Relatório do segundo trabalho

Maria Eduarda de Melo Hang (17202304) e Matheus Leonel Balduino (17202305)

15 de julho de 2021

1 Introdução

A aplicação representa um chat em tempo real, possuindo as funcionalidades mais comuns em aplicativos populares, e foi desenvolvida em Java 8, utilizando-se ferramentas como Maven e Makefiles (make) para automatizar o processo de *build*, execução e simplificação de comandos rotineiros durante o desenvolvimento. Na camada de framework, utilizou-se JGroups para gerenciamento dos *clusters* envolvidos.

O foco da implementação que acompanha este relatório está no aprendizado prático da teoria de Computação Distribuída, portanto a interface de interação com usuário é implementada diretamente em um terminal, fazendo uso de cores e estilos de fonte para distinção de contexto.

O relatório está divido de maneira que os tópicos utilizem o contexto dos tópicos anteriores, portanto, é encorajada a leitura seguindo a ordenação natural da numeração dos tópicos. Em primeiro lugar, serão discutidos os detalhes de implementação que compõe a aplicação, já apresentando as funcionalidades implementadas, seguido do estudo de alguns dos cenários possíveis em execuções do programa.

Ao término do relatório, são abordadas discussões referentes às limitações da implementação, no âmbito da área de computação distribuída, seguido de um tópico que destaca as principais dificuldades encontradas durante o desenvolvimento.

2 Implementação

Nessa seção serão descritas as decisões de projeto e as funcionalidades implementadas para a aplicação em conjunto com o código.

Em relação aos protocolos existentes no JGroups, utilizamos os que se encontram na Tabela 1. Na camada de transporte decidimos utilizar o UDP porque ele permite as mensagens tanto *broadcast* quanto *unicast*, diferente do que é encontrado no TCP. Caso usássemos o TCP, seria necessário criar uma conexão para cada membro do *cluster* para enviar uma mensagem *broadcast*, causando um *overhead* completamente desnecessário. Sobre os outros, escolhemos os mais simples que foram encontrados na biblioteca e eles serão descritos em seguida, uma vez que não vimos a necessidade de utilizar algo mais complexo de forma a diminuir a eficiência da aplicação.

O protocolo PING da camada de descoberta, que é responsável por encontrar o coordenador do *cluster* para poder entrar, foi utilizado por ser o mais simples e compatível com o UDP (alguns protocolos como TCPPING, TCPGOSSIP e MPING utilizam o TCP). Esse protocolo basicamente manda diversos pacotes de ping para descobrir o coordenador do *cluster* para poder solicitar sua entrada ao grupo posteriormente. O protocolo MERGE3 é responsável por reunir *subclusters* caso tenha acontecido uma partição entre os membros devido à uma falha, sendo responsabilidade dele também decidir quem será o coordenador do *cluster*. Em relação aos protocolos de detecção de falhas, temos o FD_ALL e VERIFY_SUSPECT, cujo primeiro é responsável por avaliar se um nodo parou através do protocolo *heartbeat*, em que cada membro deve mandar periodicamente mensagens de que está vivo (*heartbeat*) para todos, enquanto que o segundo deve avaliar se o membro realmente deu problema, e após o FD_ALL relatar que talvez tenha encontrado um nodo com falha. Esse protocolo irá enviar uma mensagem de ping direcionada para o alvo para evitar exclusões de membros desnecessárias, já que as suspeitas de que um membro falhou podem estar equivocadas. Para garantir a confiabilidade (entrega em ordem e tratamento de perda de pacotes) das mensagens de *multicast* e *unicast*, foi necessário adotar os protocolos NAKACK2 e UNICAST3 respectivamente, que garantem uma política

do tipo FIFO (*First In First Out* para as mensagens de um mesmo usuário através de números de sequências nas mensagens enviadas e retransmissão caso alguma mensagem tenha sido perdida. Esses protocolos foram necessários devido à deficiência do protocolo UDP de garantir uma confiabilidade à aplicação. Ainda falando da retransmissão, é necessário guardar as mensagens que potencialmente precisarão ser reenviadas caso aconteça uma falha e isso pode acabar enchendo a memória, deixando o sistema mais lento, com isso, torna-se importante a remoção das mensagens que já foram vistas por todos os membros. Essa responsabilidade cabe ao protocolo STABLE, que realiza um consenso com os nodos para descobrir se todos já viram as mensagens, e caso tenham visto, é solicitado que removam as mensagens para liberar a memória.

Além disso, como queremos que novos membros entrem no *cluster*, é fundamental ter um protocolo de *Group Membership*, cuja função é tratar a entrada de novos membros, solicitações de saídas e a remoção dos membros que o VERIFY_SUSPECT tenha confirmado a falha. Para fragmentar as mensagens em pacotes que sejam do tamanho adequado ao receptor, o protocolo FRAG2 se torna fundamental, uma vez que não temos garantia de que todos os membros possuem a mesma *Maximum Transimission Unit* (MTU). Como não queremos que retransmissões sejam feitas constantemente por causa do descarte de pacotes no roteador do receptor, adicionamos os protocolos UFC e MFC para as mensagens *multicast* e *unicast* respectivamente, evitando que o transmissor mande os pacotes à uma taxa superior do que o receptor consegue receber (evitando o descarte de pacotes). Por fim, o COUNTER é usado para tratar as requisições relacionadas aos contadores atômicos distribuídos do JGroups que serão utilizados pelas enquetes. Não foi necessário incluir um protocolo de transferência de estado entre membros, porque não há um compartilhamento de estados entre os membros do *cluster*.

Camada	Protocolo(s)
Transporte	UDP
Descoberta	PING
Merging	MERGE3
Detecção de falhas	FD_ALL e VERIFY_SUSPECT
Confiabilidade	NAKACK2 e UNICAST3
Limpeza	STABLE
Group Membership	GMS
Controle de fluxo	UFC e MFC
Fragmentação de mensagens	FRAG2
Contadores	COUNTER

Tabela 1: Pilha de protocolos utilizada na aplicação

Antes do usuário ter acesso aos comandos da aplicação, é necessário pegar o nome de usuário e do chat que ele entrará e depois criar o *JChannel* com esses valores. A primeira parte é responsabilidade da classe App, que também guarda todos as instâncias da classe Chat para permitir que o usuário possa trocar o grupo de conversa quando quiser (como pode ser visto na linha 2). Já a segunda tarefa cabe à classe Chat, que deverá criar os canais, atribuir o nome do usuário no lugar do endereço criado pela API, tratar a entrada do usuário e seus comandos. O código abaixo mostra o começo da aplicação em que solicitamos o nome do usuário e do grupo da conversa e criamos uma nova instância da classe Chat, passando o próprio app, o nome do usuário, nome do chat e que ele estará ativo, já que é o primeiro.

```java
public class App {
    // Guardando todos os chats
    public final HashMap<String, Chat> chats = new HashMap<>();

    public App() {
        TerminalGUI.clear();
        ChatUtil.greet();

        // Solicitando o nome do usuário
        TerminalGUI.printLn("Seu nome de usuário:");
        String username = TerminalGUI.read(50);
        TerminalGUI.printLnInfo("Nome de usuário: " + username);

        // Solicitando o nome do chat
        TerminalGUI.printLn("Nome do chat:");
        String chatname = TerminalGUI.read(50);
        TerminalGUI.printLnInfo("Chat: " + chatname);

        // Omitindo os logs
        LogFactory.setCustomLogFactory(new app.LogFactory());

        // Criando o primeiro chat e deixando ele como ativo
        Chat newChat = new Chat(this, username, chatname, true);

        // Colocando no hashmap
        this.chats.put(chatname, newChat);

        // Ativando o primeiro chat
        newChat.activate();
    }
}
```

Listing 1: Início da aplicação

O método **activate**, que pode ser visto logo abaixo, irá habilitar o atributo **isActive** do chat (linha 2) para dizer que é o chat atual, criar o *JChannel* com o arquivo de configuração da pilha de protocolos e adicionar ele mesmo como o receptor dos eventos e mensagens do canal (linhas 5 a 11) , colocar o nome de usuário (linha 13) e tentar se conectar com o canal para permitir que o usuário mande mensagens aos outros membros (linhas 15 a 19). Além disso, criará um serviço de contadores para manipular os contadores atômicos do JGroups que serão utilizados pela enquete e, caso seja o coordenador (ou seja, quem gerou o chat), irá criar o primeiro contador para gerenciar o número das enquetes que serão geradas pelos usuários. Por fim, irá receber as entradas do usuário através do método **handleUserInput**.

```java
public void activate() {
        // Marca o chat como o atual
        isActive = true;
        TerminalGUI.clear();

        try {
            InputStream configXML = Thread.currentThread().getContextClassLoader().getResourceAsStream("config.xml");
            // Cria o canal com as configurações
            // Coloca ele mesmo com receptor das mensagens e eventos do canal
            channel = new JChannel(configXML).setReceiver(this);
        } catch (Exception e) {
            TerminalGUI.printLnError(e.getMessage());
            return;
        }

        // Coloca o nome do usuário
        channel.setName(username);

        try {
            // Tenta se conectar
            channel.connect(chatname);
        } catch (Exception e) {
            TerminalGUI.printLnError(e.getMessage());
        }

        // Cria o serviço de contadores
        service = new CounterService(channel);

        // Se for o coordenador, deve criar um contador para gerenciar os identificadores das enquetes
        if (isCoord())
            service.getOrCreateCounter(SURVEYS_COUNTER, 0);


        TerminalGUI.clear();
        TerminalGUI.printLnInfo("Você entrou no chat '" + chatname + "'");

        // Trata os comandos do usuário
        handleUserInput();
    }
```

Listing 2: Ativando um chat

Como descrito anteriormente, o método **handleUserInput** é responsável por tratar as entradas do usuário. O trecho de código abaixo mostra como é feito esse tratamento. Caso o usuário tenha enviado um comando ele será redirecionado para o método **handleCommand**, senão será enviada uma mensagem ao grupo no chat através do método **sendMessage**. Para enviar mensagem a todos os membros, é necessário criar uma mensagem com o destinatário como nulo [Ban] (linha 27). O código abaixo mostra como é feito o tratamento da entrada do usuário e o envio das mensagens através do canal.

```java
private void handleUserInput() {
        // Enquanto o usuário não quiser sair da aplicação...
        while (true) {
            // Solicita a entrada do usuário
            String userInput = TerminalGUI.read(100);

            assert userInput != null;

            if (userInput.length() == 0)
                continue;

            // Verifica se é um comando
            if (ChatUtil.isCommand(userInput)) {
                boolean shouldQuit = handleCommand(userInput);
                if (shouldQuit)
                    break;
                continue;
            }

            // Verifica se existe um canal para esse chat
            if (channel == null) {
                TerminalGUI.printLnError("Você não está conectado a nenhum chat");
                continue;
            }

            // Envia a mensagem para todos
            sendMessage(new Message(null, userInput));
        }

        exit(0);
    }
}

// Enviando a mensagem através do canal
private void sendMessage(Message msg) {
    try {
        channel.send(msg);
    } catch (Exception e) {
        TerminalGUI.printLnError(e.getMessage());
    }
}
```

Listing 3: Tratando as entradas do usuário e enviando as mensagens

Devido ao fato do uso de uma interface gráfica extremamente simplificada para interagir com o usuário, foi desenvolvido o conceito de comandos, que podem ser enviados pelo usuário e serão tratados pelo **handleCommand**, como foi descrito anteriormente.

Todo comando que o usuário possa utilizar deve iniciar com o caractere de ponto de exclamação, seguido do identificador do comando. Note que alguns comandos podem exigir que o usuário informe algum argumento, como um identificador de algum recurso a ser manipulado, por exemplo, o comando a seguir é utilizado para realizar o envio de mensagens privadas, substituindo a string $< nome\_membro >$ pelo nome do membro alvo:

$$!priv \ < nome\_membro > < mensagem > .$$

A seguir é apresentada uma lista com os comandos disponíveis para uso:

1. **!sair:** Comando utilizado para que o usuário finalize a execução do processo por completo, saindo de todos os chats;

2. **!entrar <nome_chat>:** Caso o usuário deseje se conectar a mais de um chat ao mesmo tempo, pode utilizar esse comando. Note que este comando exige que o usuário informe um argumento, destacado como <nome_chat>, que representa o nome do chat ao qual a conexão deve ser feita;

3. **!membros:** Mostra uma lista com o nome dos membros conectados no chat atual;

4. **!desconectar:** Desconecta do chat atual;

5. **!priv <nome_membro> <mensagem>:** Envia uma mensagem privada para um único membro do chat atual, onde <nome_membro>deve ser uma string igual ao nome de algum dos membros do chat, e <mensagem> é a mensagem a ser enviada;

6. **!enquete <time-out> <titulo> <opção_1>,<opção_2>,...,<opção_n>:** Utilizado para criar uma nova enquete dentro do grupo atual. Nesse comando é obrigatório informar um time-out (em minutos) para que seja finalizada a enquete, bem como um título acompanhado de quantas opções se considerar necessário. Note que após a criação da enquete, um número identificador será criado. Este identificador é utilizado para votar nas enquetes;

7. **!votar <num_enquete> <opção>:** Após a abertura de um enquete (com o comando !enquete), é possível que os membros do grupo realizem votações. Para isso deve-se utilizar este comando, deve fornecer o identificador da enquete na qual se deseja votar, acompanhado do identificador da opção desejada;

8. **!ajuda:** Escreve na tela um texto de ajuda, explicando o uso detalhado de cada comando.

Para tratar os comandos, foi necessário criar um *enum*, que é apresentado logo abaixo. Com o enumerador criado, ficou muito simples tratar a entrada do usuário, porque basta criar um *switch* para cada valor do enumerador, como pode ser visto no código 5. Em relação ao tratamento dos comandos, primeiro, verificamos se esse suposto comando pertence ao enumerador, uma vez que o usuário pode enviar um comando inválido ao colocar o primeiro caractere como o ponto de exclamação (linhas 4 a 14). Caso seja inválido, avisamos o usuário e retornamos falso, porque o usuário não pediu para sair da aplicação, caso contrário, chamamos o método adequado ao comando. Caso ele queira sair, precisamos retornar verdadeiro para o **handleUserInput** sair do laço. Os métodos mais complexos serão explicados nas subseções seguintes.

```java
public enum Commands {
        ENTER("!entrar", true),
        QUIT("!sair", false),
        MEMBERS("!membros", false),
        DISCONNECT("!desconectar", false),
        PRIVATE_MSG("!priv", true),
        SURVEY("!enquete", true),
        VOTE("!votar", true),
        HELP("!ajuda", false);

    // Atributos
        public String argument;
        private final String cmd;
        private final boolean hasArg;

    // Construtor
        Commands(String cmd, boolean hasArg) {
                this.cmd = cmd;
                this.hasArg = hasArg;
        }

    // Verificando se uma string é um comando, e caso seja deixa o atributo argument preenchido
        public static Commands fromString(String cmd) throws Exception {
                for (Commands c : Commands.values()) {
                        if (cmd.startsWith(c.cmd)) {
                                if (!c.hasArg)
                                        return c;

                                String[] split = cmd.split(" ", 2);
                                if (split.length != 2)
                                        throw new Exception("Comando '" + c.cmd + "' necessita de um argumento");

                                c.argument = split[1];
                                return c;
                        }
                }
                return null;
        }
}
```

Listing 4: Comandos da aplicação

```java
private boolean handleCommand(String command) {
        Commands c;

        try {
            c = Commands.fromString(command);
        } catch (Exception e) {
            TerminalGUI.printLnError(e.getMessage());
            return false;
        }

        if (c == null) {
            TerminalGUI.printLnError("Comando '" + command + "' desconhecido. Caso não lembre dos comandos, digite !ajuda.");
            return false;
        }

        switch (c) {
            case ENTER:
                isActive = false;
                app.chats
                        .computeIfAbsent(c.argument, k -> new Chat(app, username, k, true))
                        .activate();
                break;
            case QUIT:
                quit();
                return true;
            case MEMBERS:
                getMembers();
                break;
            case DISCONNECT:
                disconnect();
                break;
            case PRIVATE_MSG:
                sendPrivateMessage(c.argument);
                break;
            case SURVEY:
                makeSurvey(c.argument);
                break;
            case VOTE:
                vote(c.argument);
                break;
            case HELP:
                ChatUtil.showCommands();
                break;
        }

        return false;
    }
```

Listing 5: Tratando os comandos enviados pelo usuário

2.1 Mensagens diretas

Para o envio de mensagens diretas, um-para-um (*unicast*), utilizamos o mesmo método disponibilizado pela API do JGroups para enviar mensagens de *broadcast*, porém com um destinatário especificado.

De forma a facilitar o usuário na hora de mandar as mensagens privadas, deixamos como responsabilidade da aplicação descobrir o endereço do destinatário, com isso o usuário precisaria apenas colocar o nome e a mensagem. Para isso funcionar, foi necessário desenvolver dois métodos: **sendPrivate-**

Message e **getAddress** que podem ser vistos logo abaixo. O primeiro é responsável por validar o comando !*priv*, verificando se há argumentos válidos (linhas 2 a 6 e linhas 17 a 20) e mandar a mensagem direta através do **sendMessage** (linha 22), enquanto que o segundo deve encontrar o endereço de um membro com o nome dele. Para encontrar o endereço de um membro é bem simples, basta verificar na *view* do canal se esse nome se encontra na lista de membros.

Como queríamos que a mensagem direta tivesse um sublinhado e uma cor diferente, foi necessário adicionar no começo da mensagem a string **!PRIVATEMSG#** para que o receptor, quando recebesse a mensagem através do método **receive**, pudesse identificá-la como uma mensagem privada e adicionar a formatação do texto como desejado. Como temos a garantia que o usuário nunca poderá mandar uma mensagem começando com o ponto de exclamação, torna-se simples tratar esses comandos entre os nodos sem precisar de outro canal.

```
1  private void sendPrivateMessage(String args) {
2          String[] splittedArgs = args.split(" ", 2);
3          if (splittedArgs.length != 2) {
4              TerminalGUI.printLnError("Você deve passar o nome do usuário e a mensagem.
5              Tente novamente com o comando: !priv <nome> <mensagem>");
6              return;
7          }
8
9          String dest = splittedArgs[0];
10         String msg = splittedArgs[1];
11
12         Address addr = this.getAddress(dest);
13         if (addr == null) {
14             TerminalGUI.printLnError("Não existe um usuário com o nome '" + dest + "'. Tente novamente com outro nome.");
15             return;
16         }
17
18         if (msg.length() == 0) {
19             TerminalGUI.printLnError("A mensagem está vazia!");
20             return;
21         }
22
23         sendMessage(new Message(addr, "!PrivateMSG#" + msg));
24
25         TerminalGUI.printLn(WHITE, "Você" + ": " + msg);
26 }
27
28 private Address getAddress(String name) {
29     View view = channel.view();
30     return view.getMembers().stream()
31             .filter(address -> name.equals(address.toString()))
32             .findAny().orElse(null);
33 }
```

Listing 6: Enviando mensagens diretas

2.2 Chats múltiplos

É possível que um processo, de um usuário específico, deseje se conectar simultaneamente a vários chats. Esse processo é facilmente implementado ao realizar o desacoplamento do chat e da aplicação. No modelo escolhido para modelar o sistema, existe um objeto Java que representa a aplicação, que é acompanhado de vários outros objetos que representam individualmente cada chat.

Ao realizar a chamada do comando !*entrar* < *nome_chat* >, a aplicação irá criar um novo objeto Chat, inserir o mesmo na estrutura de dados da aplicação que controla os chats (*HashMap*) e passar a aceitar as mensagens deste novo chat em prejuízo das mensagens dos outros chats, que podem ser

considerados em *background*.

Note, no entanto, que ao fazer a troca de um chat A para outro chat B, não é realizada nenhuma desconexão. Os membros do chat A não recebem mensagem informando que um membro foi para o chat B, pois ele ainda continua conectado (porém em *background*). As mensagens recebidas em todos os chats, exceto *foreground*, são completamente ignoradas), como pode ser visto no código abaixo. As linhas de 12 a 15 estão relacionadas às enquetes e serão explicadas posteriormente.

```
1     @Override
2     public void receive(Message msg) {
3         // Caso não esteja ativo, ignore a mensagem
4         if (!isActive)
5             return;
6
7         String content = msg.getObject();
8         Address source = msg.getSrc();
9         String nameSource = isMe(source) ? "Você" : source.toString();
10
11        if (ChatUtil.isCommand(content)) {
12            // Enquetes
13            String[] contents = content.split("#", 2);
14            if (contents[0].startsWith("!Remove")) {
15                removeVote(Long.parseLong(contents[1]));
16                return;
17            }
18            // Mensagens privadas
19            if (contents[0].startsWith("!PrivateMSG")) {
20                TerminalGUI.printLn(WHITE, nameSource + ": (PRIVADO) " +  contents[1], true);
21            }
22        } else {
23            // Caso esteja ativo, mostre a mensagem na tela
24            TerminalGUI.printLn(WHITE, nameSource + ": " +  content);
25        }
26
27    }
```

Listing 7: Recebendo as mensagens e ignorando-as caso seja um chat inativo

2.3 Enquetes

A funcionalidade de criar enquetes foi a mais complicada durante o desenvolvimento da aplicação, uma vez que tentamos diversas formas para implementá-la. Primeiramente, tentamos com o *RpcDispatcher*, criando um método na classe Chat que ficava esperando o usuário mandar a resposta da enquete, podendo mandar -1 caso não quisesse participar. No entanto, essa abordagem acabou não funcionando, uma vez que em alguns momentos os métodos de todos os membros eram chamados e em outros não, e também havia uma condição de corrida com o *stdin* que deixava o comportamento não determinístico, uma vez que não se sabia se o que o usuário digitasse iria para o **handleUserInput** ou para o método que trataria a votação da enquete ou até poderia travar a aplicação, o que não nos deixou satisfeitos. Além disso, percebemos que não seria interessante para o usuário, uma vez que precisaria esperar que todos votassem para poder usar a aplicação novamente.

Dessa forma, procuramos outras formas de solucionar esse problema e encontramos os contadores atômicos e distribuídos do JGroups, que resolvem o problema das enquetes. A partir disso criamos um contador global para decidir qual seria o identificador ao criar uma enquete, que foi mostrado no código 2 nas linhas 30 e 31, assim duas enquetes não teriam o mesmo identificador.

Quando o usuário solicitar a criação de uma enquete, ele será redirecionado para o método **makeSurvey**, que se encontra logo abaixo. Além de verificar se os argumentos são válidos, a função irá obter o identificador para a enquete através do contador que é criado durante a inicialização de um Chat, cujo nome está guardando na variável **SURVEY_COUNTER**, através do método **getOr-**

CreateCounter do *Counter Service*. Esse método irá pegar o contador que tem o nome colocado no primeiro argumento e irá ignorar o segundo argumento caso ele exista, caso contrário criará um novo com o valor do segundo argumento, nesse caso -1. Colocamos esse valor negativo para descobrirmos se o contador foi criado durante a execução do **makeSurvey**, porque o contador, caso tenha sido gerado corretamente, deveria estar com um valor positivo ou zero. Se estiver tudo certo, será criado um contador para cada opção da enquete com o nome no formato **ENQ#NumeroEnquete#NumeroOpcao** e com o valor zerado. Em seguida, enviamos uma mensagem para todos do grupo para avisar que uma enquete foi criada, incrementamos o contador das enquetes e criamos uma SurveyThread para contabilizar os votos,deixando o usuário livre para mandar mensagens e comandos. Em relação à classe Survey, a criamos para representar uma enquete e simplificar o conteúdo da mensagem, já que seria necessário apenas um **toString()**.

```java
private void makeSurvey(String args) {
        String[] splittedArgs = args.split(" ", 3);

        if (splittedArgs.length != 3) {
            TerminalGUI.printLnError("É necessário um timeout (em minutos), título e uma lista opções para mandar uma enquete
            return;
        }

        long timeout = Long.parseLong(splittedArgs[0]);
        String title = splittedArgs[1];
        String options = splittedArgs[2];

        List<String> splitedOptions = Arrays.asList(options.split(","));

        if (splitedOptions.size() <= 1) {
            TerminalGUI.printLnError("É necessário mais de uma opção para ser uma enquete");
            return;
        }

        Survey survey = new Survey(title, splitedOptions);

        Counter count = service.getOrCreateCounter(SURVEYS_COUNTER, -1);

        long idx = count.get();

        // Verifica se o contador existia mesmo
        if (idx < 0) {
            TerminalGUI.printLnError("Não foi possível criar uma enquete no momento. Tente novamente mais tarde.");
            return;
        }

        // Cria um contador para cada opção da enquete
        for (int i = 0; i < splitedOptions.size(); i++) {
            service.getOrCreateCounter(ChatUtil.createCounterName(idx, i), 0);
        }

        // Manda uma mensagem avisando que criou uma enquete
        sendMessage(new Message(null,"Criei uma nova enquete!\n"
                + survey.toString()
                + "Para votar nela, digite: !votar " + idx + " <opcao>\n"
                + "Ela acabará em " + timeout + " minuto(s)!\n"
                )
        );

        // Incrementa o contador do identificador de enquetes
        count.incrementAndGet();

        // Cria uma Thread para cuidar do resultado da enquete
        SurveyThread t = new SurveyThread(service, idx, survey, timeout, SURVEYS_COUNTER, channel);
        t.start();
}
```

Listing 8: Criando uma enquete

Em relação a SurveyThread, ela é a classe responsável por esperar o tempo da enquete acabar, contabilizar os votos, mostrar o resultado ao grupo e avisar que a enquete acabou. A primeira coisa que a *thread* irá fazer é esperar o tempo que foi solicitado pelo usuário (linha 15) e, em seguida, pegará os valores guardados, deletará os contadores das opções, decrementará o contador de enquetes e enviará

o resultado e uma mensagem solicitando aos membros que deletem a enquete das que já foram votadas, essa última parte ajuda a aplicação a impedir que o usuário vote duas vezes na mesma enquete. O código da SurveyThread é mostrado abaixo.

```java
public SurveyThread(CounterService service, long surveyNumber, Survey survey, long sleepTimeInMinutes,
        String surveyCounters, JChannel channel) {
        super();
        this.service = service;
        this.surveyNumber = surveyNumber;
        this.survey = survey;
        sleepTime = sleepTimeInMinutes * 60000;
        this.surveyCounters = surveyCounters;
        this.channel = channel;
}

public void run() {

        try {
                sleep(sleepTime);
        } catch (InterruptedException e) {
                TerminalGUI.printLnError(e.getMessage());
                return;
        }

        HashMap<String, Long> results = new HashMap<>();

        List<String> options = survey.getOptions();

        float total = 0;

        for (int i = 0; i < options.size(); i++) {
                String name = ChatUtil.createCounterName(surveyNumber, i);
                Counter counter = service.getOrCreateCounter(name, 0);
                long votes = counter.get();
                total += votes;
                results.put(options.get(i), votes);
                service.deleteCounter(name);
        }

        Counter counter = service.getOrCreateCounter(surveyCounters, 0);
        counter.decrementAndGet();

        String result = "Resultado da enquete #" + surveyNumber + " (" + survey.getTitle() + "): \n";

        for (String key : results.keySet()) {
                float percentage = (results.get(key) / total) * 100;
                result += String.format("A opção %s recebeu %d votos (%.2f %%).\n", key, results.get(key), percentage);
        }

        try {
                channel.send(new Message(null, result));
                channel.send(new Message(null, "!Remove#" + surveyNumber));
        } catch (Exception e) {
                TerminalGUI.printLnError(e.getMessage());
        }

}
```

Listing 9: Código da SurveyThread

O método **vote** é responsável por tratar o comando de votação de uma enquete. Primeiramente, ele irá verificar se conseguiu acessar o contador das enquetes para avaliar se o número da enquete que

foi dado é válido (linhas 11 a 25). Caso consiga acessar e seja um valor correto , verificará se o usuário já não votou nessa enquetes (linhas 28 a 31), e caso não tenha votado, avaliará se o número dado da opção também é coerente (linhas 33 a 46), com a mesma técnica que foi utilizada no **makeSurvey**, colocando um valor negativo para o contador. Se estiver tudo certo, irá incrementar o valor do contador (linha 48), adicionar essa enquete como votada, assim evitamos que ele vote novamente, (linha 50) e avisar ao usuário que o voto foi contabilizado.

Como salvamos as enquetes votadas, precisamos removê-las assim que uma determinada enquete acabar, caso contrário o usuário nunca mais poderá votar em uma enquete com o mesmo identificador de uma enquete antiga. Dessa forma, foi necessário adicionar a mensagem no código da SurveyThread, que pode ser encontrado no código 9, que será recebida pelo método **receive** para os membros removerem esse identificador da lista através do método **removeVote**, que é basicamente remover um elemento da lista.

```java
public void vote(String args) {
    String[] splittedArgs = args.split(" ", 2);

    if (splittedArgs.length < 2) {
        TerminalGUI.printLnError("É necessário o número da enquete e qual a opção que você votou");
        return;
    }

    long survey = Long.parseLong(splittedArgs[0]);

    Counter surveyCounter = service.getOrCreateCounter(SURVEYS_COUNTER, -1);

    long numOfSurveys = surveyCounter.get();

    // É possível acessar o contador das enquetes?
    if (numOfSurveys < 0) {
        TerminalGUI.printLnError("Não foi possível ver enquete no momento");
        return;
    }

    // É um valor válido para uma enquete?
    if (survey > numOfSurveys - 1) {
        TerminalGUI.printLnError("A enquete de número " + survey + " não existe.");
        return;
    }

    // Você já votou nessa enquete?
    if (votes.contains(survey)) {
        TerminalGUI.printLnError("Você já votou nessa enquete.");
        return;
    }

    int option = Integer.parseInt(splittedArgs[1]);

    String counterName = ChatUtil.createCounterName(survey, option);

    Counter count = service.getOrCreateCounter(counterName, -1);

    long value = count.get();

    // Essa opção é válida?
    if (value < 0) {
        TerminalGUI.printLnError("A enquete de número " + survey + " não tem a opção " + option);
        service.deleteCounter(counterName);
        return;
    }

    // Incrementa o contador da opção
    count.incrementAndGet();

    // Adiciona a lista das enquetes votadas
    votes.add(survey);

    TerminalGUI.printLnSuccess("Voto contabilizado!");
}
```

Listing 10: Votando na enquete

2.4 Visualização dos eventos

A API do JGroups oferece algumas facilidades para acompanhamento do estado do *cluster* ao qual o processo está conectado. Utilizando-se de algumas dessas facilidades, implementamos maneiras de informar ao usuários sobre eventos que acontecem no *cluster* conectado atualmente.

Através da sobrescrita da implementação do método **viewAccepted**, da classe *ReceiverAdapter*, é possível receber atualizações sobre o *cluster*. No caso específico da implementação desejada, os evento que podem acontecer, relacionados com o estado do *cluster*, são aquele referentes à entrada ou saída de membros no chat.

Suponha que um processo qualquer, chamado A, está efetivamente conectado no chat X, e um outro processo se conecta, também, no chat X. Ao ocorrer a conexão completa do novo membro, a *view* do *cluster* é alterada, necessitando que os membros que já estavam conectados (no caso, somente o processo A) sejam informados dessa mudança. É neste método que implementamos a lógica que define o procedimento a ser feito, sendo composta basicamente de uma escrita na tela com o nome dos membros novos ou que saíram. Esse método é apresentado abaixo.

```
1     @Override
2     public void viewAccepted(View newView) {
3         // Se não estiver ativo, ignore
4         if (!isActive)
5             return;
6
7         // Uma nova atualização chegou
8         if (newView != null) {
9             // Obtêm os novos membros ao avaliar a diferença entre as views
10            List<Address> newMembers = View.newMembers(lastView, newView);
11
12            // Os mesmo para os membros que saíram
13            List<Address> exitedMembers = View.leftMembers(lastView, newView);
14
15            // Tira o próprio usuário da lista de novos membros
16            newMembers = newMembers.stream()
17                    .filter(a -> !a.equals(channel.getAddress()))
18                    .collect(Collectors.toList());
19
20            if (!newMembers.isEmpty()) {
21                String msg = newMembers.size() > 1 ? "Novos membros entraram: ": "Um novo membro entrou: ";
22                TerminalGUI.printLnInfo(msg + newMembers.toString());
23            }
24            if (!exitedMembers.isEmpty()) {
25                String msg = exitedMembers.size() > 1 ? "Alguns membros saíram: ": "Um membro saiu: ";
26                TerminalGUI.printLnInfo(msg + exitedMembers.toString());
27            }
28        }
29
30        // Atualiza a última view
31        lastView = newView;
32    }
```

Listing 11: Recebendo as atualizações da *view*

3 Cenários

Para descrever de forma mais aprofundada essa seção do relatório, preparamos um vídeo rápido onde as funcionalidades são apresentadas e explicadas de forma prática. Caso tenha problemas para acessar o vídeo via o link, entre em contato.

4 Limitações

Essa implementação, apesar de útil no âmbito acadêmico e de experimentação prática, possui alguns problemas relacionados tanto com a teoria da computação distribuída, como com o domínio do problema que selecionamos para tratar.

4.1 Chats múltiplos

Usuários que tentem manter várias conexões em vários chats ao mesmo tempo, podem enfrentar problemas decorrentes do uso excessivo de memória, devido ao uso de objetos para representar cada chat individualmente. Essa limitação, no entanto, não deve ser um problema para a grande maioria dos usuários.

4.2 Interface gráfica

A falta de uma interface gráfica completa e rica em interação acaba limitando, e em alguns casos pode até causar certo desconforto, o envio das mensagens. Repare que as regras para o envio de comandos e mensagens compartilham um mesmo mecanismo de envio, podendo acarretar em erros e futuras vulnerabilidades. O ideal, nesse caso, seria isolar, em uma camada lógica, os comandos das mensagens, permitindo ao usuário enviar mensagens como !sair, sem executar o comando !sair realmente.

4.3 Permissões por usuário

Não existe uma separação clara de poderes entre coordenador do grupo e membros, no domínio do problema. Isso significa que dentro da aplicação não existem administradores. O JGroups oferece uma facilidade ao executar algoritmos de eleição de líder para definir o coordenador, e caso fosse decidido implementar a ideia de administradores, seria necessário reimplementar essa lógica para que a aplicação fosse útil.

Referências

[Ban] Bela Ban. Reliable group communication with jgroups.