UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS DE QUIXADÁ

Relatório
Gerenciador de disciplinas da UFC

Disciplina: Sistemas Distribuídos – 2022.2

Equipe: Guilherme Perdeu Freire Sampaio
Victor Emanuel Bernardino Lourenço

Quixadá - CE.

1. **Visão geral do serviço remoto.**

Este serviço tem como funcionalidade o manejo para com as disciplinas da UFC, onde possui funções como, Criar Disciplina, Listar Disciplinas, Deletar Disciplina, e Buscar Disciplina. O serviço foi implementado empregando o ambiente de execução Node.js, já que o mesmo é uma excelente ferramenta para aplicações multi direcionais, que ocorrem em tempo real. O serviço foi desenvolvido em cima do udp, logo será suscetível a falhas, a partir disso, foram implementados métodos para o tratamento de falhas tornando o serviço mais eficiente e menos suscetível a falhas.

Repositorio do projeto
https://github.com/VictorLourencoo/socket_udp

2. **Descrição dos Métodos Remotos.**

Como foi citado anteriormente, o serviço Gerenciador de Disciplinas da UFC contém os seguintes métodos: Criar Disciplina, Listar Disciplinas, Deletar Disciplina, e Buscar Disciplina, que são encontrados no arquivo Servente.js, onde foram implementados. Seriam estes:

Criar Disciplina:

```
function Criar_Disciplina(args){
  try {
     ArrayDisciplinas.push(args)
     const mensagemResposta = "Disciplina Cadastrada com sucesso"
     return mensagemResposta
  } catch (error) {
     return error
  }
```

O método Criar Disciplina recebe o objeto args como parâmetro, adicionando args no vetor ArrayDisciplinas e retorna uma constante mensagemResposta.

Listar Disciplinas:

```
async function listar_Disciplinas(){
  return ArrayDisciplinas;
}
```

O método Listar Disciplinas retorna o vetor ArrayDisciplinas contendo as disciplinas cadastradas.

Deletar Disciplina:

```
async function Deletar_Disciplinas(ID){
for (let index = 0; index < ArrayDisciplinas.length; index++) {
  const element = ArrayDisciplinas[index].id;
 if(ID === element){
  ArrayDisciplinas.splice(index, 1)
 }

}
return ArrayDisciplinas
}
```

O método Deletar Disciplina recebe como parâmetro o ID da disciplina, onde o vetor ArrayDisciplinas é percorrido, e a disciplina com o ID equivalente ao ID passado como parâmetro é excluída do vetor ArrayDisciplinas.

Buscar Disciplina:

```
async function Buscar_Disciplina(Nome_curso){
  let arrayDisciplinasCurso = []
  for (let index = 0; index < ArrayDisciplinas.length; index++) {
     const element = ArrayDisciplinas[index].curso;
    if(Nome_curso === element){
     arrayDisciplinasCurso.push(ArrayDisciplinas[index])
    }
```

```
}
return arrayDisciplinasCurso
}
```

O método Buscar Disciplinas recebe como parâmetro o nome do curso, o vetor arrayDisciplinas é percorrido e a disciplina com nome do curso equivalente é adicionada ao vetor arrayDisciplinasCurso, em seguida o vetor arrayDisciplinasCurso é retornado.

3. **Descrição dos Dados transmitidos.**

Para a viabilizar a interação do cliente e do servidor, um objeto é criado, compactado e enviado ao servidor, esse objeto contém as informações necessárias para a usabilidade dos métodos implementados, sendo estes o ID da disciplina, nome da disciplina, nome do curso e a sua capacidade.

```
const Object_Disciplina ={
        id: ID,
        nome: nome,
        curso: curso,
        capacidade: capacidade
    }
```

4. **Descrição das classes implementadas nos lados Cliente e Servidor.**

Após a criação do objeto no arquivo menu.js, o objeto é passado para a função Criar_disciplina() presente no arquivo proxy.js onde o arquivo json é criado, nessa função o arquivo json, a referência ao objeto e o método são referenciados e recebidos como referência pela função DoOperation(); presente no arquivo Operation.js, a função DoOperation() encapsula o pacote e cria um datagrama, empacota a mensagem em um buffer para finalmente ser enviada em um buffer datagrama para o servidor. O servidor recebe a mensagem e a direciona a função DesempacotaRequisicao(), já que nela o servidor recebe o datagrama em buffer, ou seja, um array de bytes, essa função transforma o array em

json, em seguida o Despachante() recebe o datajson, que seria o objeto ainda em json e assim descompacta para objeto. Com a mensagem descompactada em objeto, torna-se possível identificar o método requerido, com isso, a função é solicitada de acordo com o método, assim o esqueleto contido no arquivo esqueleto.js apenas seleciona a função a ser executada de acordo com o datagrama solicitado pelo cliente, o servente contido no arquivo Servente.js apenas realiza a operação pretendida pelo esqueleto, seja ela criar disciplina, listar disciplina, buscar disciplina e deletar disciplina. Após isso, o Despachante empacota a mensagem em um datagrama de resposta que é enviada pro servidor, onde a função EmpacotaResposta() transforma a mensagem em buffer e devolve para o cliente, que recebe a mensagem, manda a mesma para a função DesempacotaMenssagem() presente no arquivo Desenpacotador.js que desempacota a mensagem e a mensagem é mostrada ao usuario

5. **Descrever o modelo de falhas**

   O Modelo de falhas tratamos em 3 pontos

   1° Servidor desconectado

Aplicamos um modelo de retransmissão quando não consta uma resposta do servidor.

Através do timeout de resposta, quando é disparado, e nao contem uma resposta do servidor, chamamos o método da requisição novamente

```
20      // console.log(`Dados Recebidos do servidor : ${msg.toString()}`);
21      console.log('Received %d bytes from %s:%d\n', msg.length, info.address, info.port);
22    });
23    setTimeout(() => {
24      if (buffer === undefined) {
25        buffer = 500;
26      }
27      resolve(buffer);
28    }, 3000);
29  });
30
31  module.exports = {
32    EnviarMensagem,
33  };
34
```

NO menu,js realizamos a chamada novamente

```
while (statusCode === 500) {
  const ObjectDisciplina = {
    id: ID,
    nome,
    curso,
    capacidade,
  };
  // console.log(Object_Disciplina)
  // eslint-disable-next-line no-await-in-loop
  const resposta = await Proxy.CriarDisciplina(ObjectDisciplina);
  if (resposta.Code === 500) {
    console.log(resposta.error);
  } else {
    statusCode = 200;
    console.log(resposta.arguments);
  }
}
```

2° Verificação de integridade da mensagem

Para levar segurança a aplicação criamos um hash do datagrama a ser enviado,

e encapsulamos em um objeto mensagem junto com a mensagem principal, e enviamos ao servidor.

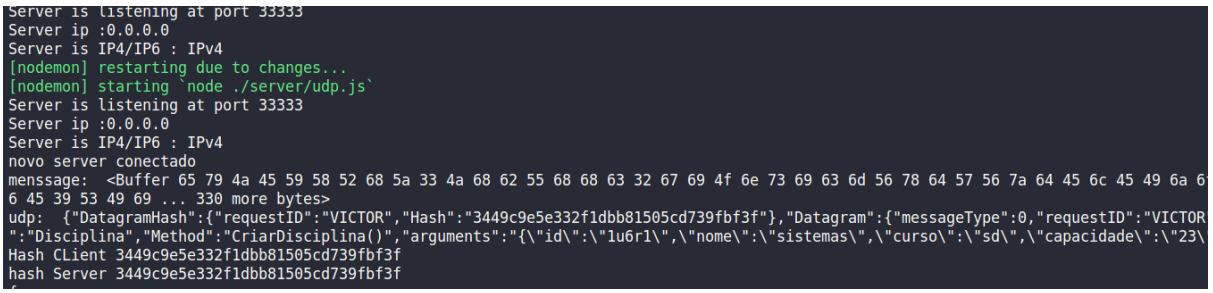
No servidor, pegamos a mensagem original enviada e aplicamos uma função hash, logo após, comparamos o hash enviado pelo cliente com o hash da mensagem feito pelo servidor

Ambos obrigatoriamente precisam ser idênticos, caso contrário enviamos uma exceção ao cliente

```
async function DoOperation(ReferenceObject, Method, JSON) {
  // const ID = Math.floor(Date.now() * Math.random()).toString(36);
  const ID = 'VICTOR';
  const Datagram = {
    messageType: 0,
    requestID: ID,
    ObjectReference: ReferenceObject,
    Method,
    arguments: JSON,
  };

  const HashMessage = md5(Datagram);
  const DatagramHash = {
    requestID: ID,
    Hash: HashMessage,
  };
  const Message = {
    DatagramHash,
    Datagram,
  };
```

```
8   async function Despachante(dataJson) {
9     const DataGramCompleto = JSON.parse(dataJson);
10    const dataVerify = DataGramCompleto.DatagramHash;
11    const HashClient = dataVerify.Hash;
12    const dataObject = DataGramCompleto.Datagram;
13
14    const HashServer = md5(dataObject);
15    console.log('Hash CLient', HashClient);
16    console.log('hash Server', HashServer);
17    HistorySucess(dataObject);
18
19    const RequestVerify = HistoryVerify(dataObject);
20    if (RequestVerify !== 200) {
21      return RequestVerify;
22    }
23
24    if (HashServer !== HashClient) {
25      const dataError = DatagramError(dataObject);
26      HistoryFailure(dataObject);
27      return dataError;
28    }
```



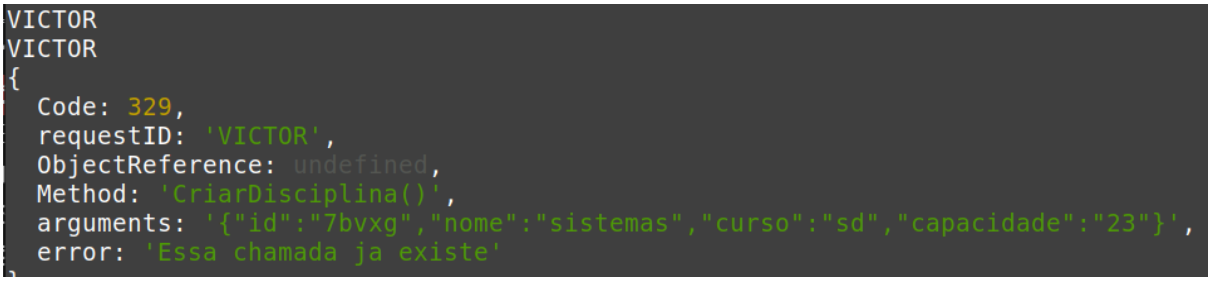
Dados sendo recebidos e verificados no servidor

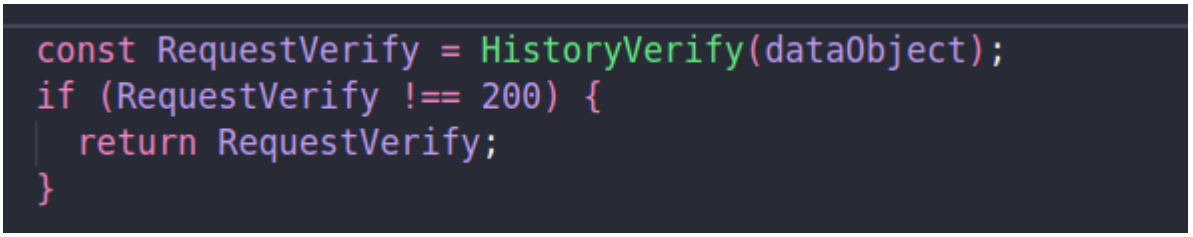
3° Verificação de duplicidade de requisições e respostas

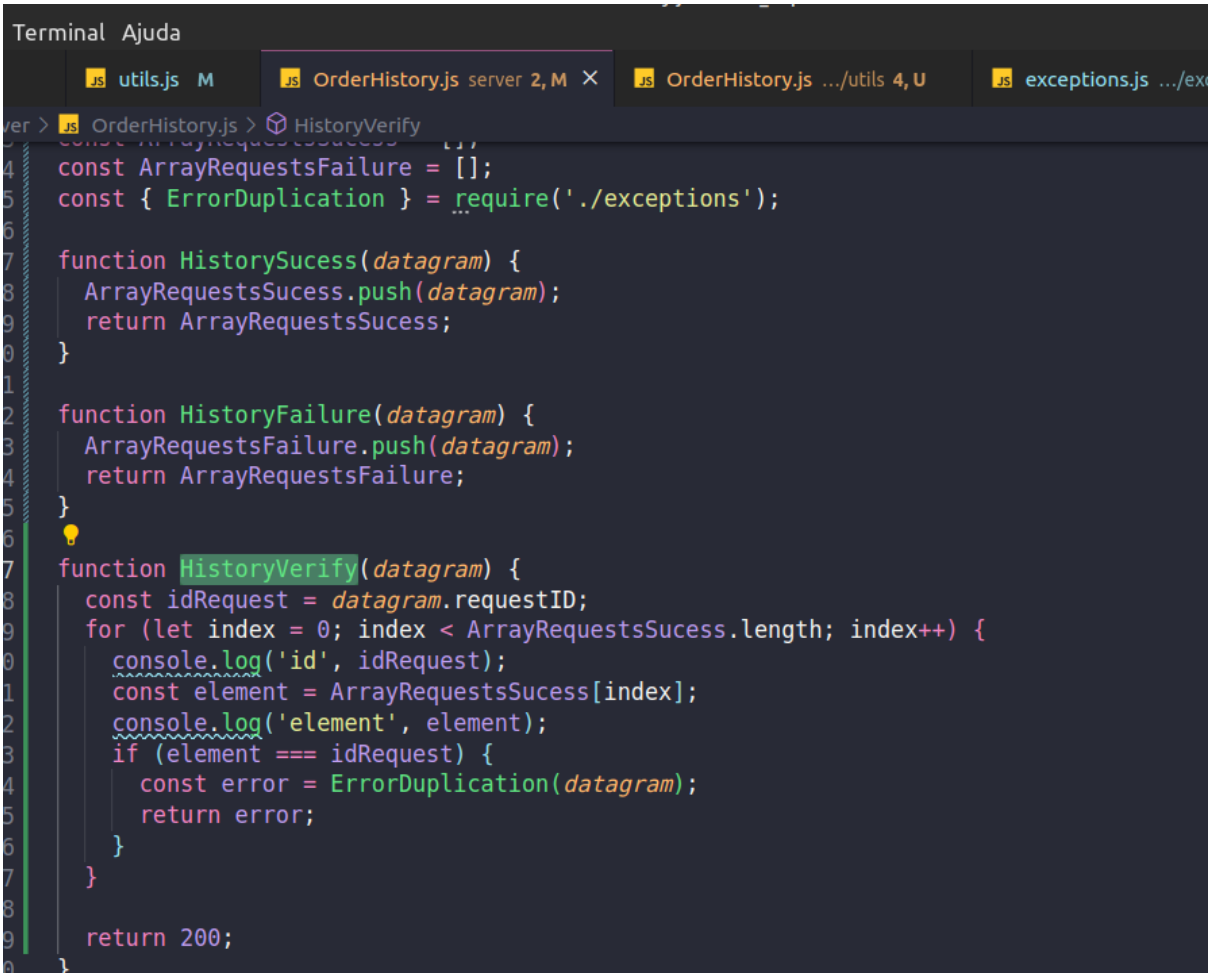
Com o tratamento de falhas anteriores, temos a necessidade de criar um histórico de chamadas, para que não criarmos chamadas iguais, lotando o serviço

Toda chamada que e feito pelo cliente, ela é armazenada nos 2 pontos

cliente e servidor, e é verificada seu id correspondente unico, obrigatoriamente, nao podemos ter 2 chamadas iguais em nenhum dos pontos






Terminal Ajuda
utils.js M
OrderHistory.js server 2, M
OrderHistory.js .../utils 4, U
exceptions.js .../ex
ver > OrderHistory.js > HistoryVerify
const ArrayRequestsFailure = [];
const { ErrorDuplication } = require('./exceptions');
function HistorySucess(datagram) {
ArrayRequestsSucess.push(datagram);
return ArrayRequestsSucess;
}
function HistoryFailure(datagram) {
ArrayRequestsFailure.push(datagram);
return ArrayRequestsFailure;
}
function HistoryVerify(datagram) {
const idRequest = datagram.requestID;
for (let index = 0; index < ArrayRequestsSucess.length; index++) {
console.log('id', idRequest);
const element = ArrayRequestsSucess[index];
console.log('element', element);
if (element === idRequest) {
const error = ErrorDuplication(datagram);
return error;
}
}
return 200;
}