可靠的广播原语

BestEffortBroadcast, instance beb.

属性

1. **请求（Request）**：`< beb, Broadcast | m >`
   - 这个请求指示“beb”实例将消息“m”广播给所有进程。
   - 广播意味着消息将被发送给系统中的每一个进程。
2. **指示（Indication）**：`< beb, Deliver | p, m >`
   - 这个指示表示消息“m”被广播者“p”发送，并且已经被接收。
   - 当一个进程接收到一条广播消息时，它将触发这个指示。
3. **属性（Properties）**
   - **BEB1. 有效性（Validity）**：如果一个正确的进程广播了消息“m”，那么每个正确的进程最终都会交付“m”。这意味着消息的传递是可靠的，至少对于正确的（即没有发生故障的）进程来说是如此。
   - **BEB2. 无重复（No duplication）**：任何消息都不会被一个进程交付多于一次。这避免了重复处理相同的消息。
   - **BEB3. 无创造（No creation）**：如果一个进程交付了由发送者“s”广播的消息“m”，那么“m”之前必须是由“s”广播的。这保证了广播的消息是由声明的发送者实际发送的，没有被错误创造或篡改。

实现

```
Implements: BestEffortBroadcast, instance beb.
Uses: PerfectPointToPointLinks, instance pl.

upon event < beb, Broadcast | m > do
    for all q ∈ Π do
        trigger < pl, Send | q, m >;

upon event < pl, Deliver | p, m > do
    trigger < beb, Deliver | p, m >;
```

ReliableBroadcast, instance rb

属性

“ReliableBroadcast”（可靠广播，简称rb），扩展了“BestEffortBroadcast”（尽力而为广播）的基本特性，增加了更强的一致性保证。

1. **请求（Request）**：`< rb, Broadcast | m >`
   - 这个请求指示“rb”实例将消息“m”广播给所有进程。

2. **指示（Indication）**：`< rb, Deliver | p, m >`
    - 这个指示表示消息“m”被广播者“p”发送，并且已经被接收。
3. **属性（Properties）**
    - **RB1–RB3**：与“BestEffortBroadcast”（beb）的属性BEB1–BEB3相同。
    - **RB4. 协议（Agreement）**：如果某个正确的进程交付了消息“m”，那么“m”最终会被所有正确的进程交付。这意味着一旦一个正确的进程确认收到消息，其他所有正确的进程也将最终收到该消息。

实现

Implements: ReliableBroadcast, instance rb.

Uses: BestEffortBroadcast, instance beb; PerfectFailureDetector, instance P.

```
# 当可靠广播初始化时触发的事件
upon event < rb, Init > do
    correct := Π;  # 初始化correct集合，包含所有进程
    delivered := ∅;  # 初始化delivered集合，用于跟踪已交付的消息
    for all p in Π do
        from[p] := ∅;  # 初始化from字典，记录每个进程p接收到的消息

# 当有广播消息请求时触发的事件
upon event < rb, Broadcast | m > do # 要广播消息m
    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 # 当一条进程都收到了p广播的来自s的消息m
    if m not in delivered then
        delivered := delivered ∪ {m};  # 将消息添加到delivered集合，避免重复交付
        trigger < rb, Deliver | s, m >;  # 触发可靠广播的交付事件
        from[p] := from[p] ∪ {s, m};  # 记录进程p接收到的来自s的消息m
        if p not in correct then
            trigger < beb, Broadcast | [DATA, s, m] >;  # 如果进程p不在correct集合中，系统将再次通过尽力而为广播发送消息m。这是因为p的状态不确定可能影响了消息m的可靠传递。

# 当完美故障检测器检测到进程崩溃时触发的事件
upon event < P, Crash | p > do
    correct := correct \ {p};  # 从correct集合中移除崩溃的进程
    for all {s, m} in from[p] do
        trigger < beb, Broadcast | [DATA, s, m] >;  # 重新广播该崩溃进程接收过的所有消息
```

工作原理是：通过尽力而为广播（beb）传递消息，并使用完美故障检测器（P）来跟踪进程的崩溃。一旦某个消息被一个进程接收，它就被记录下来，并确保所有正确的（没有崩溃的）进程最终都能交付这条消息。如果检测到进程崩溃，系统将重新广播该进程接收到的所有消息，以确保所有消息都能被正确的进程接收和交付。这种机制增加了消息交付的可靠性，确保了即使在面临进程崩溃的情况下，所有正确的进程也能接收到广播的消息。

表现

1. **最佳情况**
   - **条件**：初始发送者没有崩溃。
   - **性能**：
     - **通信步骤**：单个通信步骤。
     - **消息数量**：O(N)条消息，其中N是系统中进程的数量。
   - **解释**：在这种情况下，由于初始发送者没有崩溃，其发送的消息会被所有其他进程接收并处理。每个进程只需要接收并交付一次消息，因此总消息数量与进程数成线性关系。
2. **最坏情况**
   - **条件**：进程按顺序依次崩溃。
   - **性能**：
     - **通信步骤**：O(N)步，每个进程崩溃时都可能需要一次额外的通信步骤。
     - **消息数量**：$O(N^2)$条消息。
   - **解释**：在最坏情况下，每个进程在崩溃前可能已经接收到了消息并需要对其进行广播。由于进程依次崩溃，每个后续的进程都需要重新广播这些消息，导致总消息数量呈二次方增长。

另一个版本

```
# 当可靠广播系统初始化时触发的事件
upon event < rb, Init > do
    delivered := ∅;  # 初始化delivered集合，用于跟踪已交付的消息

# 当有广播消息请求时触发的事件
upon event < rb, Broadcast | m > do
    trigger < beb, Broadcast | [DATA, self, m] >;  # 通过尽力而为广播发送包含消息m的广播请求

# 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
    if m not in delivered then  # 如果消息m尚未被交付
        delivered := delivered ∪ {m};  # 将消息m标记为已交付
        trigger < rb, Deliver | s, m >;  # 触发可靠广播的消息交付事件
        trigger < beb, Broadcast | [DATA, s, m] >;  # 重新通过尽力而为广播发送消息m
```

表现：

最佳情况：单个通信步骤和O（N2）消息

最坏情况：进程按顺序崩溃，O（N）步和O（N2）消息

UniformReliableBroadcast, instance urb.

属性

“UniformReliableBroadcast”（统一可靠广播，简称urb），扩展了常规的“ReliableBroadcast”（可靠广播）机制，增加了更强的一致性保证。

2. **请求（Request）**：`< urb, Broadcast | m >`
    - 这个请求指示“urb”实例将消息“m”广播给所有进程。
3. **指示（Indication）**：`< urb, Deliver | p, m >`
    - 这个指示表示消息“m”被广播者“p”发送，并且已经被接收。
4. **属性（Properties）**
    - **URB1–URB3**：与常规可靠广播（RB1–RB3）的属性相同。
        - **URB4: 统一协议（Uniform agreement）**：如果某个进程（无论是正确的还是有故障的）交付了消息“m”，那么“m”最终会被所有正确的进程交付。这意味着一旦任何进程（无论状态如何）确认收到消息，所有其他正确的进程也将最终收到该消息。

确保即使在面对进程故障的情况下，所有正确的进程也能以一致的方式接收到每条广播的消息。

实现

```
# 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，用于跟踪待交付的消息
    correct := Π;  # 初始化correct集合，包含所有进程
    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

# 当完美故障检测器检测到进程崩溃时触发的事件
upon event < P, Crash | p > do
    correct := correct \ {p};  # 从correct集合中移除崩溃的进程

# 检查是否可以交付消息的函数
function candeliver(m) returns Boolean is
    return (correct ⊆ ack[m]);  # 如果所有正确的进程都确认接收了消息m，则返回true

# 检查并交付消息的逻辑
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 >;  # 触发统一可靠广播的交付事件
```

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是系统中进程的数量。
   - **解释**：在最佳情况下，每个进程都需要广播一次消息，然后再根据其他进程的响应决定是否需要再次广播。由于每个进程都可能参与广播，所以总消息数量与进程数的平方成正比。
2. **最坏情况**
   - **通信步骤**：N+1步。
   - **消息数量**：$O(N^2)$条消息。
   - **解释**：在最坏的情况下，可能需要多个步骤来确保所有正确的进程都接收到了消息。每个进程至少参与一次广播，加上可能的重复广播，使得总步骤数可能达到N+1。同样，由于每个进程都可能参与广播，总消息数量也是与进程数的平方成正比。
3. **与常规可靠广播的比较**
   - 相比于常规的可靠广播（ReliableBroadcast），统一可靠广播（UniformReliableBroadcast）需要额外的一个步骤来确保即使是崩溃的进程也能被考虑在内，从而保证消息的统一交付。这导致在最坏情况下，统一可靠广播的步骤数比常规可靠广播多一步。

因果顺序广播原语

Logical clocks (Lamport’s clocks)逻辑时钟（兰波特时钟）

1. **每个进程维护自己的事件计数器**：
   - `cnt(p)`：这是进程`p`维护的一个事件计数器。每个进程都有自己的计数器来跟踪它经历的事件数量。
2. **计数器随每个步骤增加**：
   - 每当进程`p`执行一个操作（比如发送消息、处理本地事件等），它就会将`cnt(p)`增加1。这样，计数器反映了该进程经历的事件总数。
3. **发送消息时包含计数器**：
   - 当进程`p`发送消息时，它会在消息中包含当前的计数器值`cnt(p)`。这允许接收消息的进程了解发送者在发送消息时的“时间”状态。
4. **接收消息时更新计数器**：
   - 当进程`q`接收来自进程`p`的消息时，它会执行以下操作：
     - `cnt(q) := max(cnt(q), cnt(p)) + 1`：进程`q`将其计数器更新为`cnt(q)`和`cnt(p)`中的较大值加1。这个操作确保了`q`的计数器不仅反映了它自己的事件，还考虑了从`p`接收到的消息中蕴含的事件顺序。

尽管逻辑时钟是一种有用的工具，它们在表达事件间的因果关系方面存在一些局限性。下面我将解释这个问题以及向量时钟（Vector Clocks）作为解决方案。

1. **逻辑时钟的局限性**
    - 逻辑时钟通过为每个事件分配一个单一的数值来提供事件的顺序，但这种方法不能完全传达事件间的因果关系。
    - 举个例子：如果我们有两个事件a和b，即使逻辑时钟的值`lc(a)`小于`lc(b)`（意味着事件a在事件b之前发生），我们也不能肯定事件a导致了事件b。这是因为逻辑时钟不能区分并发事件和因果关联的事件。
2. **向量时钟的解决方案**

Vector clocks矢量时钟

每个进程维护一个向量计数器，用于记录系统中所有进程的事件发生情况。

(p1的时钟，p2的时钟，p3的时钟，...)

CausalOrderReliableBroadcast, instance crb.

属性

这个机制不仅提供了常规可靠广播的特性，还增加了额外的因果顺序保证。

1. **请求（Request）**：`< crb, Broadcast | m >`
    - 这个请求指示“crb”实例将消息“m”广播给所有进程。
2. **指示（Indication）**：`< crb, Deliver | p, m >`
    - 这个指示表示消息“m”被广播者“p”发送，并且已经被接收。
3. **属性（Properties）**
    - **CRB1–CRB4**：与常规可靠广播（ReliableBroadcast）的属性RB1–RB4相同。
        - 这包括消息的有效性、无重复、无创造，以及如果一个消息被某个正确的进程交付，那么它最终会被所有正确的进程交付。
    - **CRB5. 因果交付（Causal delivery）**：对于任何因果先行的消息`m1`和`m2`（即`m1 → m2`），没有进程会交付`m2`，除非它已经交付了`m1`。
        - 这意味着如果一个消息`m2`在逻辑上依赖于另一个消息`m1`（例如`m1`的内容影响了`m2`的生成或内容），那么系统会确保每个进程首先交付`m1`，然后才是`m2`。

实现

```
# 当因果顺序可靠广播系统初始化时触发的事件
upon event < crb, Init > do
    delivered := ∅;  # 初始化delivered集合，用于跟踪已交付的消息
    past := [];  # 初始化past列表，用于记录已发送和已接收的消息的历史记录

# 当有广播消息请求时触发的事件
upon event < crb, Broadcast | m > do
    trigger < rb, Broadcast | [DATA, past, m] >;  # 通过可靠广播发送消息，消息包含过去的消息历史和当前消息
    append(past, (self, m));  # 将当前消息添加到past列表中
```

```
# 当通过可靠广播交付消息时触发的事件
upon event < rb, Deliver | p, [DATA, mpast, m] > do
    if m not in delivered then  # 如果消息m尚未被交付
        for all (s, n) in mpast do  # 遍历消息历史
            if n not in delivered then
                trigger < crb, Deliver | s, n >;  # 交付历史中尚未交付的消息
                delivered := delivered ∪ {n};  # 将消息标记为已交付
                if (s, n) not in past then
                    append(past, (s, n));  # 将历史消息添加到past列表中
        trigger < crb, Deliver | p, m >;  # 交付当前消息
        delivered := delivered ∪ {m};  # 将当前消息标记为已交付
        if (p, m) not in past then
            append(past, (p, m));  # 将当前消息添加到past列表中
```

这个算法的核心思想是：

- 在广播消息时，每个进程不仅发送当前的消息，还发送它知道的所有先前消息的历史记录。
- 当一个进程接收到一条消息时，它会首先确保所有历史消息（即因果先于当前消息的消息）都已被交付，然后才交付当前消息。
- 这样做保证了消息的交付遵循因果顺序，即任何消息`m2`只有在导致`m2`的所有消息`m1`（`m1 → m2`）都被交付之后才被交付。

因此，“CausalOrderReliableBroadcast”通过维护消息的因果关系，确保了消息交付的顺序与其产生的顺序一致，这对于分布式系统中的一致性和逻辑正确性非常重要。

表现

1. **通信步骤的性能**
   - 相比于常规的可靠广播算法，因果顺序可靠广播算法并不增加额外的通信步骤。
2. **消息大小的问题**
   - 一个显著的问题是，随着时间的推移，每个消息中包含的“past”历史记录会线性增长。这是因为进程在每次广播消息时，都会附加它之前知道的所有消息的历史记录。
3. **优化方案：截断“past”历史记录**
   - 为了解决消息大小不断增长的问题，一种可行的方法是截断“past”历史记录。这可以通过使用完美故障检测器（Perfect Failure Detector）来实现。
   - 使用完美故障检测器，可以确定哪些历史消息已经被所有进程接收（或者哪些进程已经崩溃，不再需要接收消息）。基于这些信息，可以安全地从消息中移除那些已知已被所有活跃进程接收的历史记录。

```
# 当因果顺序可靠广播系统初始化时触发的事件
upon event < crb, Init > do
    delivered := ∅;  # 初始化delivered集合，用于跟踪已交付的消息
    past := [];  # 初始化past列表，用于记录已发送和已接收的消息的历史记录
    correct := Π;  # 初始化correct集合，包含所有进程
    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

# 当完美故障检测器检测到进程崩溃时触发的事件
upon event < P, Crash | p > do
    correct := correct \ {p};  # 从correct集合中移除崩溃的进程
```

```
# 当存在已交付的消息且当前进程尚未确认该消息时
upon exists m ∈ delivered such that self not in ack[m] do
    ack[m] := ack[m] ∪ {self};  # 将当前进程添加到消息m的确认集合中
    trigger < rb, Broadcast | [ACK, m] >;  # 广播对消息m的确认

# 当通过可靠广播交付确认消息时触发的事件
upon event < rb, Deliver | p, [ACK , m] > do # 当收到进程p的对于消息m的ack确认时
    ack[m] := ack[m] ∪ {p};  # 更新消息m的确认集合，添加确认消息的进程p

# 当所有正确的进程都确认了消息m时
upon correct ⊆ ack[m] do
    remove(past, (*, m));  # 从past列表中移除所有与消息m相关的条目
```

我们如何才能完全消除“过去”？：使用矢量时钟（的变体）

```
# 当因果顺序可靠广播系统初始化时触发的事件
upon event < crb, Init > do
    V := [0]N;  # 初始化一个大小为N的向量V，所有元素初始值为0
    lsn := 0;  # 初始化逻辑序列号lsn，用于标记发送的每条消息
    pending := ∅;  # 初始化pending集合，用于存储待处理的消息

# 当有广播消息请求时触发的事件
upon event < crb, Broadcast | m > do
    W := V;  # 创建当前向量V的副本W
    W[rank(self)] := lsn;  # 在W中更新当前进程的逻辑序列号
    lsn := lsn + 1;  # 逻辑序列号自增
    trigger < rb, Broadcast | [DATA, W, m] >;  # 通过可靠广播发送消息m和向量W

# 当通过可靠广播交付消息时触发的事件,如果消息满足交付条件（即它的向量时钟小于或等于接收进程的向量时钟），则进行交付，并更新接收进程的向量时钟。
upon event < rb, Deliver | p, [DATA, W, m] > do
    pending := pending ∪ {(p, W, m)};  # 将接收到的消息添加到pending集合中
    while exists (p', W', m') ∈ pending such that W' ≤ V do  # 检查是否存在可交付的消息
        pending := pending \ {(p', W', m')};  # 从pending中移除该消息
        V[rank(p')] := V[rank(p')] + 1;  # 更新向量V中对应进程的值
        trigger < crb, Deliver | p', m' >;  # 交付消息m'
```

表现2

“CausalOrderReliableBroadcast”（因果顺序可靠广播，简称crb）算法相对于常规的可靠广播（ReliableBroadcast）算法

1. **通信步骤**
   - **与可靠广播相比**：因果顺序可靠广播算法没有增加额外的通信步骤。这意味着，在完成消息传递的过程中，两种算法需要的通信轮数是相同的。换句话说，消息从发送到被所有正确的进程接收的总轮数保持不变。
2. **消息大小**

- **O(n)消息大小**：因果顺序可靠广播算法中，每条消息都包含一个向量时钟，其大小随着系统中进程数量的增加而线性增长（O(n)）。这是因为向量时钟需要为系统中的每个进程维护一个计数器。
- **向量时钟的作用**：向量时钟记录了发送消息时，发送进程视角下系统中所有进程的逻辑时间。这对于保证消息按照因果顺序被交付是必要的，但随之带来的是消息大小的增加。

总的来说，尽管因果顺序可靠广播算法在通信步骤上与常规可靠广播相似，但由于引入了向量时钟，每条消息的大小随着系统中进程数量的增加而增大。这可能会对系统的通信效率和存储需求产生影响，特别是在有大量进程参与的大型分布式系统中。

如何获得一致可靠因果序广播？:How do we get the Uniform Reliable Causal Order Broadcast?

简单地用urbBroadcast取代rbBroadcast

CausalOrderUniformReliableBroadcast, instance curb.

属性

它结合了统一可靠广播（UniformReliableBroadcast）的特性和因果顺序广播（CausalOrderBroadcast）的因果顺序保证。

1. **请求（Request）**：`< curb, Broadcast | m >`
   - 这个请求指示“curb”实例将消息“m”广播给所有进程。
2. **指示（Indication）**：`< curb, Deliver | p, m >`
   - 这个指示表示消息“m”被广播者“p”发送，并且已经被接收。
3. **属性（Properties）**
   - **CURB1–CURB4**：与统一可靠广播（UniformReliableBroadcast）的属性URB1–URB4相同。
     - 这包括消息的有效性、无重复、无创造，以及统一协议（即消息被某个进程交付，则最终被所有正确的进程交付）。
   - **CURB5**：与因果顺序广播（CausalOrderBroadcast）的属性CRB5相同。
     - 即如果消息`m1`因果先行于消息`m2`（`m1 → m2`），则没有进程会交付`m2`，除非它已经交付了`m1`。这保证了消息按照它们的因果关系顺序被交付。

“CausalOrderUniformReliableBroadcast”在需要严格维护事件顺序和因果关系的分布式系统中非常重要。它确保了即使在进程崩溃和网络问题的情况下，所有正确的进程都能以一致的顺序接收到消息，且这些消息的顺序反映了它们之间的因果关系。