Nome: João Victor Magalhães Souza. **Matrícula:** 3483
Lucas Ranieri Oliveira Martins **Matrícula:** 3479
Disciplina: CCF 355 - Sistemas Distribuídos e Paralelos.
Professora: Dra. Thais Regina de Moura.
Ideia: Álbum de figurinhas de Linguagens de Programação.

Programming Language Collection
Implementação via API de *Sockets*

Introdução

Primeiramente, nós modelamos nosso problema mantendo uma visão geral mas já pensando em quais tipos de aplicações teríamos na construção deste projeto. Dessa forma, criamos um esquemático abaixo que descreve bem nossa ideia de implementação e o funcionamento do nosso projeto como um todo. A linguagem de programação escolhida para implementar esse projeto foi Python.

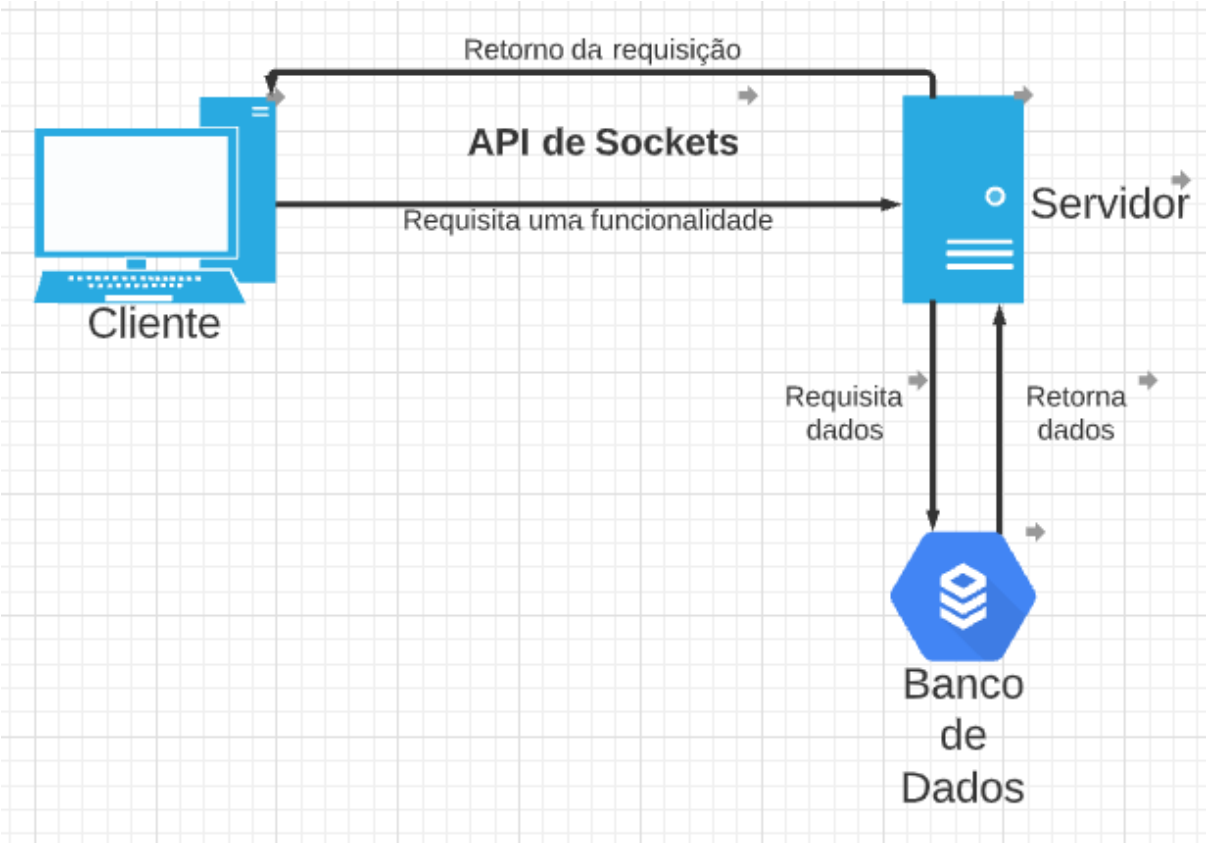

Figura 1 - Representação em alto nível da nossa implementação.

Modelagem do Banco de Dados

A nossa proposta aqui é ilustrar como organizamos o comportamento e o fluxo de dados entre as entidades que se relacionam no nosso projeto. Para tal, utilizamos como referência a nossa especificação na Parte 1 deste projeto. Foi utilizado o SGBD **MySQL** para criação dos diagramas e das *querys* para geração e manipulação do Banco de Dados.

O diagrama que representa o fluxo de comunicação dos dados em nosso sistema pode ser visto a seguir:

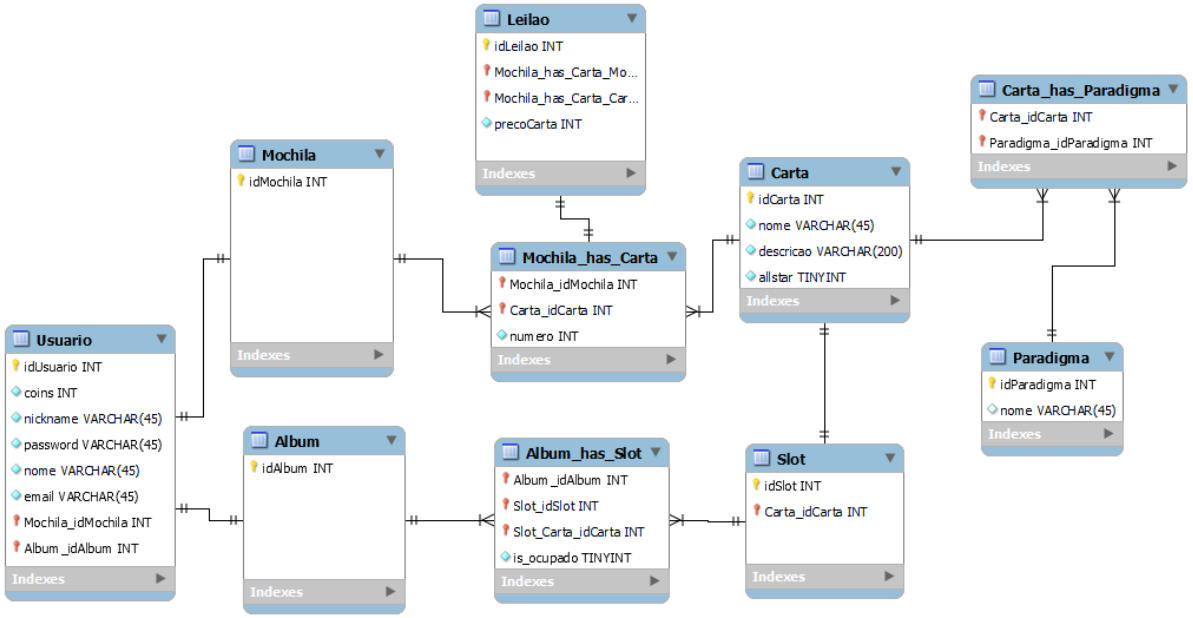

Figura 2 - Diagrama ER do nosso Banco de Dados.

Basicamente, temos o seguinte esquemático:

- ➔ Um **Usuário** possui um **Álbum ;**
  - ◆ Um **Álbum** possui vários **Slots ;**
    - Cada **Slot** possui uma **Carta ;**
      - Cada **Carta** possui um **Paradigma ;**
- ➔ Um **Usuário** possui uma **Mochila ;**
  - ◆ Uma **Mochila** possui várias **Cartas ;**
    - Um **Leilão** possui várias **Cartas de Diferentes Mochilas ;**

Arquitetura do Cliente

Visando representar o cliente, criamos uma classe denominada “Cliente”. Essa classe faz requisições no “Servidor”, que será mais bem detalhado posteriormente, no endereço ***localhost (127.0.0.1)*** na porta **9000.** É importante salientarmos que a porta utilizada pelo Cliente para comunicar com o servidor pode variar, fazendo todo sentido. No caso abaixo, o processo Cliente foi executado duas vezes.

```
Servidor iniciado em localhost:9000
Connected to MySQL database... MySQL Server version on  8.0.26
Atendendo o cliente ('127.0.0.1', 52347)
Mensagem: ['login', 'Admin', 'admin']
('127.0.0.1', 52347) -> requisição atendida !
Atendendo o cliente ('127.0.0.1', 52347)
Erro na conexão ('127.0.0.1', 52347):(10054, 'Foi forçado o cancelamento
 de uma conexão existente pelo host remoto', None, 10054, None)
Atendendo o cliente ('127.0.0.1', 52356)
Mensagem: ['login', 'Admin', 'admin']
('127.0.0.1', 52356) -> requisição atendida !
Atendendo o cliente ('127.0.0.1', 52356)
Erro na conexão ('127.0.0.1', 52356):(10054, 'Foi forçado o cancelamento
 de uma conexão existente pelo host remoto', None, 10054, None)
```

Figura 3 - Variação de diferentes portas para um mesmo Cliente.

Postada a estrutura do Cliente, pensamos em simular uma interface gráfica via terminal, de modo que o usuário possa ir escolhendo o que ele quer fazer e o objeto instanciado do Cliente envie informações ao Server. Dito isso, fornecemos algumas funcionalidades como “Login” e “Cadastro”. Ao efetuar o “Login” o usuário acessa uma ampla gama de funcionalidades para simular o *Card Game*, como mostrado nas imagens abaixo.

```
######################################################################
#    Olá ! Seja bem vindo ao Programming Language Collection    #
#    1) Fazer cadastro.                                              #
#    2) Fazer login.                                                 #
#    3) Sair do sistema.                                             #
######################################################################


Digite a operação: 2
######################################################################
#                      INFORMAÇÕES DE LOGIN                        #
######################################################################
Digite seu nickname: Admin
Digite sua senha: admin


=====> Login realizado com sucesso !
```

Figura 4 - Processo de *login*.

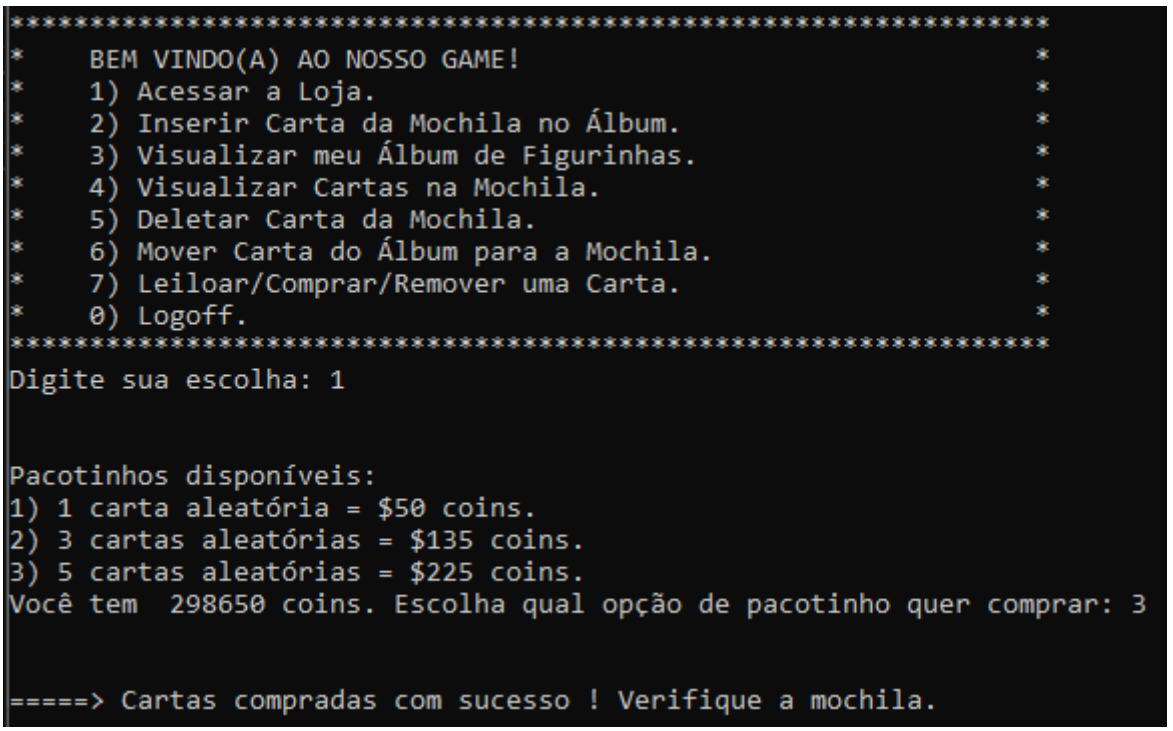

Figura 5 - Escolha do usuário por uma funcionalidade.

Mas como o Server entende o que deve ser feito ? Simples: padronizamos nossas mensagens do Cliente → Server da seguinte forma:

- ➔ Se o Cliente quiser realizar uma operação de **Cadastro,** pedimos a ele os dados de cadastro e enviamos uma mensagem para o Server da seguinte forma: **cadastro**:**0:*nickname*:senha:nome:email,** espelhando no modelo de **cabeçalhos** da comunicação em Redes de Computadores. Logo, quando o servidor recebe a mensagem acima, ele consegue identificar através da utilização de *split()* que deve ser feito um **cadastro** utilizando os argumentos da posição 1 (0), posição 2 (*nickname*), posição 3 (senha), posição 4 (nome) e posição 5 (email), fazendo todas as verificações com os dados fornecidos e os dados do Banco de Dados e retornando uma mensagem amigável ao usuário com o resultado da operação.
- ➔ Da mesma forma, quando o Cliente solicita um **Login,** solicitamos as credenciais e enviamos a mensagem: **login**:***nickname*:senha.** Quando o Server recebe a mensagem, ele identifica o cabeçalho **login,** já recebe as credenciais e faz as solicitações e verificações mediadas pelos retornos do Banco de Dados. Após isso, ele retorna uma mensagem ao Cliente.

Logo, para qualquer operação que o usuário queira fazer, nós usamos a convenção de especificar a operação que ele quer fazer em uma espécie de cabeçalho da mensagem e passamos os demais parâmetros separados por "**:**", como exemplificado anteriormente. Tal abordagem funcionou de maneira extremamente satisfatória.

Arquitetura do Servidor

Para a arquitetura do servidor foi utilizado uma classe principal para o servidor, chamada “Server”. Essa classe é responsável por **dois** aspectos principais: inicializar e utilizar o **banco de dados** e realizar as **respostas das requisições** dos clientes.

A classe do servidor possui um método principal chamado “start()”, como o nome sugere, realiza a inicialização do servidor a partir de um endpoint do host (para serviços do servidor) e de uma porta. A partir desse estabelecimento do servidor, é feito um laço de repetição responsável pela conexão com clientes, e, a partir dessa conexão, o servidor recebe potencialmente várias requisições de um ou de vários clientes.

```
def start(self):
    # Inicia o serviço do servidor
    # Definir uma porta específica para o serviço
    endpoint = (self._host, self._port)
    try:
        self._tcp.bind(endpoint)
        self._tcp.listen(1)  # Possíveis conexões
```

Figura 6 - Conexão com o host local (localhost) para estabelecer os serviços do servidor.

```
finally:

    while True:
        # Con é informações do cliente | cliente é ip e porta do cliente
        con, cliente = self._tcp.accept()
        # Executar os serviços.
        self._service(con, cliente, cursor, connection)
```

Figura 7 - Responsável por esperar um cliente específico, esse cliente pode utilizar vários serviços várias vezes.

Outra importante tarefa da função start() do servidor é o estabelecimento da conexão com o banco de dados da aplicação, que é responsável por guardar informações dos usuários, seus álbuns e mochilas, como representado na figura 2. Essa conexão se dá da seguinte forma:

```
try:
    connection = mysql.connector.connect(host='localhost',
                                         database='bd_distribuidos',
                                         user='root',
                                         password='1234')

    if connection.is_connected():
        cursor = connection.cursor()
        db_Info = connection.get_server_info()
        print(
            "Connected to MySQL database... MySQL Server version on ", db_Info)
except db_error:
    print("Error while connecting to MySQL", db_error)
```

Figura 8 - Note que em “user” e “password” vai depender de onde o usuário da aplicação armazenou o banco de dados, além de sua senha.

Outro método importante da classe servidor é o “_service()”, esse é nosso método central para serviços, que centraliza as requisições dos usuários e despacha essas requisições para outros métodos específicos para lidar com elas. Assim, como um cliente pode realizar vários serviços, existe um laço de repetição “while” em serviço que recebe potencialmente várias requisições e as despacham para os métodos específicos.

```
def _service(self, con, cliente, cursor, connection):
    # Método dos serviços do servidor (banco de dados)
    rt = None  # timer
    while True:
        print(f"Atendendo o cliente {cliente}")
        try:
            # Recebendo a mensagem do cliente, dados brutos, fluxo de bytes
            mensagem = con.recv(2048)
            mensagem_decodificada = str(mensagem.decode(
                'ascii'))  # Bytes representam caracteres
            """
                A nossa ideia é splitar a mensagem com ":".
                Mensagem de login -> login:usuario:senha
                Mensagem de cadastro-> cadastro:coins:nickname:password:nome:email
            """
```

Figura 9 - Responsável por criar um *loop* que é responsável por receber requisições (con.recv()) de um usuário.

Dentro do método “_service()” existe, então, uma parte do código responsável por despachar os serviços requisitados por um usuário.

```
msg = mensagem_decodificada.split(
    ":")  # Nossa mensagem é da forma: acao:operadores:...
print(f"Mensagem: {msg}")
if (msg[0] == "cadastro"):
    resposta = self.__cadastro(
        cursor, connection, msg[1], msg[2], msg[3], msg[4], msg[5])
elif (msg[0] == "login"):
    if (rt != None):
        rt.stop()
    rt = RepeatedTimer(
        120, self.__getCoins, cursor, connection, msg[1])
    resposta = self.__login(cursor, connection, msg[1], msg[2])
```

Figura 10 - Responsável por receber uma requisição do usuário e despachar para outro método realizar seu tratamento.

```
elif (msg[0] == "loja"):
    cartas = []
    for i in range(4, len(msg)):

        cartas.append(int(msg[i]))

    resposta = self.__compraCartaLoja(
        cursor, connection, msg[1], msg[2], msg[3], cartas)
elif (msg[0] == "minhaMochila"):
    resposta = self.__minhaMochila(cursor, connection, msg[1])
elif (msg[0] == "insereAlbum"):
    resposta = self.__insereAlbum(
        cursor, connection, msg[1], msg[2], msg[3])
elif (msg[0] == "visualizaAlbum"):
    resposta = self.__visualizaAlbum(
        cursor, connection, msg[1])
elif (msg[0] == "leiloaCarta"):
    resposta = self.__colocaCartaLeilao(
        cursor, connection, msg[1], msg[2], msg[3])
elif (msg[0] == "mostraCartasLeilao"):
    resposta = self.__mostraCartasLeilao(cursor, connection)
elif (msg[0] == "vendeLeilao"):
    resposta = self.__vendeLeilao(
        cursor, connection, msg[1], msg[2])
```

Figura 11 - Responsável por receber uma requisição do usuário e despachar para outro método realizar seu tratamento.

```
elif (msg[0] == "retiraAlbum"):
    # print(
    #     f'{msg[1].strip()}:{msg[2].strip()}:{msg[3].strip()}')
    resposta = self.__retiraCartaAlbum(
        cursor, connection, msg[1], msg[2], msg[3])
elif (msg[0] == 'deletaCarta'):
    resposta = self.__deletaCarta(
        cursor, connection, msg[1], msg[2])
# __deletaCarta
elif (msg[0] == "retiraCartaLeilao"):
    resposta = self.__retiraCartaLeilao(
        cursor, connection, msg[1])
elif (msg[0] == "logout"):
    rt.stop()
    rt = None
else:
    break

# Converter a resposta para bytes também.
con.send(bytes(str(resposta), 'ascii'))
print(f"{cliente} -> requisição atendida !")
```

Figura 12 - Responsável por receber uma requisição do usuário e despachar para outro método realizar seu tratamento, além de fazer o envio da resposta de volta para o usuário.

Para entender como ocorre a comunicação com o banco de dados, é importante adentrar a fundo em pelo menos algumas das funções citadas nas figuras acima. Para esse fim, iremos analisar as funções: “__cadastro()”, “__login()” e da “__getCoins()”.

A função **cadastro**, como o nome sugere, faz a inserção de um novo usuário no banco de dados. Porém, para essa funcionalidade ocorrer de forma correta, precisamos verificar se o nickname escolhido já existe, visto que eles são únicos.

```
def __cadastro(self, cursor, connection, coins, nickname, password, nome, email):
    try:
        """
            Verificar se essas credenciais já estão no banco:
        """
        queryVerificacao = """SELECT * FROM usuario WHERE (usuario.nickname = '"""+str(
            nickname)+"""');"""
        # print(queryVerificacao)
        cursor = connection.cursor()
        cursor.execute(queryVerificacao)
        verificacao = cursor.fetchall()
        #print(f"Numero de registros: {len(verificacao)}")
        if (len(verificacao) > 0):
            return("======> [ERRO] Nickname ja existente! Tente outro.")
        else:
```

Figura 13 - Verificar se o nickname escolhido já existe.

No caso do nickname ser válido, precisamos criar um álbum para esse usuário (dado que cada usuário possui um e apenas um álbum).

```
queryCriaAlbum = """INSERT INTO album VALUES();"""
result = cursor.execute(queryCriaAlbum)
connection.commit()
```

Figura 14 - Responsável por criar um novo álbum para o usuário.

Após isso, precisamos pegar o id gerado desse álbum (que é o último inserido no banco, portanto seu valor é o maior) e gerar 30 slots vazios.

```
cursor = connection.cursor()
cursor.execute("SELECT MAX(idAlbum) AS ultimoValor FROM album")
resultado = cursor.fetchall()
for id in resultado:
    resultado = id[0]
"""
    Setando as cartas nos slots respectivos desse usuário
"""
for i in range(1, 31):
    queryInsertSlot = """INSERT INTO album_has_slot values (""" + str(
        resultado)+""","""+str(i)+""","""+str(i)+""",False);"""
    # print(queryInsertSlot)
    result = cursor.execute(queryInsertSlot)
    connection.commit()
```

Figura 15 - Criar 30 slots para o álbum que criamos.

Posteriormente, é criado a mochila do usuário, que inicia sem cartas. E, por fim, é criado o usuário em si, que faz a utilização de todos esses dado realizados anteriormente, que se traduzem, de forma direta, em no usuário conter um álbum e uma mochila.

```
queryCriaMochila = """INSERT INTO mochila values();"""
result = cursor.execute(queryCriaMochila)
connection.commit()
#print("==> Mochila criada com sucesso !")
"""
    Montando a query de inserção do usuário em sino Banco de Dados
"""
query = """INSERT INTO usuario (coins,nickname,password,nome,email,
        Mochila_idMochila,Album_idAlbum) values (""" + \
    coins+",'"+nickname+"','"+password+"','"+nome+"','" + \
        email+"',"+str(resultado)+","+str(resultado)+""");"""
# print(f"{query}")
result = cursor.execute(query)
connection.commit()
#print("===> Todas as querys foram executadas")
return("=====> Cadastro realizado com sucesso !")
```

Figura 16 - Criando mochila e o usuário que possui a mochila e álbum criados.

Partindo para a função **login**, ela é responsável basicamente por fazer um "match" de nickname e senha, e retornar uma estrutura de dados que "contém" os dados do usuário que está efetuando o login. Vale ressaltar que as informações de login ficam armazenadas no cliente, por isso retornamos esse valor.

```
def __login(self, cursor, nickname, senha):
    """
        Formato padrão da query de seleção
    """
    query = """SELECT * FROM usuario WHERE (BINARY nickname= '""" + \
        nickname+"'"+" and password='"+senha+"');"
    # print(f"{query}")
    """
        Tentando executar a query de seleção
    """
    try:
        result = cursor.execute(query)
        """
            Retorna uma lista com os registros encontrados:
        """
        resultados = cursor.fetchall()
        if(len(resultados) > 0):
            """
                PRÓXIMO PASSO: MANDAR ESSAS INFORMAÇÕES DE LOGIN
                PARA SEREM CARREGADAS NA NOSSA TELA INICIAL DO GAME
                PARA PODER CARREGAR O ALBUM DESSE USUÁRIO, SUA
                MOCHILA E DEMAIS INFORMAÇÕES.
            """
            return(("login",) + tuple(resultados[0]))
        else:
            return("=====> [ERRO] Usuario ou senha invalidos!")
    except db_error:
        return("=====> [ERRO NO BANCO] Erro ao realizar o login !")
```

Figura 17 - Realização de login, observe que primeiro são feito dois filtros, um para nickname (que é único), verificando se o mesmo bate com algo no banco de dados, e também é verificado se a senha informada é igual àquela do registro encontrado no banco de dados.

Para a função **getCoins**, é importante indicar o motivo de sua existência: fazer com que o usuário ganhe 25 moedas (coins) a cada 2 minutos logado. Para realizar essa funcionalidade foi feita a utilização de **threads**, onde assim que o login é realizado uma nova thread é criada com a função gerar as coins a cada 2 minutos e assim que o logout é realizado. Para a criação e destruição das threads foi criado uma classe chamada "RepeatedTimer", que recebe como argumento do "construtor" o tempo (120 segundos), a função que é realizada a cada 2 minutos (getCoins) e os argumentos da própria função getCoins, que são os dois objetos de comunicação

com o banco de dados (cursor e connection) e o nickname do usuário que ganhará as coins por estar logado a 2 minutos.

```
class RepeatedTimer(object):
    def __init__(self, interval, function, *args, **kwargs):
        self._timer = None
        self.interval = interval
        self.function = function
        self.args = args
        self.kwargs = kwargs
        self.is_running = False
        self.start()

    def _run(self):
        self.is_running = False
        self.start()
        self.function(*self.args, **self.kwargs)

    def start(self):
        if not self.is_running:
            self._timer = Timer(self.interval, self._run)
            self._timer.start()
            self.is_running = True

    def stop(self):
        self._timer.cancel()
        self.is_running = False
```

Figura 18 - Classe RepeatedTimer, para criação e destruição de threads.

```
elif (msg[0] == "login"):
    if (rt != None):
        rt.stop()
    rt = RepeatedTimer(
        120, self.__getCoins, cursor, connection, msg[1])
    resposta = self.__login(cursor, connection, msg[1], msg[2])
```

Figura 19 - Geração de uma nova thread ao realizar login.

```
elif (msg[0] == "logout"):
    rt.stop()
    rt = None
```

Figura 20 - Destruição da *thread* ao realizar logout.

Nosso servidor disponibiliza vários serviços além dos três citados anteriormente, para isso consulte a tabela a seguir:

Método (Serviço)	Descrição
loja()	Responsável por abrir a loja para o cliente escolher a compra de **pacotinhos** (1, 3 ou 5 cartas aleatórias).
minhaMochila()	Mostra as cartas que estão na mochila do usuário com login realizado.
insereAlbum()	Responsável por inserir uma carta do usuário com login realizado no álbum.
retiraAlbum()	Responsável por retirar uma carta escolhida pelo usuário com login realizado do álbum.
visualizaAlbum()	Responsável por mostrar o álbum do usuário com login realizado.
colocaCartaLeilao()	Responsável por colocar uma carta escolhida pelo usuário com login realizado no leilão (anunciar carta).
mostraCartasLeilao()	Responsável por mostrar todas as cartas que estão atualmente em leilão.
vendeLeilao()	Responsável por comprar uma carta que está em leilão.
deletaCarta()	Responsável por deletar uma carta do usuário com login realizado, essa carta não pode ser recuperada (basicamente jogar fora.
retiraCartaLeilao()	Responsável por retirar uma carta do usuário com login realizado do leilão.

Comunicação

Nesta seção, explicaremos em alto nível como o processo de comunicação Cliente-Server está implementado. Como já introduzido anteriormente, tanto o Servidor quanto o Cliente estão conectados localmente no IP ***localhost*.**

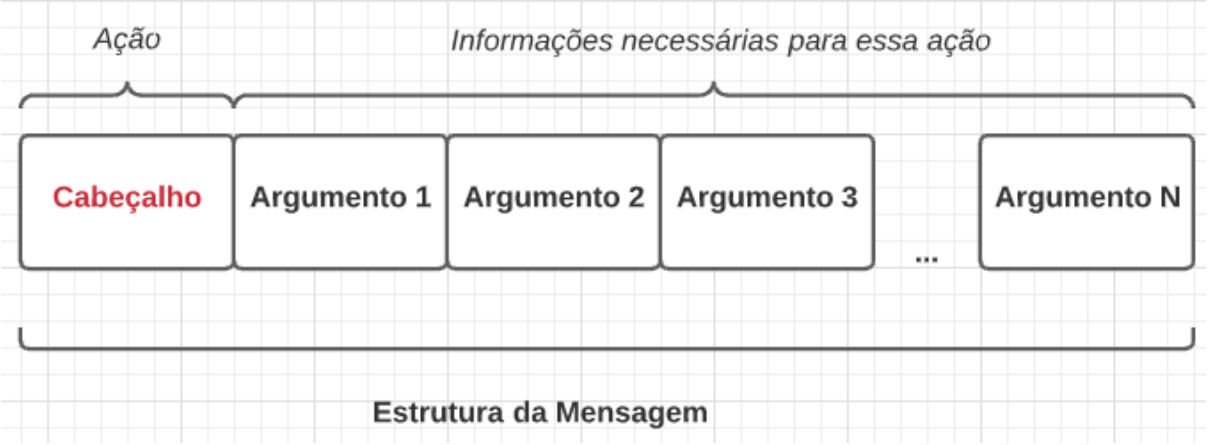

Figura 21 - Primeiro, montamos a mensagem com base na estrutura acima no Cliente e a enviamos ao Server.

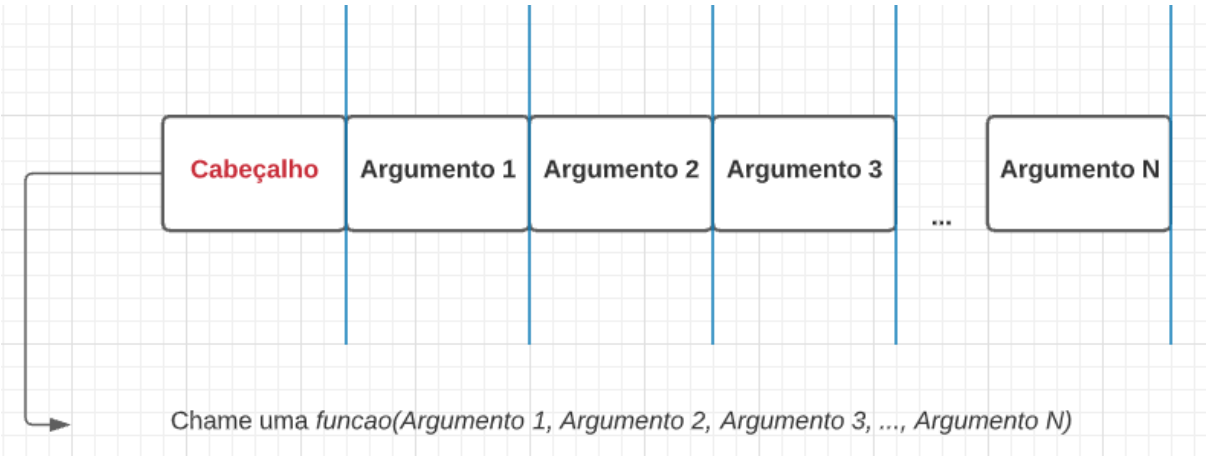

Figura 22 - Depois, o Server identifica o que deve ser feito observando o cabeçalho da mensagem e chama uma função específica que resolva aquela requisição.

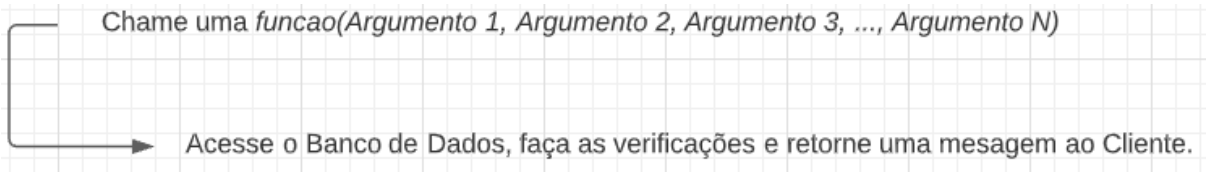

Figura 23 - Essa função do Server faz acessos à dados persistentes no Banco de Dados a fim de processar a requisição do Cliente e retornar uma resposta.

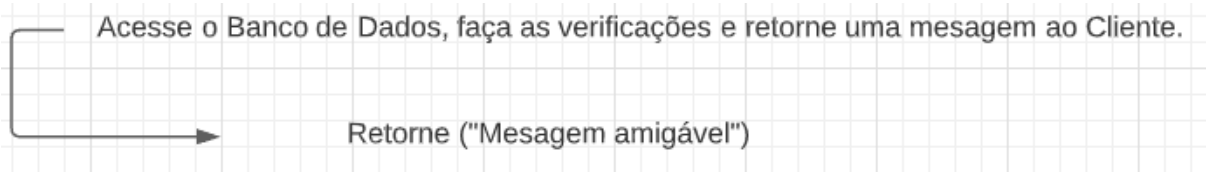

Figura 24 - Por fim, as funções do Server retornam mensagens amigáveis ao Cliente, informando se a solicitação pôde ser processada como deveria, se houve algum erro ou problema relacionado às regras de negócio.

Conclusão

Um dos principais benefícios de um trabalho prático é a concretização de conhecimentos teóricos em um arsenal de conhecimentos práticos para os alunos, podemos concluir, portanto, que o trabalho acima apresentado proporciona, de fato, o alavancar da curiosidade e o refinamento dos conhecimentos obtidos em sala de aula.

Inicialmente o grupo teve uma pequena dificuldade para entender a forma de comunicação entre o cliente e o servidor a partir da utilização de *sockets*, mas com a utilização de materiais auxiliares disponibilizados na própria disciplina, tais dificuldades foram sanadas sem mais nenhum bloqueio.

É possível, então, concluir que o trabalho cumpre seus objetivos acadêmicos, gerando um resultado final satisfatório e, de fato, frutífero no que diz respeito à obtenção de um conhecimento mais ancorado em uma prática, que o torna mais completo e eficaz.