

# Trabalho Final – Implementação de uma biblioteca de comunicação confiável e algoritmo distribuído.

## Sumário

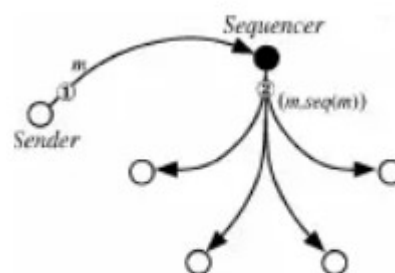
Trabalho Final – Implementação de uma biblioteca de comunicação confiável e algoritmo distribuído.....	1
Biblioteca para comunicação em grupo.....	1
Message.....	2
GroupCommunicator.....	2
VectorClock.....	3
OrderedGroupCommunicator.....	3
Validação.....	3
Algoritmo de Chang e Roberts.....	3
ConfigLoader.....	4
BroadcastSample.....	4
UnicastSample.....	7
ChangRobertsSample.....	9
Compilação.....	11

## Biblioteca para comunicação em grupo

Foram implementadas duas versões de uma classe para gerenciar a comunicação em grupo. Uma sem nenhuma garantia de ordem, e a outra com garantia causal em um unicast, e total em broadcast.

As classes utilizam Sockets sobre TCP para implementação da comunicação.

Na versão ordenada, classe **OrderedGroupCommunicator**, a garantia de **ordem total** nas mensagens de **broadcast** foi feita utilizando o algoritmo de **Sequenciador, na variação UB**.



(a) variant UB

Para garantia de **ordem causal** em **unicast**, foi implementado o algoritmo de **Schiper-Eggle-Sandoz**, com relógios vetoriais.

## Message

É a classe que carrega uma mensagem. Ela conta com um tipo, definido por **MessageType**, um objeto, que seria o conteúdo da mensagem, um número de sequência e um conjunto de relógios lógicos. O tipo da mensagem pode ser:

- **Message:** mensagens usadas na versão não ordenada;
- **Unicast:** mensagens com um destinatário. Carregam relógios lógicos e respeitarão ordem causal. Usadas na versão ordenada;
- **Broadcast:** mensagem com múltiplos destinatários. Carregam um número de sequência. Usadas na versão ordenada;
- **Seq:** mensagem de requisição de **Broadcast**. Tem como destino sempre o sequenciador.

## GroupCommunicator

O construtor deve receber um **id**, que é o id do nodo, uma tabela hash concretizada em um **Map<Integer, InetAddress>**, que são os endereços dos nodos por para cada id, e os ids ordenados de alguma forma (importante para a implementação do algoritmo distribuído escolhido).

- **bind** cria um **ServerSocket**, no endereço do id do nodo. Para inicialização do sistema, é necessário que todos os nodos realizem o bind, para que depois sejam feitas **connect**, **accept**, e **start**, nesta ordem.
- **connect** cria Stream Sockets que conectam à todos os nodos.
- **accept** espera pelas conexões de todos os nodos. Depois de aceitados todos, guarda os fluxos de saída dos sockets usados para conectar pela função **connect**. Guarda os fluxos de entrada dos sockets obtidos nos accepts feitos aqui nessa função e faz chamada à **start**.
- **start** inicia todas as threads de recepção de mensagem e de entrega. Cada thread de recepção deve fazer a leitura de um fluxo de entrada de um dos socket obtidos na fase **accept** e colocar as mensagens na lista **pending**. A thread de entrega deve retirar as mensagens de pending e colocar na fila bloqueante **delivered**.

As threads de recebimento e entrega de mensagens interagem através de um semáforo. Quando uma mensagem é adicionada à pending o semáforo tem uma liberação. A thread de entrega então, adiciona a mensagem de pending à delivered.

- **receive** retira uma mensagem de delivered e retorna seu conteúdo.
- **send (id, payload )** envia uma mensagem com o conteúdo **payload** para o nodo **id**.
- **broadcast ( payload )** envia uma mensagem para todos os nodos.

A classe conta com a função **stop** que propaga em broadcast uma mensagem nula, usada como indicador de para as threads de recepção, que a thread deve terminar. Quando recebido um **null** ativa uma **flag** no objeto de comunicação para que as threads não repitam a iteração de recepção, e repropaga a mensagem. O efeito da chamada é a parada de todas as threads de recepção do grupo

A função **close**, espera pelo fim das threads e fecha os sockets.

Há também uma função para obtenção do nó “vizinho”, utilizado para a validação com o algoritmo de **Chang-Roberts**.

## VectorClock

A classe representa um relógio vetorial definido por uma tabela hash na forma de um **Map** de **Integer** para **Long**. São definidas as funções de comparação, **lessEqual**, **equals**, **less** e **concurrent**, além de uma função de **merge** utilizada no algoritmo de **Schiper-Eggle-Sandoz**.

## OrderedGroupCommunicator

Esta classe se diferencia da sua super classe por implementar restrições quanto à ordem de entrega das mensagens.

No construtor, a classe recebe, além dos parâmetros recebidos pela superclasse, um parâmetro de indicação de qual nodo será o **sequenciador**. Na construção também é inicializado o **próximo (next)** número de sequência que deve ser aceito na recepção de um broadcast, o número de sequência (**sequence**) usado somente pelo sequenciado para atribuir as sequências às mensagens, o relógio vetorial do nodo e o conjunto de relógios vetoriais que as mensagens devem carregar.

Na recepção de mensagens, agora, se uma mensagem lida do socket é do tipo **Seq**, é atribuído o número de sequência, alterado o tipo da mensagem para **Broadcast** e feito um **broadcast**.

- **broadcast** agora faz um **unicast** de uma mensagem do tipo **Seq** para o nodo sequenciador.
- **send** incrementa seu relógio vetorial e envia a mensagem de tipo **unicast** que carrega o conjunto de relógios vetoriais que representam o estado do sistema como conhecido pelo nodo remetente. Depois do envio, atualiza o relógio vetorial referente ao destinatário.
- **deliver**, função executada pela thread de entrega, se divide na recepção de unicasts e broadcasts
- **deliverOrderedBroadcast** passa de **pending** para **delivered** mensagens que tem tipo **broadcast** e são o próximo na sequência de recepção (guardado no atributo **next** ).
- **deliverOrderedUnicast** passa de **pending** para **delivered** mensagens que tem tipo **unicast** e satisfazem as condições de que o relógio vetorial do destino, guardado pela mensagem, é menor ou igual ao guardado pelo próprio destino.

## Validação

Para validação foram criados os exemplos **BroadcastSample**, **UnicastSample** e **ChangRobertsSample**. O arquivo de configuração, “node.config”, que contém as configurações dos nós, deve estar no diretório de execução.

## Algoritmo de Chang e Roberts

O algoritmo escolhido foi o algoritmo para eleição de líder de Chang e Roberts. É um algoritmo descentralizado e assíncrono, que considera um grupo finito de processos. Para o algoritmo, é

necessário que seja possível organizar a comunicação dos processos em forma de um anel unidirecional. Então para realizar o algoritmo deve ser garantido que:

- Deve existir um número finito de processos;
- Cada processo deve ter um identificador único;
- Um processo não deve invocar uma nova eleição enquanto não obtiver a resposta de uma eleição previamente feita por ele;
- Os processos podem se organizar em forma de um anel unidirecional.
- Os processos não falham;
- A comunicação é confiável.

O algoritmo ocorre em duas rodadas. A primeira para identificação do líder e a segunda para divulgação do vencedor.

Para convocar uma eleição, um nodo deve enviar uma mensagem de eleição ao seu vizinho. A mensagem de eleição é uma mensagem que carrega o identificador do convocador e o maior identificador, como em ["eleição", "maior-id", "id"].

Um processo, com identificador ID e que guarda o valor de líder em LIDER, quando recebe uma mensagem de eleição ["eleição", MID, CID]:

- Se ele não for o processo que convocou a eleição ( $CID \neq ID$ ), envia ao seu vizinho ["eleição",  $\max(ID, MID)$ , CID];
- Se ele for o processo que convocou a eleição ( $CID = ID$ ), atualiza o valor  $LIDER = MID$  e envia uma mensagem ["eleito", MID, CID] de divulgação de eleito para o seu vizinho.

Um processo, com identificador ID e que guarda o valor de líder em LIDER, quando recebe uma mensagem ["eleito", MID, CID] de divulgação de eleito:

- Se ele não for o processo que convocou a eleição ( $CID \neq ID$ ), atualiza o valor  $LIDER = MID$ .

## ConfigLoader

Classe criada para facilitar a configuração dos nodos. A função **loadOGC** instancia e retorna um nodo dado um arquivo de configuração. É possível passar o **id** do nodo por parâmetro, sobrescrevendo o que estiver definido no arquivo de configuração, o que facilita no teste local.

O arquivo de configuração é de formato igual ao do enunciado do trabalho.

## BroadcastSample

Neste exemplo é demonstrado a **ordenação total em broadcasts**. Para isso é usada uma variação da função de broadcast feito pelo sequenciador. Nessa função é simulado o atraso no recebimento da mensagem por parte de um nodo através do atraso na escrita pelo sequenciador. Assim o sequenciador acabará escrevendo fora de ordem algumas mensagens. Já que as mensagens carregam um número de sequencia, isso será resolvido no destinatário.

```

297- /**
298-  * For test
299-  * Broadcast with delay when delivering to node
300-  * with position in outstream == sequence number % outstream size
301-  * @param payload
302-  * @throws IOException
303-  */
304- public void broadcastDelayed(Message message) throws IOException {
305-     int toDelay = ((int) this.sequence.get() % this.outStream.size());
306-     int i = 0;
307-     for (Map.Entry<Integer, ObjectOutputStream> entry : outStream.entrySet()) {
308-         ObjectOutputStream out = entry.getValue();
309-
310-         if (i == toDelay) {
311-             new Thread(() -> {
312-                 try {
313-                     Thread.sleep(1000);
314-                     synchronized (out) {
315-                         out.writeObject(message);
316-                     }
317-                 } catch (IOException | InterruptedException e) {
318-                     e.printStackTrace();
319-                 }
320-             }).start();
321-         } else {
322-             synchronized (out) {
323-                 out.writeObject(message);
324-             }
325-         }
326-         i++;
327-     }
328- }

```

Figura 1: OrderedGroupCommunicator.java

```

98- /**
99-  * Adds sequence number to message and broadcasts it.
100-  *
101-  * @param message
102-  * @throws IOException
103-  */
104- public void sequencerBroadcast(Message message) throws IOException {
105-     long sequence = this.sequence.getAndIncrement();
106-     message.setSequence(sequence);
107-     message.setType(MessageType.BROADCAST);
108-     if (this.delayedBroadcast == true) {
109-         this.broadcastDelayed(message);
110-     } else {
111-         this.broadcast(message);
112-     }
113- }

```

Figura 2: OrderedGroupCommunicator.java

```

32     sock.setDelayedBroadcast(true);
33     if (sock.getId() == 0) {
34         String m = "0";
35         sock.broadcast(m);
36         System.err.println((String) (sock.receive()));
37         System.err.println((String) (sock.receive()));
38         System.err.println((String) (sock.receive()));
39     }
40     if (sock.getId() == 1) {
41         System.err.println((String) (sock.receive()));
42         String m = "1";
43         sock.broadcast(m);
44         System.err.println((String) (sock.receive()));
45         System.err.println((String) (sock.receive()));
46     }
47     if (sock.getId() == 2) {
48         System.err.println((String) (sock.receive()));
49         System.err.println((String) (sock.receive()));
50         String m = "2";
51         sock.broadcast(m);
52         System.err.println((String) (sock.receive()));
53     }
54
55     scanner.nextLine();
56
57     sock.stop();
58     sock.close();
59
60     scanner.close();

```

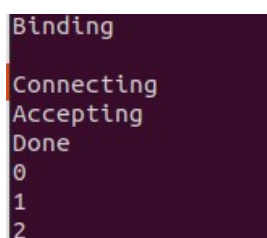
Figura 3: UnicastSample.java

Para essa demonstração, no código são referenciados os ids dos nodos, que devem ser os mesmos presentes no arquivo de configuração.

Para executar a demonstração execute, para cada id em terminais diferentes:

**java BroadcastSample <id>**

Depois que todos os nodos estiverem em “Bind” de um “enter” em cada terminal. No fim da execução, deve estar assim em cada terminal:



```

Binding
Connecting
Accepting
Done
0
1
2

```

Figura 4: Processos

Depois de finalizada de um “enter” novamente para encerrar.

O objetivo do exemplo é simular o comportamento da imagem à seguir:

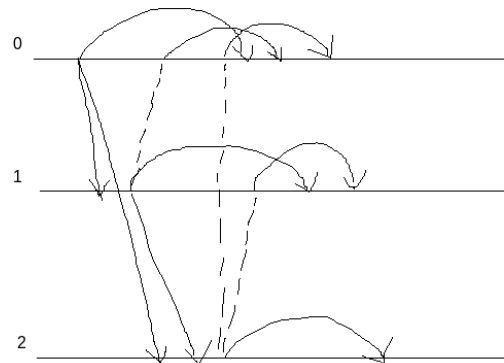


Figura 5: Gráfico

## UnicastSample

Neste exemplo é demonstrado a **ordenação causal** em **unicasts**.

Para isso é utilizada uma modificação na função **send**, onde é adicionado um **delay** na escrita, da mesma forma feita no broadcast, com o lançamento de uma thread. A mensagem será escrita mais tarde, mas com as configurações dos relógios vetoriais de quando a função **send** foi chamada.

```
247- /**
248-  * For test
249-  * Sends message with a delay Follows Schiper-Eggle-Sandoz protocol to
250-  * maintain causal order
251-  *
252-  * @throws ClassNotFoundException
253-  */
254- public void send(int id, Object payload, int delay)
255-     throws IOException, ClassNotFoundException {
256-     synchronized (vectorClocks) {
257-         this.vectorClocks.get(this.id).increment(this.id);
258-         //copy to prevents the vectors in message to change
259-         Message message = copy(new Message(MessageType.UNICAST,
260-             payload, this.vectorClocks,
261-             this.vectorClocks.get(this.id)));
262-
263-         ObjectOutputStream out = this.outStream.get(id);
264-         new Thread(() -> {
265-             try {
266-                 Thread.sleep(delay);
267-                 synchronized (out) {
268-                     out.writeObject(message);
269-                 }
270-             } catch (IOException | InterruptedException e) {
271-                 e.printStackTrace();
272-             }
273-         }).start();
274-
275-         if (this.vectorClocks.get(id) == null) {
276-             this.vectorClocks.put(id, new VectorClock(this.socketAddresses.keySet()));
277-         }
278-         this.vectorClocks.get(id).set(this.vectorClocks.get(this.id));
279-     }
280- }
```

Figura 6: OrderedGroupCommunicator.java

```

31
32     if (sock.getId() == 0) {
33         String m = "0";
34         sock.send(2, m, 1000);
35         sock.send(1, m);
36     }
37     if (sock.getId() == 1) {
38         System.err.println((String) (sock.receive()));
39         String m = "1";
40         sock.send(2, m);
41     }
42     if (sock.getId() == 2) {
43         for (int i = 0; i < 2; i++) {
44             System.err.println((String) (sock.receive()));
45         }
46     }
47 }

```

Figura 7: UnicastSample.java

Para essa demonstração, no código são referenciados os ids dos nodos, que devem ser os mesmos presentes no arquivo de configuração.

Para executar a demonstração execute, para cada id em terminais diferentes:

**java UnicastSample <id>**

Depois que todos os nodos estiverem em “Bind” de um “enter” em cada terminal. No fim da execução, deve estar assim no terminal do nodo 1 e 2 respectivamente:

```

Binding
Connecting
Accepting
Done
0

```

Figura 9: Processo 1

```

Binding
Connecting
Accepting
Done
0
1

```

Figura 8: Processo 2

Depois de finalizada de um “enter” novamente para encerrar.

O comportamento da demonstração deve simular o seguinte:

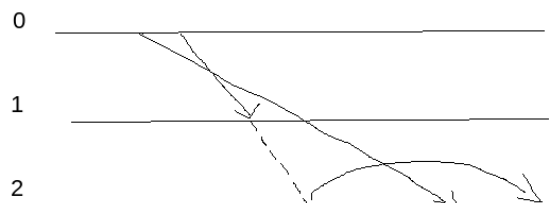


Figura 10: Gráfico



## ChangRobertsSample

Neste exemplo de validação, é testado o algoritmo de **ChangRoberts** para **eleição de lider**. É possível experimentar unicasts, broadcasts, recepção, parada, fechamento, e uma eleição.

Diferentemente dos exemplos anteriores, este não tem restrições quanto aos ids do processos.

O exemplo lança uma thread para recepção de mensagens, e é nela que a maior parte da lógica da eleição é implementada. A eleição se inicia quando é dada entrada no comando eleição em um dos nós:

```
118         }else if(comando.equals("eleger")){
119             Object[] election = {"election", sock.getId(), sock.getId()};
120             sock.send(sock.getNeighbor(), election);
```

A mensagem composta por ["election", "maior-id", "id-de-quem-começou"] começa a circular.

```
if(message[0].equals("election")) {
    //eleição
    if(((int)message[2]) == sock.getId()) {
        //é o q iniciou a eleição
        //inicia divulgação do eleito
        int next = sock.getNeighbor();
        message[0] = "elected";
        leader = (int) message[1];

        System.out.println("O eleito foi definido: "+leader);
        System.out.println("Enviando para "+next);

        Thread.sleep(1000);
        sock.send(next, message);
    }else if(((int)message[1])<sock.getId()){
        //não é o q iniciou a eleição e é maior
        //atualiza o maior
        int next = sock.getNeighbor();

        System.out.println("Recebi:"+message[1]
            +", meu ID, "+ sock.getId()+", é maior");
        System.out.println("Enviando para "+next);

        message[1] = sock.getId();
        Thread.sleep(1000);
        sock.send(next, message);
    }else {
        //não é o q iniciou a eleição e não é maior
        //repassa a mensagem
        int next = sock.getNeighbor();
        System.out.println("Repassando"+ message[1] +" para "+next);

        Thread.sleep(1000);
        sock.send(next, message);
    }
}
```

Figura 11: ChangRobertsSample.java

```
}else if(message[0].equals("elected")){
    if(((int)message[2]) != sock.getId()) {
        //mensagem de divulgação de eleito e
        //não é o que iniciou a eleição
        //repassa a divulgação
        leader = (int) message[1];
        int next = sock.getNeighbor();
        System.out.println("Elegido: "+leader);
        System.out.println("Repassando para "+next);

        Thread.sleep(1000);
        sock.send(next, message);
    }else {
        System.out.println("Fim eleição");
    }
}
```

Figura 12: ChangRobertsSample.java

Para executar a demonstração execute, para cada id em terminais diferentes:

**java ChangRobertsSample <id>**

Depois que todos os nodos estiverem em “Bind” de um “enter” em cada terminal.

Será possível executar alguns comandos de uma lista.

```

Binding
Connecting
Accepting
Done
Comandos:
    unicast <id> <mensagem>
    broadcast <mensagem>
    elege
    idlizer
    stop
    close

```

*Figura 13:*

## Comandos

Executando “eleger” iniciamos o algoritmo de eleição. Ao fim temos algo assim em cada termial (nesta execução foram 3 nodos).

```
0 eleito foi definido: 2
Enviando para 1
Fim eleição
```

*Figura 14: Processo 0*

```
Recebi:0, meu ID, 1, é maior
Enviando para 2
Elegido: 2
Repassando para 2
```

*Figura 16: Processo 1*

```
Recebi:1, meu ID, 2, é maior
Enviando para 0
Elegido: 2
Repassando para 0
```

*Figura 15: Processo 2*

Para enviar a sinalização de parada use o comando **stop** em pelo menos um dos processos. Depois, para encerrar use **close** em cada terminal.

Atualizando os valores de “node.config” para os valores em “node.config.2:

```
processos = 5
id = 45
45 = localhost:6006
2 = localhost:6007
90 = localhost:6008
12 = localhost:6009
67 = localhost:6010
```

Temos 5 processos:

```
douglas@douglas-Aspire-A515-54G:~/eclipse-workspace/Final$ java ChangRobertsSample 45
Binding

Connecting
Accepting
Done
Comandos:
unicast <id> <mensagem>
broadcast <mensagem>
eleger

1 processos = 5
2 id = 45
3 45 = localhost:6006
4 2 = localhost:6007
5 90 = localhost:6008
6 12 = localhost:6009
7 67 = localhost:6010
8

douglas@douglas-Aspire-A515-54G:~/eclipse-workspace/Final$ java ChangRobertsSample 2
Binding

Connecting
Accepting
Done
Comandos:
unicast <id> <mensagem>
broadcast <mensagem>
eleger

douglas@douglas-Aspire-A515-54G:~/eclipse-workspace/Final$ java ChangRobertsSample 90
Binding

Connecting
Accepting
Done
Comandos:
unicast <id> <mensagem>
broadcast <mensagem>
eleger
```

Figura 17: Depois de estabelecida a conexão

Podemos verificar eleições concorrentes convocadas por dois processos diferentes. Os processos 45 e 12 convocarão eleições. Adicionei mais uma informação nos prints, para saber quem convocou a eleição que gerou a mensagem recebida. O resultado é o seguinte:

```

idlider
stop
close
eleger
Convocação de 12
Não sou maior, repassando 67 para 2
O eleito foi definido: 90
Enviando para 2
Elegido: 90
Repassando para 2
Fim eleição

idlider
stop
close
Convocação de 45
Não sou maior, repassando 45 para 90
Convocação de 12
Não sou maior, repassando 67 para 90
Elegido: 90
Repassando para 90
Elegido: 90
Repassando para 90

close
Convocação de 45
Recebi: 45, meu ID, 90, é maior
Enviando para 12
Convocação de 12
Recebi: 67, meu ID, 90, é maior
Enviando para 12
Elegido: 90
Repassando para 12
Elegido: 90
Repassando para 12

1 processos = 5
2 id = 45
3 45 = localhost:6006
4 2 = localhost:6007
5 90 = localhost:6008
6 12 = localhost:6009
7 67 = localhost:6010
8

idlider
stop
close
eleger
Convocação de 45
Não sou maior, repassando 90 para 67
O eleito foi definido: 90
Enviando para 67
Elegido: 90
Repassando para 67
Fim eleição

stop
close
Convocação de 12
Recebi: 12, meu ID, 67, é maior
Enviando para 45
Convocação de 45
Não sou maior, repassando 90 para 45
Elegido: 90
Repassando para 45
Elegido: 90
Repassando para 45
  
```

Figura 18: Convocação concorrente por 45 e 12

## Compilação

No diretório que contém as classes de exemplo, execute:

```
javac gc/*.java
```

```
jar cf gc.jar gc/GroupCommunicator.class gc/Message.class gc/MessageType.class
gc/OrderedGroupCommunicator.class gc/VectorClock.class
```

```
javac -cp gc.jar *.java
```