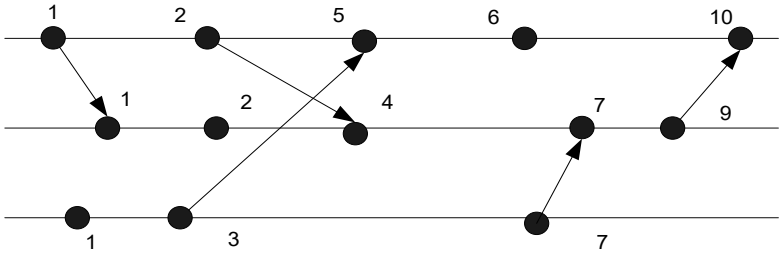
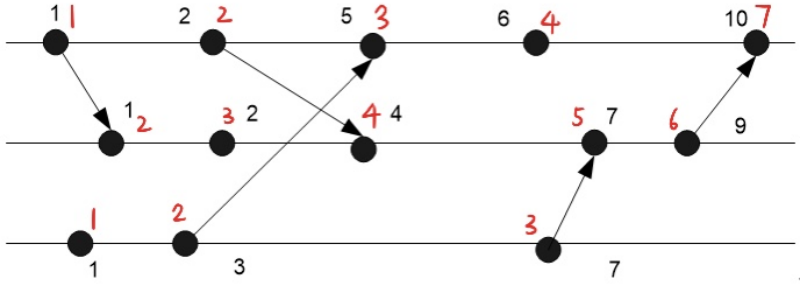
2020 年秋季学期/并行和分布式计算
分布式计算作业

钟赟 202028013229148

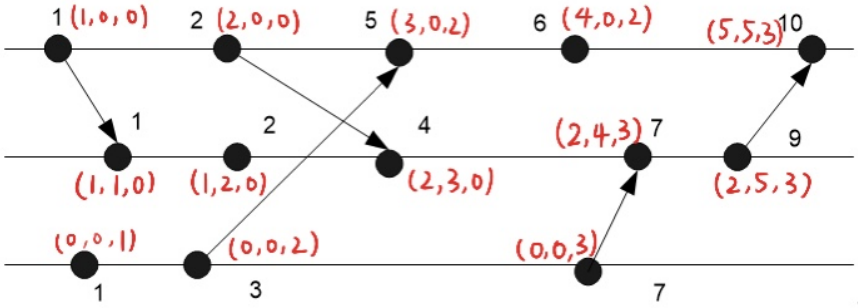
1. **下图中，直线上小黑点给出了时钟计数，请分别用 Lamport 逻辑时钟和向量时钟给图上的事件设置时间戳，并给出一致割集和非一致割集的例子。**



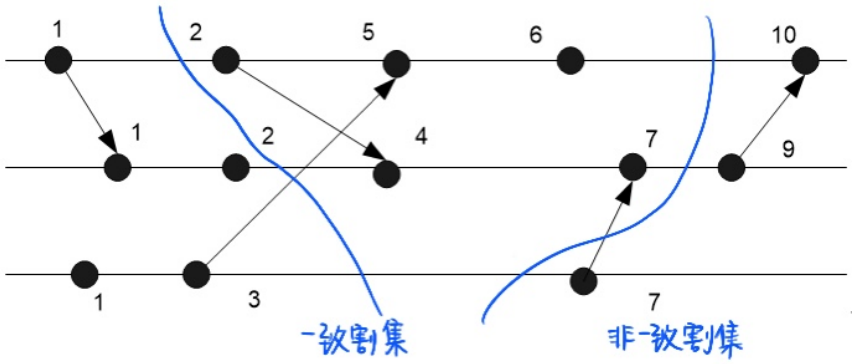
答：逻辑时钟设置时间戳（下图中红色数字）：



向量时钟设置时间戳：



一致割集和非一致割集的例子：



2. 考虑在异步分布式系统中使用的两个通信服务。在服务 A 中，消息可能丢失、被复制或延迟，校验和仅应用到消息头。在服务 B 中，消息可能丢失、延迟或发送得太快以致接收方无法处理它，但到达目的地的消息其内容一定正确。描述每个服务会有的故障类型。根据对有效性和完整性的影响将故障分类。服务 B 能被描述成一个可靠的通信服务吗？

答：

服务 A 可能发生的故障：遗漏故障，随机故障，时序故障。

服务 B 可能发生的故障：遗漏故障，时序故障。

遗漏故障中的通道遗漏属于影响有效性的故障，随即故障属于影响完整性的故障。

服务 B 不能被描述成一个可靠的通信服务，因为消息可能会丢失，不满足有效性。

3. 请证明 Lamport 的互斥算法满足 ME1、ME2 和 ME3。其中，ME1 指在临界区一次最多有一个进程可以执行，ME2 指进入和离开临界区的请求最终成功执行，ME3（→顺序）指如果一个进入临界区的请求发生在先，那么进入临界区任按此顺序。

证明：

Lamport 算法步骤如下：

1. P_i makes a request by sending a message ***Request*** ***($C(req_i),i$)*** to all process, including itself
2. When P_j receives a request message *Request ($C(req_i),i$)* from P_i, it puts into Q and sends a reply message ***Reply ($C(rep_j),j$)*** to P_i 马上给应答
3. P_i can enter its CS and use the resource when the following two conditions are true:
   a) P_i's own request is at the head of Q, and
   b) P_i has received a message r_k (either Request, Reply or Release) from all process P_k with timestamp $C(r_k)$, such that $(C(req_i),i) < (C(r_k),k)$
4. P_i release the source by removing its request from Q, and send a release message ***Release ($C(rel_i),i$)*** to all processes.
5. When P_j receives a release message from P_i, it removes P_i's request from Q.

1) ME1：

若进程 Pi 和 Pj 能够同时进入临界区，说明即 Pi 和 Pj 的请求都位于本地队列 Q 的最前面。FIFO 保证了每个进程中队列的排序方式一样，队列最前面的请求是唯一的，产生矛盾。因此当 Q 不为空，每次最多只可能有 Q 最前面的请求对应的进程可以进入临界区，没有带着更小时间戳的请求可以到达临界区。满足 ME1。

2) ME2：

当进程 Pj 收到 Pi 的请求消息时，会立即给出应答消息，回复 Pi 的消息最终会到达，因此 step 3-b 会发生；step 4 和 step5 表示每个进入临界区的进程最后都会退出，因此 step3-a 也总会发生。因此，step 3 总会发生，即请求进入临界区的进程最终会进入临界区。又因为进入临界区的进程离开临界区不需要请求，总会成功执行，因此 Lamport 算法满足 ME2。

3) ME3：

设(i,reqj/ackj)表示进程 Pi 发请求/确认给 j 的时间戳。假设进程 Pi 的请求的时间戳小于进程 Pj 的请求的时间戳。若 Pj 比 Pi 先进入临界区，那么 Pj 应该还没有收到来自 Pi 的请求。根据 step 3-b，它应该收到来自 Pi 的对 Pj 请求的确认(i,reqj)。由

于通道是 FIFO，所以来自 Pi 的对 Pj 请求的确认应该在来自 Pi 的请求 之前发生(i,ackj) < (i,reqj)。由于先有请求，后才有确认，所以，来自 Pj 的请求在来自 Pi 的对 Pj 请求的确认之前发生(j,reqi) < (i,ackj)。所以，来自 Pj 的请求在来自 Pi 的请求之前发生。因此 Lamport 算法满足 ME3。

4. 请证明 Ricart-Agrawala 的互斥算法满足 ME2 和 ME3。

证明：

1) ME2：

进程 Pi 退出临界区不需要请求，可以成功退出临界区。当 Pi 退出临界区时会对每个挂起的请求发送应答，假设 Pj 的请求是挂起的请求之一，它将收到 Pi 的应答，即 Pj 将收到时间戳比自己的时间戳小的请求的进程的应答；在 Pj 请求进入临界区后，不想进入临界区的无关进程会向 Pj 发送应答，想进入临界区并且请求的时间戳比 Pj 的请求的时间戳大的进程也会向 Pj 发送应答，因此 Pj 最终将收到其他所有进程的应答，保证 Pj 可以进入临界区。满足 ME2。

2) ME3：

假设进程 Pi 的请求的时间戳 Ci 小于进程 Pj 的请求的时间戳 Cj。若 Pj 比 Pi 先进入临界区，那么说明 Pj 进入临界区之前收到了其他所有进程的 reply，包括 Pi 的。而在 Pj 广播其请求后，由于 Ci<Cj，Pi 不会向 Pj 发送 reply，与 Pj 收到了 Pi 的 reply 矛盾。故满足 ME3。

5. 在 Ricart-Agrawala 互斥算法中，原始假定系统的进程是不出故障的。请修改算法增加处理一个进程崩溃的情况。

答：

Ricart-Agrawala 互斥算法中，由于不应大会被认为是资源占用，所以如果有某个节点故障，会导致该算法的异常终止。因此我们做出改进：进入临界区时不再请求其他所有进程的许可，而是只请求大多数进程的许可。

进程 Pi 请求临界区资源，向其他所有进程发送请求；不想进入临界区的进程 Pj 收到 Pi 的请求后向 Pi 做出应答，当且仅当 Pj 没有向除 Pi 以外的其他进程发送过应答，当 Pj 收到 Pi 的 release 消息后，才能向其他进程发送应答。这样保证了优先级最高的请求收到的应答数量是最多的。

进程 Pi 进入临界区的条件是：Pi 收到了$\lceil (n+1)/2 \rceil$个应答。避免了某个进程崩溃后 Pi 无法收到应答而无法进入临界区。

当 Pi 退出临界区后，向曾经收到的应答的发送者发送 release 消息。

6. 改进基于环的互斥算法使得它能检测权标的丢失并重新生成权标。

答：

使用两个权标 A 和 B，其中 A 负责检测 B 可能的丢失，B 负责控制共享资源的访问。令两个权标按照相反方向沿着环访问进程，每个进程记录是否被 A 或 B 访问，如果一个进程被同一个权标连续访问两次，说明另一个权标丢失了，此时重构丢失的权标。

权标 A 和 B 各带有一个变量 (NA, NB)。设环中共有 N 个进程 P1，P2, …,PN，每个进程带有一个变量 Mi(1<=i<=N)。当两个权标相遇时，更新 NA，NB。

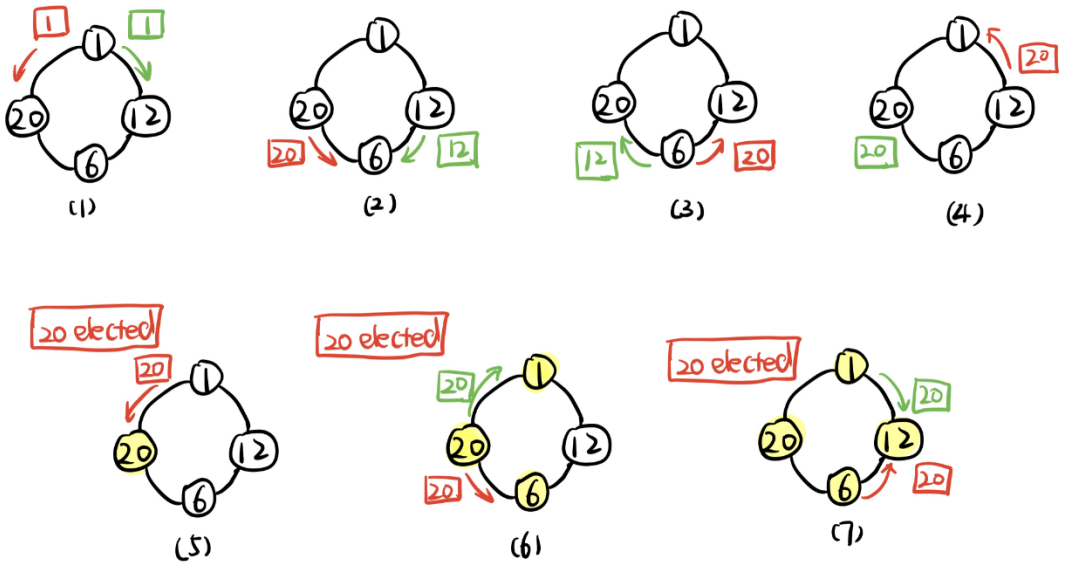
1) NA:=1, NB:=-1, Mi:=0, 1<=i <=N
2) 当进程 Pi 收到权标 A 时（收到权标 B 时同理）：
   if Mi == NA:
   then
   //权标 B 丢失：权标在没有改变 NA 的情况下绕着环赚了一周，也同时表示权标 B 在这段时间内没有访问过 Pi
   重构权标 B；
   else
   // 权标无丢失
   Mi:=NA;
3) 当两个权标相遇时：
   NA:=NA+1;
   NB:=NB - 1;
4) 当进程 Pi 重构权标 B 时（重构权标 A 时同理）：
   // 此时进程 Pi 持有两个权标
   NA:=NA+1;
   NB:= - NB ;

7. 基于环的选举算法是建立在单向环的假设之上的，为了获得更快的选举速度，现采用双向环结构，即每个节点可以同时向顺时针和逆时针两个方向发送选举消息，请列出新算法的高层描述，并用一个四节点的双向环来说明你的方法。

答：

1) 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。
2) 开始选举的进程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，并把消息分别发送到它的顺时针邻居和逆时针邻居
3) 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自己的标识符：
   i. 如果到达的标识符较小，且它是非参加者，则标记为参加者，把消息里的标识符替换为自己的，并转发消息给它一个方向的邻居，这个方向按照它收到消息的方向来确定
   ii. 如果到达的标识符较小，且它已经是参加者，则不转发消息，标识符替换为自己的
   iii. 如果到达的标识符较大，它把消息转发到它一个方向的邻居，这个方向按照它收到消息的方向来确定
   iv. 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。协调者向它的顺时针邻居和逆时针邻居发送当选消息，宣布它的当选，并将它的标识符放入消息中
4) 当进程收到一个当选消息时，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。如果它的邻居已经是非参加者，就不再转发。

四节点的双向环的例子如下图：



8. 节点之间按照生成树方式连接，仅有边相连的节点能通信，请基于此网络拓扑，设计一个选举算法，给出其伪码。当仅有一个进程发起选举，你的选举算法所需的消息量是多少？

答：

要求生成树中至少所有的叶子节点是算法的初始进程。

1) 以发起选举的进程为根节点，向邻居节点传播，直到唤醒所有的叶子进程；
2) 每一个叶子进程被唤醒后，向父节点发送选举消息，包含自己的标识符；
3) 节点收到子节点的标识符后，将自己标记为参加者，将收到的标识符与自己的标识符进行比较，将最大的标识符继续传递给自己的父节点；
4) 直到根节点收到子节点的标识符，将自己标记为非参加者，此时选出最大的标识符即为协调者。根节点向下发送当选消息。
5) 每个收到当选消息的节点将自己标记为非参加者，并将 elected 置为协调者的标识符，并将消息转发给子节点，直到叶子节点收到消息。

伪代码如下：

```
On initialization:
   state := NOT_PAR;
To start election:
      spread out wake-up message to all leaves;

对于非参加者的节点：
if (node.state == NOT_PAR){
      if (it is a leaf) {
         node.state:= PAR;
         send my ProcessId to its parent;
      }
      else{
         while(it has not receive message from all children);
```

```
            candidate := max(ProcessId of all children, my ProcessId);
            if(node is not root) {
               node.state := PAR;
               send message to parent;
            }else{
               ELECTION := candidate;
               send message to all children;
            }
         }
      }
      对于非参加者的节点：
   if (node.state==PAR){
      receive message from parent;
      node.state := NOT_PAR;
      ELECTION := ProcessId in message;
      If (node is not leaf){
         send elected message to all children;
      }
   }
```

假设生成树共有 N 个结点。选举算法中，从某一个进程开始唤醒叶子节点需要发送 N-1 个消息。从叶子节点进行选举到根节点，根节点再将当选消息发送回每个节点共需要 2(N-1) 个消息，所以总共需要消息量为 3(N-1)。

9. 一个服务器管理对象 a1, a2, ... an ，它为客户提供下面两种操作：read (i)返回对象 ai 的值。write(i, Value)将对象 ai 设置为值 Value。

事务 T 和 U 定义如下：

***T*: *x* = *read*(*j*); *y* = *read* (*i*); *write*(*j*, 44); *write*(*i*, 33)**

***U*: *x* = *read*(*k*); *write*(*i*, 55); *y* = *read* (*j*); *write*(*k*, 66)**

请给出事务 T 和 U 的 3 个串行化等价的交错执行。

答：

事务 T 和 U 的冲突操作有：

T: read(i) 和 U: write(i, 55)

T: write(i, 33) 和 U: write(i, 55)

T: write(j, 44) 和 U: read(j)

串行等价性要求事务 T 在事务 U 之前对 i、j 执行冲突访问，或者事务 T 在事务 U 之后对 i、j 执行冲突访问。下面是三种串行等价的交错执行：

事务 T	事务 U
	x = *read*(*k*)
x = *read*(*j*)	
y = *read* (*i*)	
write(*j*, 44)	
write(*i*, 33)	
	write(*i*, 55)

	y = read (j) *write*(*k*, 66)

事务 T	事务 U
x = read(*j*)	
	x = read(*k*)
	write(*i*, 55)
y = read (i)	
	y = read (j)
	write(*k*, 66)
write(*j*, 44)	
write(*i*, 33)	

事务 T	事务 U
x = read(*j*)	
	x = read(*k*)
	write(*i*, 55)
	y = read (j)
y = read (i)	
	write(*k*, 66)
write(*j*, 44)	
write(*i*, 33)	

事务 T	事务 U
	x = read(*k*)
x = read(*j*)	
y = read (i)	
write(*j*, 44)	
write(*i*, 33)	
	write(*i*, 55)
	y = read (j)
	write(*k*, 66)

10. 考虑将乐观并发控制应用于下列事务 T 和 U 的情况：

***T*: *x* = *read*(*i*); *write*(*j*, 44);**

***U*: *write*(*i*, 55); *write*(*j*, 66);**

如果事务 T 和 U 同时处于活动状态，试描述以下几种情况的结果如何：

1. **服务器首先处理 T 的提交请求，使用向后验证方式。**
2. **服务器首先处理 U 的提交请求，使用向后验证方式。**
3. **服务器首先处理 T 的提交请求，使用向前验证方式。**

4. 服务器首先处理 U 的提交请求，使用向前验证方式。

对于上面的每种情况，描述事务 T 和 U 的操作顺序，注意写操作在验证通过之后才真正起作用。

答：

1) 服务器首先处理 T 的提交请求，需要验证 U 的读集合与 T 的写集合是否有交集。由于 U 没有读操作，因此总能通过验证。操作顺序为先执行事务 T 再执行事务 U。
2) 服务器首先处理 U 的提交请求，需要验证 T 的读集合与 U 的写集合是否有交集，T：$x = read(i)$与 $U: write(i, 55)$产生交集，因此放弃事务 T，最终结果是只执行事务 U。
3) 服务器首先处理 T 的提交请求，T 的写集合需要与活动事务 U 的读集合进行比较。由于 U 没有读操作，所以一定会验证成功。最终结果是先执行事务 T 再执行事务 U。
4) 服务器首先处理 U 的提交请求，U 的写集合需要与活动事务 T 的读集合进行比较。$U: write(i, 55)$ 与 T：$x = read(i)$产生冲突，因此放弃其中一个。如果放弃事务 U，则只执行事务 T；如果放弃事务 T，则只执行事务 U。