UNIVERSIDADE FEDERAL DO ABC

**SISTEMAS DISTRIBUÍDOS**

Prof. Dr. Vladimir Emiliano Moreira Rocha

# Projeto EP2

Gabriel Sponda Freitas Bettarello

11201932580

Santo André, 2023

## *Introdução*

*O sistema foi desenvolvido de acordo com os requisitos do projeto. Foram criadas as seguintes classes: Cliente.java, Mensagem.java e Servidor.java. Além disso foram usadas três classes aninhadas: ThreadSendRequest.java (em Cliente.java), ThreadReceiveRequest.java e ThreadReplication.java (em Servidor.java).*

*Neste documento, você encontrará instruções completas sobre a operação adequada do programa, além de informações detalhadas sobre sua implementação. Uma das seções contém a visão geral da implementação sem muitos detalhes. Os detalhes de implementação foram organizados em três seções principais, correspondentes às classes. Dentro de cada seção, são abordados aspectos específicos, como o uso das threads. Note que a última seção foi dedicada para a simulação de um atraso e, consequentemente, a simulação de um erro "TRY_OTHER_SERVER_OR_LATER".*

## *Link do vídeo:*

https://youtu.be/93vA1AKvPt8

## 1. Instruções

Para operar o programa corretamente, siga estas etapas:

- Dentro da pasta “src” encontram-se os arquivos “Cliente.java”, “Mensagem.java”, “Servidor.java” e “gson-2.10.1.jar”. Acesse a pasta e compile os arquivos “.java” pelo terminal usando os comandos “javac -cp .;gson-2.10.1.jar Cliente.java”, “javac Mensagem.java” e “javac -cp .;gson-2.10.1.jar Servidor.java”.

- A partir do terminal, ainda na pasta “src”, inicialize três servidores utilizando o comando “java -cp .;gson-2.10.1.jar Servidor”, e inicialize dois clientes utilizando o comando “java -cp .;gson-2.10.1.jar Cliente”. Um identificador será exibido no topo de cada console.

- Em seguida, entre com os dados dos servidores. Primeiro, responda se o servidor é o líder, depois decida se quer utilizar as informações padrões. Caso opte por usar as informações padrões no servidor líder, este será o servidor 127.0.0.1:10099, é recomendado que opte por usar as informações padrões durante a inicialização dos outros dois servidores, informando apenas a porta de cada um deles, que deve ser 10097 para um e 10098 para o outro. Caso opte por não usar as informações padrões, basta digitar as informações conforme é pedido. Um novo identificador aparece no topo de cada servidor.

- No console do cliente, uma linha começando com “MENU: “ é exibida, use essa linha para selecionar a opção do menu. Entre com a opção 1 (INIT) em cada cliente. Caso opte por inserir as informações padrões, ele automaticamente preencherá os campos destinados aos IPs e portas de escuta de cada servidor, caso contrário, você deve preenchê-las.

- Após fazer a inicialização do cliente, novamente a linha de seleção de opção aparece e você pode optar por fazer um “PUT” (opção 2) ou um “GET” (opção 3). Qualquer que seja a escolha, basta digitar o que for requisitado. Você não precisa aguardar uma resposta para entrar com uma nova requisição, porém, como o sistema roda em localhost, o tempo de resposta é muito alto.

- Para simular um atraso na resposta, utilize a opção “PUT”, entre com a chave “teste” e um valor qualquer. Note que imediatamente após o envio, a linha de opção é exibida e você

pode fazer uma nova requisição. Quando o servidor responder, uma mensagem será exibida no console. Note que nesse momento a linha de seleção de opção pode ficar deslocada para cima da resposta, mas ao digitar um número, ele aparecerá abaixo da resposta, o que não impede o uso do programa, basta que digite a opção desejada e tecle enter.

## 2. Visão geral da solução

Foram usados mapas de hash para armazenar as informações, tanto no cliente quanto nos servidores.

A ideia é que o cliente possa enviar requisições através de threads, permitindo que faça novas requisições enquanto aguarda pela resposta. O cliente cria uma instância de Mensagem e a envia para a thread (ThreadSendRequest), que por sua vez abre uma comunicação com um servidor, acrescenta na Mensagem o IP e a porta do cliente utilizada nessa comunicação e a encaminha para um servidor aleatório, utilizando a biblioteca gson para transformar a Mensagem em um Json. Após o envio, a thread fecha o socket que usou para se comunicar com o servidor, liberando a porta e então a transforma numa porta de escuta para aguardar que o servidor faça um pedido de conexão e envie a resposta.

Os servidores recebem o pedido de conexão e o enviam para uma thread (ThreadReceiveRequest) para serem tratados por ela. Ao receber a requisição de “GET”, qualquer servidor a processa, fazendo as checagens necessárias de acordo com as instruções do projeto. O servidor converte a mensagem recebida de Json para Mensagem usando a biblioteca gson. O servidor então abre um socket utilizando o IP e a porta recebidas pela Mensagem e, com a comunicação aceita, inicia o envio da resposta.

Caso não seja o líder, ao receber a requisição de “PUT”, repassa ao líder enviando uma requisição através da Mensagem que recebeu, como se fosse um cliente. O líder então processa o “PUT” e inicia duas threads (ThreadReplication) para a replicação nos servidores secundários, aguarda pela resposta de “REPLICATION_OK” duas vezes, e então cria um socket utilizando os dados do cliente, que está com a porta de escuta ativa, e faz o envio da resposta.

O cliente, ao receber a resposta, fecha a comunicação com o servidor, verifica se a requisição foi bem-sucedida ou não e a thread é eliminada.

Note que após fazer um “GET” ou um “PUT”, o cliente vai salvar o timestamp recebido pelo servidor em seu mapa de hash local.

Os detalhes estão nas seções abaixo.

## 3. Cliente

A estrutura escolhida para o armazenamento de dados foi um ConcurrentHashMap, para evitar alguns problemas de concorrência. Nesse mapa, a chave é uma *String* que contém a chave (key), enquanto o valor armazenado é um Integer, que representa o timestamp do cliente relacionado à chave.

De modo geral, o cliente pode enviar requisições “PUT” e “GET” após ser inicializado. Em ambos os casos, cria-se uma instância de Mensagem e a envia para uma thread, que é executada e permite que o cliente possa realizar outras requisições enquanto aguarda pela resposta, em outras palavras, as requisições do cliente não são bloqueantes. As três opções do menu são detalhadas abaixo, juntamente à thread usada (ThreadSendRequest).

### 3.1. INIT *[linha 103]*

Ao ser inicializado, o cliente deve utilizar a opção “INIT” *[linha 101]* do menu, que consiste na captura, a partir do teclado, das informações dos servidores.

Em seguida, o usuário pode escolher entre fazer um “PUT” ou um “GET”, nos dois casos é verificado se o cliente já fez o “INIT”.

### 3.2. PUT *[linha 128]*

Caso opte por fazer um “PUT” *[linha 129]*, deverá informar a chave (key) e valor (value) a serem inseridos. Uma verificação é feita para saber se essa chave já existe no

mapa de hash local e, caso não exista, é inserida e o timestamp é inicializado em 0. Com a garantia de que ela existe, o mapa é atualizado, incrementando o timestamp relacionado a essa chave. Em seguida, uma instância de Mensagem é criada, passando como parâmetros a chave, o valor, o timestamp e o tipo de requisição, que nesse caso é "PUT". Então esse objeto é enviado para uma thread que é executada e permite que o usuário faça novas requisições enquanto aguarda a resposta. Tanto a classe Mensagem quanto a classe ThreadSendRequest serão abordadas mais adiante.

## 3.3. GET *[linha 155]*

Se a opção escolhida for "GET" *[linha 156]*, deve informar a chave para a qual se deseja saber o valor armazenado no servidor. Caso a chave não exista no mapa local, é inserida com um timestamp igual a 0, mas, diferentemente do "PUT", o timestamp não é incrementado e é usado para comparações entre o cliente e o servidor. Uma instância de Mensagem é criada, passando como parâmetros a chave, um valor null, já que é indiferente para esse tipo de requisição, o timestamp e "GET". O objeto é enviado para uma thread que é executada e também permite o cliente realizar outras requisições enquanto aguarda pela resposta.

## 3.4. ThreadSendRequest *[linha 21]*

A classe ThreadSendRequest é responsável por fazer o envio das requisições e também por aguardar a resposta e processá-la. Ao ser iniciada, cria o mecanismo de comunicação com um servidor aleatório, utilizando um número gerado aleatoriamente e usando-o como índice do vetor que contém os IPs e no vetor que contém as portas dos servidores.

Utilizando métodos específicos, captura o IP e a porta do cliente *[linha 39]* que estão sendo usadas nessa comunicação, acrescenta esses dados através do construtor adequado da classe Mensagem, converte a mensagem para Json e envia para o servidor *[linha 44]*.

Após o envio, fecha o socket que estava sendo usado na comunicação e utiliza a mesma porta para criar um server socket. Cria um novo socket que vai receber o aceite da comunicação na nova porta de escuta *[linha 48]*. Após o aceite da conexão, ou seja, no momento em que o servidor já pede para enviar a resposta, o cliente cria o mecanismo para fazer a leitura, recebe a mensagem em Json e a converte para Mensagem para que possa fazer sua análise *[linha 53]*.

No campo String request do construtor da Mensagem, o servidor coloca o tipo de resposta, ou seja, "PUT_OK", os erros ou então o valor armazenado relacionado à chave buscada. Imprime no console do cliente de acordo com o valor recebido *[linha 56]*. Se o cliente receber um "PUT_OK" ou qualquer valor como resposta de um "GET", ele atualiza o timestamp relacionado à chave *[linha 59 e linha 66]*, copiando o valor do servidor para seu mapa de hash local.

## 4. Mensagem

A classe Mensagem é usada para a comunicação entre cliente e servidor, conforme requisitado nas instruções do projeto. Possui como parâmetros a String key, a String value, o Integer timestamp, a String request, a String clientIP e o int clientPort. Esses são os parâmetros que precisamos enviar entre as comunicações.

Note que essa classe possui dois métodos construtores, sendo que um deles dispensa o IP e a porta do cliente. Cada um desses métodos é usado em diferentes situações e ficará mais claro ao longo do texto, mas repare que o objeto criado pelo cliente durante à escolha da requisição é o método no qual não precisamos informar o IP e a porta, que serão obtidos durante a execução da thread de envio de requisição.

## 5. Servidor

A classe Servidor também utiliza um ConcurrentHashMap para armazenar os valores relacionados às chaves, mas neste caso usa um List<Object> como "valor" relacionado à chave e, dentro dessa estrutura, três campos são usados: no index 0 está o valor que

efetivamente é armazenado e pelo qual procuramos, no index 1 está o timestamp do servidor e o index 2 é usado durante o processo de replicação da mensagem. Note que, sempre que falarmos do value, do timestamp e do replication do servidor, estamos nos referindo aos index 0, 1 e 2 respectivamente da List<Object> relacionada à chave em questão.

Durante a inicialização do servidor *[linha 220]*, é informado se este é o líder ou não. Caso seja o líder, captura seu próprio IP e porta de escuta, que são armazenados também nas variáveis leaderIP e leaderPort, captura também os IPs e portas de escuta dos servidores secundários, armazenados nos vetores scondaryIP e secondaryPort e coloca a variável booleana isLeader como true. Caso não seja o líder, também captura seu próprio IP e porta de escuta, captura o IP e porta de escuta do líder e armazena esses valores em leaderIP e leaderPort.

Cada servidor cria um server socket na sua porta de escuta e roda um loop infinito *[linha 289]* que é responsável por criar um socket e aceitar o pedido de conexão recebido. Essa comunicação é passada para uma thread (ThreadReceiveRequest), que fica responsável por tratar a comunicação, e então volta a escutar novas requisições.

## 5.1. ThreadReceiveRequest *[linha 55]*

Ao iniciar essa thread *[linha 63]*, o servidor cria o mecanismo de leitura e escrita, mas note que apenas o mecanismo de leitura é configurado inicialmente. O servidor recebe a mensagem em Json e converte para Mensagem.

Utilizando um switch, verifica qual é o tipo de requisição *[linha 76]*, e a processa de acordo. Abaixo, alguns detalhes da implementação de cada requisição:

- **PUT** *[linha 79]*

Checa se é o líder. Caso não seja *[linha 83]*, imprime a mensagem no console do servidor que a recebeu e então abre um socket para se comunicar com o líder. Configura o mecanismo de escrita e envia a requisição para o líder, da mesma forma que a recebeu, sem alterações.

Caso seja o líder *[linha 95]*, imprime uma mensagem em seu console, inicializa uma List<Object> com uma String vazia no index 0, e nos index 1 e 2 coloca o valor 0. Checa se o mapa de hash local possui a chave a ser armazenada e, caso a tenha, copia o timestamp armazenado para essa nova lista. Em seguida, coloca o value no index 0 dessa lista e faz uma comparação entre o timestamp existente no servidor somado de 1 e o timestamp recebido pelo cliente, o que for maior é escolhido para ser armazenado no index 1 dessa lista *[linha 105]*, e então armazena essa lista no mapa de hash local relacionando-a à chave. Repare que, caso a chave ainda não exista, o timestamp será 1 ou aquele recebido através do cliente.

O líder então cria uma instância de Mensagem, usando como parâmetros a chave, o value, seu timestamp e no campo request coloca “REPLICATION”. Duas threads (ThreadReplication) são executadas, enviando essa mensagem para cada uma delas. O funcionamento dessa thread será tratado mais adiante.

Um loop é executado até que o replication da chave que está sendo inserida seja 2 *[linha 115]*, esse valor é incrementado cada vez que o líder recebe um “REPLICATION_OK”. Portanto, como temos dois servidores secundários, é necessário receber dois “REPLICATION_OK”.

Ao sair do loop, o líder cria uma nova instância de Mensagem e passa como parâmetro a chave, o value armazenado, seu timestamp relacionado a essa chave e a string “PUT_OK”.

Então cria um socket *[linha 128]* usando os dados do cliente recebidos pela mensagem de requisição, configura o mecanismo de escrita e envia a Mensagem para o cliente.

- **REPLICATION** *[linha 136]*

Os servidores secundários vão receber essa requisição através da ThreadReplication, que ainda não foi abordada. Assim que a recebem *[linha 143]*, imprimem uma mensagem em seus consoles e, de forma similar ao líder, criam uma List<Object>, fazem a checagem se a chave já existem e procedem da mesma

forma, alocando dados nessa lista para depois inseri-las em seus mapas de hash locais.

Após realizar a inserção, criam um socket para se comunicar com o líder *[linha 156]*, configuram o mecanismo de escrita, criam uma instância de Mensagem, que contém a key, o value, seu timestamp e a request "REPLICATION_OK", e enviam para o líder.

- **REPLICATION_OK** *[linha 166]*

Ao receber uma requisição "REPLICATION_OK" *[linha 167]*, o líder incrementa o valor do replication relacionado à chave da Mensagem, ou seja, o líder acessa a List<Object> relacionada à chave contida na Mensagem, acessa o index 2 dessa lista e aumenta seu valor 1.

Essa é a única ação do líder ao receber essa requisição, mas a partir do momento em que recebe duas delas, consegue dar sequência no loop mencionado anteriormente, durante o processamento da requisição "PUT".

- **GET** *[linha 171]*

Assim que recebe uma requisição "GET" *[linha 174]*, cria uma List<Object> de forma similar ao que ocorre no "PUT" e a nomeia de requestedData. Verifica se a key buscada existe no seu mapa de hash local e, caso exista, copia a lista associada a essa key na requestedData.

Note que, se a chave não existir, a requestedData permanece com os valores com que foi inicializada, ou seja, value (index 0) igual a uma string vazia e timestamp (index 1) igual a 0.

O servidor verifica então se o seu timestamp é menor que o do cliente *[linha 186]*, isso pode ocorrer em uma situação, por exemplo, na qual o cliente já fez a requisição "PUT" e aumentou seu timestamp, porém a replicação ainda não aconteceu nesse servidor e seu value e timestamp não foram atualizados. Dessa

forma, o servidor cria uma instância de Mensagem contendo a key, o value (ainda que seja uma string vazia), o timestamp e no campo request armazena "TRY_OTHER_SERVER_OR_LATER".

Caso o timestamp seja igual ou maior, o servidor verifica se o value é uma string vazia *[linha 192]* e, caso seja, cria uma instância de Mensagem que também contém a key, o value como uma string vazia, o timestamp e no campo request armazena "NULL".

No caso em que tanto o timestamp é igual ou maior, quanto o value não é vazio, *[linha 198]* o servidor cria uma instância de Mensagem contendo a key, value, seu timestamp e armazena no campo request o value novamente, apenas para indicar que não houve nenhum erro durante o processamento da requisição.

O servidor então cria um socket *[linha 205]* para iniciar uma conexão na porta de escuta do cliente, usando os dados recebidos pela mensagem de requisição, configura o mecanismo de escrita e envia a resposta ao cliente.

## 5.2. ThreadReplication *[linha 27]*

Durante a requisição "PUT", o servidor líder inicia duas dessas threads *[linha 111]*. Essa thread recebe como parâmetro um int, que representa um dos servidores armazenados no vetores secondaryIP e secondaryPort, e a Mensagem de replicação criada pelo líder *[linha 109]*.

A thread cria um mecanismo de comunicação com o servidor correspondente ao número recebido *[linha 41]*, e encaminha a mensagem para eles. Lembre-se que essa mensagem contém a request "REPLICATION", então cada servidor subordinado vai fazer o processamento dela conforme descrito anteriormente.

# 6. Simulação de atraso *[Servidor:linha 138]*

Para simular o erro “TRY_OTHER_SERVER_OR_LATER”, a chave (key) “teste” foi reservada. Durante a execução de uma requisição “REPLICATION” por parte dos servidores secundários, um teste é feito para verificar se a chave reservada foi usada.

Caso essa chave seja usada, o servidor gera um atraso proposital de 20 segundos antes de dar início à replicação. Dessa forma, podemos simular a situação em que o servidor líder já inseriu a chave e o value, e também atualizou seu timestamp, mas os servidores secundários ainda não o fizeram, de modo que um erro possa ocorrer durante um “GET” por parte de um cliente que possui um timestamp já atualizado e maior do que os timestamps dos servidores secundários.

## Referências

Nenhuma parte deste código foi copiada indiscriminadamente (ou alterada) de outras fontes. Durante o desenvolvimento, foram consultados sites populares, como Stack Overflow, Geeks for Geeks e GitHub, bem como a documentação oficial do Java. O Chat GPT foi utilizado como uma ferramenta adicional para obter *insights* e sugestões ao longo do processo de desenvolvimento.