China-pub.com
下载

第4章 互斥和选举算法

互斥是分布式系统设计的关键问题。互斥保证了相互冲突的并发进程可以共享资源。我们讨论三个和互斥相关的问题：选举、投标和自稳定。

4.1 互斥

互斥问题，或者说定义基本的操作来解决共享资源的多个并发进程的冲突问题已经成为与分布式系统相关的主要困难。

互斥问题最初是在集中式计算机系统中为独占控制之间的同步而考虑的。Dekker[13]提出了这个问题的第一个软件解决方案。Dijkstra对Dekker的解决方案做了详尽的介绍并引入了信号量。硬件解决方案通常基于特殊的原子指令如：Test-and-Set、Compare-and-Swap（在IBM 370系列中），和Fetch-and-Add[24]。对互斥问题的集中式解决方案的详细讨论见[3、31、38]。

分布式互斥的一个简单解决方案使用了一个协调者。每个要求进入临界区的进程向协调者发出请求，协调者对所有的请求进行排队并根据一定的规则如根据它们的时戳授予许可。显然，这个协调者是个瓶颈。分布式互斥问题的一般解决方案分为非基于令牌的和基于令牌的。

在基于令牌的算法中，一个唯一的令牌在不同的进程间共享。当一个进程拥有令牌时，它就可以进入临界区。所以，互斥是自动得以保证的。在非基于令牌的算法中，为了得到对临界区的访问权，进程间需要一个或几个连续回合的消息交换。一个互斥算法，当它的行为独立于系统的状态时它是静态的；当它的行为取决于系统状态时它是动态的。

互斥的主要目标是保证在一个时刻只能有一个进程访问临界区。一些扩展互斥问题也已被研究。k—排斥问题([1、37])是传统互斥问题的扩展，它保证系统不会进入有多于k个进程在执行临界区的状态。一个正常的互斥算法必须满足：

- 不死锁。当临界区可用时，进程不应该无限等待而没有一个进程可以进入。
- 无饥饿现象。每个对临界区的请求最后都必须得到满足。
- 公平性。对临界区的请求必须基于一定的公平原则予以批准，通常是基于由逻辑时钟确定的请求时间。饥饿现象和公平性是相互关联的，但后者所要求的条件更强。

互斥算法的性能由以下参数衡量：

1) 每个请求的消息数。

2) 同步延迟，以一个进程离开临界区到下一个进程进入该临界区之间的时间来衡量。

3) 反应时间，以一个进程发出请求到该进程离开该临界区之间的间隔来衡量。

注意，上面3个参数中，第1和第2个参数用于衡量一个给定的互斥算法的性能。第3个参数更多地取决于系统的负载和/或每个临界区执行时间的长短。系统可能是重负载的（有很多未决的请求）或轻负载的（没有很多未决的请求）。在很多情况下一个算法的性能取决于系统的

China-pub.com 下载

负载。

以下关于系统的假设针对分布式互斥问题。进程间通过消息传递进行通信。网络在逻辑上是全连接的。传输是无错的。缺省的通信方式是异步的，也就是说，消息传输延迟是有限的但不可预测。除非特别说明，否则消息可以按照发送的顺序递交。

4.2 非基于令牌的解决方案

在非基于令牌的算法中，所有进程相互通信来决定哪个进程可以执行临界区。大多数这一类算法是真正分布式的，也就是说，决策是以分散的方式做出的。

4.2.1 Lamport算法的简单扩展

在分布式系统中，使用消息交换的控制机制用于分散式的资源管理和同步。利用时戳，上一章讨论的Lamport算法需要$3(n-1)$个消息来保证n个进程集的互斥。在图4-1的例子中，有三个进程P_1、P_2和P_3，其中，P_1和P_2请求临界区（假设P_1在P_2之前请求）。图中显示了三种类型的消息：实线代表请求信号，虚线代表应答信号，点线代表释放信号。横杆线对应于处于临界区中的进程。在Lamport算法中，当一个进程收到请求资源的消息时，它把该请求放在自己的本地请求队列中并发回一个带时戳的应答。一个进程只有收到所有进程的应答后才能进入临界区。要释放资源，P_i发送一个带时戳的释放资源的消息给所有的进程。

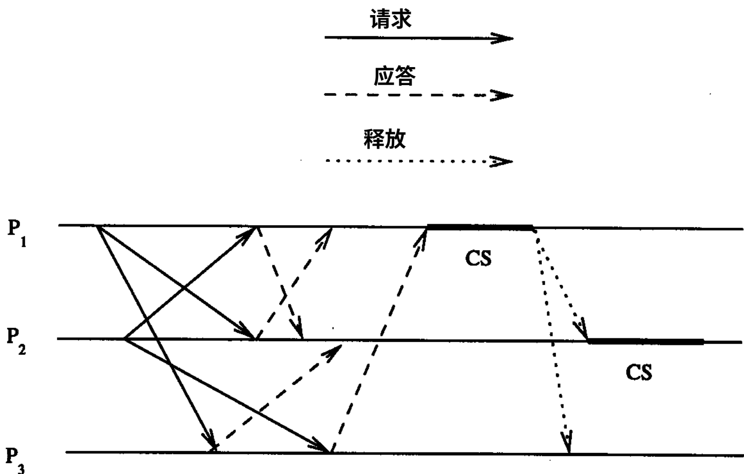

图4-1 使用Lamport算法的例子

我们可以如下轻松改进Lamport算法：假设进程P_j在发出它自己的请求后收到一个来自进程P_i的请求，而它的请求的时戳大于P_i的请求的时戳。在这种情况下它没有必要发送应答，这是因为当进程P_i收到P_j的请求时，它发现P_j的请求的时戳大于自己的请求的时戳，于是它就知道P_j没有任何时戳更小的未决请求。注意，在Lamport算法中使用的应答不是出于可靠性的目的。然而，以上改进在最坏情况下仍需要$3(n-1)$个消息。

图4-2表示应用改进型方法于图4-1中的例子的结果。由于P_2在发出自己的请求后收到来自P_1

China-pub.com
下载

的请求，该请求的时戳小于P_2发出的请求的时戳，所以P_2无需再发送应答。

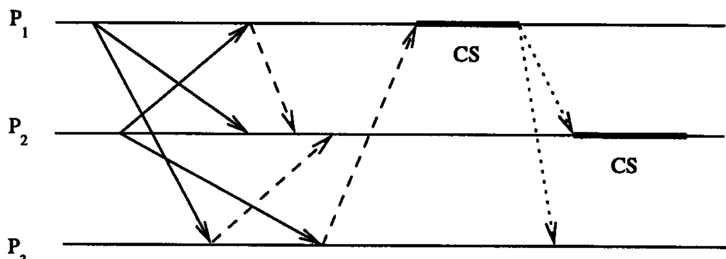

图4-2 使用改进型Lamport算法的例子

4.2.2 Ricart和Agrawala的第一个算法

Ricart和Agrawala算法[40]需要$2(n-1)$个消息交换：$n-1$个消息用于进程P_i通知所有其他进程它对临界区的请求，另外$n-1$个消息用于传达同意信息（agreement），也就是说，它把应答（acknowledge）消息和释放（realease）消息合并到一个消息中，即回答（reply）消息。回答消息是一种许可。一个进程只有收到所有其他进程的许可后才能访问临界区。与Lamport的方法相比，该算法不需要对回答消息进行任何逻辑操作来判定它的合格性（eligibility）。算法如下：当进程P_j收到来自P_i的请求时，如果P_j既不请求临界区也不执行临界区，或者它正请求临界区而且它的请求的时戳大于P_i的请求的时戳，那么进程P_j给P_i发回一个回答消息。释放临界区时，进程发送一个回答消息给所有被延迟的请求。基本上，进程会保存它的所有低优先级的请求，即使该进程不能收到所有的回答消息，也就是说，它不能访问临界区。

图4-3表示将Ricart和Agrawala算法应用于图4-1中的例子的结果。当进程P_1收到来自P_2的请求时，它将保留应答消息因为P_1有最高的优先级。进程P_3没有任何请求，所以它发送应答消息给两个请求进程。

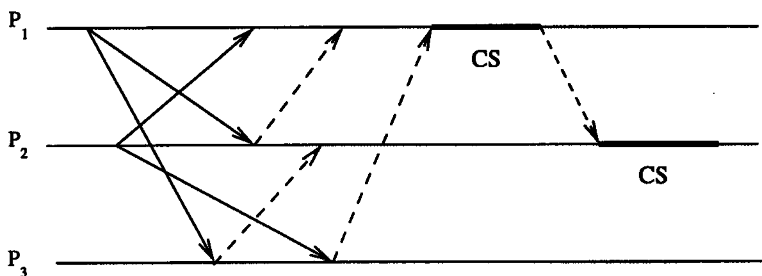

图4-3 使用Ricart和Agrawala算法的例子

在Ricart和Agrawala的算法中，一个进程可以同时给许多进程授予许可。如果一个进程没有正在使用临界区或者它自己的请求的优先级更低时，它给请求进程授予许可。也就是说，决策是由请求进程做出的，但接收请求的进程可以通过发送或保留它的许可信号对决策进行一定程度的控制。一旦一个进程从所有其他进程处得到许可信号，它就可以进入临界区。注意，一个进程可能发送许多许可信号给不同的请求进程。显然，每次只有一个进程可以收集到所有的许

China-pub.com 下载

可信号。

为了让接收请求的进程对决策过程有更多的控制权，进程可以基于优先级原则一次只给一个请求进程授予许可。一个请求进程只能在收到所有其他进程的许可信号后才能访问临界区。对于从所有其他进程获得的许可信号，条件也可放松：一个请求进程可以在收到预先定义的进程子集（这样的子集称做请求子集）的许可信号后访问临界区。为了避免两个进程同时进入临界区，任何两个进程的请求子集的交集必须非空。一个（属于两个请求子集的）的公共进程担当仲裁者。由于这个进程只有一个许可信号，所以它只能给予一个请求者。下一小节讨论的Maekawa的算法就是这样的一个例子。

4.2.3 Maekawa的算法

在Maekawa的算法[30]中，进程不用请求每一个其他进程的许可，而只需请求一个进程子集的许可。假设R_i和R_j分别是进程P_i和P_j的请求子集，要求$R_i \cap R_j \neq \phi$。由于每个进程授予一个许可信号给一个请求进程，所以互斥自动得以执行。当进程 P_i请求临界区时，它只给R_i中的进程发送请求消息。当进程P_j收到一个请求消息时，如果它自从上一次释放临界区后还没有发出过回答消息给任何进程，它就发出一个回答消息。否则，请求消息被放入队列中。注意，批准请求的手续不是基于时戳，而是基于每个请求的到达时间，也就是说，基于时戳的强公平性在这里不被执行。然而，如果每个通道的通信延迟是有限的，就不会发生饥饿现象。进程 P_i只有在收到R_i中的所有进程的回答消息后才能访问临界区。在释放临界区时， P_i只给R_i中的进程发送释放消息。

考虑一个七个进程的例子，每个进程的请求集如下：

$$R_1: \{P_1, P_3, P_4\}$$
$$R_2: \{ P_2, P_4, P_5\}$$
$$R_3: \{ P_3, P_5, P_6\}$$
$$R_4: \{ P_4, P_6, P_7\}$$
$$R_5: \{ P_5, P_7, P_1\}$$
$$R_6: \{ P_6, P_1, P_2\}$$
$$R_7: \{ P_7, P_2, P_3\}$$

在以上请求子集中，任何两个请求子集恰有一个公共的进程。每个进程只能发出一个回答消息从而使互斥得以保证。如果请求子集的大小为 k并且每个请求子集都一样，进程数为 n，则n的最小值为$n=k(k-1)+1$。显然，对每次访问，发出的请求消息的个数是 $k=O(\sqrt{n})$。

让我们考虑另外两种R_i的选择。在集中式互斥中，P_c（c $\{1，2，…，n\}$）作为唯一的仲裁者，我们有：

$$R_i:\{P_c\}，1\leq i\leq n$$

另一种极端是完全分布式的互斥算法，其中请求进程向所有其他进程请求许可：

$$R_i:\{P_1，P_2，…，P_n\}，1\leq i\leq n$$

Maekawa的算法容易死锁，因为进程可能被其他进程锁住。例如，考虑七个进程例子的第一个解法，其中三个进程 P_1、P_6和P_7同时请求临界区。由于不同的通信延迟， P_3可能给P_1回答，

China-pub.com
下载

P_4给P_6、P_2给P_7。三个进程中没有一个可以同时得到三个回答消息，于是死锁发生了。

死锁问题的一种解决方法是要求一个进程放弃对锁的申请，如果它的请求的时戳比其他等待同一个锁的请求的时戳大。另一种保守的方法保证进程只有在没有更小时戳的其他请求时才对请求进程给予回答，它类似于Lamport的方法。然而，这种情况需要相当数量的消息交换。

我们还可以通过引入通知集（inform set）进一步推广Maekawa的算法，通知集包括一系列进程，释放信号将被发送给这些进程。注意，在 Maekawa算法中，请求集和通知集是相同的。这个推广算法的详细讨论见[43]。

4.3 基于令牌的解决方案

在基于令牌的算法中引入了令牌的概念。令牌代表了一个控制点，它在所有的进程间传递。一个进程拥有令牌时就可以进入临界区。

4.3.1 Ricart和Agrawala的第二个算法

作为对Ricart和Agrawala算法的改进，Carvalho和Roucairol[11]提出了另一个算法。通过引入令牌即动态的单一点控制，使用对称的不同定义，这个算法要求的消息数在 0到$2(n-1)$之间。在这个算法中，进程P_i收到来自P_j的回答消息后可以进入临界区（假设它得到所有其他进程的回答消息）任意次而无需再请求P_j的许可。因为当进程P_i收到P_j的回答消息时，隐含在该消息中的授权保持有效直到P_i收到来自P_j的请求消息。

Ricart和Agrawala[41]提出了进一步改进如下（见图 4-4）：进入临界区的进程保留令牌。初始时，令牌被赋予任意一个进程。希望使用临界区的进程 P_j不知道哪个进程拥有令牌，所以它通过向所有其他进程广播一个带时戳的消息来请求令牌。如果当前拥有令牌的进程 P_i不再需要使用临界区，它就按照$i+1, i+2, \cdots, n, 1, 2, \cdots, i-1$的顺序搜索其他进程，找出第一个$j$，满足$P_j$最后一次请求令牌的时戳大于在令牌中记录的$P_j$最后一次拥有令牌的时戳，也就是说，$P_j$有一个未决的请求。于是，$P_i$把令牌传递给$P_j$。注意，优先级不是严格基于每个请求的时戳的，但是，由于令牌是沿着一个方向环绕传递的，所以不会有饥饿现象发生。

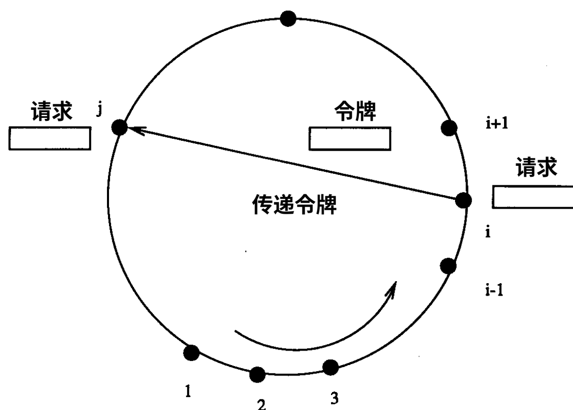

图4-4 Carvalho和Roucairol的算法

China-pub.com 下载

Ricart和Agrawala的第二个算法：

P(*i*)::＝*[请求资源
□ 消费
□ 释放资源
□ 处理-请求-消息
□ 其他
]

分布式－互斥（distributed-mutual-exclusion）::＝|| *P* (*i*:1.. *n*)

以下变量用于每个P_i中：

clock：0，1，…，（初始化为0）
token_present：**Boolean**（除了一个进程，对所有其他进程均为 *F*）
token_held：**Boolean**（*F*）
token：**array**（1..*n*）**of** *clock*（初始化为0）
request：**array**（1..*n*）**of** *clock*（初始化为0）

注意，每个进程有一个局部 *clock*（根据Lamport的规则进行更新）、*token_present*和*token_held*的拷贝，但系统中只有一个*token*数组的拷贝（见图4-4）。每个P_i中的函数定义如下：

其他::=所有其他不请求进入临界区的动作
消费::=进入临界区后消费资源
请求资源::=

```
[ token_present=T
   →[ send (request_signal, clock, i) to all;
     receive (access_signal, token);
     token_present:=T;
     token_held:=T
    ]
]
```

释放资源::=

```
[ token (i):=clock;
  token_held:=F;
  min j in the order [i+1,…, n, 1, 2,…, i+2, i−1]
     ∧ ( request (j) > token (j) )
     → [  token_present:=F;
          send ( access_signal, token ) to P_j
       ]
]
```

处理-请求-消息::=

China-pub.com
下载

[**receive** (*request_signal*, ***k***, *j*)
→ [*request* (*j*):=max (*request* (*j*), ***k***);
token_present ∧¬ *token_held*→ **释放资源**
]
]

当请求进程没有持有令牌时，以上算法需要 n个消息（$n-1$个用于广播请求，1个用于传送令牌）；当请求进程持有令牌时，以上算法需要 0个消息。注意，在以上算法中，请求资源可能在*token_present*:=T之后而在*token_held:* =T之前被处理-请求-消息所中断。这种情况下，被中断的进程将不得不释放刚刚收到的令牌。解决这个问题的一个方法是让这两个语句合并为一个原子语句，也就是说，这两个语句作为一个语句来对待。另一种解决方法是把 *token_held*:=T放在*token_present*:=T之前。

容易证明以上算法保证了互斥，因为所有进程中只有一个 *token_present*为真。它的公平性基于令牌的绕环传递。通过假设令牌的持有和传送时间是有限的（而且没有令牌的丢失），那么所有提出请求的进程都将最终得到令牌并进入临界区。注意，如果优先级是基于每个请求的时戳，则更强的公平性得以保证。然而，决策过程就会和 Lamport算法中的决策过程一样相当复杂。一种折衷是基于当前请求队列中的所有请求的时戳做出决策，可能会有时戳更小的请求仍在传送中。

4.3.2 一个简单的基于令牌环的算法

如果进程P_i，0 ($i \le n-1$，连接成一个环，这个环可以基于底层网络，那么存在一个简单的互斥算法[39]。同样，使用令牌代表一个动态的单一控制点（见图 4-5a)，该令牌绕环传递。

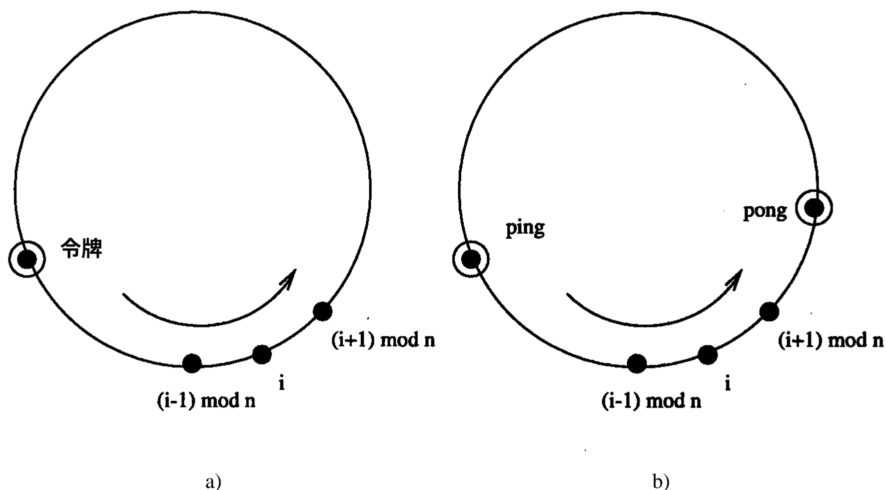

图4-5 a）基于令牌环的简单算法，b）基于令牌环的容错算法

P（i : 0 .. $n-1$） ::=[**receive** *token* **from** P （($i-1$） mod n） ；

China-pub.com 下载

如果需要消费资源;
send *token* **to** P （(i＋1） mod n）
]
分布式-互斥::＝|| P（i：0..n－1）

以上算法极为简单易行。这个算法在高负荷的情况下工作地很好，这种情况下持有令牌的进程需要进入临界区的概率很高。然而，它在轻负荷的情况下工作地很差，因为在轻负荷的情况下只有很少的进程请求进入临界区。这个算法在传递令牌时浪费了 CPU的时间。

4.3.3 一个基于令牌环的容错算法

动态单一控制点算法（dynamic single contrl point algorithm）有可能丢失控制点（令牌）。一些容错算法[28]利用了超时机制。[32]中的算法不需要有关消息传播延迟或进程身份的知识。该算法使用两个令牌（ping和pong）（见图4-5b），其中每一个令牌负责检测另一个令牌可能的丢失。令牌的丢失将被某个进程检测到，如果这个进程被同一个令牌连续两次访问。每个令牌都和一个数相关联（*nbping*，*nbpong*），这两个数满足以下条件：

$$nbping + nbpong = 0$$

在基于令牌环的容错算法中，每个进程的控制过程如下：

P (i: 0 .. n－1) ::=
  [**receive** (*ping*, *nbping*) **from** P ((i－1) mod n)
    →[m_i＝*nbping*
      →[*nbping* :＝(*nbping*＋1) mod n;
        nbpong :＝－*nbping*;
        send (*ping*, *nbping*) **to** P ((i＋1)mod n)
        send (*pong*, *nbpong*) **to** P ((i＋1)mod n)
      ]
      /* 令牌*pong*丢失*/
    □ m_i ≠ *nbping*
      →[m_i :＝*nbping*;
        send (*ping*, *nbping*) **to** P ((i＋1)mod n)
      ]
      /* 正常状态*/
    ]
  □ **receive** (*pong*, *nbpong*) **from** P ((n－1)mod n)
    →(同上面程序，只需交换*ping*和*pong*)
  □ meet (*ping*, *pong*) at P_i
    →[*nbping* :＝(*nbping*＋1) mod n;
      nbpong :＝(*nbpong*－1) mod n;

China-pub.com 下载


```
      send (ping, nbping) to P ((i+1)mod n)
      send (pong, nbpong) to P ((i+1)mod n)
    ]
    /* ping和pong相碰*/
]
```

以下主程序初始化所有参数并调用所有的进程 Pi。

```
main-program ::=[ nbping :=1; nbpong :=−1;
                  m_i :=0，0≤i≤n−1；
                  || P( i: 0 .. n−1 )
                ]
```

变量m_i用于保存最近刚访问过进程P_i的令牌的相关数字。所以，如果m_i的值和当前令牌的相关数字匹配，则意味着令牌的丢失。由于每个令牌在一次循环中被更新的次数不会多于n次，所以对令牌进行模n加1是足够的。

4.3.4 基于令牌的使用其他逻辑结构的互斥

对基于令牌的互斥算法，除了环形结构，还经常用到图结构和树结构。在基于图结构的算法[34]中，进程排列为一个有向图，其中的一个接收节点（sink node）持有令牌。对令牌的请求和令牌的传递按照和如下介绍的基于树的算法类似的方式处理。

在基于树的算法[36]中，进程被排为一棵有向树，根节点持有令牌。对令牌的请求沿着从请求者到根节点的路径向上传递。令牌从根节点传向请求者并经过从根节点到请求者的路径上的边。与此同时，沿该路径的边的方向被反过来，于是请求者成了新的根节点。图 4-6a表示一个从叶节点到根节点的请求。图4-6b表示根节点把令牌传给叶节点（新的根节点）后的相应结果。如图 4-6b所示，新的树中的每个节点仍然可以沿着一条有向路径把它的请求发给持有令牌的节点。

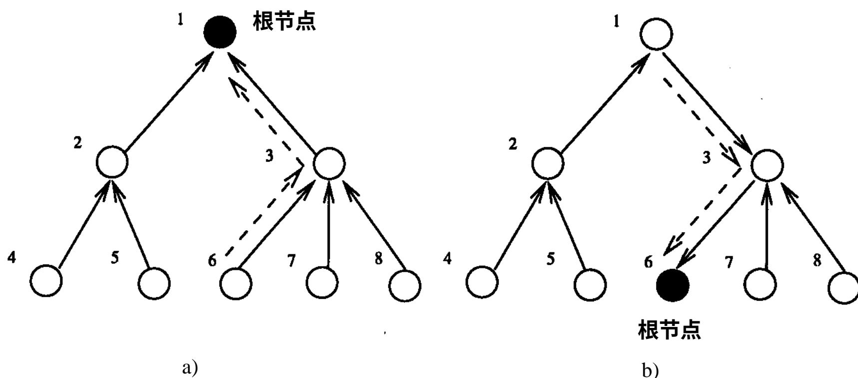

图4-6 基于树的互斥算法

一般的基于令牌和基于树的互斥算法见[22]。有关分布式互斥的综述见[46]。

其他相关问题

China-pub.com 下载

大部分讨论的算法都尽量减小消息复杂性。有些研究人员认为在同步延迟方面时间复杂性也是很重要的。例如，Lamport的算法需要3（$n-1$）个消息和两个时间单位的延迟。Maekawa的算法只需要$O(\sqrt{n})$个消息和两个时间单位的延迟来进入临界区，但它在发送回答消息给某个请求进程之前还需要一个时间单位的延迟释放被锁住的节点。Chang[10]研究了同时将时间复杂性和消息复杂性减到最小的问题。使用的方法是一种结合几种算法的混合式分布式互斥算法。Goscinski[19]提出了两个适合于要求优先级的环境（如实时系统）的互斥算法。算法以及严格的证明见[7、23]。

分布式互斥还可以通过投标（bidding）[8、47]，选举（election）[17]和自稳定（self-stabilization）[14]来实现，这些将在接下来的章节中讨论。

4.4 选举

选举（election）[17]是分布式系统中的一种常用的计算类型，它从进程集中选出一个进程执行特别的任务。例如，在分布式系统出现故障后，通常需要重新组织活动的节点使它们继续执行有用的任务。在这个重新组织和配置的过程中，第一步就是要选出一个协调者来管理这些操作。故障的检测通常是基于超时机制的。如果一个进程超过一定的时间没有收到协调者的响应，它就怀疑协调者出了故障并启动选举过程。

选举可以在以下领域得到应用[25]：群服务器（group server）、负载平衡、重复数据更新、应急恢复、连接组（joining group）和互斥。

基于所使用的拓扑类型：完全图、环和树，人们提出了不同的分布式选举算法。这些算法还可以这样进行分类，即基于所使用的网络类型：（a）存储－转发网络，包括单向环[9、15、35]、双向环[16]、完全图[6、26]、树[12、36]和弦环[33]，（b）广播网络[25]。

一般一个选举过程需要两个阶段：（a）选择一个具有最高优先级的领导者，（b）通知其他进程谁是领导者（优胜者）。两个阶段都需要在系统中发布进程id，这可以通过点到点通信或广播来实现。在全连接网络中可以轻易实现广播，否则，就静态或动态构造一个生成子图。例如，如果使用单向环作为生成子图，那么两个阶段中消息都在环中传递。另一个常用的生成子图是树，它可以静态选择或动态创建。

如果生成树是预先知道的，那么所有的叶节点启动选举过程并把它们的id向上传递给它们的祖先，沿着每条到根节点的路径只有最高优先级的id可以保留下来。当最高优先级的id到达根节点时，第一个阶段结束。接下来第二个阶段开始，它沿着生成树从根节点向所有的叶节点广播优胜者的id。如果生成树是动态生成的，那么由一个进程（根节点）启动生成一个生成树的过程。这个过程终止在每个叶节点上，然后每个叶节点按照静态生成树的情况启动选举过程。

大部分选举算法是基于全局优先级的，其中，每个进程（处理机）预先分配一个优先级。这些算法也称做extrema-finding 算法。对extrema-finding 算法的主要反对意见是一旦选中一个领导者时，就必须保证它是一个正确的选择，比如，从性能或可靠性的观点来看。最近，在基于优先选择的算法[45]方面做了更多的工作，其中的选举更一般地基于个人的优先选择，如，局部性、可靠性估计等等。同样，优先级也可以不是线性顺序的，就像在基于优先选择的算法中那样。

我们这里考虑三种类型的 extrema-finding算法。Chang和Robert的算法是基于环形拓扑的，

China-pub.com
下载

其中每个进程不知道其他进程的优先级。 Garcia-Molina的bully算法[17]是基于全连接拓扑的，其中每个进程知道其他进程的优先级。这两个算法都是基于比较的，通过比较所有进程的 id和发送接收消息中的id选出一个领导者。第三种类型是非基于比较的，其中消息被“编码”在以回合（round）表示的时间中（这种类型的算法只能工作在同步系统中）。

选举和互斥之间的相似性是相当明显的。使用选举，所有希望进入临界区的进程将通过选举竞争决定哪个进程可以进入临界区。不同之处在于互斥算法必须保证不发生饥饿现象而选举算法更注重于快速的选举过程。同样，必须通知参与进程谁是选举的优胜者。在互斥算法中参与进程不关心当前谁在临界区中。

4.4.1　Chang和Roberts的算法

以下是Chang和Roberts[9]提出的针对互连成单向环的进程的选举算法，它使用选择性停止（selective extinction）的原则，只需要O（n log n）个消息，其中n是进程数。它是LeLann[28]提出的算法的推广。

在该算法中，n个进程按任意顺序排列在一个环上。我们假设所有进程的 id各不相同，最小的id优先级最高。任何时候进程 P_i收到一个选举消息（一个进程的 id），它把这个消息和自己的id(i)相比较并把小的那个 id传给它的顺时针方向或逆时针方向的邻居。当 P_i收到它自己的id时，它就成了协调者并通过发送被选中进程的 id通知所有其他进程。

每个进程P_i中的参数：

id (*i*) ：**cons**

participant：**var Boolean**（**F**）；

coordinator：**var integer**；

Chang和Roberts的算法：

```
P ( i : 0.. n−1) ::=
    ( [  initiate election →
        [  participant :=T;
           send ( election, i, id ( i ) )  to  P ( (i+1) mod n )
        ]
     □ ( receive ( election, j, id ( j)) →
        [   id ( j ) < id ( i ) →
                 [  send ( election, j, id ( j ) )  to  P ( (j+1) mod n )
                 ||  participant :=T
                 ]
          □ id ( j ) > id ( i )∧participant →Φ
          □ id ( j ) > id ( i ) ∧¬ participant →
                     [  send ( election, i, id (i) )  to  P ((i+1) mod n )
                     ||  participant :=T
                     ]
```

China-pub.com 下载

```
    □ id ( j )＝id ( i ) →
                send ( elected, i ) to P ( (i＋1) mod n)
        ]
□ receove ( elected, j ) →
        [ coordinate :＝j
        || participant :＝F
        || j≠ i → send ( elected, j ) to P ( (j＋1) mod n )
        ]
]
election-algorithm ::＝|| P ( i : 0 .. n－1)
```

显然，该算法选择一个而且仅选择一个 id ——所有进程中最小的*id*。选出一个协调者所需要的时间是绕环一周所需时间的二到三倍。显然，类型为选中的消息的传送次数为 n。为了计算类型为选举的消息的传送次数，我们首先考虑两种极端的情况。一种情况是选举是由一个有最大优先级的进程启动的，这时选举消息正好绕环一周，终止在发起方。传送的消息数是 O（n）。另一种（最为不利的）情况是进程按降序排列而且每个进程同时启动选举过程。这样，P_i发出的消息经历i＋1次传送，所以总的消息数是

$$\sum_{i=0}^{n-1} i+1 = \frac{1}{2}n\ (n+1)$$

或O（n_2）。

一般，设P（i，k）为消息i被传输k次的概率（或者说P_i的k－1个邻居的id小于i，而第k个邻居的id大于i）。显然，

$$P(i,\ k) = \frac{\binom{i-1}{k-1} \times (n-i)}{\binom{n-1}{k-1} \times (n-k)}$$

携带数字n的消息被传输n次，消息i的平均传送次数为

$$E_i = \sum_{k=1}^{n-1} kP(i,\ k)$$

所以，平均总的传送次数是

$$E = n + \sum_{i=1}^{n-1} Ei$$

可以表示为O（n log n）[9]。注意，以上复杂性是在平均情况下而不是在最坏情况下。

Franklin[16]提出了一种在最坏情况下复杂性为 O（n log n）的算法，但该算法假设环形网络是双向的。初始时每个进程都是活动的，它把自己的 id和两边方向上的相邻活动进程的 id相比。如果它是三个id中最小的，它就保持活动状态；否则，它变为被动的。在每轮比较后只有一半的参与进程可以保留下来。所以，一共只需要 log n轮的比较。注意，当一个进程变为被动时，它不参与下一轮的比较但它仍会转发进来的 id。在[15]和[35]中这个算法已经被扩展到单向环网络，

China-pub.com
下载

但具有和双向环网络相同的消息复杂性。

对于基于比较的算法，$O(n \log n)$是一个下限。我们以后将说明可以通过非基于比较的算法来改进这个下限。

Garcia-Molina的bully算法

对于每个进程知道所有其他进程的优先级的系统而言，bully算法[17]是一个可靠的选举算法。同样，具有最高优先级的进程被选中。我们假设进程可能随时发生故障并恢复。如果一个进程恢复时发现没有其他活动进程的优先级比它高，它将强迫所有优先级比它低的进程让它成为协调者。

任何时候如果进程P通过超时机制检测到协调者出了故障，它给所有优先级比它高的进程广播选举消息并等待其中任何一个进程的回答。如果在规定时间内没有收到任何响应，就认为所有这些进程都出了故障，于是P宣布它自己为新的协调者并把它的id广播给所有优先级比它低的进程。如果在规定时间内收到了一个响应，进程P就设置另一个计时器等待接收新的协调者的id。这时进程P要么在新的规定时间内收到新的协调者的id，要么没收到任何消息于是P重新发出选举消息。

当优先级高于P的进程收到P的请求时，它对P做出响应并通过给所有优先级更高的进程发送一个选举消息启动它自己的选举算法。如果这个进程自己有最高的优先级，它马上就可以宣布自己是优胜者。

一般来说，只有系统出现故障时才会进行选举，所以，故障的检测和处理通常是选举算法的一个集成部分。可以很容易地扩展bully算法，使它在同步系统下包括一个可靠的故障检测器，也就是说，消息传递的时间和节点响应的时间是预先知道的。一个更实际的算法，称为邀请算法（invitation algorithm）[17]，是基于异步模型的，但它比bully算法要复杂得多。

还有很多其他可靠的（或弹性的）互斥和选举算法。一个可靠的互斥算法使用令牌，它检查令牌是否在网络故障中丢失并在需要的时候重新产生一个。通过使用超时机制，三种类型的网络故障、PE故障：通信控制器故障和通信链路故障是可以容许的。在[25]中，King等人提出了一个可靠的选举算法，它可以容许故障—停止和拜占庭两种类型的故障（一种复杂的故障类型，将在第8章讨论）。

4.4.2 非基于比较的算法

在非基于比较的算法中，领导者不是通过和其他进程比较id选出的。它是通过其他方式选出的，而且通常它的id是和用回合（round）表示的时间联系在一起的。所以，这些算法是面向同步系统的，其中，所有的进程根据回合协调它们的行为。在每个回合，进程执行本地更新并和其他进程交换消息。

以下过程用于把一个进程的id编码到时间里，也就是说，接收方可以得到发送方的消息而无需发送方物理上发送该消息：

- 如果发送方P_i的id是$id(i)$，在回合x，P_i发送（$start$，i）给接收方P_j。
- 在回合$x+id(i)$，P_i发送（end，i）给接收方P_j。
- 如果P_j在回合x收到（$start$，i），在回合$x+id(i)$收到（end，i），它就得到P_i的id为[$x+id$

China-pub.com 下载

$(i)]-x=id(i)$。

注意，以上算法虽然降低了消息复杂性尤其是长消息的比特复杂性，但它增加了时间复杂性——接收方P_j花了$id(i)$个回合才得到发送方P_i的一个消息。

Lynch[29]提出了以下两个非基于比较的选举算法，这两个算法都是针对同步系统的：

时间片算法（Time-slice algorithm）：

假设n是总的进程数。进程P_i（它的id为$id(i)$）在回合$id(i)\cdot 2n$发送它的id，也就是说，在每$2n$个连续的回合中最多有一个进程发送它的id。一旦一个id回到它最初的发送方，该发送方就被选中了。于是它在环上发送一个信号通知其他进程它已经是优胜者了。

对于领导者（id最小的进程），在其他进程开始发送id之前通知其他进程它$2n$个回合已足够用来传递它的id并在任何其他进程开始传递它们的id之前通知它们自己获胜的情况。显然，以上算法的消息复杂性是$O(n)$。然而，它的时间复杂性是$\min\{id(i)\}\cdot n$，这是一个很大的数。另一个限制是每个进程必须预先知道进程的数目n。以下算法适用于n为未知的情况。

可变速度算法（Variable-speed algorithm）：

当进程P_i发出它的id（$id(i)$）时，这个id以每$2^{id(i)}$个回合一次传送的速率进行传送。如果一个id回到它最初的发送方，该发送方就被选中了。

通过以下观测可以容易求得消息复杂性：当最小的id循环一次时，次小的id最多只能循环一半。所以，总的消息交换次数最多为：

$$n+\frac{n}{2}+\frac{n}{2^2}+\cdots+\frac{n}{2^{n-1}}<2n$$

也就是$O(n)$，但它的时间复杂性是$2^{\min\{id(i)\}}n$。

4.5 投标

选举过程的一种特殊实现是投标[8]，其中，每个竞争者从一个给定的集合中选择一个投标值，并把它的投标值发送给系统中的所有其他竞争者。每个竞争者都承认同一个优胜者。大多数投标方案本质上都是概率性的。参加这样一个概率性算法的竞争者首先通过抛硬币或通过产生一个随机数从给定的集合中选择一个值。接着它把自己的值和其他竞争者选择的值进行比较。

Chang[8]给出了一个投标方案满足防守竞争者的保护（protection against competitors）的原则。如果进程从集合B中选择投标值，$|B|=n$，那么不管其他进程选择的投标值是多少，它取胜的概率为$1/n$。这个方案基于以下四个假设：

1) 竞争者只能通过交换消息互相通信。
2) 通信是无错的。
3) 每个竞争者都将在最后期限前发出它的投标值。
4) 竞争者不能发送相互冲突的信息给不同的竞争者。

Chang还提出了以下投标方案的变体和扩展。

- 不同加权的投标。
- 多于一个优胜者的投标。
- 竞争者个数未知的投标。

China-pub.com
下载

为了降低以交换的消息总数衡量的通信成本，Wu和Fernandez建议采用一个基于群（gang）[47]的概念的解决方案。一个群是一组一起工作设法控制投标结果的竞争者。系统中可能存在多个群。定义非群子集（gang-free subset）为一组竞争者，其中至少包括一个不属于任何群的竞争者。定义m为最小的整数，满足所有元素个数大于等于m的子集都是非群⊖。我们称那些元素个数为m的子集为最小非群子集（minimum gang-free subsets）（记做MGFS）。在我们的解决方案中，MGFS的大小是预先知道的，而且大小为m的子集被任意选择参与投标。投标的结果在竞争者而不是在选择子集中发布。

以下符号用于投标过程：

n　系统中的竞争者数

i　竞争者的索引

P_i　第i个竞争者

b_i　第i个竞争者选择的投标值，范围从1到n

k　优胜者（竞争者之一）的索引，范围从1到n

m　最小非群子集（MGFS）的大小

f　决策算法

在Chang的方法中：

$$f(b_1, b_2, \cdots, b_n) = P_k$$

并且

$$k = \sum_{i=1}^{n} b_i \bmod n + 1$$

在扩展投标方案中，不失一般性，我们选择一个元素个数为m最小的非群子集，其中竞争者的索引从1到m。容易看出，如果这些在MGFS中的元素有分散的索引，这个方案仍然可行。投标问题的解决方案可以表示如下：

$$f(b_1, b_2, \cdots, b_m) = P_k$$

并且

$$k = \sum_{i=1}^{m} b_i \bmod n + 1$$

显然，优胜者的索引仍然处于1到n之间并取决于所有MGFS中的竞争者选择的投标值。由于Chang的解法只使用一个MGFS，它是所有n个竞争者的集合，所以他的解法可以认为是一种特殊例子。

这两种方案的性能可以用一些特别的网络拓扑来评价。我们这里考虑两种类型的网络：全连接网络和超立方网络。我们假设每个代表一个竞争者的进程被分配到不同的处理机（或节点）。

⊖ 如果我们假设不同群中的成员永远不会对控制结果达成一致意见，那么非群子集可以定义为一组竞争者，其中没有群，或者有多个群因为效果就和没有群一样。在这种情况下，如果M是系统中所有群的大小的最大值，那么任何大小为$m=M+1$的子集是非群。

China-pub.com
下载

在一个全连接网络中，每对节点可以直接交换消息而且每个节点可以同时和所有其他节点交换消息。在这个假设下，Chang的算法需要$n(n-1)$次消息交换和一个交换回合。建议的算法在选举优胜者阶段（阶段 1）需要$m(m-1)$次消息交换，在发布结果阶段（阶段 2）需要$n-m$次消息交换。显然，Chang的算法比建议的算法多需要 $(n-m)(n+m-2)$次消息交换。当 $m=1$[⊖] 时，消息交换次数的差值达到最大值。

全连接网络虽然有一些很好的特性，却很少用在分布式系统中，尤其是大型并行系统。在其他网络结构中，超立方 [42]由于其规则性和对称性已成为占主导地位的拓扑结构之一。在 l维超立方结构中，每个节点分配一个不同的 l比特二进制地址，从 0到2^{l-1}。两个节点通过一条链路直接相连（所以它们是邻居）当且仅当它们的二进制地址只有一位不同。为了减少选举优胜者过程中的消息交换总数，每个节点计算目前为止收集到的所有投标值的部分和。部分和的结果传给相邻的一个节点。

以下算法在一个l维超立方Q_l上实现Chang的方案，其中$2^l=n$。每个竞争者c选择一个投标值作为每个c的初始积聚值（initial accumulated value）（v）。我们用$c(d)$表示c沿着第d维的相邻节点（它的初始积聚值为v^d）。

$P(c)$ ::＝ ∗ [[**send** v **to** $c^{(d)}$ ‖ **receive** v^d **from** $c^{(d)}$];

$v:=(v+v^d) \bmod n$

]

hypercube-bidding ::＝ $_{c\ Ql}$ $P(c)$

在建议的方案中，我们假设 $m=2^s$，$n=2^l$，$s<l$并且地址为$00\cdots 0a_s\cdots al$的竞争者被选为MGFS。算法中两个阶段的实现如下：

1) 优胜者的选择：在选定的MGFS上使用超立方-投标算法，即子立方 $00\ldots 0a_s\ ..\ a_l$。

2) 结果的发布：MGFS中的每个P发送它的积聚值给2^{l-s}个竞争者，这些竞争者和P在地址码上有相同的s位最低有效位（LSB）。

图4-7和图4-8表示应用在一个4维超立方上的上述两种算法，其中，子立方 $0a_3a_2a_1$被选为MGFS。在图4-7和4-8中分别需要64和32次消息交换。两种情况都需要四个消息交换回合，和每条链路相关联的数字指出了该链路上发生的双向消息交换的回合。

在超立方上实现的Chang的算法中，需要$l2^l=n\log n$个消息，是超立方中总链路数的两倍，还需要l次消息交换回合[⊖]。在建议方法的实现中，如图 4-8所示，没参与选举过程的节点用白色节点表示，实现的第一个阶段使用了$m\log m$个消息，第二个阶段使用了$n-m$个消息。总的消息数是$m\log n+(n-m)$，同样也需要l次消息交换回合：其中的s次用于第一阶段，剩下的$l-s$次用于结果的发布。总的来说，当建议的解决方案用于非全连接网络如超立方网络时，消息数减少了但消息交换的回合数保持不变。所以建议的解决方案看来特别适合于非全连接网络。

⊖ 令$f(m)=(n-m)(n+m-2)$，m从1到n。因为$f(m)'=-2m+2$小于零，所以$f(m)$单调递减，当$m=n$时，达到最小值0，当$m=1$时，达到最大值$(n-1)(n-3)$。

⊖ 数值计算的复杂性，即每个节点上的决策算法f被忽略。

China-pub.com
下载

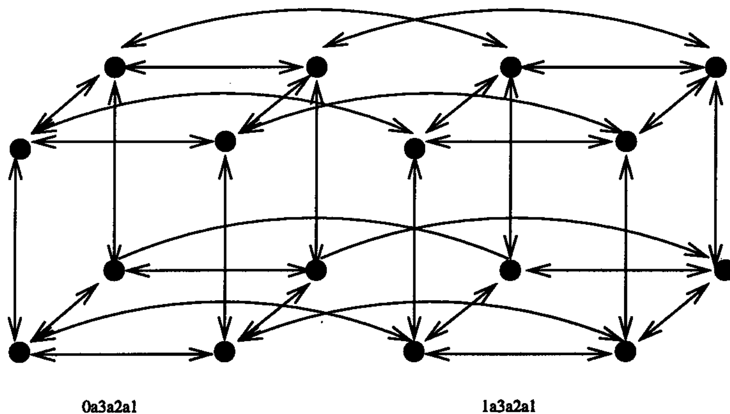

图4-7　在4维超立方上实现的Chang算法

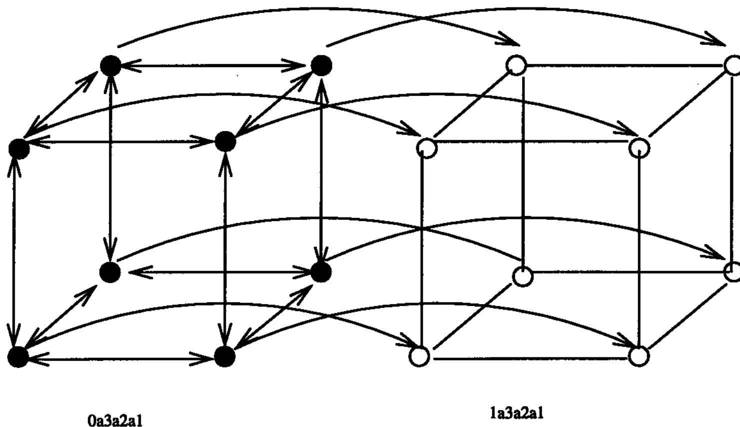

图4-8　在4维超立方上实现的建议的算法

当竞争者的个数未知时，一个实用而有效的需要两个同步回合的方法定义如下：

1) 在第一个回合，进程P_i把一个信号和它的id一起发送给其他进程。

2) 在P_i收到所有信号后，它计算实际参与投标的进程数（令它为 n）。在第二个回合，使用正常的投标问题解决方案。

如果MGFS的大小预先知道，MGFS的概念也可用于这个解决方案。

在不同权重的投标中，每个进程有不同的取胜概率。假设每个进程 P_i有一个权重w_i，w_i是一个整数，w_i越大，P_i取胜的概率也越大。例如，如果 $w_2=2$，$w_1=1$，那么进程P_2的取胜概率是P_1的两倍。一个简单的方法如下：假设有 n个进程P_i，$1\leq P_i\leq n$，权重分别是w_i。假设 $m=\sum_{i=1}^{n} w_i$ 。决策算法是：

$$f(b_1, b_2, \cdots, b_n)=k$$

并且

$$k=\sum_{i=1}^{3} b_i \bmod m+1$$

China-pub.com 下载

其中，[1..m]中的不同元素w_i被预先映射到P_i。所以，如果k被映射到P_c，那么P_c就是优胜者。例如，考虑一个有三个进程的系统。设w_1、w_2、w_3分别是2、1、3。m＝2＋1＋3＝6，并且值{1，2}、{3}和{4，5，6}将分别映射到P_1、P_2和P_3。决策算法是：

$$k = \sum_{i=1}^{3} b_i \bmod 6+1$$

4.6 自稳定

让我们考虑一个有n个进程的系统。一个系统，如果无论它的初始状态是什么，总能保证在有限的步骤内到达一个合法的状态（由谓词P描述），那么它是自稳定的。每个步骤都是一次由特权（privilege）定义的状态转换。与每个进程相关的特权是关于本地进程状态和邻居进程状态的布尔函数。系统处于合法状态当且仅当一个进程拥有特权，否则，它就处于非法状态。当两个特权“出现”（也就是说布尔函数为真）在两个相邻进程时将可能发生冲突。我们假设有一个中央守护进程能“选择”出现的特权之一。

更正式地讲，一个系统S，如果它满足以下两个性质[44]，那么它关于谓词P是自稳定的。

1) 终止性（closure）。P在S的执行下是终止（closed）的，也就是说，一旦P在S中被建立（established），它就不能被伪造（falsified）。

2) 收敛性（convergence）。从任意全局状态开始，S能保证在有限次状态转换内到达一个满足P的全局状态。

Dijkstra在[14]中给出了一个自稳定系统的例子，该系统是一个有限状态自动机的环。在这样一个系统中，有一个“特别的”进程，它对于状态转换使用独特的特权，其他进程使用另一个特权。给定n个k-状态进程，$k > n$，标号分别为P_0到P_{n-1}，其中P_0是“特别的（distinguished）”，其他进程P_i，$0< i \le n-1$是一样的。P_i的状态转换如下：

$$P_i \ne P_{i-1} \rightarrow P_i := P_{i-1}，\ 0 < i \le n-1$$

对于P_0我们有

$$P_0 = P_{n-1} \rightarrow P_0 := (P_0 + 1) \bmod k$$

表4-1 Dijkstra的自稳定算法

P_0	P_1	P_2	特权进程（privileged processes）	要移动的进程（process to make move）
2	1	2	P_0、P_1、P_2	P_0
3	1	2	P_1、P_2	P_1
3	3	2	P_2	P_2
3	3	3	P_0	P_0
0	3	3	P_1	P_1
0	0	3	P_2	P_2
0	0	0	P_0	P_0
1	0	0	P_1	P_1
1	1	0	P_2	P_2

China-pub.com
下载

（续）

P0	P1	P2	特权进程 (privileged processes)	要移动的进程 (process to make move)
1	1	1	P_0	P_0
2	1	1	P_1	P_1
2	2	1	P_2	P_2
2	2	2	P_0	P_0
3	2	2	P_1	P_1
3	3	2	P_2	P_2
3	3	3	P_0	P_0

表4-1表示一个有三个进程的系统实例，$k=4$。初始时，每个进程有一个特权并且移动（move）的选择是随机的。两步（移动）之后，系统到达一个合法的状态——正好只有一个进程有特权的状态，其后系统就一直保持合法状态。

以上算法使用k-状态进程，其中k取决于进程数。另一种算法只使用三个状态：

P_0的状态转换如下：

$$(P_0+1) \bmod 3 = P_1 \rightarrow P_0 := (P_0-1) \bmod 3$$

P_{n-1}的状态转换如下：

$$P_{n-2}=P_0 \wedge (P_{n-2}+1) \bmod 3 \neq P_{n-1} \rightarrow P_{n-1} := (P_{n-2}+1) \bmod 3$$

对所有其他进程：

$$\begin{aligned}&[\ (P_i+1) \bmod 3 = P_{i+1} \rightarrow P_i := P_{i+1}\\ &\square\ (P_i+1) \bmod 3 = P_{i-1} \rightarrow P_i := P_{i-1}\\ &]\end{aligned}$$

为了把自稳定问题和互斥问题联系起来，我们可以把每个特权认为是一个对应于访问临界区权限的令牌。也就是说，如果一个进程有这些令牌之一，它就可以进入它的临界区。刚开始系统中可能有多个令牌，但经过一段有限的时间，系统中只剩下一个令牌在进程间传递。

人们已经在扩展上述简单自稳定算法上做了很多工作。这些扩展可以分为：

• 守护进程的作用。

守护进程的作用是选择一个可以以分散方式实现的特权。另一种方法[4]定义了非干扰（non-interfering）的状态转换。

• 非对称的作用。

很容易看出来，如果进程是相同的，一般来说，自稳定问题是不可解决的。可能有两种形式的非对称：可以选择让所有的进程都不一样或者选择让某一个进程“特别”而其他进程一样。然而，对称并非总是不可得的。Burns和Pachl[5]说明了一种对称的有素数个进程的自稳定环。

• 拓扑的作用。

自稳定问题的最初解决方案是在单向环上的。在其他拓扑结构（如双向环和树）上的自稳定问题人们已经做了一些工作来解决[27]。

• 状态数（number of state）的作用。

China-pub.com 下载

在最初的解决方案中，状态数 k是很大的。一般，使用最小状态数的解决方案是最合适的。对于环而言最小的k是3，尽管存在一种特别的网络只需要二进制－状态机[18]。

自稳定的概念已经在许多领域得到应用，包括自稳定通信协议[21]如滑动窗口协议，两次握手协议、容错时钟同步[20]和分布式进程控制[2]。

一个最重要的应用领域是用于解决暂时性故障的容错设计。传统的容错设计是通过违背它们的个体目标实现的。这样做，许多开发的设计方法都没有通用性。自稳定为暂时性故障提供了一种统一的解决方法，它通过正式把故障合并到设计模型中来实现。至少以下暂时性故障在自稳定系统中是容许的：

- **不一致的初始化，它导致和其他进程不一致的本地状态。**
- **导致发送方和接收方状态不一致的传输错误。**
- **进程的故障和恢复，它导致进程的本地状态和其他进程不一致。**
- **导致本地状态丢失的存储器崩溃。**

大部分自稳定系统的一个弱点就是它们没有提供关于一个非法状态要多长时间才能收敛为合法状态的信息。在有些情况下可能是不够快的。例如，Dijkstra的k－状态算法对每个节点要求$n^{1.5}$阶数的本地状态变化，它的消息复杂性是 n^2。在许多情况下需要在系统的稳定速度和系统的运行速度之间取得折衷。

参考文献

1 Agrawal, D., O. Egecioglu, and A. El Abbadi, “Analysis of quorum-based protocols for distributed (k＋1)-exclusion” *IEEE Transactions on Parallel and Distributed Systems*, 8, 5, May 1997, 533-737

2 Bastani, F. and M. Kam, “A self-stabilizing ring protocol for load balancing in distributed real-time process control systems”, Tech. Report No. UH-CS-87-8, Dept. of Computer Science, Univ. of Houston, Texas, Nov. 1987

3 Brinch Hanson, P., Operating *System Principles*, Prentice-Hall, Inc., 1973.

4 Burns, J.E., “Self-stabilizing rings without demons”, Tech. Rep. GIT-ICS-87/36, Georgia Inst. of Technology, 1987

5 Burns, J. E. and J. Pachl, “Uniform self-stabilizing rings”, *ACM Transactions on Programming Languages and Systems*, 11, 2, 1989, 330-344

6 Chan, M. Y. and F. Y. L. Chin, “Distributed election in complete networks”, *Distributed Computing*, 3, 1988, 19-22

7 Chandy, K. M. and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley Publishing Company, 1989

8 Chang, C. K., “Bidding against competitors”, *IEEE Transactions on Software Engineering*, 16, 1, 1990, 100-104

9 Chang, E. G. and R. Roberts, “An improved algorithm for decentralized extrema-finding in circular configurations of processors”, *Communications of the ACM*, 22, 5, May 1979, 281-

China-pub.com 下载

283

10 Chang, Y. I., "A hybrid distributed mutual exclusion algorithm", *Microprocessing and Microprogramming*, 41, 1996, 715-731

11 Carvalho, O. and G. Roucairol, "On mutual exclusion in computer networks", *Communications of the ACM*, 26, 2, Feb. 1983, 146-147

12 Chow, R., K. Luo, and R. Newman-Wolfe, "An optimal distributed algorithm for failure-driven leader election in bounded-degree networks", Proc. of *IEEE Workshop on Future Trends of Distributed Computing Systems*, 1992, 136-153

13 Dijkstra, E. W., "Cooperating sequential processes", in *Programming Languages,* F. Genuys, ed., Academic Press, 1968, 43-112

14 Dijkstra, E. W., "Self-stabilizing systems in spite of distributed control", *Communications of the ACM*, 17, 1974, 643-644

15 Dolev, D., M. Klawe, and M. Roth, "An O(n log n) unidirectional distributed algorithm for extrema finding in a circle", *Journal of Algorithms*, 3, 1982, 245-260

16 Franklin, W. R., "On an improved algorithm for decentralized extrema-finding in circular configurations of processors", *Communications of the ACM*, 25, 5, May 1982, 336-337

17 Garcia-Molina, H. "Elections in a distributed computing system", *IEEE Transactons on Computers*, 51, 1, Jan 1982, 48-59

18 Ghosh, H., "Self-stabilizing distributed systems with binary machines", *Proc. of 28th Allerton Conf. on Communication*, *Control, and Computing*, 1990

19 Goscinski, A., "Two algorithms for mutual exclusion in real-time distributed computer systems", *Journal of Parallel and Distributed Computing*, 1990, 77-85

20 Gouda, M. and T. Herman, "Stabilizing unison", *Information Processing Letters*, 35, 1990, 171-175

21 Gouda, M. and N. Multari, "Self-stabilizing communication protocols", *IEEE Transactions on Computers*, 40, 4, Apr. 1991, 448-458

22 Helary, J.－M, A. Mostefaoui, and M. Raynal, "A general scheme for token- and tree-based distributed mutual exclusion algorithms", *IEEE Transactions on Parallel and Distributed Systems*, 5, 11, Nov. 1994, 1185-1196

23 Hofri, M., "Proof of a mutual exclusion algorithm－ a classic example", *Operating Systems Review*, 24, 1, Jan 1990, 18-22

24 Hwang, K. and F. A. Briggs, *Computer Architecture and Parallel Processing,* McGraw-Hill Publishing Company, 1984

25 King, G. T., T. B. Gendreau, and L. M. Ni, "Reliable election in broadcast networks", *Journal of Parallel and Distributed Computing*, 1, 1989, 521-546

26 Korach, E., S. Moran, and S. Zabs, "The lower and upper bounds for some distributed algorithms for a complete network of processors", *Proc. of TAP ACM Conf.*, 1984, 199-

China-pub.com 下载

205

27 Kruijer, H., “Self-stabilization (in spite of distributed control) in tree structure systems”, *Information Processing Letters,* 8, 2, 1979, 91-95

28 LeLann, G., “Distributed systems: towards a formal approach”, *Proc. of IFIP Congress*, 1977, 155-160

29 Lynch, N. A., *Distributed Algorithms*, Morgan Kaufmann Publishing, Inc., 1996

30 Maekawa, M., “A square-root(N) algorithm for mutual exclusion in decentralized systems”, *ACM Transactions on Computer Systems*, 2, 4, May 1985, 145-159

31 Milenkovic, M., *Operating Systems: Concepts and Design*, McGraw-Hill Publishing Company, 1987

32 Misra, J., “Detecting termination of distributed computations using markers”, *Proc. of 2nd ACM Conf. on Principles of Distributed Computing,* 1983, 290-294

33 Mans, B. and N. Santoro, “Optimal elections in faulty loop networks and applocations”, *IEEE Transactions on Computers,* 47, 3, Mar. 1998, 286-297

34 Nishio, S., K. F. Li, and E. G. Manning, “A resilient mutual exclusion algorithm for computer networks”, *IEEE Transactions on Parallel and Distributed Systems*, 1, 3, July 1990, 344-355

35 Peterson, G. L., “An O(n log n) un*id*irectional algorithm for the circular extrema problem”, *ACM TOPLAS*, 4, 4, Oct. 1982, 758-762

36 Raymond, K., “A tree-based algorithm for distributed mutual exclusion”, *ACM Transactions on Computer Systems,* 7, 1, Feb. 1989, 61-77

37 Raymond, K., “A distributed algorithm for multiple entries to a critical section”, *Information Processing Letters*, 30, 4, Feb. 1989, 189-193

38 Raymal, M., *Algorithms for Mutual Exclusion*, The MIT Press, 1986

39 Raynal, M., *Distributed Algorithms and Protocols*, John Wiley & Sons, 1988

40 Ricart, G. and A. K. Agrawala, “An optimal algorithm for mutual exclusion in computer networks”, *Communications of the ACM,* 24, 1, Jan. 1981, 9-17

41 Ricart, G. and A. K. Agrawala, “Author’s response to ‘on mutual exclusion in computer networks’ by Carvalho and Roucairol”, *Communications of the ACM,* 26, 2, Feb. 1983, 147-148

42 Saad, Y. and M. H. Schultz, “Topological properties of hypercubes”, *IEEE Transactions on Computers*, 37, 7, July 1988, 867-872

43 Sanders, B., “The information structure of distributed mutual exclusion algorithms”, *ACM Transactions on Computer Systems*, 5, Aug. 1987, 284-299

44 Schneider, M., “Self-stabilization”, *ACM Computing Surveys*, 25, 1, Mar. 1993, 45-67

45 Singh, S. and J. Kurose, “Electing ‘good’ leaders”, *Journal of Parallel and Distributed Computing*, 21, 2, 1994, 184-201

China-pub.com
下载

46 Singhal, M., "A taxonomy of distributed Mutual exclusion", *Journal of Parallel and Distributed Computing*, 18, 1, 1993, 94-101

47 Wu, J. and E. B. Fernandez, "An extended b*id*ding scheme for the distributed consensus problem", Technical Report, TR-CSE-92-38, Florida Atlantic University, 1992

习题

1．为以下系统改进 Lamport的互斥算法：每对 *PE*之间只有一条固定传播延迟的通信路径（由FIFO通道序列组成）。不同的路径可能有不同的延迟而且系统启动时每条路径的延迟是不知道的。只使用物理时钟，所有时钟都是精确的而且一开始是同步的。考虑不同的改进方法并讨论它们的优缺点。

2．说明在Ricart和Agrawala的算法中，临界区是按照与请求相关的时戳的递增顺序批准进入的。解释为什么在 Maekawa算法中不使用时戳并修改 Maekawa算法使得批准信号是基于请求的时戳发出。

3．假设两个进程可以同时进入临界区。提出 Lamport算法和基于令牌环的简单算法的可能扩展。

4．假设一个六进程系统的请求集如下：

R_1：{ P_1，P_2，P_3，P_4，P_5，P_6}

R_2：{ P_1，P_3 }

R_3：{ P_2，P_3 }

R_4：{ P_1，P_2 }

R_5：{ P_2，P_3 }

R_6：{ P_1，P_3 }

以上安排可能有什么潜在的问题？

5．为了将Chang和Roberts的选举算法应用于超立方，可以先在给出的超立方上产生一个生成环（见图4-9的一个3维超立方例子）。

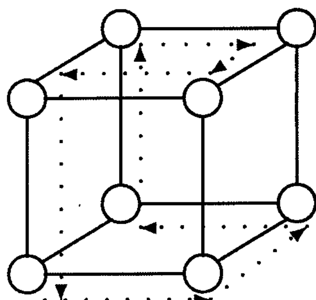

图4-9　3维超立方中的生成环

假设一个进程一次启动一个选举过程。在最坏的情况下，几乎需要两个回合来选出优胜者。对于超立方的拓扑结构，请通过利用超立方提供的多路径改进 Chang和Roberts的算法以获得更快的选举过程。假设每个节点可以同时发送消息给多个邻居节点。只需要算法的高层描述并使

China-pub.com 下载

用一个3维超立方的例子来说明你的方法。

6．**请修改Misra和Chandy的ping-pong算法使得ping和pong沿相反方向传递并比较这两个算法的性能和其他相关方面。**

7．**设计一个修改的ping-pong类型的弹性算法，其中三个令牌（t_1、t_2和t_3）用于一个由n个进程组成的逻辑环。你的算法必须允许 2个令牌的丢失。请清楚说明你的算法如何检测三种不同的情况：(a）没有令牌丢失，(b）丢失一个令牌，(c）丢失两个令牌。你的算法必须包括一个恢复过程。**

8．**对以下系统：**$n=3$，$k=5$，**使用Dijkstra的自稳定算法，请表示出状态的转换序列。假设** $P_0=3$，$P_1=1$，$P_2=4$。

9．**对于任意给定的n维超立方，$n>3$，请找出一条路径使得超立方中的每对（顺序的）节点至少出现一次。也就是说，这条路径由两条连续的 Hamiltonian通路组成。在一个环上扩展Dijkstra的k-状态自稳定算法，从一个特权扩展到两个特权。也就是说，在有限步的状态转换内，环上的节点中有两个特权。**

10．**在投标算法中，可能定义如下决策函数吗？**

$$k = \sum_{i=1}^{n} b_i \bmod n + 1$$

（提示：保证每个进程有相同的取胜机会）

11．**在Lynch的时间片算法中，假设我们改变算法使得进程 P_i在以下回合发送它的id：**

(a）n

(b）$n/2$

相应修改剩余的算法保证它仍然可以正确地选出领导者。另外时间复杂性必须保持不变。

12．**如果给定进程id的范围，也就是说，**$id(i)$ $[a..b]$，**其中a和b是整数，通过降低时间复杂性改进Lynch的变速算法（variable-speed algorithm）。**