

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

## Sumário

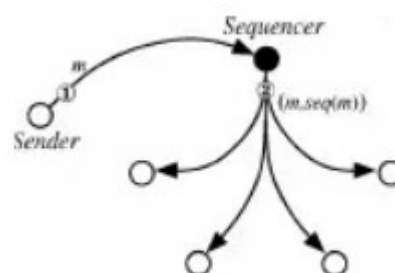
Trabalho Final – Implementação de uma biblioteca de comunicação confiável e algoritmo distribuído.....	1
Biblioteca para comunicação em grupo.....	1
Message.....	2
GroupCommunicator.....	2
VectorClock.....	3
OrderedGroupCommunicator.....	3
Compilação.....	3
Validação.....	4
Algoritmo de Chang e Roberts.....	4
ConfigLoader.....	4
BroadcastSample.....	5
UnicastSample.....	7
ChangRobertsSample.....	9

## 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

Figura 1: Sequenciador 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, InetSocketAddress>**, 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;
- **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, que executam a função **receiveMessages**, 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, que executa a função **deliver** deve retirar as mensagens de pending e colocar na fila bloqueante **delivered**;
- **init** realiza todas as operações necessárias para inicialização (essas acima);
- **stop** 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;
- **close**, espera pelo fim das threads e fecha os sockets.

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. Também tem versão com timeout;
- **send (id, payload )** envia uma mensagem com o conteúdo **payload** para o nodo **id**;

- **broadcast ( payload )** envia uma mensagem para todos os nodos.

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-Eggli-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 sequencia que deve ser aceito na recepção de um broadcast, o número de sequencia (**sequence**) usado somente pelo sequenciador para atribuir as sequencias 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 sequencia, 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 sequencia 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;
- **sequencerBroadcast** é a função executada pelo sequenciador para enviar as mensagens à todos os processos. O sequenciador é sempre o processo com o último id definido nos arquivos de configuração.

## ConfigLoader

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

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

## Compilação

No diretório que contém as classes de exemplo, para compilar a biblioteca, execute:

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

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

Para compilar os programas de exemplo:

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

## 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 convocada 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  $LIDER = MID$ .

## BroadcastSample

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

```

354  /**
355   * For test
356   * Broadcast with delay when delivering to node
357   * with position in outstream == sequence number % outstream size
358   * @param payload
359   * @throws IOException
360   */
361  public void broadcast(Message message, int toDelay, int delay) throws IOException {
362      for (Map.Entry<Integer, ObjectOutputStream> entry : outStream.entrySet()) {
363          int id = entry.getKey();
364          ObjectOutputStream out = entry.getValue();
365
366          if (id == toDelay) {
367              new Thread(() -> {
368                  try {
369                      Thread.sleep(1000);
370                      synchronized (out) {
371                          out.writeObject(message);
372                      }
373                  } catch (IOException | InterruptedException e) {
374                      e.printStackTrace();
375                  }
376              }).start();
377          } else {
378              synchronized (out) {
379                  out.writeObject(message);
380              }
381          }
382      }
383  }

```

Figura 2: GroupCommunicator.java – função de broadcast, com delay em um dado processo

Em cada broadcast a mensagem para algum nodo aleatório atrasará por 500 à 2500 milisegundos. Assim o sequenciador acabará escrevendo fora de ordem algumas mensagens. Para que o broadcast aconteça com atraso é preciso definir o atributo **delayedBroadcast**, através de um *Setter*, no sequenciador.

```
--
99  /**
100   * Adds sequence number to message and broadcasts it.
101   *
102   * @param message
103   * @throws IOException
104   */
105  public void sequencerBroadcast(Message message) throws IOException {
106      long sequence = this.sequence.getAndIncrement();
107      message.setSequence(sequence);
108      message.setType(MessageType.BROADCAST);
109      if(this.delayedBroadcast == true) {
110          Random rand = new Random();
111          int pos = rand.nextInt(this.ids.size());
112          int delay = rand.nextInt(2000) + 500;
113          int toDelay = this.ids.get(pos);
114          this.broadcast(message, toDelay, delay);
115      }else {
116          this.broadcast(message);
117      }
118  }
```

Figura 3: OrderedGroupCommunicator.java – função em que o sequenciador faz o broadcast

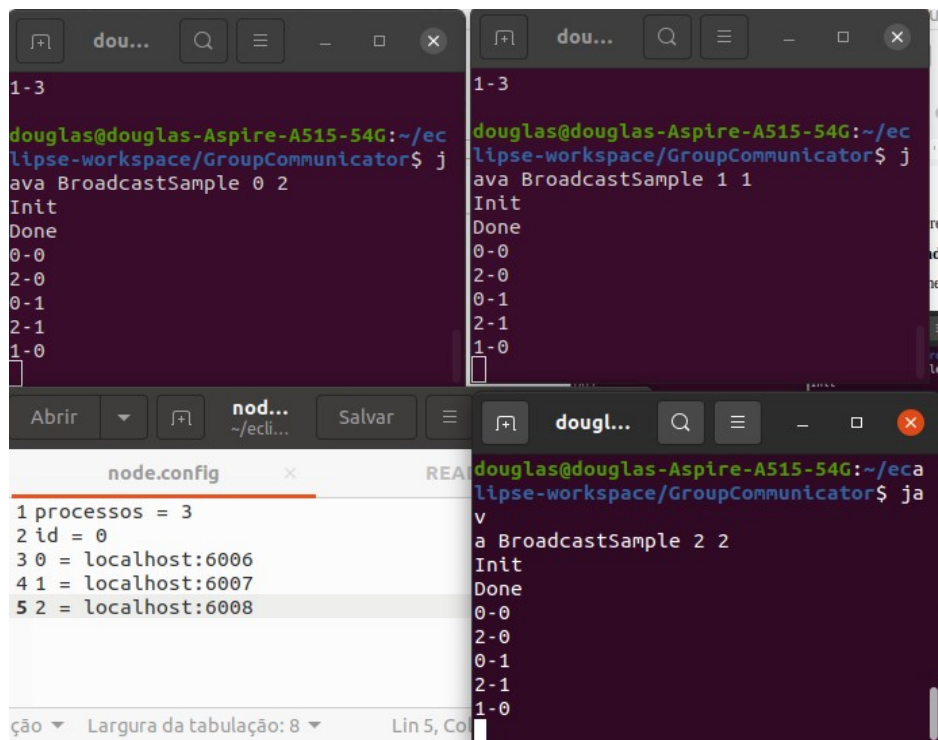
```
28  gc.setDelayedBroadcast(true);
29  int broadcastsAmount = Integer.valueOf(args[1]);
30  for (int i = 0; i < broadcastsAmount; i++) {
31      String m = gc.getId()+"-"+i;
32      gc.broadcast(m);
33  }
34
35  Thread receiveingThread = new Thread()->{
36      try {
37          while (!stop) {
38              Object received = gc.receive(100);
39              if(received != null) {
40                  System.err.println((String)received);
41              }
42          }
43      } catch (InterruptedException e) {
44          e.printStackTrace();
45      }
46  });
47
48  receiveingThread.start();
49  scanner.nextLine();
50  stop = true;
51  receiveingThread.join();
```

Figura 4: BroadcastSample.java

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

**java BroadcastSample <id> <número de broadcasts que o processo fará>**

As mensagens mandadas por um processo serão “<id>-<n>” sendo o “n” de enésima mensagem. No fim da execução a sequencia de mensagens recebidas deve ser a mesma:



The image shows three terminal windows and a node.config file. The top-left terminal shows the execution of BroadcastSample 0 2, the top-right shows BroadcastSample 1 1, and the bottom-right shows BroadcastSample 2 2. The node.config file in the bottom-left shows the configuration for three processes, each with a unique ID and a list of addresses to connect to.

```
1-3
doug...
doug...
doug...
node.config
1 processos = 3
2 id = 0
3 0 = localhost:6006
4 1 = localhost:6007
5 2 = localhost:6008
ção ▾ Largura da tabulação: 8 ▾ Lin 5, Col 1
```

Figura 5: Processos

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

## 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 que atrasará o envio. A mensagem será escrita mais tarde, mas com as configurações dos relógios vetoriais de quando a função **send** foi chamada. A técnica para simular o atraso é a mesma do teste de broadcast.

O processo com id == 0 manda mensagens “0”. O processo com id == 1 manda a mensagem “1”.

```

247- /**
248-  * For test
249-  * Sends message with a delay Follows Schiper-Eggli-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 – send com atraso.

```

26         if (gc.getId() == 0) {
27             String m = "0";
28             gc.send(2, m, 1000);
29             gc.send(1, m);
30
31         }
32         if (gc.getId() == 1) {
33             System.err.println((String) (gc.receive()));
34             String m = "1";
35             gc.send(2, m);
36         }
37         if (gc.getId() == 2) {
38             for (int i = 0; i < 2; i++) {
39                 System.err.println((String) (gc.receive()));
40             }
41         }

```

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>**

No fim da execução, deve estar assim no terminal do nodo 1 e 2 respectivamente:



```

2
douglass@douglas-Aspire-A515-54G:~/ec
lipse-workspace/GroupCommunicator$ j
ava UnicastSample 0
Init
Done
0

douglass@douglas-Aspire-A515-54G:~/ec
lipse-workspace/GroupCommunicator$ j
ava UnicastSample 1
Init
Done
0

douglass@douglas-Aspire-A515-54G:~/ec
lipse-workspace/GroupCommunicator$ j
ava UnicastSample 2
Init
Done
0
1

1 processos = 3
2 id = 0
3 0 = localhost:6006
4 1 = localhost:6007
5 2 = localhost:6008

```

Figura 8: Processo 1

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

O comportamento da demonstração deve simular o seguinte:

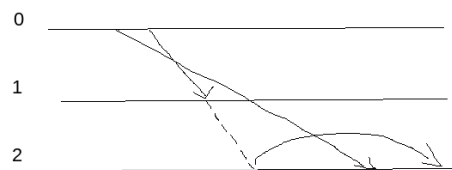


Figura 9: Gráfico

## ChangRobertsSample

Neste exemplo de validação, é testado o algoritmo de **ChangRoberts** para **eleição de líder**. É possível experimentar unicasts, broadcasts, recepção, parada, fechamento, e uma eleição. Diferentemente dos exemplos anteriores, este não tem restrições quanto aos ids dos 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 comandado eleição em um dos processos:

```

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

```

Figura 10: Início de eleição

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

```

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

        System.out.println("0 eleito foi definido: "+leader);
        System.out.println("Enviando para "+next);

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

        System.out.println("Convocação de "+message[2]);
        System.out.println("Recebi: "+message[1]
            +", meu ID, "+ gc.getId()+" é maior");
        System.out.println("Enviando para "+next);

        message[1] = gc.getId();
        Thread.sleep(1000);
        gc.send(next, message);
    }else {
        //não é o q iniciou a eleição e não é maior
        //repassa a mensagem
        int next = gc.getNeighbor();
        System.out.println("Convocação de "+message[2]);
        System.out.println("Não sou maior, repassando "+ message[1] +" para "+next);

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

```

Figura 11: ChangRobertsSample.java

```

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

        Thread.sleep(1000);
        gc.send(next, message);
    }else {
        System.out.println("Fim eleição");
    }
}else if(message[0].equals("message")) {
    System.out.println(message[1]);
}
}

```

Figura 12: ChangRobertsSample.java

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

**java ChangRobertsSample <id>**

Será possível executar alguns comandos de uma lista. É possível testar também a ordenação causal em mensagens unicast com invocação do comando **unicast <id> <delay> <mensagem>**. Executando “eleger” iniciamos o algoritmo de eleição. Ao fim temos algo assim em cada terminal (nesta execução foram 3 nodos).

```

Init
Done
Comandos:
unicast <id> <mensagem>
broadcast <mensagem>
eleger
idlider
stop
close
eleger
0 eleito foi definido: 2
Enviando para 1
Fim eleição

Comandos:
unicast <id> <mensagem>
broadcast <mensagem>
eleger
idlider
stop
close
Convocação de 0
Recebi: 0, meu ID, 1, é maior
Enviando para 2
Elegido: 2
Repassando para 2

Comandos:
unicast <id> <mensagem>
broadcast <mensagem>
eleger
idlider
stop
close
Convocação de 0
Recebi: 1, meu ID, 2, é maior
Enviando para 0
Elegido: 2
Repassando para 0

```

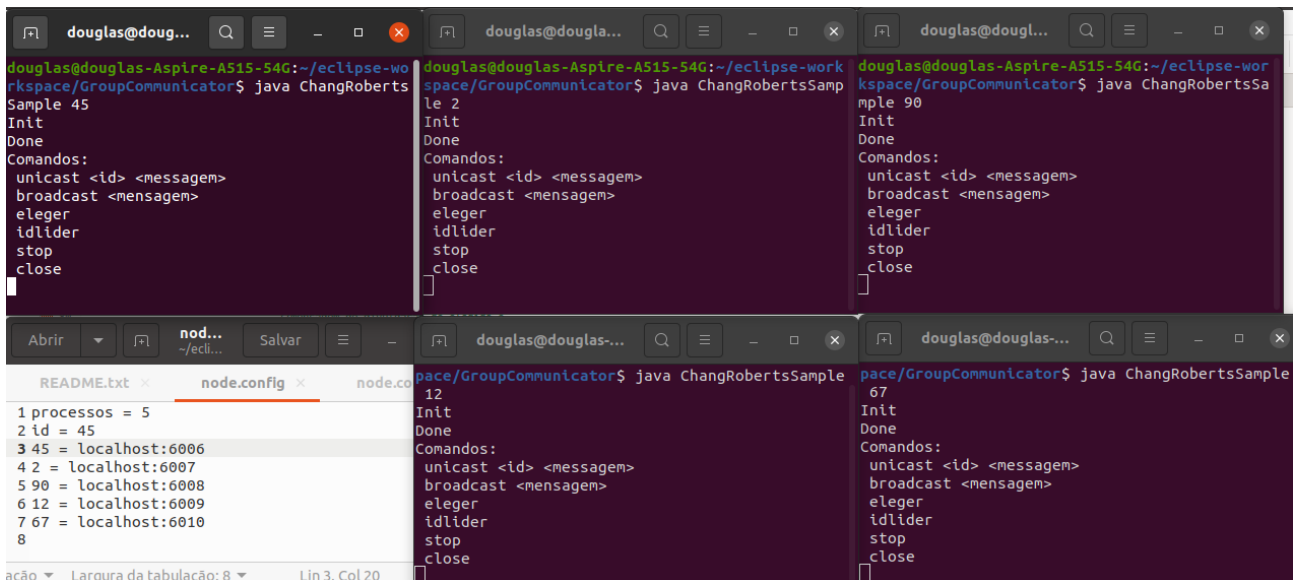
Figura 13: ChangRobertsSample - depois da eleição convocada pelo processo 0

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 (renomear node.config.2 para node.config):

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

Temos 5 processos:



The image shows a screenshot of a development environment. At the top, there are three terminal windows. Each terminal window shows the output of running 'java ChangRobertsSample' with different IDs. The first terminal shows ID 45, the second shows ID 2, and the third shows ID 90. Each terminal output includes 'Init', 'Done', and a list of commands: 'unicast <id> <mensagem>', 'broadcast <mensagem>', 'eleger', 'idlider', 'stop', and 'close'. Below the terminals, there is an IDE window showing a file named 'node.config'. The file contains a list of processes: '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', and '8'. The IDE window also shows a menu bar with 'Abrir', 'nod...', 'Salvar', and 'node.co'.

Figura 14: ChangRobertsSample - depois da inicialização

Podemos verificar eleições concorrentes, cenário previsto pelo algoritmo, convocadas por dois processos diferentes. O 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:

```

broadcast <mensagem>
eleger
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

broadcast <mensagem>
eleger
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

idlider
stop
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

Abrir  nod...  Salvar  node.co
README.txt  node.config  node.co
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
ação  Largura da tabulação: 8  Lin 3, Col 20

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 15: ChangRobertsSample - depois do fim da convocação de eleições concorrentemente pelos processos 45 e 12