# TRABALHO 1 - SISTEMAS DISTRIBUÍDOS

**Caroline Gomes Pereira da Silva - 554228**
**Davi de Lima Cruz - 474377**
**Laura Cavalcante Campêlo - 539183**

# Repositório

O  código está disponível no GitHub pelo link abaixo:

https://github.com/Davi0Cruz/distribuidos_trabalho_sockets.git

# Ambiente Inteligente

**O ambiente inteligente simula um escritório conectado com dispositivos inteligentes que interagem via sockets e utilizam Protocol Buffers para serialização de mensagens. Foram implementados os seguintes dispositivos:**

- **Ar-Condicionado (atuador): Permite ligar/desligar, ajustar a temperatura (16-30°C), selecionar modos de operação (COOL, HEAT, FAN) e velocidades do ventilador (LOW, MEDIUM, HIGH, AUTO).**

- **Lâmpada Inteligente (atuador): Permite ligar/desligar e ajustar o brilho (0-100%).**

- **Sensor de Temperatura (sensor contínuo): Envia leituras periódicas da temperatura ambiente a cada 2 segundos e simula variações baseadas no estado do ar condicionado.**

# Ambiente Inteligente

- Sensor de Brilho (sensor contínuo): Mede o nível de luminosidade ambiente a cada 2 segundos e é influenciado pelo estado da lâmpada.

- Sensor de Potência (sensor contínuo): Monitora a potência total consumida pelo escritório, somando o consumo do ar-condicionado e da lâmpada, e envia informações a cada 2 segundos.

- Todos os dispositivos foram simulados. Os estados e dados sensoriados são enviados periodicamente ao Gateway.

# Linguagens de Programação e Bibliotecas Utilizadas

O projeto foi implementado inteiramente em Python e utiliza bibliotecas padrão e de terceiros para a comunicação, serialização e simulação de dispositivos.
A seguir, destacamos as principais bibliotecas utilizadas e suas funções no sistema:

- socket: Usada em todos os componentes para implementar a comunicação via sockets TCP e UDP.

  - Criação de conexões TCP no Gateway e dispositivos: socket.socket(socket.AF_INET, socket.SOCK_STREAM)

  - Envio de mensagens UDP multicast para descoberta: sock.setsockopt(socket.IPPROTO_IP, socket.IP_MULTICAST_TTL, 2)

# Linguagens de Programação e Bibliotecas Utilizadas

- struct: Empregada para manipulação de dados binários, essencial para serializar/deserializar os tamanhos de mensagens antes de enviar pelos sockets.
  - Exemplo:

    ```
    struct.pack("4sl", socket.inet_aton(self.MCAST_GRP), socket.INADDR_ANY)
    ```

- threading: Usada para executar tarefas simultâneas, como escutar conexões TCP, multicast UDP e enviar dados sensoriados periodicamente.
  - Função importante:

    ```
    Thread(target=self.listen_for_discovery).start()
    ```

# Linguagens de Programação e Bibliotecas Utilizadas

- time: Controla a periodicidade do envio de dados sensoriados e fornecem timestamps para as mensagens enviadas.

```
sensor_data.timestamp = int(time.time())
```

- json: Serializa e desserializa o estado dos dispositivos e parâmetros em formato JSON.

```
response.attributes["status"] = json.dumps(self.state)
```

# Linguagens de Programação e Bibliotecas Utilizadas

- protobuf: Utilizado para serialização e deserialização das mensagens trocadas entre os componentes. Mensagens como DeviceCommand, DeviceResponse, e SensorData são definidas no arquivo device.proto e compiladas para Python.

  - Exemplo de serialização:
    `data = sensor_data.SerializeToString()`
  - Exemplo de deserialização:
    `command_msg.ParseFromString(data)`

- subprocess: Usado para verificar a atividade de processos simulando o estado de dispositivos conectados (ex.: ar-condicionado ou lâmpada).

```
subprocess.check_output("ps -aux | grep air_conditioner.py", shell=True, text=True)
```

# Linguagens de Programação e Bibliotecas Utilizadas

- tkinter: Permite a criação de interface gráfica desktop para usuários.

```
self.mode_var = tk.StringVar(value="COOL")
```

- ttkbootstrap: Uma extensão de tema para a biblioteca tkinter.

```
tb.Label(fan_frame, text="Fan Speed:").pack(side=LEFT, padx=5)
```

# Envio de Mensagens Multicast (Discovery):

**FUNÇÃO NO GATEWAY: SEND_DISCOVERY_MESSAGE(). ENVIA MENSAGENS MULTICAST UDP PARA DESCOBRIR DISPOSITIVOS DISPONÍVEIS.**

**RESPOSTA NO DISPOSITIVO (EX.: SMARTLAMP): LISTEN_FOR_DISCOVERY(), QUE RESPONDE AO GATEWAY COM INFORMAÇÕES DO DISPOSITIVO VIA MENSAGEM DEVICEDISCOVERY.**

# Formato das Mensagens de Descoberta:

- AS MENSAGENS DE DESCOBERTA ENVIADAS PELO GATEWAY SÃO DO TIPO DEVICECOMMAND COM O CAMPO COMMAND DEFINIDO COMO "GATEWAY_DISCOVERY".

- OS DISPOSITIVOS RESPONDEM COM MENSAGENS DO TIPO DEVICEDISCOVERY, CONTENDO OS CAMPOS: DEVICE_TYPE (TIPO DO DISPOSITIVO), IP (ENDEREÇO IP), PORT (PORTA TCP) E STATUS (ESTADO INICIAL DO DISPOSITIVO EM JSON).

# Código (arquivo device.proto):

```
message DeviceCommand {
    string command = 1;
    string parameters = 2;
}
```

USADA PELO GATEWAY PARA ENVIAR A MENSAGEM DE DESCOBERTA PARA OS DISPOSITIVOS.

# Código (arquivo device.proto):

```
message DeviceDiscovery {
    string device_type = 1;
    string ip = 2;
    int32 port = 3;
    string status = 4;
}
```

USADA PELOS DISPOSITIVOS PARA INFORMAR SEU TIPO, IP E ESTADO AO GATEWAY APÓS DESCOBERTA.

# Gerenciamento de Comandos no Gateway e Dispositivos:

- **O GATEWAY GERENCIA COMANDOS RECEBIDOS DO CLIENTE POR MEIO DA FUNÇÃO *HANDLE_CLIENT_REQUEST()*, QUE PROCESSA REQUISIÇÕES COMO "LIST_DEVICES", "CONTROL_DEVICE" E "GET_STATUS".**
- **CADA DISPOSITIVO POSSUI UMA FUNÇÃO *HANDLE_COMMAND()* PARA PROCESSAR COMANDOS RECEBIDOS DO GATEWAY. POR EXEMPLO:**

**No ar-condicionado, o comando "SET_TEMPERATURE" ajusta a temperatura alvo e atualiza o arquivo de estado associado.**

# Formato das Mensagens para Gerenciamento de Atuadores:

- **DEVICECOMMAND (PARA O GATEWAY ENVIAR COMANDOS)**
- **DEVICERESPONSE (PARA OS DISPOSITIVOS RETORNAREM RESPOSTAS)**

- DeviceCommand:
  - command: "ON", "OFF", "SET_TEMPERATURE", etc.
  - parameters: JSON com parâmetros adicionais, como {"temperature": 22}.

- DeviceResponse:
  - success: Indica sucesso ou falha do comando.
  - message: Mensagem de resposta descritiva.
  - status: Estado atualizado do dispositivo em JSON.

# Formato das Mensagens para Gerenciamento de Atuadores:

- **DEVICECOMMAND (PARA O GATEWAY ENVIAR COMANDOS)**
- **DEVICERESPONSE (PARA OS DISPOSITIVOS RETORNAREM RESPOSTAS)**

```
message DeviceCommand {
    string command = 1;
    string parameters = 2;
}
```

```
message DeviceResponse {
    bool success = 1;
    string message = 2;
    string status = 3;
    map<string, string> attributes = 4;
}
```

# Formato das Mensagens para Gerenciamento de Atuadores:

- **ENVIO DE COMANDO PELO GATEWAY - GATEWAY.PY:**

```
command_msg = device_pb2.DeviceCommand()
command_msg.command = "SET_TEMPERATURE"
command_msg.parameters = json.dumps({"temperature": 22})
sock.send(command_msg.SerializeToString())
```

# Formato das Mensagens para Gerenciamento de Atuadores:

- **PROCESSAMENTO DE COMANDO NO DISPOSITIVO (AIR_CONDITIONER.PY):**

```
if command == "SET_TEMPERATURE":
    if "temperature" in params:
        temp = int(params["temperature"])
        if 16 <= temp <= 30:
            self.state["temperature"] = temp
            response.success = True
            response.message = f"Temperature set to {temp}°C"
        else:
            response.success = False
            response.message = "Temperature must be between 16 and 30°C"
```

# Simulação de Sensores:

- **FUNÇÃO NO SENSOR DE TEMPERATURA: SIMULATE_ENVIRONMENT_TEMPERATURE(), QUE AJUSTA A TEMPERATURA AMBIENTE COM BASE NO ESTADO DO AR-CONDICIONADO.**

- **FUNÇÃO NO SENSOR DE BRILHO: SIMULATE_BRIGHTNESS(), QUE AJUSTA A LUMINOSIDADE CONFORME O BRILHO DA LÂMPADA.**

- **ENVIO PERIÓDICO DE DADOS SENSORIADOS EM TODOS OS SENSORES, EX:**

**periodically_send_state() no Sensor de Potência, que envia mensagens SensorData via UDP ao Gateway.**

# Formato das Mensagens para Recebimento de Informações Periódicas:

- **OS SENSORES ENVIAM MENSAGENS DO TIPO SENSORDATA AO GATEWAY, COM OS CAMPOS:**

  - device_id: Identificador do sensor.
  - sensor_type: Tipo de leitura (ex.: "temperature").
  - value: Valor sensoriado (ex.: 23.5 para temperatura).
  - unit: Unidade da leitura (ex.: "°C").
  - timestamp: Marcação de tempo.

# Formato das Mensagens para Recebimento de Informações Periódicas:

- **OS SENSORES ENVIAM MENSAGENS DO TIPO SENSORDATA AO GATEWAY, COM OS CAMPOS:**

```
message SensorData {
    string device_id = 1;
    string sensor_type = 2;
    double value = 3;
    string unit = 4;
    int64 timestamp = 5;
}
```

# Formato das Mensagens para Recebimento de Informações Periódicas:

- **ENVIO PERIÓDICO PELO SENSOR DE TEMPERATURA - TEMPERATURE_SENSOR.PY:**

```
sensor_data = device_pb2.SensorData()
sensor_data.device_id = f"{self.device_type}_{self.get_local_ip()}_{self.TCP_PORT}"
sensor_data.sensor_type = "temperature"
sensor_data.value = self.state["temperature"]
sensor_data.unit = self.state["unit"]
sensor_data.timestamp = int(time.time())
self.udp_socket.sendto(sensor_data.SerializeToString(), (self.gateway_ip, 50002))
```

```
data, addr = self.sensor_socket.recvfrom(2048)
sensor_data = device_pb2.SensorData()
sensor_data.ParseFromString(data)

device_id = sensor_data.device_id
if device_id not in self.devices:
    self.devices[device_id] = {
        'id': device_id,
        'type': sensor_data.sensor_type,
        'ip': addr[0],
        'port': 0,
        'status': "{}",
        'last_seen': time.time()
    }

device = self.devices[device_id]
device['last_seen'] = time.time()
```

**RECEBIMENTO PERIÓDICO DE DADOS PELO GATEWAY**

# Formato das Mensagens entre Cliente e Gateway:

- **O CLIENTE ENVIA MENSAGENS DO TIPO CLIENTREQUEST, ENQUANTO O GATEWAY RESPONDE COM CLIENTRESPONSE.**

- **ClientRequest:**
  - **command: "LIST_DEVICES", "CONTROL_DEVICE", etc.**
  - **device_id: Identificador do dispositivo alvo.**
  - **parameters: JSON com parâmetros de configuração.**
- **ClientResponse:**
  - **devices: Lista de dispositivos disponíveis.**
  - **success: Indica sucesso ou falha.**
  - **message: Descrição da resposta.**

# Formato das Mensagens entre Cliente e Gateway:

- **O CLIENTE ENVIA MENSAGENS DO TIPO CLIENTREQUEST, ENQUANTO O GATEWAY RESPONDE COM CLIENTRESPONSE.**

```
message ClientRequest {
    string command = 1;
    string device_id = 2;
    string action = 3;
    string parameters = 4;
}
```

```
message ClientResponse {
    bool success = 1;
    string message = 2;
    repeated DeviceInfo devices = 3;
}
```

# Formato das Mensagens entre Cliente e Gateway:

- **REQUISIÇÃO DO CLIENTE AO GATEWAY - CLIENT.PY**

```
request = device_pb2.ClientRequest()
request.command = "CONTROL_DEVICE"
request.device_id = "air_conditioner_192.168.0.10_5001"
request.action = "SET_TEMPERATURE"
request.parameters = json.dumps({"temperature": 22})
```

# Formato das Mensagens entre Cliente e Gateway:

- **RESPOSTA DO GATEWAY AO CLIENTE - GATEWAY.PY:**

```
response = device_pb2.ClientResponse()
response.success = True
response.message = "Device controlled successfully"
response_data = response.SerializeToString()
client_socket.send(len(response_data).to_bytes(4, byteorder='big'))
client_socket.send(response_data)
```

# Interface de Usuário do Cliente:

- **VISUALIZAR OS DISPOSITIVOS CONECTADOS COM SEUS ESTADOS;**
- **ENVIAR COMANDOS PARA DISPOSITIVOS ESPECÍFICOS POR MEIO DE BOTÕES E MENUS SUSPENSOS;**
- **CONFIGURAR DISPOSITIVOS DETALHADAMENTE (EX.: AJUSTAR TEMPERATURA DO AR-CONDICIONADO OU BRILHO DA LÂMPADA).**

```
def on_list_devices(self):
    response = self.client.list_devices()
    for device in response.devices:
        self.device_tree.insert("", "end", values=(device.device_id, device.device_type))
```

**CLIENTE GUI:**
**INTERFACE GRÁFICA BASEADA EM BOTÕES E TABELAS.**

# Interface Gráfica no Cliente (client_gui.py):

- **IMPLEMENTADA COM AS BIBLIOTECAS TKINTER E SUA EXTENSÃO TTKBOOTSTRAP, APRESENTA UMA LISTA DE DISPOSITIVOS E PERMITE CONTROLAR CADA UM DE FORMA INTUITIVA.**

- **FUNÇÕES IMPORTANTES INCLUEM:**

  - **on_list_devices(): Atualiza a lista de dispositivos conectados ao Gateway.**

  - **on_device_config(): Abre um pop-up para exibição de status e envio de comandos ao dispositivo selecionado.**