Lab Worksheet 5: Peer Discovery in a Simple P2P Network

Codebase: `p2p_peer_discovery_v2_fix_patched.zip`

Core idea: nodes discover peers on the local network via **UDP multicast beacons** every 60 seconds (group **239.255.0.1:41000**), and can also connect via **manual bootstrap**. This lab measures discovery latency, registry states, and behavior under churn.

Learning goals

By the end of this lab, you will be able to: - Explain how multicast **LAN discovery** works and when it triggers. - Use and compare **automatic discovery vs. manual bootstrap**. - Interpret the **Peer Registry** (attempts, connected, disconnected, counters). - Measure **time- to- discover** and the effect of beacon intervals. - Observe resilience under **join/leave churn** and across **multi- host** setups.

What's in the repo (high-level)

- `node.py` — CLI node with a tiny REPL (commands: `echo`, `put`, `get`, `peers`, `quit`).
- `peer_to_peer/discovery.py` — **UDP multicast discovery** (`MCAST_GRP=239.255.0.1`, `MCAST_PORT=41000`, `BEACON_INTERVAL=60s`).
- `peer_to_peer/network.py` — TCP listener/outbound connects, connection tracking.
- `peer_to_peer/connection.py` — length- prefixed JSON messaging.
- `peer_to_peer/peer_registry.py` — tracks status per peer: `attempted|connected|disconnected`, with `success_count/fail_count` and timestamps.
- `README.md` — quick start.

Important discovery details: Each node periodically multicasts a small JSON beacon identifying itself (node id + TCP port). Receivers ignore loopback senders, maintain a `seen` map to deduplicate, and call a provided `on_peer((ip, port))` handler when a new peer is observed.

1) Setup (5–10 min)

Open two or more terminals. Python 3.8+ is fine (standard library only).

Start Node A (no bootstrap):

```
python node.py --port 50001
```

Start Node B (no bootstrap — relies on discovery):

```
python node.py --port 50002
```

(Optional) **Start Node C (bootstrap to A for immediate connectivity):**

```
python node.py --port 50003 --bootstrap 127.0.0.1:50001
```

> If you're on Windows, use `py -3 node.py ...`. If a port is busy, pick another.

2) Baseline: time-to-discover

Discovery beacons go out every **60s**, so allow up to ~60s after a node starts.

Procedure 1. Start A, then B (both **without** `--bootstrap`). 2. In each console, run `peers` every ~10s until the other appears. 3. Record the timestamp when B first shows A, and when A first shows B.

Table: Baseline LAN discovery

Origin	Other node seen?	Time started	First seen at	Δ seconds
A (50001)	B (50002)			
B (50002)	A (50001)			

Questions 1. Did discovery appear roughly within 60s? If not, what might delay it (OS multicast, firewall, NIC selection)? 2. Why does multicast discovery typically remain within a local subnet/VLAN?

3) Bootstrap vs. discovery

Compare immediate bootstrap to periodic discovery.

Procedure 1. Stop Node C if running. Start C **with bootstrap** to A: `bash python node.py --port 50003 --bootstrap 127.0.0.1:50001` 2. Immediately run `peers` on A, B, and C. 3. After ~60s, run `peers` again.

Table: Bootstrap vs discovery

Node	Immediately sees peers via bootstrap?	After 60s (discovery beacons)
A (50001)		
B (50002)		
C (50003, bootstrapped)		

Questions 3. What advantage does `--bootstrap` give compared to waiting for beacons? 4. Once one connection exists, how might the network learn about third peers sooner than a full beacon cycle?

4) Peer Registry: attempts, connects, disconnects

The registry records every **outbound attempt**, successful **incoming/outgoing connects**, and **disconnects**.

Procedure 1. With A, B, C running, execute `peers` on each and note statuses (`attempted|connected|disconnected`) and counters. 2. **Kill B** (Ctrl+C). On A and C, run `peers` again. 3. **Restart B** with `python node.py --port 50002` and after discovery/handshake, run `peers` on all.

Table: Registry observations

Observation point	A's view of B	A's view of C	B's view of A	C's view of A
Steady state				
After B killed				
After B restarts				

Questions 5. Which counters changed when B was down? Why might `fail_count` increment even if multicast later rediscovers B? 6. How do you expect the registry to behave if two nodes learn each other simultaneously (race between incoming vs outgoing)?

5) Churn and re-discovery

Measure how long it takes to heal after a node leaves and returns.

Procedure 1. Stop A for 20–30s, leaving B and C up. 2. Restart A and record how quickly B and C show A again on `peers`. 3. Compare the latency to Section 1.

Table: Re- discovery latency

Node that returned	Seen by which node	Time restarted	First seen again	Δ seconds
A	B			
A	C			

Question 7. Does re- discovery rely solely on the next 60s beacon, or do existing TCP connections/handshakes help propagate awareness sooner?

6) Faster lab runs : adjust the beacon interval (Optional)

For rapid iteration, you can temporarily set the beacon interval to **5s**.

Edit `peer_to_peer/discovery.py`:

```
BEACON_INTERVAL = 5.0  # seconds  (default is 60.0)
```

Restart nodes and **repeat Sections 1 & 4**, measuring the new Δ seconds.

Table: Interval comparison

Scenario	Interval	Avg Δ seconds
Initial discovery (A↔B)	60s	
Initial discovery (A↔B)	5s	
Re- discovery after churn	60s	
Re- discovery after churn	5s	

Question 8. What trade- offs come with shorter beacon intervals (network overhead, CPU wakeups, battery, log noise)?

7) Multi-host LAN test (optional)

Run nodes on two machines on the same LAN.

Procedure 1. On **Machine 1** run A (`--port 50001`). On **Machine 2** run B (`--port 50002`). 2. Ensure both machines allow UDP multicast and inbound TCP on your chosen ports. 3. Observe discovery latency and `peers` states.

Questions 9. Why won't discovery cross most routers by default? What would you add (e.g., a tracker/seed server, relays, or mDNS) to reach peers across subnets? 10. If a machine has multiple NICs (Wi-Fi + Ethernet), how could the multicast **outgoing interface** selection affect who hears your beacons?

8) (Deep dive) Inspecting the discovery logic

Open `peer_to_peer/discovery.py` and find: - `MCAST_GRP`, `MCAST_PORT`, and `BEACON_INTERVAL`. - The **send** thread that periodically multicasts a JSON like `{type:"p2p_discovery", port: <tcp_port>, id: <uuid>}`. - The **receive** loop that ignores loopback (`127.*`), updates `seen[(ip,port)] = now`, and invokes `on_peer((ip,port))`.

Questions 11. Why is there a `seen` map? What would happen without it (hint: duplicate events every beacon)? 12. What simple heuristic could you add to **age out** peers that haven't been seen in a while?

Troubleshooting

- **Nothing discovered after 60s:** Check OS firewall for UDP multicast and inbound TCP. Try `--bootstrap` once to seed connections.
- **Multiple NICs:** Prefer one NIC or disable the unused one; ensure your OS is sending multicast on the intended interface.
- **Same- host quirkiness:** If discovery on a single machine seems flaky, leave one node bootstrapped to the other; multicast behavior can vary by OS when only loopback is active.
- **Address in use:** Another node is running on that port; stop it or pick a different port.

Quick command reference

```
# Start nodes
python node.py --port 50001
python node.py --port 50002
python node.py --port 50003 --bootstrap 127.0.0.1:50001

# In the REPL (per node)
peers      # show registry status per peer (attempted/connected/disconnected
+ counters)
put k v    # store a key on this node (gossiped best-effort)
get k      # fetch a key
echo text  # demo message round-trip
quit       # exit
```

Remember: Discovery beacons are periodic; bootstrap gives immediate connectivity. Use both to balance fast joins and low background noise.