DIEGO SOUSA SANTOS – 11044616

UFABC

RELATÓRIO EP-KVSTORE – SISTEMAS DISTRIBUÍDOS

SÃO CAETANO DO SUL – SP

11/2022

LINK DO VÍDEO: https://youtu.be/4J3EH-njqiA

1. FORMATO DA MENSAGEM TRANSFERIDA

As mensagens são transferidas através do objeto Message, descrito no diagrama UML abaixo:

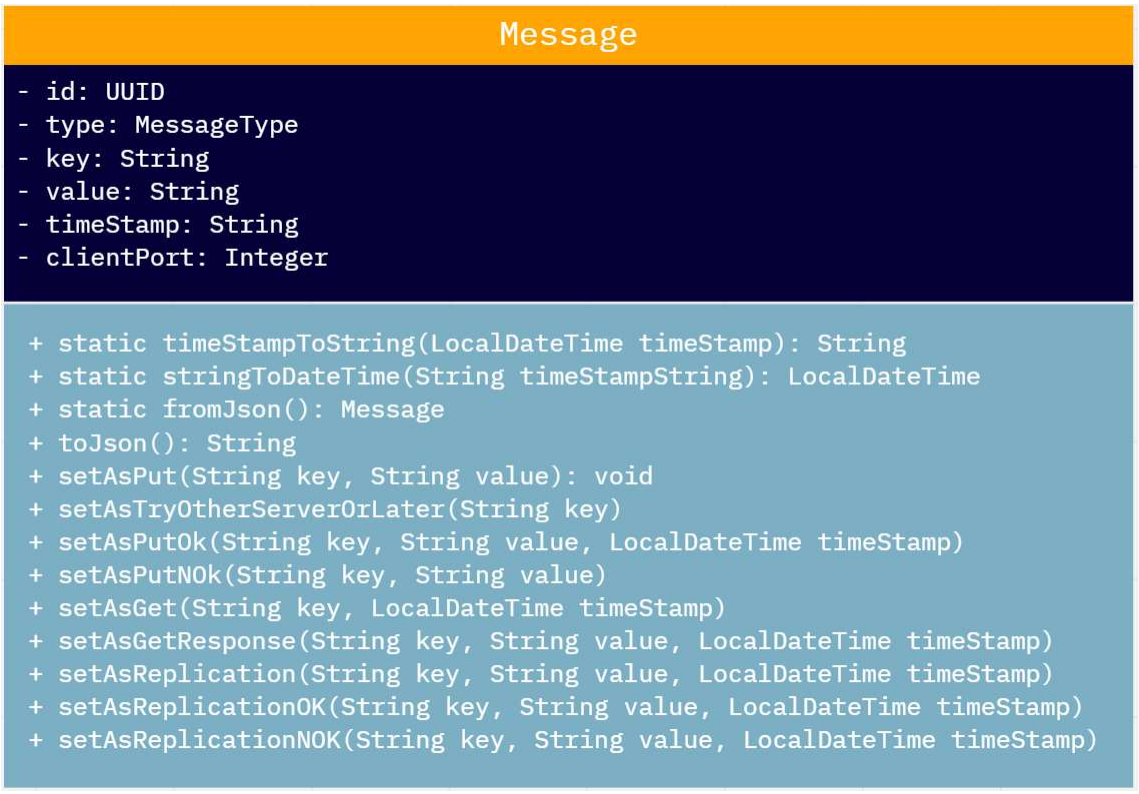


Figura 1 UML da classe Message

O atributo "id" define o identificador da mensagem através de um UUID que permite tornar única cada mensagem.

O atributo "type" é um struct que classifica a mensagem de acordo com os tipos PUT, GET, REPLICATION, GET_RESPONSE, PUT_OK, PUT_NOK, TRY_OTHER_SERVER_OR_LATER, REPLICATION_OK, REPLICATION_NOK, possibilitando diferenciar esses tipos de mensagens.

O atributo "key" é o identificador do objeto referenciado na mensagem.

O atributo "value" é o valor do objeto referenciado na mensagem.

O atributo "timeStamp" é a referência temporal do cadastro do objeto referenciado na mensagem.

O atributo "clientPort" é a porta de origem da mensagem.

O método "initializeUUID" gera o UUID da mensagem.

O método "toJson" utiliza a biblioteca gson para converter o objeto Mensagem em Json.

Os métodos “setAs...” classificam e criam a mensagem de acordo com os parâmetros exigidos para cada tipo especificado.

O método estático “timeStampToString” retorna uma String a partir de um LocalDateTime.

O método estático “stringToDateTime” retorna um LocalDateTime a partir de uma String.

O método estático “fromJson” retorna uma Mensagem a partir de um json.

2. CLIENT

2.1 MÉTODO MAIN

No método main existe um loop para leitura dos dados do usuário. É impresso um menu composto das opções “1. INIT”, “2. PUT” e “3. GET”, as quais o usuário pode selecionar informando o número ou o texto da opção.

Caso a opção de INIT seja selecionada, o usuário deverá informar as três portas dos servidores envolvidos no sistema distribuído. Após isso, cada porta especificada é inserida no Set “serverPorts” do cliente, para armazenar os servidos conhecidos.

Caso a opção de PUT seja selecionada, o usuário deverá informar primeiro a chave e, logo após, o valor do objeto key-value a ser registrado nos servidores.

Caso a opção de GET seja selecionada, o usuário deverá informar apenas a chave do objeto procurado.

```
156     // MÉTODO MAIN ----------------------------------------
157
158     public static void main(String[] args) {
159         keyboard = new Scanner(System.in);
160         Client client = null;
161
162         while (true) {
163             System.out.println("1. INIT\n2. PUT\n3. GET");
164
165             keyboard = new Scanner(System.in);
166             String option = keyboard.next();
167
168             if (option.isEmpty())
169                 option = null;
170
171             if (option.contains("1") || option.toUpperCase().contains("INIT")) {
172                 System.out.println("Informe as portas dos servidores: ");
173
174                 // inicializando cliente e referenciando os servidores
175                 client = new Client();
176                 client.addServerPort(keyboard.nextInt());
177                 client.addServerPort(keyboard.nextInt());
178                 client.addServerPort(keyboard.nextInt());
179
180             } else if (option.contains("2") || option.toUpperCase().contains("PUT")) {
181                 // captura informações do teclado e envia mensagem de put
182                 System.out.print("Informe a chave: ");
183                 keyboard.nextLine();
184                 String key = keyboard.nextLine();
185
186                 System.out.print("\nInforme o valor: ");
187                 String value = keyboard.nextLine();
188
189                 client.sendPutMessage(key, value);
190             } else if (option.contains("3") || option.toUpperCase().contains("GET")) {
191                 // captura informações do teclado e envia mensagem de get
192                 System.out.print("Informe a chave: ");
193                 keyboard.nextLine();
194                 String key = keyboard.nextLine();
195
196                 client.sendGetMessage(key);
197             } else
198                 continue;
199
200         }
201     }
```

Figura 2 método main da classe Client

2.2 DEFINIÇÃO DO CLIENT

Os peers são definidos através da classe descrita no diagrama UML abaixo:

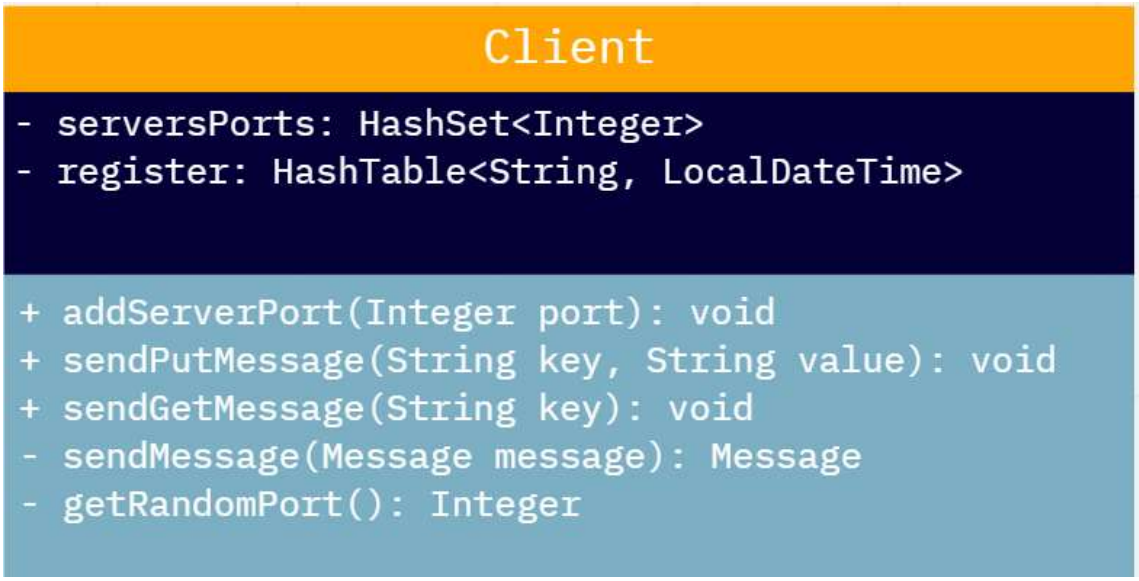


Figura 3 UML da classe Client

2.2.1 ATRIBUTOS

O atributo "serversPorts" conjunto de portas únicas de servidores conhecidos, informados no método main.

O atributo "register" armazena o registro que foram cadastrados com sucesso nos servidores através de mensagens PUT.

2.2.2 MÉTODOS

addServerPort:

```
35    /**
36     * Adiciona portas de servidores à lista
37     * @param port - porta do servidor
38     */
39    public void addServerPort(Integer port) {
40        serversPorts.add(port);
41    }
42
```

Figura 4 método addServerPort

sendPutMessage: Envia de forma assíncrona uma mensagem do tipo PUT com os parâmetros informados pelo usuário. Se a resposta da requisição for um PUT_OK, o registro é salvo no register do client.

```java
    /**
     * Envia uma mensagem do tipo PUT a um servidor aleatória da lista
     * De servidores
     * @param key - chave do novo item
     * @param value - valor do novo item
     */
    public void sendPutMessage(String key, String value) {
        Thread th = new Thread(() -> {
            // criando mensagem
            Message message = new Message();
            message.setAsPut(key, value);

            // aguardando resposta
            Message response = sendMessage(message);

            // registrando key + timestamp
            if(response.getType() == MessageType.PUT_OK)
                register.put(key, response.getTimeStamp());
            else
                System.out.println("The put request was not successful");
        });

        th.start();
    }
```

Figura 5 método sendPutMessage

sendGetMessage: Envia de forma assíncrona uma mensagem do tipo GET com os parâmetros informados pelo usuário.

```java
    /**
     * Envia uma mensagem do tipo GET a um servidor aleatório da lista
     * @param key - chave do item buscado
     */
    public void sendGetMessage(String key) {
        Thread th = new Thread(() -> {

                // criando mensagem
                Message message = new Message();
                message.setAsGet(key, register.get(key));

                // aguardando resposta
                Message response = sendMessage(message);

        });

        th.start();
    }
```

Figura 6 método sendGetMessage

sendMessage: Envia a mensagem genérica (PUT ou GET) para um dos servidores registrados aleatoriamente. Após isso, aguarda a resposta e imprime o resultado na tela.

```java
/**
 * Envio de mensagens aos servidores
 * @param message - mensagem a ser enviada
 * @return resposta do servidor
 */
private Message sendMessage(Message message) {
    try {
        // pegando servidor aleatório
        Integer serverPort = getRandomPort();

        // criando socket e streams de escrita/leitura
        Socket s = new Socket("127.0.0.1", serverPort);
        OutputStream os = s.getOutputStream();
        DataOutputStream writer = new DataOutputStream(os);
        InputStreamReader is = new InputStreamReader(s.getInputStream());
        BufferedReader reader = new BufferedReader(is);

        // enviando mensagem
        message.setClientPort(s.getLocalPort());
        String msgJson = message.toJson();
        writer.writeBytes(msgJson + "\n");
        // aguardando o retorno
        String response = reader.readLine();
        Message responseMsg = Message.fromJson(response);

        String msgType = MessageType.getName(responseMsg.getType());
        // imprimindo resultado
        System.out.println(String.format("%s key '%s' value '%s' timestamp %s %s no servidor %s:%d",
                msgType, responseMsg.getKey(), responseMsg.getValue(),
                Message.timeStampToString(responseMsg.getTimeStamp()),
                (msgType == "GET" || msgType == "TRY_OTHER_SERVER_OR_LATER") ? "obtida": "realizada","127.0.0.1", serverPort));

        s.close();
        return responseMsg;

    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

Figura 7 método sendMessage

getRandomPort: retorna uma porta aleatória dentre as que estão registradas na lista serverPorts.

```java
/**
 * Seleciona aleatoriamente uma porta da lista de portas
 * @return porta de servidor aleatória
 */
private Integer getRandomPort() {
    // criando random
    Random random = new Random();
    int randomIndex = random.nextInt(serversPorts.size());
    // capturando iterator
    Iterator<Integer> iterator = serversPorts.iterator();

    int currentIndex = 0;
    Integer randomElement = null;

    // iterate the HashSet
    while (iterator.hasNext()) {

        randomElement = iterator.next();

        // verificando o index igual ao número randômico
        if (currentIndex == randomIndex)
            return randomElement;

        currentIndex++;
    }

    return randomElement;
}
```

Figura 8 método getRandomPort

3. SERVER

3.1 MÉTODO MAIN

No método main são capturados na sequência: IP do servidor, porta do servidor e porta do líder. Após isso, a instância do servidor é inicializada.

```java
    // MÉTODO MAIN E DEPENDÊNCIAS ------------------------------------

    private static Scanner keyboard;

    public static void main(String[] args) throws Exception {

        keyboard = new Scanner(System.in);
        System.out.println("Informe o IP: ");
        String ip = keyboard.next();
        System.out.println("Informe a porta: ");
        Integer port = keyboard.nextInt();

        System.out.println("Informe a porta do líder: ");
        Integer leaderPort = keyboard.nextInt();

        try {
            new Server(ip, port, leaderPort);
            System.out.println("Servidor online");
        } catch (UnknownHostException e) {
            throw new Exception("Falha ao inicializar servidor: " + e.getMessage());
        }

    }
```

Figura 9 método main da classe Server

3.2 DEFINIÇÃO DO SERVER

Os peers são definidos através da classe descrita no diagrama UML abaixo:

Figura 10 UML da classe Server

3.2.1 ATRIBUTOS

O atributo “serverSocket” referencia a instância de ServerSocket criada juntamente com o objeto Server.

O atributo “ip” refere-se ao ip do servidor.

O atributo “port” refere-se à porta do servidor.

O atributo “leaderPort” registra a porta do servidor líder do sistema distribuído.

O atributo “register” é uma hashtable que armazena os objetos key-value que são registrados no servidores através das mensagens do cliente.

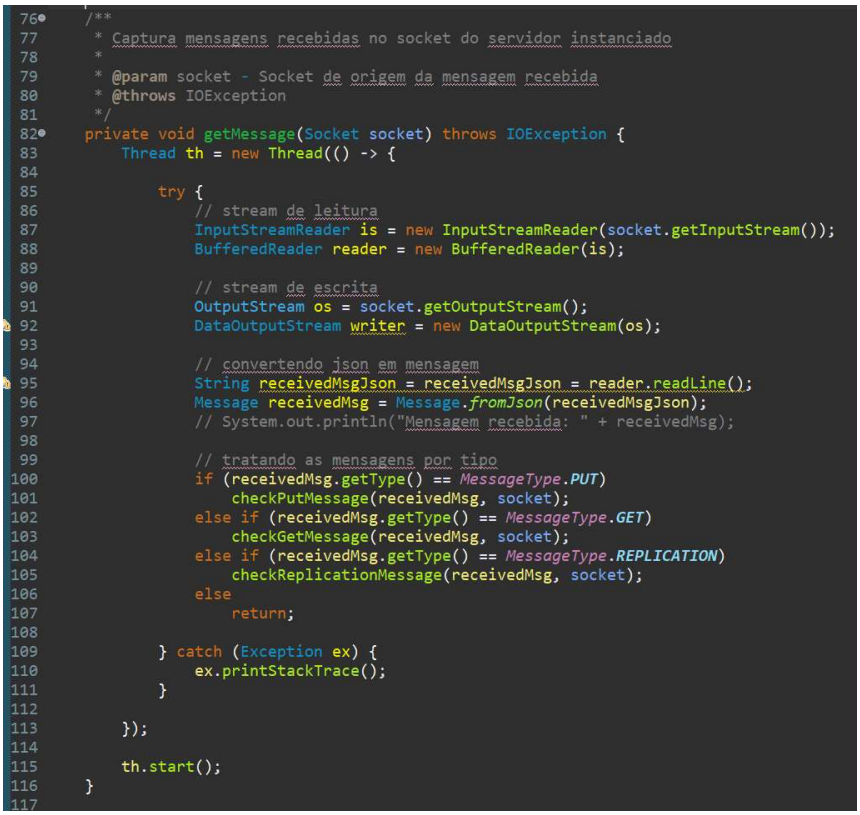
3.2.2 MÉTODOS

createSocket: responsável por inicializar o socket da instância de Server em questão. O socket é inicializado de forma assíncrona, e após isso é criado um loop para identificar e tratar o recebimento de mensagens.

```
43●     /**
44       * Cria o Socket de comunicação do servidor instanciado O recebimento de
45       * mensagens é assíncrono
46       */
47●     private void createSocket() {
48          Thread th = new Thread(() -> {
49              // canal de comunicação não orientado à conexão
50              serverSocket = null;
51              try {
52                  // criando socket
53                  serverSocket = new ServerSocket(port);
54              } catch (IOException e) {
55                  e.printStackTrace();
56              }
57
58              while (true && serverSocket != null) {
59                  try {
60                      // recebendo pacote
61                      Socket client = serverSocket.accept();
62                      // capturando mensagem
63                      getMessage(client);
64
65                  } catch (IOException e) {
66                      if (!e.getMessage().toUpperCase().equals("Socket Closed".toUpperCase()))
67                          e.printStackTrace();
68                  }
69              }
70          });
71
72          th.start();
73      }
```

Figura 11 método createSocket

getMessage: responsável por, de forma assíncrona, capturar o pacote recebido com a mensagem do client/server e, a partir do tipo da mensagem direcionar o tratamento correto.



```java
/**
 * Captura mensagens recebidas no socket do servidor instanciado
 *
 * @param socket - Socket de origem da mensagem recebida
 * @throws IOException
 */
private void getMessage(Socket socket) throws IOException {
    Thread th = new Thread(() -> {

        try {
            // stream de leitura
            InputStreamReader is = new InputStreamReader(socket.getInputStream());
            BufferedReader reader = new BufferedReader(is);

            // stream de escrita
            OutputStream os = socket.getOutputStream();
            DataOutputStream writer = new DataOutputStream(os);

            // convertendo json em mensagem
            String receivedMsgJson = receivedMsgJson = reader.readLine();
            Message receivedMsg = Message.fromJson(receivedMsgJson);
            // System.out.println("Mensagem recebida: " + receivedMsg);

            // tratando as mensagens por tipo
            if (receivedMsg.getType() == MessageType.PUT)
                checkPutMessage(receivedMsg, socket);
            else if (receivedMsg.getType() == MessageType.GET)
                checkGetMessage(receivedMsg, socket);
            else if (receivedMsg.getType() == MessageType.REPLICATION)
                checkReplicationMessage(receivedMsg, socket);
            else
                return;

        } catch (Exception ex) {
            ex.printStackTrace();
        }

    });

    th.start();
}
```

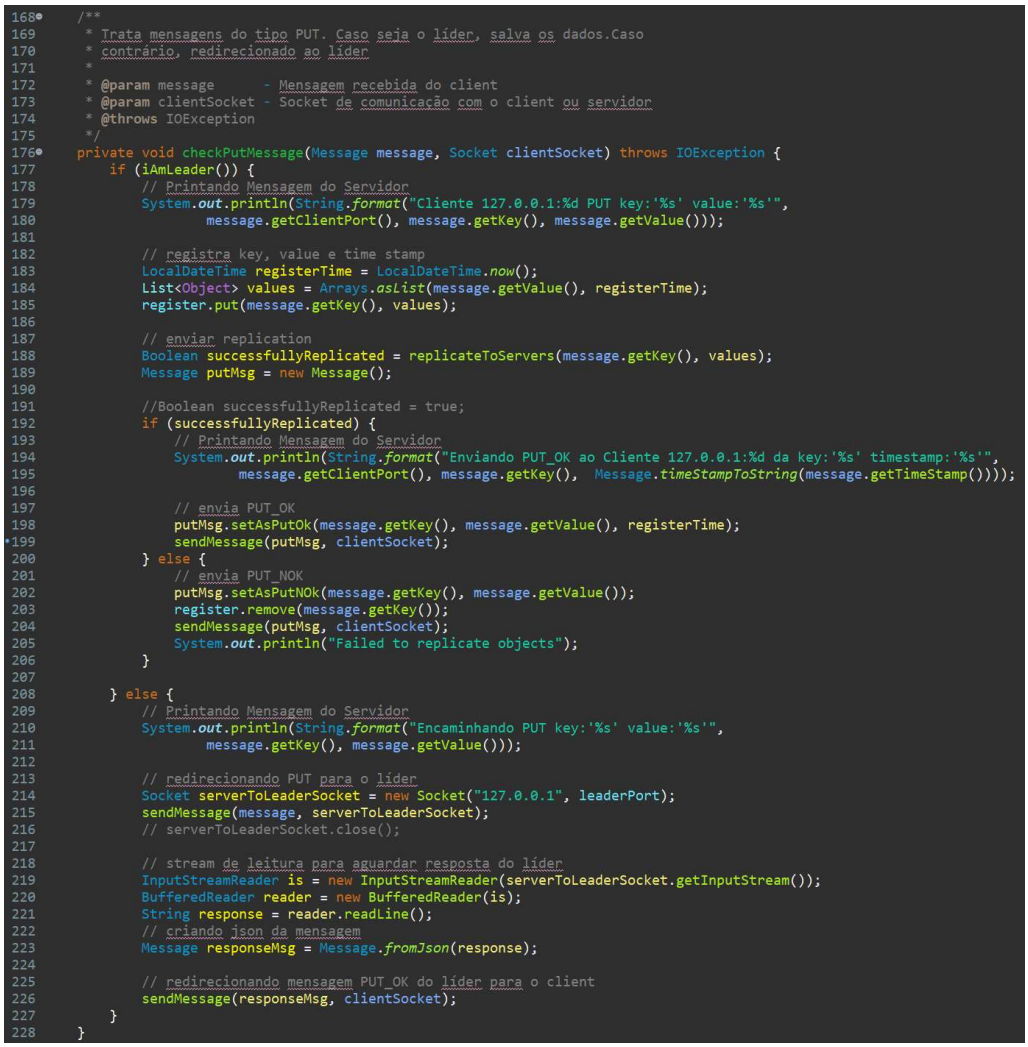
Figura 12 método getMessage

checkGetMessage: responsável por tratar as mensagens recebidas do tipo GET. Com a chave informada na mensagem, o servidor procura o objeto no register e, caso não exista, retorna nulo. Caso exista, o servidor retorna o objeto procurado. OBSERVAÇÃO: para simular um pacote desatualizado foi criada uma variável random (linhas 137 a 140) para gerar falhas aleatórias, com 30% de chances de ocorrer. Isso porque como o sistema roda em local host, a chance de ocorrerem falhas na replicação é praticamente nula.

```java
/**
 * Trata mensagens do tipo GET.
 *
 * @param message      - Mensagem recebida do client
 * @param clientSocket - Socket de comunicação com o client ou servidor
 * @throws IOException
 */
private void checkGetMessage(Message message, Socket socket) throws IOException {
    String key = message.getKey();
    Message response = new Message();
    List<Object> objValues = register.get(key);

    // verifica se a chave está registrada
    if(objValues != null) {
        // caso esteja registrada, retorna o valor
        LocalDateTime timeStampRef = (LocalDateTime)objValues.get(1);

        // índice randômico para simular falhas
        Double rand = new Random().nextDouble();

        // verificando se o registro do servidor é mais antigo ou igual ao do cliente
        if(rand > 0.3 && (timeStampRef.isBefore(message.getTimeStamp()) || timeStampRef.isEqual(message.getTimeStamp()))) {
            // Printando Mensagem do Servidor
            System.out.println(String.format("Cliente 127.0.0.1:%d GET key:'%s' timestamp:'%s'. "
                    + "Meu timestamp é '%s', portanto devolvendo %s",
                    message.getClientPort(), message.getKey(), Message.timeStampToString(message.getTimeStamp()),
                    Message.timeStampToString((LocalDateTime)objValues.get(1)), (String)objValues.get(0)));

            response.setAsGetResponse(key, (String)objValues.get(0), (LocalDateTime)objValues.get(1));

        }
        else {
            // Printando Mensagem do Servidor
            System.out.println(String.format("Cliente 127.0.0.1:%d GET key:'%s' timestamp:'%s'. "
                    + "SIMULAÇÃO DE CHAVE DESATUALIZADA, portanto devolvendo null (TRY_OTHER_SERVER_OR_LATER)",
                    message.getClientPort(), message.getKey(), Message.timeStampToString(message.getTimeStamp()),
                    Message.timeStampToString((LocalDateTime)objValues.get(1)), (String)objValues.get(0)));

            response.setAsTryOtherServerOrLater(key);
        }

    } else {
        // caso não esteja registrada, retorna valor nulo
        response.setAsGetResponse(key, null, message.getTimeStamp());
    }

    sendMessage(response, socket);
}
```

Figura 13 método checkGetMessage

checkPutMessage: responsável por tratar as mensagens do tipo PUT. A partir dos parâmetros recebidos, se o servidor for líder o objeto é cadastrado no register. Caso contrário, a mensagem é redirecionada ao líder. Após isso, o líder replica o objeto para os demais servidores e, ocorrendo a replicação com sucesso, a resposta é enviada ao cliente como PUT_OK, com o timestamp do registro. Caso contrário, é enviado um PUT_NOK confirmando a falha no registro.



```java
/**
 * Trata mensagens do tipo PUT. Caso seja o líder, salva os dados.Caso
 * contrário, redirecionado ao líder
 *
 * @param message      - Mensagem recebida do client
 * @param clientSocket - Socket de comunicação com o client ou servidor
 * @throws IOException
 */
private void checkPutMessage(Message message, Socket clientSocket) throws IOException {
    if (iAmLeader()) {
        // Printando Mensagem do Servidor
        System.out.println(String.format("Cliente 127.0.0.1:%d PUT key:'%s' value:'%s'",
                message.getClientPort(), message.getKey(), message.getValue()));

        // registra key, value e time stamp
        LocalDateTime registerTime = LocalDateTime.now();
        List<Object> values = Arrays.asList(message.getValue(), registerTime);
        register.put(message.getKey(), values);

        // enviar replication
        Boolean successfullyReplicated = replicateToServers(message.getKey(), values);
        Message putMsg = new Message();

        //Boolean successfullyReplicated = true;
        if (successfullyReplicated) {
            // Printando Mensagem do Servidor
            System.out.println(String.format("Enviando PUT_OK ao Cliente 127.0.0.1:%d da key:'%s' timestamp:'%s'",
                    message.getClientPort(), message.getKey(),  Message.timeStampToString(message.getTimeStamp())));

            // envia PUT_OK
            putMsg.setAsPutOk(message.getKey(), message.getValue(), registerTime);
            sendMessage(putMsg, clientSocket);
        } else {
            // envia PUT_NOK
            putMsg.setAsPutNOk(message.getKey(), message.getValue());
            register.remove(message.getKey());
            sendMessage(putMsg, clientSocket);
            System.out.println("Failed to replicate objects");
        }

    } else {
        // Printando Mensagem do Servidor
        System.out.println(String.format("Encaminhando PUT key:'%s' value:'%s'",
                message.getKey(), message.getValue()));

        // redirecionando PUT para o líder
        Socket serverToLeaderSocket = new Socket("127.0.0.1", leaderPort);
        sendMessage(message, serverToLeaderSocket);
        // serverToLeaderSocket.close();

        // stream de leitura para aguardar resposta do líder
        InputStreamReader is = new InputStreamReader(serverToLeaderSocket.getInputStream());
        BufferedReader reader = new BufferedReader(is);
        String response = reader.readLine();
        // criando json da mensagem
        Message responseMsg = Message.fromJson(response);

        // redirecionando mensagem PUT_OK do líder para o client
        sendMessage(responseMsg, clientSocket);
    }
}
```

Figura 14 método checkPutMessage

checkReplicationMessage: responsável por tratar mensagens do tipo Replication. Com base nos parâmetros recebidos na mensagem, o objeto é cadastrado no register do servidor caso ele não seja o líder e devolve uma mensagem REPLICATION_OK. Caso contrário, devolve uma mensagem REPLICATION_NOK.

```java
    /**
     * Trata mensagens do tipo Replication recebidas
     * @param receivedMsg - mensagem recebida
     * @param socket - socket de origem da mensagem
     * @throws IOException
     */
    private void checkReplicationMessage(Message receivedMsg, Socket socket) throws IOException {

        try {
            // Printando Mensagem do Servidor
            System.out.println(String.format("REPLICATION key:'%s' value:'%s' timestamp: '%s'",
                    receivedMsg.getKey(), receivedMsg.getValue(), Message.timeStampToString(receivedMsg.getTimeStamp())));

            // registra a key e os valores no register
            register.put(receivedMsg.getKey(), Arrays.asList(receivedMsg.getValue(), receivedMsg.getTimeStamp()));
            Message message = new Message();
            // enviando resposta
            message.setAsReplicationOK(receivedMsg.getKey(), receivedMsg.getValue(), receivedMsg.getTimeStamp());
            sendMessage(message, socket);

        } catch (Exception ex) {
            // em caso de falha, envia REPLICATION_NOK e printa o erro
            Message message = new Message();
            message.setAsReplicationNOK(receivedMsg.getKey(), receivedMsg.getValue(), receivedMsg.getTimeStamp());
            sendMessage(message, socket);
            System.out.println(ex.getMessage());
        }
    }
```

Figura 15 método checkReplicationMessage

replicateToServers: responsável por replicar, a partir do líder, uma mensagem aos demais servidores. A lista de portas de servidores está fixada em 10097, 10098 e 10099. Para cada porta, o método mapeia um socket para enviar uma mensagem do tipo REPLICATION para cadastro dos objetos. Após isso o líder aguarda a resposta e o método devolve true ou false de acordo com ela.

```java
    /**
     * Replica um objeto chave-valor para os demais servidores
     * @param key - chave
     * @param values - lista de tamanho 2 com valor e timestamp
     * @return true ou false
     * @throws IOException
     */
    private Boolean replicateToServers(String key, List<Object> values) throws IOException {
        Set<Integer> serversPorts = new HashSet<>();
        // portas fixadas hard-coded
        serversPorts.add(10097);
        serversPorts.add(10098);
        serversPorts.add(10099);

        // redirecionando para outros servidores
        for (Integer port : serversPorts) {
            // ignora o próprio líder
            if (!port.equals(this.port)) {
                Message message = new Message();
                message.setAsReplication(key, (String) values.get(0), (LocalDateTime) values.get(1));

                Socket serverSocket = null;
                try {
                    // caso não exista servidor em uma das portas, ignora
                    serverSocket = new Socket("127.0.0.1", port);
                } catch(Exception ex) {
                    continue;
                }

                // enviando mensagem
                sendMessage(message, serverSocket);

                // stream de leitura para aguardar resposta
                InputStreamReader is = new InputStreamReader(serverSocket.getInputStream());
                BufferedReader reader = new BufferedReader(is);
                String response = reader.readLine();

                // criando json da mensagem
                Message responseMsg = Message.fromJson(response);

                // se qualquer um dos servidores responder como NOK, retorna falso
                if (responseMsg.getType() == MessageType.REPLICATION_NOK)
                    return false;

                System.out.println("Replicado com sucesso para [" + port + "]");
                serverSocket.close();
            }
        }

        return true;
    }
```

Figura 16 método replicateToServers

iAmLeader: responsável por identificar se o servidor em questão é o líder ou não.

```
311     /**
312      * Verifica se a instância de servidor atual é o líder
313      * @return Boolean true ou false
314      */
315     private Boolean iAmLeader() {
316         return leaderPort.equals(port);
317     }
318
```

Figura 17 método iAmLeader

sendMessage: responsável por enviar uma mensagem qualquer à um socket qualquer.

```
319     /**
320      * Envia uma mensagem ao socket informado
321      *
322      * @param message    - Mensagem a ser enviada
323      * @param destSocket - Socket de destino
324      * @throws IOException
325      */
326     private void sendMessage(Message message, Socket destSocket) throws IOException {
327         // stream de escrita no socket de origem
328         OutputStream os = destSocket.getOutputStream();
329         DataOutputStream writer = new DataOutputStream(os);
330         // convertendo mensagem para json
331         String msgJson = message.toJson();
332         // enviando mensagem
333         writer.writeBytes(msgJson + "\n");
334     }
335
```

Figura 18 método sendMessage