CS305 2025 春季学期最终项目 - 区块链网络模拟报告

对等方初始化

`peer_manager.py`

该部分实现了每个节点在启动时对其自身身份、连接状态和行为记录等方面的初始化。

主要变量和结构：

```
# 节点状态追踪
from collections import defaultdict

peer_status = {}                  # 当前对等方状态（如“ALIVE”、“UNREACHABLE”）
last_ping_time = {}               # 记录上次收到ping/pong的时间戳
rtt_tracker = {}                  # 节点与其他对等方的RTT信息
blacklist = set()                 # 黑名单
peer_offense_counts = defaultdict(int)    # 节点违规行为次数
```

主要函数：

```
def record_offense(peer_id):
    peer_offense_counts[peer_id] += 1
    if peer_offense_counts[peer_id] >= 3:
        blacklist.add(peer_id)
```

说明：每个节点会记录其他节点的不良行为（如发送错误区块）。如果某节点累计3次违规，则被加入黑名单，后续将不再接收其消息。

```
def update_peer_heartbeat(peer_id):
    last_ping_time[peer_id] = time.time()
```

说明：更新最近一次收到该节点心跳的时间。

```
def start_peer_monitor():
    def loop():
        while True:
            now = time.time()
            for peer, last_time in last_ping_time.items():
                if now - last_time > 30:
                    peer_status[peer] = "UNREACHABLE"
                else:
                    peer_status[peer] = "ALIVE"
            time.sleep(5)
    threading.Thread(target=loop, daemon=True).start()
```

说明：周期性检查每个节点最近的心跳时间，如果超过30秒未收到ping或pong，即被标记为不可达。

对等方发现

`peer_discovery.py`

该部分负责在网络中发现新的对等节点，并维持与已知节点的连接信息。

主要数据结构：

```
known_peers = {}          # {peer_id: (ip, port)}
peer_flags = {}           # 标志信息，如 NAT 状态、是否轻节点等
reachable_by = defaultdict(set)  # 某个目标节点可被哪些节点中继访问
peer_config = {}          # 本地节点配置（如self_id, self_ip等）
```

注册新节点函数：

```
def handle_hello_message(msg, self_id):
    sender_id = msg.get("sender_id")
    sender_ip = msg.get("ip")
    sender_port = msg.get("port")
    sender_flags = msg.get("flags", {})

    if sender_id == self_id:
        return []

    if sender_id not in known_peers:
        known_peers[sender_id] = (sender_ip, sender_port)
        peer_flags[sender_id] = {
            "nat": sender_flags.get("nat", False),
            "light": sender_flags.get("light", False)
        }

    if sender_id not in reachable_by:
        reachable_by[sender_id] = set()

    reachable_by[sender_id].add(self_id)
    return [sender_id]
```

说明：接收到“HELLO”消息后调用此函数。如果是新节点，将其添加到 `known_peers` 和 `peer_flags`，同时更新 `reachable_by` 映射。

发送hello消息：

```
def start_peer_discovery(self_id, self_info):
    from outbox import enqueue_message

    def loop():
        msg = {
            "type": "HELLO",
            "sender_id": self_id,
            "ip": self_info["ip"],
            "port": self_info["port"],
            "flags": {
                "nat": self_info.get("nat", False),
                "light": self_info.get("light", False)
```

```
            },
            "message_id": generate_message_id()
        }
        msg_str = json.dumps(msg) + "\n"

        for peer_id, (peer_ip, peer_port) in known_peers.items():
            enqueue_message(peer_id, peer_ip, peer_port, msg_str)

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

说明：启动一个后台线程，定期向所有已知对等方发送当前节点的“HELLO”消息，传达自身存在。

套接字服务监听

`socket_server.py`

该部分负责建立TCP连接，监听其他节点的连接请求，并处理接收到的JSON消息。

主要函数：

```
def start_socket_server(self_id, self_ip, port):
    def listen_loop():
        server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

        bind_ip = self_ip if self_ip in ["127.0.0.1", "localhost"] else "0.0.0.0"
        server_socket.bind((bind_ip, port))
        server_socket.listen(10)

        while True:
            client_socket, addr = server_socket.accept()

            def handle_client(sock):
                try:
                    sock.settimeout(30)
                    buffer = b""

                    while True:
                        chunk = sock.recv(4096)
                        if not chunk:
                            break
                        buffer += chunk

                        while b"\n" in buffer:
                            msg_bytes, buffer = buffer.split(b"\n", 1)
                            if msg_bytes:
                                try:
                                    msg_str = msg_bytes.decode()
                                    json_data = json.loads(msg_str)
                                    dispatch_message(json_data, self_id, self_ip)
                                except Exception as e:
                                    logger.error(f"处理消息失败: {e}")
                finally:
                    sock.close()
```

```
            threading.Thread(target=handle_client, args=(client_socket,),
daemon=True).start()

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

说明：

- 启动监听线程绑定并监听指定端口。
- 接收到新连接时启动一个线程处理其消息。
- 每条消息以 `\n` 分隔，支持多条连续接收。
- 接收后立即解析为JSON格式并传递至 `dispatch_message` 进行进一步处理。

区块和交易的生成与验证

transaction.py

这段代码实现了一个简单的交易系统，包括生成和管理交易、交易池管理、以及广播交易等功能。交易通过 `TransactionMessage` 类进行表示，并存储在本地交易池中。定时生成交易并广播到已知的对等节点（`known_peers`）。交易池中的交易 ID 被追踪，以确保不会添加重复交易。

1. `transaction_generation`

   功能：定期生成交易并将其添加到本地交易池中，交易的发送方是 `self_id`，接收方从已知对等节点中随机选择。每隔指定时间（默认为15秒）生成一个新的交易并广播。

2. `add_transaction`

   功能：将新的交易添加到本地的交易池中（`tx_pool`）。该函数会首先检查交易的 `id` 是否已经存在于交易池中，如果不存在，则将交易添加到池中，并将其 `id` 添加到 `tx_ids` 集合中，确保不会有重复的交易。

3. `get_recent_transactions`

   功能：返回当前交易池中的所有交易，以字典格式返回。

4. `clear_pool`

   功能：清空交易池（`tx_pool`）和交易 ID 集合（`tx_ids`）。

block_handler.py

该代码实现了一个简单的区块链系统，涵盖区块的生成、同步、验证、存储等核心功能。系统中节点通过网络传播区块和交易信息，确保区块链的去中心化和一致性。每个节点可以生成新的区块并将其广播到网络中的其他节点，同时也可以接收其他节点的区块进行同步，处理孤块（即缺失前置区块的区块），并维护区块链的最新状态。

以下为主要函数的功能介绍：

1. `request_block_sync`

   功能：用于请求区块同步。当新节点加入时，它需要请求已知节点发送区块头信息。同步请求分为两种：

   - 对于新节点，会请求一定范围内的区块头（每次最多100个区块头）。
   - 对于常规节点，会请求最新的区块头信息。

2. `block_generation`

功能：定期生成新区块。区块生成过程涉及等待区块同步完成，生成新的区块（可能包含恶意模式，生成随机区块ID）并广播。

3. `create_dummy_block`

   功能：生成一个虚拟区块，包含交易信息，并计算区块的哈希值。

4. `compute_block_hash`

   功能：计算区块的哈希值（排除 `block_id` 字段）。

5. `handle_block`

   功能：处理接收到的区块，验证区块的有效性，确保区块的顺序和一致性。

6. `create_getblock`

   功能：创建 `GETBLOCK` 请求消息，用于请求特定区块。

inv_message.py

这段代码主要用于处理区块链中的区块清单广播（`INV` 消息）。在一个去中心化的区块链网络中，节点之间会广播他们所拥有的区块ID（即区块清单），以便其他节点同步区块。该代码实现了生成区块清单消息、获取本地区块清单、以及广播区块清单的功能。

1. `create_inv`

   功能：生成一个 `INV` 消息，这个消息包含了发送者的ID、发送的区块ID和消息ID。

2. `get_inventory`

   功能：返回本地区块链（`received_blocks`）中的所有区块ID。

3. `broadcast_inventory`

   功能：广播本地区块链的区块清单。

发送消息处理`outbox.py`

基本功能

在区块链P2P网络中，`outbox.py` 负责处理所有从本节点发送到其他节点的消息。其主要实现了以下核心功能：

1. **消息队列管理**：使用优先级队列管理发往不同节点的各类消息，确保高优先级消息（如区块和交易）优先发送。

```
# 优先级设置
PRIORITY_HIGH = {"PING", "PONG", "BLOCK", "INV", "GETDATA"}
PRIORITY_MEDIUM = {"TX", "HELLO"}
PRIORITY_LOW = {"RELAY"}

# 队列实现
queues = defaultdict(lambda: defaultdict(deque))
```

2. **速率限制**：实现了对每个节点的发送频率限制，防止短时间内向同一节点发送过多消息。

```
def is_rate_limited(peer_id):
    str_peer_id = str(peer_id)
    current_time = time.time()
```

```
    # 移除过期的时间戳
    timestamps = peer_send_timestamps[str_peer_id]
    while timestamps and TIME_WINDOW < current_time - timestamps[0]:
        timestamps.pop(0)

    # 检查发送频率
    if len(timestamps) >= RATE_LIMIT:
        return True

    # 记录当前发送时间
    timestamps.append(current_time)
    return False
```

3. **NAT穿透**: 实现了对NAT节点的消息中继功能。当目标节点为NAT时，会选择合适的中继节点转发消息。

```
def relay_or_direct_send(self_id, dst_id, message):
    is_nated = False
    if dst_id in peer_flags and peer_flags[dst_id].get("nat",False):
        is_nated = True

    if is_nated:
        relay_peer = get_relay_peer(self_id, dst_id)

        if relay_peer:
            # 创建中继消息
            relay_message = {
                "type": "RELAY",
                "sender_id": self_id,
                "target_id": dst_id,
                "payload": message,
            }
            # ...发送中继消息
```

4. **消息重试**：实现了消息发送失败时的重试机制，最多重试3次，超过则丢弃。

```
if not success:
    retries[target_id] = retries.get(target_id, 0) + 1

    if retries[target_id] <= MAX_RETRIES:
        # 重新入队，但降低优先级
        priority = classify_priority(message) + 1
        with lock:
            queues[target_id][priority].append((message, ip, port, time.time()))
    else:
        logger.warning(f"发送到 {target_id} 的消息已达最大重试次数，放弃发送")
        retries[target_id] = 0
```

5. **消息广播**：实现了gossip协议用于消息广播，支持将消息高效传播到网络中的其他节点。

```
def gossip_message(self_id, message, fanout=3):
    """使用gossip协议广播消息到网络中"""
    # 随机选择fanout个节点
```

```
    available_peers = []
    for peer_id, (ip, port) in known_peers.items():
        if peer_id != self_id:
            available_peers.append((peer_id, ip, port))

    if not available_peers:
        return

    if len(available_peers) <= fanout:
        selected_peers = available_peers
    else:
        selected_peers = random.sample(available_peers, fanout)

    # 向选中的节点发送消息
    for peer_id, ip, port in selected_peers:
        enqueue_message(peer_id, ip, port, message)
```

优化改进

针对消息发送机制，我实现了以下优化和改进：

1. **公平队列调度**：实现了轮询发送机制，确保每个目标节点都能公平地获得发送机会，避免某些节点消息堆积。

```
# 轮询处理每个节点的消息
if last_peer_index >= len(peers):
    last_peer_index = 0

target_id = peers[last_peer_index]
last_peer_index = (last_peer_index + 1) % len(peers)
```

2. **网络条件模拟**：实现了网络延迟和丢包模拟，可以更真实地测试系统在不同网络条件下的表现。

```
def apply_network_conditions(send_func):
    def wrapper(ip, port, message):
        # 检查发送容量限制
        if not rate_limiter.allow():
            # 更新丢弃统计
            msg_type = message.get("type", "OTHER")
            if msg_type in drop_stats:
                drop_stats[msg_type] += 1
            else:
                drop_stats["OTHER"] += 1
            return False

        # 模拟丢包
        if random.random() < DROP_PROB:
            msg_type = message.get("type", "OTHER")
            if msg_type in drop_stats:
                drop_stats[msg_type] += 1
            else:
                drop_stats["OTHER"] += 1
            return False

        # 模拟网络延迟
```

```
        min_latency, max_latency = LATENCY_MS
        latency = random.randint(min_latency, max_latency) / 1000.0
        time.sleep(latency)

        # 执行实际发送
        return send_func(ip, port, message)

    return wrapper
```

3. **智能中继节点选择**：通过分析网络拓扑和节点状态，智能选择最优的中继节点，提高NAT穿透的成功率。

```
def get_relay_peer(self_id, dst_id):
    from peer_discovery import known_peers,peer_flags
    from peer_manager import rtt_tracker
    from peer_discovery import known_peers, reachable_by

    """为NAT节点找到最佳中继节点"""
    candidate_relays_info = [] # Store (peer_id, ip, port)
    # 尝试从reachable_by中查找已知可以作为目标节点中继的节点
    if dst_id in reachable_by and reachable_by[dst_id]:
        for relay_id in reachable_by[dst_id]:
            if relay_id in known_peers:
                ip, port = known_peers[relay_id]
                candidate_relays_info.append((relay_id, ip, port))

    if not candidate_relays_info:
        return None
    # 选择最佳中继 (例如, RTT最低的)
    # 如果有多个候选，选择RTT最小的；如果RTT不可用，则随机选择一个
    best_relay_candidate = None
    min_rtt_to_relay = float('inf')

    # 基于RTT选择
    relays_with_rtt = []
    for relay_id, relay_ip, relay_port in candidate_relays_info:
        rtt = rtt_tracker.get(relay_id, float('inf'))
        relays_with_rtt.append(((relay_id, relay_ip, relay_port), rtt))

    if relays_with_rtt:
        # 按RTT排序并选择第一个
        relays_with_rtt.sort(key=lambda x: x[1])
        best_relay_candidate = relays_with_rtt[0][0]
        min_rtt_to_relay = relays_with_rtt[0][1]
    elif candidate_relays_info: # 如果没有RTT信息，但有候选者，则随机选择
        best_relay_candidate = random.choice(candidate_relays_info)

    if best_relay_candidate:
        return best_relay_candidate # (peer_id, ip, port)
    else:
        return None
```

4. **消息超时处理**：实现了消息发送超时检测，避免长时间阻塞队列中的其他消息。

```
# 检查消息是否超时
if time.time() - enqueue_time > 30:  # 30秒超时
    logger.warning(f"消息发送超时，丢弃: {message.get('type')} 到 {target_id}")
    continue
```

这些优化使得消息发送系统在高负载和复杂网络环境下能够更加稳定和高效地运行。

接收消息处理`message_handler.py`

基本功能

`message_handler.py`是区块链P2P网络的核心组件之一，负责处理接收到的各类消息。它实现了不同消息类型的处理逻辑，确保网络中的信息能够正确流通和处理。以下是其主要功能和处理流程：

1. **消息分发机制**：根据消息类型将接收到的消息分发到对应的处理函数。

```
def dispatch_message(msg, self_id, self_ip):
    try:
        msg_type = msg.get("type")
        # 根据消息类型调用不同的处理函数
        if msg_type == "RELAY":
            # 处理中继消息
            # ...
        elif msg_type == "HELLO":
            # 处理节点发现消息
            # ...
        elif msg_type == "BLOCK":
            # 处理区块消息
            # ...
        # 其他消息类型处理...
    except Exception as e:
        logger.error(f"处理消息时出错: {e}")
```

2. **消息重复检测**：通过维护已见消息ID的集合，避免重复处理同一消息，防止重放攻击。

```
msg_id = msg.get("message_id", str(hash(str(msg))))
current_time = time.time()
if msg_id in seen_message_ids:
    if current_time - seen_message_ids[msg_id] < SEEN_EXPIRY_SECONDS:
        logger.warning(f"收到来自节点 {sender_id} 的重复消息，类型为{msg_type}，丢弃")
        message_redundancy[msg_id] = message_redundancy.get(msg_id, 0) + 1
        drop_stats["DUPLICATE"] += 1
        return
# 记录消息ID和时间戳
seen_message_ids[msg_id] = current_time
```

3. **入站速率限制**：防止单个节点在短时间内发送过多消息，有效预防DoS攻击。

```
def is_inbound_limited(peer_id):
    current_time = time.time()
    str_peer_id = str(peer_id)

    # 记录当前时间戳
    peer_inbound_timestamps[str_peer_id].append(current_time)

    # 删除过期的时间戳
    peer_inbound_timestamps[str_peer_id] = [ts for ts in
peer_inbound_timestamps[str_peer_id]
                                            if current_time - ts <=
INBOUND_TIME_WINDOW]

    # 检查剩余的时间戳数量是否超过入站速率限制
    return len(peer_inbound_timestamps[str_peer_id]) > INBOUND_RATE_LIMIT
```

4. **区块消息处理**：验证接收到的区块，包括区块哈希校验，然后将其加入本地区块链。

```
# 验证区块ID是否正确
computed_hash = compute_block_hash(msg)
if computed_hash != msg["block_id"]:
    logger.warning(f"来自节点 {block_sender_id} 的区块ID验证失败")
    # 将节点记录为恶意节点
    record_offense(block_sender_id)
    return

# 处理区块
handle_block(msg, self_id)

# 创建并广播INV消息
inv_msg = create_inv(self_id, [block_id])
gossip_message(self_id, inv_msg)
```

5. **交易消息处理**：验证交易哈希，然后将其添加到本地交易池并广播给其他节点。

```
# 验证交易ID正确性
if compute_tx_hash(msg) != msg["id"]:
    record_offense(msg["from_peer"])
    logger.warning(f"来自节点{msg['from_peer']}的transaction消息id验证不通过,丢弃")
    return

# 添加交易到交易池
if tx_id not in seen_txs:
    seen_txs.add(tx_id)
    add_transaction(msg)
    # 广播交易
    gossip_message(self_id, msg)
```

6. **节点发现与心跳**：处理HELLO、PING、PONG等消息，维护网络拓扑和节点状态。

```
# 处理PING消息
# 更新the last ping time
update_peer_heartbeat(sender_id)

# 创建并发送PONG响应
pong_msg = create_pong(self_id, msg.get("timestamp"))
if sender_id in known_peers:
    enqueue_message(sender_id, known_peers[sender_id][0], known_peers[sender_id]
[1], pong_msg)
```

7. **中继消息处理**：处理RELAY类型的消息，用于NAT穿透场景下的消息转发。

```
target_id = msg.get("target_id")
payload = msg.get("payload", {})

if target_id == self_id:
    # 本节点是目标，处理payload
    if payload:
        dispatch_message(payload, self_id, self_ip)
else:
    # 转发RELAY消息到目标节点
    if target_id in known_peers:
        target_ip, target_port = known_peers[target_id]
        enqueue_message(target_id, target_ip, target_port, msg)
```

优化改进

在消息处理模块中，我实现了以下几点优化和改进：

1. **消息冗余度追踪**：实现了消息冗余度统计功能，可以监控网络中重复消息的情况，有助于分析网络效率。

```
def get_redundancy_stats():
    """返回重复消息次数的统计信息"""
    return message_redundancy

# 记录重复消息
message_redundancy[msg_id] = message_redundancy.get(msg_id, 0) + 1
```

2. **黑名单机制**：引入恶意节点黑名单机制，当检测到节点发送错误哈希的区块或交易时，将其加入黑名单。

```
# 检查节点是否在黑名单中
from peer_manager import blacklist
if str(sender_id) in blacklist:
    logger.warning(f"丢弃来自黑名单节点 {sender_id} 的消息")
    drop_stats["BLACKLISTED"] += 1
    return
```

3. **消息记录系统**：通过与仪表盘集成，实现了消息记录功能，便于可视化和分析网络通信。

```
from dashboard import log_received_message
log_received_message(sender_id, self_id, msg_type, msg)
```

4. **高效的重复消息过滤**：使用带过期时间的消息ID记录，既能有效过滤重复消息，又不会导致内存无限增长。

```
# 移除过期的消息ID记录
SEEN_EXPIRY_SECONDS = 600  # 10分钟过期
# 检查是否是过期记录
if current_time - seen_message_ids[msg_id] < SEEN_EXPIRY_SECONDS:
    # 处理重复消息
```

5. **智能消息验证**：根据消息类型实现不同的验证逻辑，确保消息完整性和真实性。

```
# 区块消息验证
computed_hash = compute_block_hash(msg)
if computed_hash != msg["block_id"]:
    # 验证失败处理...

# 交易消息验证
if compute_tx_hash(msg) != msg["id"]:
    # 验证失败处理...
```

6. **错误处理**：对不同类型的消息处理错误进行分类记录，便于后续故障分析和系统优化。

```
drop_stats = defaultdict(int)  # 记录每种消息类型的丢弃次数

# 记录不同原因的消息丢弃
drop_stats["INVALID"] += 1
drop_stats["DUPLICATE"] += 1
drop_stats["RATE_LIMITED"] += 1
drop_stats["BLACKLISTED"] += 1
```

这些优化使得消息处理系统能够更加高效、稳定，同时对异常情况有更好的容错能力和可观察性。

仪表盘`dashboard.py`

基本功能实现思路

仪表盘`dashboard.py`实现了一个基于Flask的Web应用，用于可视化区块链节点的运行状态和网络情况。它通过提供RESTful API和Web页面，使用户可以直观地了解网络中的各种信息。以下是各个API的实现思路：

- `peers`：显示已知对等方的信息，包括`{对等方 ID、IP 地址、端口、状态、NAT 或非 NAT、轻量级或完整}`。

```
@app.route('/peers')
def peers():
    peers_info = {}

    # 整合所有节点信息
    for peer_id, (ip, port) in known_peers.items():
        peer_id_str = str(peer_id)
```

```
        flag_info = peer_flags.get(peer_id_str, {})
        status = peer_status.get(peer_id_str, "unknown")

        # 处理NAT和light标志
        nat_status = flag_info.get("nat")
        light_status = flag_info.get("light")

        peers_info[peer_id_str] = {
            "peer_id": peer_id_str,
            "ip": ip,
            "port": port,
            "status": status,
            "nat": nat_status,
            "light": light_status
        }

    return jsonify(peers_info)
```

- `transactions`：显示本地池 `tx_pool` 中的交易。这里通过导入transaction模块获取最新交易列表。

```
@app.route('/transactions')
def transactions():
    try:
        # 获取交易数据
        from transaction import get_recent_transactions
        tx_data = get_recent_transactions()

        # 确保返回的是列表，不是其他类型
        if not isinstance(tx_data, list):
            if isinstance(tx_data, dict):
                tx_data = [tx_data]  # 如果是单个交易，转为列表
            else:
                tx_data = []  # 如果是其他类型，返回空列表

        # 过滤确保所有项都是字典
        filtered_data = []
        for item in tx_data:
            if isinstance(item, dict):
                filtered_data.append(item)

        return jsonify(filtered_data)
    except Exception as e:
        return jsonify({"error": f"获取交易数据时出错: {str(e)}"}), 500
```

- `blocks`：显示本地区块链中的区块。这里通过全局变量引用区块链数据，返回完整区块链信息。

```python
@app.route('/blocks')
def blocks():
    # 返回区块链数据
    return jsonify(dashboard_data["blocks"])

# 在仪表盘启动时设置全局引用
def start_dashboard(self_id, port=None):
    global blockchain_data_ref, known_peers_ref, dashboard_data
    from block_handler import received_blocks
    dashboard_data["peer_id"] = self_id
    blockchain_data_ref = received_blocks  # 设置对区块链数据的引用
    # ...
```

- `orphan`：显示孤立区块。通过从区块处理器模块获取孤块列表并返回。

```python
@app.route('/orphans')
def orphan_blocks():
    # 返回孤块数据
    return jsonify(dashboard_data["orphan_blocks"])

# 在更新循环中获取最新孤块数据
def update_dashboard_data(peer_id):
    # ...
    try:
        # 更新孤块数据
        from block_handler import get_orphan_blocks
        dashboard_data["orphan_blocks"] = get_orphan_blocks()
    except Exception as e:
        logger.error(f"更新孤块数据时出错: {e}")
```

- `latency`：显示对等方之间的传输延迟。这里通过peer_manager模块的RTT跟踪器实现。

```python
@app.route('/latency')
def latency():
    # 返回延迟数据
    return jsonify(dashboard_data["latency"])

def update_dashboard_data(peer_id):
    # ...
    try:
        # 更新延迟数据
        latency_data = {}
        for target_id, tracker in rtt_tracker.items():
            if tracker.has_data():
                latency_data[target_id] = {
                    "min": tracker.get_min_rtt(),
                    "max": tracker.get_max_rtt(),
                    "avg": tracker.get_average_rtt()
                }
        dashboard_data["latency"] = latency_data
    except Exception as e:
        logger.error(f"更新延迟数据时出错: {e}")
```

- `capacity`：显示对等方的发送能力。通过outbox模块的rate_limiter实现。

```
@app.route('/capacity')
def capacity():
    from outbox import rate_limiter
    # 返回节点容量
    return jsonify(rate_limiter.capacity)
```

- `redundancy`：显示收到的冗余消息数量。通过message_handler模块中的统计数据实现。

```
@app.route('/redundancy')
def redundancy_stats():
    try:
        from message_handler import get_redundancy_stats
        # 获取冗余消息统计
        redundancy_data = get_redundancy_stats()

        # 确保返回的是dict
        if not isinstance(redundancy_data, dict):
            redundancy_data = {}

        # 过滤确保所有值都是数字
        filtered_data = {}
        for key, value in redundancy_data.items():
            try:
                filtered_data[str(key)] = int(value)
            except (ValueError, TypeError):
                pass

        return jsonify(filtered_data)
    except Exception as e:
        return jsonify({"error": f"获取冗余消息统计时出错: {str(e)}"}), 500
```

此外，我还添加了一些额外的API用于增强仪表盘的功能：

1. **黑名单管理**：查看和管理被标记为恶意的节点。

```
@app.route('/api/blacklist')
def get_blacklist():
    # 只获取当前节点的黑名单列表
    return jsonify(list(blacklist))
```

2. **网络状态统计**：提供网络层面的统计信息。

```
@app.route('/api/network/status')
def network_status_api():
    # 获取网络统计
    online_peers = sum(1 for status in peer_status.values() if status == "ALIVE")
    total_peers = len(known_peers)
    nat_peers = sum(1 for flags in peer_flags.values() if flags.get("nat",
False))
    light_peers = sum(1 for flags in peer_flags.values() if flags.get("light",
False))

    return jsonify({
        "total_peers": total_peers,
```

```
        "online_peers": online_peers,
        "nat_peers": nat_peers,
        "light_peers": light_peers,
        "blacklisted_peers": len(blacklist)
    })
```

3. **消息记录**：记录节点之间的消息交互，便于分析。

```
def log_sent_message(sender_id, receiver_id, msg_type, content):
    """记录发送的消息"""
    try:
        message_record = {
            "timestamp": time.time(),
            "sender": sender_id,
            "receiver": receiver_id,
            "type": msg_type,
            "direction": "OUTBOUND",
            "content_hash": hash(str(content)) if content else None
        }

        # 将消息记录添加到全局消息记录
        with messages_lock:
            message_log.append(message_record)
            # 保持消息记录在合理大小范围内
            if len(message_log) > MAX_MESSAGES:
                message_log.pop(0)
    except Exception as e:
        logger.error(f"记录发送消息时出错: {e}")
```

前端展示部分

仪表盘前端基于Bootstrap和Vue.js构建，提供了一个现代化的用户界面来展示区块链网络的实时状态。前端部分的主要实现包括：

1. **主页面结构**：使用Bootstrap的栅格系统创建响应式布局，包括导航栏、侧边栏和主内容区域。

```
<!-- 基本页面结构 -->
<div class="container-fluid">
  <div class="row">
    <div class="col-md-3 sidebar">
      <!-- 侧边栏导航 -->
      <ul class="nav flex-column">
        <li class="nav-item"><a href="#peers">节点信息</a></li>
        <li class="nav-item"><a href="#blocks">区块链</a></li>
        <li class="nav-item"><a href="#transactions">交易池</a></li>
        <!-- 其他导航项 -->
      </ul>
    </div>
    <div class="col-md-9 main-content">
      <!-- 主内容区域 -->
      <div id="dashboard-app">
        <!-- Vue组件挂载点 -->
      </div>
    </div>
```

```
  </div>
</div>
```

2. **数据可视化**：使用Chart.js库创建网络延迟、区块生成速率等指标的动态图表。

```
// 创建网络延迟图表
function createLatencyChart(ctx, data) {
  return new Chart(ctx, {
    type: 'line',
    data: {
      labels: data.timestamps,
      datasets: [{
        label: '平均延迟 (ms)',
        data: data.values,
        borderColor: 'rgba(75, 192, 192, 1)',
        tension: 0.1
      }]
    },
    options: {
      responsive: true,
      scales: {
        y: {
          beginAtZero: true
        }
      }
    }
  });
}
```

3. **实时数据更新**：使用轮询和WebSocket技术实现面板数据的实时刷新。

```
// 定期轮询API获取最新数据
function pollData() {
  // 获取节点信息
  fetch('/peers')
    .then(response => response.json())
    .then(data => {
      app.peers = data;
    });

  // 获取区块链数据
  fetch('/blocks')
    .then(response => response.json())
    .then(data => {
      app.blocks = data;
      updateBlockchainChart(data);
    });

  // 其他数据轮询...

  // 5秒后再次轮询
  setTimeout(pollData, 5000);
}
```

4. **交互式节点网络图**：使用D3.js或Vis.js创建可交互的网络拓扑图，直观展示节点连接关系。

```
// 创建网络拓扑图
function createNetworkGraph(container, nodes, edges) {
  const data = {
    nodes: nodes.map(node => ({
      id: node.id,
      label: `Node ${node.id}`,
      color: node.status === 'ALIVE' ? '#4CAF50' : '#F44336',
      shape: node.nat ? 'diamond' : 'circle'
    })),
    edges: edges.map(edge => ({
      from: edge.source,
      to: edge.target,
      width: 1,
      length: edge.latency || 200
    }))
  };

  const options = {
    physics: {
      stabilization: false,
      barnesHut: {
        gravitationalConstant: -80000,
        centralGravity: 0.3,
        springLength: 95,
        springConstant: 0.04
      }
    }
  };

  return new vis.Network(container, data, options);
}
```

5. **响应式设计**：确保仪表盘在不同设备和屏幕尺寸上都能良好工作。

```
/* 响应式样式设计 */
@media (max-width: 768px) {
  .sidebar {
    position: static;
    width: 100%;
    margin-bottom: 20px;
  }

  .main-content {
    margin-left: 0;
  }

  .card {
    margin-bottom: 15px;
  }
}
```

通过这些技术的结合，仪表盘为用户提供了一个直观、信息丰富的界面，可以实时监控区块链网络的状态和性能指标。

Bonus

动态区块链网络

实现思路

动态区块链网络是对基础P2P网络的一个重要扩展，它允许节点在运行时动态地加入和离开网络，而不需要重启整个系统。这种机制使得区块链网络更接近真实的分布式环境，提高了系统的灵活性和可用性。

实现动态区块链网络的核心思想包括以下几个方面：

1. **节点发现与加入机制**：新节点需要有一种方式发现并加入现有网络。
2. **配置动态管理**：系统需要能够动态更新配置，以适应节点的加入和离开。
3. **状态同步**：新加入的节点需要能够快速同步当前的区块链状态。
4. **优雅退出**：节点离开网络时，需要有机制确保其未处理的交易能够被其他节点接管。

动态节点管理器

`dynamic_node_manager.py`是实现动态节点管理的核心模块，它主要负责节点的动态加入和配置更新：

```
def update_global_config(peer_id, ip, port, flags=None):
    """更新全局配置文件，添加新节点"""
    if flags is None:
        flags = {}

    try:
        # 读取当前配置
        with open('config.json', 'r') as f:
            config = json.load(f)

        # 添加新节点配置
        config["peers"][str(peer_id)] = {
            "ip": ip,
            "port": port,
            "fanout": 1
        }

        # 添加标志
        if flags.get("nat", False):
            config["peers"][str(peer_id)]["nat"] = True
        if flags.get("light", False):
            config["peers"][str(peer_id)]["light"] = True

        # 写回配置文件
        with open('config.json', 'w') as f:
            json.dump(config, f, indent=2)

        logger.info(f"全局配置已更新，添加节点 {peer_id}")
    except Exception as e:
```

```
        logger.error(f"更新全局配置失败: {e}")
```

通过这个函数，系统可以将新加入的节点信息添加到全局配置文件中，使其他节点也能发现并与之通信。

节点发现协议扩展

针对动态节点的需求，我扩展了节点发现协议，在原有HELLO消息的基础上添加了新的参数和处理逻辑：

```
def handle_hello_message(msg, self_id):
    sender_id = msg.get("sender_id")
    sender_ip = msg.get("ip")
    sender_port = msg.get("port")
    sender_flags = msg.get("flags", {})

    # 检查是否是新节点
    is_new_node = sender_flags.get("new_node", False)

    # 不处理自己的HELLO
    if sender_id == self_id:
        return []

    new_peers = []

    # 添加到已知节点表
    if sender_id not in known_peers:
        known_peers[sender_id] = (sender_ip, sender_port)
        new_peers.append(sender_id)

        # 如果是新节点，向其他所有节点广播NEW_PEER消息
        if is_new_node:
            broadcast_new_peer(sender_id, sender_ip, sender_port, sender_flags)

    # 无论节点是否已知，都更新flags信息
    peer_flags[sender_id] = {
        "nat": sender_flags.get("nat", False),
        "light": sender_flags.get("light", False)
    }

    # 可达性更新
    if sender_id not in reachable_by:
        reachable_by[sender_id] = set()

    reachable_by[sender_id].add(self_id)

    # 对于新节点，主动更新全局配置
    if is_new_node:
        from dynamic_node_manager import update_global_config
        update_global_config(sender_id, sender_ip, sender_port, sender_flags)

    return new_peers
```

当检测到新节点加入时，系统会广播这一信息，确保所有现有节点都能及时了解网络拓扑的变化。

配置动态更新

为了保持系统配置的一致性，我实现了配置的定期检查和更新：

```
def update_dynamic_config():
    """更新动态配置，确保所有节点的配置一致"""
    from peer_discovery import known_peers, peer_flags

    try:
        # 读取当前配置
        with open('config.json', 'r') as f:
            config = json.load(f)

        # 确保配置中包含所有已知节点
        for peer_id, (ip, port) in known_peers.items():
            if peer_id not in config["peers"]:
                config["peers"][peer_id] = {
                    "ip": ip,
                    "port": int(port),
                    "fanout": 1
                }

                # 添加标志
                if peer_id in peer_flags:
                    flags = peer_flags[peer_id]
                    if flags.get("nat", False):
                        config["peers"][peer_id]["nat"] = True
                    if flags.get("light", False):
                        config["peers"][peer_id]["light"] = True

        # 写回配置文件
        with open('config.json', 'w') as f:
            json.dump(config, f, indent=2)

        logger.info("动态配置已更新")
    except Exception as e:
        logger.error(f"更新动态配置失败: {e}")
```

这个函数会定期执行，确保配置文件中包含所有已知的节点信息。

状态同步优化

新节点加入网络后需要快速同步区块链状态，为此我实现了优化的区块同步请求：

```
def request_block_sync(self_id, is_new_node=False):
    """请求区块同步，新节点有特殊处理"""
    from block_handler import get_latest_block_height
    from outbox import enqueue_message

    current_height = get_latest_block_height()

    # 选择几个稳定节点作为同步源
    sync_sources = []
    for peer_id, (ip, port) in known_peers.items():
        if peer_id != self_id and peer_status.get(peer_id) == 'ALIVE':
```

```
            sync_sources.append((peer_id, ip, port))
            if len(sync_sources) >= 3:
                break

    if is_new_node and current_height == 0:
        # 新节点的初始同步 - 分批请求区块
        batch_size = 100  # 每批请求的区块数

        for source_id, source_ip, source_port in sync_sources:
            msg = {
                "type": "GET_BLOCK_HEADERS",
                "sender_id": self_id,
                "start_height": 0,
                "end_height": batch_size - 1,
                "message_id": generate_message_id()
            }
            enqueue_message(source_id, source_ip, source_port, msg)
    else:
        # 常规同步 - 请求最新区块
        for source_id, source_ip, source_port in sync_sources:
            msg = {
                "type": "GET_LATEST_BLOCK",
                "sender_id": self_id,
                "current_height": current_height,
                "message_id": generate_message_id()
            }
            enqueue_message(source_id, source_ip, source_port, msg)
```

对于新节点，系统会采用分批请求的方式加速初始同步过程。

节点优雅退出

为了确保节点可以优雅地退出网络，我实现了GOODBYE消息处理：

```
def send_goodbye_message(self_id, reason="normal_shutdown"):
    """发送优雅退出通知，并转发未确认交易"""
    from outbox import enqueue_message
    from transaction import get_recent_transactions

    # 获取本地交易池中的交易
    pending_txs = get_recent_transactions()

    msg = {
        "type": "GOODBYE",
        "sender_id": self_id,
        "reason": reason,
        "pending_transactions": pending_txs if len(pending_txs) <= 100 else [],
        "has_more_transactions": len(pending_txs) > 100,
        "message_id": generate_message_id(),
        "timestamp": time.time()
    }

    # 向所有已知节点广播退出消息
    for peer_id, (peer_ip, peer_port) in known_peers.items():
        if peer_id != self_id:
```

```
            enqueue_message(peer_id, peer_ip, peer_port, msg)

    # 如果交易太多，发送MEMPOOL_TRANSFER消息
    if len(pending_txs) > 100:
        # 将交易分批发送给几个活跃节点
        batch_size = 100
        active_peers = [peer_id for peer_id, status in peer_status.items()
                        if status == 'ALIVE' and peer_id != self_id]

        if active_peers:
            selected_peers = random.sample(active_peers, min(3,
len(active_peers)))

            for i in range(0, len(pending_txs), batch_size):
                batch = pending_txs[i:i+batch_size]
                transfer_msg = {
                    "type": "MEMPOOL_TRANSFER",
                    "sender_id": self_id,
                    "transactions": batch,
                    "batch": i // batch_size + 1,
                    "total_batches": (len(pending_txs) + batch_size - 1) //
batch_size,
                    "message_id": generate_message_id()
                }

                for peer_id in selected_peers:
                    if peer_id in known_peers:
                        peer_ip, peer_port = known_peers[peer_id]
                        enqueue_message(peer_id, peer_ip, peer_port,
transfer_msg)
```

这个功能确保了节点在退出前能够通知其他节点，并将其未处理的交易转发出去，避免交易丢失。

运行方式

要在现有区块链网络中添加新节点，需要按照以下步骤操作：

1. 首先，确保现有节点已启动并设置了`--dynamic`参数，以启用动态节点支持：

```
docker-compose run -d --name peer5000 -p 6000:5000 -p 8000:7000 peer5000 python
node.py --id 5000 --config config.json --dynamic
```

2. 然后，可以使用以下命令创建并启动新节点：

```
# Windows环境
docker run -d --name peer5022 --network starter_code_new_default -p 6022:5022 -p
8022:7022 starter_code_new-peer5000:latest python node.py --id 5022 --config
config.json --dynamic

# Linux/macOS环境
docker run -d --name peer5022 --hostname peer5022 --network
starter_code_new_default -p 6022:5022 -p 8022:7022 starter_code_new-
peer5000:latest python node.py --id 5022 --config config.json --dynamic
```

3. 新节点启动后，它会自动发现并加入现有网络：

- 首先，它会向配置文件中的初始节点发送HELLO消息
- 然后，它会接收并处理其他节点返回的信息，建立对整个网络的了解
- 最后，它会请求并同步区块链数据，赶上网络的当前状态

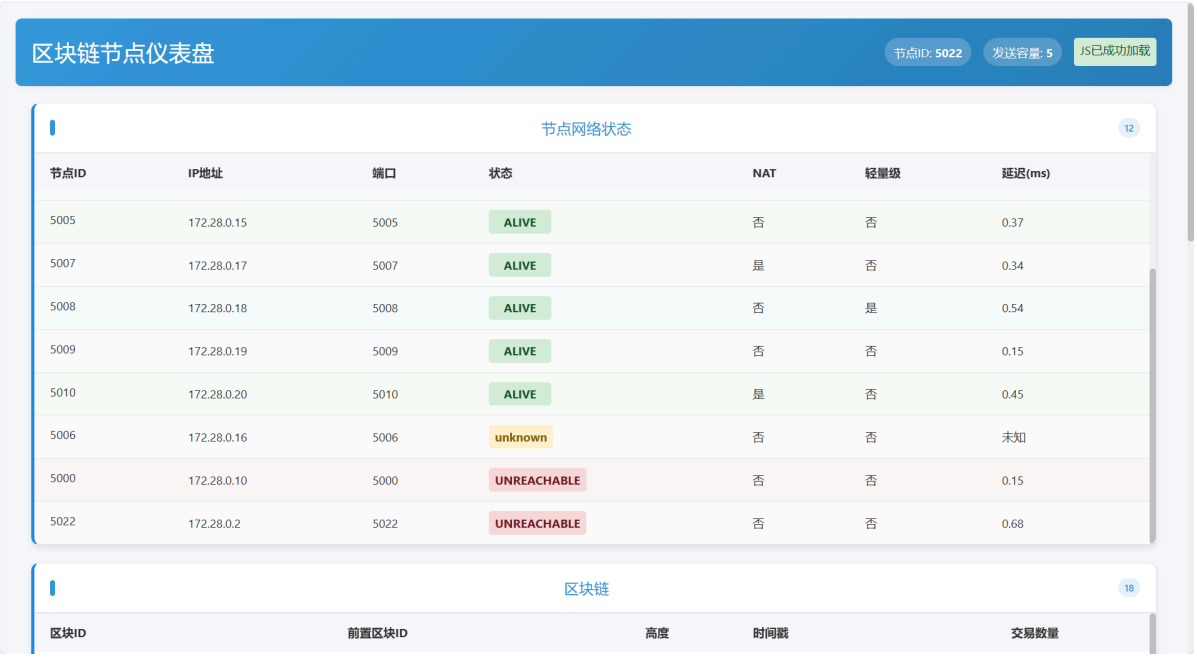
4. 节点也可以通过以下命令安全退出网络：

```
docker stop peer5022
```

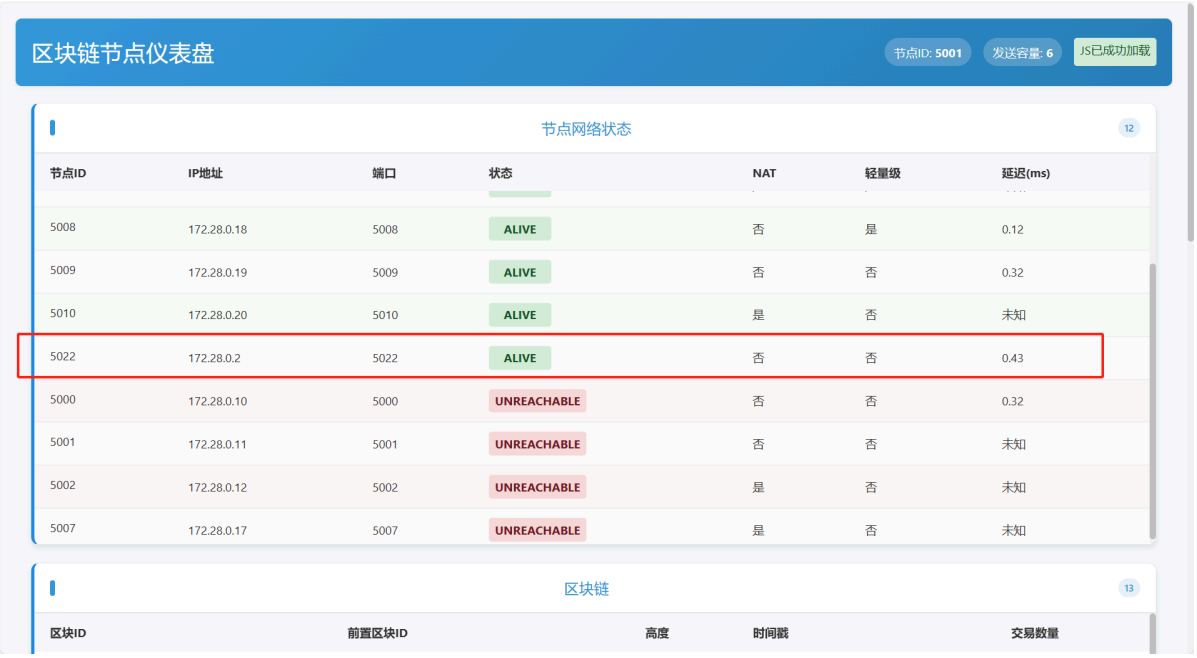
在容器停止前，节点会尝试发送GOODBYE消息并转移其未处理的交易。

结果展示

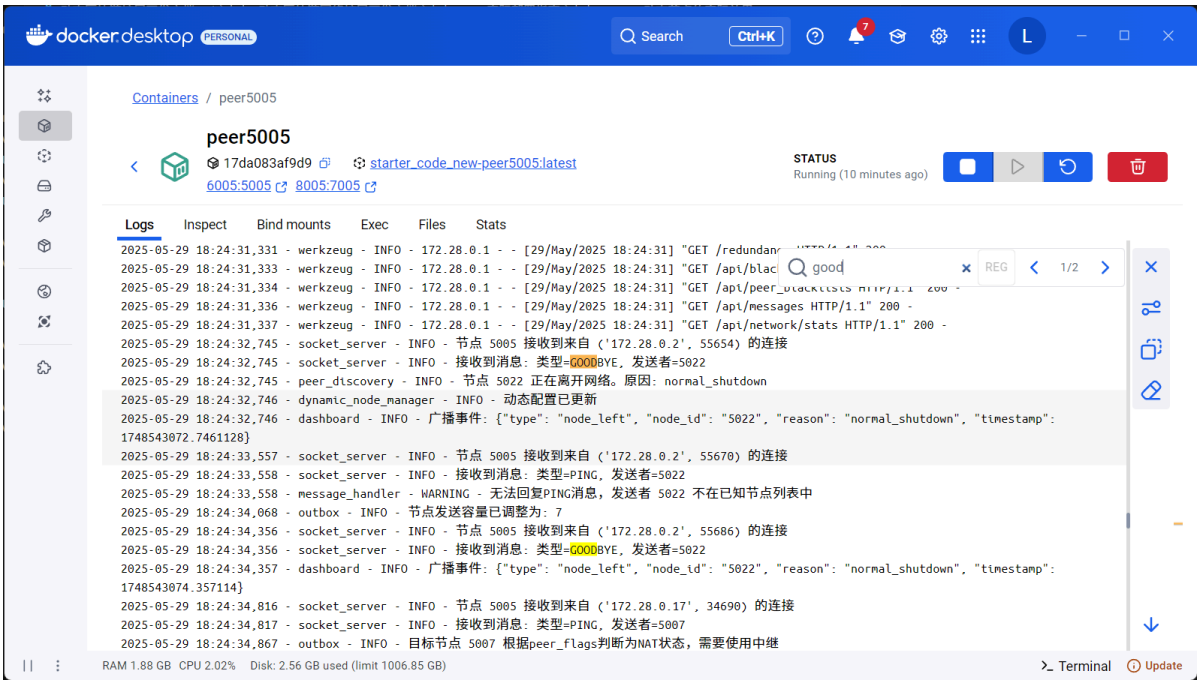
- 新节点`5022`被正确添加到网络中，并可以正确发现其他节点



- 新节点`5022`能被其他节点发现并且添加到各自的peers列表中



- 节点`5022`离开时，其他节点可以收到其`goodbye`消息并正确删除相关数据

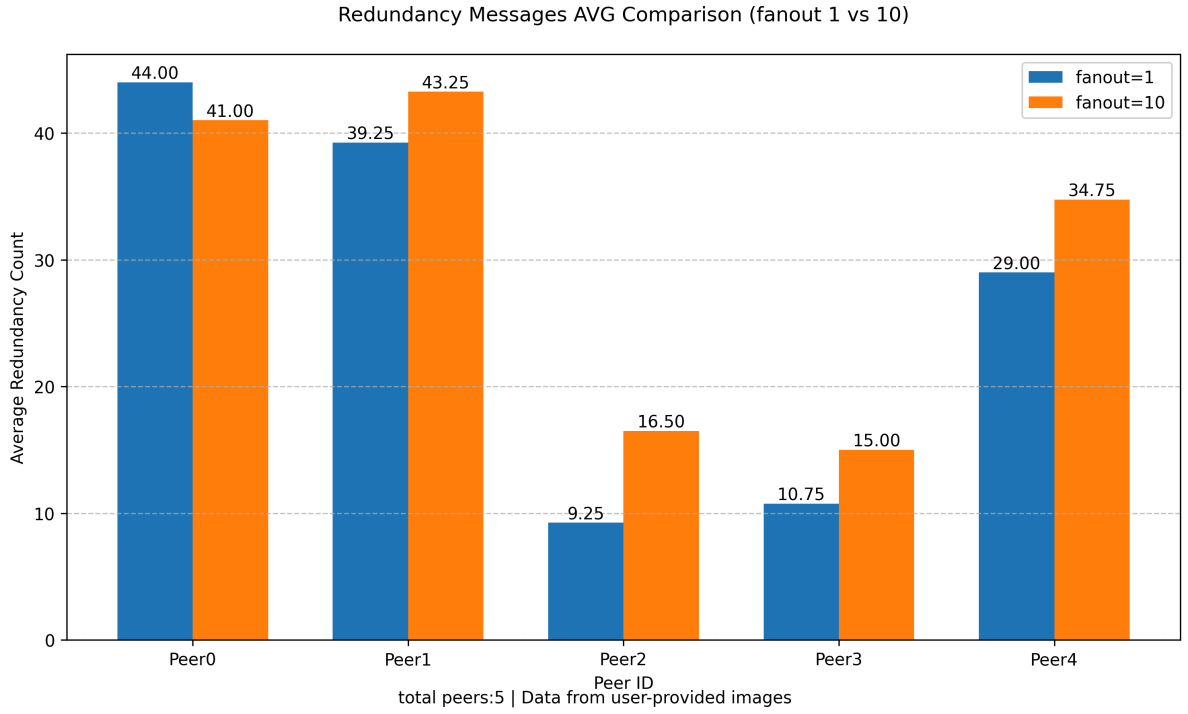


Bonus2

fanout参数变化对冗余消息的影响

本实验采用五个节点（peer0-4），控制时间在3分钟、节点总数为11个的情况下，分别对fanout为1和10测试4组，以下为实验数据以及用用`Python`程序绘制的图表：

fanout	Peer0	Peer1	Peer2	Peer3	Peer4	AVG
1	49	39	14	12	24	
1	50	37	3	10	21	
1	43	36	14	4	31	
1	34	45	6	17	40	
AVG	44	39.25	9.25	10.75	29	26.25
fanout	Peer0	Peer1	Peer2	Peer3	Peer4	AVG
10	48	39	20	17	32	
10	33	39	15	18	43	
10	38	39	17	20	36	
10	45	56	14	5	28	
AVG	41	43.25	16.5	15	34.75	30.1
3min						
total peers:11						



实验结论：

- **较高的 `fanout` 会导致更多的冗余消息**：

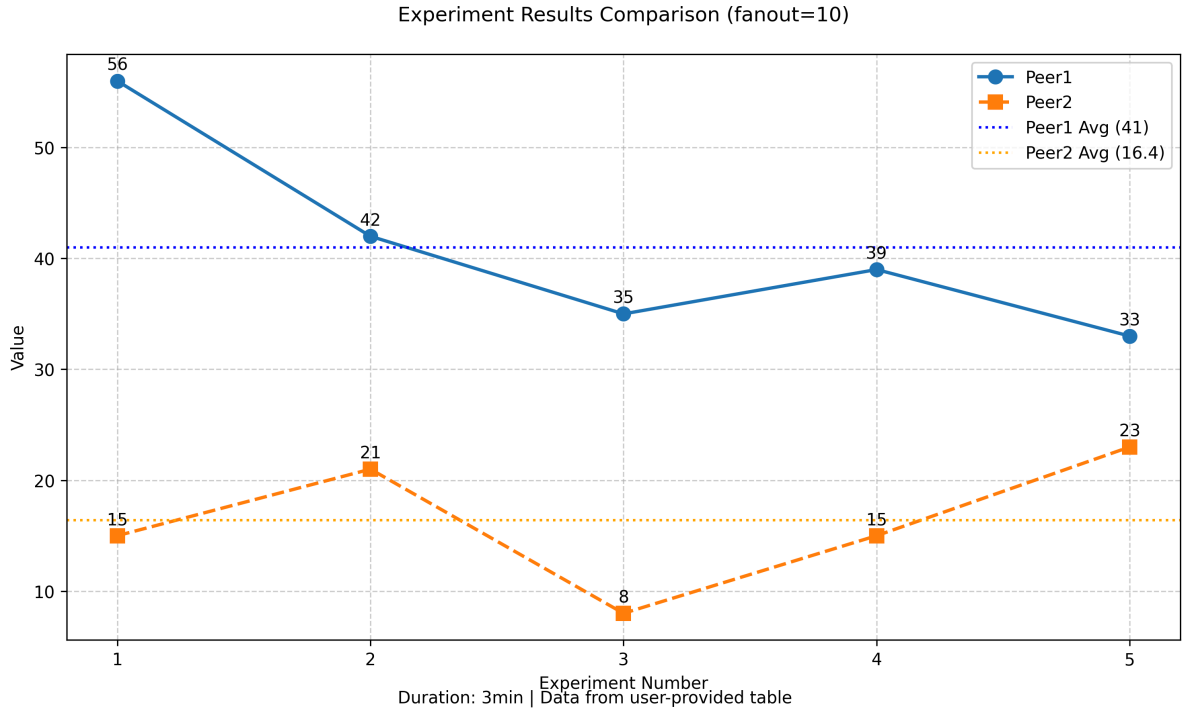
  在 `fanout=10` 的情况下，相比 `fanout=1`，冗余消息的数量显著增加。这表明，较高的 fanout 值会将消息传播到更多的对等节点，从而导致更多的冗余消息在网络中传输。

- `NAT=true` 的节点对 `fanout` 的敏感程度更高，Peer2和Peer3的增长幅度分别为78.4%和39.5%

有无NAT对冗余消息的影响

本实验在fanout=10的情况下，控制时间为3分钟，分别对Peer1 (NAT=False) 和 Peer2 (NAT=True) 测试5次，以下为实验数据以及用用 `Python` 程序绘制的图表：

实验次数	Peer1	Peer2
1	56	15
2	42	21
3	35	8
4	39	15
5	33	23
AVG	41	16.4
fanout=10		
3min		



实验结论：

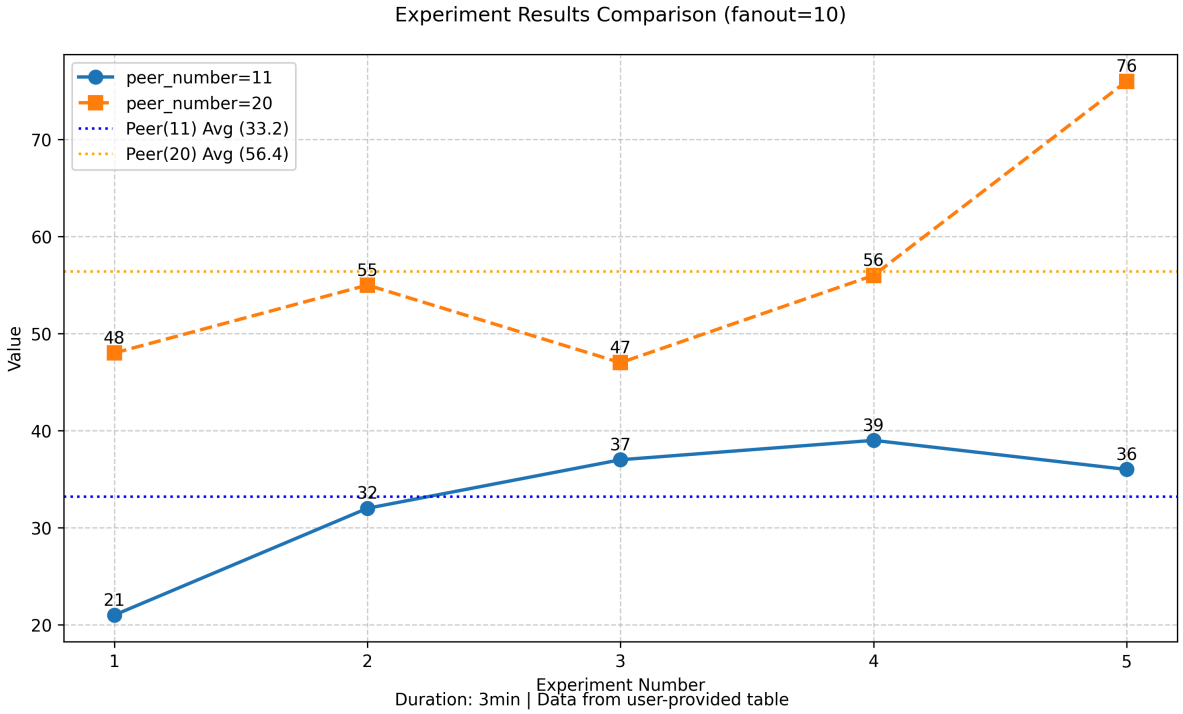
`NAT=True`时冗余消息更少，分析原因如下：

1. **NAT 限制了消息传播的范围**：
    - 在 NAT 环境中，网络地址转换会限制节点与外部网络的直接连接，因此 **Peer2** 在接收到的消息传播上会受到限制。NAT 会导致对等节点的消息只会在特定的网络范围内传播，减少了消息的冗余传播。
2. **节点之间的直接通信受限**：
    - 没有 NAT 的节点（如 **Peer1**）通常能更容易地直接与其他节点建立连接并交换信息，从而导致更多的冗余消息（由于信息传播给更多的对等节点）。
    - 但是有 NAT 的节点通常需要通过某些中介节点或路由器进行通信，消息传播的范围减少，这减少了冗余消息的数量。
3. **NAT 防止了多路径传播**：
    - 在没有 NAT 的情况下，消息可能会通过多个路径传播到其他节点，从而增加冗余消息的数量。而有 NAT 的节点通常只能通过有限的路径传播消息，导致冗余消息的传播受到控制。

节点总数对冗余消息的影响

本实验对Peer0，在fanout=10，时间控制在3分钟的情况下，分别在总节点数为11和20的情况下测试5组，以下为实验数据以及用用`Python`程序绘制的图表：

实验次数	节点数11	节点数20
1	23	48
2	31	55
3	37	47
4	39	56
5	36	76
AVG	33.2	56.4
Peer0		
fanout=10		
3min		



实验结论：

节点数为20时产生的冗余消息比节点数为11产生的冗余消息多，分析原因如下：

1. **更多的节点导致更多的消息传播**：
    - 当节点数量增加时，消息会传播给更多的节点。对于每个消息，多个节点接收到消息后，消息可能会被进一步广播给更多的对等节点，导致冗余消息数目增加。特别是在一个去中心化的网络中，增加节点数目意味着消息传播范围扩大，从而产生更多的冗余消息。
2. **消息传递的覆盖率更广**：

- 当网络中的节点更多时，每个节点的消息覆盖范围也变得更广，导致每条消息被更多的节点复制和传播。这直接导致了冗余消息的数量增加，因为相同的消息会在更多节点之间传播。

3. **信息传播的复杂性增加**：
   - 当网络中的节点增加时，信息的传播和管理变得更为复杂，特别是当不同节点之间存在不同的传播路径时。节点数越多，冗余消息被重复传递的可能性就越高。