BRENO DA SILVA NOGUEIRA - 12400392

CAIO VINÍCIUS MELO DA SILVA - 12542859

GABRIEL DOS SANTOS NASCIMENTO - 12732792

JULIANA ALMEIDA SANTOS - 12566178

Exercício-Programa: Remote Procedure Call (RPC)

Docente: Norton Trevisan Roman

Disciplina: Desenvolvimento de Sistemas de Informação Distribuídos (ACH2147)

São Paulo/SP

2023

Sumário

1) Introdução

Um sistema de informação é um conjunto organizado de pessoas, processos, dados e tecnologias que trabalham juntos para permitir a coleta, o armazenamento, o processamento e a distribuição de informações a fim de facilitar o funcionamento de uma organização. A disseminação desse tipo de programa aconteceu depois da explosão da internet, em 1993, e, desde então, esses sistemas não param de crescer.

A princípio, eles eram desenvolvidos de maneira centralizada, onde todo o processamento ocorria em um único local, porém, com o progresso da tecnologia, esse modelo foi substituído pelo desenvolvimento distribuído. Essa abordagem mais recente oferece o compartilhamento de recursos com outros tipos de interfaces, como impressoras, páginas web, banco de dados distribuídos, entre outros.

Dessa forma, é notória a importância dos sistemas de informação distribuídos na atualidade, uma vez que eles representam uma ótima estratégia para os ambientes possuírem escalabilidade, alto desempenho, tolerância a falhas e heterogeneidade. Compreendida a relevância, a disciplina de Desenvolvimento de Sistemas de Informação Distribuídos é obrigatória para os alunos do curso de Sistemas de Informação na Universidade de São Paulo, com o intuito de elucidar as técnicas necessárias para a construção de tal esquematização.

No vigente relatório, será descrita uma atividade da disciplina onde foi proposta a construção de um sistema de informações sobre peças e componentes usando Remote Method Invocation com a linguagem Java ou Remote Procedure Call com as demais linguagens. O grupo composto por 4 alunos optou por trabalhar com a linguagem de programação Python, assim, utilizando o método Remote Procedure Call.

O desenvolvimento do sistema foi proposto na arquitetura cliente-servidor, com múltiplos servidores, com cada um deles implementando um repositório de informações sobre as peças. As peças devem ser representadas por um objeto cuja interface é *Part* e cada servidor implementado tem um objeto *PartRepository*, que é uma coleção de *Parts*.

Além disso, uma peça pode ser primitiva ou não, nessa opção sendo composta por uma agregação de subcomponentes. Foi indicado que a lista de subcomponentes de uma peça seria formada por pares que indicam uma referência

ao subcomponentes da peça (*subPart*) e uma indicação de quantas unidades do subcomponente aparecem na peça (*quant*), sendo assim, vazia a lista de componentes de uma peça primitiva.

A partir dessas instruções gerais e de algumas descrições específicas a respeito dos objetos *Part* e *PartRepository,* foi construído o sistema proposto. Esse desenvolvimento será exposto mais detalhadamente ao longo deste trabalho.

2) Desenvolvimento

Para que a descrição do desenvolvimento do código seja feita da maneira mais clara possível, ela será dividida em partes. Cada parte representará uma versão do projeto, sendo assim, será dividida entre 5 commits.

I. Primeiro commit

O primeiro passo que foi realizado na construção da unidade cliente-servidor foi a montagem de um protótipo de cada um dos elementos. Esse protótipo, em sua primeira versão, apenas estabelecia a conexão entre as duas partes. Os códigos estão expostos a seguir:

```python
import sys
from xmlrpc.server import SimpleXMLRPCServer

# SERVER CONFIGS
IP = '127.0.0.1'

def main(args):
    servidor = SimpleXMLRPCServer((IP, PORTA))
    print('\tParts Server')
    print("Esperando por conexões...")

    # Lista de Partes (banco de dados)
    parts = []

    # Cria uma base de dados mock
    bootstrap_parts()

    def agenda():
        print("Clog the toilet")

    servidor.register_function(agenda, "agenda")
    servidor.serve_forever()


if __name__ == "__main__":
    # TODO: Verify if all args are good to go
    main(sys.argv[1:])
```

Imagem 1: Código da construção do protótipo do servidor.

```python
import xmlrpc.client
from actionconsole import ActionConsole

print('\tCLIENTE')

IP = input('- Digite o IP do Servidor: ')
PORTA = int(input('- Digite a PORTA: '))

servidor = xmlrpc.client.ServerProxy("http://{0}:{1}/".format(IP, PORTA))

console = ActionConsole()
while effect != 'quit':
        command = input(">")
        effect = console.run_command(command)

```

Imagem 2: Código da construção do protótipo do cliente.

Além disso, no primeiro *commit* do código também foram construídas as versões iniciais dos objetos **PartRepository.py** *e* **Part.py** e de um arquivo auxiliar chamado ActionConsole. O primeiro objeto citado foi construído com métodos que fazem as definições de nome, número de peças contidas nele, além de uma listagem de suas peças e uma listagem e busca de repositórios no servidor:

```python
from part import Part

class PartRepository:
    def __init__(self, repo_name) -> None:
        self.name = repo_name
        self.parts = { }
        self.length = 0

    def repo_info(self):
        '''Examinando o nome do repositorio e o numero de pecas nele contidas'''
        return (self.name, self.length)

    def insert_part(self, part:Part):
        '''Listando as pecas no repositorio'''
        self.parts[part.get_id()] = part

    def list_parts(self):
        return self.parts.values()

    def search_part(self, key):
        if key in self.parts.keys:
            return self.parts[key]
        else:
            return None

    def repo_part(self):
        pass

    def part_type(self):
        pass

    def subparts_info(self):
        pass

    def list_subparts(self):
        pass
```

Imagem 4: Definição da versão original do objeto PartRepository.

Já o segundo objeto, foi definido junto ao seu nome, descrição e subpartes:

```python

class Part:
    def __init__(self, id:str, name:str, desc:str, subpart:Part=[]) -> None:
        self.id = id
        self.name = name
        self.description = desc
        # Uma subpart é um array de tuplas (subpart, quantity)
        self.subpart = subpart

```

Imagem 5: Definição da versão original do objeto Part.

Por fim, o arquivo auxiliar foi construído como uma ponte entre o cliente e o servidor. Nele estão contidas todas as chamadas que o cliente pode fazer ao servidor, ou seja, todas as funções contidas no servidor:

```python

class ActionConsole:
    def __init__(self) -> None:
        pass

    def run_command(self, command:str) -> str:
        return self.__switch_function(command)

    def __switch_function(self, cmd:str) -> str:
        verb = cmd.split()[0]
        match verb:
            case "bind":
                return

            case "listp":
                return

            case "getp":
                return

            case "showp":
                return

            case "clearlist":
                return

            case "addsubpart":
                return

            case "addp":
                return

            case "help":
                return

            case "quit":
                return

            case _:
                print("Comando não reconhecido!")
                return "error"
```

Imagem 6: *Definição do arquivo auxiliar ActionConsole.*

II. Segundo commit

A primeira atualização do projeto contou com alterações apenas no arquivo **Server.py**. Nesse versionamento, foram adicionadas funções com as seguintes utilidades ao servidor: listagem de subpartes, criação de novas partes, armazenamento de informações de subpartes e armazenamento do tipo das partes. Vale ressaltar que as últimas funções mencionadas não estarão presentes na versão final do servidor; em alguma atualização elas serão descontinuadas, sendo assim, apenas funções temporárias.

Nas imagens abaixo, é possível observar as atualizações feitas no código.

```python
def list_subparts(self):
    if not id in self.parts.keys():
        return None

    return self.parts[id].subparts
```

Imagem 7: *Função para listar as subpartes.*

```python
def create_part(self, data: dict):
        if not ["id", "name", "desc", "subparts"] in data.keys():
            return None

        new_part = Part(data["id"], data['name'], data['desc'], data['subparts'])

        self.parts[new_part.id] = new_part
        return new_part
```

Imagem 8: *Função para criação de novas partes.*

```python
def subparts_info(self, id):
    if not id in self.parts.keys():
        return None
    part = self.parts[id]
    for key, value in self.parts.items():
        # TODO
        pass
```

Imagem 9: *Função para armazenamento de informações de subpartes..*

III. Terceiro commit

Em contrapartida ao anterior, o terceiro commit foi focado em atualizações no arquivo referente às funcionalidades do cliente, o **ActionConsole.py**. Também foram adicionadas novas funções ao código. Cada uma delas ofereceu uma das seguintes funcionalidades ao cliente: validação de comando recebido; conexão com o servidor; listagem de nome do repositório atual e seu número de partes; listagem de peças; retorno de uma peça em específico; listagem das características de uma peça; listagem de subpeças atuais; remoção de todas as subpeças atuais; adição de uma subpeça à lista atual; criação de uma nova peça no servidor adicionando subpeças da lista de subpeças.

Nas imagens a seguir, está exposto o código após as alterações:

```
def run_command(self, command:str) -> str:
    cmd_free = ["bind", "help", "quit"]
    if command.split()[0] not in cmd_free and self.connection == None:
        print("Não há uma conexão com um repositório para executar esse comando!")
        return "failed"

    return self.__switch_function(command)
```

Imagem 10: *Função para validação de comando recebido.*

```
match verb:
    case "bind":
        if ":" in args[0]:
            con_string = f"http://{args[0]}/"
        elif len(args) > 1:
            con_string = f"http://{args[0]}:{args[1]}/"

        try:
            self.connection = xc.ServerProxy(con_string, allow_none=True)
        except:
            print(f"Erro ao conectar com o servidor no endereço {con_string}")
            return "failed"

        print(f"Conectado com sucesso no host {con_string}")
        return "fulfilled"
```

Imagem 11: *Função para conexão com o servidor.*

```
case "repo":
    request = self.connection.repo_info()
    print(f"Repositório Atual: {request[0]}")
    print(f"Número de Partes: {request[1]}")
    return "fulfilled"
```

Imagem 12: *Função para listagem do nome do repositório atual e seu número de partes.*

```
case "listp":
    parts = self.connection.list_parts()

    for item in parts:
        print(item['id'], " => ", item['name'])
    return "fulfilled"
```

Imagem 13: *Função para a listagem de peças.*

```
case "getp":
    if args == []:
        print("Nenhum Id passado para busca!")
        return "failed"

    id = args[0].strip()
    if not id.isdigit():
        print("Id inválido!")
        return "failed"

    try:
        self.part = self.connection.search_part(id)

        if not self.part:
            print(f"Peça não encontrada no repositório atual ({self.connection.repo_info()[0]})")
            return "failed"
    except Exception as e:
        print(f"Houve um erro ao processar sua requisição pelo servidor: {e}")
        return "failed"

    self.show_part()
    return "fulfilled"
```

Imagem 14: *Função para o retorno de uma peça em específico.*

```
case "showp":
    if not self.part:
        print("Não há uma parte contexto atual, dê 'getp <id>' para procurar uma parte e retorná-la")
        return "failed"

    self.show_part()
    return "fulfilled"
```

Imagem 15: *Função para retornar as características de uma peça.*

```
case "showsp":
    if not self.subpart:
        print("Não há uma subparte contexto atual, dê 'getsp <id>' para procurar uma parte e retorná-la")
        return "failed"

    self.show_part(sub=True)
    return "fulfilled"
```

Imagem 16: *Função para a listagem de subpeças atuais.*

```
case "clearlist":
    if not self.subpart:
        print("Não há uma subparte no contexto atual, dê 'getp <id>' para procurar uma parte e retorná-la")
        return "failed"

    if self.subpart['subparts'] == []:
        print("Não há subpartes para serem apagadas.")
        return "failed"

    try:
        response = self.connection.clear_subparts(self.part['id'])
    except Exception as e:
        print(f"Houve um erro ao processar sua requisição pelo servidor: {e}")
        return "failed"

    if not response:
        print("Houve um conflito de Id's no servidor, tenha certeza que a peça é desse repositório")
        return "failed"

    self.subpart = response
    print("Subpartes apagadas com sucesso!")
    return "fulfilled"
```

Imagem 17: *Função para remoção de todas as subpeças atuais.*

```
case "addsubpart":
    if len(args) < 1:
        print("Número de peças não passado!")
        return "failed"

    if not self.subpart or not self.part:
        print("Não há uma parte ou subparte no contexto atual, procure por ambas e tente novamente!")
        return "failed"

    try:
        n_parts = int(args[0])
    except:
        print("Número de peças em formato inválido!")
        return "failed"

    response = self.connection.add_subpart(self.part['id'], self.subpart['id'], n_parts)
    if not response:
        print("Houve um conflito de Id's no servidor, tenha certeza que a peça é desse repositório")
        return "failed"

    self.part = response
    self.show_part()
    return "fulfilled"
```

Imagem 18: *Função para adição de uma subpeça à lista atual.*

```python
case "addp":
    name = input('Nome da peça: \n')

    desc = input('\nDescrição da peça: \n')

    new_part = self.connection.create_part(name, desc)
    self.part = new_part
    print(f"Peça criada! ID de número: {new_part['id']}")
    return "fulfilled"
```

Imagem 19: *Função para criação de uma nova peça no servidor adicionando subpeças da lista de subpeças.*

IV. Quarto commit

O primeiro passo na versão quatro foi a construção de um registro de nomes de servidor. Para isso, foi criado o arquivo Servers.json que seria utilizado tanto pelo servidor quanto pelo cliente. O objetivo desse arquivo era emular um servidor de nomes, aproximando-se de um DNS primitivo. Dessa forma, esse arquivo estava responsável por atribuir um nome a um endereço IP de maneira simples, uma vez que a elaboração de um servidor de nomes mais elaborado não era necessária.

A seguir, encontra-se o código elaborado para tal função:

```json
[
    {
        "name": "orchid",
        "link":"http://127.0.0.1:335"
    },
    {
        "name": "daisy",
        "link":"http://127.0.0.1:645"
    },
    {
        "name": "rose",
        "link":"http://127.0.0.1:915"
    },
    {
        "name": "tullip",
        "link":"http://127.0.0.1:785"
    }
]
```

Imagem 20: *Código contido no arquivo Servers.json.*

Em seguida, foram realizadas modificações no cliente e no servidor. Primeiro será exposto as atualizações feitas no cliente e depois no servidor. Foram adicionados comandos ao arquivo **ActionConsole.py** para que o cliente pudesse listar os servidores registrados, procurar o ID de uma peça em todos os servidores registrados, exibir o tipo de peça e adicionar subpeças a peças já existentes.

Essas modificações estão expostas a seguir:

```
case "servers":
    for s in self.servers:
        print(s['name'])

    return "fulfilled"
```

Imagem 21: *Comando para listar os servidores registrados.*

```
case "seekp":
    if args == []:
        fprint("Nenhum Id passado para busca!")
        return "failed"

    id = args[0].strip()
    if not id.isdigit():
        fprint("Id inválido!")
        return "failed"

    for server in self.servers:
        try:
            aux_connection = xc.ServerProxy(server['link'], allow_none=True)
            part = aux_connection.search_part(id)

            if part:
                self.show_part(part)
                return "fulfilled"

        except:
            continue

    fprint("A peça não foi encontrada em nenhum servidor registrado")
    return "failed"
```

Imagem 22: *Comando para procurar o IP da peça em todos os servidores registrados.*

```
case "typep":
    if not self.part:
        fprint("Não há uma parte contexto atual, dê 'getp <id>' para procurar uma parte e retorná-la")
        return "failed"

    if self.part['subparts'] == []:
        print("A peça é primitiva")
    else:
        print("A peça é agregada")
    return "fulfilled"
```

Imagem 23: *Comando para exibir o tipo da peça.*

```
case "appendsub":
    if not self.part or self.subpart == []:
        fprint("Não há uma parte no contexto atual ou não há subpartes para referenciar")
        return "failed"

    response = self.connection.add_subpart(self.part['id'], self.subpart)
    if not response:
        fprint("Houve um conflito de Id's no servidor, tenha certeza que a peça é desse repositório")
        return "failed"

    self.subpart = []
    self.part = response
    self.show_part()
```

Imagem 24: *Comando para adicionar subpeças a peças já existentes.*

Ademais, foram criadas, no arquivo **Colors.py**, funções para colorir os textos de respostas a interações do cliente para que houvesse uma melhor distinção entre exibições de erro, informações ou sucesso (respectivamente, nas cores vermelho, amarelo e verde).

```
1   OKBLUE = '\033[94m'
2   OKGREEN = '\033[92m'
3   WARNING = '\033[93m'
4   FAIL = '\033[91m'
5   ENDC = '\033[0m'
6
7   def fprint(message):
8       print(FAIL + message + ENDC)
9
10  def okprint(message):
11      print(OKGREEN + message + ENDC)
12
13  def infoprint(message):
14      print(WARNING + message + ENDC)
15
16  def helpprint(cmd, help):
17      print(OKBLUE + cmd + ENDC, WARNING + f" => {help}" + ENDC)
18
```

Imagem 25: *Função para adicionar cores às exibições decorrentes de interações do cliente.*

Por fim, as alterações no servidor buscavam adicionar um campo de nome de repositório à classe **Part.py** e a refatoração do servidos a fim de acomodar o sistema de registro de servidores:

```python
class Part:
    def __init__(self, id:str, name:str, desc:str, subparts:list, repo: str) -> None:
        # ID da peça
        self.id = id
        # Nome da peça
        self.name = name
        # Descrição da peça
        self.description = desc
        # Uma subpart é um array de tuplas (subpart, quantity)
        self.subparts = subparts
        # Repo que ele pertence
        self.repo = repo
```

Imagem 26: *Adição um campo nome de repositório à classe Part.py.*

V. Quinto commit

Como última atualização do projeto, foram realizadas mais algumas modificações no cliente no arquivo **ActionConsole.py**. Foram adicionados dois comandos: um para a listagem de todas as subpeças de uma parte e outro para a criação de um suporte ao usuário. Além disso, o comando de listagem do tipo de peça foi descontinuado porque tal funcionalidade foi anexada a outra ação:

```python
# Mostra a lista de subpeças no contexto atual
case "showsub":
    if self.subpart == []:
        fprint("Não há subpartes no contexto atual, dê 'getsp <id>' para procurar uma parte e retorná-la")
        return "failed"

    self.show_subparts()
    return "fulfilled"
```

Imagem 27: *Comando para listar todas as subpeças de uma parte.*

```
# Help para os fracos que não sabem usar
case "help":
    helpprint("servers", "Lista o nome de todos os servidores registrados")
    helpprint("bind", "Se conecta a um servidor usando o seu nome")
    helpprint("unbind", "Desconecta do servidor atual")
    helpprint("repo", "Lista as informações do repositório atual")
    helpprint("listp", "Lista todas as partes do repositório")
    helpprint("getp", "Coloca a peça com o ID passado no contexto atual")
    helpprint("showp", "Mostra as informações da peça atual")
    helpprint("seekp", "Retorna a peça com ID passado, procurando por todos os servidores registrados")
    helpprint("subp", "Lista todas as subpartes da peça atual")
    helpprint("addp", "Cria uma nova peça, e adiciona as subpeças atuais (se houver alguma)")
    helpprint("showsub", "Mostra a lista de subpeças no contexto atual")
    helpprint("addsub", "Adiciona a peça atual na lista de subpeças")
    helpprint("appendsub", "Adiciona todas as subpeças da lista atual na peça atual")
    helpprint("clearlist", "Apaga a lista de subpeças atual")
    helpprint("quit", "Sai da aplicação")
    return "fulfilled"
```

Imagem 28: *Comando para criação de um suporte ao usuário.*

E, finalmente, foi concluída a construção dos códigos com o fim da documentação através de comentários. Todo o projeto está disponível em um no arquivo conjunto a este relatório.

3) Resultados

Nesta seção, serão expostos alguns casos de uso do sistema desenvolvido. Mas, antes disso, é preciso entender quais configurações e comandos são executados para que se possa fazer uso do sistema: É necessário ter instalado o interpretador Python (o projeto foi codificado usando a versão 3.10.10, porém é preciso pelo menos a versão 3.10.4 para rodá-lo sem erros).

Uma vez instalado, para criar um servidor, é necessário primeiro registrá-lo no arquivo "servers.json" na raiz do projeto, usando o seguinte template:

```
{
    "name": "<nome servidor>",
    "link":"http://127.0.0.1:<porta servidor>",
    "data": [
        {
            "name": "<nome peça>",
            "desc": "<descrição peça>",
            "subparts": []
        }
    ]
}
```

O campo "data" é usado para inicializar o servidor com algumas partes já declaradas, logo ele pode ser vazio, sendo igual a "[]" como ocorreu no tópico subparts do exemplo. Ademais, é preciso ter cuidado com as vírgulas que separam cada registro de servidor, pois isso pode gerar erros na hora de ler o arquivo.

Após registrar o servidor, deve-se abrir o terminal no diretório raiz do projeto e rodar um dos seguinte comandos, de acordo com o sistema operacional utilizado, para iniciar o servidor:

```
//Para Windows
python servidor/server.py <nome servidor>
```

```
//Para Linux
sudo python3 servidor/server.py <nome servidor>
```

Se o comando for executado com êxito, deve ser exibido algo semelhante à imagem a seguir:

```
Admin@DESKTOP-ODLN0SM MSYS ~/codes/current-semester/DSIS (main)
$ python servidor/server.py tullip
     Parts Server ~ Repo: tulip
Esperando por conexões…
```

E para rodar o cliente, é necessário executar um dos seguintes comandos, novamente, de acordo com o sistema operacional em uso:

```
//Para Windows
python cliente/client.py
```

```
//Para Linux
python3 cliente/client.py
```

Após comprimir estes passos, o sistema já está pronto para ser executado, com as interfaces de cliente e servidor já conectadas.

O primeiro caso de uso que será visto é um caso de consultas de partes, seguido pela adesão de uma subpeça a uma peça já existente. Na imagem a seguir, é possível observar a conexão com o servidor (*bind tullip*), a listagem das peças existentes nesse servidor (*listp*), o retorno das informações sobre uma peça em específico (*getp*), a inclusão de uma subparte a lista (*addsub*) e a atribuição desta subparte a uma peça já existente (*appendsub*):

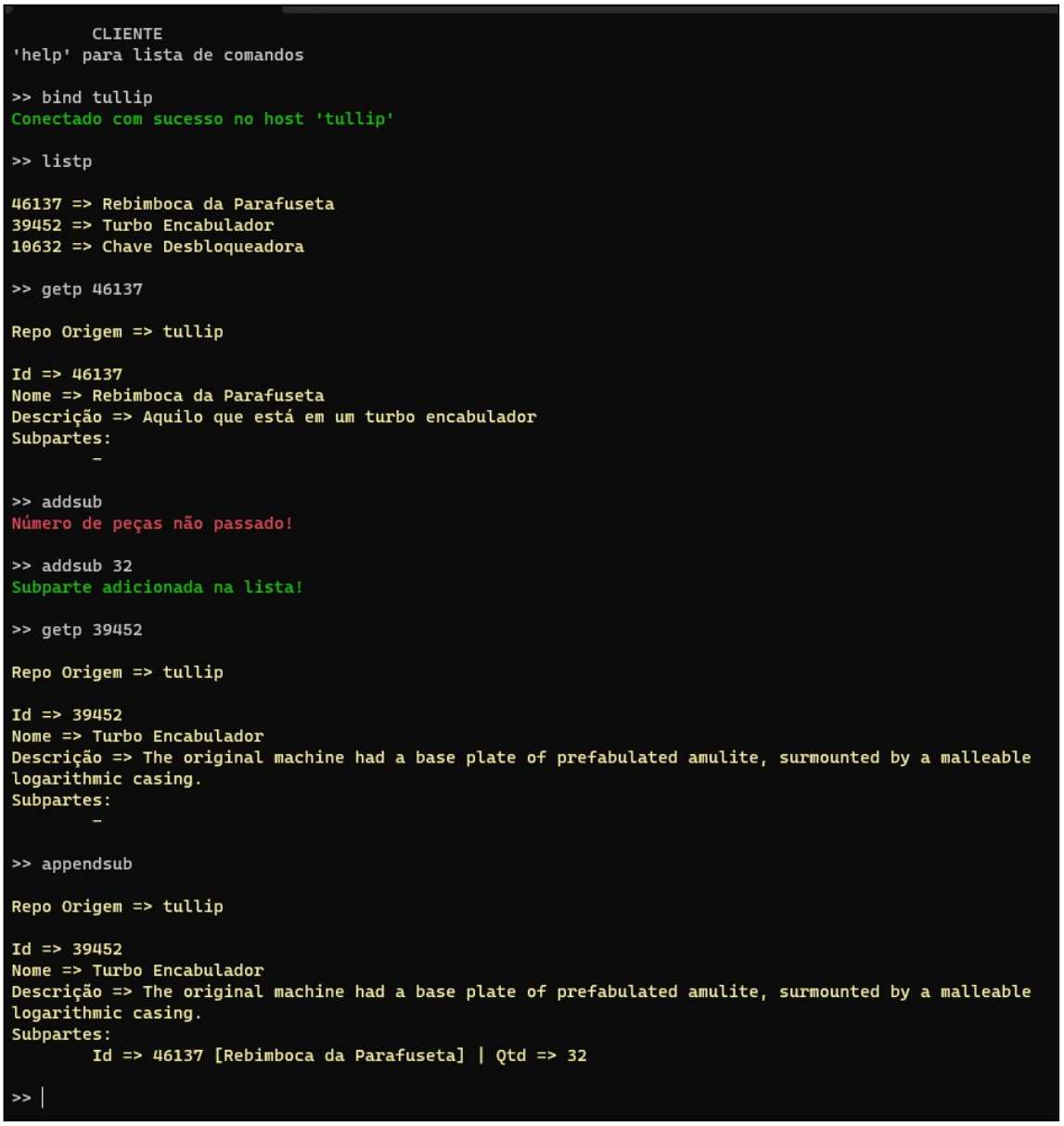

Imagem 29: *Primeiro caso de uso.*

O segundo caso tratou-se da criação de uma parte e da inserção de uma subpeça presente em um repositório diferente do atual. Nas imagens a seguir, podemos observar os comandos utilizados nessa consulta. Na primeira imagem, há a listagem de servidores registrados (*servers*), a conexão ao servidor Tullip (*bind tullip*), a listagem das peças cadastradas nele (*listp*), a exibição de informações sobre uma peça específica (*getp*), a inserção de uma subparte (*addsub*), a conexão com ao servidor Rose (*bind rose*), a adesão de uma nova peça (*addp*) e a consulta buscando informações sobre uma peça específica (*getp*), a qual retorna uma mensagem de erro por ter sido passado o ID errado:

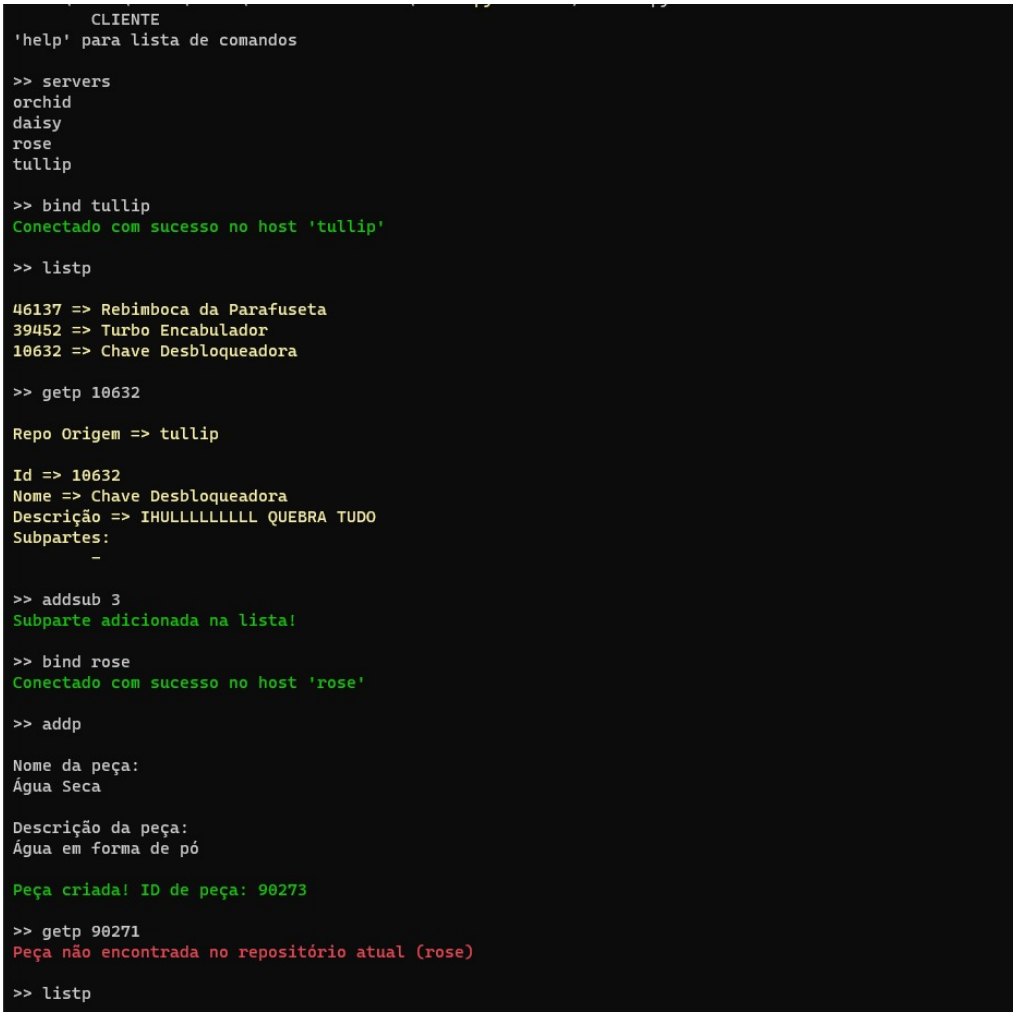

Imagem 30: *Primeira imagem do segundo caso de uso.*

Na próxima imagem, é possível observar a listagem de peças para verificação do ID da peça consultada anteriormente (*listp*) e a solicitação por informações de uma peça específica (*getp*), agora utilizando o ID correto. Nota-se que foi adicionada uma subpeça a ela, mesmo esse comando tendo sido realizado enquanto a conexão estava mantida com o servidor Tullip:

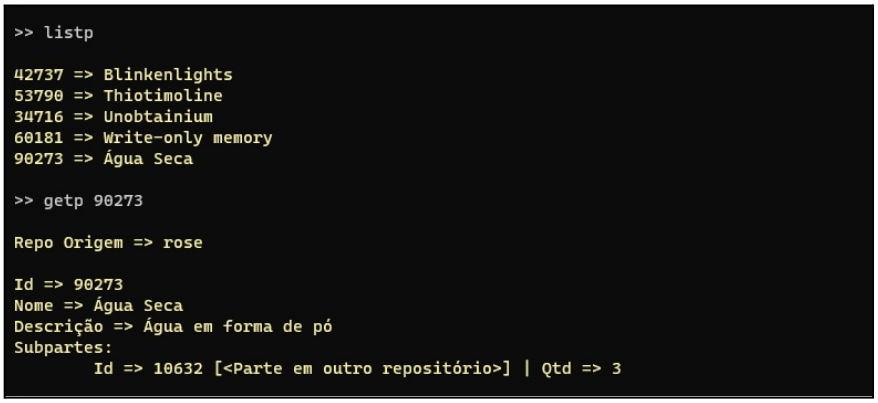

Imagem 31: *Segunda imagem do segundo caso de uso.*

O terceiro caso é uma continuação do segundo. Ele busca por uma peça com uma subpeça de outro repositório, ou seja, de um servidor diferente do que ao que se está conectado. O comando (*seekp*) é executado enquanto o cliente está conectado ao servidor Rose (*bind rose*), mas a busca é por uma peça contida no servidor Tullip:

Imagem 32: *Terceiro caso de uso.*

O quarto e último caso de uso faz consultas sobre peças e subpeças e, também, utiliza o comando help para ter mais informações sobre as ações que o cliente pode tomar. Na seguinte imagem, é conectado ao servidor Tullip (*bind Tullip*), são solicitadas informações sobre o repositório atual (*repo*), as peças presentes nele são listadas (*listp*), é realizada uma busca por informações de uma peça específica (*getp*) e também por informações de subpeças adicionadas (*subp*) e, por fim, é passado o comando *help* para visualização de todos os comandos disponíveis:

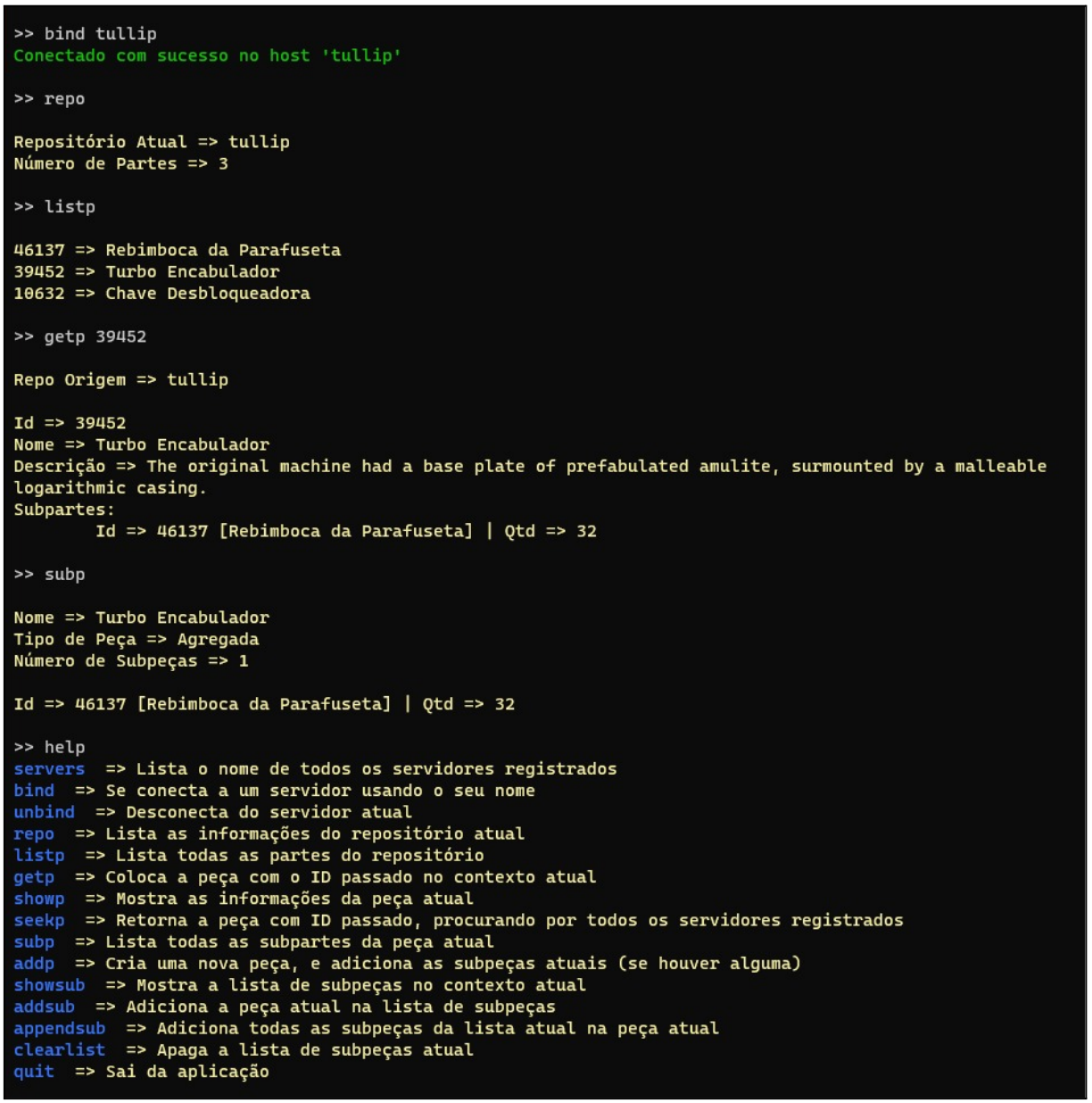

Imagem 33: *Quarto caso de uso.*

4) Conclusões

Ao fim do projeto, foi possível concluir que o desenvolvimento de um sistema de informações distribuído é uma tarefa custosa por conta da sincronização entre todas as distribuições do sistema.

Para construir todas as ações necessárias para que o modelo cliente-servidor funcionasse da maneira desejada, era necessário verificar se todas as funções e comandos estavam de acordo com a nova implementação, o que levava um determinado tempo e requeria certa atenção. Entretanto, essa construção, por mais que custosa para o desenvolvedor, é de extremamente valorosa para o cliente final, pois esse modelo oferece escalabilidade, confiabilidade, maior tolerância a falhas, compartilhamento de recursos, flexibilidade e, até mesmo, redução de custos.

Dessa forma, percebeu-se a importância do estudo dessa forma de desenvolver sistemas, e também se notou que com um planejamento adequado, um projeto bem desenvolvido e uma implementação atenta, é possível superar os empecilhos e construir um sistema de informação distribuído eficiente e confiável.

5) Referências

Sistemas de Informação Distribuídos. Brasil Escola. Disponível em: < https://www.infoescola.com/informatica/sistema-de-informacao-distribuido/>. Acesso em: 02 de jun. 2023.