General Approach

You can think of your "devices" as **software peers** (Python scripts, Node.js apps, etc.) running on your computer or across multiple computers on the same network. Each peer can represent:

- A **sensor** (e.g., generates temperature or motion data)
- An **actuator** (e.g., reacts to a message by printing "Fan ON" or "Light OFF")
- Optionally, a **controller** (e.g., aggregates data and decides actions)

For example:

A smart-home temperature system with two peers:
• Peer A (sensor) → sends temperature readings.
• Peer B (actuator) → turns a "virtual fan" on/off based on thresholds.

You'll simulate all device interactions over sockets or WebSockets, not real sensors.

Submission 1 – Design & Basic Connectivity

Goal: Demonstrate discovery and simple communication between peers.

What to do:

1. **Write a short proposal**
    - Scenario (e.g., smart home, industrial monitoring)
    - Device types (sensor, actuator)
    - Data format (JSON messages like `{"type":"temperature","value":24.5}`)
    - Simple discovery mechanism (broadcast or multicast message like "HELLO")
    - Functional requirements (connectivity, data sharing)
2. **Implement basic peers**
    - Use **Python sockets** or **asyncio**, or Node.js networking.
    - Each peer:
        - Broadcasts its presence on the network.
        - Listens for other peers' "HELLO" messages.
        - Sends/receives structured JSON messages.
3. **Test report**
    - Screenshots or logs showing:
        - Peer A discovers Peer B.
        - Peers exchange a "HELLO" JSON message.

✓ **No real sensors needed.** Just simulate readings with random numbers.

Submission 2 – Secure Communication & Discovery

Goal: Add encryption, authentication, and robust discovery.

What to do:

1. **Secure communication:**
   - Use **TLS (SSL)** sockets in Python or `ssl` in Node.js, or implement **AES encryption** with a shared key.
2. **Authentication:**
   - Use pre-shared keys or simple password verification in the handshake message.
3. **Improved discovery:**
   - Handle joining/leaving events gracefully.
   - Handle network errors with try/except or reconnection logic.
4. **Test report:**
   - Show that encrypted messages are exchanged successfully.
   - Include logs demonstrating peer authentication and error recovery.

✅ **Still no physical devices needed.**

Submission 3 – State Management & Data Persistence

Goal: Add logic, state, and persistence.

What to do:

1. **Add application logic:**
   - Example: if temperature > 25°C, actuator peer turns on fan (prints message or sets flag).
   - At least two states: "idle" / "active."
2. **Data persistence:**
   - Save readings or state in a local JSON or SQLite file.
   - Reload previous state on restart.
3. **Handle multiple peers:**
   - Allow several sensors to join and synchronize shared state.
   - Use simple coordination (e.g., broadcast "STATE_UPDATE" messages).
4. **Test report:**
   - Logs showing peers maintaining state consistency after restart.

✅ **Simulated readings are fine.**

Submission 4 – Final Application & Advanced Features

Goal: Polish the app and add advanced networking/security features.

What to do:

1. **User-friendly interface or API:**
   - Simple CLI menu, Flask REST API, or lightweight GUI (e.g., Tkinter or HTML page).
2. **Advanced networking:**
   - Add heartbeat messages for peer health checks, or asynchronous messaging using Python `asyncio` / WebSocket.
3. **Advanced security:**
   - Implement replay-attack prevention (timestamps, nonces), or role-based access control (sensor vs actuator roles).
4. **Final report:**
   - Include full code, logs, and checklist.
   - Prepare to demo functionality in the viva.

✅ **Everything can be demonstrated locally.** No real sensors or IoT boards needed.

Deliverables (Each Submission)

Your ZIP file should contain:

1. **Short report (≤500 words)** – design summary, how to run code, testing evidence.
2. **Source code** – all peers, configs, dependencies (e.g., `requirements.txt`).
3. **Implementation log** – what you did each week, test results.
4. **Feature checklist** – tick what's done.

Tip

You can simulate "sensor readings" using Python's `random` module:

```
import random, json, time

def get_temperature():
    return round(random.uniform(18.0, 30.0), 1)

while True:
    reading = {"type": "temperature", "value": get_temperature()}
    print(json.dumps(reading))
    time.sleep(2)
```

Then send these readings over your peer-to-peer socket connection.

1. Create Your Own Dataset

✅ **This is the most common and acceptable option** for student IoT projects — especially when you don't have physical sensors.

You can **generate or simulate realistic data** that represents what sensors would normally collect, such as:

Scenario	Simulated Data Examples
Smart Home	Temperature, humidity, motion detection, power usage
Environmental Monitoring	Air quality (CO_2, PM2.5), temperature, humidity
Healthcare	Heart rate, oxygen level, body temperature
Industrial	Machine vibration, motor speed, energy consumption

How to create it:

- Use random generators with realistic ranges
- `import random`
- `temp = round(random.uniform(20, 30), 2)`
- `humidity = round(random.uniform(40, 60), 1)`
- Or create a CSV/JSON file with "recorded" (simulated) data:
- `[`
- `  {"timestamp": "2025-10-24T10:00:00", "temperature": 22.5, "humidity": 55.2},`
- `  {"timestamp": "2025-10-24T10:05:00", "temperature": 22.8, "humidity": 54.8}`
- `]`
- You can also simulate **changing conditions** (e.g., rising temperature or periodic motion events).

This synthetic dataset will act as your "sensor readings."

2. You Can Also Use a Public Dataset (Optional)

If you want more realism or larger samples, you can use an **open IoT dataset**. Some examples:

Source	Description
Kaggle IoT Datasets	Many IoT & sensor datasets (smart homes, air quality, industrial)

Source	Description
UCI Machine Learning Repository	Has IoT-related datasets like “Air Quality Data Set”
OpenWeatherMap API	Live weather data you can fetch as JSON
CityPulse IoT Dataset	Real smart city sensor data (air quality, noise, traffic)

You can download a CSV or JSON and use it to simulate “live” updates in your system.

Example:
Your peer could read one row every few seconds from the dataset and treat it as if it just came from a sensor.

3. How to Integrate the Dataset Into Your P2P App

- Each “sensor peer” can read or generate values from the dataset.
- Peers exchange data using JSON messages.
- “Actuator peers” decide actions based on thresholds from that data.
- For later submissions (with persistence), you can **store readings locally** (e.g., in `data.json` or SQLite).

In Your Report

For each submission, mention:

- What dataset you used (synthetic or public)
- The type of data (e.g., temperature readings, humidity)
- How it’s generated or accessed
- Why it’s appropriate for your chosen IoT scenario

That’s enough to meet the dataset expectation.

Short Proposal (≈300 words)

Scenario – Smart Home Temperature Network

This project simulates a small peer-to-peer (P2P) IoT environment for a **smart-home climate control system**.
Each peer represents either a **temperature sensor** or a **fan actuator**.
Peers automatically discover one another over the local network and exchange simple JSON messages to share readings and status information.

Device Types

- **Sensor Peer** – periodically generates simulated temperature readings (20 – 30 °C).
- **Actuator Peer** – receives readings and prints a simulated action, e.g., *"Fan ON"* if temperature > 25 °C.

Data Format

All messages are JSON:

```
{
  "type": "temperature",
  "device": "sensor1",
  "value": 24.8
}
```

Control or discovery messages:

```
{"type": "hello", "device": "sensor1"}
```

Discovery Mechanism

Peers use **UDP broadcast** on port 50000 to announce their presence.
Each peer:

1. Broadcasts a "HELLO" JSON packet every 5 seconds.
2. Listens for HELLO packets from others and stores discovered peer addresses.

Functional Requirements

- Peers must **connect automatically** without a central server.
- **Structured JSON** used for all messages.
- **Bidirectional communication** between at least two peers.
- Output logs must demonstrate successful peer discovery and message exchange.

---

2. Implementation – Basic Peers (Python)

❗ Run at least **two terminals** on the same network/machine.
Each terminal runs the same file with a different device name argument:

```
python peer.py sensor1
python peer.py actuator1
```

File: `peer.py`

```
import socket
```

```
import json
import threading
import time
import random
import sys

BROADCAST_PORT = 50000
BROADCAST_ADDR = '<broadcast>'
INTERVAL = 5

def make_message(msg_type, device, value=None):
    data = {"type": msg_type, "device": device}
    if value is not None:
        data["value"] = value
    return json.dumps(data).encode('utf-8')

def broadcast_presence(device):
    """Periodically send HELLO and temperature messages."""
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
    while True:
        # 1. Send HELLO
        sock.sendto(make_message("hello", device), (BROADCAST_ADDR,
BROADCAST_PORT))
        # 2. Optionally send sensor reading
        temp = round(random.uniform(20, 30), 1)
        sock.sendto(make_message("temperature", device, temp),
(BROADCAST_ADDR, BROADCAST_PORT))
        print(f"[{device}] Broadcasted HELLO & temp {temp}°C")
        time.sleep(INTERVAL)

def listen_for_peers(device):
    """Listen for messages from others."""
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('', BROADCAST_PORT))

    known_peers = set()
    while True:
        data, addr = sock.recvfrom(1024)
        try:
            msg = json.loads(data.decode('utf-8'))
            sender = msg.get("device")
            if sender != device:
                if sender not in known_peers:
                    known_peers.add(sender)
                    print(f"[{device}] Discovered new peer: {sender}
@ {addr}")
                if msg["type"] == "temperature":
                    value = msg["value"]
                    print(f"[{device}] Received temperature from
{sender}: {value}°C")
                    # Simple actuator logic (for demonstration)
                    if "actuator" in device.lower():
                        if value > 25:
                            print(f"[{device}] Action: Fan ON")
                        else:
                            print(f"[{device}] Action: Fan OFF")
        except json.JSONDecodeError:
            pass
```

```
def main():
    if len(sys.argv) < 2:
        print("Usage: python peer.py <device_name>")
        sys.exit(1)

    device = sys.argv[1]
    listener = threading.Thread(target=listen_for_peers,
args=(device,), daemon=True)
    listener.start()
    broadcast_presence(device)

if __name__ == "__main__":
    main()
```

---

3. Example Test Report / Logs

Test Setup

- Two peers launched on the same Wi-Fi network:
  - `sensor1` (simulated temperature sensor)
  - `actuator1` (fan controller)
- Each peer broadcasts every 5 seconds.

Observed Console Output

`sensor1` terminal

```
[sensor1] Broadcasted HELLO & temp 24.5°C
[sensor1] Discovered new peer: actuator1 @ ('192.168.1.23', 50000)
[sensor1] Broadcasted HELLO & temp 25.8°C
```

`actuator1` terminal

```
[actuator1] Discovered new peer: sensor1 @ ('192.168.1.22', 50000)
[actuator1] Received temperature from sensor1: 24.5°C
[actuator1] Action: Fan OFF
[actuator1] Received temperature from sensor1: 25.8°C
[actuator1] Action: Fan ON
```

Result

✓ Peers discovered each other successfully.
✓ Structured JSON messages exchanged correctly.
✓ Actuator responded to sensor data.
✓ No physical hardware required.

Files to Include in ZIP

```
/Submission1/
```

```
│
├── peer.py
├── report.pdf or report.docx   # includes proposal + logs
├── implementation_log.txt
└── feature_checklist.txt
```

Step-by-Step Plan for Submission 1

📂 1. Folder Structure

Create a main project folder (for example):

```
IoT_P2P_Project/
│
├── peer.py                     # main Python code
├── dataset.json                # optional mock data file
│
├── report.docx                 # your written design + screenshots
├── implementation_log.txt      # weekly or step log
├── feature_checklist.txt       # checklist for submission
└── README.txt                  # how to run + dependencies
```

When finished, you'll ZIP this whole folder for submission.

---

2. File Contents and Purpose

□ (A) peer.py – Your main runnable program

This is the **code that simulates peers** and handles discovery + communication.
You already have this code (from the previous answer).
→ Save that exact code as `peer.py`.

📓 (B) report.docx – Design & Analysis Report (max 500 words)

Structure:

Title:

Submission 1 – Design & Basic Connectivity – Smart Home IoT P2P System

Sections:

1. **Scenario:** Explain your system (smart home, sensor/actuator roles).
2. **Device Types:** Describe sensor vs actuator functions.
3. **Data Format:** Show JSON examples.
4. **Discovery Mechanism:** Describe UDP broadcast and message flow.
5. **Functional Requirements:**
    - P2P connectivity
    - JSON messages
    - Data sharing
    - Simple communication test
6. **Design Diagram (optional):**
    - You can insert a simple box diagram showing two peers connected via Wi-Fi.
7. **Test Results:** Paste short logs or screenshots from running the peers.
8. **Analysis:** 3–5 sentences summarizing what worked and what could be improved for Submission 2.

Save it as `report.docx` or `report.pdf`.

(C) implementation_log.txt

Tracks your development and testing progress.
Example content:

```
Date: 2025-10-05
- Created base peer structure and socket communication.

Date: 2025-10-08
- Implemented JSON message formatting and broadcast discovery.

Date: 2025-10-10
- Added temperature simulation and actuator reaction logic.
- Tested between two terminals on localhost. Success.

Date: 2025-10-12
- Prepared report and screenshots for submission.
```

✅ (D) feature_checklist.txt

Lists the required features and marks completion:

```
Submission 1 – Design & Basic Connectivity

[✓] Scenario description included in report
[✓] Two device types (sensor, actuator)
[✓] Structured JSON messages
[✓] Peer discovery via UDP broadcast
[✓] Peers exchange "HELLO" messages
[✓] Test logs/screenshots included
[ ] Secure communication (for Submission 2)
```

```
[ ] Authentication
```

□ (E) dataset.json *(optional file)*

If you'd like to show mock data instead of random values:

```
[
  {"timestamp": "2025-10-24T10:00:00", "temperature": 22.5},
  {"timestamp": "2025-10-24T10:05:00", "temperature": 25.8},
  {"timestamp": "2025-10-24T10:10:00", "temperature": 27.1}
]
```

You can modify `peer.py` to read from this file instead of generating random numbers, if you want.

□ (F) README.txt

Explain how to run your code:

```
IoT_P2P_Project – Submission 1 (Design & Basic Connectivity)

Requirements:
- Python 3.10 or later
- Works on Windows, macOS, or Linux
- No additional libraries needed (uses built-in socket, json,
threading)

How to Run:
1. Open two terminals (or two Python consoles).
2. In the first terminal:
      python peer.py sensor1
3. In the second terminal:
      python peer.py actuator1
4. Wait a few seconds. Each peer will discover the other and exchange
messages.
5. Check console output for HELLO and temperature messages.

Expected Result:
- Peers discover each other (displayed as "Discovered new peer")
- Actuator prints "Fan ON" or "Fan OFF" depending on temperature
value.
```

---

3. How to Run the Project

□ Option 1 – Run in VS Code

1. Open VS Code → **File > Open Folder…** → select `IoT_P2P_Project`.
2. Open `peer.py`.

3. Open a new terminal inside VS Code (**Ctrl+`**).
4. Run first peer:
5. `python peer.py sensor1`
6. Open another terminal tab (click + in the terminal pane).
7. Run second peer:
8. `python peer.py actuator1`
9. Watch both terminals for discovery logs.

□ Option 2 – Run in PyCharm

1. Open PyCharm → **Open** > **IoT_P2P_Project**.
2. Right-click `peer.py` → "Run 'peer'".
3. To start a second peer:
   - Go to the top toolbar → dropdown → "Edit Configurations…"
   - Click "+" → "Python"
   - Set *Script path* = `peer.py`, *Parameters* = `actuator1`
   - Name it "Actuator" → Apply → OK.
   - Run both configurations ("Sensor" and "Actuator") simultaneously.

✅ Expected Console Output

sensor1 terminal

```
[sensor1] Broadcasted HELLO & temp 24.5°C
[sensor1] Discovered new peer: actuator1 @ ('192.168.1.23', 50000)
```

actuator1 terminal

```
[actuator1] Discovered new peer: sensor1 @ ('192.168.1.22', 50000)
[actuator1] Received temperature from sensor1: 24.5°C
[actuator1] Action: Fan OFF
```

🎁 4. Final ZIP for Submission

When you've verified everything works:

```
IoT_P2P_Project.zip
│
├── peer.py
├── dataset.json
├── report.docx
├── implementation_log.txt
├── feature_checklist.txt
└── README.txt
```

Upload **only this single ZIP file** to Blackboard.

What students must know (prerequisites)

- Basic Python (variables, functions, modules, `if __name__ == "__main__"`).
- Basic command line (how to run `python file.py arg`).
- Familiarity with JSON at a conceptual level (key/value pairs).
- Basic networking concepts: IP, port, UDP vs TCP.
- How to open multiple terminals or run multiple IDE run configurations.

Core concepts explained

1) JSON (JavaScript Object Notation)

- **What it is:** Text format for structured data (human-readable). Examples:
- `{"type":"temperature","device":"sensor1","value":24.5}`
- **Why use it:** Easy to serialize (convert objects to text) and parse across languages and over network.
- **Python usage:** `json.dumps(obj)` → string; `json.loads(string)` → Python dict.
- **In our code:** we `json.dumps()` a dict, encode it (`.encode('utf-8')`) and send bytes over socket; on receive we `.decode('utf-8')` then `json.loads(...)`.

---

2) Sockets (network I/O)

- **Socket:** an endpoint for network communication. Created with `socket.socket(...)`.
- **Important functions & options:**
  - `sock.bind((host, port))` — listen on a port.
  - `sock.sendto(data, (addr,port))` — send UDP datagram.
  - `sock.recvfrom(bufsize)` — receive UDP datagram (returns data, addr).
  - `sock.connect((addr,port))`, `sock.send`, `sock.recv` — for TCP.
  - `sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)` — enable UDP broadcast.
  - `sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)` — allow rebinding address.
- **UDP vs TCP**
  - **UDP:** connectionless, packet-based, good for discovery (broadcast/multicast). Unreliable (no guarantee of delivery).
  - **TCP:** connection-oriented, reliable stream — good for direct peer-to-peer message exchange needing reliability.

---

3) Threading & Concurrency

- **Why:** Need to listen and send concurrently. If you block on `recvfrom()` you can't also broadcast.
- **Python approach:** `threading.Thread(target=..., daemon=True).start()` to run listener while main thread broadcasts or does other work.
- **Alternative:** `asyncio` (asynchronous I/O) — more scalable but slightly more advanced.

Class design for a simple P2P Peer (recommended structure)

Design the peer with small, single-responsibility classes to keep code readable and testable.

```
Peer
 ├─ Broadcaster
 ├─ Listener
 └─ Message (or utility functions for message encoding/decoding)
```

Responsibilities

- **Peer**: high-level; composes Broadcaster and Listener, holds device name and peer list, implements application logic (actuator/sensor behavior).
- **Broadcaster**: periodically sends HELLO and sensor readings via UDP broadcast.
- **Listener**: receives incoming UDP messages, parses JSON, informs Peer.
- **Message utils**: encode/decode JSON, standardize message types.

Class-based peer example (detailed, commented)

Save as `peer_class.py`. This is an improved, class-based rewrite of the earlier script.

```
# peer_class.py
import socket
import json
import threading
import time
import random
import sys

BROADCAST_PORT = 50000
BROADCAST_ADDR = '<broadcast>'
BROADCAST_INTERVAL = 5  # seconds

def encode_msg(mtype, device, value=None):
    """Return JSON bytes for a message."""
    payload = {"type": mtype, "device": device}
```

```
    if value is not None:
        payload["value"] = value
    return json.dumps(payload).encode('utf-8')

def decode_msg(data_bytes):
    """Return Python dict from JSON bytes (or None on error)."""
    try:
        return json.loads(data_bytes.decode('utf-8'))
    except Exception:
        return None

class Broadcaster:
    """Sends periodic broadcast messages (HELLO + sensor
readings)."""
    def __init__(self, device, port=BROADCAST_PORT,
interval=BROADCAST_INTERVAL):
        self.device = device
        self.port = port
        self.interval = interval
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST,
1)

    def start(self):
        t = threading.Thread(target=self._run, daemon=True)
        t.start()

    def _run(self):
        while True:
            # HELLO
            self.sock.sendto(encode_msg("hello", self.device),
(BROADCAST_ADDR, self.port))
            # Temperature reading
            temp = round(random.uniform(20, 30), 1)
            self.sock.sendto(encode_msg("temperature", self.device,
temp), (BROADCAST_ADDR, self.port))
            print(f"[{self.device}] Broadcast HELLO & temp {temp}°C")
            time.sleep(self.interval)

class Listener:
    """Listens for broadcast messages and calls a handler."""
    def __init__(self, device, handler, port=BROADCAST_PORT):
        self.device = device
        self.port = port
        self.handler = handler  # function to call with (msg_dict,
addr)
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR,
1)
        self.sock.bind(('', self.port))

    def start(self):
        t = threading.Thread(target=self._run, daemon=True)
        t.start()

    def _run(self):
        while True:
            data, addr = self.sock.recvfrom(4096)
            msg = decode_msg(data)
            if msg:
                self.handler(msg, addr)
```

```
class Peer:
    """High-level peer: known_peers, application logic
(sensor/actuator)."""
    def __init__(self, device_name):
        self.device = device_name
        self.known_peers = set()
        self.broadcaster = Broadcaster(self.device)
        self.listener = Listener(self.device, self.handle_message)

    def start(self):
        self.listener.start()
        self.broadcaster.start()
        # keep main thread alive
        try:
            while True:
                time.sleep(1)
        except KeyboardInterrupt:
            print("Exiting...")

    def handle_message(self, msg, addr):
        sender = msg.get("device")
        if sender is None or sender == self.device:
            return
        if sender not in self.known_peers:
            self.known_peers.add(sender)
            print(f"[{self.device}] Discovered new peer: {sender} @
{addr}")

        mtype = msg.get("type")
        if mtype == "temperature":
            val = msg.get("value")
            print(f"[{self.device}] Received temperature from
{sender}: {val}°C")
            if "actuator" in self.device.lower():
                # Simple actuator rule:
                if val > 25:
                    print(f"[{self.device}] Action: Fan ON")
                else:
                    print(f"[{self.device}] Action: Fan OFF")

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python peer_class.py <device_name>")
        sys.exit(1)
    peer = Peer(sys.argv[1])
    peer.start()
```

Explanation of the code

- `encode_msg` / `decode_msg`: message serialization helpers (JSON ↔ Python). Always centralize this so message format is consistent.
- `Broadcaster`: opens a UDP socket with `SO_BROADCAST` enabled and periodically `sendto()` JSON bytes to `<broadcast>` on `BROADCAST_PORT`.
- `Listener`: binds to `('', port)` to receive broadcast datagrams. Uses `recvfrom()` and calls `handler(msg, addr)` when message arrives.

- `Peer`: composes the above, keeps a `known_peers` set, and contains `handle_message()` with simple logic. This is where students will add application logic later.

Simple TCP client-server example (class-based)

This example demonstrates reliable, connection-based communication (useful for the later stage where you move from discovery to secure P2P connections).

Two files: `tcp_server.py` and `tcp_client.py`.

tcp_server.py

```
# tcp_server.py
import socket
import threading

HOST = '0.0.0.0'
PORT = 50010

def handle_client(conn, addr):
    print(f"Connection from {addr}")
    with conn:
        while True:
            data = conn.recv(1024)
            if not data:
                break
            print(f"Received from {addr}: {data.decode('utf-8')}")
            # echo back
            conn.sendall(b"ACK: " + data)

def start_server():
    srv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    srv.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    srv.bind((HOST, PORT))
    srv.listen()
    print(f"Server listening on {HOST}:{PORT}")
    while True:
        conn, addr = srv.accept()
        thread = threading.Thread(target=handle_client, args=(conn,
addr), daemon=True)
        thread.start()

if __name__ == "__main__":
    start_server()
```

tcp_client.py

```
# tcp_client.py
import socket
import sys

HOST = '127.0.0.1'  # server address
PORT = 50010
```

```
def run_client(name):
    with socket.create_connection((HOST, PORT)) as sock:
        for i in range(3):
            msg = f"{name} says hello {i}"
            print("Sending:", msg)
            sock.sendall(msg.encode('utf-8'))
            data = sock.recv(1024)
            print("Received:", data.decode('utf-8'))

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python tcp_client.py <client_name>")
        sys.exit(1)
    run_client(sys.argv[1])
```

How it works

- Server binds and `listen()`s. `accept()` returns a new socket for each client; `handle_client` runs in a thread to serve that client.
- Client `create_connection(...)`, sends messages, waits for echo.

Practical running instructions (step-by-step)

Files to create (exact)

- `peer_class.py` (class-based UDP broadcast listener/broadcaster)
- `peer.py` (optionally the earlier script — choose one)
- `tcp_server.py`
- `tcp_client.py`
- `README.txt` (how to run)
- `report.docx` / `report.pdf`
- `implementation_log.txt`
- `feature_checklist.txt`
- (optional) `dataset.json` for reading pre-recorded values

How to run the P2P UDP peer

1. Open **two terminals** (or use two run configurations in your IDE).
2. In terminal 1:
3. `python peer_class.py sensor1`
4. In terminal 2:
5. `python peer_class.py actuator1`
6. Wait ~5 seconds. You will see:
    - Each side broadcasting HELLO + temperature.
    - Each side discovering the other.
    - Actuator printing “Fan ON” or “Fan OFF” based on readings.

How to run TCP example (for demonstration)

1. Start server:
2. `python tcp_server.py`
3. Start client (in another terminal):
4. `python tcp_client.py clientA`
5. Observe server logging and client receipts.

What to include in documentation / report (what a student should write)

- **Short introduction**: scenario and goals (smart home P2P discovery).
- **Architecture diagram**: sensor ↔ broadcast network ↔ actuator.
- **Message formats**: show `hello` and `temperature` JSON schemas.
- **Classes and responsibilities**: list classes (`Peer`, `Broadcaster`, `Listener`, utils) and a sentence for each.
- **How to run**: exact commands used to launch peers (copy/paste).
- **Observed logs**: paste console output showing discovery and message exchange.
- **Next steps**: short notes on what to add in Submission 2 (TLS, authentication).

Common gotchas and debugging tips

- **Broadcast not received on some networks** (e.g., guest Wi-Fi or Windows firewall):
  - Try running both peers on same machine using `localhost` and TCP testing, or disable firewall temporarily for testing.
  - For Windows, you may need to allow Python in firewall rules.
- **`OSError: [Errno 98] Address already in use`**: use `SO_REUSEADDR` or ensure no other app uses the port.
- **No discovery seen**: check the port number match and that both scripts use same `BROADCAST_PORT`.
- **JSON errors**: wrap `json.loads` in try/except and log raw bytes to debug malformed messages.

What is a Socket?

Definition

A **socket** is like a **virtual endpoint** that lets two programs communicate across a network (or even within the same machine).
It's how Python (or any language) sends and receives data between computers.

How it works

Every networked process connects via:

- **An IP address** → identifies the machine
- **A Port number** → identifies the specific application on that machine

A socket combines these:

(IP address, Port number) = one endpoint

When two sockets connect → a communication channel is formed.

Types of sockets

Type	Description	Protocol	Example Use
TCP socket	Connection-oriented, reliable	Transmission Control Protocol	Web servers, file transfer
UDP socket	Connectionless, fast but unreliable	User Datagram Protocol	Real-time data, discovery, IoT broadcast

---

Example: TCP Socket (reliable connection)

Server:

```
import socket

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # AF_INET
= IPv4, SOCK_STREAM = TCP
server.bind(('127.0.0.1', 5000))  # Bind to local address
server.listen(1)
print("Server waiting for connection...")
conn, addr = server.accept()
print("Connected by", addr)
while True:
    data = conn.recv(1024)  # receive up to 1024 bytes
    if not data:
        break
    print("Server received:", data.decode())
    conn.sendall(b"Message received")
conn.close()
```

Client:

```
import socket

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

```
client.connect(('127.0.0.1', 5000))
client.sendall(b"Hello, server!")
response = client.recv(1024)
print("Client got:", response.decode())
client.close()
```

✅ Run the server first, then the client in another terminal.

Example: UDP Socket (connectionless)

Sender:

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.sendto(b"Hello, anyone?", ('255.255.255.255', 6000))  # broadcast
message
```

Receiver:

```
import socket
r = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
r.bind(('', 6000))
print("Listening for UDP messages...")
data, addr = r.recvfrom(1024)
print("Received:", data.decode(), "from", addr)
```

UDP is what we use in your **peer-to-peer discovery** example.

---

2. What is Threading?

Definition

Threading means running **multiple parts of a program concurrently** (at the same time) in the same process.

- A **thread** is a lightweight unit of execution.
- All threads share the same memory space, but each runs independently.

Why use threads with sockets?

Because **network I/O blocks** — when you call `recv()` or `accept()`, the program waits until data arrives.
Without threads, your program would "freeze" while waiting.

With threading:

- One thread can **listen** for messages (`recv()`),
- While another thread **sends** or performs other tasks.

This is why our peer-to-peer example uses:

```
threading.Thread(target=listen_for_peers, daemon=True).start()
```

It allows listening in the background while broadcasting messages at the same time.

Simple example: multi-threaded server

```
import socket
import threading

def handle_client(conn, addr):
    print("Connected:", addr)
    while True:
        data = conn.recv(1024)
        if not data:
            break
        print(f"Received from {addr}: {data.decode()}")
        conn.sendall(b"ACK")
    conn.close()

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('127.0.0.1', 5001))
server.listen()
print("Server listening...")

while True:
    conn, addr = server.accept()
    thread = threading.Thread(target=handle_client, args=(conn,
addr))
    thread.start()
```

Each client gets its own thread, so multiple clients can connect simultaneously.

Summary: when to use threads

✅ Use **threads** when you:

- Need to handle multiple connections at once.
- Must listen and send simultaneously.
- Have blocking I/O operations (sockets, files).

But be careful:

- Too many threads can slow things down.
- Threads share memory — can cause race conditions.

3. What is Asynchronous I/O (asyncio)?

Concept

`asyncio` is Python's **modern alternative to threading**.
It allows **non-blocking**, **single-threaded** concurrency using *coroutines* (functions that can pause while waiting).

Instead of creating many threads, you define `async` functions that **yield control** while waiting for I/O — letting others run.

□ Key terms

Term	Meaning
`async def`	Declares an asynchronous function
`await`	Pauses execution until an async operation finishes
`event loop`	Core scheduler that runs async tasks

Example: async TCP echo server

Server:

```
import asyncio

async def handle_client(reader, writer):
    addr = writer.get_extra_info('peername')
    print("Connected:", addr)
    while True:
        data = await reader.read(100)
        if not data:
            break
        print(f"Received from {addr}: {data.decode()}")
        writer.write(b"ACK\n")
        await writer.drain()
    writer.close()
    await writer.wait_closed()

async def main():
    server = await asyncio.start_server(handle_client, '127.0.0.1',
5002)
    print("Async server running...")
    async with server:
        await server.serve_forever()
```

```
asyncio.run(main())
```

Client:

```
import asyncio

async def tcp_client():
    reader, writer = await asyncio.open_connection('127.0.0.1', 5002)
    writer.write(b"Hello async server!\n")
    await writer.drain()
    data = await reader.read(100)
    print("Received:", data.decode())
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_client())
```

✅ Run the server first, then the client.

Why asyncio instead of threads?

Feature	Threading	asyncio
Concurrency model	Multiple OS threads	Single-threaded event loop
Memory	More overhead	Lightweight coroutines
Speed	Slower for thousands of connections	Scales efficiently
Difficulty	Easier to understand	Requires async/await syntax

For IoT projects where each peer may handle **many small, frequent network messages**, `asyncio` can be more efficient.

---

asyncio in IoT (example idea)

Imagine your sensor peer:

- Waits for incoming temperature requests.
- Periodically sends data to multiple actuators.

With asyncio, you can:

- `await` for incoming UDP/TCP data,
- `await` for timers or network sends,
- Do everything inside one loop, **no threads needed**.

Summary Table

Concept	Purpose	Python Module	Example Use in IoT
Socket	Low-level network communication endpoint	`socket`	Sending sensor data or discovery
Threading	Run multiple blocking tasks in parallel	`threading`	Listening + broadcasting concurrently
Multithreading	Many threads running at once	`threading`	Multi-client server
asyncio	Non-blocking concurrency using event loop	`asyncio`	Scalable communication without threads

What students should do first

1. **Run simple TCP socket examples** → understand basic send/receive.
2. **Modify to UDP broadcast** → understand discovery (no connection).
3. **Add threading** → run listener + broadcaster concurrently.
4. **(Optional)** Try the asyncio echo example → understand event loop.
5. **Then** move to your P2P system (Submission 1).

Each section:

- **Concept explanation**
- **Key takeaway**
- **Runnable Python code**
- **Expected output**
- **Analysis of what's happening**

You can run each example in **VS Code** or **PyCharm** — just open two terminals or run in two separate windows where needed.

Example 1 – Simple TCP Client and Server

Concept

This demonstrates the most basic **reliable connection** between two programs using **TCP sockets**.
TCP guarantees that messages arrive intact and in order.

Code

server_tcp.py

```
import socket

# Create TCP socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('127.0.0.1', 5000))   # local IP + port
server.listen(1)
print("Server waiting for connection...")

conn, addr = server.accept()
print(f"Connected by {addr}")

while True:
    data = conn.recv(1024)
    if not data:
        break
    print("Server received:", data.decode())
    conn.sendall(b"Message received by server")

conn.close()
```

client_tcp.py

```
import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('127.0.0.1', 5000))
client.sendall(b"Hello, server! I'm the client.")
response = client.recv(1024)
print("Client got:", response.decode())
client.close()
```

Run

1. Start the server: `python server_tcp.py`
2. Then run the client: `python client_tcp.py`

Expected Output

Server:

```
Server waiting for connection...
Connected by ('127.0.0.1', 56318)
Server received: Hello, server! I'm the client.
```

Client:

```
Client got: Message received by server
```

Analysis

- `socket.AF_INET` → IPv4
- `socket.SOCK_STREAM` → TCP
- `bind()` attaches to IP/port
- `accept()` waits (blocks) until a client connects
- `recv()` waits for data
- This blocking behavior is why we later use threads or asyncio.

Example 2 – UDP Broadcast (Connectionless)

Concept

UDP sends **datagrams** without setting up a connection — fast but unreliable.
Perfect for IoT **discovery messages** (like "HELLO").

Code

udp_receiver.py

```
import socket
receiver = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
receiver.bind(('', 6000))
print("Listening for UDP messages...")
while True:
    data, addr = receiver.recvfrom(1024)
    print("Received:", data.decode(), "from", addr)
```

udp_sender.py

```
import socket
sender = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sender.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
sender.sendto(b"HELLO from UDP sender!", ('255.255.255.255', 6000))
print("Broadcast sent")
```

Run

1. Run `udp_receiver.py` first.
2. Then run `udp_sender.py`.

Output

```
Received: HELLO from UDP sender! from ('192.168.1.23', 53321)
```

Analysis

- UDP doesn't establish connections (`connect()` not needed).
- `255.255.255.255` sends to all devices on local network.

- Good for peer discovery in IoT networks.

Example 3 – Multi-Threaded TCP Server

Concept

A server that can talk to **multiple clients** simultaneously by creating a **new thread** for each connection.

Code

server_threaded.py

```
import socket, threading

def handle_client(conn, addr):
    print("Connected:", addr)
    while True:
        data = conn.recv(1024)
        if not data:
            break
        print(f"Received from {addr}: {data.decode()}")
        conn.sendall(b"ACK")
    conn.close()

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('127.0.0.1', 5001))
server.listen()
print("Server listening on port 5001...")

while True:
    conn, addr = server.accept()
    thread = threading.Thread(target=handle_client, args=(conn,
addr))
    thread.start()
```

client_threaded.py

```
import socket, time
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('127.0.0.1', 5001))

for i in range(3):
    msg = f"Hello {i}".encode()
    client.sendall(msg)
    print("Sent:", msg)
    print("Received:", client.recv(1024).decode())
    time.sleep(1)

client.close()
```

Run

1. Run the server.
2. Open two terminals and run two clients simultaneously.

Output

The server will show two clients' messages interleaved:

```
Connected: ('127.0.0.1', 56234)
Connected: ('127.0.0.1', 56235)
Received from ('127.0.0.1', 56234): Hello 0
Received from ('127.0.0.1', 56235): Hello 0
...
```

Analysis

- Each client handled by a separate **thread** → concurrency.
- Threads share memory, so care is needed with shared data.
- Good for small to medium IoT systems.

Example 4 – Threaded UDP Listener + Broadcaster

Concept

A peer that **listens and sends** at the same time using threads — similar to your **P2P IoT sensor**.

Code

peer_udp_threaded.py

```
import socket, threading, time, json, random

def listen():
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind(('', 6001))
    print("Listening for peers...")
    while True:
        data, addr = sock.recvfrom(1024)
        message = json.loads(data.decode())
        print(f"[RECEIVED from {addr}] {message}")

def broadcast():
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
    while True:
        msg = {"type": "temperature", "value":
round(random.uniform(20, 30), 1)}
        sock.sendto(json.dumps(msg).encode(), ('255.255.255.255',
6001))
        print("[SENT]", msg)
        time.sleep(3)
```

```
threading.Thread(target=listen, daemon=True).start()
broadcast()
```

Run

1. Open two terminals.
2. Run the same script in both.

Output

Each peer shows both "sent" and "received" messages:

```
[SENT] {'type': 'temperature', 'value': 25.1}
[RECEIVED from ('192.168.1.45', 6001)] {'type': 'temperature',
'value': 22.8}
```

Analysis

- Uses **UDP broadcast** for discovery.
- Uses **JSON** for structured messages.
- Uses **threading** to allow simultaneous listening and sending.
- This forms the **foundation of your Submission 1 P2P app**.

Example 5 – AsyncIO TCP Echo Server and Client

Concept

Replace threads with **async coroutines** using `asyncio`.
Everything runs in one event loop → more scalable.

Code

server_asyncio.py

```
import asyncio

async def handle_client(reader, writer):
    addr = writer.get_extra_info('peername')
    print("Connected:", addr)
    while True:
        data = await reader.read(100)
        if not data:
            break
        print(f"Received: {data.decode()}")
        writer.write(b"ACK\n")
        await writer.drain()
    writer.close()
    await writer.wait_closed()

async def main():
```

```
    server = await asyncio.start_server(handle_client, '127.0.0.1',
5002)
    print("Async server running...")
    async with server:
        await server.serve_forever()

asyncio.run(main())
```

client_asyncio.py

```
import asyncio

async def tcp_client():
    reader, writer = await asyncio.open_connection('127.0.0.1', 5002)
    writer.write(b"Hello async server!\n")
    await writer.drain()
    data = await reader.read(100)
    print("Client received:", data.decode())
    writer.close()
    await writer.wait_closed()

asyncio.run(tcp_client())
```

Run

1. Start the server: `python server_asyncio.py`
2. Run the client: `python client_asyncio.py`

Output

```
Async server running...
Connected: ('127.0.0.1', 56320)
Received: Hello async server!
Client received: ACK
```

Analysis

- `async def` defines coroutine functions.
- `await` yields control while waiting for I/O → non-blocking.
- Efficient for **many connections** with minimal CPU overhead.
- Used in **modern IoT platforms** (MQTT, CoAP, etc.) for event-driven data.

Summary of Learning Steps

Step	Example	Concept Learned	Relevance to IoT P2P
1	TCP Server/Client	Basic socket send/receive	Core communication
2	UDP Broadcast	Connectionless messages	Peer discovery
3	Threaded Server	Handling multiple peers	Scalable P2P

Step	Example	Concept Learned	Relevance to IoT P2P
4	Threaded UDP Peer	Listen + broadcast concurrently	Submission 1 base
5	AsyncIO Echo	Non-blocking concurrency	Advanced scalability