Resumo das implementações do trabalho 1 de Sistemas Distribuídos
Marcelo Silva de Lima - 510097

Resumo dos códigos

1) Cliente

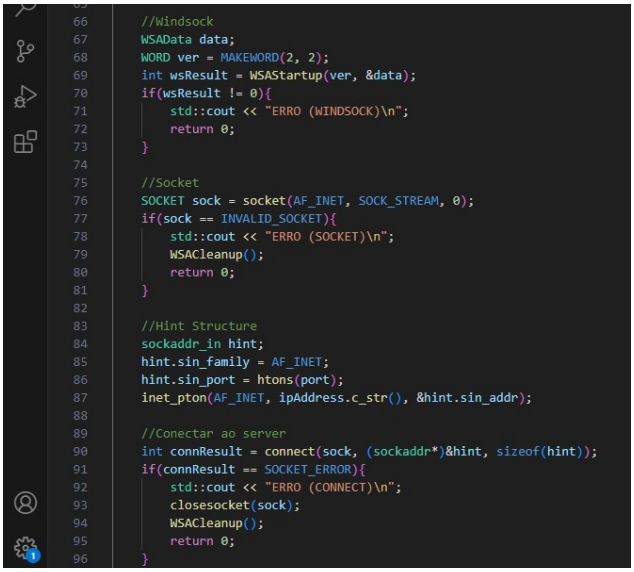
Para o funcionamento apropriado do código, foi necessário a inclusão de diversas bibliotecas (para os outputs no console e em arquivo), bem como definições específicas para que os sockets funcionassem corretamente (os **define** e o **pragma comment**).

```
#ifndef _WIN32_WINNT
    #define _WIN32_WINNT 0x0600
#elif _WIN32_WINNT < 0x0600
    #undef _WIN32_WINNT
    #define _WIN32_WINNT 0x0600
#endif


#include <fstream>
#include <iostream>
#include <string>
#include <vector>
#include <sstream>
#include <WS2tcpip.h>
#pragma comment (lib, "ws2_32.lib")
```

A classe “Pessoa” representa o objeto a ser enviado para o servidor. Nela, foram incluídos dois métodos: o “to_str()” converte os dados de “Pessoa” para uma string que será enviada pela rede. No outro lado, o servidor reinterpretará essa string e criará um novo objeto análogo ao do cliente. Já o “operator«” serve para printar o objeto pelo console.

```
class Pessoa {
    std::string nome;
    std::string cpf;
    int idade;
public:
    Pessoa(std::string nome, std::string cpf, int idade){
        this->cpf = cpf;
        this->idade = idade;
        this->nome = nome;
    }

    std::string to_str(){
        std::string str;
        str = '$' + nome + '$' + cpf + '$' + std::to_string(idade);
        return str;
    }

    friend std::ostream& operator<<(std::ostream& os, const Pessoa& pessoa){
        os << "Nome: " << pessoa.nome << " | CPF: " << pessoa.cpf << "\nIdade: " << pessoa.idade << '\n';
        return os;
    }

};
```

Abertura do socket, junto com as configurações adicionais. Junto deles, foram adicionadas algumas condições para responder em caso de erro onde houve a falha.



```
//Windsock
WSAData data;
WORD ver = MAKEWORD(2, 2);
int wsResult = WSAStartup(ver, &data);
if(wsResult != 0){
    std::cout << "ERRO (WINDSOCK)\n";
    return 0;
}

//Socket
SOCKET sock = socket(AF_INET, SOCK_STREAM, 0);
if(sock == INVALID_SOCKET){
    std::cout << "ERRO (SOCKET)\n";
    WSACleanup();
    return 0;
}

//Hint Structure
sockaddr_in hint;
hint.sin_family = AF_INET;
hint.sin_port = htons(port);
inet_pton(AF_INET, ipAddress.c_str(), &hint.sin_addr);

//Conectar ao server
int connResult = connect(sock, (sockaddr*)&hint, sizeof(hint));
if(connResult == SOCKET_ERROR){
    std::cout << "ERRO (CONNECT)\n";
    closesocket(sock);
    WSACleanup();
    return 0;
}
```

Para o envio dos dados, o programa usa de uma execução do send()/recv() para enviar o tamanho do vetor, seguido por um loop que envia cada um dos elementos do vetor (usando do método to_str() da classe Pessoa, já citado anteriormente).

```
char buff[4096];
std::string userInput;


std::string aux_quant = '&' + std::to_string(quant) + '&';


int sendResult = send(sock, aux_quant.c_str(), aux_quant.size() + 1, 0);
ZeroMemory(buff, 4096);
int bytesReceived = recv(sock, buff, 4096, 0);


std::cout << "-----------------------------------\n";
std::cout << "Retorno Server\n";
std::cout << "-----------------------------------\n";

if(bytesReceived > 0){
    std::cout << "SERVER " << std::string(buff, 0, bytesReceived) << '\n';
}
```

Para enviar o tamanho do vetor;

```
for(int i = 0; i < quant; i++){
    userInput = pessoas[i]->to_str();

    if(userInput.size() > 0){
        int sendResult = send(sock, userInput.c_str(), userInput.size() + 1, 0);
        if(sendResult != SOCKET_ERROR){
            ZeroMemory(buff, 4096);
            int bytesReceived = recv(sock, buff, 4096, 0);

            if(bytesReceived > 0){
                std::cout << "SERVER " << std::string(buff, 0, bytesReceived);
            }
        }

    }
}
```

Para enviar os dados do objeto;

2) Servidor

O server possui um vetor do objeto pessoa (o servidor possui as mesmas bibliotecas e classes do cliente), seguido de um socket “listening”, que é usado para ouvir pelo cliente.

```
int main(){
    std::vector<Pessoa*> pessoas;

    //Windsock
    WSADATA wsData;
    WORD ver = MAKEWORD(2, 2);

    int wsOk = WSAStartup(ver, &wsData);
    if (wsOk != 0){
        std::cout << "ERRO (WSA)\n";
        return 0;
    }

    //Socket
    SOCKET listening = socket(AF_INET, SOCK_STREAM, 0);
    if(listening == INVALID_SOCKET){
        std::cout << "ERRO (SOCKET)\n";
        return 0;
    }

    //IP e porta
    sockaddr_in hint;
    hint.sin_family = AF_INET;
    hint.sin_port = htons(54000);
    hint.sin_addr.S_un.S_addr = INADDR_ANY;

    //BIND
    bind(listening, (sockaddr*)&hint, sizeof(hint));
```

O socket “listening” recebe o cliente e o aceita, criando a socket “clientSocket” para receber os dados do cliente. Após isso, a socket “listening” é fechada.

```
//Windsock e BIND
listen(listening, SOMAXCONN);

//Conexão
sockaddr_in client;
int clientSize = sizeof(client);

SOCKET clientSocket = accept(listening, (sockaddr*)&client, &clientSize);
if(clientSocket == INVALID_SOCKET){
    std::cout << "ERRO (SOCKET CLIENTE)\n";
    return 0;
}

char host[NI_MAXHOST];
char service[NI_MAXHOST];

ZeroMemory(host, NI_MAXHOST);
ZeroMemory(service, NI_MAXSERV);

if(getnameinfo((sockaddr*)&client, sizeof(client), host, NI_MAXHOST, service, NI_MAXSERV, 0) == 0){
    std::cout << "HOST CONECTADO\n";
}
else{
    inet_ntop(AF_INET, &client.sin_addr, host, NI_MAXHOST);
    std::cout << "HOST CONECTADO!\n";
}

//fecha socket
closesocket(listening);
```

Inicia-se um loop para receber os dados através do socket “clientSocket”, usando o recv(). O loop se encerra quando o cliente desconectar ou se houver um erro no recv()

```
while (true){
    ZeroMemory(buff, 4096);

    //std::cout << 1;

    int byteReceived = recv(clientSocket, buff, 4096, 0);
    if(byteReceived == SOCKET_ERROR){
        std::cout << "ERRO RECV()\n";
        break;
    }

    if(byteReceived == 0){
        std::cout << "CLIENTE DESCONECTOU\n";
        break;
    }
```

Uma string “receiver” recebe os caracteres do buffer, logo em seguida sendo interpretada. O cliente envia as informações separadas por “&” ou “$”, sendo que os dados com “&” representam o tamanho do vetor, enquanto os separados por “$” são os dados do objeto Pessoa gerados pelo método “to_str()”. Nesse último caso, a string é quebrada em “nome”, “cpf” e “idade”, que logo são convertidos em um objeto no vetor “pessoas”.

Um stringstream recebe os objetos convertidos pelo “operator«”, com a finalidade de serem enviados de volta ao cliente.

```
    std::string receiver(buff, 0, byteReceived);

    std::stringstream ss;
    std::string aux;

    if (receiver[0] == '&'){

        ss << "Tamanho do Vetor: " << receiver[1] << '\n';
        //std::cout << "!!!!" << aux;
        //send(clientSocket, aux.c_str(), aux.length() + 1, 0);
    }
    else if(receiver[0] == '$'){
        std::string nome, cpf;
        int idade;
        receiver = receiver.substr(receiver.find('$') + 1);
        nome = receiver.substr(0, receiver.find('$'));

        receiver = receiver.substr(receiver.find('$') + 1);
        cpf  = receiver.substr(0, receiver.find('$'));

        receiver = receiver.substr(receiver.find('$') + 1);
        idade = stoi(receiver.substr(0, receiver.find('$')));

        pessoas.push_back(new Pessoa(nome, cpf, idade));


        ss << *pessoas[pessoas.size()-1];
    }

    aux = ss.str();
    send(clientSocket, aux.c_str(), aux.length() + 1, 0);
```

No fim, tanto o cliente quanto o server exibem o output pelo console e por um arquivo.

```
std::cout << "Tamanho do Vetor: " << pessoas.size() << '\n';
for(int i = 0; i < pessoas.size(); i++){
    std::cout << *pessoas[i];
}

//////////////////////////     ARQUIVO
//////////////////////////
std::fstream fs;
fs.open ("server.txt", std::fstream::in | std::fstream::out | std::fstream::app);

fs << "Tamanho do Vetor: " << pessoas.size() << '\n';

for(int i = 0; i < pessoas.size(); i++){
    fs << *pessoas[i];
}

fs.close();
//////////////////////////
//////////////////////////


closesocket(clientSocket);
WSACleanup();
```

Servidor

```
    //Print
    std::cout << "---------------------------------\n";
    std::cout << "Print\n";
    std::cout << "---------------------------------\n";
    std::cout << "Tamanho do Vetor: " << pessoas.size() << '\n';
    for(int i = 0; i < pessoas.size(); i++){
        std::cout << *pessoas[i];
    }


    //ARQUIVO

    std::fstream fs;
    fs.open ("client.txt", std::fstream::in | std::fstream::out | std::fstream::app);

    fs << "Tamanho do Vetor: " << pessoas.size() << '\n';

    for(int i = 0; i < pessoas.size(); i++){
        fs << *pessoas[i];
    }

    fs.close();

    closesocket(sock);
    WSACleanup();
```

Cliente

3) Sistema de gestão de Apiário

Para a criação do sistema do apiário, foram criadas duas classes: a classe “Apicultor”, que representa um funcionário; e a classe Apiário, que representa um conjunto de colmeias e funcionários. Os Apicultores podem ser adicionados a um Apiário para trabalhar.

```
import socket
import pickle


class Apicultor:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def str_apicultor(self):
        return "Nome: " + self.nome + " | Idade: " + str(self.idade)


class Apiario:
    def __init__(self, id, num_colmeias):
        self.id = id
        self.num_colmeias = num_colmeias
        self.apicultores = []

    def str_apiario(self):
        strg = ""
        strg += "Apiario " + str(self.id) + " com " + str(self.num_colmeias) + " colmeias"
        for apic in self.apicultores :
            strg += '\n' + apic.str_apicultor()

    def add_apicultores(self, apicultor):
        self.apicultores.append(apicultor)

    def set_id(self, id):
        self.id = id
```

Inicialização do socket “s”, conectado ao IP 127.0.0.1, na porta 54000

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("127.0.0.1", 54000))
```

Ao executar o cliente, é possível usar uma variedade de comandos para se comunicar com o servidor. É possível adicionar Apicultores e Apiarios e consultar um ou todos os Apiarios.

```
# Using indica se é uma mensagem de comando ou um objeto/resposta. Se True, o programa não lê imput
# Comandos (para o cliente):
#   add_apiario {quantidade de colmeias}    ->  (client) Adicionar Apiário
#   add_apicultor {nome} {idade} {apiario}  ->  (client) Adicionar Apicultor
#   consulta {operacao} {id do apiario}     ->  (client) Consulta um ou mais Apiários.
#                                               operacao = "apiario"    -> Consulta um Apiário específico
#                                               operacao = "all"        -> Consulta todos os Apiários
```

O programa usa de um sistema de requisição e resposta de mais de uma etapa, ou seja, para enviar um objeto, será necessário enviar primeiro uma mensagem para informar o servidor qual o objeto a ser enviado, seguida por uma segunda com o objeto em si.

Para isso funcionar, o cliente precisa ficar sem receber input por uma iteração (na qual será enviado o objeto). Essa ausência de input é marcada pela variável “using”, que quando verdadeira não recebe entrada.

```
using = False
while True :
    line = ""

# Using indica se é uma mensagem de comando ou um objeto/resposta. Se True, o programa não lê imput
    if using == False :
        line = input(line)
        cmd = line.split(" ")
```

O client também possui os comandos “exit” e “end”, que servem para finalizar o loop e o server. Cada um dos comandos é tratado por uma sequência de ifs e elifs, enviando mensagens ao servidor de acordo com o tipo de requisição (note como quando “using” é False, envia-se uma mensagem, enquanto com o “using” True, é enviado um objeto).

```
obj = ''
if cmd[0] == "exit" or cmd[0] == "end":
    obj = "end"
    msg = pickle.dumps(obj)
    s.send(msg)
    break

elif cmd[0] == "add_apiario" :
    if using == False :
        obj = "$A1"
    else :
        obj = Apiario(0,int(cmd[1]))

elif cmd[0] == "add_apicultor" :
    if using == False :
        obj = "$A2" + cmd[3]
    else :
        obj = Apicultor(cmd[1],int(cmd[2]))

elif cmd[0] == "consulta" :
    if cmd[1] == "apiario":
        obj = "$C0" + cmd[2]
    elif cmd[1] == "all":
        obj = "$C1"

#Serializando a mensagem
msg = pickle.dumps(obj)
s.send(msg)
```

Por fim, o “obj” é serializado usando a biblioteca “pickle” do Python, sendo enviado em seguida.

No fim do loop, o socket recebe o retorno do servidor e o desserializa usando o pickle. Nesse retorno, o socket pode receber os comandos “$A1” e “$A2”, que representam, respectivamente, a “autorização” do servidor para enviar os objetos do Apiario e do Apicultor); “#A1”, que representa uma mensagem em texto, que é printada logo em seguida; ou o comando “#E0”, que encerra o loop. No final, o socket é fechado.

```
    # Comandos de retorno #
    data = s.recv(1024)

    #Desserialização
    data = pickle.loads(data)

    #Comando para os if/else
    retn = data[0:3]


    if retn == "$A1" or retn == "$A2":
        using = True

    elif retn == "#A1":
        print(data[3:])
        using = False

    elif retn == "#E0":
        break
s.close()
```

Já o servidor, além de contar com as mesmas classes e métodos do cliente, possui uma lista que guarda os Apiarios (bem como os apicultores presentes dentro das suas listas).

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.bind(("127.0.0.1", 54000))
s.listen()
clientsocket, address = s.accept()
print(f"Conectado com {address}!")

cmd = b''
lista = []

# "Using" indica se é uma mensagem de comando ou um objeto/resposta. Caso for True, o programa não lerá input
# Comandos (para o servidor):
#   $A{numero} ->   (client) Adicionar objeto (numero = 1 (Apiario) ; numero = 2 (Apicultor))
#                   (server) Autorização para enviar objeto
#   $C{numero}{index} ->   (client) Consulta {numero = 0 (Apiario) ; numero = 1 (Todos os Apiarios)}
#                                   {index = posição do Apiario na lista}
#   #A1 -> Resposta do servidor, em texto.
```

O cliente envia uma série de comandos para o servidor:

- “$A1” e “$A2”: Autorizam a adição adicionam um novo Apiario e Apicultor, respectivamente;
- “$C0” e “$C1”: Consulta de um e todos os apiários, respectivamente. No caso de “$C0”, é enviado a posição do Apiario que deve ser consultado (“$C02”, por exemplo, consulta o terceiro Apiario na lista).
- #A1: Representa uma mensagem de texto qualquer que o servidor envia para o cliente.

Algumas das condicionais usadas no loop do servidor. “pickle.loads()” é uma etapa de desserialização.

```
    elif cmd == "$A1" :
        #Using = False -> Comando para indicar o futuro envio de um objeto
        #Using = True  -> O objeto em si
        if using == False :
            using = True
            retn = cmd
        else :
            #Dessempacotamento da mensagem (se for um objeto Apiário)
            obj = pickle.loads(msg)
            obj.set_id(len(lista))
            lista.append(obj)

            using = False
            retn = "#A1Apiario Adicionado!"

    #Para adicionar um Apicultor
    elif cmd[0:3] == "$A2" :
        #Using = False -> Comando para indicar o futuro envio de um objeto
        #Using = True  -> O objeto em si
        if using == False :
            using = True
            retn = cmd[0:3]
        else :
            #Dessempacotamento da mensagem (se for um objeto Apicultor)
            obj = pickle.loads(msg)

            if(int(cmd[3:]) < len(lista)):
                lista[int(cmd[3])].add_apicultores(obj)
                retn = "#A1Apicultor Adicionado!"
            else:
                retn = "#A1ERRO: Apiario invalido"
            using = False
```

No final, a mensagem de resposta (gerada de acordo com a condicional equivalente a mensagem do cliente) é empacotada e enviada pelo socket.

```
    # Consultas
    #   $C0 = Consultar um Apiário específico
    #   $C1 = Consultar todos os Apiários
    elif cmd[0:3] == "$C0" :
        retn += "#A1"
        retn += lista[int(cmd[3])].str_apiario()

    elif cmd[0:3] == "$C1" :
        retn += "#A1"
        for apiario in lista :
            retn += '\n' + apiario.str_apiario()

    #Caso não houver um comando reconhecível, retorna "#E0", indicando falha
    else :
        retn = "#E0"

    #Empacotamento da mensagem de retorno
    retn = pickle.dumps(retn)
    clientsocket.sendall(retn)
s.close()
```

4) Sistema de votação

No sistema de votação, foram usadas as bibliotecas socket, threading (para as threads), json (para o envio dos arquivos) e datetime (para definir o horário limite). A classe “Candidato” representa o principal objeto do sistema.

```python
import socket
import threading
import json
from datetime import datetime

class Candidato:
    def __init__(self, nome, partido,numero):
        self.nome = nome
        self.partido = partido
        self.numero = numero
        self.votos = 0
        self.voto_percentual = 0

    def add_voto(self):
        self.votos += 1

    def get_votos(self):
        return self.votos

    def get_numero(self):
        return self.numero

    def str_candidato(self):
        return "Num: " + self.numero + " | Nome: " + self.nome + " | Partido: " + self.partido
```

Foi usado um conceito similar ao dos anteriores, com alguns comandos sendo enviados ao servidor no formato JSON indicando qual é a finalidade da comunicação. Tais comandos foram:

END: Termina o loop do cliente/admin, encerrando a conexão;
LOGIN: Faz login no servidor através de um código;
VOTAR: Vem acompanhado do número do candidato, adicionando o voto ao candidato;
LISTEN: Interrompe a capacidade do programa de inserir dados pelo input, limitando-o a ouvir;
LOG-A: Login do administrador;
ADD-C: Adiciona um candidato, seu partido e seu número;
DEL-C: Apaga um candidato pelo seu número;
ALERT: Envia ao servidor uma mensagem que deve ser mandada por multicast;
TIMER: Envia ao servidor uma mensagem contendo o tempo atual, é parte de um terceiro tipo de cliente “timer”.

No server, a função handle_client() é responsável por receber as mensagens dos votantes (client), administradores (admin) e timer. Ela é usada no threading para gerar a concorrência de execução entre os clientes.

A função handle_client() possui mais de 130 linhas de código, portanto detalharei apenas os comandos mais importantes:

O comando LOGIN recebe um cpf, que é inserido em um dicionário “connections”. Antes disso, é avaliado se o cliente já está logado ou se já logou antes, impedindo que o cliente faça login mais de uma vez.

Caso o login seja bem-sucedido, o votante pode votar pelo comando “VOTAR”, acompanhado do número do candidato (enviado pelo servidor após o login). Após isso, o cpf do votante recebe o valor “COMVOTO”, que o impede de votar de novo.

```
elif cmd == "LOGIN":
    cpf = json_recv["cpf"]
    if logged == False and cpf not in connections:
        cpf_votante = cpf

        connections[cpf] = "SEMVOTO"

        logged = True
        retn_msg = "[SERVER] LOGADO COM SUCESSO!"
        retn_msg += "\n\tEscolha seu candidato:"
        for cdt in candidatos:
            retn_msg += "\n\t\t" + candidatos[cdt].str_candidato()

    else:
        retn_msg = "[SERVER] VOCÊ JÁ ESTÁ LOGADO!"

elif cmd == "VOTAR":
    numero = json_recv["num"]
    if numero in candidatos and voted == False and logged == True:
        candidatos[numero].add_voto()
        votos += 1
        connections[cpf_votante] = "COMVOTO"
        voted = True

        print(f"Voto para {candidatos[numero].str_candidato()}, {candidatos[numero].get_votos()} votos totai

        retn_msg = "[SERVER] VOTO EFETUADO!"
        retn["cmd"] = "LISTEN"

    elif logged == False:
        retn_msg = "[SERVER] FAÇA O LOGIN!"
```

Entre os comandos do admin, estão o LOG-A (igual ao LOGIN, porém é exclusivo do ADMIN), o ADD-C (adicionar candidato, enviado no formato “ADD-C Nome Partido 0000”), que adiciona um candidato novo, junto de seu partido e número. O número vira a chave para o dicionário “candidatos”, que sempre é checado antes de ser adicionado para evitar mais de um candidato por número.

Já o DEL-C deleta um candidato pelo seu número, bem como subtrai seus votos do total de votos.

```
#Comandos do Admin
elif cmd == "LOG-A":
    cpf = json_recv["cpf"]
    if logged == False:
        connections[cpf] = "ADMIN"
        logged = True
        admin = True
        retn_msg = "[SERVER] LOGADO COM SUCESSO!"
    else:
        retn_msg = "[SERVER] VOCÊ JÁ ESTÁ LOGADO!"

elif cmd == "ADD-C" and admin == True:
    numero = json_recv["numero"]
    if numero not in candidatos:
        novo_candidato = Candidato(json_recv["nome"], json_recv["partido"], numero)
        candidatos[numero] = novo_candidato
        retn_msg = "[SERVER] CANDIDATO ADICIONADO!"
    else:
        retn_msg = "[SERVER] PARTIDO JÁ POSSUI UM CANDIDATO!"

elif cmd == "DEL-C" and admin == True:
    numero = json_recv["numero"]
    if numero in candidatos:
        votos -= candidatos[numero].get_votos()
        del candidatos[numero]
        retn_msg = "[SERVER] CANDIDATO DELETADO!"
    else:
        retn_msg = "[SERVER] CANDIDATO INEXISTENTE!"
```

O comando ALERT é enviado por um administrador, e, através de um socket UDP, envia para os votantes mensagens inportantes.

```
elif cmd == "ALERT" and admin == True:
    retn_msg = "[ALERTA] " + json_recv["msg"]
    list_json = {}
    list_json["msg"] = retn_msg
    retn_msg = json.dumps(list_json)
    udp_socket.sendto(retn_msg.encode("utf-8"), ("224.1.1.1", 55000))

    retn_msg = "[SERVER] MENSAGEM ENVIADA!"
```

Finalmente, o comando TIMER recebe de um cliente timer.py o tempo atual, que é comparado com o tempo limite determinado no início da execução. Caso o tempo enviado pelo timer superar o máximo, o programa envia o resultado por multicast.

```
#TIMER
if cmd == "TIMER" and time_left(max_time, json_recv["time_now"]) <= 0:
    print(max_time)
    list_aux = {}

    for x in candidatos:
        vt_percentual = str(float("{:.2f}".format((candidatos[x].get_votos()/votos)*100))) + "%"
        list_aux[vt_percentual] = candidatos[x]

    list_aux = dict(sorted(list_aux.items(), reverse=True))
    list_results = {}

    list_results["msg"] = "VENCEDOR: " + list(list_aux.values())[0].str_candidato() + " | Com " + str(candidatos[1

    for x in list_aux:
        list_results[x] = list_aux[x].str_candidato()
    list_results["cmd"] = "BREAK"

    retn_msg = json.dumps(list_results)
    udp_socket.sendto(retn_msg.encode("utf-8"), ("224.1.1.1", 55000))

    retn_msg = "BREAK"
    resultado_final = True
```

Durante todo o processo, os arquivos são enviados e recebidos em JSON, sendo reinterpretados sempre que chegam ao receptor.

```
msg = clientsocket.recv(1024).decode("utf-8")

retn = {}
retn_msg = ""
json_recv = json.loads(msg)
```

```
retn["msg"] = retn_msg

retn_json = json.dumps(retn)
clientsocket.sendall(retn_json.encode("utf-8"))

if retn["msg"] == "BREAK" or cmd == "END":
    clientsocket.close()
```

A main() do server, contendo as threads e dois sockets (TCP e UDP). O votante também possui dois sockets dos dois protocolos, já que recebe mensagens por multicast.

```
def main():

    now = datetime.now()
    tm_now = now.strftime("%H:%M:%S")

    print(datetime.now())
    limit_time = input("Insira a data limite (Formato HH:MM:SS): ")
    print(time_left(limit_time,tm_now))

    #SOCKET TCP
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind(("127.0.0.1", 54000))

    s.listen()

    #SOCKET UDP
    ttl = 5

    s_udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
    s_udp.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, ttl)

    #clientsocket, address = s.accept()

    while True:
        clientsocket, address = s.accept()

        print(f"Conectado com {address}!")
        thread = threading.Thread(target=handle_client, args=(clientsocket, address, s_udp, limit_time))
        thread.start()

        print(f"[CONEXÕES ATIVAS] {threading.active_count() - 1}")
```

Sockets e main() do cliente. A biblioteca struct se faz necessária para receber os pacotes UDP.

```python
import socket
import struct
import json

def main():
    #SOCKET TCP
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(("127.0.0.1", 54000))

    #SOCKET UDP
    MCAST_GRP = '224.1.1.1'
    MCAST_PORT = 55000

    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    sock.bind(('', MCAST_PORT))
    mreq = struct.pack("4sl", socket.inet_aton(MCAST_GRP), socket.INADDR_ANY)
    sock.setsockopt(socket.IPPROTO_IP, socket.IP_ADD_MEMBERSHIP, mreq)
```

```python
    cmd = ""
    connected = True
    while connected :
        line = ''
        if cmd != "LISTEN":
            line = input(">")
            line_slice = line.split(" ")
            cmd = line_slice[0]

        obj = { "operacao" : cmd }

        if cmd == "END":
            connected = False

        elif cmd == "LOGIN":
            obj["cpf"] = line_slice[1]

        elif cmd == "VOTAR":
            obj["num"] = line_slice[1]

        json_string = json.dumps(obj)

        if cmd != "LISTEN":
            s.send(json_string.encode("utf-8"))

        data = b''
        if cmd != "LISTEN":
            data = s.recv(1024).decode("utf-8")
        else:
            data = sock.recv(1024).decode("utf-8")
```

```python
        if "cmd" in data and data["cmd"] == "LISTEN":
            cmd = "LISTEN"
        elif "cmd" in data and data["cmd"] == "BREAK":
            cmd = "BREAK"

        print(data["msg"])

        if cmd == "BREAK":
            for x in data:
                if x != "msg" and x != "cmd":
                    str_aux = "\t " + x + " | " + data[x]
                    print(str_aux)
            break
    s.close()
```

main() do administrador, contendo seus comandos

```python
def main():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(("127.0.0.1", 54000))

    connected = True
    while connected :
        line = input(">")

        line_slice = line.split(" ")

        cmd = line_slice[0]

        obj = { "operacao" : cmd}

        if cmd == "END":
            connected = False

        elif cmd == "LOG-A":
            obj["cpf"]      = line_slice[1]

        elif cmd == "ADD-C":
            obj["nome"]     = line_slice[1]
            obj["partido"]  = line_slice[2]
            obj["numero"]   = line_slice[3]

        elif cmd == "DEL-C":
            obj["numero"]   = line_slice[1]

        elif cmd == "ALERT":
            obj["msg"]      = line[6:]
            print(line[5:])
```

```python
          json_string = json.dumps(obj)
          s.send(json_string.encode("utf-8"))

          data = s.recv(1024).decode("utf-8")
          data = json.loads(data)

          print(data["msg"])

     s.close()
```

main() do timer, consistindo em um loop que envia o tempo atual constantemente.

```python
def main():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(("127.0.0.1", 54000))

    connected = True
    prev = ""
    while connected :
        now = datetime.now()
        tm_now = now.strftime("%H:%M:%S")

        obj = {}
        obj["operacao"] = "TIMER"
        obj["time_now"] = tm_now

        print(obj)

        json_string = json.dumps(obj)
        if prev != tm_now:
            s.send(json_string.encode("utf-8"))
            prev = tm_now

        data = s.recv(1024).decode("utf-8")
        data = json.loads(data)
        print(data)
        if data["msg"] == "BREAK":
            break
        elif data["msg"] == "ok":
            prev = ""
    s.close()
```

Execução dos programas:

1 e 2 -
Cliente adicionando duas pessoas ao vetor, informando a quantidade seguida pelas pessoas em três linhas cada (representando seus nomes, cpf e idade).

```
PS C:\Users\marce\Desktop\SD\SD\SD 1-2> g++ client.cpp -o client -lws2_32
PS C:\Users\marce\Desktop\SD\SD\SD 1-2> ./client
2
Marcos
12345678900
47
Higor
78945612345
22
```

Retorno do servidor e do console (através do std::cout)

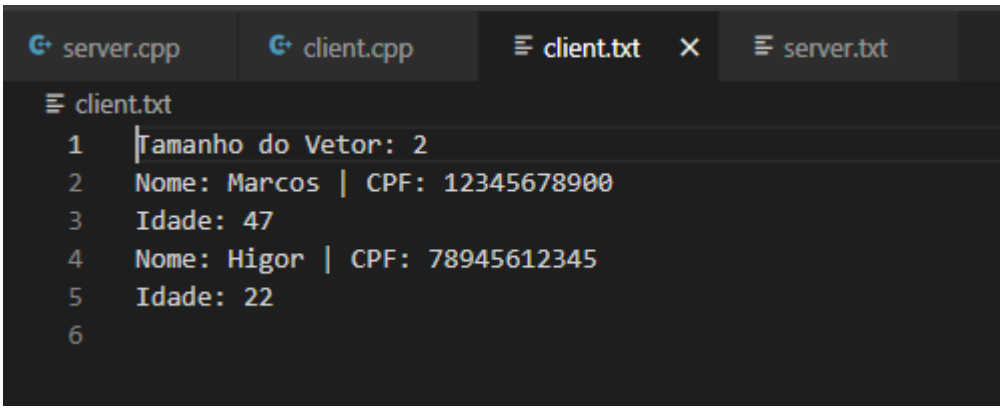
```
---------------------------------
Retorno Server
---------------------------------
SERVER Tamanho do Vetor: 2

SERVER Nome: Marcos | CPF: 12345678900
Idade: 47
SERVER Nome: Higor | CPF: 78945612345
Idade: 22
```

```
---------------------------------
Print
---------------------------------
Tamanho do Vetor: 2
Nome: Marcos | CPF: 12345678900
Idade: 47
Nome: Higor | CPF: 78945612345
Idade: 22
```

O servidor também retorna pelo console usando o std::cout:

```
PS C:\Users\marce\Desktop\SD\SD\SD 1-2> ./server
HOST CONECTADO
CLIENTE DESCONECTOU
Tamanho do Vetor: 2
Nome: Marcos | CPF: 12345678900
Idade: 47
Nome: Higor | CPF: 78945612345
Idade: 22
```

O sistema também apresenta output por um arquivo do cliente e outro do servidor. Ambos guardam o mesmo texto.



3 -

Adição de múltiplos apiários, seguidos por uma consulta geral

```
PS C:\Users\marce\Desktop\SD\SD\SD 3> python client.py
add_apiario 7
Apiario Adicionado!
add_apiario 12
Apiario Adicionado!
add_apiario 3
Apiario Adicionado!
add_apiario 7
Apiario Adicionado!
```

```
consulta all

Apiario 0 com 7 colmeias
Apiario 1 com 12 colmeias
Apiario 2 com 3 colmeias
Apiario 3 com 7 colmeias
```

Adição de Apicultores (com nome, idade e apiário designado) seguidos por uma consulta geral

```
add_apicultor Igor 35 1
Apicultor Adicionado!
add_apicultor Antonio 25 0
Apicultor Adicionado!
add_apicultor Sergio 31 2
Apicultor Adicionado!
consulta all

Apiario 0 com 7 colmeias
        Nome: Antonio | Idade: 25
Apiario 1 com 12 colmeias
        Nome: Igor | Idade: 35
Apiario 2 com 3 colmeias
        Nome: Joao | Idade: 22
        Nome: Sergio | Idade: 31
```

4 -

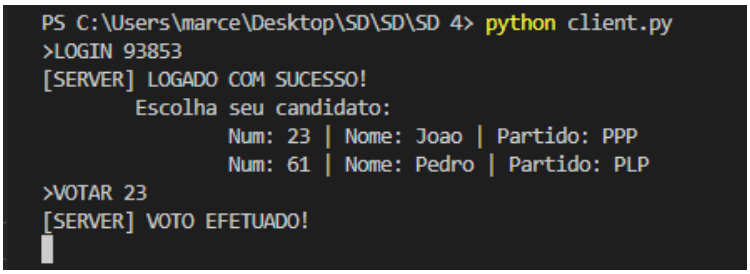
A data limite é inserida no servidor. Para facilitar a medição, o tempo restante (em segundos) e mostrado em seguida.

```
PS C:\Users\marce\Desktop\SD\SD\SD 4> python server.py
2023-11-08 21:05:10.252685
Insira a data limite (Formato HH:MM:SS): 21:07:30
140
```

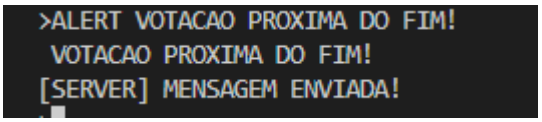
Login do admin e adição de dois candidatos

```
PS C:\Users\marce\Desktop\SD\SD\SD 4> python admin.py
>LOG-A 123-456-789-10
[SERVER] LOGADO COM SUCESSO!
>ADD-C Joao PPP 23
[SERVER] CANDIDATO ADICIONADO!
>ADD-C Pedro PLP 61
[SERVER] CANDIDATO ADICIONADO!
>
```

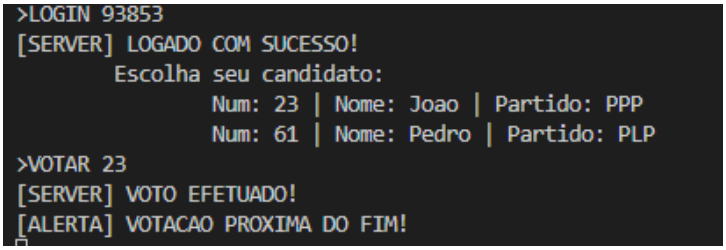
Login de um votante, seguido pela lista de candidatos. No fim, o eleitor vota e fica em modo de espera (comando LISTEN)



O administrador envia uma mensagem informando que a votação está acabando para o server, que repassa para os votantes através de multicast.

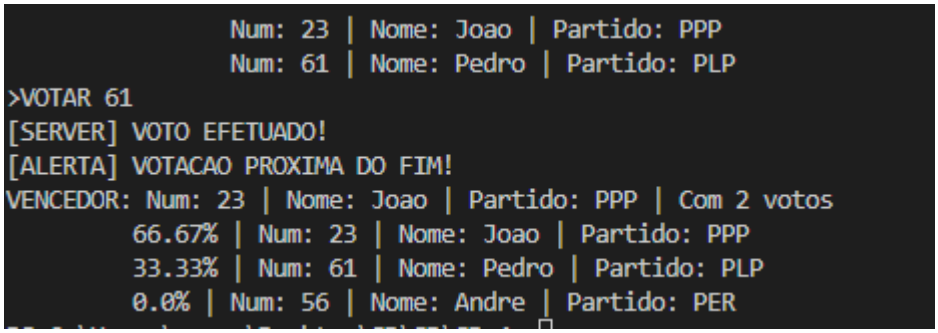


admin

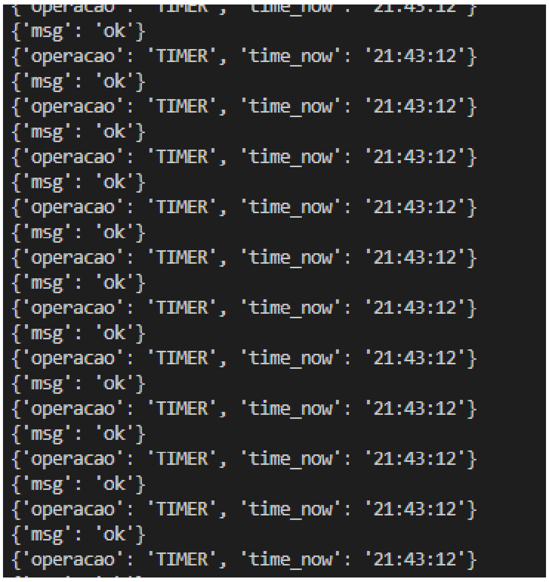


client (votante)

Ao encerrar o tempo, o vencedor é anunciado, seguido pelos percentuais. Note como um terceiro candidato (André) foi adicionado após o votante realizar seu voto.

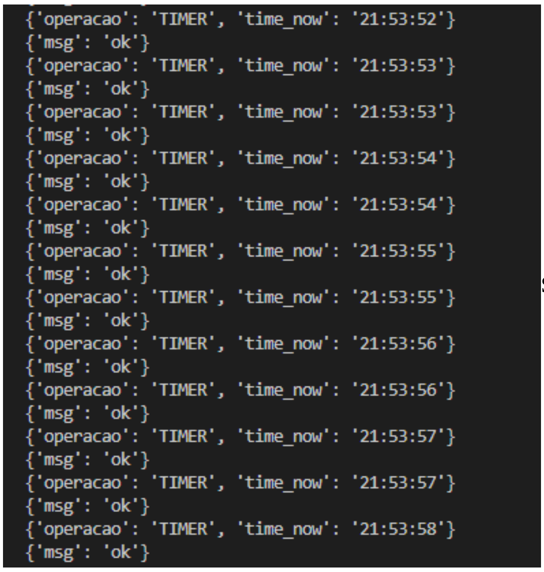


Ao ser executado, o timer envia o tempo ao servidor constantemente. O ritmo em que essas mensagens são enviadas pode ser limitado ao adicionar a biblioteca “time” e uma linha “time.sleep()” de código no loop principal.



```
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:43:12'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:43:12'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:43:12'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:43:12'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:43:12'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:43:12'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:43:12'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:43:12'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:43:12'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:43:12'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:43:12'}
```

Sem time.sleep()



```
{'operacao': 'TIMER', 'time_now': '21:53:52'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:53:53'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:53:53'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:53:54'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:53:54'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:53:55'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:53:55'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:53:56'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:53:56'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:53:57'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:53:57'}
{'msg': 'ok'}
{'operacao': 'TIMER', 'time_now': '21:53:58'}
{'msg': 'ok'}
```

Com time.sleep(0.5), pausando o programa a cada 0,5 segundos