并行与分布式计算 - June 22'th, 2018

Process Model Concepts

https://github.com/rh01

进程协作

- 进程协作举例：多个并发进程要求访问共享资源时，进程在共享资源上如何协调工作，才能不致于发生冲突，且保证共享资源的正确性和完整性?
- 进程协作分类：
  - 如何保证临界区资源的互斥利用—分布式互斥
  - 如何从多个并发进程中选举出一个进程扮演协调者—选举
  - 如何就一组进程间发生的事件的发生顺序达成一致—事件排序；组通信中的排序组播
  - 要求一组进程对共享资源进行公平的原子访问，不出现死锁，不出现饿死—分布式死锁
  - 在异步网络中模拟时钟的滴答 (tick,round)—同步器

分布式互斥

- **目标：** 实现对进程共享资源的排他性访问，保证访问共享资源的一致性
- **途径：** 保证在任何一个时刻，最多只能有一个进程访问临界区
- **具体手段：** 基于消息传送
  - 在分布式系统中，共享变量或者单个本地操作系统内核提供的设施都不能被用来解决这个问题

基本概念

- 进程的历史：由这个进程发生的事件组成
- 系统的全局历史：系统中的单个进程历史的并集
- 系统执行的割集 (cut)：进程历史前缀的并集形成系统全局历史的子集
- 割集的边界：是指由进程处理的最后一个事件的集合
- 一致的割集：如果对这个割集包含的每个事件，它也包含了所有在该事件之前发生的所有事件
- 一致的全局状态：指相对于一致割集的状态
- **走向 (run)**是对全局历史中所有事件的全排序，同时，它与每个本地历史排序是一致的
- **一致的走向 (consistent run) 或线性化走向 (linearization)** 是与全局历史上的发生在先关系一致的所有事件的一个排序
- 不是所有的走向都经历一致的全局状态，但所有线性化走向仅经历一致的全局状态
- 如果有一个经过状态 S 和 S' 的线性化走向，那么**状态 S' 是从状态 S 可达的**

Essential requirements for mutual exclusion

- ME1: (mutual exclusion, a safety property) at most one process may execute in the critical section (CS) at a time
- ME2: (freedom from deadlocks and livelocks) requests to enter and exit the critical section eventually

succeed

  - ME2 implies freedom from both deadlock (at least one process must be eligible to enter its critical section: a safety property) and starvation (every process trying to enter the CS must eventually succeed: a liveness property)

- ME3: ($\rightarrow$ ordering) if one request to enter the CS happened-before another, then entry to the CS is granted in that order
- Fairness property
  - the absence of starvation: progress
  - keep the fair order in which processes enter the CS

中央服务器互斥算法

- 假设系统是异步的，进程不出故障，并且采用异步消息传递 (optional)，消息传递是可靠的，这样任何传递的消息最终都被完整地发送恰好一次
- 为进入一个临界区，一个进程向服务器发送一个请求消息并等待服务器的权标 (token)。如果在请求时没有其它进程拥有这个权标，服务器就立刻应答来授予权标。如果此时另一进程持有这个权标，服务器就不应答而是把请求放入队列
- 进程在退出临界区时，发一个消息给服务器，交回这个权标。如果等待进入临界区的进程队列不空，那么，服务器就会选择时间最早的进程，把它从进程队列中删除并给这个进程一个应答。这样，这个进程就持有了权标

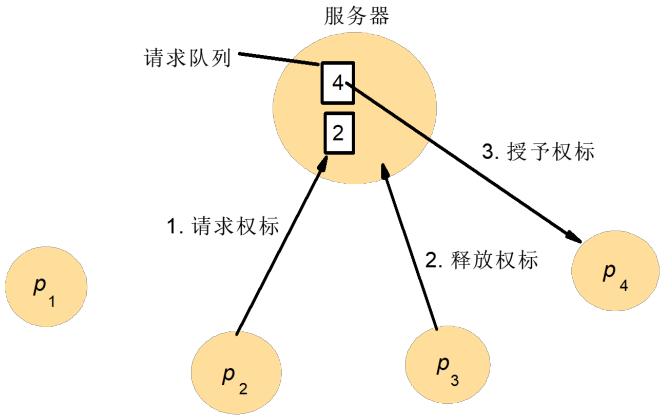


图 1：中央服务器互斥算法

- 算法分析
  - 如果系统不发生故障，满足 ME1、ME2
  - 不满足 ME3(顺序条件)
    * 反例：进程 A 先发进入临界区的请求 ra，然后给进程 B 发一个消息 m。B 在收到 m 后，发进入临界区的请求 rb。如果要满足顺序条件，应该先处理 A 的请求，再处理 B 的请求。由于消息传递有延迟，B 的请求可能会先于 A 的请求到达服务器，所以服务器会先处理 B 的请求。
  - 性能 (假设采用异步消息传递)
    * 消息数量和延迟：进入临界区需要两个消息 (请求和随后的授权)，客户被延迟了这对消息的往返

时间；退出临界区需要一个释放消息，客户端没有发生延迟

    * 同步延迟：发到服务器的释放消息，和随后让下一进程进入临界区的授权消息，这两个消息之间花的总时间
    * 吞吐量：服务器可能会成为整个系统的一个性能瓶颈
  - 服务器会造成单点失败
    * 需要后备 (backup) 节点

基于环的互斥算法

- 把进程安排成环，环的拓扑结构可以与计算机之间的物理互连无关。每个进程 pi 只需与环中下一个进程 p(i+1) mod N 有一个通信通道
- 权标在进程环中沿着环顺时针或逆时针传递，获得权标，就能访问共享资源
- 如果一个进程在收到权标时不需要进入临界区，那么它立即把权标传给它的邻居。需要权标的进程将一直等待，直到接收到权标，它在访问共享资源时，会保留权标。为了退出临界区，进程把权标发送到它的邻居

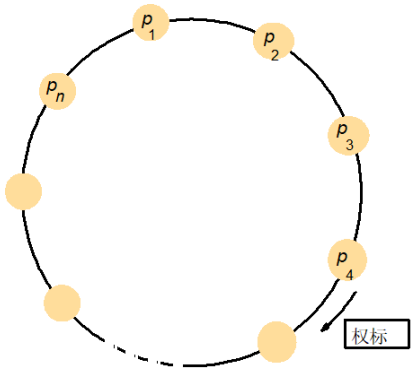


图 2：基于环的互斥算法

- 算法分析
  - 满足安全性、活性条件
  - 不满足顺序条件 (ME3)
    * 反例：P1→ P2→ P3→P1。P1，P2，P3 形成一个传递权标的通信环路。P1 先发请求，想进入临界区，然后 P1 给 P3 发消息，即 P1、P3 构成发生在先关系，P1→P3，然后 P3 发请求，也想进入临界区。但目前权标在 P2，所以先传到了 P3，所以 P3 的请求先获得了满足
  - 性能
    * 该算法连续地消耗网络带宽
    * 请求进入临界区的进程会延迟 0 个 (这时，它正好收到权标) 到 N 个 (这时，它刚传递了权标) 消息
    * 退出临界区只需要一个消息
    * 在一个进程离开和下一个进程进入临界区之间的同步延迟可以是 1 到 N 个消息传输
  - 故障的检测和处理
    * 故障：进程崩溃，消息 (token) 丢失

Lamport Algorithm: Using multicast and logical clocks

- Assumptions: message passing is FIFO and reliable, but asynchronous.
- The extended timestamp is in the form of (C(event), process id)
- For events a and b, a→b if either of the following is true
  - C(a)<C(b)
  - C(a)=C(b) but pid(a)<pid(b), where pid is the process identifier
- Each process Pi maintain a local request queue Q.
- Timestamped requests are sorted in total order → in Q.
- There are three messages: Request, Reply, Release message
- Steps
  - Pi makes a request by sending a message Request (C(reqi),i) to all process, including itself
  - When Pj receives a request message Request (C(reqi),i) from Pi, it puts into Q and sends a reply message Reply (C(repj),j) to Pi
  - Pi can enter its CS and use the resource when the following two conditions are true:
  - Pi' s own request is at the head of Q, and
  - Pi has received a message rk (either Request, Reply or Release) from all process Pk with timestamp C(rk), such that (C(reqi),i) < (C(rk),k)
  - Pi release the source by removing its request from Q, and send a release message Release (C(reli),i) to all processes.
  - When Pj receives a release message from Pi, it removes Pi' s request from Q.

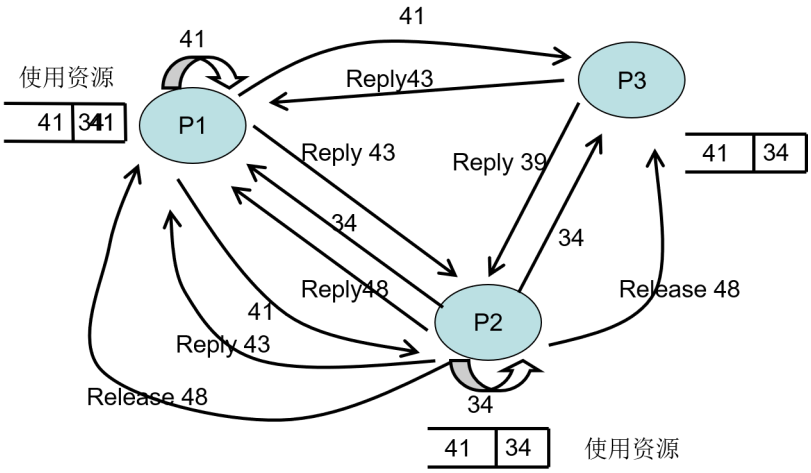


图 3：Example

The same example in terms of space-time diagram

- Note that at local time 44, P2 may start using the resource, since it has already received a message from P1 with a larger timestamp of value 41 than P2' s request of value 34.
- Furthermore, this solution grants resource in an order consistent with the happens-before relation (no priority inversion).

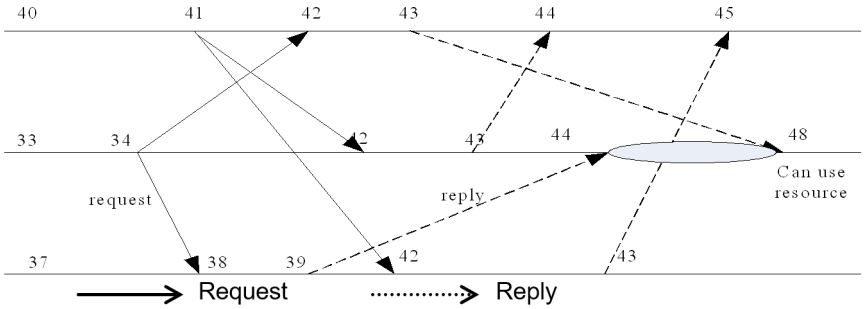


图 4：Example

- Correctness
  - Mutual exclusion
    * All queues are ordered the same way, and a request cannot be granted until it is at head (step 3-a) and no request of a smaller timestamp can arrive (step 3-b)
    * This depends on reliable FIFO messages: receiving a message means that all old messages are received.
    * Receiving a message from all processes with timestamps larger than that of request means no future request will come with a smaller timestamp.
    * No two processes get resource at the same time
  - Progress
    * Step 2 guarantees that after Pi requests the resource, reply messages to Pi will eventually arrive, and so the condition in step 3-b will eventually hold.
    * Steps 4 and 5 imply that if each process which is granted the resource eventually release it, then the condition in step 3-a will eventually hold.
    * As a result, Step 3 will be executed.
  - FIFO fairness
    * All local queues are sorted according to $\rightarrow$, which is the same on each process.
    * All accesses to the resource follow the order of the queue.

Ricart-Agrawala algorithm: an optimized version of Lamport' s solution

- timestamped request to every other process in the system
- A process receiving a request sends a reply back to the sender, only when (i) the process is not interested in entering its CS, or (ii) the process is trying to enter its CS, but its timestamp is larger than that of the sender. If the process is already in its CS, then it will buffer all requests until its exit from CS
- A process enters its CS, when it receives an reply from each of the remaining n-1 processes
- Upon exit from its CS, a process must send reply to each of the pending requests before making a request or executing other actions

- Progress
  - Step 2 guarantees that after Pi requests the resource, reply messages to Pi will eventually arrive,

and so the condition in step 3-b will eventually hold.

  - Steps 4 and 5 imply that if each process which is granted the resource eventually release it, then the condition in step 3-a will eventually hold.
  - As a result, Step 3 will be executed.

- FIFO fairness
  - All local queues are sorted according to $\rightarrow$, which is the same on each process.
  - All accesses to the resource follow the order of the queue.

- Each process seeking entry into its CS sends a timestamped request to every other process in the system
- A process receiving a request sends a reply back to the sender, only when (i) the process is not interested in entering its CS, or (ii) the process is trying to enter its CS, but its timestamp is larger than that of the sender. If the process is already in its CS, then it will buffer all requests until its exit from CS
- A process enters its CS, when it receives an reply from each of the remaining n-1 processes
- Upon exit from its CS, a process must send reply to each of the pending requests before making a request or executing other actions

```
On initialization:
    state := RELEASED;
To enter the section:
    state := WANTED;
    multicast request <T, pi> to all processes where T:= request' s timestamp;
    wait until  (number of replies received= (N − 1));
    state := HELD;
On receipt of a request <Ti, pi> at pj (i $\ne$ j):
    if  (state = HELD or (state = WANTED and (T, pj) < (Ti, pi)))
    then
        queue request from pi without replying;
    else
        reply immediately to pi;
    end if
To exit the critical section:
    state := RELEASED;
    reply to any queued requests;
```

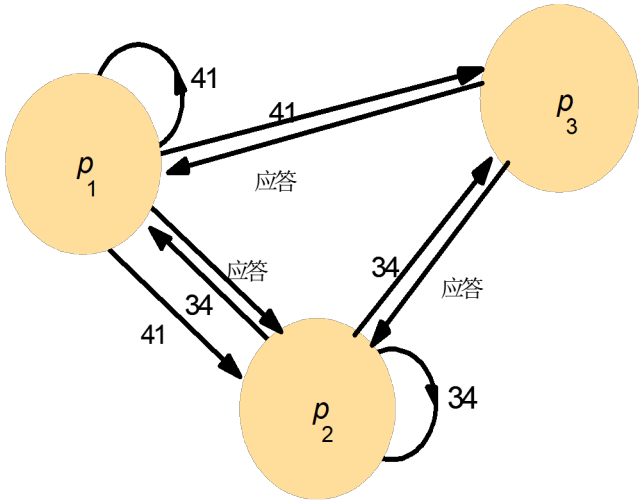


图 5：Example

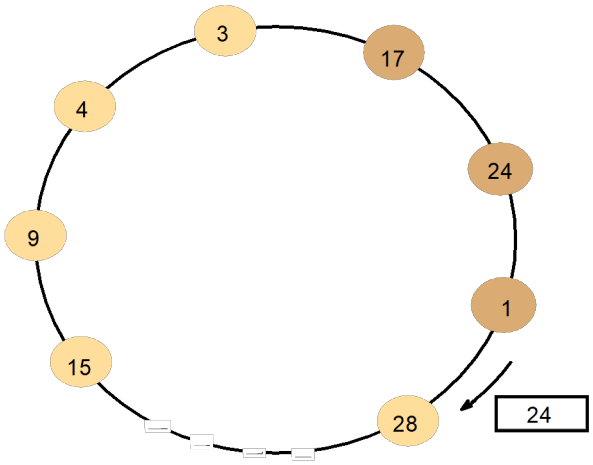
- 达到安全性特性 ME1。反证法：如果两个进程 pi 和 pj（i≠j）能同时进入临界区，那么这两个进程必须已经互相回答了对方。但是，因为 <Ti, pi> 对是全排序的，所以这是不可能的
- 满足 ME2 和 ME3
- 性能
  - 在 n 个进程的环境中，一个进程进入临界区，需要 2(n-1) 次消息传递。或者，如果硬件支持组播，请求只需要一个消息，总数是 n 个消息
  - 同步延迟仅是一个消息传输时间
- The management of the multicast group membership
- N points of failure: some processes fail
  - Improvement: the receiver always sends a reply, either granting or denying permission.
- Why does a process need to gather permission from all other processes?
  - Improvement: ask only permission from a majority of processes / the quorum of processes

选举

- 选举算法是所有进程都同意某个进程来扮演特定角色的过程，不失一般性，现在要求选择具有最大标识符的进程为当选进程
- 选举算法的基本要求：
  - 假设每个进程 pi(i =1, 2⋯, N) 有一个变量 $elected_i$，用于包含当选进程的标识符
  - E1(安全性)：参与的进程 pi 或者 $elected_i = \perp$ 或者 $elected_i = P$，其中，P 是在运行结束时具有最大标识符的非崩溃进程
  - E2(活性)：所有进程 pi 都参加选举并且最终置 $elected_i \neq \perp$ 或者崩溃
- 按下列标准评价选举算法：
  - 总的网络带宽使用，它与发送消息的总数成比例
  - 算法的回转时间，即从启动算法到终止算法之间的串行消息传输的次数

基于环的选举算法

- 假设系统是异步的并且系统不发生故障
- 假定一组进程排列在一个环上，每个进程 pi 有一个到下一进程 p(i + 1) mod N 的通信通道，消息沿着环顺时针发送
  - 单向环 (a unidirectional ring)
  - 进程之间仅需具有最小的先验知识：每个进程只知道如何与邻居通信
- 算法的目标：选举具有最大标识符的进程 (协调者)
- 当一个进程收到一个选举消息时，它比较消息里的标识符和它自己的标识符。如果到达的标识符较大，它把消息转发到它的邻居。如果到达的标识符较小，且接收进程不是一个参加者，它把消息里的标识符替换为自己的，并转发消息；如果到达的标识符较小，且它已经是一个参加者，它就不转发消息
- 如果收到的标识符是接收者自己的，那么，这个进程的标识符一定最大，该进程就成为协调者。协调者再次把自己标记为非参加者并向它的邻居发送一个当选消息，宣布它的当选并将它的标识符放入消息中
- 最初，每个进程被标记为选举中的一个非参加者。可以从任何一个进程开始一次选举。它把自己标记为一个参加者，然后，把自己的标识符放到一个选举消息里，并把消息发送到它的顺时针邻居
- 当进程收到一个当选消息时，它把自己标记为非参加者，置变量 $elected_i$ 为消息里的标识符，并且把消息转发到它的邻居，除非它是新的协调者
- 算法分析
  - 满足条件 E1：参与的进程 pi 或者 $elected_i = \perp$ 或 $elected_i$=P
  - 满足条件 E2：所有进程 pi 都参加选举并且最终置 $elected_i \neq \perp$ 或者崩溃
  - 性能
    * 总的网络带宽使用 (发送消息的数量)：最坏的执行情况是启动选举的进程，它的逆时针邻居具有最大的标识符。这时到达该邻居需要 N-1 个消息，并且还需要 N 个消息再完成一个回路，才能宣布它的当选。接着当选消息被发送 N 次，共计 3N-1 个消息
    * 回转时间：3N-1 个消息



注：选举从进程17开始。
到目前为止，所遇到的最大的进程标识符是 24。
参与的进程用深色显示。

图 6：Example

霸道（Bully）算法-算法的假设

- 假设：进程可能崩溃；消息传递是可靠的，并能在一段时间内完成；所有进程知道对方的标识符 (pid)
- 假设系统是同步的，并允许在选举期间进程崩溃，(它可以构造一个可靠的故障检测器 (使用超时) 来检测进程故障)
  - 基于环的算法可用于异步系统，霸道算法仅用于同步系统
  - 基于环的算法是假设没有故障的，霸道算法假定进程可能在选举期间崩溃
  - 基于环的算法假定进程相互之间具有最小的先验知识。霸道算法假定每个进程知道哪些进程有较高的标识符，并且可以和所有这些进程通信
- 霸道算法中有三种类型的消息：选举消息用于宣布选举；回答消息用于回复选举消息；协调者消息用于宣布当选进程的身份——新的“协调者”
- 一个进程在通过超时发现协调者已经出现故障时开始一次选举
  - 几个进程可能并发地观察到此现象

霸道算法-算法的步骤

- 有较低标识符的进程开始一次选举的过程：发送选举消息给那些有较高标识符的进程，并等待回复的回答消息。如果在时间 T 内没有消息到达，该进程认为自己是协调者，并发送协调者消息给所有有较低标识符的进程来宣布这一结果。否则，该进程再等待时间 T’ 用于接收从新的协调者发来的消息。如果没有消息到达，它开始另一次选举
- 知道自己有最高标识符的进程可以发送协调者消息给所有有较低标识符的进程，来选举自己为协调者
- 如果进程 pi 收到一个协调者消息，它把它的变量 $elected_i$ 置为消息中包含的协调者的标识符，并把这个进程作为协调者
- 如果进程 pi 收到一个选举消息，它回送一个回答消息并开始另一次选举——除非它已经开始了一次选举

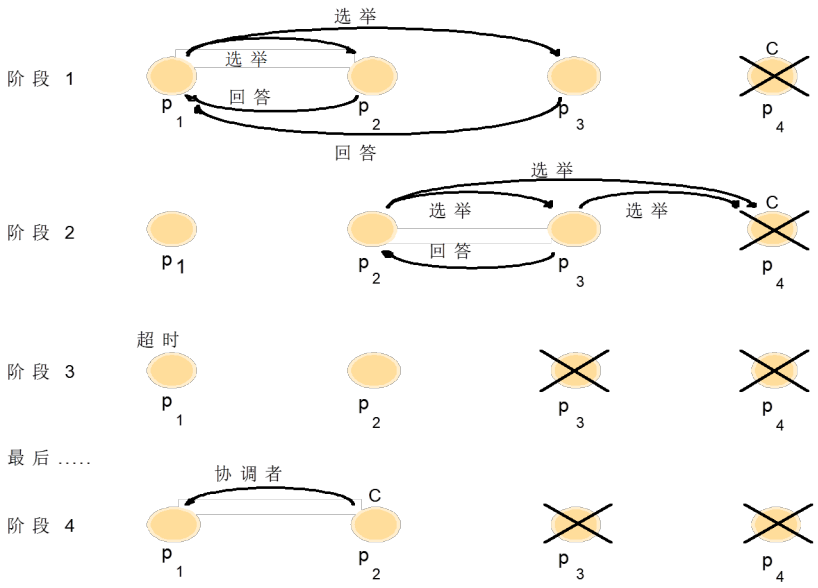


图 7：Example

- 算法分析

- 如果没有进程被替换 (指进程崩溃后重启)，算法满足条件 E1
- 如果允许进程被替换，那么如果崩溃的进程被替换为具有相同标识符的进程，这个算法就不能保证满足 E1
  * 一个进程已经检测到进程 p 的崩溃，然后决定它有最高的标识符，这时，p 的替换进程激活了，替换 p 的进程可能决定它有最高的标识符。两个进程可能并发地宣布它们自己为协调者。因为在消息的传输顺序上没有保证，所以，这些消息的接收者可能就谁是协调者得出不同的结论
- 如果假定的超时值被证明是不准确的，即如果进程的故障检测器是不可靠的，条件 E1 可能会被违反
  * 假设或者 p3 没有崩溃但运行异乎寻常地慢 (即系统同步的假定是不正确的)，或者 p3 已经崩溃但被替换。正在 p2 发送它的协调者消息时，p3 (或替换者) 也做同样的事情。p2 在发送自己的协调者消息后收到 p3 的，因此置 elected2 = p3。由于消息传输延迟，p1 在收到 p3 的协调者消息后收到 p2 的，因此最终 elected1 = p2。条件 E1 被违反
- 满足活性条件 E2
- 性能
  * 最好情况是具有次高标识符的进程发现了协调者的故障。它可以立即选举自己并发送 N-2 个协调者消息。回转时间是一个消息
  * 在最坏情况下，霸道算法需要 O(N^2) 个消息——即具有最小标识符的进程首先检测到协调者的故障。然后 N-1 个进程一起开始选举，每个都发送消息到有较高标识符的进程

Election in arbitrary networks

- **A ring algorithm** can be used, if a ring is embedded on the given topology
- The orientation of the embedded ring helps messages propagate in a predefined manner
- **Use flooding** to construct a leader election algorithm that will run in rounds
- Initially, ∀i:L(i)=i
- In each round, every node sends the id of its leader to all its neighbors. A process i picks the largest id from the set {L(i) ∪ the set of all ids received}, assigns it to L(i), sends L(i) out to its neighbors
- The algorithm terminates after D rounds, where D is the diameter of the graph. The message complexity is O(δD), where δ is the maximum degree f a node

排序组播

- 组播不能保证消息按序到达
  - 用单播实现组播时，同属一组的一些成员从同一个发送者接收的数据包的顺序可能与其他一些成员不一样
- 应用需要有序组播
  - 排序：BBS，在最低程度上，BBS 需要 FIFO 排序，另外，一个贴子和它的回贴应该是有序出现的即按因果序出现
- 一个组称为是封闭的组，如果只有组的成员可以组播到它
- 一个组称为是开放的组，如果组外的进程可以发送组播消息给它
- 重叠组，一个进程属于两个组
- 假设

- Processes communicate reliably over one-to-one channels
- Processes may fail only by crashing
- For ordering property, processes are restricted to being members of at most one group at a time, otherwise processes are allowed to belong to several groups
- multicast(g,m) sends the message m to all members of the group g of processes
- deliver(m) delivers the message m sent by multicast to the calling process
- Every message m carries the unique identifier of the process sender(m) that sent it , and the unique destination group identifier group(m)

Basic multicast primitive B-multicast

- B-multicast: a correct process will eventually deliver the message, as long as the multicaster does not crash
- B-deliver: the corresponding basic delivery primitive
- A way to implement B-multicast :

```
To B-multicast(g,m):
    for each process p g, send(p,m);
    //send is a reliable one-to-one operation
On receive(m) at p: B-deliver(m) at p.
```

- Ack & Ack-implosion

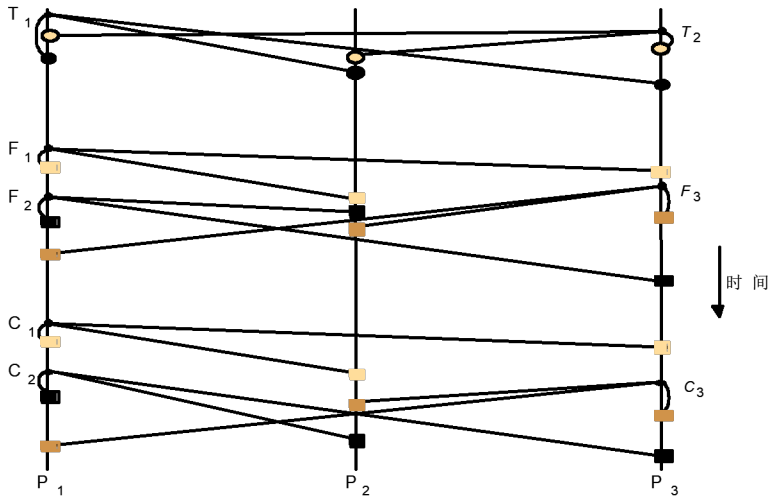


图 8：有序组播并不隐含可靠性。例如，在全排序下，如果正确的进程 p 传递消息 m 然后传递 m’，那么正确的进程 q 可以传递 m 而不传递 m’ 或按序排在 m 后的任何消息。

实现 FIFO 排序组播 FO-multicast

- 假设进程组不重叠
- 设置进程 p 的两个变量：S_g^p 是进程 p 已发送到组 g 的消息计数，R_g^q 是 p 已传递的来自进程 q 并且发往组 g 的最近的消息的顺序数

- p 要 FO-multicast 一个消息到组 g 时，它在消息上捎带值 S_g^p，接着 B-multicast 消息到 g，然后把 S_g^p 加 1
- p 收到来自 q 的顺序数为 S 的消息时，p 检查是否 S = $R_g^q + 1$。如果是，这个消息是预期的来自发送进程 q 的下一个消息，p FO-deliver 消息，并且置 R_g^q:=S。如果 S > $R_g^q + 1$, 它把消息放到保留队列中，直到介于其间的消息已被传递且 S = $R_g^q + 1$

全排序组播

- 实现全排序的基本途径是为组播消息指定一个全排序标识符 s_g，维持一个组的顺序数，使得每个进程可以基于这些标识符做出相同的排序决定
- 指定全排序标识符的方法
  - 由定序者 (sequencer) 的进程来指派
  - 分布式协商
- r_g 是预期的接收组播消息的顺序数。一开始，每个成员的 r_g 为 0

1. 组成员**p** 的算法
On initialization:
r_g:=0;
To TO-multicast message m to group g:
B-multicast(g ∪ {sequencer(g)}, <m, i>);
On B-deliver(<m,i>) with g=group(m)
Place <m,i> in hold-back queue;
On B-deliver(M_{order} = <"order", i, S>) with g=group(M_{order}):
Wait until <m,i> in hold-back queue and S==r_g;
TO-deliver m;//在从保留队列中删除它之后
r_g=S+1;
2. 定序者 **g**的算法
On initialization: S_g:=0;
On B-deliver(<m,i>) with g=group(m):
B-multicast(g,<"order", i, s_g>);
s_g:=s_g+1;

图 9：实现全排序组播 TO-multicast 算法

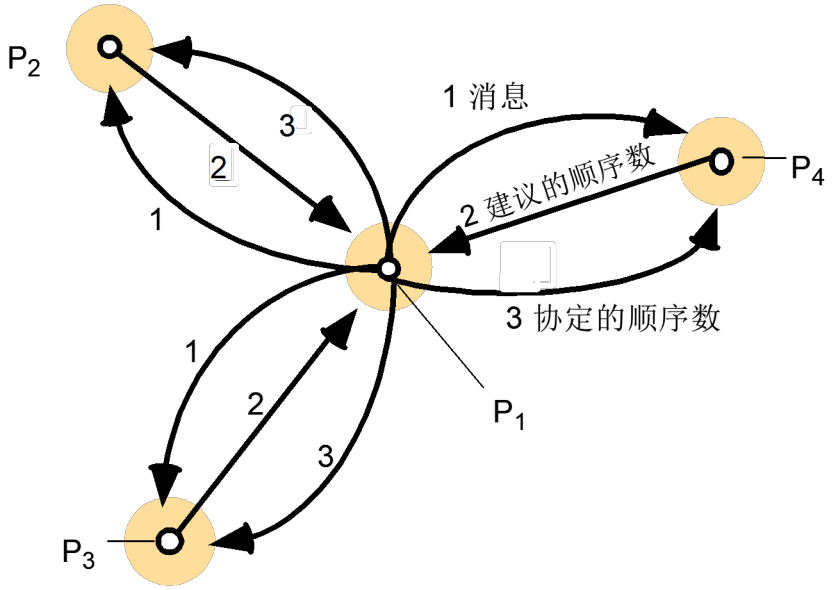


图 10：Example

- 每个进程 q 设置两个变量：
- A_g^q 是迄今为止从组 g 观察到的最大的协定顺序数；
- P_g^q 是它自己提出的最大顺序数；
- 进程 p 组播消息 m 到组 g 的算法：
- p B-multicast <m, i> 到 g, 其中 i 是消息 m 的一个唯一的标识符
- 每个进程 q 回答发送者 p，提议 $P_g^q = \text{Max}(A_g^q, P_g^q) + 1$ 为此消息的协定顺序数。每个进程把提议的顺序数分配给消息，并把消息放入它的保留队列中，在保留队列中，最小的顺序数在队首
- p 收集所有提议的顺序数，并选择最大的数 a 作为下一个协定顺序数。然后它 B-multicast <i, a> 到 g。g 中每个进程 q 置 $A_g^q = \text{Max}(A_g^q, a)$, 并把 a 附加到消息 (标识为 i) 上。如果协定顺序数与提议的不一样，它把保留队列中的消息重新排序

实现因果排序组播 CO-multicast

- 针对非重叠的封闭组
- 对组成员 pi (i=1,2,⋯,N) 的算法

```
On initialization:
    Vgi[j]:=0 (j=1,2,⋯,N); //记录来自j进程的、已发生的、按因果关系在先的组播消息计数
To CO-multicast message m to group g:
    Vgi[i]:= Vgi[i] +1;
    B-multicast(g,< Vgi,m>);
On B-deliver(<Vgj,m>)from pj (i$\ne$j) with g=group(m)
    将<Vgj,m>放入保留队列
    等待直到Vgj[j]= Vgi[j]+1并且Vgj[k] <= Vgi[k] (k$\ne$j)
    CO-deliver m; //在把它从保留队列中删除后
    Vgi[j]:= Vgi[j] +1
```