Quorum

法定人数是确保一致性（安全性）的关键抽象，任何仲裁都可以在其他进程失败的情况下取得进展。

为了在分布式系统中达成一致或执行某些操作所需要的最小数量的参与者（通常是进程或节点）。这个概念源自政治和法律领域，其中"Quorum"指的是进行有效投票或会议的最小出席人数。

属性

1. **安全性（Safety）**
   - **多数法定人数**：为了保证安全性，一个Quorum通常被定义为系统中大多数进程的集合。这意味着任何两个Quorums都至少有一个共同的进程。这个属性防止了所谓的“分脑”问题（Split-Brain Problem）。
   - **分脑问题**：在分布式系统中，如果系统被分成两个或多个彼此不通信的部分，且每个部分都认为自己是完整的系统并独立操作，就会发生分脑问题。这可能导致数据不一致和其他错误。通过要求多数法定人数，可以确保系统不会在无法通信的子集中独立作出决策。
2. **活性（Liveness）**
   - **即使少数进程失败**：即便系统中有少数进程失败，仍然总会存在至少一个只包含正确进程的Quorum。这意味着系统仍然可以继续进行操作和决策，保证了活性。
   - **保证活性**：这个属性确保了即使在面对部分故障的情况下，系统仍然能够正常工作，可以执行必要的操作，如数据读写、状态更新等。

安全性和活性的结合

1. **活性（Liveness）**
   - **N−f个进程可以取得进展**：这意味着即使有f个进程发生故障，剩余的N−f个进程应该足够形成一个Quorum，从而使系统能够继续进行必要的操作，如数据读写或状态更新。
   - **Quorum大小**：这里的关键是确保Quorum的大小设置为不超过N−f，这样即使f个进程崩溃，系统中仍然有足够的进程来形成一个有效的Quorum。
2. **安全性（Safety）**
   - **任意两个Quorum至少有一个正确的进程相交**：为了确保安全性，任意两个Quorum都应至少有一个正确的进程共同参与。这可以防止“分脑”问题，保证系统不会做出矛盾的决策。
   - **推导公式**：`2(N−f) > N`，即两个Quorum（每个至少N−f个进程）的总大小超过了系统总大小N，确保了它们至少有一个共同的进程。
   - **进程数限制**：由此可推导出`N > 2f`，即系统中的总进程数N必须大于两倍的可能故障数f。为了满足整数的要求，我们进一步得到`N ≥ 2f+1`。

综上所述，为了同时满足安全性和活性，系统的总进程数N需要至少是可能故障数f的两倍加一。

拜占庭故障情况下Quorum

拜占庭故障是指系统中的一部分节点可能出现任意类型的错误行为，包括发送错误或矛盾的信息。

1. **对于任意两个Quorum的交集**

- 在拜占庭Quorum系统中，任何两个Quorum `Q_1` 和 `Q_2` 的交集至少要包含`f+1`个进程，其中`f`是可能的拜占庭故障的数量。这是为了确保至少有一个正确的（非拜占庭）进程出现在任何两个Quorum的交集中。

2. **活性（Liveness）**
   - 即使存在`f`个拜占庭故障，`N-f`个正确的进程仍然可以取得进展，形成一个有效的Quorum。
3. **安全性（Safety）**
   - 安全性要求在考虑拜占庭故障的情况下，任意两个Quorum都至少有一个正确的进程相交。
   - 为了确保安全性，必须满足`2(N-f) > N+f`，即两个没有拜占庭故障的Quorum的大小之和要大于总进程数加上可能的拜占庭故障数。
   - 这可以推导出`N > 3f`，即系统中的总进程数N必须大于拜占庭故障数f的三倍。为了满足整数的要求，我们进一步得到`N ≥ 3f+1`。

Majority-Ack Uniform Reliable Broadcast

1. **在没有完美故障检测器的情况下实现URB**：
   - **使用Quorum系统**：在一个假设大多数进程是正确的系统中（即`N > 2f`，其中`N`是进程总数，`f`是可能的故障进程数），可以使用基于Quorum的方法来实现URB。这种方法不依赖于完美的故障检测，而是通过确保任何两个Quorums至少有一个共同的正确进程来维护一致性。
   - **Quorum的作用**：在这种设置中，Quorum充当一种达成共识的机制，确保消息被大多数正确的进程接收和确认。由于任何两个Quorums都至少有一个共同的正确进程，这保证了消息的一致性和完整性。

实现

```
Implements: UniformReliableBroadcast, instance urb.
Uses: BestEffortBroadcast, instance beb.

# 当统一可靠广播系统初始化时触发的事件
upon event < urb, Init > do
    delivered := ∅;  # 初始化delivered集合，用于跟踪已交付的消息
    pending := ∅;  # 初始化pending集合，用于存储待处理的消息
    for all m do ack[m] := ∅;  # 为每个消息m初始化一个空集合，用于跟踪确认接收消息的进程

# 当有广播消息请求时触发的事件
upon event < urb, Broadcast | m > do
    pending := pending ∪ {(self, m)};  # 将消息添加到pending集合
    trigger < beb, Broadcast | [DATA, self, m] >;  # 通过尽力而为广播发送消息

# 当尽力而为广播传递消息时触发的事件
upon event < beb, Deliver | p, [DATA, s, m] > do
    ack[m] := ack[m] ∪ {p};  # 记录确认接收消息m的进程p
    if (s, m) not in pending then
        pending := pending ∪ {(s, m)};  # 如果消息不在pending集合中，则添加
        trigger < beb, Broadcast | [DATA, s, m] >;  # 重新广播消息

# 判断消息m是否可以交付的函数
```

```
function candeliver(m) returns Boolean is
    return #(ack[m]) > N/2;  # 如果确认接收消息m的进程数超过一半，则返回true

# 当存在可交付的消息时触发的事件
upon exists (s, m) ∈ pending such that candeliver(m) ∧ m not in delivered do
    delivered := delivered ∪ {m};  # 将消息标记为已交付
    trigger < urb, Deliver | s, m >;  # 交付消息m
```

这个算法的核心思想是：
当一个进程广播一条消息时，它将消息添加到待处理集合`pending`中，并通过尽力而为广播发送这条消息。
当一个进程接收到一条消息时，它会记录已确认接收这条消息的进程。如果这条消息尚未加入到待处理集合`pending`中，则加入该消息并重新广播。
一个消息只有在超过系统中一半以上的进程都确认接收后，才被认为是可以交付的。这通过`candeliver`函数来判断。
当一个消息可以被交付时，它会被从`pending`集合移动到`delivered`集合，并触发交付事件。

正确性

正确性涵盖了有效性、无重复、无创造和统一协议的几个关键方面。

1. **有效性（Validity）**
   - 如果发送者是正确的（即没有发生故障），那么每个正确的进程都会交付这个DATA消息。
   - 这里的有效性包括几个部分：
     - 每个正确的进程都会交付DATA消息。
     - 每个正确的进程都会广播DATA消息。
     - 每个正确的进程都会交付超过`N/2`个DATA消息，这是因为大多数进程是正确的（根据`N > 2f`，即进程总数大于故障进程数的两倍加一）。
2. **无重复、无创造（No Duplication, No Creation）**
   - 这部分相对直接。系统确保每个消息只被交付一次（无重复），且每个交付的消息都是由某个进程广播的（无创造）。
3. **统一协议（Uniform Agreement）**
   - 假设`N = 2f + 1`，其中`f`是可能的故障进程数。
   - 假设一个正确的进程`p1`以某种方式交付了消息`m`。
   - 为了交付`m`，`p1`必须至少接收到`f + 1`个关于`m`的DATA消息，这至少包括一个来自正确进程的DATA消息。
   - 这个正确进程的DATA消息也会被剩余的`f`个正确进程交付，然后它们会广播这个DATA消息。
   - 因此，每个正确的进程最终都会交付消息`m`，因为至少有`f + 1`个正确的进程已经发送了一个DATA消息。

表现

1. **最佳情况**
   - **通信步骤**：两个通信步骤。**消息数量**：$O(N^2)$条消息，其中N是系统中进程的总数。

- **解释**：在最佳情况下，每个进程都能迅速接收到其他所有进程的消息，并且每个进程都只需要发送和接收一轮消息。因此，总共需要两轮通信步骤：一轮用于发送消息，另一轮用于接收消息。每轮中，每个进程都会发送和接收大约N条消息，因此总消息数量是$O(N^2)$。

2. **最坏情况**
   - **通信步骤**：N/2+2步。**消息数量**：$O(N^2)$条消息。
   - **第1步**：原始的广播。每个进程发送它的消息。
   - **第2步至N/2+1步**：处理确认和可能的重传。在这些步骤中，进程们交换消息确认，以及可能的重传请求和重传消息。在最坏情况下，我们假设这个过程需要N/2步，因为可能有进程未能在前几轮通信中接收到某些消息。
   - **第N/2+2步**：最后的确认。在这一步中，确保所有正确的进程都已收到并确认了所有消息。
   - **解释**：最坏情况下，可能需要等待一定时间才能收到足够多的确认，以确保每个消息被大多数进程接收。因此，总共需要N/2+2轮通信步骤。在每一步中，系统的消息交换量仍然大约是$O(N^2)$。

Byzantine-tolerant broadcast

在处理拜占庭容错问题时，广播不仅仅是传递信息，更是一个复杂的协议，其中每次广播都是独立的，且专注于在单个消息上达成一致。

Consistent broadcast primitives

属性

“ByzantineConsistentBroadcast”（拜占庭一致广播，简称bcb）是一个专门设计来应对拜占庭故障的广播协议。这个协议在分布式系统中用于确保即使在存在恶意或故障节点的情况下，消息的广播也能保持一致和可靠。以下是对bcb的属性的解释：

1. **请求（Request）**：`< bcb, Broadcast | m >`
   - 这个请求指示实例bcb将消息`m`广播给所有进程。这个操作只由特定的进程`p`执行。
2. **指示（Indication）**：`< bcb, Deliver | p, m >`
   - 这个指示表示消息`m`由进程`p`广播，并且已经被接收。
3. **属性（Properties）**
   - **BCB1. 有效性（Validity）**：如果一个正确的进程`p`广播了消息`m`，那么每个正确的进程最终都会交付`m`。
   - **BCB2. 无重复（No Duplication）**：每个正确的进程至多交付一条消息。
   - **BCB3. 完整性（Integrity）**：如果某个正确的进程交付了由正确的进程`p`发送的消息`m`，那么`m`之前必定是由`p`广播的。
   - **BCB4. 一致性（Consistency）**：如果某个正确的进程交付了消息`m`，而另一个正确的进程交付了消息`m’`，那么`m = m’`。这保证了所有正确的进程交付的是相同的消息。

假设

1. **假设**
   - **异步系统**：在这样的系统中，消息的传递时间是不确定的，也没有全局时钟或其他同步机制。
   - **故障进程数量**：系统中故障（可能为拜占庭故障）的进程数量`f`小于总进程数量`N`的三分之一。
   - **无故障检测器（FD）**：系统中没有故障检测器（尤其是拜占庭故障检测器），这是因为拜占庭故障检测器在实现上可能非常复杂。
2. **为什么`f < N/3`？**
   - 这个条件是拜占庭一致性广播在异步系统中能够成功实现的一个重要前提。在这样的系统中，如果故障进程的数量超过了总进程数量的三分之一，就无法保证系统的所有正确进程都能达成一致性。
3. **为什么在`N = 3f`时BCB无法实现?**
   - 因为在拜占庭环境下N>=3f+1

总的来说，这段话解释了为什么在拜占庭容错模型下，异步系统在没有故障检测器且故障进程数量达到一定比例时，无法实现拜占庭一致性广播。这是因为在这样的条件下，系统无法有效地区分正确进程和拜占庭故障进程，从而无法保证消息的一致性和可靠性。

Authenticated Echo Broadcast（通过身份验证的回声广播）

实现

```
implements: ByzantineConsistentBroadcast, instance bcb, with sender s.
Uses: AuthPerfectPointToPointLinks, instance al.
# 当拜占庭一致性广播系统初始化时触发的事件
upon event < bcb, Init > do
    sentecho := FALSE;  # 初始化sentecho标志，表示是否已发送过ECHO消息
    delivered := FALSE;  # 初始化delivered标志，表示是否已交付过消息
    echos := [⊥]N;  # 初始化echos数组，用于跟踪收到的ECHO消息

# 当有广播消息请求时触发的事件
upon event < bcb, Broadcast | m > do
    for all q ∈ Π do
        trigger < al, Send | q, [SEND, m] >;  # 向所有进程发送[SEND, m]消息

# 当从发送者s接收到[SEND, m]消息且尚未发送过ECHO时触发的事件
upon event < al, Deliver | p, [SEND, m] > such that p = s and sentecho = FALSE do
    sentecho := TRUE;  # 标记已发送ECHO消息
    for all q ∈ Π do
        trigger < al, Send | q, [ECHO, m] >;  # 向所有进程发送[ECHO, m]消息

# 当收到[ECHO, m]消息时触发的事件
upon event < al, Deliver | p, [ECHO, m] > do
    if echos[p] = ⊥ then
        echos[p] := m;  # 记录从进程p收到的ECHO消息

# 当存在消息m被超过(N+f)/2个进程确认且尚未交付时触发的事件m交付,必须满足`2(N-f) > N+f`这是安全性的要求
```

```
upon exists m ≠ ⊥ such that #{p ∈ Π | echos[p] = m} > (N+f)/2 and delivered = FALSE
do
    delivered := TRUE;  # 标记消息为已交付
    trigger < bcb, Deliver | s, m >;  # 交付消息m
```

这个算法的核心思想是：

- 当有一个广播请求时，发送者会向所有进程发送一个带有消息`m`的`SEND`消息。
- 当一个进程第一次从发送者接收到`SEND`消息时，它会向所有进程发送`ECHO`消息，表示它已经“听到”了这条消息。
- 每个进程跟踪它从其他进程收到的`ECHO`消息。一旦一个进程发现超过`(N+f)/2`个进程发送了对同一消息的`ECHO`确认，它就会交付这个消息。

正确性和表现

1. **有效性（Validity）**
   - 如果发送者是正确的，则每个正确的进程都会收到一个`SEND`消息。每个正确的进程在收到`SEND`消息后都会发送一个`ECHO`消息。每个正确的进程都会至少收到`N - f`个`ECHO`消息，其中`N`是总进程数，`f`是可能的拜占庭故障进程数。由于`N - f`大于`(N + f)/2`，因此每个正确的进程也会交付在`ECHO`消息中包含的消息`m`。
2. **无重复、完整性（No Duplication, Integrity）**
   - 这部分比较直接。算法确保了每个消息只被交付一次（无重复），且每个交付的消息都是由某个进程广播的（完整性）。
3. **一致性（Consistency）**
   - 为了让一个正确的进程`p`交付某个消息`m`，它需要收到超过`(N + f)/2`个包含消息`m`的`ECHO`消息。任何两个这样的`ECHO`消息集合至少在`f+1`个进程中有交集。至少有一个这样的进程是正确的，并且不会对两个不同的消息发送`ECHO`。因此，可以保证所有正确的进程最终交付的是相同的消息。
4. **性能分析**
   - **时间复杂度**：2个消息延迟。这指的是从原始的`SEND`消息广播到最终的消息交付所需的时间。
   - **消息复杂度**：$O(N^2)$。每个进程都需要向其他所有进程发送消息，因此消息的总数是进程数的平方级别。

Signed Echo Broadcast（签名回声广播）

签名有帮助吗，可以绕过f（f < N/3）的下界？答:no,场景是通用的，没有讨论任何的认证机制。但引入签名后会有复杂度的优化。

（不需要所有人给所有人通信了，由原始广播者来统一）

实现

```
Implements: ByzantineConsistentBroadcast, instance bcb, with sender s.
Uses: AuthPerfectPointToPointLinks, instance al.

# 当拜占庭一致性广播系统初始化时触发的事件
```

```
upon event < bcb, Init > do
    sentecho := FALSE;  # 初始化sentecho标志，表示是否已发送过ECHO消息
    sentfinal := FALSE;  # 初始化sentfinal标志，表示是否已发送过FINAL消息
    delivered := FALSE;  # 初始化delivered标志，表示是否已交付过消息
    echos := [⊥]N;  # 初始化echos数组，用于跟踪收到的ECHO消息
    Σ := [⊥]N;  # 初始化Σ数组，用于存储收到的签名

# 当有广播消息请求时触发的事件，仅由发送者s执行
upon event < bcb, Broadcast | m > do
    for all q ∈ Π do
        trigger < al, Send | q, [SEND, m] >;  # 向所有进程发送[SEND, m]消息

# 当从发送者s接收到[SEND, m]消息且尚未发送过ECHO时触发的事件
upon event < al, Deliver | p, [SEND, m] > such that p = s and sentecho = FALSE do
    sentecho := TRUE;
    σ := sign(self, [ECHO, m]);  # 用自己的私钥对[ECHO, m]消息进行签名
    trigger < al, Send | s, [ECHO, m, σ] >;  # 向s进程发送签名的[ECHO, m, σ]消息

# 当收到[ECHO, m, σ]消息时触发的事件，仅由发送者s执行
upon event < al, Deliver | p, [ECHO, m, σ] > do
    if echos[p] = ⊥ ∧ verifysig(p, [ECHO, m, σ]) then
        echos[p] := m;  # 记录从进程p收到的ECHO消息
        Σ[p] := σ;  # 存储对应的签名

# 当存在消息m被超过(N+f)/2个进程确认且尚未发送FINAL时触发的事件
upon exists m ≠ ⊥ such that #{p ∈ Π | echos[p] = m} > (N+f)/2 and sentfinal = FALSE
do
    sentfinal := TRUE;
    for all q ∈ Π do
        trigger < al, Send | q, [FINAL, m, Σ] >;  # 向所有进程发送[FINAL, m, Σ]消息

# 当收到[FINAL, m, Σ]消息时触发的事件
upon event < al, Deliver | p, [FINAL, m, Σ] > do
    if #{q ∈ Π | Σ[q] ≠ ⊥ ∧ verifysig(q, [ECHO, m, Σ[q]])} > (N+f)/2 and delivered
= FALSE do
        delivered := TRUE;
        trigger < bcb, Deliver | s, m >;  # 交付消息m
```

这个算法的核心思想是：

- 使用数字签名来增加额外的安全性和验证。当一个进程接收到`SEND`消息时，它会发送一个带有签名的`ECHO`消息。
- 其他进程在收到带有签名的`ECHO`消息后，会验证签名，并在验证成功后记录该消息。
- 一旦超过半数的进程确认了同一条消息，就会发送一个`FINAL`消息，其中包括了所有确认该消息的进程的签名。
- 最后，当一个进程收到`FINAL`消息，并且验证了超过半数进程的签名后，它将交付该消息。

表现

Time complexity：3 message

delaysMessage：complexityO(N)

Reliable broadcast primitives

属性

“ByzantineReliableBroadcast”（拜占庭可靠广播，简称brb）

1. **请求（Request）**：`< brb, Broadcast | m >`
   - 这个请求指示实例brb将消息`m`广播给所有进程。这个操作只由特定的进程`s`执行。
2. **指示（Indication）**：`< brb, Deliver | p, m >`
   - 这个指示表示消息`m`由进程`p`广播，并且已经被接收。
3. **属性（Properties）**
   - **BRB1–BRB4**：与拜占庭一致性广播（Byzantine Consistent Broadcast，简称BCB）的属性BCB1–BCB4相同。
   - **BRB5. 全体性（Totality）**：如果任何一个正确的进程交付了某条消息，那么每个正确的进程最终都会交付这条消息。

实现 Authenticated Double-Echo Broadcast（认证双回声广播）

```
Implements: ByzantineReliableBroadcast, instance brb, with sender s.
Uses: AuthPerfectPointToPointLinks, instance al.

# 当拜占庭可靠广播系统初始化时触发的事件
upon event < brb, Init > do
    sentecho := FALSE;  # 初始化sentecho标志，表示是否已发送过ECHO消息
    sentready := FALSE;  # 初始化sentready标志，表示是否已发送过READY消息
    delivered := FALSE;  # 初始化delivered标志，表示是否已交付过消息
    echos := [⊥]N;  # 初始化echos数组，用于跟踪收到的ECHO消息
    readys := [⊥]N;  # 初始化readys数组，用于跟踪收到的READY消息

# 当有广播消息请求时触发的事件
upon event < brb, Broadcast | m > do
    for all q ∈ Π do
        trigger < al, Send | q, [SEND, m] >;  # 向所有进程发送[SEND, m]消息

# 当从发送者s接收到[SEND, m]消息且尚未发送过ECHO时触发的事件
upon event < al, Deliver | p, [SEND, m] such that p = s and sentecho = FALSE do
    sentecho := TRUE;
    for all q ∈ Π do
        trigger < al, Send | q, [ECHO, m] >;  # 向所有进程发送[ECHO, m]消息

# 当收到[ECHO, m]消息时触发的事件
upon event < al, Deliver | p, [ECHO, m] > do
    if echos[p] = ⊥ then
```

```
        echos[p] := m;  # 记录从进程p收到的ECHO消息

# 当超过(N+f)/2个进程发送了相同的ECHO消息且尚未发送READY时触发的事件
upon exists m ≠ ⊥ such that #{p ∈ Π | echos[p] = m} > (N+f)/2 and sentready = FALSE
do
    sentready := TRUE;
    for all q ∈ Π do
        trigger < al, Send | q, [READY, m] >;  # 向所有进程发送[READY, m]消息

# 当收到[READY, m]消息时触发的事件
upon event < al, Deliver | p, [READY, m] > do
    if readys[p] = ⊥ then
        readys[p] := m;  # 记录从进程p收到的READY消息

# 当超过f个进程发送了相同的READY消息且尚未发送READY时触发的事件
upon exists m ≠ ⊥ such that #{p ∈ Π | readys[p] = m} > f and sentready = FALSE do
    sentready := TRUE;
    for all q ∈ Π do
        trigger < al, Send | q, [READY, m] >;  # 再次向所有进程发送[READY, m]消息

# 当超过2f个进程发送了相同的READY消息且尚未交付消息时触发的事件
upon exists m ≠ ⊥ such that #{p ∈ Π | readys[p] = m} > 2f and delivered = FALSE do
    delivered := TRUE;
    trigger < brb, Deliver | s, m >;  # 交付消息m
```

这个算法的核心思想是：

- 在广播请求时，发送者向所有进程发送 `SEND` 消息。
- 当第一次从发送者接收到 `SEND` 消息时，进程会向所有进程发送 `ECHO` 消息。
- 当一个进程收到超过 `(N+f)/2` 个相同的 `ECHO` 消息时，它会向所有进程发送 `READY` 消息。
- 当一个进程收到超过 `f` 个相同的 `READY` 消息时，如果它尚未发送 `READY` 消息，它会再次发送 `READY` 消息。
- 最后，当一个进程收到超过 `2f` 个相同的 `READY` 消息时，它将交付这个消息。

这种方法通过使用双重确认（`ECHO` 和 `READY` 消息）来增加额外的安全性和验证，从而确保即使在拜占庭容错模型下，只要有足够多的正确进程确认了某条消息，这条消息就会被所有正确的进程交付。

解释

“Amplification Step”（放大步骤）在拜占庭可靠广播（ByzantineReliableBroadcast，简称brb）中的应用。

1. **假设条件**：
    - 假设系统中总进程数为 `N = 3f + 1`，其中 `f` 是可能的拜占庭故障进程数。
2. **放大步骤的逻辑**：
    - 如果一个正确的进程 `p1` 以某种方式交付了消息 `m`，那么它必须已经接收到了 `N - f = 2f + 1` 个包含消息 `m` 的 `READY` 消息。
    - 这至少有 `f + 1` 个 `READY` 消息来自正确的进程。

- 这 `f + 1` 个正确进程的 `READY` 消息也会被剩余的 `f` 个正确进程接收，随后这些进程将发送 `READY` 消息（这就是所谓的“放大步骤”）。
- 结果是，每个正确的进程最终都会交付消息 `m`，因为至少有 `2f + 1` 个正确的进程已经发送了包含消息 `m` 的 `READY` 消息。

3. **正确性（Correctness）的指导原则**：
   - **一致性（Consistency）**：根据“Echo”算法的一致性，没有一个正确的进程会在 `READY` 消息中发送与 `m` 不同的消息。
   - **全体性（Totality）**：放大步骤在这里是关键。它确保了即使某些进程可能出现故障，正确的进程仍然能够达成一致。但是，不能仅仅放大 `ECHO` 消息，还需要第三轮消息（即 `READY` 消息）来确保全体性。
4. **良性或诚实进程**：
   - 如果一个进程是正确的或已崩溃，它被认为是良性的或诚实的，即忠实地遵循算法。
5. **均匀全体性（Uniform Totality）**：
   - 如果任何良性进程交付了某条消息，那么每个正确的进程最终也会交付这条消息。
   - Bracha的算法在这种情况下仍然是正确的。

综上所述，通过放大步骤和额外的消息轮次，Bracha的算法能够在拜占庭容错模型下确保一致性和全体性，即使在有恶意进程存在的情况下。这种算法在确保分布式系统中消息可靠传递方面是非常有效的。

表现

Time complexity：3 message

delaysMessage：complexityO(N2)