Self-stabilizing Systems in Spite of Distributed Control

作者：*Edsger W. Dijkstra* – Burroughs Corporation 1974
原文：http://www-csag.ucsd.edu/teaching/cse291s03/Readings/p643-Dijkstra.pdf
译者：peile.duan@alibaba 2018-10-3
译文：http://duanple.com/?p=1031

[序：本文发表于 1974 年的 CACM，并获得 2002 年第三届 PODB 最具影响力论文奖。全文只有两页，但是却是分布式领域非常重要的一篇论文。该论文发表之后，一开始并没有引起关注，除了 Lamport 读了该文后写了一篇<<On Self-stabilizing Systems>>对其进行了扩展。此后直到 9 年后的 1983 年，在 PODC 的一次会议上，Lamport 再次提到该论文，人们才重新认识到其重要性，具体可以参见<<Solved Problems, Unsolved Problems and Non- Problems in Concurrency>>。

在 Lamport 看来，该论文是 Dijkstra 最非凡的工作之一，是容错领域里程碑式的工作，虽然该论文中从未直接提到“容错”和“可靠性”这样的名词。但是 Lamport 预测 Self-stabilizing 将会成为容错领域的重要概念，以及成为新的重要研究领域。

实际证明 Lamport 是预测是对的，此后 self-stabilization 成为容错领域的重要概念，并成为重要的研究课题。而 Lamport 关于该论文的评价也被认为是他最重要的贡献之一，正是因为他人们才认识到该论文提出的 self-stabilization 的重要性。
]

可以把松散耦合的cyclic sequential processes间的同步任务，看成是要保持一个这样的不变性：“系统要处于一种合法状态”。因此每个进程在运行每一个可能会改变不变性的步骤之前都要先检查一下是可以执行，还是要延迟执行。如果允许不同进程互斥地访问记录有“当前系统状态”的公共存储，解决方案就很简洁—并且可以很系统化地实现。

如果没有一个所有进程都可以访问的公共存储模块的话，复杂度就会增加，因为当前系统的状态就不得不分布式地存储在各个进程内的变量中；进一步地，如果再对通信进行限定，比如每个进程只能与它的邻居进行通信，复杂度还会上升。问题的复杂性在于单个进程的状态只能被整个系统状态中对它可用的那一部分影响，而进程本地动作要基于本地信息实现全局性的目标。这样的系统(可以称之为“分布式控制”系统)已经被设计出来，但是据我所知的所有设计都不是“self-stabilization”的，也就是说它们一旦进入非法状态，就会永远处于这种状态。

考虑一个由少数边连接的图，图中每个节点对应一个有限状态机；图中直接相连的那些状态机，它们相互之间称为邻居。每个状态机定义一个或多个"privileges"，比如是关于它自己和邻居状态的一组布尔函数；当布尔函数值为 true 时，我们就认为对应的 privilege“存在”。我们引入一个中央守护进程{!采用中央守护进程只是为了方便说明，实际上可以采用一个分布式的 daemon 来完成，具体算法本

文没有涉及。一个简单的观察是如果享有 privilege 的是不相邻的状态机，它们的转换是可以并行进行的，它们不需要同步，因为并行运行的结果与通过一个中央 daemon 选择一个进行执行是一致的，因为选择一个执行之后，由于它们两个不相邻，因此另一个的邻居状态不会改变，在新的状态中另一个依然是享有特权者，下一次可以继续选择它，而相邻的实际上因为可以相互通信，它们自己可以协调到底是谁进行状态转换。在下面的状态转换手动演练中，实际上我们就充当了中央守护进程的角色，在有多个 privileges 存在的情况下，选择其中一个进行状态转换}，通过它从“存在”的 privileges 中选择一个进行状态转换{!每个合法状态中可能有多个 privilege“存在”，所以需要从里面选择一个进行状态转换。允许多个享有特权者是一种更通用的需求，只有一个享有特权者的情况是互斥锁，而多个则可以类比信号量}。拥有该被选定的 privilege 的状态机可以进行状态转换—根据一个以它的老状态和邻居状态为输入的状态函数进入新状态。对于不止有一个 privilege“存在”的状态机来说，新状态可能还依赖于被选定的 privilege。状态转换完成后，守护进程再选择一个新的 privilege。

可以通过如下全局规则判断系统是否处于合法状态。要求：

1. 每个合法状态，必须至少要有一个 privilege“存在”
2. 在合法状态的每一个执行步骤都要确保系统还是会处于合法状态
3. 每个 privilege 至少出现在一个合法状态中
4. 对于任意两个合法状态来说，总是可以通过一定执行步骤从一个到达另一个

我们称一个系统是“self-stabilization”的，当且仅当无论系统初始状态如何也无论每次为下一个执行步骤选定的 privilege 是谁，可以保证在有限数目的执行步骤之后总是至少存在一个 privilege 并且系统可以发现它自己处于合法状态。至少据我所知—“self-stabilization”的系统是否存在？作为一个开放性问题已经超过一年时间了。是否能通过各节点的本地执行步骤来满足上述的全局性条件，本来也不是可以很直接地得出结论。而守护进程的不确定性引入了额外的复杂度。实际上该问题可以通过如下三个方案解决。

在如下三个解决方案中，我们假设有 N+1 个进程，它们以 0…N 进行编号。假设当前机器编号为 nr.i。

L，代表机器的左邻居的状态，对应的机器编号为 nr.(i-1) mod (N+1)

S，代表当前机器自己的状态，编号为 nr.i

R，代表机器的右邻居的状态，对应的机器编号为 nr.(i+1) mod (N+1)

换句话说，我们假设机器是像环一样连接在一块。机器 nr.0 又称为“the bottom machine”，机器 nr.N 则又被称为“the top machine”。同时在这里我们将那些只有一个 privilege“存在”的状态作为合法状态。同时所有的方案描述，均采用了如下格式：

“if ***privilege*** then ***corresponding move*** fi”

Solution with K-state Machines (K > N)

每个机器的状态由一个整数 S 表示，并且 0 =< S < K。对于每个机器来说，只定义

一个 privilege，具体如下：
对于 Bottom machine 来说，if L = S then S := (S+1)mod K fi
对于其他机器来说，if L != S then S :=L fi

说明 1：对于一个中央式的守护进程来说，K >= N 就可以了
说明 2：C.S. Scholten 扩展了该解决方案，使之可以应用到任意网络结构。
{!根据格式“if ***privilege*** then ***corresponding move*** fi ”，我们可以看出，在上面的解决方案中，对于 Bottom machine 来说 L=S 就是 ***privilege***，意味着如果 L=S 它就是享有特权者，对于其他机器来说 L!=S 就是 ***privilege***，意味着如果 L !=S 它就是享有特权者。我们可以以 3 个状态机为例，令 K=4，根据上面的算法进行模拟运行可以发现，即使一开始不满足有一个特权者，执行一段时间后会进入只有一个特权者的状态，并且此后一直处于只有一个特权者的状态。比如如下状态转换序列，如下数组中的值分别代表编号为 0,1,2 的状态机的状态值
[0, 1, 2] -> [0, 0, 2] -> [0, 0, 0] -> [1, 0 , 0] -> [1, 1, 0] -> [1, 1, 1] -> [2, 1, 1] -> [2, 2, 1] ->[2, 2, 2]……
如上，初始状态状态机 0,1,2 对应的 S 值为[0, 1, 2]。根据方案描述：对于状态 1 来说状态值是 1，左右值分别是 0 和 2，因此满足 L!=S，所以此时状态机 1 享有特权，对于状态机 2 来说状态值是 2，左右值分别是 1 和 0，因此也满足 L!=S，因此它也是享有特权者，对于 0 来说，要看 L 是否等于 S，由于 0 的左右值是 2 和 1，因此它不享有特权。可以看到这个初始状态，有两个享有特权者，不满足上面关于合法状态中只有一个特权者的定义。然后我们根据状态转换函数，从初始状态开始不断进行状态转换，可以看到到了[0, 0, 0]后，变成了只有状态机 0 为特权者，而此后就一直处于只有一个享有特权者的状态。

可以看到整个过程中，就是从初始处于非法状态，然后经过一定步骤进入了合法状态，此后就一直处于合法状态。而这个过程中，进程只看了它左右邻居的状态，通过局部信息就实现了只有一个享有特权者这一全局性需求。

这实际上说明了在分布式系统中，我们仅通过部分节点的局部信息不需要得到全局状态，就可以实现全局性的状态要求。
}

其他两个方案代表的含义与上述解决方案类似，有个区别是状态机定义了两个 privilege。
{!以第三个方案为例，我们再手动模拟一个状态转换过程，假设有 4 个状态机，它们的初始状态值分别为[0,1,2,0]：
按照下面的算法描述，可以得到如下一个状态转换过程
[0,1,2,0] 此时 0,1,2 均为享有特权者，选择 0 进行状态转换
[2,1,2,0] 此时 1,2 为享有特权者，选择 1 进行状态转换
[2,2,2,0] 此时 2,3 为享有特权者，选择 2 进行状态转换
[2,2,0,0] 此时 1 为享有特权者，1 进行状态转换
[2,0,0,0] 此时 0 为享有特权者，0 进行状态转换
……
可以看到，依然是初始不合法，然后一定步骤后变成合法，之后一直合法。

}
方案 3 具体算法如下：

Solution with Three-state Machines

Here each machine state is represented by an integer value S, satisfying $0 \leq S < 3$. The privileges are defined as follows:

for the bottom machine:

if $(S+1)$**mod** 3 $=$ R **then** $S := (S-1)$**mod** 3 **fi**

for the top machine:

if $L = R$ **and** $(L+1)$**mod** 3 $\neq S$ **then** $S := (L+1)$**mod** 3 **fi**

for the other machines:

if $(S+1)$**mod** 3 $= L$ **then** $S := L$ **fi;**

if $(S+1)$**mod** 3 $= R$ **then** $S := R$ **fi**

{!上述三个解决方案说明：对于一个分布式系统来言，即使节点只能跟所有节点中的部分节点进行通信，对于上述特权者(互斥，任意时刻只有一个特权者)问题而言，存在一个“self-stabilization”的算法。无论系统初始状态如何，经历一定步骤之后它都可以进入合法状态。

“self-stabilization”属性使得分布式算法可以从暂态错误中恢复。Dijskstra 在本论文中提出了“self-stabilization”概念，并以上述“令牌环”问题为例给出了相应的自稳定算法。在“令牌环”场景下，计算机网络连接成环状，每个计算机可以获取它前面的那个计算机的状态，该状态可以显示该计算机“持有令牌”还是“不持有令牌”。对应的算法要满足如下两个条件：1.任意时刻只有一个计算机持有令牌 2.持有令牌的计算机会将令牌传给它后面的计算机，最终该令牌可以在环上循环流转。不持有令牌对于单个计算机来说是合法的，但是如果所有计算机都不持有令牌，对于整个系统来说就是非法的。类似的，如果有不止一个计算机持有令牌，也是非法的，但是对于每个计算机来说这很难发现，因为他们每个都只能与邻居通信。因此，本论文中的算法并没有去检测错误，而是确保系统不断向着合法状态前进。而且那个时候用于错误检测的传统方法很困难而且很耗时。

但是随着新的更高效的错误检测算法的提出，基于错误检测还可以将非自稳定算法结合自稳定算法实现更高效的自稳定算法，具体可以参考 wiki。
}

http://en.wikipedia.org/wiki/Self-stabilization
http://selfstabilization.org/

http://selfstabilization.org/bibliography/access.html
http://www.cs.uic.edu/~ajayk/SelfStabilization.pdf
http://www.podc.org/influential/2002-influential-paper/