به نام خدا

گزارش پروژه

شبکه‌های کامپیوتری

استاد: امیرمهدی صادق‌زاده

ارائه‌دهندگان:

سارا قضاوی – زهرا قصابی

۴۰۲۱۰۶۳۴۸ – ۴۰۲۱۰۶۳۳۷

دانشکده مهندسی کامپیوتر

دانشگاه صنعتی شریف

فاز اول

در فاز اول پروژه هدف ما این است که قبل از شروع پیاده‌سازی، پروتکل Gossip را به‌صورت دقیق و قابل‌اجرا مشخص کنیم تا رفتار شبکه، ساختار پیام‌ها و تصمیم‌گیری هر گره مشخص باشد. هر گره در این طراحی دارای موارد زیر است:

۱. یک شناسه یکتا NodeID (مانند UUID مستقل از IP:Port)

۲. یک آدرس محلی (SelfAddr)

۳. Peer List شامل همسایه‌ها و متادیتاهایی از آنها (آخرین زمان پاسخ هر همسایه، شمارنده عدم پاسخ برای تشخیص همسایه‌های مرده و ...)

۴. Seen Set شامل msg_id پیام‌های دیده شده برای جلوگیری از پردازش تکراری و forward چندباره

۵. مجموعه پارامترهای قابل تنظیم: fanout، TTL، peer_limit، ping_interval و peer_timeout (از طریق خط فرمان یا فایل تنظیمات تغییر می‌کنند)

پیام‌ها روی UDP و در قالب JSON با هدر مشترکی شامل version، msg_id، msg_type، sender_id، sender_addr، timestamp_ms و ttl ارسال می‌شوند و payload بسته به نوع پیام متفاوت است. پروتکل پیام‌های زیر را پشتیبانی می‌کند:

- پیام HELLO برای معرفی گره جدید

```
{
  "payload": {
    "capabilities": ["udp", "json"]
  }
}
```

- پیام GET_PEERS برای درخواست همسایه

```
{
  "payload": {
    "max_peers": 20
  }
}
```

- پیام PEERS_LIST برای دریافت لیست همسایه‌ها

```
{
  "payload": {
    "peers": [
      {"node_id": "...", "addr": "127.0.0.1:8001"},
      {"node_id": "...", "addr": "127.0.0.1:8002"}
    ]
  }
```

```
}
```

- پیام GOSSIP برای انتشار داده

```
{
  "payload": {
    "topic": "news",
    "data": "Hello network!",
    "origin_id": "node-uuid",
    "origin_timestamp_ms": 1730
  }
}
```

- پیام‌های PING و PONG برای مدیریت همسایه‌ها

```
{
  "payload": {
    "ping_id": "uuid-or-counter",
    "seq": 17
  }
}
{
  "payload": {
    "ping_id": "uuid-or-counter",
    "seq": 17
  }
}
```

در این پروتکل با استفاده از پیام‌های PING و PONG می‌توان به صورت دوره‌ای زنده بودن همسایه‌ها را بررسی کرده و همسایه‌های بی‌پاسخ را پس از عبور از peer_timeout و چند بار تکرار، از Peer List حذف کرد. همچنین اندازه Peer List یا همان peer_limit محدود می‌شود و برای جایگزینی همسایه‌ها، یک سیاست مشخص (مثل حذف قدیمی‌ترین یا کم‌فعال‌ترین) در نظر گرفته می‌شود.

منطق Bootstrap به این صورت طراحی می‌شود که گره تازه‌وارد آدرس یک یا چند گره seed را از قبل می‌داند، ابتدا HELLO و سپس GET_PEERS را به گره seed می‌فرستد، PEERS_LIST را دریافت کرده و لیست خود را تا سقف peer_limit تکمیل می‌کند و سپس بدون وابستگی دائمی به seed وارد چرخه عادی شبکه می‌شود. منطق اصلی Gossip نیز این است که هر گره پس از دریافت پیام GOSSIP ابتدا msg_id را با Seen Set بررسی می‌کند. اگر پیام قبلا دیده شده باشد آن را نادیده می‌گیرد و در غیر این صورت پیام را پردازش و در Seen Set ثبت کرده و زمان دریافت را در لاگ ذخیره می‌کند، سپس TTL را یک واحد کاهش می‌دهد و اگر هنوز TTL > 0 باشد پیام را به تعداد fanout همسایه تصادفی (بدون تکرار) فوروارد می‌کند تا انتشار با هزینه کم و به‌صورت مقیاس‌پذیر انجام شود و در عین حال TTL از انتشار بی‌نهایت جلوگیری کند. در پایان طراحی، قالب لاگ‌گیری نیز مشخص می‌شود تا در فاز تحلیل بتوان زمان همگرایی (لحظه رسیدن پیام به ۹۵٪ گره‌ها) و سربار پیام (تعداد کل پیام‌های ارسال‌شده شامل داده و کنترل تا رسیدن به پوشش ۹۵٪) را از روی لاگ‌ها استخراج و مقایسه کرد.

فازدوم

در فاز پیاده‌سازی، یک برنامه اجرایی به نام node.py طراحی می‌شود که نقش یک گره در شبکه P2P را ایفا می‌کند. ارتباطات شبکه‌ای مطابق توضیح پروژه با UDP انجام شده و قالب پیام‌ها JSON در نظر گرفته شده است. به صورت خلاصه این برنامه نقش یک گره را می‌سازد که به گره‌های دیگر وصل می‌شود (discovery و boostrap)، همسایه‌ها را نگه می‌دارد و سلامتشان را با PING/PONG بررسی می‌کند، پیام‌های GOSSIP را با TTL و fanout پخش می‌کند، با Seen set از پردازش تکراری پیام‌ها جلوگیری می‌کند و در آخر رخدادها را به صورت JSON، لاگ می‌کند. برای همزمانی گوش دادن به پیام‌های ورودی، ارسال پیام‌های دوره‌ای و دریافت ورودی کاربر، از مدل event-driven مبتنی بر asyncio استفاده کردیم تا بدون پیچیدگی چند فعالیت به صورت همزمان و non-blocking اجرا شوند. هر گره دارای شناسه یکتا و آدرس محلی است و پارامترهای قابل تنظیم مانند fanout، TTL، peer_limit، ping_interval، peer_timeout و seed از طریق ترمینال دریافت می‌شوند.

ساختارهای کلی

- کتابخانه argparse برای گرفتن پارامترهای اجرا از CLI
- کتابخانه asyncio برای اجرای همزمان چند عملیات (ping دوره‌ای، ورودی کاربر، شبکه، discovery دوره‌ای)
- کتابخانه uuid برای ساخت node_id و msg_id یکتا
- کتابخانه OrderedDict برای پیاده‌سازی Seen Set به صورت LRU
- ساختار PeerState که یک dataclass است، وضعیت هر همسایه را نگهداری می‌کند:

```python
@dataclass  2 usages
class PeerState:
    peer_id: Optional[str]
    addr: Addr
    last_seen_ms: int
    fail_count: int = 0
```

- ساختار SeenLRU به عنوان Seen-Set عمل می‌کند و پیام‌های دیده شده را با ظرفیت محدود نگه می‌دارد:

```python
class SeenLRU:  1 usage
    def __init__(self, capacity: int = 50000):
        self.capacity = capacity
        self.od = OrderedDict()

    def __contains__(self, k: str) -> bool:
        return k in self.od

    def add(self, k: str, ts_ms: int) -> None:  2 usages
        if k in self.od:
            return
        self.od[k] = ts_ms
        self.od.move_to_end(k)
        if len(self.od) > self.capacity:
            self.od.popitem(last=False)
```

متد add پیام جدید را اضافه می‌کند و اگر ظرفیت پر شود قدیمی‌ترین پیام را حذف می‌کند.

- ساختار NodeProtocol لایه UDP در asyncio است که با DatagramProtocol کار می‌کند:

```python
class NodeProtocol(asyncio.DatagramProtocol):  1 usage
    def __init__(self, node: "GossipNode"):
        self.node = node

    def connection_made(self, transport):
        self.node.transport = transport

    def datagram_received(self, data: bytes, addr: Addr):
        asyncio.create_task(self.node.on_datagram(data, addr))
```

در متد connection_made، transport داخل گره ذخیره می‌شود و در datagram_received پردازش بسته async می‌شود تا دریافت بسته‌ها بلاک نشود.

کلاس اصلی، کلاس GossipNode است:

```python
class GossipNode:  2 usages
    def __init__(self, args):
        self.cfg = args
        self.node_id = str(uuid.uuid4())
        self.self_addr: Addr = ("127.0.0.1", args.port)

        self.transport = None
        self.peers: Dict[Addr, PeerState] = {}
        self.seen = SeenLRU(capacity=50000)
        self.store: Dict[str, dict] = {}

        self.rng = random.Random(args.seed)
        self.protocol = NodeProtocol(self)

        self.pending_pings: Dict[str, Tuple[Addr, int]] = {}
        self.pending_by_addr: Dict[Addr, str] = {}
```

این کلاس شامل متغیرهای لازم برای ساخت یک گره است: node_id یکتا، آدرس نود و پورت آن، دیکشنری همسایه‌ها، لیست Seen Set، دیکشنری store برای ذخیره پیام‌های gossip دریافتی و تولیدی، مقدار رندم RNG برای انتخاب همسایه‌ها به صورت قابل تکرار، دو دیکشنری pending_pings و pending_by_addr برای دنبال کردن ping های بدون پاسخ

لاگ‌اندازی

```python
def log(self, event: str, **fields):  19 usages
    rec = {
        "ts_ms": now_ms(),
        "node_id": self.node_id,
        "self": addr_to_str(self.self_addr),
        "event": event,
        **fields,
    }
    print(json.dumps(rec, ensure_ascii=False), flush=True)
```

از این تابع برای لاگ کردن event ها به صورت JSON استفاده می‌شود.

```python
def pack(self, msg_type: str, payload: dict, ttl: int, msg_id: Optional[str] = None) -> bytes:  6 usages
    mid = msg_id or str(uuid.uuid4())
    msg = {
        "version": 1,
        "msg_id": mid,
        "msg_type": msg_type,
        "sender_id": self.node_id,
        "sender_addr": addr_to_str(self.self_addr),
        "timestamp_ms": now_ms(),
        "ttl": ttl,
        "payload": payload,
    }
    return json.dumps(msg).encode("utf-8")
```

در این تابع، هدر استاندارد با متغیرهای لازم و payload در قالب یک پیام ساخته می‌شوند.

```python
async def sendto(self, msg_bytes: bytes, to_addr: Addr, meta: dict):  9 usages (1 dynamic)
    try:
        self.transport.sendto(msg_bytes, to_addr)
        self.log( event: "send", to=addr_to_str(to_addr), **meta)
    except Exception as e:
        self.log( event: "send_error", to=addr_to_str(to_addr), err=str(e), **meta)
```

در این تابع بسته ارسال می‌شود و لاگ send می‌اندازد.

مدیریت PeerList

```python
def peer_add(self, addr: Addr, peer_id: Optional[str]):  2 usages
    if addr in self.peers:
        ps = self.peers[addr]
        ps.peer_id = peer_id or ps.peer_id
        ps.last_seen_ms = now_ms()
        return

    if len(self.peers) >= self.cfg.peer_limit:
        stale_addr = min(self.peers.values(), key=lambda p: p.last_seen_ms).addr
        self.remove_peer(stale_addr,  reason: "peer_limit")

    self.peers[addr] = PeerState(peer_id=peer_id, addr=addr, last_seen_ms=now_ms(), fail_count=0)
    self.log( event: "peer_added", peer=addr_to_str(addr), peer_id=peer_id)
```

در این تابع همسایه جدید اضافه می‌شود و اگر ظرفیت لیست پر شود، قدیمی‌ترین همسایه حذف می‌شود.

```python
def peer_touch(self, addr: Addr, peer_id: Optional[str] = None):  1 usage
    if addr not in self.peers:
        self.peer_add(addr, peer_id)
    else:
        ps = self.peers[addr]
        ps.last_seen_ms = now_ms()
        ps.fail_count = 0
        if peer_id and not ps.peer_id:
            ps.peer_id = peer_id
```

در این تابع وقتی از یک همسایه پیام می‌رسد، last_seen به‌روزرسانی می‌شود و fail_count صفر می‌شود.

```python
def remove_peer(self, addr: Addr, reason: str):  2 usages
    removed = self.peers.pop(addr, None)
    if removed:
        pid = self.pending_by_addr.pop(addr, None)
        if pid:
            self.pending_pings.pop(pid, None)
        self.log( event: "peer_removed", reason=reason, peer=addr_to_str(addr))
```

در این تابع یک همسایه حذف می‌شود و ping های مربوط به آن پاک می‌شوند.

```python
def pick_peers(self, k: int, exclude: Optional[Addr] = None) -> List[Addr]:  5 usages
    addrs = list(self.peers.keys())
    if exclude and exclude in addrs:
        addrs.remove(exclude)
    if not addrs:
        return []
    k = min(k, len(addrs))
    self.rng.shuffle(addrs)
    return addrs[:k]
```

در این تابع به صورت تصادفی، k همسایه به صورت تصادفی حذف می‌شوند. همچنین امکان استفاده از exclude وجود دارد.

مسیر دریافت بسته

```python
async def on_datagram(self, data: bytes, addr: Addr):  1 usage
    try:
        s = data.decode( encoding: "utf-8", errors="strict")
        msg = json.loads(s)
    except Exception:
        self.log( event: "recv_bad_json", from_addr=addr_to_str(addr))
        return

    mtype = msg.get("msg_type")
    mid = msg.get("msg_id")
    sender_id = msg.get("sender_id")
    ttl = msg.get("ttl")

    if not isinstance(mid, str) or not mid:
        self.log( event: "recv_bad_msg_id", from_addr=addr_to_str(addr), msg_type=str(mtype))
        return

    if mid in self.seen:
        self.log( event: "recv_duplicate", from_addr=addr_to_str(addr), msg_type=str(mtype), msg_id=mid)
        return
    self.seen.add(mid, now_ms())

    self.log( event: "recv", from_addr=addr_to_str(addr), msg_type=mtype, msg_id=mid)

    if isinstance(sender_id, str):
        self.peer_touch(addr, sender_id)
```

در این تابع ابتدا داده دیکود می‌شود و سپس به JSON تبدیل می‌شود. msg_id اعتبارسنجی می‌شود و Seen Set بررسی می‌شود تا پیام تکراری نباشد. اگر تکراری نباشد، به لیست اضافه می‌شود و لاگ recv ایجاد می‌شود. در آخر همسایه به‌روزرسانی می‌شود و بر اساس msg_type، dispatch می‌شود:

```python
if mtype == "HELLO":
    await self.handle_hello(msg, addr)
elif mtype == "GET_PEERS":
    await self.handle_get_peers(msg, addr)
elif mtype == "PEERS_LIST":
    await self.handle_peers_list(msg)
elif mtype == "PING":
    await self.handle_ping(msg, addr)
elif mtype == "PONG":
    await self.handle_pong(msg, addr)
elif mtype == "GOSSIP":
    if not isinstance(ttl, int):
        return
    await self.handle_gossip(msg, addr)
else:
    self.log( event: "recv_unknown_type", from_addr=addr_to_str(addr), msg_type=str(mtype))
```

هندلرها

```python
async def handle_hello(self, msg: dict, addr: Addr):  1 usage
    return
```

این تابع کاری انجام نمی‌دهد چون peer_touch قبلا انجام شده است.

```python
async def handle_get_peers(self, msg: dict, addr: Addr):  1 usage
    payload = msg.get( key: "payload",  default: {}) or {}
    max_peers = payload.get( key: "max_peers", self.cfg.peer_limit)
    try:
        max_peers = int(max_peers)
    except Exception:
        max_peers = self.cfg.peer_limit

    candidates = [a for a in self.peers.keys() if a != addr and a != self.self_addr]
    if candidates:
        k = min(max_peers, len(candidates))
        chosen = self.rng.sample(candidates, k=k)
    else:
        chosen = []

    peers_list = []
    for a in chosen:
        ps = self.peers.get(a)
        if not ps:
            continue
        peers_list.append({"node_id": ps.peer_id or "", "addr": addr_to_str(a)})

    out = self.pack( msg_type: "PEERS_LIST",  payload: {"peers": peers_list}, ttl=1)
    out_mid = json.loads(out.decode("utf-8")).get("msg_id", "")
    await self.sendto(out, addr,  meta: {"msg_type": "PEERS_LIST", "msg_id": out_mid})
```

این تابع از همسایه‌های شناخته شده نمونه می‌گیرد و PEERS_LIST را می‌فرستد.

```python
async def handle_peers_list(self, msg: dict):  1 usage
    payload = msg.get( key: "payload",  default: {}) or {}
    peers = payload.get( key: "peers",  default: []) or []
    for p in peers:
        try:
            a = parse_addr(p["addr"])
            pid = p.get("node_id") or None
            if a != self.self_addr:
                self.peer_add(a, pid)
        except Exception:
            continue
```

لیست peers دریافتی را پارس می‌کند و سپس به PeerList اضافه می‌کند.

```python
async def handle_ping(self, msg: dict, addr: Addr):  1 usage
    payload = msg.get( key: "payload",  default: {}) or {}
    pong_payload = {"ping_id": payload.get("ping_id"), "seq": payload.get("seq")}
    out = self.pack( msg_type: "PONG", pong_payload, ttl=1)
    out_mid = json.loads(out.decode("utf-8")).get("msg_id", "")
    await self.sendto(out, addr,  meta: {"msg_type": "PONG", "msg_id": out_mid})
```

این تابع یک پاسخ از جنس PONG با همان seq و ping_id می‌فرستد.

```python
async def handle_pong(self, msg: dict, addr: Addr):  1 usage
    payload = msg.get( key: "payload",  default: {}) or {}
    ping_id = payload.get("ping_id")
    if not isinstance(ping_id, str):
        return

    info = self.pending_pings.get(ping_id)
    if not info:
        return

    peer_addr, sent_ts = info
    if peer_addr != addr:
        self.log( event: "pong_addr_mismatch", expected=addr_to_str(peer_addr), got=addr_to_str(addr), ping_id=ping_id)
        return

    self.pending_pings.pop(ping_id, None)
    self.pending_by_addr.pop(addr, None)
    self.log( event: "ping_ok", peer=addr_to_str(addr), rtt_ms=max(0, now_ms() - sent_ts))
```

این تابع ping معلق را می‌بندد و RTT را لاگ می‌کند. همچنین اگر mismatch وجود داشته باشد، آن را تشخیص می‌دهد.

```python
async def handle_gossip(self, msg: dict, from_addr: Addr):  1 usage
    mid = msg.get("msg_id")
    ttl = msg.get("ttl")

    if not isinstance(mid, str):
        return

    self.store[mid] = msg
    self.log( event: "gossip_delivered", msg_id=mid, ttl=ttl, origin=msg.get( key: "payload",  default: {}).get( key: "origi

    new_ttl = int(ttl) - 1
    if new_ttl <= 0:
        self.log( event: "gossip_drop_ttl0", msg_id=mid)
        return

    fwd = dict(msg)
    fwd["ttl"] = new_ttl
    fwd["sender_id"] = self.node_id
    fwd["sender_addr"] = addr_to_str(self.self_addr)
    fwd["timestamp_ms"] = now_ms()
    out = json.dumps(fwd).encode("utf-8")

    targets = self.pick_peers(self.cfg.fanout, exclude=from_addr)
    if not targets:
        targets = self.pick_peers(self.cfg.fanout, exclude=None)
    self.log( event: "gossip_forwarded", msg_id=mid, ttl=new_ttl, fanout=len(targets), targets=[addr_to_str(t) for t in
    for t in targets:
        await self.sendto(out, t,  meta: {"msg_type": "GOSSIP", "msg_id": mid, "ttl": new_ttl})
```

این تابع پیام را در store ذخیره می‌کند و لاگ gossip_delivered می‌اندازد. سپس ttl را یک واحد کم می‌کند و اگر منفی شود drop می‌کند. در غیر این صورت با همان msg_id و ttl جدید به تعداد fanout همسایه فوروارد می‌کند. توجه شود که یافتن همسایه‌ها در دو تلاش است: در تلاش اول فرستنده قبلی exclude می‌شود، اما اگر شکست بخورد بدون exclude تکرار می‌کند.

Bootstrap اولیه

```python
async def bootstrap(self):
    if not self.cfg.bootstrap:
        return
    b = parse_addr(self.cfg.bootstrap)

    hello = self.pack( msg_type: "HELLO",  payload: {"capabilities": ["udp", "json"]}, ttl=1)
    hello_mid = json.loads(hello.decode("utf-8")).get("msg_id", "")
    await self.sendto(hello, b,  meta: {"msg_type": "HELLO", "msg_id": hello_mid})

    getp = self.pack( msg_type: "GET_PEERS",  payload: {"max_peers": self.cfg.peer_limit}, ttl=1)
    getp_mid = json.loads(getp.decode("utf-8")).get("msg_id", "")
    await self.sendto(getp, b,  meta: {"msg_type": "GET_PEERS", "msg_id": getp_mid})
```

در این تابع اگر bootstrap– داده شده باشد، یک HELLO و سپس یک GET_PEERS ارسال می‌شود و سپس از طریق PEERS_LIST شبکه را کشف می‌کند.

حلقه‌های دوره‌ای

چندین حلقه در برنامه به صورت همزمان اجرا می‌شوند:

```python
async def discovery_loop(self):  1 usage
    while True:
        await asyncio.sleep(self.cfg.get_peers_interval)
        if not self.peers:
            continue
        targets = self.pick_peers(min(self.cfg.get_peers_fanout, len(self.peers)))
        for t in targets:
            out = self.pack( msg_type: "GET_PEERS",  payload: {"max_peers": self.cfg.peer_limit}, ttl=1)
            out_mid = json.loads(out.decode("utf-8")).get("msg_id", "")
            await self.sendto(out, t,  meta: {"msg_type": "GET_PEERS", "msg_id": out_mid})
```

این حلقه در هر get_peers_interval از چند همسایه GET_PEERS می‌خواهد و آن را لاگ می‌کند.

```python
async def ping_loop(self):  1 usage
    while True:
        await asyncio.sleep(self.cfg.ping_interval)
        if self.peers:
            targets = self.pick_peers(min(3, len(self.peers)))
            for t in targets:
                if t in self.pending_by_addr:
                    continue
                ping_id = str(uuid.uuid4())
                seq = self.rng.randint( a: 1,  b: 1_000_000)
                out = self.pack( msg_type: "PING",  payload: {"ping_id": ping_id, "seq": seq}, ttl=1)
                out_mid = json.loads(out.decode("utf-8")).get("msg_id", "")
                self.pending_pings[ping_id] = (t, now_ms())
                self.pending_by_addr[t] = ping_id
                await self.sendto(out, t,  meta: {"msg_type": "PING", "msg_id": out_mid})

        deadline = int(self.cfg.peer_timeout * 1000)
        now = now_ms()
        expired = []
        for pid, (a, ts) in list(self.pending_pings.items()):
            if now - ts > deadline:
                expired.append((pid, a))
```

```python
    for pid, a in expired:
        self.pending_pings.pop(pid, None)
        if self.pending_by_addr.get(a) == pid:
            self.pending_by_addr.pop(a, None)
        ps = self.peers.get(a)
        if not ps:
            continue
        ps.fail_count += 1
        self.log( event: "peer_suspect", peer=addr_to_str(a), fail_count=ps.fail_count, reason="ping_timeout")
        if ps.fail_count >= 3:
            self.remove_peer(a,  reason: "timeout")
```

این حلقه در هر ping_interval پیام PING را می‌فرستد و سپس timeout های ping ها را بررسی می‌کند و مقدار fail_count را زیاد می‌کند و پس از سه بار timeout، آن peer را حذف می‌کند.

```python
async def stdin_loop(self):  1 usage
    while True:
        line = await asyncio.to_thread(sys.stdin.readline)
        if not line:
            await asyncio.sleep(0.1)
            continue
        line = line.strip()
        if not line:
            continue

        if line == "peers":
            self.log( event: "peers_dump", peers=[addr_to_str(a) for a in self.peers.keys()])
            continue

        if line.startswith("gossip "):
            parts = line.split( sep: " ",  maxsplit: 2)
            topic = parts[1] if len(parts) >= 2 else "chat"
            data = parts[2] if len(parts) >= 3 else ""
        else:
            topic = "chat"
            data = line
```

```python
    mid = str(uuid.uuid4())
    payload = {
        "topic": topic,
        "data": data,
        "origin_id": self.node_id,
        "origin_timestamp_ms": now_ms(),
    }
    msg = {
        "version": 1,
        "msg_id": mid,
        "msg_type": "GOSSIP",
        "sender_id": self.node_id,
        "sender_addr": addr_to_str(self.self_addr),
        "timestamp_ms": now_ms(),
        "ttl": self.cfg.ttl,
        "payload": payload,
    }

    self.seen.add(mid, now_ms())
    self.store[mid] = msg
    self.log( event: "gossip_created", msg_id=mid, ttl=self.cfg.ttl, topic=topic)

    out = json.dumps(msg).encode("utf-8")
    targets = self.pick_peers(self.cfg.fanout)
    self.log( event: "gossip_forwarded", msg_id=mid, ttl=self.cfg.ttl, fanout=len(targets), targets=[addr_to_str(t)
    for t in targets:
        await self.sendto(out, t,  meta: {"msg_type": "GOSSIP", "msg_id": mid, "ttl": self.cfg.ttl})
```

این حلقه ورودی کاربر را می‌خواند. اگر دستور peers باشد، لیست همسایه‌ها را لاگ می‌کند. اگر دستور <topic> <text> gossip یا <text> gossip باشد، آن را به پیام GOSSIP تبدیل و منتشر می‌کند. همچنین پیام تولیدی به seen و store اضافه می‌شود تا پردازش تکراری نشود.

تابع main و اجرای برنامه

```python
async def main():  1 usage
    ap = argparse.ArgumentParser()
    ap.add_argument( *name_or_flags: "--port", type=int, required=True)
    ap.add_argument( *name_or_flags: "--bootstrap", type=str, default="")
    ap.add_argument( *name_or_flags: "--fanout", type=int, default=3)
    ap.add_argument( *name_or_flags: "--ttl", type=int, default=8)
    ap.add_argument( *name_or_flags: "--peer-limit", dest="peer_limit", type=int, default=20)
    ap.add_argument( *name_or_flags: "--ping-interval", dest="ping_interval", type=float, default=2.0)
    ap.add_argument( *name_or_flags: "--peer-timeout", dest="peer_timeout", type=float, default=6.0)
    ap.add_argument( *name_or_flags: "--seed", type=int, default=42)
    ap.add_argument( *name_or_flags: "--get-peers-interval", dest="get_peers_interval", type=float, default=5.0)
    ap.add_argument( *name_or_flags: "--get-peers-fanout", dest="get_peers_fanout", type=int, default=2)

    args = ap.parse_args()

    node = GossipNode(args)
    loop = asyncio.get_running_loop()
    transport, _ = await loop.create_datagram_endpoint(
        lambda: node.protocol,
        local_addr=("127.0.0.1", args.port)
    )

    node.log( event: "node_started", port=args.port, bootstrap=args.bootstrap)
```

ابتدا پارامترهای CLI تعریف می‌شوند:

port, bootstrap, fanout, ttl, peer-limit, ping-interval, peer-timeout, seed, get-peers-interval, get-peers-fanout

سپس گره ساخته می‌شود و UDP endpoint روی آدرس 127.0.0.1 و پورت ورودی بالا می‌آید.

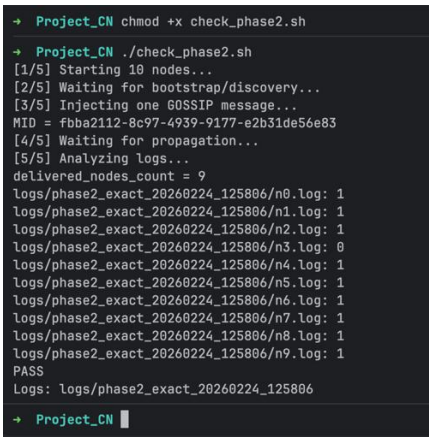
```python
    await node.bootstrap()

    tasks = [
        asyncio.create_task(node.ping_loop()),
        asyncio.create_task(node.discovery_loop()),
        asyncio.create_task(node.stdin_loop()),
    ]
    try:
        await asyncio.gather(*tasks)
    finally:
        transport.close()

if __name__ == "__main__":
    try:
        asyncio.run(main())
    except KeyboardInterrupt:
        pass
```

در ادامه bootstrap اجرا می‌شود و سه تسک stdin_loop، discovery_loop و ping_loop به صورت همزمان اجرا می‌شوند.

برای آزمایش این فاز، اسکریپت check_phase2.sh را می‌نویسیم که ۱۰ گره با پارامترهای مطلوب در صورت پروژه می‌سازد و یک پیام را پخش می‌کند و سپس بررسی می‌کند پیام به چند همسایه رسیده است. این اسکریپت در ضمیمه موجود است و نتیجه آن به صورت زیر است:



فاز سوم

در این فاز یک اسکریپت نوشتیم تا شبکه را آنالیز کند. اسکریپت phase3.py یک پایپ‌لاین است که شبکه را در اندازه‌های ۱۰ و ۲۰ و ۵۰ اجرا می‌کند. برای هر N، ۵ بار با seed متفاوت آزمایش انجام می‌دهد و در هرکدام یک پیام GOSSIP تزریق می‌کند. در آخر از لاگ‌ها معیارهای Convergence Time و Message Overhead را محاسبه می‌کند.

کتابخانه‌های لازم

- کتابخانه argparse برای پارامترهای اجرای اسکریپت
- کتابخانه subprocess برای اجرای همزمان چند گره
- کتابخانه socket برای تزریق پیام GOSSIP از طریق UDP
- کتابخانه json برای پارس کردن لاگ‌ها
- کتابخانه statistics برای به‌دست آوردن میانگین و انحراف معیار

ساختار داده‌ها

```
MSG_TYPES_FOR_OVERHEAD = {"GOSSIP", "HELLO", "GET_PEERS", "PEERS_LIST", "PING", "PONG"}
```

این مقدار ثابت تعیین می‌کند چه پیام‌هایی در overhead شمرده می‌شوند.

```python
@dataclass  5 usages
class Config:
    name: str
    fanout: int
    ttl: int
    peer_limit: int
    ping_interval: float
    peer_timeout: float
```

این که یک dataclass است، پارامترهای هر سناریو را نگه می‌دارد.

```python
def wait_by_n(n: int) -> Tuple[int, int]:  1 usage
    if n <= 10:
        return 25, 8
    if n <= 20:
        return 35, 12
    return 50, 18
```

این تابع بر اساس اندازه شبکه، یک زمان انتظار مناسب برمی‌گرداند. در این پروتکل به دو زمان انتظار نیاز داریم: wait_boot برای شکل گیری topology و wait_prop برای انتشار پیام پس از تزریق آن. برای N بزرگ‌تر، این زمان‌ها بیشتر می‌شوند تا اندازه‌گیری پایدارتر باشد.

```python
def start_node(node_file: str, port: int, cfg: Config, seed: int, log_path: Path, bootstrap: Optional[str]) -> subprocess.Popen:
    cmd = [
        "python3", node_file,
        "--port", str(port),
        "--fanout", str(cfg.fanout),
        "--ttl", str(cfg.ttl),
        "--peer-limit", str(cfg.peer_limit),
        "--ping-interval", str(cfg.ping_interval),
        "--peer-timeout", str(cfg.peer_timeout),
        "--seed", str(seed),
    ]
    if bootstrap:
        cmd += ["--bootstrap", bootstrap]

    f = log_path.open( mode: "w", encoding="utf-8")
    p = subprocess.Popen(cmd, stdout=f, stderr=subprocess.STDOUT, stdin=subprocess.DEVNULL)
    return p
```

این تابع پارامترهای CLI گره را تنظیم می‌کند و در صورت نیاز bootstrap اضافه می‌کند. سپس خروجی را در یک فایل ذخیره می‌کند و گره را با subprocess.Popen بالا می‌آورد. همچنین از stdin=DEVNULL استفاده کردیم تا پردازه‌ها suspeneded نشوند.

```python
def stop_all(procs: List[subprocess.Popen]) -> None:  1 usage
    for p in procs:
        if p.poll() is None:
            p.terminate()
    time.sleep(1.0)
    for p in procs:
        if p.poll() is None:
            p.kill()
```

این تابع در پایان هر اجرا تمام گره‌ها را terminate (و در صورت نیاز kill) می‌کند تا محیط تمیز شود.

```python
def inject_gossip(mid: str, bootstrap_port: int, ttl: int) -> int:  1 usage
    t0 = int(time.time() * 1000)
    msg = {
        "version": 1,
        "msg_id": mid,
        "msg_type": "GOSSIP",
        "sender_id": "injector",
        "sender_addr": "127.0.0.1:9999",
        "timestamp_ms": t0,
        "ttl": ttl,
        "payload": {
            "topic": "phase3",
            "data": "phase3-measurement",
            "origin_id": "injector",
            "origin_timestamp_ms": t0
        }
    }
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.sendto(json.dumps(msg).encode("utf-8"), ("127.0.0.1", bootstrap_port))
    s.close()
    return t0
```

این تابع یک پیام GOSSIP را (با msg_id یکتا) می‌سازد و به bootstrap (127.0.0.1:port) می‌فرستد و همچنین زمان تزریق t0 را برمی‌گرداند تا مبنای محاسبه پارامترها باشد.

تابع parse_run هسته تحلیل است:

```python
def parse_run(run_dir: Path, n: int, msg_id: str, t0_ms: int) -> Dict:  1 usage
    first_delivery_by_node: Dict[str, int] = {}
    send_events: List[Tuple[int, str]] = []

    for lf in sorted(run_dir.glob("n*.log")):
        with lf.open( mode: "r", encoding="utf-8", errors="ignore") as f:
            for line in f:
                try:
                    o = json.loads(line)
                except Exception:
                    continue

                ev = o.get("event")
                ts = o.get("ts_ms")
                if not isinstance(ts, int):
                    continue

                if ev == "gossip_delivered" and o.get("msg_id") == msg_id:
                    nid = o.get("node_id") or lf.name
                    if nid not in first_delivery_by_node or ts < first_delivery_by_node[nid]:
                        first_delivery_by_node[nid] = ts

                if ev == "send":
                    mt = o.get("msg_type")
                    if mt in MSG_TYPES_FOR_OVERHEAD:
                        send_events.append((ts, mt))
```

این تابع لاگ‌های همان اجرا را می‌خواند و اولین زمان دریافت gossip_delivered را برای هر گره پیدا می‌کند و تمام رخدادهای send از نوع پیام‌های تعریف شده را جمع می‌کند.

```
    delivered = len(first_delivery_by_node)
    threshold = math.ceil(0.95 * n)
    reached_95 = delivered >= threshold

    t95_ms = None
    conv_ms = None
    overhead = None

    if reached_95:
        sorted_ts = sorted(first_delivery_by_node.values())
        t95_ms = sorted_ts[threshold - 1]
        conv_ms = t95_ms - t0_ms
        overhead = sum(1 for ts, _ in send_events if t0_ms <= ts <= t95_ms)

    return {
        "delivered_nodes": delivered,
        "delivery_ratio": delivered / n,
        "threshold_95": threshold,
        "reached_95": reached_95,
        "t0_ms": t0_ms,
        "t95_ms": t95_ms,
        "convergence_ms": conv_ms,
        "overhead": overhead,
    }
```

سپس آستانه ۹۵٪ را حساب می‌کند:

$$threshold = \lfloor 0.95 \times N \rfloor$$

اگر ۹۵٪ پوشش حاصل شده باشد، با استفاده از t95 و t0 مقدار convergence را محاسبه می‌کند و تعداد پیام‌های send را در این بازه جمع می‌کند تا overhead به‌دست بیاید. همچنین اگر پوشش حاصل نشده باشد این معیارها None می‌مانند.

```
def summarize(rows: List[Dict]) -> List[Dict]:  1 usage
    grouped: Dict[Tuple[str, int], List[Dict]] = {}
    for r in rows:
        key = (r["config"], r["N"])
        grouped.setdefault(key, []).append(r)

    out = []
    for (cfg, n), vals in sorted(grouped.items(), key=lambda x: (x[0][0], x[0][1])):
        conv = [v["convergence_ms"] for v in vals if v["convergence_ms"] is not None]
        ov = [v["overhead"] for v in vals if v["overhead"] is not None]
        pass_rate = sum(1 for v in vals if v["reached_95"]) / len(vals)

        out.append({
            "config": cfg,
            "N": n,
            "runs": len(vals),
            "pass_rate_95": pass_rate,
            "conv_mean_ms": statistics.mean(conv) if conv else None,
            "conv_std_ms": statistics.pstdev(conv) if len(conv) > 1 else 0.0 if conv else None,
            "over_mean": statistics.mean(ov) if ov else None,
            "over_std": statistics.pstdev(ov) if len(ov) > 1 else 0.0 if ov else None,
        })
    return out
```

این تابع نتایج خام را بر اساس (config, N) گروه‌بندی می‌کند. سپس برای هر گروه مقادیر زیر را به‌دست می‌آورد:

Pass_rate_95, conv_mean_ms, conv_std_ms, over_mean, over_std

```python
def write_csv(path: Path, rows: List[Dict], fieldnames: List[str]) -> None:  2 usages
    with path.open( mode: "w", newline="", encoding="utf-8") as f:
        w = csv.DictWriter(f, fieldnames=fieldnames)
        w.writeheader()
        for r in rows:
            w.writerow(r)
```

این تابع نتایج در CSV ذخیره می‌کند.

تابع اصلی main به صورت زیر است:

```python
def main():  1 usage
    ap = argparse.ArgumentParser()
    ap.add_argument( *name_or_flags: "--node-file", type=str, default="node.py")
    ap.add_argument( *name_or_flags: "--base-port", type=int, default=9000)
    ap.add_argument( *name_or_flags: "--ns", type=str, default="10,20,50")
    ap.add_argument( *name_or_flags: "--seeds", type=str, default="42,43,44,45,46")
    ap.add_argument( *name_or_flags: "--out", type=str, default="results/phase3")
    args = ap.parse_args()

    ns = [int(x.strip()) for x in args.ns.split(",") if x.strip()]
    seeds = [int(x.strip()) for x in args.seeds.split(",") if x.strip()]
    out_dir = Path(args.out)
    out_dir.mkdir(parents=True, exist_ok=True)

    configs = [
        Config("base", fanout=3, ttl=8, peer_limit=20, ping_interval=2.0, peer_timeout=6.0),
        Config("fanout5", fanout=5, ttl=8, peer_limit=20, ping_interval=2.0, peer_timeout=6.0),
        Config("ttl12", fanout=3, ttl=12, peer_limit=20, ping_interval=2.0, peer_timeout=6.0),
        Config("peerlimit10_timeout10", fanout=3, ttl=8, peer_limit=10, ping_interval=2.0, peer_timeout=10.0),
    ]
```

ابتدا آرگومان‌ها را می‌خوانیم و config های مقایسه‌ای را تعریف می‌کنیم.

```python
raw_rows: List[Dict] = []

for cfg in configs:
    for n in ns:
        for sd in seeds:
            run_tag = f"{cfg.name}_N{n}_seed{sd}"
            run_dir = out_dir / "runs" / run_tag
            run_dir.mkdir(parents=True, exist_ok=True)

            subprocess.run( args: ["pkill", "-f", f"python3 {args.node_file}"], check=False, stdout=subprocess.DEVNULL,
                           stderr=subprocess.DEVNULL)
            time.sleep(1)

            procs = []
            try:
                procs.append(start_node(args.node_file, args.base_port, cfg, sd, run_dir / "n0.log",  bootstrap: None))
                for i in range(1, n):
                    port = args.base_port + i
                    procs.append(start_node(args.node_file, port, cfg, sd + i,
                                            run_dir / f"n{i}.log",  bootstrap: f"127.0.0.1:{args.base_port}"))

                time.sleep(2)
                alive = all(p.poll() is None for p in procs)
                if not alive:
                    raw_rows.append({
                        "config": cfg.name, "N": n, "seed": sd, "status": "node_crash",
                        "delivered_nodes": 0, "delivery_ratio": 0.0, "reached_95": False,
                        "convergence_ms": None, "overhead": None
                    })
                    continue
```

سپس یک حلقه سه لایه روی config ها، N ها و seed ها می‌زنیم و برای هر اجرا:

- پردازه‌های قبلی را پاکسازی می‌کنیم.
- به تعداد N گره بالا می‌آوریم.
- برای گره‌ها health-check انجام می‌دهیم.

```
                wb, wp = wait_by_n(n)
                time.sleep(wb)

                mid = str(uuid.uuid4())
                t0 = inject_gossip(mid, args.base_port, cfg.ttl)
                time.sleep(wp)

                metrics = parse_run(run_dir, n, mid, t0)
                raw_rows.append({
                    "config": cfg.name,
                    "N": n,
                    "seed": sd,
                    "status": "ok",
                    "msg_id": mid,
                    **metrics
                })
            finally:
                stop_all(procs)
                time.sleep(1)

raw_fields = [
    "config", "N", "seed", "status", "msg_id",
    "delivered_nodes", "delivery_ratio", "threshold_95", "reached_95",
    "t0_ms", "t95_ms", "convergence_ms", "overhead"
]
write_csv(out_dir / "raw_results.csv", raw_rows, raw_fields)
```

- برای bootstrap منتظر می‌مانیم.
- یک GOSSIP تزریق می‌کنیم.
- لاگ‌ها را تحلیل می‌کنیم.

```
summary_rows = summarize(raw_rows)
sum_fields = [
    "config", "N", "runs", "pass_rate_95",
    "conv_mean_ms", "conv_std_ms",
    "over_mean", "over_std"
]
write_csv(out_dir / "summary_results.csv", summary_rows, sum_fields)

print("Done.")
print(f"Raw: {out_dir / 'raw_results.csv'}")
print(f"Summary: {out_dir / 'summary_results.csv'}")
```

- نتایج را ذخیره می‌کنیم.
- تمام گره‌ها را خاموش می‌کنیم.

در ادامه کد را اجرا می‌کنیم. نتایج به صورت زیرند:

مقدار pass_rate_95 در چندین حالت از ۱ کمتر است که در این اجراها میانگین convergence محاسبه نمی‌شود. نتایج آزمایش در فایل‌های csv ذخیره شده‌اند:

	config	N	runs	pass_rate_95	conv_mean_ms	conv_std_ms	over_mean	over_std
1	base	10	5	0.2	4	0.0	30	0.0
2	base	20	5	0.6	8	0.816496580927726	57	2.449489742783178
3	base	50	5	0.4	16.5	1.5	162.5	7.5
4	fanout5	10	5	0.8	2.75	1.299038105676658	48.75	2.165063509461097
5	fanout5	20	5	0.8	12	4.301162633521313	88	7.14142842854285
6	fanout5	50	5	0.6	10.666666666666666	1.247219128924647	260.3333333333333	9.46337971105226
7	peerlimit10_timeout10	10	5	0.8	6.5	1.5	29.25	0.82915619758885
8	peerlimit10_timeout10	20	5	0.8	10.5	3.201562118716424 3	52.5	3.840572873934304
9	peerlimit10_timeout10	50	5	0.4	19.5	2.5	166	12.0
10	ttl12	10	5	0.4	4.5	0.5	26.5	0.5
11	ttl12	20	5	0.4	7	0.0	57	0.0
12	ttl12	50	5	0.6	13.666666666666666	11.32352516764202	161.66666666666666	20.741798914805393

تحلیل اثر پارامترها:

اندازه شبکه (N)

با افزایش N سربار پیام در همه حالت‌ها رشد قابل توجهی دارد و زمان همگرایی نیز بدتر می‌شود، البته به علت تصادفی بودن نوسان وجود دارد.

اندازه fanout

افزایش fanout نرخ پاس شدن و پوشش را بهتر می‌کند، اما overhead را به میزان زیادی افزایش می‌دهد.

مقدار TTL

افزایش TTL در N های کوچک بهبود محسوسی در پوشش نمی‌دهد، اما در N=50 پوشش بهتر شده ولی لزوما افزایش TTL به تنهایی تضمین‌کننده عملکرد بهتر نیست.

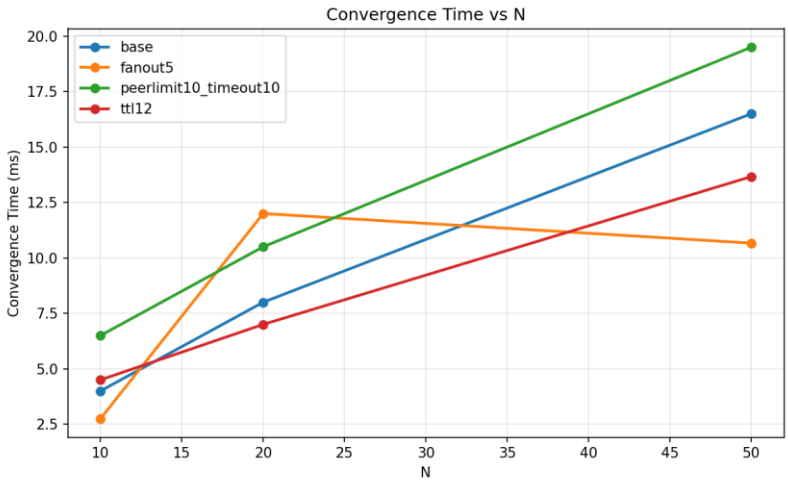
سیاست همسایه‌ها (peer_timeout و peer_limit)

در N=10 و N=20 نرخ پاس شدن خوب و overhead نسبتا کم است، اما در N=50 افت واضحی رخ داده است. در نتیجه محدودیت شدید همسایه برای شبکه‌های کوچک مناسب است، اما برای شبکه بزرگ باعث افت پوشش و کندی انتشار می‌شود.

در نتیجه:

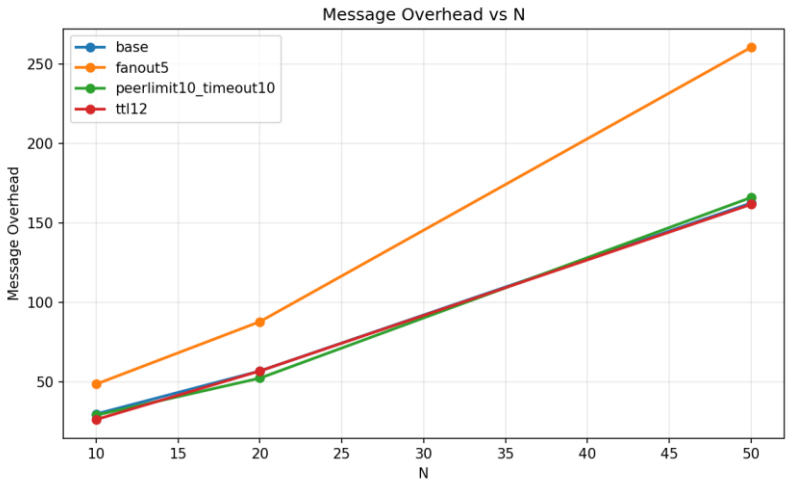
- اگر اولویت پوشش ۹۵٪ باشد، fanout5 بهترین پایداری نسبی را با overhead زیاد دارد.
- اگر اولویت سربار کمتر باشد، base یا سیاست برای N کوچک مناسب‌ترند.

در ادامه با استفاده از یک اسکریپت، از روی فایل‌های csv نمودارها را رسم می‌کنیم:



شکل 1 زمان همگرایی برحسب اندازه شبکه

این نمودار میانگین زمان همگرایی را برای شبکه‌های مختلف در اندازه‌های مختلف نشان می‌دهد. محور افقی N و محور عمودی convergence time برحسب میلی‌ثانیه را نمایش می‌دهد. این میانگین‌ها فقط روی اجراهایی محاسبه شده‌اند که به آستانه ۹۵٪ پوشش رسیده‌اند.



شکل 2 سربار پیام برحسب اندازه شبکه

این نمودار میانگین زمان سربار پیام را برای شبکه‌های مختلف در اندازه‌های مختلف نشان می‌دهد. محور افقی N و محور عمودی تعداد کل پیام های ارسالی تا زمان رسیدن به ۹۵٪ پوشش را نمایش می‌دهد. این میانگین‌ها فقط روی اجراهایی محاسبه شده‌اند که به آستانه ۹۵٪ پوشش رسیده‌اند.

فاز چهارم

بخش اول)

1- تغییرات اعمال‌شده در node.py (طراحی و پیاده‌سازی Hybrid)

در نسخه‌یPush-only ، هر گره هنگام اولین دریافت یک پیام GOSSIP آن را برای fanout همسایه‌ها فوروارد می‌کند، بدون اینکه از وضعیت دیدن پیام توسط همسایه‌ها اطلاعی داشته باشد.. این کار سریع است اما باعث ارسال تکراری و سربار بالا می‌شود. برای کاهش ارسال‌های بی‌فایده، مکانیزم Hybrid Push-Pull در این بخش اضافه شد که شامل دو پیام کنترلی جدید است:

- **IHAVE**: اعلان «شناسه‌ی پیام‌های جدید» بدون ارسال payload کامل
- **IWANT**: درخواست «شناسه‌های دیده‌نشده» از فرستنده‌یIHAVE

برای پیاده سازی این قسمت ابتدا سه پارامتر جدید به آرگومان‌های خط فرمان اضافه شد:

- --mode {push,hybrid}: فعال/غیرفعال کردنHybrid
- --pull-interval: فاصله‌ی زمانی ارسالIHAVE(مثلاً هر ۱ یا ۲ ثانیه)
- --ihave-max-ids: سقف تعداد msg_idهایی که در هر IHAVE قرار می‌گیرد (مثلاً ۳۲)

و همچنین برای اینکه در Hybrid فقط شناسه‌ها اعلان شوند و در صورت نیاز پیام کامل ارسال شود، دو حافظه‌ی جدید اضافه شد:

store: Dict[msg_id -> full_gossip_msg]

وقتی یک GOSSIP دریافت یا ساخته می‌شود، نسخه‌ی کامل آن ذخیره می‌شود تا اگر همسایه IWANT فرستاد، بتوان پیام کامل را ارسال کرد.

recent_gossip: OrderedDict[msg_id -> ts]

و همچنین لیست پیام‌های اخیر (با ظرفیت محدود) برای ساخت payload پیامIHAVE نیز نگه داشته میشود و این لیست به صورت bounded نگه داشته می‌شود تا مصرف حافظه کنترل شود.

همچنین یک timestamp به نام last_ihave_ts_ms نگه داشته شد تا IHAVE تکراری برای پیام‌های قدیمی تولید نشود و فقط شناسه‌های جدید از آخرین اعلان ارسال شوند. برای ارسال دوره ای یک لوپ (ihave_loop) اضافه شد که هر pull_interval یکبار اگر در مود hybrid باشیم لیست recent_gossip را بررسی می‌کند و تنها شناسه‌هایی که از زمان آخرین اعلان اضافه شده‌اند انتخاب می‌شوند و طول لیست را با ihave_max_ids محدود می‌کند و سپس به تعداد pull_fanout همسایه، پیام IHAVE می‌فرستد(با payload فقط شناسه ها) و سپس وقتی یک گره پیام IHAVE را دریافت می‌کند ids را با Seen Set (seen_gossip در کد)مقایسه می‌کند و هر msg_id که در seen نباشد، در لیست missingقرار می‌گیرد و اگر missingخالی نبود، پیام IWANT با payload شامل همین missingارسال می‌شود:

```
    async def handle_ihave(self, msg: dict, addr: Addr):
        if getattr(self.cfg, "mode", "push") != "hybrid":
            return

        payload = msg.get("payload", {}) or {}
        ids = payload.get("ids", []) or []
        if not isinstance(ids, list):
            return

        missing = []
        for mid in ids:
            if isinstance(mid, str) and mid and (mid not in self.seen_gossip):
                missing.append(mid)
```

```
        if not missing:
            self.log("ihave_no_missing", from_addr=addr_to_str(addr),
n_ids=len(ids))
            return

        out = self.pack("IWANT", {"ids": missing}, ttl=1)
        out_mid = json.loads(out.decode("utf-8")).get("msg_id", "")
        self.log("iwant_sent", to=addr_to_str(addr), n_ids=len(missing))
        await self.sendto(out, addr, {"msg_type": "IWANT", "msg_id": out_mid})
```

به صورت مشابه وقتی یک گره پیام IWANT دریافت می‌کند برای هر msg_id درخواست‌شده اگر در store موجود بود، همان پیام کامل GOSSIPرا (با sender جدید و timestamp جدید)برای درخواست‌کننده ارسال می‌کند و اگر موجود نبود، آن id را نادیده می‌گیرد (چون ممکن است فرستنده‌ی IHAVE خودش پیام را داشته ولی ما هنوز ذخیره نکرده باشیم یا محدودیت حافظه باعث حذف شده باشد) پس در نتیجه payload کامل فقط زمانی منتقل می‌شود که گیرنده واقعاً پیام را نداشته باشد:

```
    async def handle_iwant(self, msg: dict, addr: Addr):
        if getattr(self.cfg, "mode", "push") != "hybrid":
            return

        payload = msg.get("payload", {}) or {}
        ids = payload.get("ids", []) or []
        if not isinstance(ids, list):
            return

        served = 0
        for mid in ids:
            if not isinstance(mid, str) or not mid:
                continue

            g = self.store.get(mid)
            if not g:
                continue

            reply = dict(g)
            reply["sender_id"] = self.node_id
            reply["sender_addr"] = addr_to_str(self.self_addr)
            reply["timestamp_ms"] = now_ms()
            reply["ttl"] = int(self.cfg.ttl)

            out = json.dumps(reply).encode("utf-8")
            await self.sendto(out, addr, {
                "msg_type": "GOSSIP",
                "msg_id": mid,
                "ttl": reply["ttl"],
```

```
                "reason": "iwant_reply"
            })
            served += 1

            if mid not in self.seen_gossip:
                self.seen_gossip.add(mid, now_ms())

        self.log("iwant_served", to=addr_to_str(addr), asked=len(ids),
served=served)
```

همچنین برای اینکه مقایسه‌ی منصفانه انجام شود در حالت push، handlerهای IHAVE/IWANT عملاً هیچ کاری نمی‌کنند و این پیام‌های کنترلی فقط در mode=hybrid فعال‌اند. لازم به ذکر است که در حالتHybrid ، مکانیزم Push حذف نشده و همچنان انتشار اولیه پیام بر اساس Push انجام می‌شود و Pull صرفاً به عنوان مکانیزم تکمیلی برای بازیابی پیام‌های از دست رفته عمل می‌کند.

2- توضیح کد تست(Phase4.py)

برای مقایسه‌ی Push-only وHybrid Push-Pull ، یک اسکریپت تست نوشته شد که دقیقاً سناریوی پروژه را اجرا می‌کند و برای اندازه شبکه: $N \in \{10, 20, 50, 100\}$ و برای چند اجرای تکراری با seed های متفاوت، شبکه را روی localhost بالا می‌آورد، یک پیام GOSSIP را تزریق می‌کند و تا رسیدن پیام به حداقل ۹۵٪ گره‌ها صبر می‌کند (یا در صورت عدم تحقق، تایم‌اوت می‌دهد). در نهایت، خروجی خام و خلاصهٔ آماری (میانگین و انحراف معیار) و همچنین نمودارهای مقایسه‌ای تولید می‌شود. برای اینکه مقایسهٔ Push و Hybrid منصفانه باشد، پارامترهای اصلی شبکه در یک دیتاکلاس به نام RunConfig نگهداری می‌شوند. این ساختار شامل پارامترهای مشترک (مثل fanout, ttl, peer_limit, ping_interval, peer_timeout, get_peers_interval, get_peers_fanout) و پارامترهای مخصوص Hybrid (مثل pull_interval, ihave_max_ids, pull_fanout) است. همچنین pow_k در این بخش صفر نگه داشته شده تا تمرکز صرفاً روی Push-Pull باشد.

تابع start_node هر نود را به صورت یک پردازهٔ مستقل با subprocess.Popen اجرا می‌کند و خروجی استاندارد نود (stdout + stderr) را داخل فایل لاگ اختصاصی همان نود می‌ریزد(n0.log, n1.log, ...) . پارامترهای CLI نود از روی cfg ساخته می‌شوند تا دقیقاً همان تنظیمات در هر ران اعمال شود. برای جلوگیری از تداخل ورودی کاربر، stdin نودها به DEVNULL متصل شده است:

```
def start_node(
    node_file: str,
    port: int,
    cfg: RunConfig,
    seed: int,
    log_path: Path,
    bootstrap: Optional[str],
) -> Tuple[subprocess.Popen, object]:
    cmd = [
        "python3", node_file,
        "--port", str(port),
        "--fanout", str(cfg.fanout),
        "--ttl", str(cfg.ttl),
        "--peer-limit", str(cfg.peer_limit),
        "--ping-interval", str(cfg.ping_interval),
        "--peer-timeout", str(cfg.peer_timeout),
```

```
        "--seed", str(seed),
        "--get-peers-interval", str(cfg.get_peers_interval),
        "--get-peers-fanout", str(cfg.get_peers_fanout),
        "--mode", cfg.mode,
        "--pull-interval", str(cfg.pull_interval),
        "--ihave-max-ids", str(cfg.ihave_max_ids),
        "--pull-fanout", str(cfg.pull_fanout),
        "--pow-k", str(cfg.pow_k),
    ]
    if bootstrap:
        cmd += ["--bootstrap", bootstrap]

    f = log_path.open("w", encoding="utf-8")
    p = subprocess.Popen(
        cmd,
        stdout=f,
        stderr=subprocess.STDOUT,
        stdin=subprocess.DEVNULL,
        text=True,
        bufsize=1,
        preexec_fn=os.setsid if os.name != "nt" else None
    )
    return p, f
```

تابع stop_all در پایان هر ران تلاش می‌کند تا هیچ پروسه‌ای از ران قبلی باقی نماند و باعث تداخل روی پورت‌ها نشود:

```
def stop_all(procs: List[Tuple[subprocess.Popen, object]]) -> None:
    for p, f in procs:
        try:
            if p.poll() is None:
                if os.name == "nt":
                    p.terminate()
                else:
                    os.killpg(os.getpgid(p.pid), signal.SIGINT)
        except Exception:
            pass

    time.sleep(0.8)

    for p, f in procs:
        try:
            if p.poll() is None:
                if os.name == "nt":
                    p.kill()
```

```
                else:
                    os.killpg(os.getpgid(p.pid), signal.SIGKILL)
        except Exception:
            pass
        try:
            f.flush()
            f.close()
        except Exception:
            pass
```

برای شروع آزمایش، به جای ارسال از طریق stdin (که در حالت غیرتعاملی مشکل‌ساز است)، یک پیام GOSSIP با UDP به نود بوت‌استرپ تزریق می‌شود. این پیام شامل msg_id تصادفی و timestamp تولید (t0) است. مقدار t0 همان زمان شروع انتشار پیام در شبکه محسوب می‌شود و مبنای محاسبهٔ convergence قرار می‌گیرد .یکی از مشکلات اصلی در آزمایش‌های خودکار این است که اگر هنوزPeer List ها کامل نشده باشد، انتشار پیام ممکن است به ۹۵٪ نرسد و ران تایم‌اوت بخورد. برای کاهش این مشکل، قبل از تزریق پیام، یک مرحله Warm-up تطبیقی انجام می‌شود، در واقع با بررسی تعداد رخدادهای peer_added در لاگ هر نود، تخمینی از میزان اتصال شبکه به دست می‌آید(peer_counts) و سپس تابع has_enough_connectivity بررسی می‌کند آیا حداقل ratio_ok از نودها حداقل min_peers_per_node همسایه اضافه کرده‌اند یا نه و اگر شرط برقرار نباشد، اسکریپت مرحلهٔ warm-up را با گام‌های زمانی (warmup_step_s) ادامه می‌دهد تا به سقف warmup_max_s برسد. علاوه بر این، مکانیزم warmup_retry اضافه شده است: اگر بعد از یک warm-up کامل هنوز شبکه به حد کافی شکل نگرفته باشد، چند بار دیگر هم کمی بیشتر صبر می‌کند (بدون گیرکردن بی‌نهایت). این کار احتمال تایم‌اوت ناشی از تشکیل ناقص شبکه را کم می‌کند. هدف این بخش این است که در زمان تزریق پیام، شبکه واقعاً آماده‌تر باشد و مقایسه Push و Hybrid منصفانه‌تر و پایدارتر انجام شود.

پس از تزریق پیام، اسکریپت به صورت polling و دوره‌ای (delivery_poll_s) لاگ‌ها را می‌خواند و دو معیار اصلی را محاسبه می‌کند:

1. Convergence Time(زمان همگرایی تا 95٪): برای هر نود، اولین timestamp رخداد gossip_delivered برای همان msg_id استخراج می‌شود. تعداد نودهای دریافت‌کننده شمارش می‌شود و آستانهٔ ۹۵٪ برابر ceil(0.95*N) محاسبه می‌شود و t95 برابر زمانی است که **آستانهٔ** ۹۵٪-اُمین نود پیام را دریافت کرده است و در نهایت convergence_ms = t95 - t0.
2. Message Overhead(سربار پیام): همهٔ رخدادهای send(شامل دیتا و کنترل) شمارش می‌شوند، اما فقط برای انواع پیام‌های مشخص‌شده در MSG_TYPES_FOR_OVERHEAD(GOSSIP, HELLO, GET_PEERS, PEERS_LIST, PING, PONG, IHAVE, IWANT). overhead برابر تعداد کل sendها در بازهٔ زمانی [t0, t95] است.

و اگر تا زمان delivery_max_s هنوز ۹۵٪ دریافت نشده باشد، ران با وضعیت timeout ثبت می‌شود.

```
def run_one(
    node_file: str,
    base_port: int,
```

```
    N: int,
    seed: int,
    cfg: RunConfig,
    run_dir: Path,
    warmup_initial_s: float,
    warmup_step_s: float,
    warmup_max_s: float,
    warmup_retry: int,
    min_peers_per_node: int,
    ratio_ok: float,
    delivery_poll_s: float,
    delivery_max_s: float,
) -> Dict:
    run_dir.mkdir(parents=True, exist_ok=True)

    procs: List[Tuple[subprocess.Popen, object]] = []
    try:
        p0, f0 = start_node(node_file, base_port, cfg, seed, run_dir / "n0.log",
bootstrap=None)
        procs.append((p0, f0))

        boot = f"127.0.0.1:{base_port}"
        for i in range(1, N):
            port = base_port + i
            pi, fi = start_node(node_file, port, cfg, seed + i, run_dir /
f"n{i}.log", bootstrap=boot)
            procs.append((pi, fi))

        time.sleep(0.5)
        if not all(p.poll() is None for p, _ in procs):
            return {"ok": False, "reason": "node_crash_early"}
        waited = 0.0
        attempt = 0

        while True:
            time.sleep(warmup_initial_s)
            waited += warmup_initial_s

            while waited < warmup_max_s:
                if has_enough_connectivity(run_dir, N, min_peers_per_node,
ratio_ok):
                    break
                time.sleep(warmup_step_s)
                waited += warmup_step_s
```

```
            if has_enough_connectivity(run_dir, N, min_peers_per_node, ratio_ok):
                break

            attempt += 1
            if attempt > warmup_retry:
                break
            time.sleep(warmup_step_s * 2)
            waited += warmup_step_s * 2

        mid = str(uuid.uuid4())
        t0 = inject_gossip(mid, base_port, cfg.ttl)

        t_start = time.time()
        last_del = 0
        last_need = math.ceil(0.95 * N)

        while True:
            prog = parse_progress(run_dir, N, mid, t0)

            if prog["done"]:
                return {
                    "ok": True,
                    "status": "ok",
                    "mode": cfg.mode,
                    "N": N,
                    "seed": seed,
                    "msg_id": mid,
                    "warmup_wait_s": waited,
                    "delivered_nodes": prog["delivered_nodes"],
                    "threshold_95": prog["threshold_95"],
                    "reached_95": True,
                    "t0_ms": t0,
                    "t95_ms": prog["t95_ms"],
                    "convergence_ms": prog["convergence_ms"],
                    "overhead": prog["overhead"],
                }

            last_del = prog["delivered_nodes"]
            last_need = prog["threshold_95"]

            if time.time() - t_start > delivery_max_s:
                return {
                    "ok": False,
                    "status": "timeout",
```

```
                    "reason":
f"timeout_before_t95_delivered_{last_del}_need_{last_need}",
                    "mode": cfg.mode,
                    "N": N,
                    "seed": seed,
                    "msg_id": mid,
                    "warmup_wait_s": waited,
                    "delivered_nodes": last_del,
                    "threshold_95": last_need,
                    "reached_95": False,
                    "t0_ms": t0,
                    "t95_ms": None,
                    "convergence_ms": None,
                    "overhead": None,
                }

            time.sleep(delivery_poll_s)

    finally:
        stop_all(procs)
        time.sleep(0.3)
```

در main دو کانفیگ ساخته می‌شود:

1. cfg_push با mode="push"
2. cfg_hybrid با mode="hybrid"

تمام پارامترهای دیگر مشترک هستند تا مقایسه منصفانه باشد. سپس برای هر N و هر seed ، یک ران انجام می‌شود. برای جلوگیری از تداخل پورت‌ها بین ران‌های مختلف و بین حالتpush/hybrid ، base_port به صورت محاسبه‌شده تغییر می‌کند و برای هر ران خروجی خام در raw_rows ذخیره می‌شود و فایل‌های لاگ در مسیر out/runs/<mode>_N... ایجاد می‌شوند:

```
def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--node-file", type=str, default="node.py")
    ap.add_argument("--out", type=str, default="results_phase4_compare")
    ap.add_argument("--ns", type=str, default="5,10,20,50,100")
    ap.add_argument("--seeds", type=str, default="42,43,44,45,46")
    ap.add_argument("--base-port", type=int, default=12000)

    ap.add_argument("--warmup-initial-s", type=float, default=2.0)
    ap.add_argument("--warmup-step-s", type=float, default=2.0)
    ap.add_argument("--warmup-max-s", type=float, default=45.0)
    ap.add_argument("--warmup-retry", type=int, default=2)
```

```
    ap.add_argument("--min-peers-per-node", type=int, default=3)
    ap.add_argument("--ratio-ok", type=float, default=0.7)

    ap.add_argument("--delivery-poll-s", type=float, default=0.25)
    ap.add_argument("--delivery-max-s", type=float, default=90.0)

    args = ap.parse_args()

    ns = [int(x.strip()) for x in args.ns.split(",") if x.strip()]
    seeds = [int(x.strip()) for x in args.seeds.split(",") if x.strip()]

    out_dir = Path(args.out)
    out_dir.mkdir(parents=True, exist_ok=True)

    common = dict(
        fanout=5,
        ttl=14,
        peer_limit=120,
        ping_interval=1.0,
        peer_timeout=10.0,
        get_peers_interval=0.7,
        get_peers_fanout=4,
        pull_interval=2.0,
        ihave_max_ids=64,
        pull_fanout=4,
        pow_k=0,
    )

    cfg_push = RunConfig(mode="push", **common)
    cfg_hybrid = RunConfig(mode="hybrid", **common)
    all_cfgs = [cfg_push, cfg_hybrid]

    raw_rows: List[Dict] = []

    for cfg in all_cfgs:
        for N in ns:
            for i, sd in enumerate(seeds):
                mode_off = 0 if cfg.mode == "push" else 500
                base_port = args.base_port + mode_off + (i * 200) + (N * 5)

                run_tag = f"{cfg.mode}_N{N}_seed{sd}"
                run_dir = out_dir / "runs" / run_tag

                print(f"[RUN] mode={cfg.mode} N={N} seed={sd}
base_port={base_port}", flush=True)
```

```
                res = run_one(
                    node_file=args.node_file,
                    base_port=base_port,
                    N=N,
                    seed=sd,
                    cfg=cfg,
                    run_dir=run_dir,
                    warmup_initial_s=args.warmup_initial_s,
                    warmup_step_s=args.warmup_step_s,
                    warmup_max_s=args.warmup_max_s,
                    warmup_retry=args.warmup_retry,
                    min_peers_per_node=args.min_peers_per_node,
                    ratio_ok=args.ratio_ok,
                    delivery_poll_s=args.delivery_poll_s,
                    delivery_max_s=args.delivery_max_s,
                )
                raw_rows.append(res)

                if res.get("ok"):
                    print(f"  OK  warmup={res['warmup_wait_s']:.1f}s
conv={res['convergence_ms']/1000:.3f}s overhead={res['overhead']}", flush=True)
                else:
                    print(f"  FAIL reason={res.get('reason')}", flush=True)

    raw_fields = [
        "ok", "status", "reason",
        "mode", "N", "seed", "msg_id",
        "warmup_wait_s",
        "delivered_nodes", "threshold_95", "reached_95",
        "t0_ms", "t95_ms", "convergence_ms", "overhead",
    ]
    write_csv(out_dir / "raw_results.csv", raw_rows, raw_fields)

    summary_rows = summarize(raw_rows)
    sum_fields = [
        "mode", "N", "runs", "success_runs", "pass_rate_95",
        "conv_mean_ms", "conv_std_ms",
        "over_mean", "over_std"
    ]
    write_csv(out_dir / "summary_results.csv", summary_rows, sum_fields)

    p1, p2 = plot(summary_rows, out_dir)

    print("\nDone.")
```

```
    print(f"Raw:     {out_dir / 'raw_results.csv'}")
    print(f"Summary: {out_dir / 'summary_results.csv'}")
    print(f"Plots:   {p1} , {p2}")
```

معتبر) وارد محاسبهٔ میانگین و انحراف معیار می‌شوند. overheadو convergence_ms و مقادیرreached_95=True فقط ران‌های موفق(یعنی summarizeدر تابع میانگین را خراب نمی‌کند: None شود، با مقدار صفر یا timeout به این ترتیب اگر یک ران

```
def summarize(rows: List[Dict]) -> List[Dict]:
    grouped: Dict[Tuple[str, int], List[Dict]] = {}
    for r in rows:
        key = (r["mode"], r["N"])
        grouped.setdefault(key, []).append(r)

    out = []
    for (mode, n), vals in sorted(grouped.items(), key=lambda x: (x[0][0], x[0][1])):
        succ = [
            v for v in vals
            if v.get("reached_95")
            and v.get("convergence_ms") is not None
            and v.get("overhead") is not None
        ]

        conv = [v["convergence_ms"] for v in succ if isinstance(v.get("convergence_ms"), (int, float)) and v["convergence_ms"] > 0]
        ov = [v["overhead"] for v in succ if isinstance(v.get("overhead"), (int, float)) and v["overhead"] > 0]

        pass_rate = sum(1 for v in vals if v.get("reached_95")) / len(vals)
        success_runs = len(succ)

        out.append({
            "mode": mode,
            "N": n,
            "runs": len(vals),
            "success_runs": success_runs,
            "pass_rate_95": pass_rate,
            "conv_mean_ms": statistics.mean(conv) if conv else None,
            "conv_std_ms": statistics.pstdev(conv) if len(conv) > 1 else (0.0 if conv else None),
            "over_mean": statistics.mean(ov) if ov else None,
            "over_std": statistics.pstdev(ov) if len(ov) > 1 else (0.0 if ov else None),
        })
    return out
```

در پایان، دو نمودار تولید می‌شود:

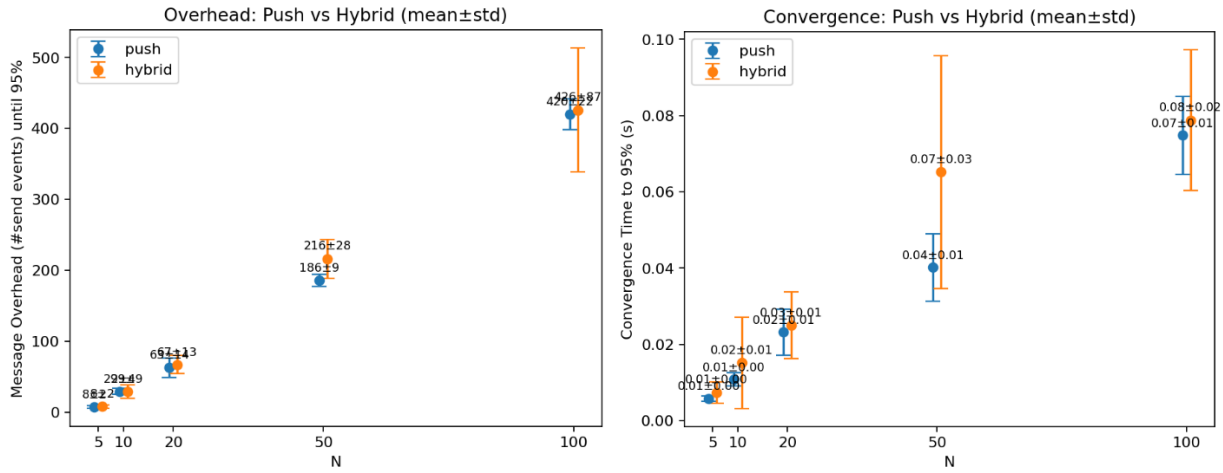
- **Convergence Push vs Hybrid (mean±std)**
- **Overhead Push vs Hybrid (mean±std)**

اسکریپت در مسیر خروجی (out--) خروجی های اصلی زیر را تولید می‌کند:

raw_results.csv: نتایج تک‌به‌تک ران‌ها(موفق/ناموفق، زمان همگرایی، overhead، ...)

summary_results.csv: خلاصه آماری به ازای هر N و هر mode (میانگین و انحراف معیار)

با ران کردن این تست روی کد دو تصویر زیر از روی خروجی های فایل جیسون ساخته شده:



این نتایج نشان می‌دهد که در اندازه‌های مختلف شبکه(N=5تاN=100)، در اغلب موارد زمان همگرایی در حالت Push-only برابر یا حتی کمتر از حالت Hybrid بوده است، در حالی که سربار پیام در حالت Hybrid معمولاً بیشتر بوده است.

این رفتار در نگاه اول ممکن است غیرمنتظره به نظر برسد، زیرا از نظر تئوریک انتظار می‌رود افزودن مکانیزم Pull باعث کاهش tail و بهبود سرعت رسیدن به درصد بالای پوشش شود. با این حال، بررسی دقیق‌تر شرایط آزمایش نشان می‌دهد که محیط اجرا (localhost، بدونpacket loss ، با latency بسیار کم و اتصال مناسب بین نودها) عملاً یک شبکه تقریباً ایده‌آل ایجاد کرده است. در چنین شرایطی، مکانیسم Push به تنهایی قادر است پیام را با سرعت بالا و تقریباً بدون اتلاف به کل شبکه منتشر کند. در نتیجه، عملاً tail قابل توجهی ایجاد نمی‌شود تا Pull بتواند آن را بهبود دهد.

در حالتHybrid ، علاوه بر پیام‌های GOSSIP مربوط بهPush ، پیام‌های کنترلی اضافی نظیر IHAVE و IWANT نیز تولید می‌شوند. حتی اگر پیام‌های تکراری به دلیل مکانیزم seen_all مجدداً منتشر نشوند، خود این پیام‌های کنترلی باعث افزایش سربار کل شبکه می‌شوند. بنابراین در سناریوی فعلی، Hybrid بدون آن‌که بهبود محسوسی در زمان همگرایی ایجاد کند، سربار بیشتری نسبت به Push-only تحمیل کرده است.

از نظر تحلیلی، نقش Pull بیشتر در شرایطی برجسته می‌شود که برخی نودها به دلیلloss ، تأخیر زیاد یا توپولوژی ضعیف، پیام را از طریق Push در زمان کم دریافت نکرده باشند. در چنین حالتی، Pull با ارسال دوره‌ای اعلان وجود پیام (IHAVE) می‌تواند نودهای جا مانده را شناسایی کرده و از طریق IWANT پیام را بازیابی کند، در نتیجه tail کاهش یافته و رسیدن به 95٪ سریع‌تر می‌شود.

پارامتر pull_interval در این میان نقش کلیدی دارد. اگر pull_interval کوچک‌تر انتخاب شود، نودها با فرکانس بالاتری وضعیت پیام‌های خود را به همسایگان اعلام می‌کنند. این موضوع دو اثر متضاد دارد:

احتمال این‌که نودهای جا مانده سریع‌تر از طریق Pull پیام را دریافت کنند افزایش می‌یابد، بنابراین زمان همگرایی کاهش می‌یابد.

تعداد پیام‌های کنترلی (IHAVE/IWANT) افزایش می‌یابد، در نتیجه سربار شبکه بیشتر می‌شود.

بنابراین بین سرعت همگرایی و سربار پیام یک trade-off وجود دارد. کاهش pull_interval می‌تواند همگرایی را بهبود دهد، اما به قیمت افزایش ترافیک کنترلی. برعکس، افزایش pull_interval باعث کاهش سربار می‌شود، اما در شبکه‌های با loss یا توپولوژی ضعیف ممکن است زمان رسیدن به درصد بالای پوشش را افزایش دهد.

بخش دوم) مقاومت در برابر Sybil با Proof-of-Work

در شبکه‌هایP2P ، مهاجمان می‌توانند با ایجاد تعداد زیادی هویت جعلی (Sybil) Peer List سایر گره‌ها را آلوده کند و با پر کردن لیست همسایه‌ها، کیفیت کشف همسایه و انتشار پیام را کاهش دهند. در این بخش برای افزایش هزینه‌ی ساخت هویت‌های متعدد، یک مکانیزم **Proof-of-Work** به فرآیند پذیرش پیام **HELLO** اضافه شده است.

تغییرات اعمال‌شده در طراحی و پیاده‌سازی

1) افزودن پارامتر قابل تنظیم pow_k: به CLI گره پارامتر pow-k--اضافه شد. مقدار pow_k تعداد صفرهای ابتدایی لازم در هش SHA-256 است. اگر pow_k=0باشد، PoW غیرفعال می‌شود.
2) استخراج تابع هش و حل PoW در سمت فرستنده: هر گره هنگام ارسال HELLO (در بوت‌استرپ یا هنگام معرفی به همسایه جدید)، یک nonce پیدا می‌کند به‌طوری‌که H(node_id || nonce) با pow_k صفر ابتدایی شروع شود. این اطلاعات در فیلد payload.pow پیام HELLO قرار می‌گیرد (hash_alg, difficulty_k, nonce, digest_hex).
3) اعتبارسنجی PoW : در سمت گیرنده قبل از اضافه شدن به Peer List در handler مربوط به HELLO ابتدا PoW اعتبارسنجی می‌شود(difficulty_k دقیقاً برابر pow_k تنظیم‌شده روی گیرنده باشد). digest_hex دوباره با sha256(node_id|nonce) محاسبه می‌شود و باید برابر digest_hex دریافتی باشد و با pow_k صفر شروع شود. در صورت نامعتبر بودن، HELLO رد می‌شود و فرستنده به Peer List اضافه نمی‌گردد. برای جلوگیری از آلوده شدن Peer List، ثبت همسایه برای HELLO فقط بعد از پذیرش PoW انجام می‌شود (یعنی قبل از اعتبارسنجی، peer_add/peer_touchانجام نمی‌دهیم).

مزیت امنیتی: PoW چگونه Sybil را هزینه‌بر می‌کند؟

PoW باعث می‌شود هر هویت جدید برای پیوستن به شبکه مجبور باشد محاسبه‌ی نسبتاً پرهزینه یافتن nonce را انجام دهد. مهاجم برای ایجاد هزاران گره جعلی باید هزاران بار این محاسبه را انجام دهد پس این هزینه رابطه مستقیم و خطی با تعداد sybil ها دارد و سرعت ایجاد هویت جعلی و افزودن آن به peer list را کاهش میدهد و با این کاهش نرخ ورود به لیست احتمال آلوده شدن آن به طرز چشمگیری کاهش پیدا میکند.

هزینه و اثر جانبی: زمان متوسط تولید PoW و اثر روی گره‌های سالم

PoW مستقیماً باعث افزایش زمان پیوستن گره سالم می‌شود چون قبل از ارسال HELLO باید nonce پیدا شود. بنابراین هرچه pow_k بزرگ‌تر باشد، زمان join شدن بیشتر می‌شود. برای اندازه گیری این متغییر اسکریپت زیر را نوشتیم :

```
import argparse
import hashlib
import json
import time
import uuid
```

```
from statistics import mean, pstdev

def pow_digest(node_id: str, nonce: int) -> str:
    s = f"{node_id}|{nonce}".encode("utf-8")
    return hashlib.sha256(s).hexdigest()

def compute_pow(node_id: str, k: int, start_nonce: int = 0):
    prefix = "0" * k
    nonce = start_nonce
    t0 = time.perf_counter()
    tries = 0
    while True:
        h = pow_digest(node_id, nonce)
        tries += 1
        if h.startswith(prefix):
            dt = time.perf_counter() - t0
            return {
                "nonce": nonce,
                "digest_hex": h,
                "time_s": dt,
                "tries": tries,
            }
        nonce += 1

def bench_one_k(k: int, trials: int, base_nonce: int):
    times = []
    tries = []
    samples = []
    for i in range(trials):
        node_id = str(uuid.uuid4())
        res = compute_pow(node_id, k, start_nonce=base_nonce + i * 100000)
        times.append(res["time_s"])
        tries.append(res["tries"])
        samples.append({
            "trial": i,
            "node_id": node_id,
            **res
        })

    return {
        "pow_k": k,
        "trials": trials,
        "time_mean_s": mean(times),
        "time_std_s": pstdev(times) if trials >= 2 else 0.0,
        "time_min_s": min(times),
```

```
        "time_max_s": max(times),
        "tries_mean": mean(tries),
        "tries_min": min(tries),
        "tries_max": max(tries),
        "samples": samples[:3],
    }

ap = argparse.ArgumentParser()
ap.add_argument("--ks", type=str, default="3,4,5", help="comma-separated pow_k
values")
ap.add_argument("--trials", type=int, default=10)
ap.add_argument("--base-nonce", type=int, default=0)
ap.add_argument("--out", type=str, default="pow_bench.json")
args = ap.parse_args()

ks = [int(x.strip()) for x in args.ks.split(",") if x.strip()]
report = {
    "machine_note": "Run on my laptop/PC (fill CPU model in report)",
    "timestamp": time.time(),
    "results": [],
}

for k in ks:
    r = bench_one_k(k, args.trials, args.base_nonce)
    report["results"].append(r)
    print(f"pow_k={k}  mean={r['time_mean_s']:.4f}s  min={r['time_min_s']:.4f}s
 max={r['time_max_s']:.4f}s  tries_mean={r['tries_mean']:.1f}")

with open(args.out, "w", encoding="utf-8") as f:
    json.dump(report, f, indent=2)
print(f"\nSaved: {args.out}")
```

در این کد برای هر مقدار pow_k (3و4و5) یک node_id تصادفی ایجاد شده و با nonse brute force را افزایش میدهیم تا SHA256(node_id|nonce) دارای k صفر ابتدایی شود و در حین این کار زمان شروع و پایان و تعداد تلاش ها ثبت شده و با تکرار این فرآیند در نهایت میانگین زمان و کمینه و بیشینه آن را به دست میاوریم. با 15 تکرار برای هر مقدار نتایج زیر را به دست آورده ایم:

```
zahra@DESKTOP-D6MAGMQ:/mnt/c/users/asus/downloads/Telegram Desktop$ python3 pow_phaze4.py --ks 3,4,5 --trials 15 -
-out pow_bench.json
pow_k=3  mean=0.0029s  min=0.0002s  max=0.0072s  tries_mean=2868.3
pow_k=4  mean=0.1103s  min=0.0016s  max=0.4079s  tries_mean=98379.9
pow_k=5  mean=1.0223s  min=0.0812s  max=2.5612s  tries_mean=1004220.1

Saved: pow_bench.json
```

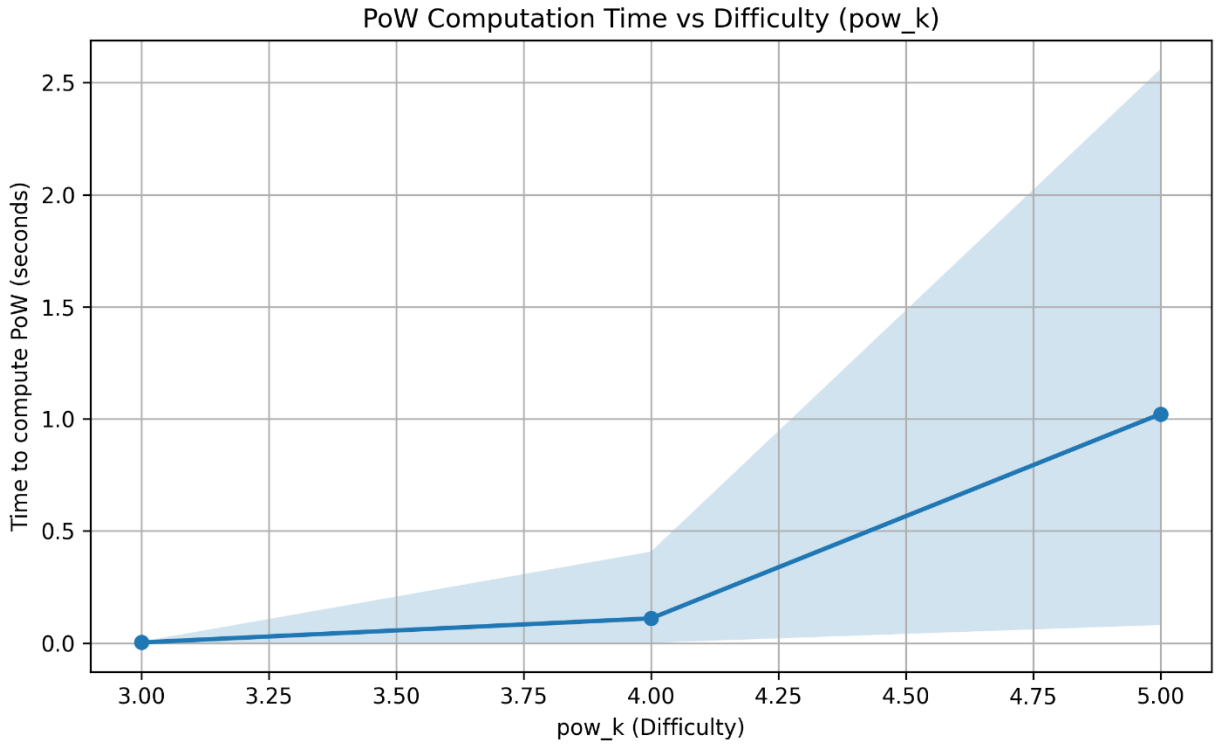
POW_K	میانگین زمان (ثانیه)	کمینه	بیشینه	میانگین تلاش
3	0.0029	0.0002	0.0072	2,868
4	0.1103	0.0016	0.4079	98,380

5	1.0223	0.0812	2.5612	1,004,220

از نظر تئوری، چون هر رقم هگزادسیمال ۴ بیت است $P(success)=16^{-k}$ پس $E[tries]\approx 16^{k}$ که رشد نمایی دارد طبق نتایج جدول از k=3 به k=4 زمان حدود ۳۸ برابر افزایش یافته و از k=4 به k=5 زمان تقریباً ۹ برابر افزایش یافته و میانگین تعداد تلاش ها برای k=5 نزدیک به مقدار تئوریک 16^{5}=1,048,576است و این نشان می‌دهد پیاده‌سازی PoW صحیح است و رفتار نمایی مورد انتظار مشاهده شده است.

برای k = 5 انتخاب شده میانگین زمان انتظار در حدود 1 ثانیه است و در این حالت هر گره سالم باید برای پیوستن به شبکه حدود 1 ثانیه تاخیر را تجربه کند که این مقدار قابل قبول است زیرا مانع استفاده عادی از شبکه نشده و امنیت را افزایش داده است زیرا اگر مهاجم بخواهد به طور مثال 1000 گره جعلی بسازد باید حدود 1000 ثانیه که 16 دقیقه میشود زمان صرف کند که نشان میدهد نرخ ورود sybil به شبکه به شدت کاهش یافته و آلوده سازی لیست دشوار میشود و این حمله زمانبر و هزینه بر شده. دلیل انتخاب 5 این است که با توجه به نتایج pow_k = 3 بسیار سریع است (3 میلی‌ثانیه) و در نتیجه امنیت کافی ندارد و pow_k = 4 حدود 0.1 ثانیه طول میکشد و امنیت متوسطی دارد و pow_k = 5 باحدود 1 ثانیه تعادل مناسب بین امنیت و کارایی دارد

با استفاده از مقادیر داخل جدول که در فایل جیسون ذخیره شده اند کد plot_pow.py را روی فایل جیسون ران کرده و نمودار مقادیر به دست امده مطابق زیر میشود:



شکل فوق نشان می‌دهد که زمان تولید PoW با افزایش سختی pow_k به‌صورت نمایی رشد می‌کند. این رفتار با تحلیل تئوریک $E[tries]\approx 16^{k}$ سازگار است.