        setEhCoordenador(pEhCoordenador);
    }

    public int getPid() {
        return mPid;
    }

    public void setPid(int pPid) {
        this.mPid = pPid;
    }

    public boolean ehCoordenador() {
        return ehCoordenador;
    }

    public void setEhCoordenador(boolean pEhCoordenador) {
        this.ehCoordenador = pEhCoordenador;
    }
```

Para representar um processo, a classe “Processo” foi implementada e tem como identificador a variável id onde cada processo possui uma ID e uma variável booleana para identificar se ele é coordenador ou não.

```java
/**
 * Realizar requisição ou iniciar eleição, caso necessário.
 * @return boolean
 */
public boolean enviaRequisicao() {
    boolean requisicaoFeita = false;
    // Procura coordenador na lista de processos ativos (estática).
    for(Processo p: Anel.processosAtivos) {
        if(p.ehCoordenador()) {
            requisicaoFeita = p.recebeRequisicao(this.mPid);
        }
    }
    // Caso não exista um coordenador, fazer eleição.
    if(!requisicaoFeita) {
        this.novaEleicao();
    }
    System.out.println("Fim da requisicao");
    return requisicaoFeita;
}
```

O método acima simula a solicitação de execução de um processo. Este método procura por um coordenador na lista de processos ativos para receber a requisição, caso não encontre, uma nova eleição de coordenador é realizada.

```java
/**
 * Método responsável pelo tratamento da requisição.
 * @param pIdOrigemRequisicao
 * @return boolean
 */
private boolean recebeRequisicao(int pIdOrigemRequisicao) {
    /* Executa requisição ... */

    System.out.println("Requisição do processo ID: " + pIdOrigemRequisicao + " executada.");
    return true;
}
```

O método que recebe a requisição do coordenador apresenta uma resposta de execução, simulando um algoritmo de tratamento de requisições, retornando true para retornar como resposta de sucesso ao processo coordenador.

```java
/**
 * Seleciona novo coordenador através do processo de eleição.
 * O processo de maior ID será selecionado como coordenador.
 */
private void novaEleicao() {
    System.out.println("Eleição iniciada");
    LinkedList<Integer> idListaParaEleicao = new LinkedList<>();
    // Consultar processos ativos e adiciona-los numa nova lista.
    for(Processo p : Anel.processosAtivos) {
        p.verificaProcesso(idListaParaEleicao);
    }
    // Procurando pelo maior ID
    int idNovoCoordenador = this.getPid();
    for(Integer id : idListaParaEleicao) {
        if(id > idNovoCoordenador) {
            idNovoCoordenador = id;
        }
    }
    // Atualiza novo coordenador
    boolean ehCoordenadorValido = false;
    ehCoordenadorValido = atualizarCoordenador(idNovoCoordenador);
    if(ehCoordenadorValido) {
        System.out.println("Eleição concluída. Novo coordenador - processo ID: " + idNovoCoordenador + ".");
    } else {
        System.out.println("Eleição Falhou! Novo coordenador não encontrado.");
        // Em caso de falha, uma nova eleição será feita pela próxima requisição.
    }
}
```

Caso seja realizada uma nova eleição, o método acima irá adicionar a uma lista de candidatos a id de todos os processos válidos para participar da eleição. Nesta simulação, todos os processos ativos irão entrar na lista de candidatos. Por simplicidade, é feita apenas a comparação de ID 's para atribuição de coordenador ao processo de maior ID. Caso haja falha na eleição e nenhum coordenador seja selecionado, uma nova eleição acontece normalmente ao ser realizada uma nova requisição por um processo qualquer.

```java
/**
 * Executar validação do processo para indicação na lista de eleição.
 * @param ListaEleitoresValidos
 */
private void verificaProcesso(LinkedList<Integer> ListaEleitoresValidos) {
    /* Executa validação do processo ... */

    ListaEleitoresValidos.add(this.getPid());
}
```

Durante a eleição, o processo é verificado para saber se é um processo válido para se candidatar como coordenador. Este método é ilustrativo, bem como o método de execução da requisição.

```java
    /**
     * Atribuir ou atualizar um processo como coordenador.
     * @param pIdNovoCoordenador
     * @return boolean
     */
    private boolean atualizarCoordenador(int pIdNovoCoordenador) {
        for(Processo p : Anel.processosAtivos) {
            if(p.getPid() == pIdNovoCoordenador) {
                p.setEhCoordenador(true);
            } else {
                // Garantir que todos os processos restantes não sejam coordenadores.
                p.setEhCoordenador(false);
            }
        }
        return true;
    }
}
```

Finalmente, o coordenador é atualizado após a eleição, e para garantir que apenas um processo seja coordenador, o loop seleciona todos os outros processos como não-coordenadores.

```java
import java.util.ArrayList;
import java.util.Random;

/**
 * Controlar as especificações do processo de eleição.
 * Cada método implementa uma Thread que realiza as operações em paralelo.
 *
 */
public class Anel {
    private final int LOOP_NOVO_PROCESSO = 4000;
    private final int LOOP_NOVA_REQUISICAO = 3000;
    private final int LOOP_INATIVAR_PROCESSO = 8000;
    private final int LOOP_INATIVAR_COORDENADOR = 12000;

    public static ArrayList<Processo> processosAtivos;
    /*
     * Objeto genrérico para controlar a sincronização de execução das threads,
     * evitando execução de processo de eleição concorrentemente.
     */
    private final Object objControle = new Object();

    public Anel() {
        processosAtivos = new ArrayList<Processo>();
    }
```

A classe “Anel” é responsável por executar as threads que farão a simulação de execução dos processos. Esta classe possui um tempo arbitrário estipulado para cada execução de uma thread (implementada nos respectivos métodos). Foi setado 4 segundos para a criação de um novo processo, 3 segundos para que um processo aleatório faça uma requisição, 8 segundos para deixar um processo aleatório inativo e 12 segundos para inativar o coordenador.

Um objeto genérico foi instanciado para ser usado no controle das multithreads, evitando que duas requisições sejam feitas simultaneamente, por exemplo [3]. Existe uma variável estática que utiliza uma LinkedList para simular a cadeia de nós dos processos em anel, onde toda execução é feita (inclusão/remoção de processo e busca).

```java
/**
 * Cria um novo processo periodicamente.
 */
public void novoProcesso() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            while(true) {
                synchronized(objControle) {
                    // Verificar se existe processo criado anteriormente.
                    if(processosAtivos.isEmpty()) {
                        // Caso não exista, o primeiro processo é criado como coordenador.
                        processosAtivos.add(new Processo(1, true));
                    } else {
                        // Caso contrário, apenas adiciona um novo processo.
                        processosAtivos.add(new Processo(
                                processosAtivos.get(processosAtivos.size() - 1).getPid() + 1, false));
                    }
                    System.out.println("Processo ID: " +
                            (processosAtivos.get(processosAtivos.size() - 1).getPid()) + " criado.");
                }
                try {
                    // Um novo processo será criado após o tempo especificado abaixo.
                    Thread.sleep(LOOP_NOVO_PROCESSO);
                } catch(Exception e) {
                    System.err.println("Erro em Anel->novoProcessos(): " + e.getMessage() + "\n--");
                    e.printStackTrace();
                }
            }
        }
    }).start();
}
```

A primeira thread se trata da criação de um novo processo a cada tempo (armazenado em LOOP_NOVO_PROCESSO), ao ser executada indefinidamente pelo bloco "*while(true)*" verifica se já existe um processo criado anteriormente, caso negativo significa que é o primeiro processo a ser criado e este é setado como coordenador pela chamada "*processosAtivos.add(new Processo(1, true))*", caso não seja o primeiro processo, um novo processo não-coordenador é adicionado à LinkedList em "*processosAtivos.add(new Processo(processosAtivos.get(processosAtivos.size()-1) .getPid()+1, false))*". Esta execução atribui o próximo ID ao novo processo criado, pois passa como parâmetro o tamanho da lista subtraído de 1 (para buscar o index da posição em que se encontra o último processo armazenado) e adiciona 1 ao ID do mesmo.

```java
/**
 * A cada período especificado, uma nova requisição é realizada.
 */
public void novaRequisicao() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            while(true) {
                try {
                    // Após o tempo especificado abaixo, um processo  fará requisições periodicamente.
                    Thread.sleep(LOOP_NOVA_REQUISICAO);
                } catch(Exception e) {
                    System.err.println("Erro em Anel->fazRequisicoes(): " + e.getMessage() + "\n--");
                    e.printStackTrace();
                }
                synchronized(objControle) {
                    // Caso exista processo ativo na lista de processos, uma requisição será feita.
                    if(processosAtivos.size() > 0) {
                        int idProcessoAleatorio = new Random().nextInt(processosAtivos.size());
                        // Recuperando um processo aleatório para fazer requisição.
                        Processo requisicaoDeProcesso = processosAtivos.get(idProcessoAleatorio);
                        System.out.println("Processo ID: " + requisicaoDeProcesso.getPid() + " fazendo requisição.");
                        requisicaoDeProcesso.enviaRequisicao();
                    }
                }
            }
        }
    }).start();
}
```

Uma nova requisição é feita de tempos em tempos, onde é selecionado um processo aleatório “*random com o tamanho da lista de processos ativos*”. Caso a lista de processos ativos esteja vazia, nenhuma ação é executada e o método aguarda novamente antes de solicitar uma nova requisição.

```java
/**
 * Inativa um processo aleatório periodicamente.
 */
public void inativarProcesso() {
    new Thread(new Runnable() {
        @Override
        public void run() {
            while(true) {
                try {
                    // Após o tempo especificado abaixo, um processo  será inativado periodicamente.
                    Thread.sleep(LOOP_INATIVAR_PROCESSO);
                } catch(Exception e) {
                    System.err.println("Erro em Anel->inativaProcessos(): " + e.getMessage() + "\n--");
                    e.printStackTrace();
                }
                synchronized(objControle) {
                    // Um processo aleatório será inativado apenas se a lista conter algum processo ativo.
                    if(!processosAtivos.isEmpty()) {
                        int idProcessoAleatorio = new Random().nextInt(processosAtivos.size());
                        Processo processoParaRemover = processosAtivos.get(idProcessoAleatorio);
                        // O processo, se válido, ficará inativo caso não seja coordenador.
                        if(processoParaRemover != null && !processoParaRemover.ehCoordenador()) {
                            processosAtivos.remove(processoParaRemover);
                            System.out.println("Processo ID: " + processoParaRemover.getPid() + " inativado.");
                        }
                    }
                }
            }
        }
    }).start();
}
```

De maneira similar a nova requisição, um processo aleatório é inativado pelo método acima, caso exista pelo menos um processo ativo na lista e não seja coordenador.

```java
    /**
     * Um coordenador aleatório é inativado.
     */
    public void inativarCoordenador() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                while(true) {
                    try {
                        // Após o tempo especificado abaixo, um coordenador será inativado periodicamente.
                        Thread.sleep(LOOP_INATIVAR_COORDENADOR);
                    } catch(Exception e) {
                        System.err.println("Erro em Anel->inativaCoordenador(): " + e.getMessage() + "\n--");
                        e.printStackTrace();
                    }
                    synchronized(objControle) {
                        Processo coordenador = null;
                        // Percorre lista de processos ativos procurando pelo coordenador.
                        for(Processo p : processosAtivos) {
                            if(p.ehCoordenador()) {
                                coordenador = p;
                            }
                        }
                        // Caso encontre o coordenador (not null) proceder a remoção da lista de processos ativos.
                        if(coordenador != null) {
                            processosAtivos.remove(coordenador);
                            System.out.println("Processo coordenador ID: " + coordenador.getPid() + " inativado.");
                        }
                    }
                }
            }
        }).start();
    }
}
```

E periodicamente um coordenador é inativado também, neste caso é realizada uma busca pelo coordenador na LinkedList e o mesmo é removido, caso seja encontrado. Caso não exista um coordenador na lista nenhuma ação é executada, e uma nova eleição será feita caso seja executada uma nova requisição.

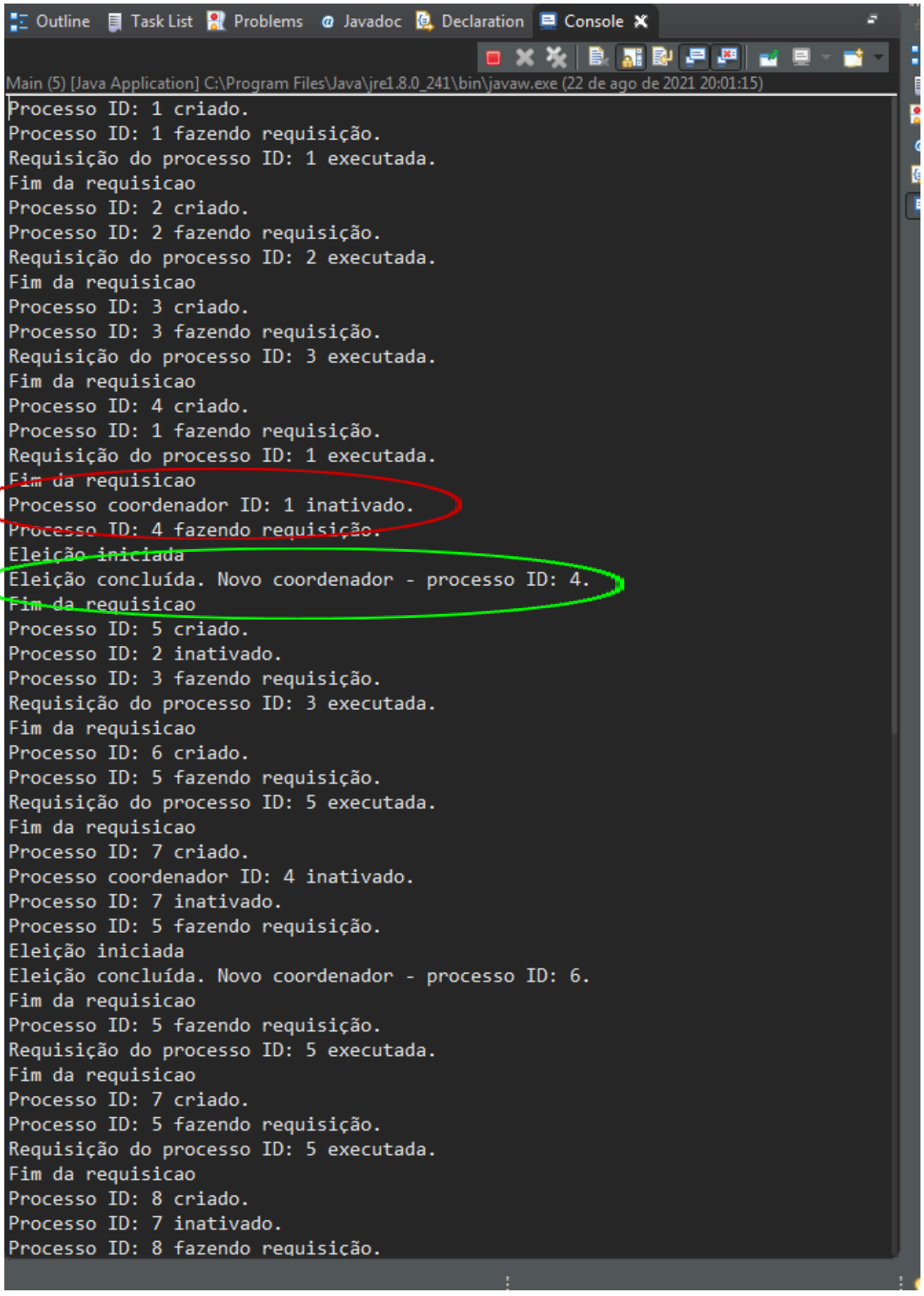
```java
public class Main {

    public static void main(String[] args) {
        Anel anel = new Anel();

        /*
         * Ao chamar cada método do anel pela primeira vez,
         * suas respectivas threads começam a realizar as
         * operações especificadas periodicamente.
         */
        anel.novoProcesso();
        anel.novaRequisicao();
        anel.inativarProcesso();
        anel.inativarCoordenador();
    }
}
```

A classe “Main” irá executar em seu método principal todos os métodos a classe Anel apenas uma vez, onde cada método irá abrir sua thread em um loop infinito, executados de tempos em tempos, respectivamente.

Abaixo temos o resultado da execução do código, onde eventualmente o coordenador será inativado e novos coordenadores serão eleitos ao passar do tempo.



Acima temos uma nova eleição logo após o processo coordenador 1 ser inativado e o processo 4 fazer uma requisição. Outras eleições são observadas posteriormente.

CONCLUSÃO

A implementação foi feita de forma simplificada, seguindo a ideia apresentada pela estrutura dos algoritmos. Outras maneiras de implementação foram pensadas, como tratar um número indefinido de clientes no algoritmo de Berkeley, sendo a classe cliente um objeto genérico, ou construção de relógios reais com desvio proposital de alguns minutos.

A maior dificuldade foi elaborar a comunicação do servidor para clientes individuais (onde cada cliente no Berkeley recebe uma informação diferente) utilizando RMI. Ao ver um exemplo de calculadora RMI deduzi uma “fácil” implementação do algoritmo de Berkeley, entretanto cheguei ao impasse ao tentar fazer esta comunicação com os clientes. Foi utilizado um exemplo (citado) para adaptação do código.

Para efeito de simplicidade e para execução em tempo hábil, os programas mostraram os resultados esperados e satisfatórios dentro da expectativa de mostrar como os algoritmos funcionam.

MATERIAIS UTILIZADOS

[1] TANENBAUM, Andrew S.; STEEN, Maarten van. **Sistemas Distribuídos: Princípios e Paradigmas**. 2ª Edição. Prentice-Hall, 2007.

[2] GUJ. **Fórum: JAVA RMI - Cliente A e B enviar info, servidor dar resposta**. Disponível em: <https://www.guj.com.br/t/java-rmi-cliente-a-e-b-enviar-info-servidor-dar-resposta/349650> , acesso em: 19/08/2021.

[3] DevMedia. **Threads: paralelizando tarefas com os diferentes recursos dos Java**. Disponível em: <https://www.devmedia.com.br/threads-paralelizando-tarefas-com-os-diferentes-recursos-do-java/34309>, acesso em: 14/08/2021.