共识

1. **共识问题的应用场景**
   - **总序广播**：确保消息以相同的顺序被所有进程接收。
   - **状态机复制**：在多个服务器之间复制服务状态，以实现高可用性和容错性。
   - **原子提交**：在分布式数据库事务中确保所有变更要么全部提交，要么全部撤销。
   - **终止可靠广播**：确保消息被所有正确的进程接收，即使有进程可能崩溃。

Consensus

属性

1. **请求（Request）**
   - `< c, Propose | v >`：提出一个值`v`作为共识的候选值。当一个进程希望其他进程考虑这个值时，会发起这个请求。
2. **指示（Indication）**
   - `< c, Decide | v >`：输出共识决定的值`v`。当进程就某个值达成共识后，它会发出这个指示。
3. **属性（Properties）**
   - **C1. 终止（Termination）**：每个正确的进程最终都会决定某个值。这保证了共识过程能够完成，而不会无限期地等待。
   - **C2. 有效性（Validity）**：如果一个进程决定了值`v`，那么`v`必须是由某个进程提出的。这避免了决定了一个从未被提出的值的情况。
   - **C3. 完整性（Integrity）**：没有进程会决定两次。这确保了一旦进程做出决定，它就不会改变这个决定。
   - **C4. 一致性（Agreement）**：没有两个正确的进程会做出不同的决定。这是共识问题的核心，确保所有正确的进程都会就同一个值达成一致。

实现

```
Hierarchical Consensus
Implements: Consensus, instance c.
Uses: BestEffortBroadcast, instance beb; PerfectFailureDetector, instance P.

# 当共识实例初始化时触发的事件
upon event < c, Init > do
    detectedranks := ∅;  # 初始化已检测到的崩溃进程的排名集合
    round := 1;  # 初始化轮次为1
    proposal := ⊥;  # 初始化提案为未定义
    proposer := 0;  # 初始化提案者编号为0
    delivered := [FALSE]N;  # 初始化一个长度为N的数组，记录是否已经接收到决定
    broadcast := FALSE;  # 初始化广播状态为未广播

# 当检测到进程崩溃时触发的事件
upon event < P, Crash | p > do
```

```
    detectedranks := detectedranks ∪ {rank(p)};  # 将崩溃进程的排名添加到detectedranks
集合中

# 当有提案请求时触发的事件
upon event < c, Propose | v > such that proposal = ⊥ do
    proposal := v;  # 设置提案值为v

# 如果当前轮次是自己的排名，且提案不为空，且尚未广播，则广播提案
upon round = rank(self) ∧ proposal ≠ ⊥ ∧ broadcast = FALSE do
    broadcast := TRUE;  # 标记已广播
    trigger < beb, Broadcast | [DECIDED, proposal] >;  # 广播[DECIDED, proposal]消息
    trigger < c, Decide | proposal >;  # 触发决定事件，确定提案值

# 如果当前轮次在detectedranks集合中或已接收到该轮次的决定，则进入下一轮次
upon round ∈ detectedranks ∨ delivered[round] = TRUE do
    round := round + 1;  # 轮次递增

# 当通过尽力而为广播接收到决定消息时触发的事件
upon event < beb, Deliver | p, [DECIDED, v] > do
    r := rank(p);  # 获取发送者的排名
    if r < rank(self) ∧ r > proposer then
        proposal := v;  # 更新提案值
        proposer := r;  # 更新提案者排名
    delivered[r] := TRUE;  # 标记该排名的决定已接收
```

这个算法的主要思想是：

- 每个进程在自己的轮次中作为领导者，如果有提案，则广播这个提案。
- 进程在收到比自己排名低的其他进程的提案时，会更新自己的提案。
- 当一个进程崩溃或者已经广播了决定，其他进程将进入下一个轮次。
- 这样通过轮次和排名的机制来确保所有进程最终能就某个值达成共识。

正确性

共识算法的正确性论证通常需要证明其满足共识问题的核心属性之一：一致性（Agreement）。

1. **考虑特定的进程 `pi`**：
    - 假设 `pi` 是拥有最小排名的正确进程（就像是第一个提出建议的人）。假设 `pi` 决定了一个值 `v`（例如，提议去意大利餐厅）。
2. **论证一致性**：
    - **情况一（`i = n`）**：如果 `pi` 是唯一的正确进程（就像只有一个人提出了建议），那么显然不会有其他不同的决定。
    - **情况二（`i < n`）**：如果不止 `pi` 一个人提出建议，但 `pi` 是第一个提出建议的人，则在 `pi` 的轮次中，所有正确的进程（人）都会接收到 `pi` 的建议 `v`。由于这个建议是最先提出的，而且大家都同意了，所以之后的任何人都不会提出与 `v` 不同的建议。
3. **结论**：

- 无论是哪种情况，所有正确的进程（人）都不会决定与`pi`提出的`v`不同的值。这就证明了一致性：在任何执行序列中，不会有两个正确的进程做出不同的决定。

Uniform Consensus

属性

均匀共识（Uniform Consensus）这里称之为“uc”。均匀共识是共识问题的一个变体，其中的关键区别在于其一致性（Agreement）属性的定义。

1. **属性（Properties）**
   - **C4. 均匀一致性（Uniform Agreement）**：没有任何两个进程（无论是正确的还是已崩溃的）会做出不同的决定。这意味着所有进程（而不仅仅是正确的进程）都必须就相同的值达成一致。即使某些进程可能在决定过程中崩溃，它们在崩溃前做出的任何决定也必须与其他进程的决定保持一致。

均匀共识问题的核心在于其均匀一致性属性，即使是已崩溃的进程也不能与其他进程做出不同的决定。

实现

```
Implements: UniformConsensus, instance uc.
Uses: BestEffortBroadcast, instance beb; PerfectFailureDetector, instance P.
# 当均匀共识实例初始化时触发的事件
upon event < uc, Init > do
    correct := Π;  # 初始化所有进程的集合
    round := 1;  # 初始化当前轮次为1
    proposal := ⊥;  # 初始化提案为未定义
    proposer := 0;  # 初始化提案者编号为0
    delivered := [FALSE]N;  # 初始化一个长度为N的数组，记录是否已经接收到提案
    broadcast := FALSE;  # 初始化广播状态为未广播

# 当检测到进程崩溃时触发的事件
upon event < P, Crash | p > do
    detectedranks := detectedranks ∪ {rank(p)};  # 将崩溃进程的排名添加到detectedranks集合中

# 当有提案请求时触发的事件
upon event < uc, Propose | v > such that proposal = ⊥ do
    proposal := v;  # 设置提案值为v

# 如果当前轮次是自己的排名，且提案不为空，且尚未广播，则广播提案
upon round = rank(self) ∧ proposal ≠ ⊥ ∧ broadcast = FALSE do
    broadcast := TRUE;  # 标记已广播
    trigger < beb, Broadcast | [PROPOSAL, proposal] >;  # 广播[PROPOSAL, proposal]消息

# 当通过尽力而为广播接收到提案时触发的事件
upon event < beb, Deliver | p, [PROPOSAL, v] > do
    r := rank(p);  # 获取发送者的排名
    if r < rank(self) ∧ r > proposer then
        proposal := v;  # 更新提案值
```

```
        proposer := r;  # 更新提案者排名
    delivered[r] := TRUE;  # 标记该排名的提案已接收

# 如果当前轮次在detectedranks集合中或已接收到该轮次的提案，则进入下一轮次或做出决定
upon round ∈ detectedranks ∨ delivered[round] = TRUE do
    if round = n then
        trigger < uc, Decide | proposal >;  # 如果是最后一轮，则做出决定
    else
        round := round + 1;  # 否则，轮次递增
```

这个算法的主要思想是：

- 进程们通过轮次顺序交换提案，每个进程在其轮次广播自己的提案。
- 进程接收到的提案会根据特定条件（如排名）进行更新。
- 在最后一轮（第n轮）结束时，进程做出决定，决定的值是当前持有的提案。
- 这种机制确保了即使在进程可能崩溃的情况下，所有剩余的正确进程最终都能就一个值达成一致。

正确性

1. **选择一个特定的进程**：
   - 考虑拥有最小ID的那个做出决定的进程，假设它是`pi`。这意味着`pi`成功完成了第`n`轮。
2. `pi`**在第**`i`**轮的行为**：
   - 在第`i`轮中，所有ID大于`i`的进程`pj`（即`j > i`）都会接收到`pi`的提案并采纳它。这是因为`pi`是第一个完成所有轮次并做出决定的进程，它的提案被传递给了所有其他进程。
3. **提案的一致性**：
   - 这意味着在第`i`轮之后发送消息的每个进程在第`i`轮结束时都拥有相同的提案。因为每个进程都接收并采纳了来自`pi`的提案，所以它们都将这个提案作为自己的提案。
4. **均匀一致性的保证**：
   - 因此，即使在后续的轮次中，没有其他进程会有机会提出一个不同的提案，因为他们都已经采纳了`pi`的提案。
   - 这保证了即使某些进程崩溃或无法完成所有轮次，所有完成第`i`轮的进程都将拥有相同的提案。
5. **结论**：
   - 因此，所有的进程（包括那些可能在后续轮次中崩溃的进程）都将基于相同的提案做出决定，从而满足均匀一致性的要求。

总序广播（Total Order Broadcast）

1. **可靠广播（Reliable Broadcast）**
   - 在可靠广播中，进程可以以任何顺序交付消息，只要确保所有消息最终都被接收。
2. **因果广播（Causal Broadcast）**
   - 因果广播要求进程根据某种顺序（因果顺序）交付消息。这意味着如果消息A在消息B之前发送，并且B因果依赖于A，则B不能在A之前被交付。
   - 但是，这种顺序是部分的，因为因果无关的消息可能会被不同的进程以不同的顺序接收。

3. **总序广播（Total Order Broadcast）**
    - 在总序广播中，所有消息必须按照相同的顺序被所有进程交付，无论它们之间的因果关系如何。
    - 这种顺序不需要尊重因果性或先进先出（FIFO）规则，但可以被设计为尊重这些规则。
4. **总序广播在复制服务中的应用**
    - 状态机复制（SMR）：在这种场景中，副本需要以相同的顺序处理请求，以保持一致性。总序广播确保了每个副本接收到的请求顺序是一致的，从而使得每个副本的状态保持同步。

属性

1. **请求（Request）**
    - `< tob, Broadcast | m >`：进程`p`广播消息`m`给所有进程。
2. **指示（Indication）**
    - `< tob, Deliver | p, m >`：交付由进程`p`广播的消息`m`。
3. **属性（Properties）**
    - **TOB1. 有效性（Validity）**：如果一个正确的进程`p`广播了消息`m`，那么`p`最终会交付消息`m`。
    - **TOB2. 无重复（No duplication）**：没有消息会被交付超过一次。
    - **TOB3. 无创造（No creation）**：如果一个进程交付了一个发送者为`s`的消息`m`，那么消息`m`之前必须由进程`s`广播过。
    - **TOB4. 一致性（Agreement）**：如果一个消息`m`被某个正确的进程交付了，那么所有正确的进程最终都会交付消息`m`。
    - **TOB5. 总序（Total order）**：假设`m1`和`m2`是任意两条消息，而`p`和`q`是交付`m1`和`m2`的任意两个正确的进程。如果进程`p`先交付`m1`再交付`m2`，那么进程`q`也应该先交付`m1`再交付`m2`。

总序广播规范的关键在于其最后一个属性——总序，这确保了所有正确的进程以相同的顺序交付所有消息。

实现

```
Implements: TotalOrderBroadcast, instance tob.
Uses: ReliableBroadcast, instance rb; Consensus (multiple instances).

# 当总序广播实例初始化时触发的事件
upon event < tob, Init > do
    unordered := ∅;  # 初始化一个未排序的消息集合
    delivered := ∅;  # 初始化一个已交付的消息集合
    round := 1;  # 初始化当前轮次为1
    wait := FALSE;  # 初始化等待标志为FALSE

# 当有广播消息请求时触发的事件
upon event < tob, Broadcast | m > do
    trigger < rb, Broadcast | m >;  # 通过可靠广播机制广播消息m

# 当通过可靠广播交付消息时触发的事件
upon event < rb, Deliver | p, m > do # 收到p广播的消息m
    if m ∉ delivered then
```

```
        unordered := unordered ∪ {(p, m)};  # 如果消息m尚未被交付，则添加到未排序的消息集合中

# 如果有未排序的消息且不在等待状态，则发起共识提案
upon unordered ≠ ∅ ∧ wait = FALSE do
    wait := TRUE;  # 设置等待标志为TRUE，等到共识结果
    Initialize a new instance c.round of consensus;  # 初始化一个新的共识实例c.round
    trigger < c.round, Propose | unordered >;  # 对未排序的消息集合提出共识提案

# 当在共识实例中达成决定时触发的事件
upon event < c.r, Decide | decided > such that r = round do
    for all (s, m) ∈ sort(decided) do
        trigger < tob, Deliver | s, m >;  # 按顺序交付决定的消息
    delivered := delivered ∪ decided;  # 将决定的消息添加到已交付的消息集合中
    unordered := unordered \ decided;  # 从未排序的消息集合中移除这些消息
    round := round + 1;  # 轮次递增
    wait := FALSE;  # 重置等待标志
```

这段代码通过结合可靠广播和共识算法实现了总序广播。其核心思想是收集所有通过可靠广播接收到但尚未排序的消息，然后通过共识算法对这些消息进行排序，最终以相同的顺序交付这些消息。

Consensus algorithm III

假如心跳消息不可靠，我们需要用quorum实现共识，那怎么解决quorum？n=2f+1

使用正确多数（Quorum）和最终完美故障检测器（Eventually Perfect Failure Detector，简称→P）来实现均匀共识的算法。这种算法的核心思想是让进程轮流担任领导者（协调者）的角色，直到其中一个成功地促成了一个决定。这个算法优先考虑安全性（一致性或协议性）而不是活性（终止）。

在讨论分布式系统时，网络故障、时间假设和故障检测器的特性是核心考虑因素。不同的系统环境（异步、同步和部分同步）以及故障检测器（如最终完美故障检测器，→P）的特性对算法的设计和可靠性有着显著影响。以下是这些概念的详细解释：

1. **异步（Asynchrony）**：
    - 在异步环境中，正确的进程无法保证在已知时间范围内进行通信。这意味着消息的传递时间是不确定的，可能会有很大的延迟。
    - 网络拥塞可能导致数据包丢失，从而增加了通信的不确定性。
2. **同步（Synchrony）**：
    - 在同步环境中，存在一个预先给定的时间界限Δ，进程间的通信和操作都在这个时间界限内完成。
    - 这种环境假设网络是可靠的，没有网络故障。
3. **部分同步（Partial Synchrony）**：
    - 部分同步是异步和同步的一种折中。在系统运行的某个未知时间点之后，时间界限Δ被满足。
    - 这意味着最初系统可能表现为异步，但最终会变成同步。
4. **最终完美故障检测器（→P）**：
    - **强完整性（Strong Completeness）**：最终，每个崩溃的进程都会被所有正确的进程永久怀疑。→P最终不会错过任何一个崩溃的进程。

- **最终强准确性（Eventual Strong Accuracy）**：最终，没有任何正确的进程会被任何进程怀疑。但在最初阶段，→P可能会错误地怀疑正确的进程。

5. **错误怀疑的影响**：
   - 在某些共识算法（如算法I和II）中，对正确进程的错误怀疑可能导致算法失效。这是因为这些算法可能依赖于故障检测器的准确性来做出决定。
   - 即使→P最终会变得准确，其在初期的不准确性也可能对某些算法产生负面影响。
6. **钻石问题（Diamonds Problem）**：
   - “钻石”问题指的是在部分同步环境下，正确的进程可能被有限次错误地怀疑。

实现

```
Implements: UniformConsensus, instance uc.
Uses: BestEffortBroadcast, instance beb; EventuallyPerfectFailureDetector, instance
◇P; PerfectPointToPointLinks, instance pl.

# 当均匀共识实例初始化时触发的事件
upon event < uc, Init > do
    suspected := ∅;  # 初始化怀疑进程集合
    nacked := ∅;  # 初始化否认消息集合
    round := 1;  # 初始化当前轮次为1
    proposed := false;  # 初始化提案状态为未提出
    proposal := ⊥;  # 初始化提案为未定义
    estimate := nil;  # 初始化估计值为nil
    estround := 0;  # 初始化估计值的轮次为0
    states := [nil, 0]n;  # 初始化所有进程的状态数组
    acks := 0;  # 初始化确认计数为0

# 当故障检测器怀疑某个进程时触发的事件
upon event < ◇P, Suspect | p> do
    suspected := suspected ∪ {p};  # 将进程p添加到怀疑进程集合中

# 当故障检测器恢复对某个进程的信任时触发的事件
upon event < ◇P, Restore | p> do
    suspected := suspected \ {p};  # 将进程p从怀疑进程集合中移除

# 当有提案请求时触发的事件
upon event < uc, Propose | v > such that proposal = ⊥ do
    proposal := v;  # 设置提案值为v

# 当当前进程的排名等于轮次且尚未提出提案且提案不为空时触发的事件
upon rank(self) = round and proposed = false and proposal ≠ ⊥ do
    proposed := true;  # 标记已提出提案
    states := [nil, 0]n;  # 重置所有进程的状态
    acks := 0;  # 重置确认计数
    trigger < beb, Broadcast | [READ, round] >;  # 广播[READ, round]消息

# 当通过最佳努力广播接收到READ消息且发送者排名等于当前轮次时触发的事件
upon event < beb, Deliver | p, [READ, round] > and rank(p) = round do
```

```
    trigger < pl, Send | p, [GATHER, round, estimate, estround] >;  # 向发送者发送[GATHER, round, estimate, estround]消息

# 当通过点对点链接接收到GATHER消息时触发的事件
upon event < pl, Deliver | p, [GATHER, round, est, estrnd] > do
    states[p] := [est, estrnd];  # 更新发送者的状态信息

# 当状态数组中的状态数达到多数时触发的事件，领导者从收到的回复中选择最新的估计值作为其提案值。如果没有收到任何估计值，领导者将使用自己的初始提案值。
upon #states ≥ majority do
    if 存在非nil且非0的状态 then
        选择具有最高轮次的状态为当前提案
    states := [nil, 0]n;  # 重置状态数组
    trigger < beb, Broadcast | [IMPOSE, round, proposal] >;  # 广播[IMPOSE, round, proposal]消息

# 当通过最佳努力广播接收到IMPOSE消息且发送者排名等于当前轮次时触发的事件
upon event < beb, Deliver | p, [IMPOSE, round, v] > and rank(p) = round do
    estimate := v;  # 更新估计值为v
    estround := round;  # 更新估计值的轮次为当前轮次
    trigger < pl, Send | p, [ACK, round] >;  # 向发送者发送[ACK, round]消息

# 当通过点对点链接接收到ACK消息时触发的事件
upon event < pl, Deliver | p, [ACK, round] > do
    acks := acks + 1;  # 确认计数递增
    if acks ≥ majority then
        trigger < beb, Broadcast | [DECIDE, proposal] >;  # 广播[DECIDE, proposal]消息

# 当通过最佳努力广播接收到DECIDE消息时触发的事件
upon event < beb, Deliver | p, [DECIDE, v] > do
    trigger < uc, Decide | v >;  # 触发决定事件，决定值为v

# 当当前进程的排名等于轮次且进程被怀疑时触发的事件
upon rank(p) = round and p ∈ suspected do
    trigger < beb, Broadcast | [NACK, round] >;  # 广播[NACK, round]消息
    proposed := false;  # 标记未提出提案
    round := round + 1;  # 轮次递增

# 当通过最佳努力广播接收到NACK消息时触发的事件
upon event < beb, Deliver | p, [NACK, rnd] > do
    nacked := nacked ∪ {rnd};  # 将否认的轮次添加到否认集合中

# 当当前轮次在否认集合中时触发的事件
upon round ∈ nacked do
    proposed := false;  # 标记未提出提案
    round := round + 1;  # 轮次递增
```

正确性

1. **终止（Termination）**
   - **正确性假设**：假设有一个正确的进程做出了决定，它将会将这个决定广播给所有进程。这意味着每个正确的进程最终都会做出决定。
   - **反证法**：假设存在一个正确的进程，但没有任何正确的进程做出决定。我们来论证这是不可能的：
     - 根据正确多数假设和最终完美故障检测器（→P）的强完整性，没有一个正确的进程会永远停留在某个轮次中。
     - 根据→P的最终准确性，最终会有一个正确的进程`p`达到一个它是领导者且不被怀疑的轮次。在这个轮次中，`p`会做出决定，这与我们的假设相矛盾。
     - 通过正确的多数假设和的强完备性◇p：没有一个正确的过程会在某一轮中永远被阻止；通过的最终精度特性◇p：一些正确的过程p到达一个回合，在这个回合中它是领导者，而不是可疑的。p在那一轮中做出决定：矛盾（强翻译）
2. **一致性（Agreement）**
   - **第一个决定的轮次**：设`k`是某个进程`pi`第一次做出某个值`v`的决定的轮次，即`pi`是第`k`轮的领导者，并在`k`轮做出了值`v`的决定。
   - **多数采纳**：这意味着在第`k`轮中，大多数进程采纳了值`v`。
   - **算法保证**：算法保证在高于`k`轮的任何轮次中，没有其他的值除了`v`会被强加（因此也不会被决定），这是因为法定人数（Quorum）的存在。
   - 一致性的保证并不依赖于故障检测器，而是依赖于多数假设（Majority Assumption）或法定人数（Quorum）。
     1. **一致性与故障检测器**：
        - 即使在故障检测器可能提供错误信息的情况下，如始终怀疑所有人或从不怀疑任何人，一致性仍然得到保证。
        - 这是因为一致性取决于每轮中能够达到多数共识的能力，而非故障检测器的准确性。
     2. **一致性的正确性论证**：
        - 即使在故障检测器不可靠的情况下，之前的正确性论证仍然成立。一致性依赖于多数进程（或法定人数）能够在某一轮次中就一个值达成共识。
        - 一旦一个值在某个轮次被大多数进程采纳，就不会有其他的值被强加或决定。
     3. **终止与故障检测器**：
        - 然而，终止（Termination）的保证确实依赖于故障检测器。如果故障检测器行为不当（例如，永远怀疑每个人或从不怀疑任何人），则终止可能无法保证。
        - 在不完美的故障检测器下，可能出现所有进程都被无限期怀疑的情况，导致无法进入一个稳定的状态来做出最终的决定。
     4. **总结**：
        - 一致性的保证是基于多数进程能够达成共识的假设，而不依赖于故障检测器的行为。
        - 终止的保证则受故障检测器的影响较大，尤其是在故障检测器可能提供不准确信息的情况下。

总结来说，这个算法的正确性基于以下几点：

- **终止**：利用正确多数假设和故障检测器的特性来保证每个正确的进程最终都会做出决定。
- **一致性**：通过法定人数机制保证一旦某个值在某个轮次被决定，之后的所有轮次都不会有其他的值被强加或决定。

Paxos/Synod algorithm

Paxos算法（有时也被称为Synod算法），特别适用于环境中可能存在不可靠的网络和进程故障的情况。Paxos算法的核心思想是尽可能地将活性（liveness，即终止性）和安全性（safety，即一致性）分离开来。

1. **分离活性和安全性**：
   - Paxos算法尝试最大限度地分离决策的活性（即算法最终能够达成决定）和安全性（即所有进程对同一值达成共识）。
2. **算法的两个主要组件**：
   - **Ω - 最终领导者故障检测器（Eventual Leader Failure Detector）**：这个组件负责最终选出一个领导者，这个领导者被所有正确的进程所信任。Ω保证了活性，即使在一些进程可能失败或网络延迟的情况下。
   - **Obstruction-free Consensus（无阻塞共识）**：这个组件负责在没有对手（即没有其他进程干扰）的情况下达成共识。它保证了在一个进程可以无障碍地执行时，该进程能够做出决定。
3. **Paxos的运作原理**：
   - 在Paxos算法中，任何进程都可以提出一个值作为决定的候选。
   - 这个过程分为几个阶段，主要包括提案阶段和接受阶段。在提案阶段，领导者发送提案请求，其他进程则回应是否接受这个提案。
   - 如果一个提案被大多数进程接受，那么该提案就会进入到接受阶段，并最终被所有进程接受。

组件

最终领导者检测器（Ω）

1. **指示（Indication）**：
   - `< Ω, Trust | p >`：表示进程 `p` 被信任为领导者。
2. **属性（Properties）**：
   - **ELD1. 最终准确性（Eventual Accuracy）**：存在某个时刻之后，每个正确的进程都信任某个正确的进程。
   - **ELD2. 最终一致性（Eventual Agreement）**：存在某个时刻之后，没有两个正确的进程信任不同的进程。

无阻塞共识（ofcons）

1. **请求（Request）**：
   - `< ofcons, Propose | v >`：提出一个值 `v` 作为共识提案。
2. **指示（Indications）**：

- `< ofcons, Decide | v >`：决定了值`v`。
- `< ofcons, Abort >`：中止共识过程。

3. **属性（Properties）**：
   - **有效性（Validity）**：任何被决定的值都是某个进程提出的值。
   - **一致性（Agreement）**：没有两个正确的进程做出不同的决定。
   - **无阻塞终止（Obstruction-Free Termination）**：
     - 如果一个正确的进程`p`提出了一个提案，那么每个正确的进程最终都会做出决定或中止。
     - 此外，如果一个正确的进程无限次提出同一个值，那么它不会永远中止。
   - **完整性（Integrity）**：每个进程最多做出一次决定。

- **最终领导者检测器**（Ω）是为了确保在最终，所有正确的进程都能一致地信任某个正确的领导者。这对于实现像Paxos这样的共识算法非常重要，因为领导者需要协调共识决策过程。
- **无阻塞共识**（ofcons）则是一种共识算法，它允许在一定条件下中止共识过程。这种算法保证了即使在某些进程可能阻塞的情况下，也能够达成共识或允许进程中止尝试。这对于处理动态变化的网络环境和进程状态非常有用。

synod

实现

```
Implements: UniformConsensus, instance uc.
Uses: BestEffortBroadcast, instance beb; PerfectPointToPointLinks, instance pl;
Eventual leader detector Ω.

# 当均匀共识实例初始化时触发的事件
upon event < uc, Init > do
    ballot := rank(self) - n;  # 初始化提案轮次
    proposal := ⊥;  # 初始化提案值为未定义
    readballot := 0;  # 初始化已读轮次为0
    imposeballot := 0;  # 初始化已强加轮次为0
    estimate := ⊥;  # 初始化估计值为未定义
    decided := false;  # 初始化决定状态为false
    states := [nil,0]n;  # 初始化所有进程的状态数组
    acks[*] := 0;  # 初始化确认计数数组

# 当有提案请求时触发的事件
upon event < uc, Propose | v > do
    repeat periodically # 定期重复
        if self = leader then  # 如果自己是领导者
            proposal := v;  # 设置提案值为v
            ballot := ballot + n;  # 提案轮次增加
            states := [nil,0]n;  # 重置所有进程的状态
            trigger < beb, Broadcast | [READ, ballot] >;  # 广播[READ, ballot]消息
    until decided  # 直到做出决定

# 当信任某个领导者时触发的事件
upon event < Ω, Trust | le > do
```

```
    leader := le;  # 设置信任的领导者

# 当通过最佳努力广播接收到READ消息时触发的事件
upon event < beb, Deliver | p, [READ, ballot'] > do
    if readballot < ballot' and imposeballot < ballot' then
        readballot := ballot';  # 更新已读轮次
        trigger < pl, Send | p, [GATHER, ballot', imposeballot, estimate] >;  # 向发送者发送[GATHER, ballot', imposeballot, estimate]消息

# 当通过点对点链接接收到GATHER消息时触发的事件
upon event < pl, Deliver | p, [GATHER, ballot, estballot, est] > do
    states[p] := [est, estballot];  # 更新发送者的状态信息

# 当状态数组中的状态数达到多数时触发的事件
upon #states ≥ majority do
    if 存在非nil且非0的状态 then
        选择具有最高轮次的状态为当前提案
    states := [nil,0]n;  # 重置状态数组
    trigger < beb, Broadcast | [IMPOSE, ballot, proposal] >;  # 广播[IMPOSE, ballot, proposal]消息

# 当通过最佳努力广播接收到IMPOSE消息时触发的事件
upon event < beb, Deliver | p, [IMPOSE, ballot', v] > do
    if readballot ≤ ballot' and imposeballot < ballot' then
        estimate := v;  # 更新估计值为v
        imposeballot := ballot';  # 更新已强加轮次
        trigger < beb, Broadcast | [ACK, ballot', v] >;  # 广播[ACK, ballot', v]消息

# 当通过最佳努力广播接收到ACK消息时触发的事件
upon event < beb, Deliver | p, [ACK, ballot', v] > do
    acks[ballot'] := acks[ballot'] + 1;  # 确认计数递增
    if acks[ballot'] ≥ majority and not decided then
        trigger < uc, Decide | v >;  # 触发决定事件，决定值为v
        decided := true  # 标记已做出决定
        trigger < beb, Broadcast | [DECIDE, v] >;  # 广播[DECIDE, v]消息
```

这个实现的主要流程是：

1. 每个进程根据自身的排名和轮次提出提案。
2. 领导者（由最终领导者检测器Ω确定）尝试将其提案强加给其他进程。
3. 如果领导者的提案被多数进程接受，那么该提案将被决定，并通过最佳努力广播通知给所有进程。
4. 进程在收到决定后，将决定的值广播给所有其他进程。

表现和优化

您提到的是关于Paxos算法的性能考虑和优化，以及如何将Synod算法扩展为Paxos算法来实现状态机复制（State Machine Replication, SMR）或总序广播（Total-Order Broadcast）。让我们逐一解析这些概念：

性能：**无故障时间复杂度（Fault-Free Time Complexity）**：在没有故障的情况下，Paxos（或Synod）算法的时间复杂度是4。这意味着在最理想的情况下，从提案到决定需要4个通信步骤。

优化：省略第一个READ阶段：

- 允许一个单一的进程（假定的领导者，例如`p1`）在其第一个轮次中跳过READ阶段。
- 这可以将无故障情况下的时间复杂度降低到2。
- 类似的优化也可以应用于轮换领导者算法（允许`p1`在第一次尝试时直接进行IMPOSE而无需进行READ）。

从Synod到Paxos

1. **Paxos的目的**：
   - Paxos最初被设计为一个状态机复制协议，即用于实现总序广播。总序广播是指确保所有进程按照相同的顺序处理请求。
2. **Synod与Paxos的区别**：
   - Synod是一次性共识算法，只能就一个特定的提案达成共识。Paxos可以用于连续不断的请求处理，其中每个请求可能对应一个单独的共识实例。
3. **Paxos的工作流程**：
   - 客户端（Clients）发起请求。服务器（Servers，即进程）运行共识过程，可以有多个共识实例同时运行。
   - 例如，Synod实例25用于就第25个请求达成共识。

Paxos算法通过引入领导者的概念和对共识过程的优化，提供了一种高效的方法来在分布式系统中处理连续的请求。它的设计旨在解决状态机复制中的关键问题，即确保所有进程以相同的顺序处理请求，从而维护整个系统的一致性。通过优化，如省略某些阶段或允许领导者更快地做出决定，可以进一步提高Paxos算法的效率。

具体方法

在Paxos算法中，READ阶段的确是用来使领导者了解之前发生的事件，而不涉及更新或提出新的共识提案。基于这个观察，可以对算法进行优化，特别是在处理多个共识实例（多个Synod实例）时。

READ阶段的作用

1. **领导者赶上之前的进展**：READ阶段允许新的或现有的领导者了解系统的当前状态，特别是其他进程可能已经接受的提案。
2. **领导者通常保持不变**：在许多情况下，领导者会保持不变，尤其是在没有新请求或变化的时候。

优化策略

1. **仅在领导者变更时运行READ阶段**：
   - 当领导者发生变更时，执行READ阶段，以便新领导者能够捕获之前的所有变化和决定。可以同时对多个Synod实例执行READ阶段。
2. **使用相同的提案编号（ballot number）**：
   - 对于未来的所有Synod实例，使用从最后一次READ阶段得到的相同提案编号。未来的实例只需要运行IMPOSE阶段。
3. **每条消息包含提案编号和序列号**：
   - 例如，当尝试确定第11个操作应该是什么时，序列号（SeqNum）可能是11。

4. **增加提案编号同时进行READ**：
    - 当领导者变更时，进程会增加提案编号，并执行READ阶段。

处理空洞或间隙（Noop）

1. **在READ阶段检测到的空洞/间隙**：
    - 如果领导者在READ/GATHER阶段发现了某些请求的缺失（例如，有1-12、14和17号请求，但没有13、15和16号请求）。
2. **特殊提案：Noop值**：
    - 领导者可以针对缺失的实例（例如，13、15和16号请求）执行IMPOSE阶段，并提出一个特殊的提案，即Noop值（表示“不做任何操作”）。

通过这些优化，Paxos算法可以更有效地处理多个共识实例，并且能够在领导者变更或其他关键事件发生时迅速适应。这种优化方法在实际应用中特别重要，因为它减少了不必要的通信，提高了算法的整体效率和响应速度。