一个简化版的 PBFT共识系统系统设计模板

1. 系统架构

- **节点类型**：每个节点是一个全节点。
- **网络结构**：采用部分去中心化结构，由多个节点组成，没有中心化的管理者。节点之间通过点对点的通信建立P2P网络。
- **节点间通信协议**：采用基于HTTP的异步通信协议。节点之间通过发送HTTP POST请求进行通信。节点之间交换的消息包括请求消息、预准备消息、准备消息和提交消息。所有消息都是以JSON格式进行编码和解码。

2. 消息格式设计

- **请求消息**

```
json_data = {
    'id': (self._client_id, i),
    'client_url': self._client_url + "/" + Client.REPLY,
    'timestamp': time.time(),
    'data': str(i)
}
```

- **预准备消息**

```
preprepare_msg = {
    'leader': self._index,
    'view': self._view.get_view(),
    'proposal': {
        this_slot: json_data
    },
    'type': 'preprepare'
}
```

- **准备消息**

```
prepare_msg = {
    'index': self._index,
    'view': json_data['view'],
    'proposal': {
        slot: json_data['proposal'][slot]
    },
    'type': Status.PREPARE
}
```

- **提交消息**

```
commit_msg = {
    'index': self._index,
    'view': json_data['view'],
    'proposal': {
        slot: json_data['proposal'][slot]
    },
    'type': Status.COMMIT
}
```

3. 共识流程

1. **客户端发送请求**:

```
def request(self):
    # 如果没有 session，创建一个新的 session
    if not self._session:
        创建一个新的 aiohttp.ClientSession 对象，设置超时时间为
self._resend_interval
    # 发送 self._num_messages 条消息
    for i in range(self._num_messages):
        设置累计失败次数为 0
        标记消息是否成功发送为 False
        设置目标节点下标为 0
        创建一个 asyncio.Event 对象用于标记请求是否成功
        # 每次成功发送消息后，等待 0 到 1 秒
        await asyncio.sleep(random())
        构建消息数据
        将消息数据发送到目标节点
        # 等待请求成功或超时
        try:
            等待 self._is_request_succeed 的状态变为 True，超时时间为
self._resend_interval
        except:
            标记消息发送失败
            记录日志
            增加累计失败次数
            如果累计失败次数达到重试次数上限:
                请求视图更改
                等待 0 到 1 秒
                重置累计失败次数为 0
                更新目标节点下标
        else:
            标记消息发送成功
            如果消息发送成功:
                跳出循环
    关闭 session
```

2. **主节点预准备**：主节点接收请求并生成预准备消息，广播给所有副本节点。

```
def get_request(self, request):
    记录日志
    如果不是 leader:
        如果知道当前的 leader:
            抛出临时重定向异常，重定向到当前 leader
        否则:
            抛出服务不可用异常
    否则:
        解析请求数据为 JSON 格式
        执行预准备操作
```

3. **副本节点准备**：副本节点接收预准备消息，验证后生成准备消息并广播。

```
async def preprepare(self, json_data):
    获取当前提议的槽位号
    记录日志
    如果该槽位号对应的状态信息不存在:
        创建一个新的状态信息并存储在状态字典中
    将请求信息存储在对应槽位的状态信息中
    构建预准备消息
    发送预准备消息给其他副本节点
```

4. **共识验证**：副本节点收集2f+1个相同的准备消息后，生成提交消息并广播。

```
async def commit(self, request):
    解析接收到的 prepare 消息
    记录日志

    如果接收到的消息中的视图小于跟随视图:
        # 当接收到的消息视图小于跟随视图时，不进行任何操作
        PASS

    记录日志

    对于消息中的每个槽位:
        如果该槽位不合法:
            继续下一个槽位的处理

        如果该槽位对应的状态信息不存在:
            创建一个新的状态信息并存储在状态字典中
        获取该槽位对应的状态信息
        创建视图对象

        更新状态信息的序列号信息
        如果状态信息满足大多数准备条件:
            更新状态信息的准备证书
            构建提交消息
            发送提交消息给其他副本节点
```

5. **提交数据**：副本节点收集2f+1个相同的提交消息后，将消息内容m持久化保存。

```
async def reply(self, request):
    解析接收到的 commit 消息
    记录日志
```

```
    如果接收到的消息中的视图小于跟随视图：
        # 当接收到的消息视图小于跟随视图时，不进行任何操作
        返回空的 web.Response()

    记录日志

    对于消息中的每个槽位：
        如果该槽位不合法：
            继续下一个槽位的处理

        如果该槽位对应的状态信息不存在：
            创建一个新的状态信息并存储在状态字典中
        获取该槽位对应的状态信息
        创建视图对象

        更新状态信息的序列号信息
        如果状态信息没有提交证书且满足大多数提交条件：
            更新状态信息的提交证书
            记录日志
            如果上一个提交的槽位是当前槽位的前一个槽位且该槽位尚未提交：
                构建回复消息
                标记该槽位已提交
                更新最后提交的槽位
                # 当提交消息填满下一个检查点时，提议一个新的检查点
                如果下一个检查点已经填满：
                    提议一个新的检查点
                    记录日志
                执行提交操作
                尝试将回复消息发送给客户端
                记录日志
```

4. 容错与安全性

如果系统有 3f + 1 个节点，且其中最多 f 个节点是拜占庭节点，那么系统就能够容忍 f 个拜占庭节点的恶意行为。证明：

```
设总结点数为N，作恶的拜占庭节点数为 f，法定人数为Q
要满足liveness必须有
Q <= N - f
说明：如果共识算法需要的Q大于N-f，则当f个拜占庭故障节点都主动破坏时，算法必然不能执行下去
要满足safety必须有
2Q - N > f
说明：任何两个quorum的交集（2Q - N）中必须有非拜占庭故障节点。如果不满足2Q - N <= f，此时f个节点同时加入到两个Quorum中说不同的话，系统内会同时通过两个不同的意见，此时系统一致性无法满足
因此
N + f < 2Q <= 2(N - f)
N > 3f
if N=3f+1 此时Q的最小值为
N + f < 2Q
3f + 1 + f < 2Q
2f + 1/2 < Q
Q_min = 2f + 1
证毕
```

- **消息验证**：每个节点在接收消息时验证消息的有效性，包括视图编号、序列号和消息摘要。

```
def _update_sequence(self, view, proposal, from_node):
    使用 MD5 对提议进行哈希
    生成键值对 (视图编号, 提议的哈希值)
    如果键值对不在回复消息字典中:
        创建一个新的序列元素，并将其添加到回复消息字典中
    将来源节点添加到对应键值对的序列元素中的来源节点集合中
```

- **视图更换**：如果主节点失效，副本节点在一定时间后触发视图更换，选举新的主节点。

```
async def request_view_change(self):
    构建视图更改消息数据
    对于节点列表中的每个节点:
        尝试发送视图更改消息给节点
        如果发送失败:
            记录日志，标记发送失败
        否则:
            记录日志，标记发送成功
```

5. 性能优化

- **Request Batching**:
  使用滑动窗口机制来限制可以并行运行 的协议实例的数量。令 e 是 primary 执行的最后一批请求的序列号，令 p 是 primary 发送的 最后一个 PRE-PREPARE 的序列号。当 primary 接收到一个请求时，它会立即启动协议，除非 p ≥ e + W，其中 W 是窗口大小。在后一种情况下，它会排队请求。当请求执行时，窗口 向前滑动以允许排队的请求进行处理。然后，primary 从队列中取出第一个请求，使得它们 的大小之和低于常量边界，它为它们分配序列号，并将它们发送到单个 PRE-PREPARE 消息中。 该协议的进行方式与单个请求完全相同，除了副本执行批次的请求(按照它们被添加到 PRE-PREPARE 消息的顺序)和它们为每个请求分别发送回去的回复。
- **Digest Replies** :
  一个客户端请求指定一个副本发送结果。该副本可以随机地或使用某种 其他负载平衡方案来选择。在指定的副本执行请求之后，它会发送回来一个包含结果的回复。 其他副本只发送包含结果摘要的回复。客户端收集至少 f + 1 个回复(包括具有结果的那个 回复)，并使用摘要来检查结果的正确性。如果客户端没有从指定的副本收到正确的结果， 它则会重新发送请求(像往常一样)请求所有副本发送带有结果的回复。
- **Tentative Execution**:

  将操作调用的消息延迟数从 5 减少到 4 个。 一旦副本为请求准备好一个证书;它们的状态反射所有有较低序列号的请求的执行;以及这 些请求已被提交，则这些副本立即临时地执行请求。执行请求后，副本会向客户端发送临时 回复。由于回复是暂时的，客户端必须等待具有相同结果的答复的 quorum certificate。这确 保了请求是由 quorum 准备的，因此，它被保证最终在非故障副本上提交。如果客户端的重 传定时器在接收到这些回复之前到期，则客户端重新发送请求并等待非临时的回复。
- **Read-Only Operations**:
  客户端将只读请求多播到所有副本。副本在检查请求是否被正确认证后立即执行，客 户端具有访问权限，并且该请求实际上是只读的。只有在只读请求提交之前所有请求执行之后，副本才会发送回来一条回复。客户端等待具有相同结果的答复的 quorum certificate。如 果对影响结果的数据进行并发写入，则可能无法收集到此证书。在这种情况下，在重发定时 器超时之后，它将该请求作为常规读写请求重传。

6. 安全性保证

- **数字签名**：每个消息都包含发送者的数字签名，以确保消息的不可抵赖性和完整性。

```
def generate_key_pair():
    使用 RSA 算法生成一个私钥，其中：
        公共指数设置为 65537，
        密钥长度设置为 2048 位，
        使用默认后端
    生成相应的公钥
    返回私钥和公钥

def sign_message(private_key, message):
    使用私钥对消息进行签名，其中：
        签名算法使用 SHA256，
        填充方案使用 PSS 填充，其中：
            MGF 使用 MGF1，哈希算法为 SHA256，
            盐的长度设置为 PSS.MAX_LENGTH
    返回签名结果

def verify_signature(public_key, signature, message):
    尝试使用公钥验证签名，其中：
        签名算法使用 SHA256，
        填充方案使用 PSS 填充，其中：
            MGF 使用 MGF1，哈希算法为 SHA256，
            盐的长度设置为 PSS.MAX_LENGTH
    如果验证成功，返回 True，否则返回 False
```

- **消息摘要**：使用哈希函数为消息计算摘要，以快速验证消息内容的一致性。

```
def calculate_digest(message):
    使用 SHA256 算法计算消息的摘要
    返回摘要的十六进制表示
```

7. 系统部署

- **节点配置**：每个节点配置有其公钥和私钥对，用于签名和验证消息。
- **网络设置**：建立P2P网络，确保节点间可以进行直接通信。

```
# 节点类
class Node:
    def __init__(self, node_id, public_key, private_key):
        self.node_id = node_id
        self.public_key = public_key
        self.private_key = private_key
        self.peers = []  # 与之相连的其他节点

    def send_message(self, message, recipient):
        # 使用私钥对消息进行签名
        signature = sign(message, self.private_key)
        # 将消息和签名发送给接收者
        recipient.receive_message(message, signature, self)

    def receive_message(self, message, signature, sender):
        # 验证消息的签名
        if verify(message, signature, sender.public_key):
            # 处理接收到的消息
            process_message(message)
```

```
        else:
            # 消息签名不合法，可能是恶意节点发送的伪造消息
            handle_invalid_message()

    def add_peer(self, peer):
        # 添加一个相邻的节点
        self.peers.append(peer)

# 处理接收到的消息
def process_message(message):
    # 对接收到的消息进行处理
    pass

# 处理签名不合法的消息
def handle_invalid_message():
    # 对签名不合法的消息进行处理
    pass
```

8. 实现考虑

- **状态机**：每个副本节点维护一个状态机，以保证请求的顺序执行。状态机对于消息的状态转移图

```
class StateMachine:
    def __init__(self):
        self.state = INITIAL_STATE
        self.last_applied = 0  # 上一次应用的请求序列号
        self.message_log = {}  # 用于存储已提交的消息
    def apply(self, message):
        """
        应用消息到状态机
        """
        if message.sequence_number == self.last_applied + 1:
            # 序列号是下一个预期的
            self.state_transition(message)
            self.last_applied += 1
            self.message_log[message.sequence_number] = message
            return True
        elif message.sequence_number <= self.last_applied:
            # 已经应用了该序列号的消息
            return True
        else:
            # 丢失了前面的消息，无法应用当前消息
            return False
    def state_transition(self, message):
        """
        根据收到的消息进行状态转移
        """
        # 根据消息类型执行相应的状态转移操作
        if message.type == 'REQUEST':
            # 处理请求消息
            pass
        elif message.type == 'PRE_PREPARE':
            # 处理预准备消息
            pass
        elif message.type == 'PREPARE':
            # 处理准备消息
            pass
```

```
        elif message.type == 'COMMIT':
            # 处理提交消息
            pass
    def get_last_applied(self):
        """
        获取上次应用的消息序列号
        """
        return self.last_applied

    def get_message_by_sequence_number(self, sequence_number):
        """
        根据消息序列号获取消息
        """
        return self.message_log.get(sequence_number)
```

- **持久化存储**：副本节点需要持久化存储已提交的消息，以防止数据丢失。

```
class PersistenceStorage:
    def __init__(self):
        self.data = {}
    def store_message(self, message):
        self.data[message.id] = message

    def retrieve_message(self, message_id):
        return self.data.get(message_id)

    def delete_message(self, message_id):
        if message_id in self.data:
            del self.data[message_id]
```

流程图：