第二部分 分布式算法

汪炀

中国科学技术大学计算机系
国家高性能计算中心（合肥）

Ch.1 导论

§1.1 分布式系统

- Def：一个分布式系统是一个能彼此通信的单个计算装置的集合（计算单元：硬——处理器；软——进程）

  包括：紧耦合系统----如共享内存多处理机

  松散系统-----cow、Internet

- 与并行处理的分别(具有更高程度的不确定性和行为的独立性)
  - 并行处理的目标是使用所有处理器来执行一个大任务
  - 而分布式系统中，每个处理器一般都有自己独立的任务，但由于各种原因（为共享资源，可用性和容错等），处理机之间需要协调彼此的动作。
- 分布式系统无处不在，其作用是：

  ①共享资源

  ②改善性能：并行地解决问题

  ③改善可用性：提高可靠性，以防某些成分发生故障

§1.1 分布式系统
分布式系统软件实例简介

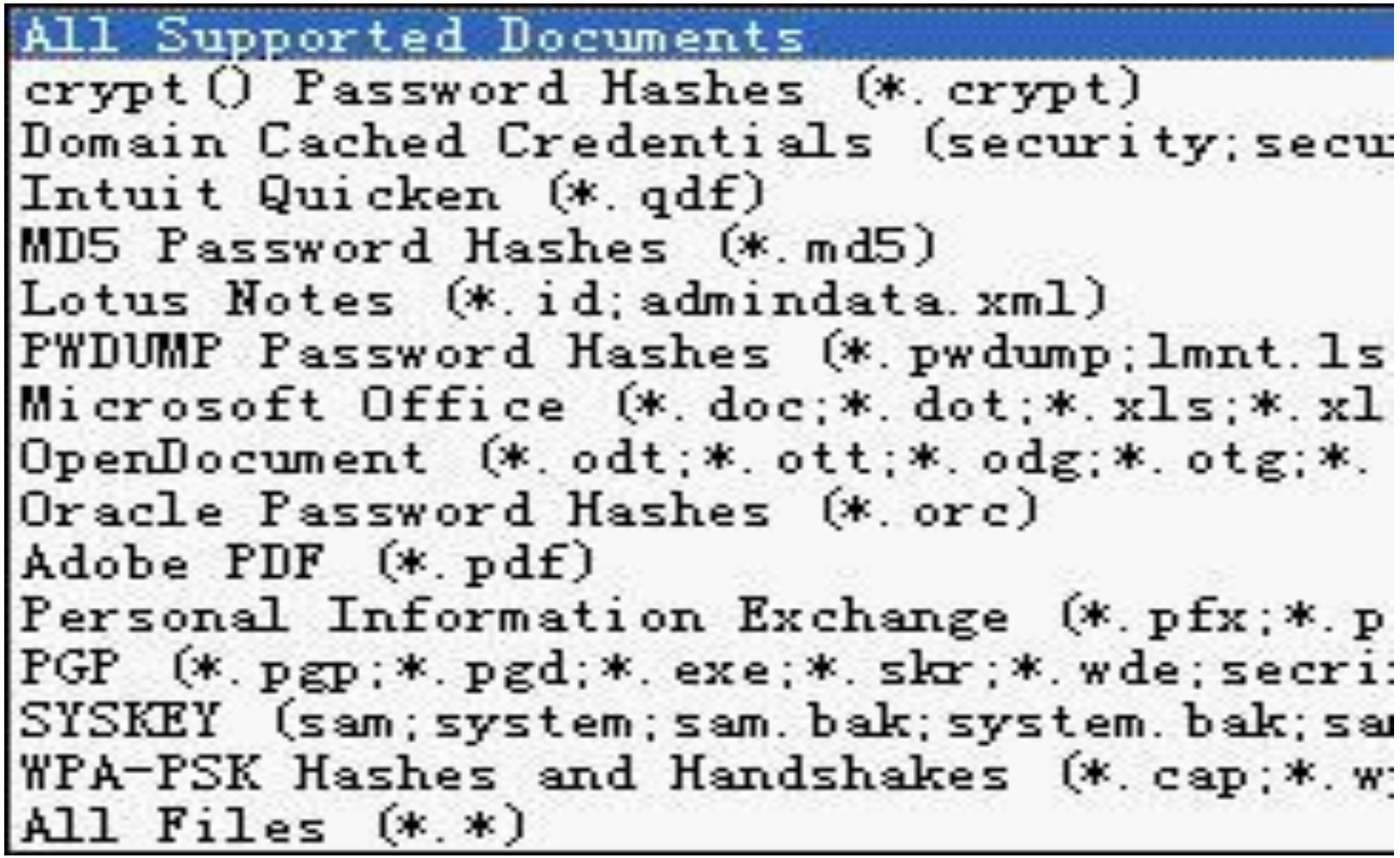
- ElcomSoft Distributed Password Recovery 是一款俄罗斯安全公司出品的分布式密码暴力破解工具
- 能够利用Nvidia显卡使WPA和WPA2无线密钥破解速度提高100倍
- 还允许数千台计算机联网进行分布式并行计算

§1.1 分布式系统
系统适用范围

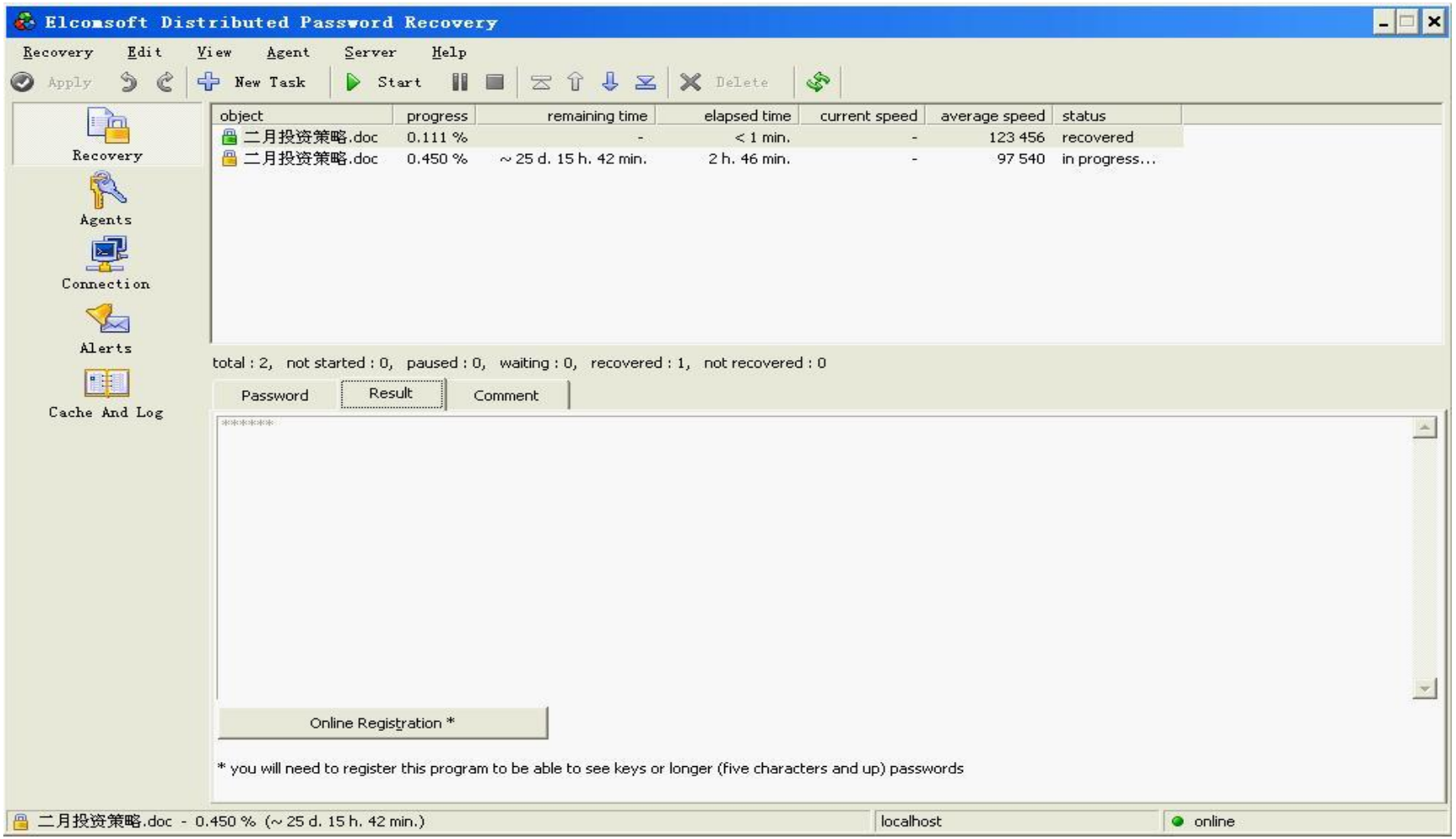
- ElcomSoft 的密码恢复软件主要是面向 Office，包括（Word, Excel, Access, Outlook, Outlook Express, VBA, PowerPoint and Visio)

- 其他的面向微软的产品有（Project, Backup, Mail, Schedule+), archive products (including ZIP, RAR, ACE and ARJ files)等

§1.1 分布式系统
演示界面-支持的文件类型

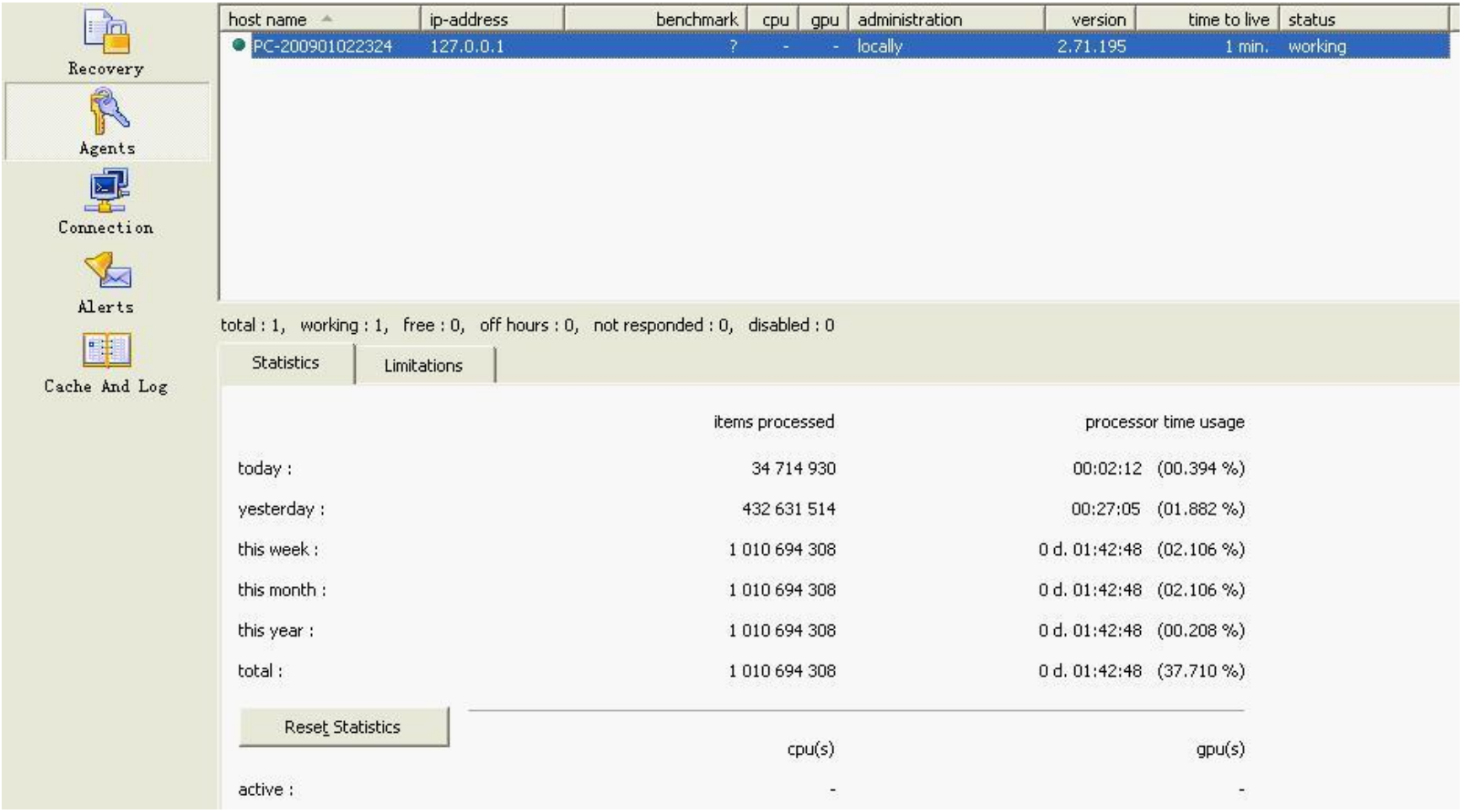


§1.1 分布式系统
演示—主界面



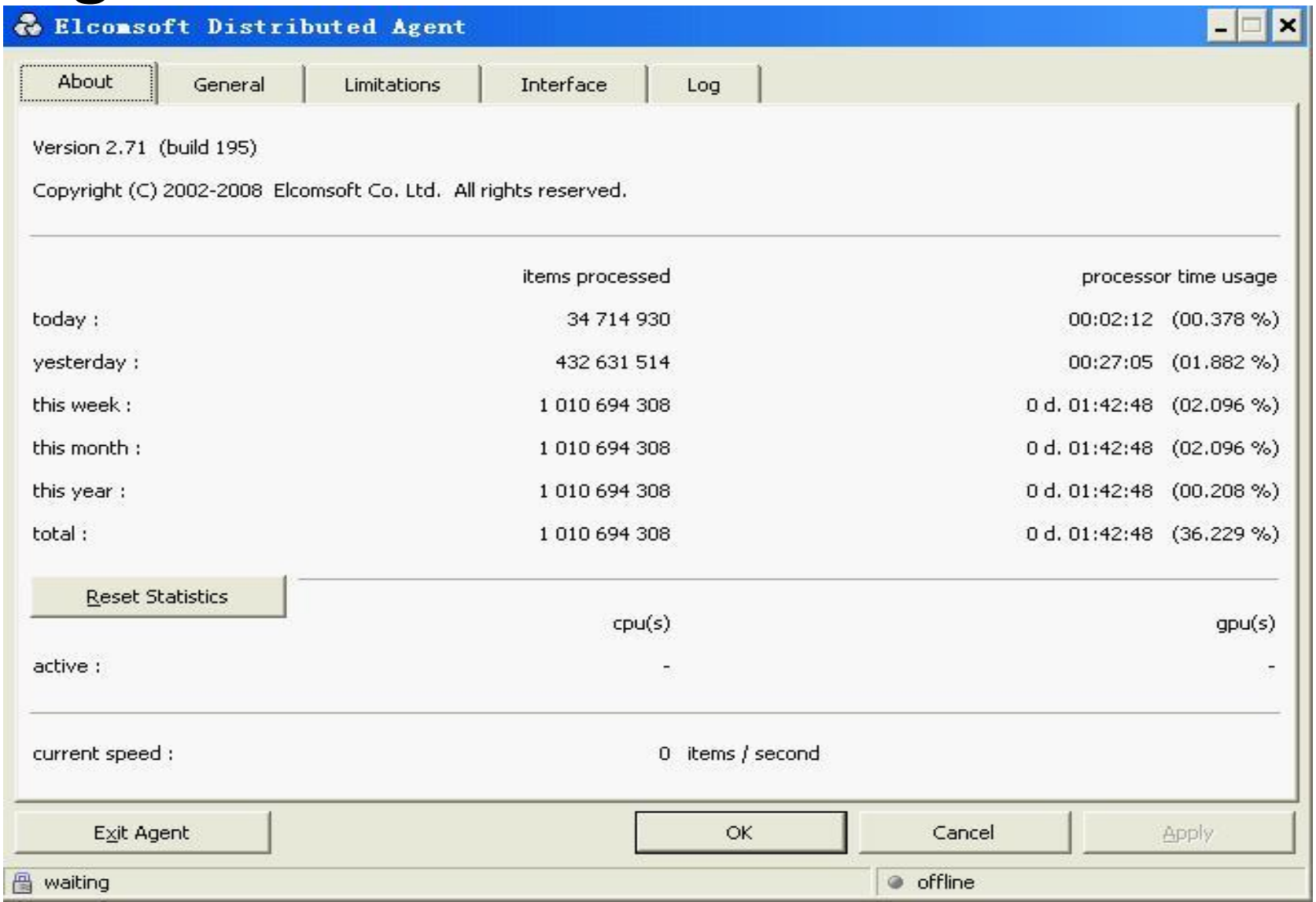
§1.1 分布式系统
最终破解效果

- DOC加密的文档，8位数字型密码小于1分钟即可成功解密



§1.1 分布式系统

Agents工作界面

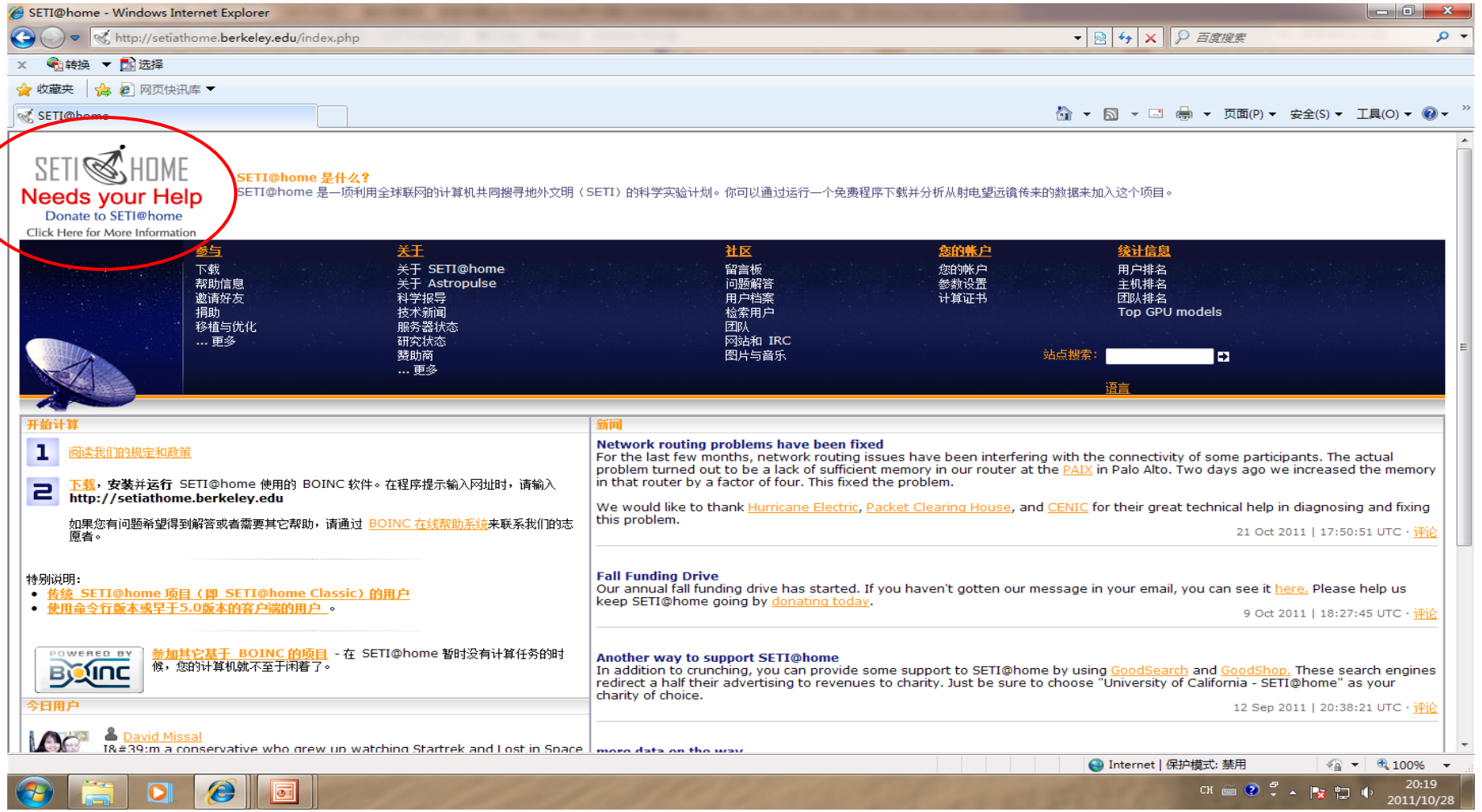


§1.1 分布式系统
NASA SETI寻找外星人计划

- SETI (搜寻外星智慧) 是一个寻找地球外智慧生命的科学性实验计划，使用射电望远镜来监听太空中的窄频无线电讯号。假设这些讯号中有些不是自然产生的，那么只要我们侦测到这些讯号就可以证明外星科技的存在。
- 射电望远镜讯号主要由噪声 (来自天体的发射源与接收者的电子干扰) 与像电视转播站、雷达和卫星等等的人工讯号所组成。现代的 Radio SETI 计划会分析这些数字信息。有更强大的运算能力就可以搜寻更广泛的频率范围以及提高灵敏度。因此，Radio SETI 计划对运算能力的需求是永无止尽的。
- 原来的 SETI 项目曾经使用望远镜旁专用的超级计算机来进行大量的数据分析。1995年，David Gedye 提议射电 SETI 使用由全球联网的大量计算机所组成的虚拟超级计算机来进行计算，并创建了 SETI@home 项目来实验这个想法。SETI@home 项目于1999年5月开始运行。

§1.1 分布式系统

NASA SETI寻找外星人计划



§1.1 分布式系统

- 分布式系统面临的困难
  - 异质性：软硬件环境
  - 异步性：事件发生的绝对、甚至相对时间不可能总是精确地知道
  - 局部性：每个计算实体只有全局情况的一个局部视图
  - 故障：各计算实体会独立地出故障，影响其他计算实体的工作。

§1.2 分布式计算的理论

- 目标：针对分布式系统完成类似于顺序式计算中对算法的研究
  - 具体：对各种分布式情况发生的问题进行抽象，精确地陈述这些问题，设计和分析有效算法解决这些问题，证明这些算法的最优性。
- 计算模型：
  - 通信：计算实体间msg传递还是共享变量？
  - 哪些计时信息和行为是可用的？
  - 容许哪些错误
- 复杂性度量标准
  - 时间，空间
  - 通信成本：msg的个数，共享变量的大小及个数
  - 故障和非故障的数目

§1.2 分布式计算的理论

- 否定结果、下界和不可能的结果

  常常要证明在一个特定的分布式系统中，某个特定问题的不可解性。

  就像NP-完全问题一样，表示我们不应该总花精力去求解这些问题。

  当然，可以改变规则，在一种较弱的情况下去求解问题。

- 我们侧重研究：
  - 可计算性：问题是否可解？
  - 计算复杂性：求解问题的代价是什么？

§1.3 理论和实际之关系

主要的分布式系统的种类，分布式计算理论中常用的形式模型之间的关系

- 种类
  - 支持多任务的OS：互斥，死锁检测和防止等技术在分布式系统中同样存在。
  - MIMD机器：紧耦合系统，它由分离的硬件运行共同的软件构成。
  - 更松散的分布式系统：由网络（局域、广域等）连接起来的自治主机构成

    特点是由分离的硬件运行分离的软件。实体间通过诸如TCP/IP栈、CORBA或某些其它组件或中间件等接口互相作用。

§1.3 理论和实际之关系

- 模型

  模型太多。这里主要考虑三种，基于通信介质和同步程度考虑。

  ① 异步共享存储模型：用于紧耦合机器，通常情况下各处理机的时钟信号不是来源于同一信号源

  ② 异步msg传递模型：用于松散耦合机器及广域网

  ③ 同步msg传递模型：这是一个理想的msg传递系统。该系统中，某些计时信息（如msg延迟上界）是已知的，系统的执行划分为轮执行，是异步系统的一种特例。

  该模型便于设计算法，然后将其翻译成更实际的模型。

- **Dijkstra E W. Co-operating Sequential Process. In programming Language. F. Genyus(ed.). [S.I.]: Academic Press, 1968, 43-112;**
- **Owicki S, Gries D. Verifying Properties of Parallel Programs: An Axiomatic Approach. Communication ACM 19, 5(1976), 279-285;**

§1.3 理论和实际之关系

- 错误的种类
  - 初始死进程

    指在局部算法中没有执行过一步。
  - Crash failure崩溃错误(损毁模型)

    指处理机没有任何警告而在某点上停止操作。
  - Byzantine failure拜占庭错误

    一个出错可引起任意的动作, 即执行了与局部算法不一致的任意步。拜占庭错误的进程发送的消息可能包含任意内容。

Ch.2 消息传递系统中的基本算法

本章介绍无故障的msg传递系统，考虑两个主要的计时模型：同步及异步。
定义主要的复杂性度量、描述伪代码约定，最后介绍几个简单算法

§2.1 消息传递系统的形式化模型

§2.1.1 系统

1.基本概念

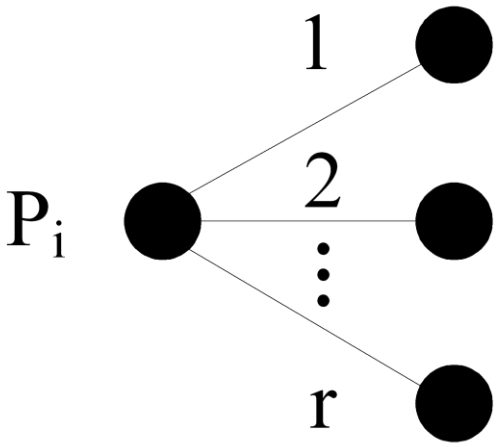
- 拓扑：无向图 结点——处理机
  边 ——双向信道

§2.1.1 系统

- 算法：由系统中每个处理器上的局部程序构成
  - 局部程序　　执行局部计算——本地机器

    发送和接收msg——邻居
  - 形式地：一个系统或一个算法是由n个处理器p_0, p_1,…p_{n-1}构成，每个处理器p_i可以模型化为一个具有状态集Q_i的状态机（可能是无限的）

§2.1.1 系统



- 状态（进程的局部状态）

  由p_i的变量，p_i的msgs构成。

  p_i的每个状态由2r个msg集构成：

  - $outbuf_i[l](1 \le l \le r)$：$p_i$经第$l$条关联的信道发送给邻居，但尚未传到邻居的msg。
  - $inbuf_i[l](1 \le l \le r)$：在$p_i$的第$l$条信道上已传递到$p_i$，但尚未经$p_i$内部计算步骤处理的msg。

  模拟在信道上传输的msgs

§2.1.1 系统

- 初始状态：
  - Q_i包含一个特殊的初始状态子集：每个$inbuf_i[l]$必须为空，但$outbuf_i[l]$未必为空。
- 转换函数(transition)：

  处理器p_i的转换函数(实际上是一个局部程序)
  - 输入：p_i可访问的状态
  - 输出：对每个信道l，至多产生一个msg输出
  - 转换函数使输入缓冲区$(1 \le k \le r)$清空。

§2.1.1 系统

- 配置：配置是分布式系统在某点上整个算法的全局状态

  向量=(q_0, q_1,…q_{n-1}), q_i是p_i的一个状态

  一个配置里的outbuf变量的状态表示在通信信道上传输的信息，由del事件模拟传输

  一个初始的配置是向量=(q_0, q_1,…q_{n-1}), 其中每个q_i是p_i的初始状态，即每个处理器处于初始状态

§2.1.1 系统

- 事件：系统里所发生的事情均被模型化为事件，对于msg传递系统，有两种：

comp(i)——计算事件。代表处理器p_i的一个计算步骤。其中，p_i的转换函数被用于当前可访问状态

del(i,j,m)——传递事件，表示msg m从p_i传送到p_j

- 执行：系统在时间上的行为被模型化为一个执行。它是一个由配置和事件交错的序列。该序列须满足各种条件，主要分为两类：

§2.1.1 系统

①Safety条件： (安全性)

表示某个性质在每次执行中每个可到达的配置里都必须成立

在序列的每个有限前缀里必须成立的条件

例如：“在leader选举中，除了p_{max}外，没有哪个结点宣称自己是leader”

非形式地：安全性条件陈述了“尚未发生坏的情况”“坏事从不发生”

§2.1.1 系统

②liveness条件： (活跃性)

表示某个性质在每次执行中的某些可达配置里必须成立。

必须成立一定次数的条件(可能是无数次)

例如：条件：“p_1最终须终止”，要求p_1的终止至少发生一次；“leader选举，p_{max}最终宣布自己是leader”

非形式地，一个活跃条件陈述：“最终某个好的情况发生”

对特定系统，满足所有要求的安全性条件的序列称为一个执行;若一个执行也满足所有要求的活跃性条件，则称为容许(合法的)(admissible)执行

第二部分 分布式算法

汪炀

第二次课

中国科学技术大学计算机系
国家高性能计算中心（合肥）

§2.1.1 系统

2.异步系统

- 异步：msg传递的时间和一个处理器的两个相继步骤之间的时间无固定上界

  例如，Internet中，email虽然常常是几秒种到达，但也可能要数天到达。当然msg延迟有上界，但它可能很大，且随时间而改变。

  因此异步算法设计时，须使之独立于特殊的计时参数，不能依赖于该上界。

- 执行片断

  一个异步msg传递系统的一个执行片断 α 是一个有限或无限的序列：

  $C_0, \Phi_1, C_1, \Phi_2, C_2, \Phi_3, \ldots$, (C_0不一定是初始配置)

  这里C_k是一个配置， Φ_k是一个事件。若 α 是有限的，则它须结束于某个配置，且须满足下述条件：

§2.1.1 系统

- 若Φ_k =del(i,j,m)，则m必是C_{k-1}里的$outbuf_i[l]$的一个元素，这里l是p_i的信道$\{p_i,p_j\}$的标号

  从C_{k-1}到C_k的唯一变化是将m从C_{k-1}里的$outbuf_i[l]$中删去，并将其加入到C_k里的$inbuf_j[h]$中，h是p_j的信道$\{p_i,p_j\}$的标号。

  即：传递事件将msg从发送者的输出缓冲区移至接收者的输入缓冲区。

- 若Φ_k =comp(i)，则从C_{k-1}到C_k的变化是

  ①改变状态：转换函数在p_i的可访问状态(在配置C_{k-1}里)上进行操作，清空$inbuf_i[l]$，$(1\leq k\leq r)$

  ②发送msg：将转换函数指定的消息集合加到C_k里的变量$outbuf_i$上。(Note:发送send，传递delivery之区别)

  即： p_i以当前状态(在C_{k-1}中)为基础按转换函数改变状态并发出msg。

§2.1.1 系统

- 执行：一个执行是一个执行片断$C_0, \Phi_1, C_1, \Phi_2, \ldots$,这里$C_0$是一个初始配置。
- 调度：一个调度(或调度片段)总是和执行(或执行片断)联系在一起的，它是执行中的事件序列：$\Phi_1, \Phi_2, \ldots$。
  并非每个事件序列都是调度。例如，del(1,2,m)不是调度，因为此事件之前，p_1没有步骤发送(send)m。
  若局部程序是确定的，则执行(或执行片断)就由初始配置C_0和调度(或调度片断)σ唯一确定，可表示为exec(C_0, σ)。

§2.1.1 系统

- 容许执行：(满足活跃性条件)
  异步系统中，若某个处理器有无限个计算事件，每个发送的msg都最终被传递，则执行称为容许的。
  Note: 无限个计算事件是指处理器没有出错，但它不蕴含处理器的局部程序必须包括一个无限循环
  非形式地说：一个算法终止是指在某点后转换函数不改变处理器的状态。

- 容许的调度：
  若它是一个容许执行的调度。

§2.1.1 系统

3.同步系统

在同步模型中，处理器按锁步骤(lock-step)执行：

执行被划分为轮，每轮里，①每个处理器能够发送一个msg到每个邻居，这些msg被传递。②每个处理器一接到msg就进行计算。

虽然特殊的分布系统里一般达不到，但这种模型对于设计算法非常方便，因为无需和更多的不确定性打交道。当按此模型设计算法后，能够很容易模拟得到异步算法。

- 轮：在同步系统中，配置和事件序列可以划分成不相交的轮，每轮由一个传递事件(将outbuf的消息传送到信道上使outbuf变空)，后跟一个计算事件(处理所有传递的msg)组成。

§2.1.1 系统

- 容许的执行：指无限的执行。
  因为轮的结构，所以
  每个处理器执行无限数目的计算步，
  每个被发送的msg最终被传递
- 同步与异步系统的区别
  在一个无错的同步系统中，一个算法的执行只取决于初始配置
  但在一个异步系统中，对于相同的初始配置及无错假定，因为处理器步骤间隔及消息延迟均不确定，故同一算法可能有不同的执行。

§2.1.2 复杂性度量

- 分布式算法的性能：msg个数和时间。

  最坏性能、期望性能

- 终止：假定每个处理器的状态集包括终止状态子集，每个的p_i的转换函数对终止状态只能映射到终止状态

当所有处理机均处于终止状态且没有msg在传输时，称系统(算法)已终止。

- 算法的msg复杂性(最坏情况)：算法在所有容许的执行上发送msg总数的最大值(同步和异步系统)

§2.1.2 复杂性度量

- 时间复杂度

  ①同步系统：最大轮数，即算法的任何容许执行直到终止的最大轮数。

  ②异步系统：假定任何执行里的msg延迟至多是1个单位的时间，然后计算直到终止的运行时间

- 计时执行(timed execution)

  指：每个事件关联一个非负实数，表示事件发生的时间。时间起始于零，且须是非递减的。但对每个单个的处理器而言是严格增的。

  若执行是无限的，则执行的时间是无界的。因此执行中的事件可根据其发生时间来排序

  不在同一处理器上的多个事件可以同时发生，在任何有限时间之前只有有限数目的事件发生。

§2.1.2 复杂性度量

- 消息的延迟

  发送msg的计算事件和处理该msg的计算事件之间所逝去的时间

  它主要由msg在发送者的outbuf中的等待时间和在接收者的inbuf中的等待时间所构成。

- 异步算法的时间复杂性

  异步算法的时间复杂性是所有计时容许执行中直到终止的最大时间，其中每个msg延时至多为1。

§2.1.3 伪代码约定

在形式模型中，一个算法将根据状态转换来描述。但实际上很少这样做，因为这样做难于理解。

实际描述算法有两种方法：

①叙述性：对于简单问题

②伪码形式：对于复杂问题

§2.1.3 伪代码约定

- 异步算法：对每个处理器，用中断驱动来描述异步算法。

  在形式模型中，每个计算事件1次处理所有输入缓冲区中的msgs。而在算法中，一般须描述每个msg是如何逐个处理的

  异步算法也可在同步系统中工作，因为同步系统是异步系统的一个特例。

  一个计算事件中的局部计算的描述类似于顺序算法的伪代码描述。
- 同步算法：逐轮描述
- 伪代码约定：

  —在p_i的局部变量中，无须用i做下标，但在讨论和证明中，加上下标i以示区别。

  —“//”后跟注释

§2.2 生成树上的广播和汇集

信息收集及分发是许多分布式算法的基础。故通过介绍这两个算法来说明模型、伪码、正确性证明及复杂性度量等概念。

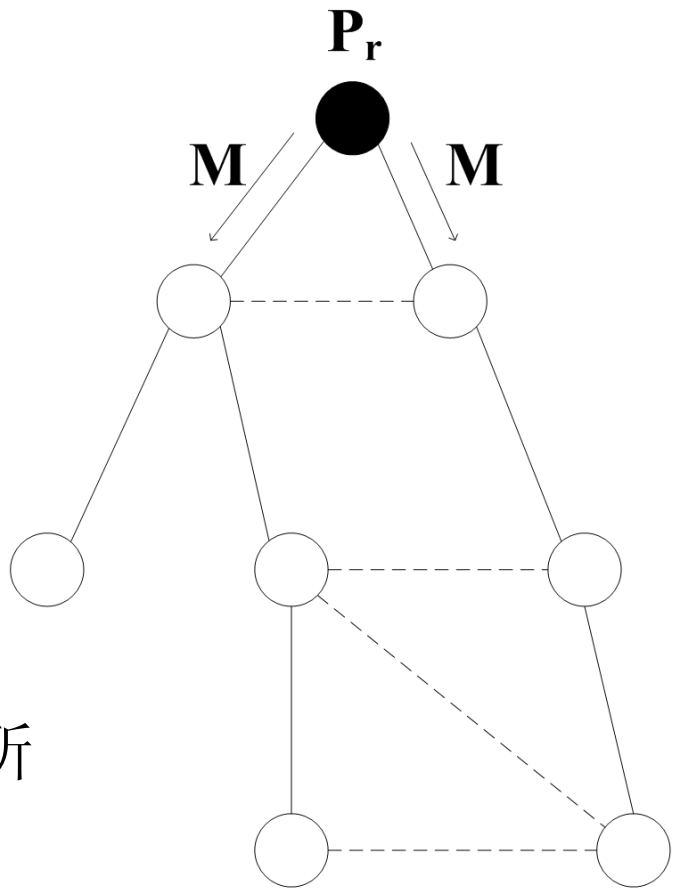
由于分布式系统中，每个节点并不知道全局拓扑，但某些算法需要在特定结构下才能达到最优，比如广播/敛播在树结构下才能达到消息复杂度最优，因此构造生成树是必要的，是其他算法的基础。

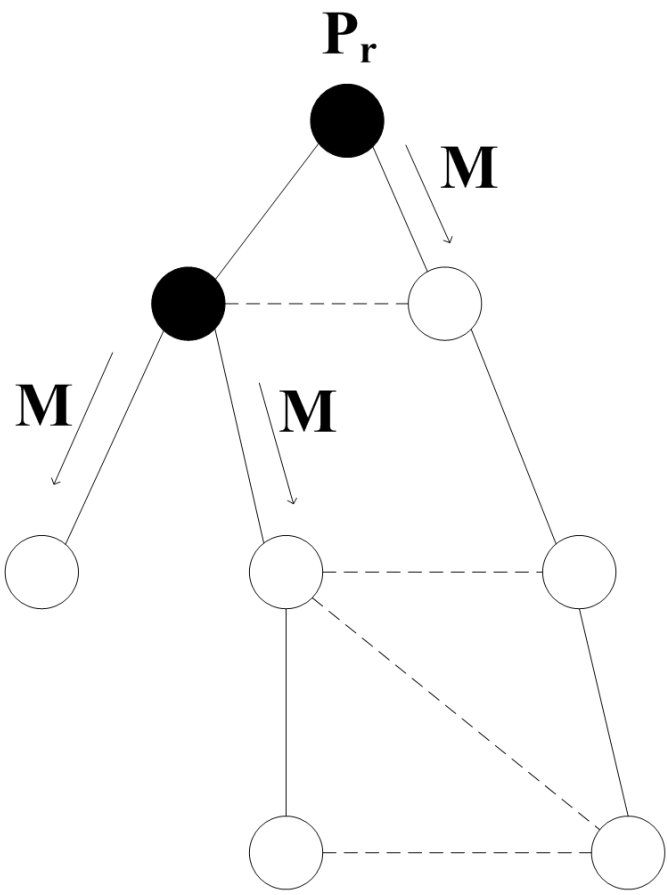
§2.2.1 广播(Broadcast)

假定网络的生成树已给定。某处理器p_r希望将消息M发送至其余处理器。

假定生成树的根为p_r，每个处理器有一个信道连接其双亲(p_r除外)，有若干个信道连接其孩子。

§2.2.1 广播





（a）　　（b）

树边，信道
非树边
已收到M的结点
未收到M的结点

- 根p_r发送M给所有孩子。(a)
- 当某结点收到父结点的M时，发送M到自己的所有孩子(b)。

§2.2.1 广播

1.伪码算法

Alg2.1 Broadcast

p_r://发动者。假设初始化时M已在传输状态

1. upon receiving no msg: //pr发送M后执行终止
2. terminate; //将terminated置为true。

p_i($i \neq r, 0 \leq i \leq n-1$):

3. upon receiving M from parent:
4. send M to all children;
5. terminate;

2.用状态转换来分析算法

- 每个处理器p_i包含状态

  —变量$parent_i$：表示处理器p_i双亲结点的标号或为nil(若i=r)

  —变量$children_i$：p_i的孩子结点标号的集合

  —布尔变量$terminated_i$：表示p_i是否处于终止状态

§2.2.1 广播

- 初始状态
  - parent和children的值是形成生成树时确定的
  - 所有terminated的值均为假
  - $outbuf_r[\ j\ ]$，　$j \forall \in children_r$持有消息M，注意j不是信道标号，而是r的邻居号。（任何系统中，均假定各节点标号互不相等）
  - 所有其他结点的outbuf变量均为空。
- comp(i)的结果

  若对于某个k，M在$inbuf_i[k]$里，则M被放到$outbuf_i[\ j\ ]$里，　$\forall j \in children_i$

§2.2.1 广播

- p_i进入终止状态

  将$terminated_i$置为true；若i=r且$terminated_r$为false，则$terminated_r$立即置为true，否则空操作。

- 该算法对同步及异步系统均正确，且在两模型中，msg和时间复杂度相同。

- Msg复杂度

  无论在同步还是异步模型中，msg M在生成树的每条边上恰好发送一次。

  因此，msg复杂性为n-1。

§2.2.1 广播

- 时间复杂性：

①同步模型：时间由轮来度量。

Lemma2.1 在同步模型中，在广播算法的每个容许执行里，树中每个距离p_r为t的处理器在第t轮里接收消息M。

pf：对距离t使用归纳法。

归纳基础：t=1，p_r的每个孩子在第1轮里接收来自于p_r的消息M

归纳假设：假设树上每个距p_r为t-1≥1的处理器在第t-1轮里已收到M。

归纳步骤：设p_i到p_r距离为t，设p_j是p_i的双亲，因p_j到p_r的距离为t-1，由归纳假设，在第t-1轮p_j收到M。由算法描述知，在第t轮里p_i收到来自于p_j的消息M

Th2.2 当生成树高度为d时，存在一个消息复杂度为n-1，时间复杂度为d的同步广播算法

§2.2.1 广播

②异步模型

Lemma2.3 在异步模型的广播算法的每个容许执行里，树中每个距离p_r为t的处理器至多在时刻t接收消息M。

pf：对距离t做归纳。
对t=1，初始时，M处在从p_r到所有距离为1的处理器p_i的传输之中，由异步模型的时间复杂性定义知，p_i至多在时刻1收到M。

$p_i \in$ {距p_r为t的处理器}，设p_j是p_i的双亲，则p_j与p_r的距离为t-1，由归纳假设知，p_j至多在时刻t-1收到M，由算法描述知，p_j发送给p_i的M至多在t时刻到达。

Th2.4 同Th2.2

下次继续！

第二部分 分布式算法

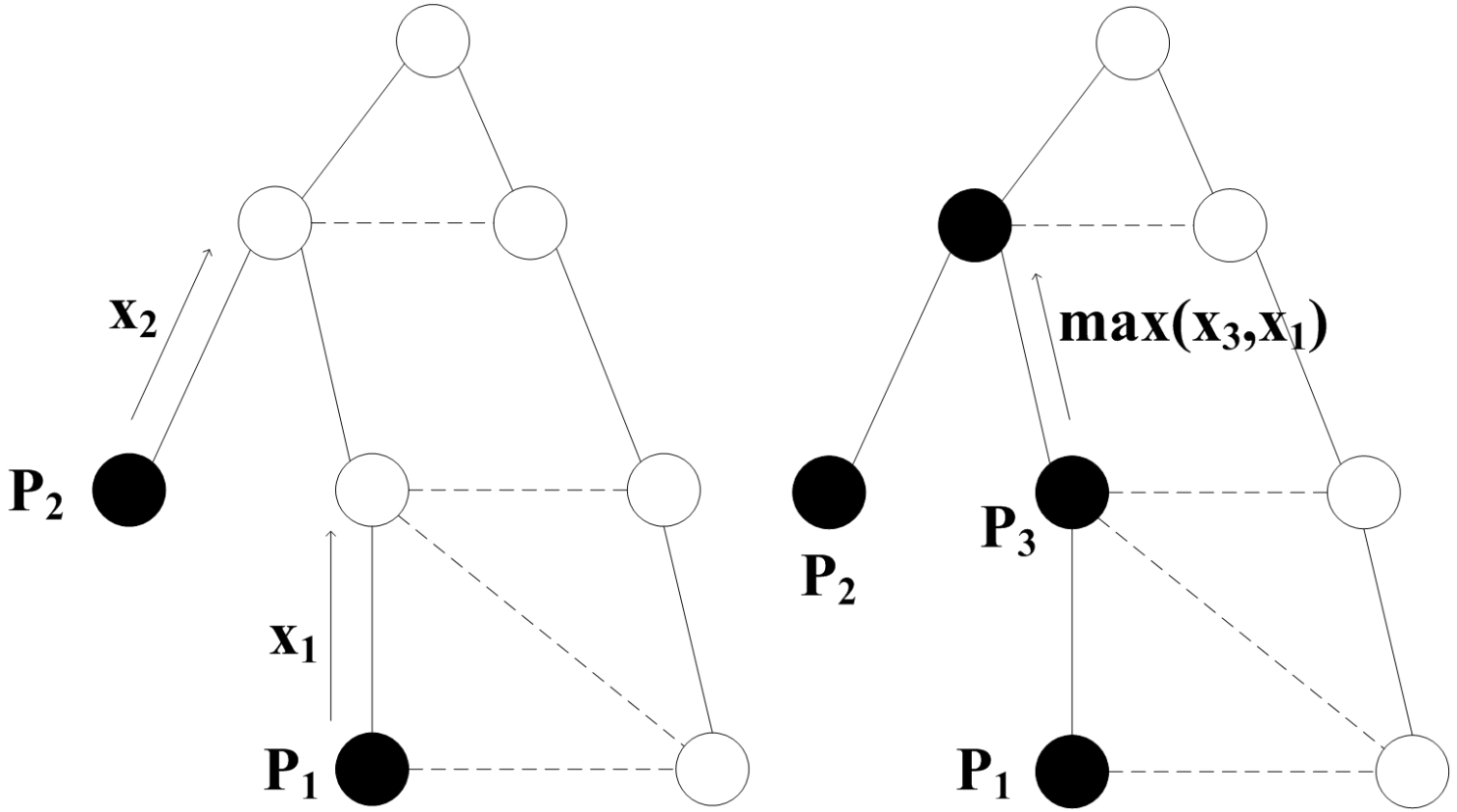
第三次课

中国科学技术大学计算机系
国家高性能计算中心（合肥）

§ 2.2.2 convergecast(汇集，敛播)

与广播问题相反，汇集是从所有结点收集信息至根。为简单起见，先考虑一个特殊的变种问题：

假定每个p_i开始时有一初值x_i，我们希望将这些值中最大者发送至根p_r。



§2.2.2 convergecast(汇集，敛播)

- 算法

  每个叶子结点p_i发送x_i至双亲。//启动者

  对每个非叶结点p_j，设p_j有k个孩子$p_{i1},\ldots,p_{ik}$，p_j等待k个孩子的msg $v_{i1},v_{i2},\ldots,v_{ik}$，当$p_j$收到所有孩子的msg之后将$v_j=\max\{x_j,v_{i1},\ldots,v_{ik}\}$发送到$p_j$的双亲。

  换言之：叶子先启动，每个处理器p_i计算以自己为根的子树里的最大值v_i，将v_i发送给p_i的双亲。

- 复杂性

  Th2.5 当生成树高为d时，存在一个异步的敛播方法，其msg复杂性为n-1，时间复杂度为d。(与Th2.2相同)

- 广播和敛播算法用途：初始化某一信息请求(广播发布)，然后用敛播响应信息至根。

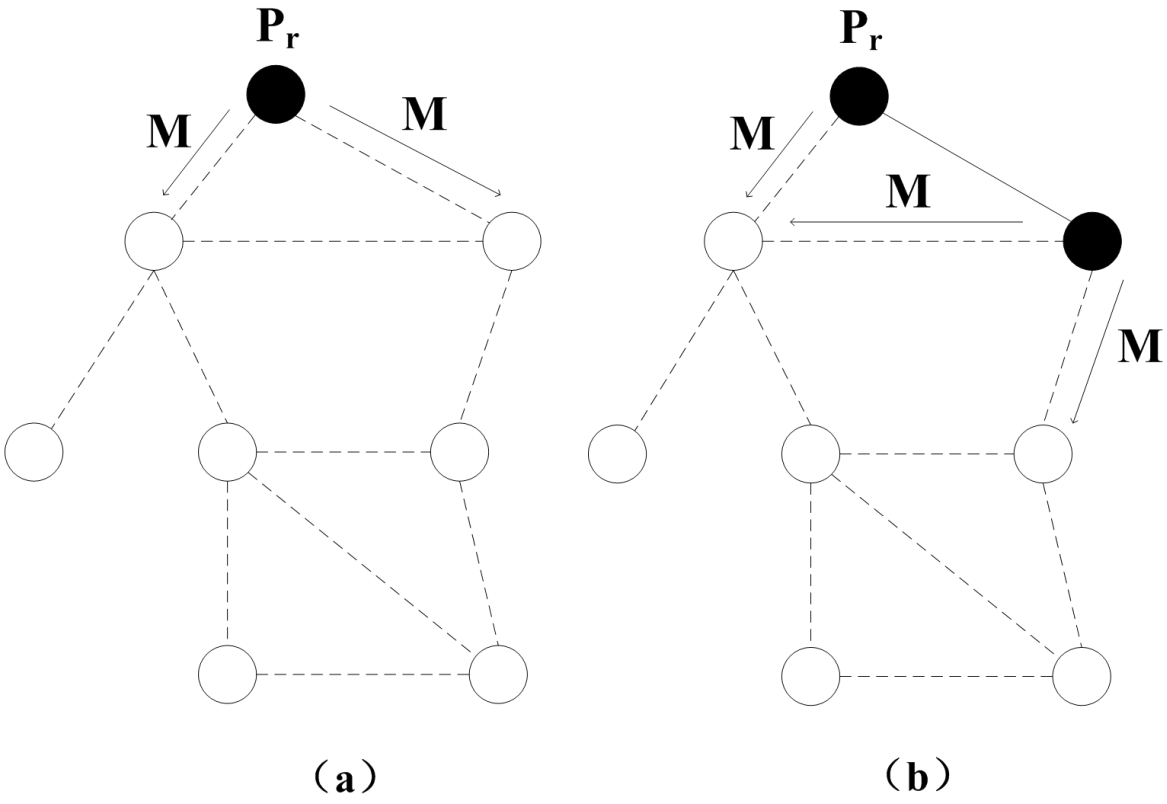
§2.3 构造生成树

上节算法均假设通信网的生成树已知。本节介绍生成树的构造问题。

1.Flooding算法(淹没)

- **算法**

设p_r是特殊处理器。从p_r开始，发送M到其所有邻居。当p_i第1次收到消息M（不妨设此msg来自于邻居p_j）时，p_i发送M到除p_j外的所有邻居。



§2.3 构造生成树

- msg复杂性
  因为每个结点在任一信道上发送M不会多于1次，所以每个信道上M至多被发送两次(使用该信道的每个处理器各1次)。
  在最坏情况下：M除第1次接收的那些信道外，所有其他信道上M被传送2次。因此，有可能要发送2m-(n-1)个msgs。这里m是系统中信道总数，它可能多达n(n-1)/2。
- 时间复杂性：O(D)　　D—网直径

2.构造生成树

对于flooding稍事修改即可得到求生成树的方法。

§2.3 构造生成树

①基本思想

- 首先，p_r发送M给所有邻居，p_r为根
- 当p_i从某邻居p_j收到的M是第1个来自邻居的msg时，p_j是p_i的双亲；若p_i首次收到的M同时来自多个邻居，则用一个comp事件处理自上一comp事件以来的所有已收到的msgs，故此时，p_i可在这些邻居中任选一个邻居p_j做双亲。
- 当p_i确定双亲是p_j时，发送<parent>给p_j，并向此后收到发来M的处理器发送<reject>msg

§2.3 构造生成树

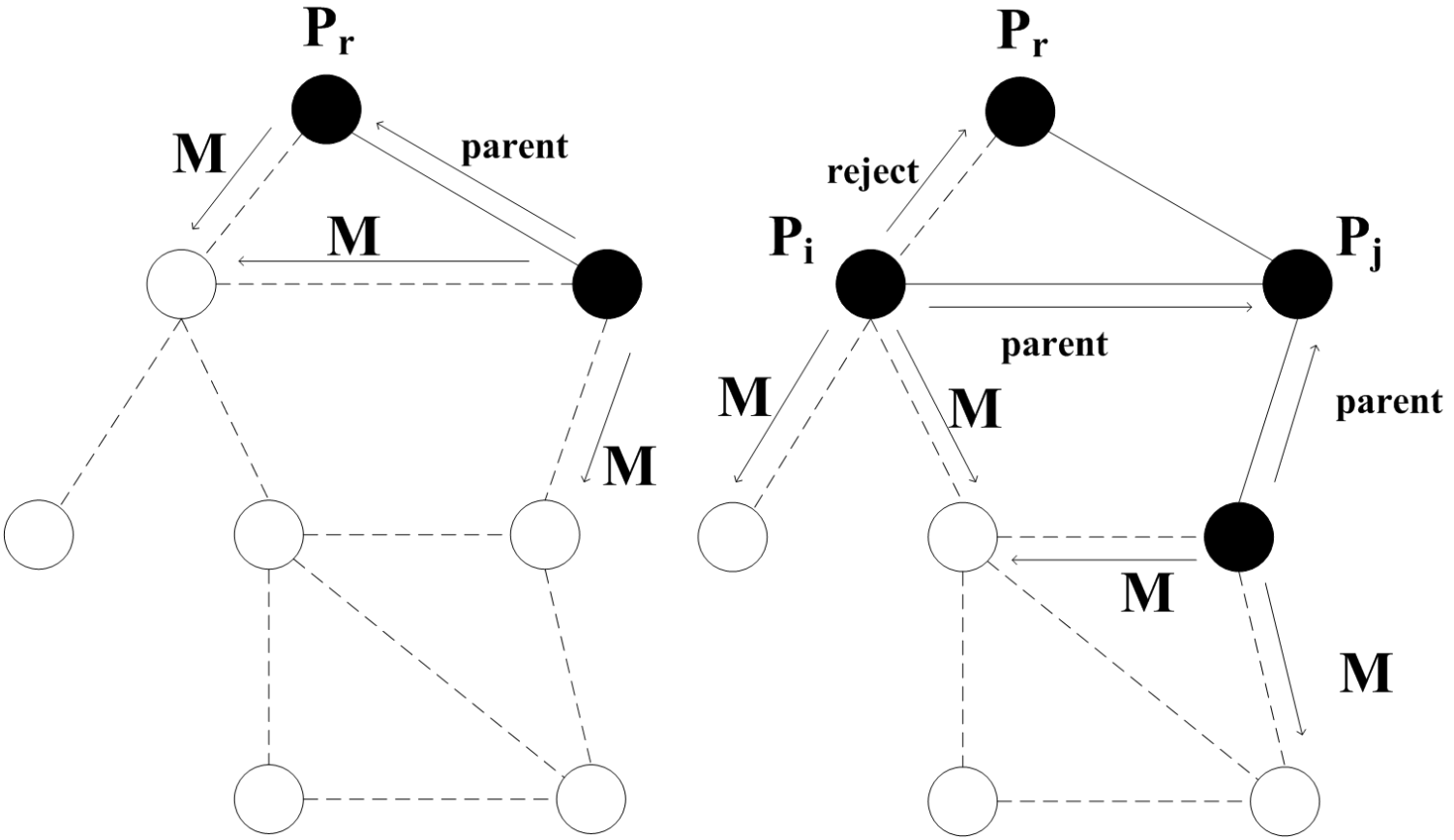
①基本思想

- 因为p_i收到p_j的M是第1个M，就不可能已收到其他结点的M，当然可能同时收到(说明p_i与这些邻居间不是父子关系，或说它们不是生成树中的边)；同时p_i将M转发给其余邻居，这些邻居尚未发M给p_i，或虽然已发M给p_i，但p_i尚未收到。
- p_i向那些尚未发M给p_i(或已发M但尚未到达p_i)的邻居转发M之后，等待这些邻居发回响应msg：<parent>或<reject>。那些回应<parent>的邻居是p_i的孩子。
- 当p_i发出M的所有接收者均已回应(<parent>或<reject>)，则p_i终止。将parent和children边保留即为生成树。

§2.3 构造生成树

②图示

p_i若认为p_j是其双亲，则p_i向p_r发出M，而p_r仍会向p_i发reject，但因为此前p_r向p_i发出过M，故p_i收到M时仍会向p_r发reject。(可以改进？)



§2.3 构造生成树

③算法： Alg2.2 构造生成树（code for p_i $0 \le i \le n-1$）

初值：parent=nil；集合children和other均为ϕ

upon receiving no message:

if i=r and parent=nil then { //根尚未发送M

send M to all neighbors;

parent:=i;} //根的双亲置为自己

upon receiving M from neighbor p_j:

if parent=nil then {//p_i此前未收到过M，M是p_i收到的第1个msg

parent:=j;

send <parent> to p_j; //p_j是p_i的双亲

send M to all neighbors except p_j;

}else //p_j不可能是p_i的双亲，p_i收到的M不是第1个msg

send<reject> to p_j;

upon receiving <parent> from neighbor p_j:

children:=children∪{ j }; //p_j是p_i的孩子，将j加入孩子集

if children∪other 包含了除parent外的所有邻居 then terminate;

upon receiving <reject> from neighbor p_j:

other:=other∪{ j }; //将j加入other，通过非树边发送的msg。

if children∪other包含了除p_i的双亲外的所有邻居 then terminate;

§2.3 构造生成树

④分析

Lemma2.6 在异步模型的每个容许执行中，算法2.2构造一棵根为p_r的生成树。(正确性)

Pf：算法代码告诉我们两个重要事实

a) 一旦处理器设置了parent变量，它绝不改变，即它只有一个双亲

b) 处理器的孩子集合决不会减小。

因此，最终由parent和children确定的图结构G是静止的，且parent和children变量在不同结点上是一致的，即若p_j是p_i的孩子，则p_i是p_j的双亲。

下述证明结果图G是根为p_r的有向生成树。

§2.3 构造生成树

- 为何从根能到达每一结点？(连通)

  反证：假设某结点在G中从p_r不可达，因网络是连通的，若存在两个相邻的结点p_i和p_j使得p_j在G中是从p_r可达的(以下简称p_j可达)，但p_i不可达。因为G里一结点从p_r可达当且仅当它曾设置过自己的parent变量(Ex2.4证明)，所以p_i的parent变量在整个执行中仍为nil，而p_j在某点上已设置过自己的parent变量，于是p_j发送M到p_i(line9)，因该执行是容许的，此msg必定最终被p_i接收，使p_i将自己的parent变量设置为j。矛盾！

§2.3 构造生成树

- 为何无环？(无环)

  假设有一环，$p_{i1},\ldots p_{ik}p_{i1}$，若$p_i$是$p_j$的孩子，则$p_i$在$p_j$第1次收到M之后第1次收到M。因每个处理器在该环上是下一处理器的双亲，这就意味着p_{i1}在p_{i1}第1次接收M之前第1次接收M。矛盾！

- 复杂性

  显然，此方法与淹没算法相比，增加了msg复杂性，但只是一个常数因子。在异步通信模型里，易看到在时刻t，消息M到达所有与p_r距离小于等于t的结点。因此有：

Th2.7 对于具有m条边和直径D的网络，给定一特殊结点，存在一个msg复杂性为O(m)，时间复杂性为O(D)的异步算法找到该网络的一棵生成树。

§2.3 构造生成树

Alg2.2在同步模型下仍可工作。其分析类似于异步情形。然而，与异步不同的是，它所构造的生成树一定是一棵广度优先搜索(BFS)树。

Lemma2.8 在同步模型下，Alg2.2的每次容许执行均构造一棵根为p_r的BFS树。

Pf：对轮t进行归纳。即要证明：在第t轮开始时刻
①根据parent变量构造的图G是一棵包括所有与p_r距离至多为t-1结点的BFS树；
②而传输中的消息M仅来自于与p_r距离恰为t-1的结点。
由此构造的树是一棵根为p_r的BFS

§2.3 构造生成树

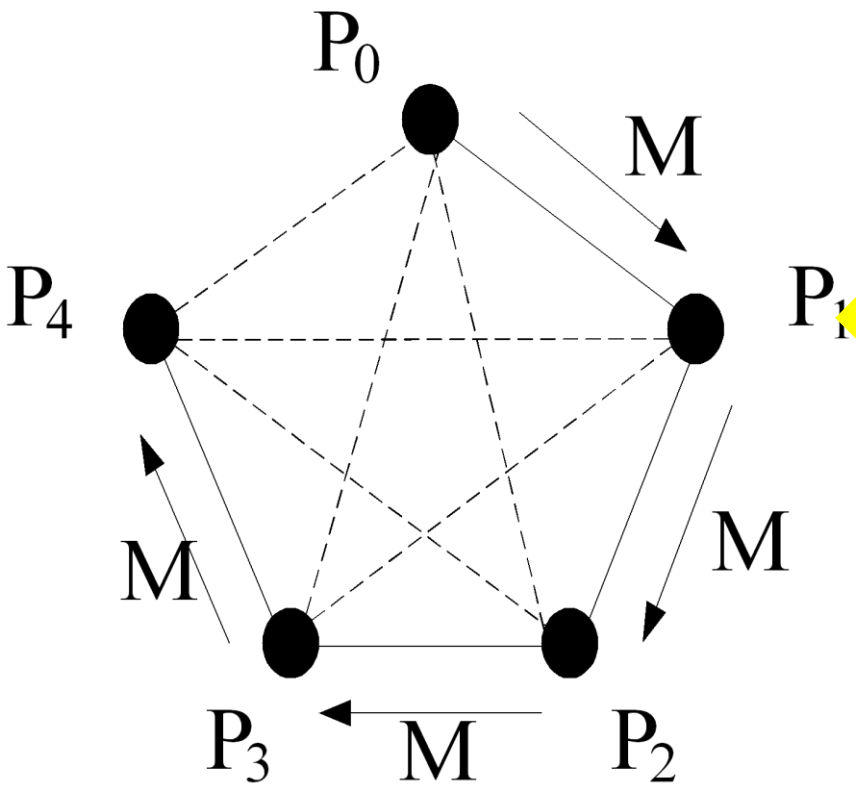
当t=1时，所有parent初值为nil，M从p_r传出。

假设引理对第t-1≥1轮为真，在该轮里，从距离 t-2的结点传出的M被接收，任何接收M的结点与p_r的距离不超过t-1(恰为t-1或更短)，那些parent值非空的接收结点显然与p_r的距离不超过t-2，他们既不改变parent的值也不转发M；而与p_r距离为t-1的结点在t-1轮里收到M，因为它们的parent为nil，故将其置为合适的双亲并转发M。距离p_r大于t-1的结点不会收到M，因此也不会转发M。因此有如下定理：

Th2.9 对于具有m条边直径为D的网络，给定一个特殊结点，存在一个同步算法在msg复杂性为O(m)，时间复杂性为O(D)内找到一棵BFS树。

§2.3 构造生成树

- 异步系统里，Alg2.2能构造BFS树？
  例如，考虑5个顶点的完全连通图



- **P_0为根，假定M消息按P_0到p_1，P_1到P_2，P_2到P_3，P_3到P_4的次序快速传播，而M在其它路径上传播较慢。结果生成树是从P_0到P_4的链，它不是BFS树**
- **虽然此图直径D=1，生成树的高度d=4，但是算法的运行时间仍然为O(D)而不是O(d)。**
  理解：P_0到P_4的M在1个时间内到达，即P_0->P_1->P_2->P_3->P_4的时间之和不超过1。

§2.3 构造生成树

- 信息的请求和收集

  将算法2.2(求生成树)和汇集算法组合即可完成。组合算法的时间复杂性在同步和异步模型中不同,设网是完全图

  – 求生成树算法：同步和异步均为　消息复杂性O(m)

    时间复杂性O(D)

  – 汇集算法：同步和异步均有　msg n-1

    time d //生成树高

  – 组合算法

    ①同步：组合算法的msg复杂性O(m+n)；BFS树中，d=1， d≤D，故时间复杂性O(D+d)=O(D)=O(1)。

    ②异步：组合算法的msg复杂性O(m+n)；生成树高d=n-1，所以时间复杂性O(D+d)=O(d)=O(n)。

    1-time复杂性的组合算法T(n)=O(D)。

§2.4 构造DFS生成树(指定根)

构造DFS树时每次加一个结点，而不像Alg2.2那样，试图在树上同时增加同一层的所有结点。

Alg2.3 构造DFS生成树，为P_r为根

Code for processor P_i, $0 \le i \le n-1$

```
var     parent:init nil;
        children:       init Φ;
        unexplored:     init all the neighbors of Pi
                                    //未访问过的邻居集
1: upon receiving no msg:
2:   if (i=r) and (parent=nil) then {  //当Pi为根且未发送M时
3:      parent := i;  //将parent置为自身的标号
4:      ∀Pj ∈ unexplored;
5:      将Pj从unexplored中删去; //若Pr是孤立结点,4-6应稍作修改
6:      send M to Pj;
     }//endif
```

§2.4 构造DFS生成树(指定根)

7: upon receiving M from neighbor P_j:
8: if parent=nil then { //P_i此前未收到M
9: parent := j; //P_j是P_i的父亲
10: 从unexplored中删P_j
11: if unexplored ≠ Φ then {
12: P_k ∈ unexplored;
13: 将P_k从unexplored中删去；
14: send M to P_k;
15: } else send <parent> to parent;
//当P_i的邻居均已访问过，返回到父亲
16: }else send <reject> to P_j; //当P_i已访问过时

§2.4 构造DFS生成树(指定根)

17: upon receiving <parent> or <reject> from neighbor P_j:

18: if received <parent> then add j to children;

//P_j是P_i的孩子

19: if unexplored = Φ then { //P_i的邻居均已访问

20: if parent ≠ i then send <parent> to parent;

//P_i非根，返回至双亲

21: terminate; //以P_i为根的DFS子树已构造好！

22: }else { //选择P_i的未访问过的邻居访问之

23: $P_k \forall \in$ unexplored;

24: 将P_k从unexplored中删去；

25: send M to P_k;

}

§2.4 构造DFS生成树(指定根)

引理2.10 在异步模型里的每个容许执行，Alg2.3构造一棵以P_r为根的DFS树。证明留作练习。

Th2.11 对于一个具有m条边，n个结点的网络，以及给定的特殊顶点，存在一个时间复杂性和消息复杂性均为O(m)的异步算法找到一棵DFS树。

Pf：每个结点在其邻接边上至多发送M一次，每个结点至多生成一个msg(<reject>或<parent>)作为对每个邻接边上收到的M的响应。因此Alg2.3至多发送4m个消息(其实大部分没有4倍)，即算法的msg复杂性为O(m)。

时间复杂性证明留作练习。

如何改进使msg的复杂性不是4m？

注意：上述算法msg复杂性较好，但时间复杂性太差。可降至O(n)。

下次继续！

第二部分 分布式算法

第四次课

中国科学技术大学计算机系
国家高性能计算中心（合肥）

§2.5 不指定根时构造DFS生成树

算法2.2和2.3构造连通网络的生成树时，必需存在一个特殊的结点作为启动者(Leader)。当这样的特殊结点不存在时，如何构造网络的一棵生成树？但本节算法须假定：各结点的标识符唯一，不妨设是自然数，§3.2仍需此假定。

不指定根构造DFS生成树，和后面的领导者选举问题一样，都是破对称问题。

1. 基本思想

– 每个结点均可自发唤醒，试图构造一棵以自己为根的DFS生成树。若两棵DFS树试图链接同一节点(未必同时)时，该节点将加入根的id较大的DFS树。

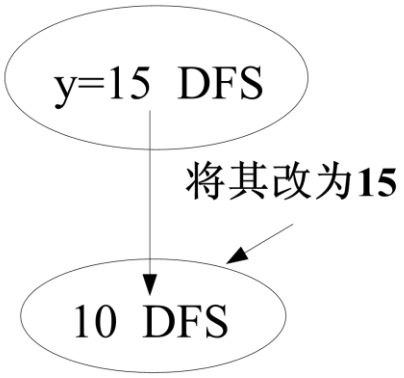
– 为了实现上述思想，须做：

  - 每个结点设置一个leader变量，其初值为0，当P_i唤醒自己时，$leader_i=id_i$；

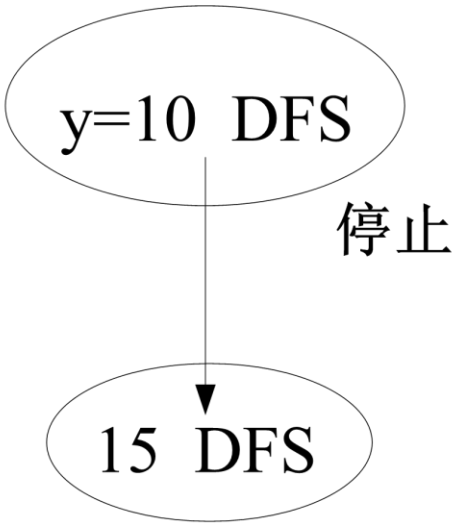
§2.5 不指定根时构造DFS生成树

- 当一结点自发唤醒时，它将自己的id(leader)发送给某一邻居；
- 当一结点P_i收到来自邻居P_j的标识符y时，P_i比较y和$leader_i$：

①若$y>leader_i$，则y可能是具有最大标识符结点的DFS子树的标记。因此，将$leader_i$置为y，并令P_j是P_i的双亲。从P_i开始继续生成标记为y的DFS树。Note：要修改原P_i所在的DFS子树中所有结点的leader。



§2.5 不指定根时构造DFS生成树



② 若$y<leader_i$，则标记为y的DFS树中最大id(y)小于目前所看到的最大标识符。此时无须发送msg，停止构造标记为y的DFS。等待最终某个更大的id的leader消息到达标记为y的树中结点时，再将该节点连接到树中。(至少标记为$leader_i$的msg会到达标记为y的树)

③ 若$y=leader_i$，则P_i已属于标记y的DFS树中。

§2.5 不指定根时构造DFS生成树

2. 算法

Alg2.4 构造生成树，不指定根

Code for Processor P_i $0 \le i \le n-1$

```
Var parent:       init nil;
    leader:       init 0;
    children:     int ϕ;
    unexplored: init all neighbors of Pi;
1: upon receiving no msg:  //wake up spontaneously
2:    if parent = nil then {
          //若非空，则Pi在某子树上，则Pi失去竞选机会
3:       leader := id;  parent := i;//试图以自己为根构造树
4:          Pj ∈ unexplored;
5:       将Pj从unexplored中删去；
6:       send <leader> to pj;
      }
```

§2.5 不指定根时构造DFS生成树

想像：有m个人竞选领袖，id是他自身的素质分，不想竞争人的id不参与比较。

竞争规则：将自己的id(如讲演片)传递给一个熟悉的人，由他再传给另一人(一次只能送一人。)

7: upon receiving <new-id> from neighbor P_j:

8: if leader<new-id then { //将P_i所在树合并到P_j所在树中

9: leader := new-id; parent := j;

//令P_i的双亲为P_j，可能是修改，而非对nil赋值

//并不一定能停止较差的竞选者传播msg

10: unexplored := all the neighbors of P_i except P_j;

//重置未访问的邻居集

11: if unexplored ≠ϕ then {

//因为new-id大，使原P_i所在DFS树修改各自id

12： $P_k \in$ unexplored;

13: 将P_k从unexplored中删去；

§2.5 不指定根时构造DFS生成树

14: send <leader> to P_K;
15: }else send <parent> to parent; // unexplored =ϕ
}else // leader≥new-id
16: if leader=new-id then send <already> to P_j;
//表示自己已经传出过此录像带，无需重传。已在同一树中
//若leader>new-id，则new-id所在DFS停止构造
//以前收到的竞选者优于new-id，不传送，使之停止传播。

17: upon receiving <parent> or <already> from neighbor P_j:
18: if received <parent> then add j to children;
19: if unexplored= ϕ then { //无尚未访问的邻居
20: if parent≠i then send <parent> to parent //返回
21: else terminates as root of the DFS tree; //根终止
22: }else { //有尚未访问的邻居

§2.5 不指定根时构造DFS生成树

23: $\forall P_k \in$ unexplored;
24: 将P_k从unexplored中删去；
25: send <leader> to P_k;
}

3. 分析：
   - 只有生成树的根显式地终止，其它结点没有终止，始终在等待msg。但可修改此算法，使用Alg2.1从根结点发送终止msg
   - 正确性

     该算法比前面的算法更复杂，这里只给出粗略的证明。

     设P_m是所有自发唤醒结点中标识符最大者，其标识符为id_m。消息id_m总是被传播，而一旦一个结点收到id_m，则该节点(P_m除外)上所有msgs被忽略。因为消息id_m的处理和Alg2.3求DFS树一致，因此产生的parent和children变量的设置是正确的。因此有：

§2.5 不指定根时构造DFS生成树

Lemma2.12 设P_m是所有自发唤醒结点中具有最大标识符的结点。在异步模型的每次容许执行里，算法2.4构造根为P_m的一棵DFS树。

Note：因为在容许执行中，网络里的所有自发唤醒结点中最大标识符结点最终会自发启动，故建立的DFS树的根是P_m

可通过广播算法从P_m发出终止msg，即使不广播，所有非P_m结点最终也会因为收到P_m的标识符而停止。因此，不可能构造一棵根不是P_m的生成树。

Lemma2.13 在异步模型的每个容许执行里，只有一个处理器终止作为一生成树的根。

§2.5 不指定根时构造DFS生成树

– 复杂性

定理：对于一个具有m条边和n个节点的网络，自发启动的节点共有p个，其中ID值最大者的启动时间为t，则算法的消息复杂度为$O(pn^2)$，时间复杂度为O(t+m)。

消息复杂性：简单地分析，最坏情况下，每个处理器均试图以自己为根构造一棵DFS树。因此，Alg2.4的msg复杂性至多是Alg2.3的n倍：O(m*n)

时间复杂性：类似于Alg2.3的msg复杂性O(m)。

§2.5 不指定根时构造DFS生成树

Ex.

2.1 分析在同步和异步模型下，convergecast算法的时间复杂性。

2.2 证明在引理2.6中，一个处理器在图G中是从P_r可达的，当且仅当它的parent变量曾被赋过值

2.3 证明Alg2.3构造一棵以P_r为根的DFS树。

2.4 证明Alg2.3的时间复杂性为O(m)。

2.5 修改Alg2.3获得一新算法，使构造DFS树的时间复杂性为O(n)。

补充

§ 2.6 小结

§补充 构造最小生成树

考虑每个信道都有权重，代表了信道的通信成本，这样最小生成树能够使得执行广播算法总开销最小。

假设每个信道的权重都已存储在其两端的节点的局部内存中。

基本思想：

每个节点都有一个所属树的变量编号，用来判断两个节点是否属于一棵树。刚开始时，每个节点独立成一棵树，其ID值为树的编号，每棵树并发的搜索自己权值最小的出边，并把这些边加入到生成森林中，同时几棵树也就合并为一棵树，取其中编号较大的一个为合并之后的树的编号，更新树中各节点的树编号变量。直至所有的树都被合并为一棵树，此即为最小生成树。

§补充 构造最小生成树

可能出现的4个问题：

- 产生环

  可以证明，如果边的权值不相等，那么图G只有唯一的最小生成树。

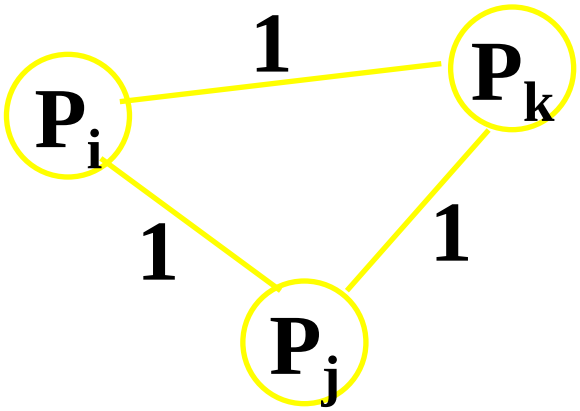
  利用三元组，将边表述为

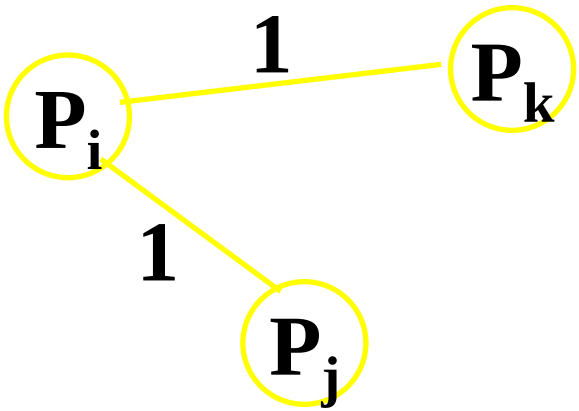
  {1,i,j},{1,j,k},{1,i,k}

  当权相同时，按照i,j,k的字典序来排序

  假设i<j<k,则有{1,i,j}<{1,i,k} <{1,j,k}

  这样就可以避免出现环





§补充 构造最小生成树

可能出现的4个问题：

- 不平衡

由于异步系统的消息延迟，可能会出现不平衡构造树，从而导致更多的消息。

极端的例子：每次都是一个节点数目很多的树，去合并一个单节点的树，这样对于节点数目大的树来说，每次查找权最小的出边花去的代价太大。

解决办法：给每棵树设置一个level层变量，刚开始时，单节点树的level为0，如果一棵树的权值最小出边所连的另一棵树的level变量大于或等于自己的level变量，则两棵树通过这条边合并，否则等待。所合并的树的level取决于两棵树中level较大者，若两棵树level相同为L，则新合并的树的level=L+1。

(level的含义：好比游戏中的练级)

§补充 构造最小生成树

可能出现的4个问题：

- 错误判断出边

异步系统中，可能出现在判断一条边是否是出边时，边连接的两点已经处在一棵树当中了，但由于消息延迟导致树编号的更新消息还未传到，因而导致因为两节点的树编号不同而产生的错误。

解决方法：

在合并之前，树中节点按边的权值由小到大的顺序开始探测此边是否是出边，当确认某条边是出边之后便停止探测其他边，并将结果敛播给根节点，由根节点确定通过哪一条边进行合并。

当两棵树合并时，合并后的树的编号以及根节点取level值较大的那棵树的根和树编号。若level值相同，取树编号较大的树的根和树编号。然后由根执行广播操作，更新所有树编号和level值，并通知寻找权值最小的出边。

（确定Pi和Pj是否属于一棵树）

①如果P_i和P_j编号相同，则肯定属于一棵树；

②如果编号不同，但P_j的level和P_i一样大，则肯定不属于一棵树，因为每棵树在一层中不可能有两个树编号，且当P_i在寻找其权值最小的出边的时候，其树编号是最新的；

③如果P_j的树编号与P_i不同且P_j的层数严格小于P_i，那么节点P_j就延迟回答P_i直到他更新到其level值上升到至少和P_i一样大。

(可以证明，这个新增的延迟将不会构成节点间的死锁)

§补充 构造最小生成树

可能出现的4个问题：

- 存在干扰

  不同层的邻接树同时寻找权值最小的出边有可能发生相互干扰。具体来说，当层低的树T合并到层高的树T’，而T’正在确定其权值最小出边时可能会发生此情况。

§补充 构造最小生成树

算法框架：

Code for Every Tree T

Begin

While (系统中的子树个数大于1) do

(1) 根节点广播ID(T)和level(T)，命令各节点寻找权值最小的边；

(2) 按权值由小到大顺序寻找权值最小的出边，找到之后敛播给根节点；

(3) 根节点收到所有节点的权值最小出边后，确定整棵树的最小出边e，边e连接树T’。

/*level(T)≤level(T’)*/

(4) T’’=T∪T’ ∪e

(5) if level(T’)>level(T) then

(5.1) ID(T’’)←ID(T’)

(5.2) level(T’’)←level(T’)

else // level(T’)=level(T)

(5.3) ID(T’’)←max{ID(T), ID(T’)}

(5.4) level(T’’)←level(T’’)+1

end if

end while

end

§补充 构造最小生成树

消息复杂度：

每个level为k的树中的节点数至少为2^k，level最大为log(n)(n为节点数)。因为每个节点都从边的权值从小到大开始探测此边是否是出边，当确定某边是出边后就停止探测并敛播给根节点，直到根确认这条边是这棵树的最小边，并进行合并操作，并更新level值，然后继续探测。

消息分两类：①每个节点探测自己的一条边是否是出边而得到否认消息，这些消息数目为O(m)；②每一层中在树T中各种消息的传递，其消息数目为O(|T|)，|T|表示树的节点数，此总数目为

$$\sum_{k=0}^{\log n} \sum_{level(T)=k} |T| \le \sum_{k-0}^{\log n} n = O(n \log n)$$

故消息总数为O(m+nlogn)

§补充 构造最小生成树

时间复杂度：

系统中所有节点，其所在的树的level值都变成k 时所需的时间为O(kn)，又因为k的最大值为log(n)，所以总时间为O(nlogn)。

§2.6 小结

Introduction to distributed alg

- 分类：
  - 单源alg：一个启动者。又称centralized alg。
  - 多源alg：任意进程(结点)子集均可是启动者，又称decentralized alg
  - 启动者(initiator)：自发地执行局部算法，即由一内部事件激发其执行
  - 非启动者：由接收一个msg(外部事件)触发其执行局部进程。
- 复杂性：
  - Msg复杂性：msg总数目
  - Bit复杂性：发送msg中bit的总数目，当msg在发送过程中其长度随时间增长时

§2.6 小结

– 时间复杂性

① 一个分布式算法的时间复杂性是满足下述两个假定的一个计算所耗费的最大时间

T1：一个进程在零时间内可计算任何有限数目的事件

T2：一个msg的发送和接受之间的时间至多为1个时间单位

缺点:针对一算法的所有计算，其结果可能是极不可能发生的计算。

② 一个分布式算法的one-time复杂性是满足下述假定的一个计算的最大时间

O1：同T1

O2：发送和接收一个msg之间的时间恰好是1个单位时间

缺点：某些计算可能被忽略，而其中可能有极其耗时的计算

§2.6 小结

表面上，1-time复杂性至少等于时间复杂性，因为T2假定下的最坏时间不会高于O2假定下的时间。但事实并非如此，而往往O1和O2假定之下的1-time复杂性是前一种时间复杂性的一个下界。

例如：在echo算法里1-time复杂性是O(D)，时间复杂性是Θ(N)，即使直径为1的网络。

③ 两种复杂性的折中：α-复杂性

假定每个msg延迟介于α-1之间(α≤1常数)

对echo 算法α复杂性为O(min(N, D/ α))

④ 概率分析：msg延迟服从某种概率分布，由此可获得精确的时间复杂性度量

§2.6 小结

⑤ 基于msg chains的分析

任何计算中最长消息链的长度。

链上msgs：$m_1,m_2,\ldots m_k$序列中，m_i因果关系领先于m_{i+1}。

- 先验知识：邻居的id，全局id等。链路FIFO假定等

下次继续！

第三章 环上选举算法

汪 炀

本章提纲

- Leader选举问题
- 匿名环
- 异步环
- 同步环

- 问题
  在一组处理器中选出一个特殊结点作为leader
- 用途

① 简化处理器之间的协作；
有助于达到容错和节省资源。
例如，有了一个leader，就易于实现广播算法

② 代表了一类破对称问题。
例如，当死锁是由于处理器相互环形等待形成时，可使用选举算法，找到一个leader并使之从环上删去，即可打破死锁。

§3.1 leader选举问题

Leader选举问题：

问题从具有同一状态的进程配置开始，最终达到一种配置状态。每个处理器最终确定自己是否是一个leader，但只有一个处理器确定自己是leader，而其他处理器确定自己是non-leader。

算法的作用：

如果要执行一个分布式算法，且没有一个优先的优选人做为算法的初始进程，就要进行进程选举。(例如指定根的DFS树的生成问题)

§3.1 leader选举问题

选举算法的定义：

（1）每个处理器具有相同的局部算法；

（2）算法是分布式的，处理器的任意非空子集都能开始一次计算；

（3）每次计算中，算法达到终止配置。在每一可达的终止配置中，只有一个处理器处于领导人状态，其余均处于失败状态。**（此项有时可以弱化）**

一个算法解决了leader选举问题需满足(根据形式化模型)：

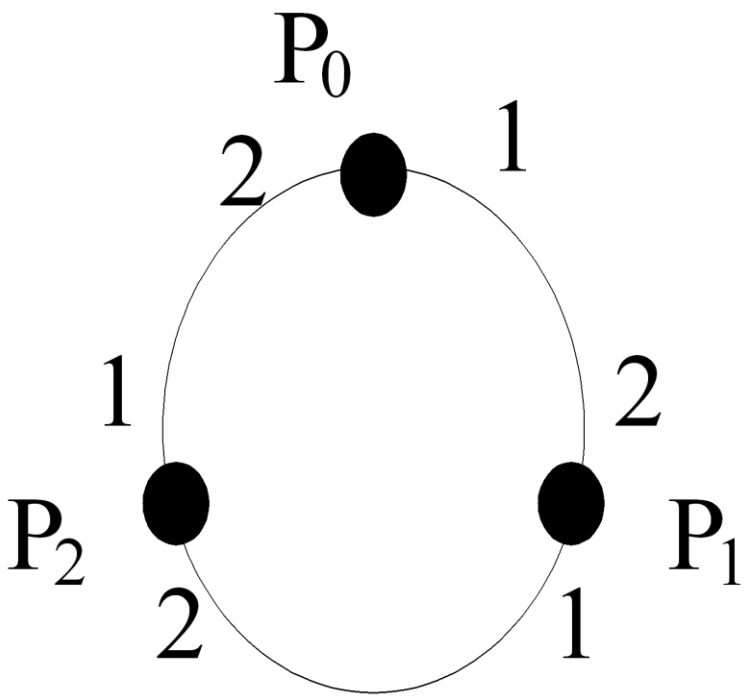
① 终止状态被划分为两类：选中和未选中状态。一旦一个处理器进入选中(或未选中)状态，则该处理器上的转换函数将只会将其变为相同的状态；

② 在每个容许执行里，只有一个处理器进入选中状态，其余处理器进入非选中(non-selected)状态。

本章只讨论系统的拓扑结构是环的情况。

§3.1 leader选举问题

- 环的形式化模型
  对每个i，0≤i ≤n-1，
  P_i到P_{i+1}的边标号为1，称为左(顺时针)
  P_i到P_{i-1}的边标号为2，称为右(逆时针)
  这里的标号加减均是mod n的



环网络之所以吸引了如此多的研究，是因为它们的行为易于描述；且从环网络推导出的下界，可应用于具有任意拓扑结构的网络算法设计

§3.2 匿名环（anonymous）

- 匿名算法：若环中处理器没有唯一的标识符，则环选举算法是匿名的。更形式化的描述：每个处理器在系统中具有相同的状态机，在这种算法里，msg接收者只能根据信道标号来区别。
- （一致性的）uniform算法：若算法不知道处理器数目，则算法称之为uniform，因为该算法对每个n值看上去是相同的。
- non-uniform算法：算法已知处理器数目n
- 形式化描述：在一个匿名、一致性的算法中，所有处理器只有一个状态机；在一个匿名、非一致性的算法中，对每个n值（处理器数目）都有单个状态机，但对不同规模有不同状态机，也就是说n可以在代码中显式表达。

§3.2 匿名环（anonymous）

- 对于环系统，不存在匿名的选举算法。
  为简单起见，我们只证明
  非均匀（非一致性）算法：非均匀算法（n已知）的不可能性=>均匀（n未知）算法的不可能性。
  Ex3.1 证明同步环系统中不存在匿名的、一致性的领导者选举算法。

  同步算法：同步算法的不可能性=>异步算法的不可能性。（同步是异步的一种特例）
  Ex3.2 证明异步环系统中不存在匿名的领导者选举算法。

§3.2 匿名环

- 同步算法的不可能性

在同步系统中，一个算法以轮的形式进行。每轮里所有待发送msg被传递，随后每个处理器进行一步计算。

一个处理器的初始状态包括在outbuf里的任何msg。这些消息在第一轮里被传递到某处理器的左和右邻居。

不可能性：

①在一个匿名环中，处理器间始终保持对称，若无某种初始的非对称(如，标识符唯一)，则不可能打破对称。在匿名环算法里，所有处理器开始于相同状态。

②因为他们执行同样的程序(即他们的状态机相同)，在每轮里各处理器均发送同样的msg，所以在每一轮里各处理器均接收同样的msg，改变状态亦相同。

因此，若选中一个处理器，则其他所有处理器亦被选中。因此，不可能有一个算法在环中选中单个处理器为leader。

§3.2 匿名环

假设R是大小为n>1的环（非均匀），A是其上的一个匿名算法，它选中某处理器为leader。因为环是同步的且只有一种初始配置，故在R上A只有唯一的合法执行。

- Lemma3.1 在环R上算法A的容许执行里，对于每一轮k，所有处理器的状态在第k轮结束时是相同的。

Pf. 对k用归纳法

K=0(第一轮之前)，因为处理器在开始时都处在相同的初始状态，故结论是显然的。

设引理对k-1轮成立。因为在该轮里各处理器处在相同状态，他们都发送相同的消息m_r到右边，同样的消息m_l到左边，所以在第k轮里，每处理器均接收右边的m_l，左边的m_r。因此，所有处理器在第k轮里接收同样的消息，又因为它们均执行同样的程序，故第k轮它们均处于同样的状态。

§3.2 匿名环

上述引理蕴含着：若在某轮结束时，一个处理器宣布自己是leader(进入选中状态)，则其它处理器亦同样如此，这和A是一个leader选举算法的假定矛盾！因此证明：

- Th3.2 对于同步环上的leader选举，不存在非均匀的匿名算法。

＋　　＋

同步环→异步环　　非一致性→一致性算法

↓↓

对于环系统，不存在匿名的选举算法

§3.3 异步环

本节将讨论异步环上leader选举问题的msg复杂性上下界。

由Th3.2知，对环而言没有匿名的leader选举算法存在。因此以下均假定处理器均有唯一标识符。

当一个状态机(局部程序)和处理器P_i联系在一起时，其状态成分变量id_i被初始化为P_i的标识符的值，故各处理器的状态是有区别的。

- 环系统：通过指派一个处理器列表按顺时针(从最小标识符起)指定环。注意是通过id排列，不是通过P_i的下标i来排列($0 \le i \le n-1$)，假定id_i是P_i的标识符。（因为下标i通常是不可获得的）

§3.3 异步环

- 在非匿名算法中，均匀（一致性）和非均匀（非一致性）的概念稍有不同
  ① 均匀算法：每个标识符id，均有一个唯一的状态机，但与环大小n无关。而在匿名算法中，均匀则指所有处理器只有同一个状态。（不管环的规模如何，只要处理器分配了对应其标识符的唯一状态机，算法就是正确的。）
  ② 非均匀算法：每个n和每个id均对应一个状态机，而在匿名非均匀算法中，每个n值对应一个状态机。（对每一个n和给定规模n的任意一个环，当算法中每个处理器具有对应其标识符的环规模的状态机时，算法是正确的。）

下面将讨论msg复杂性：$O(n^2)$→$O(n\log n)$ →$\Omega(n\log n)$

下界

§3.3.1 一个$O(n^2)$算法

Le Lann、Chang和Roberts给出，LCR算法

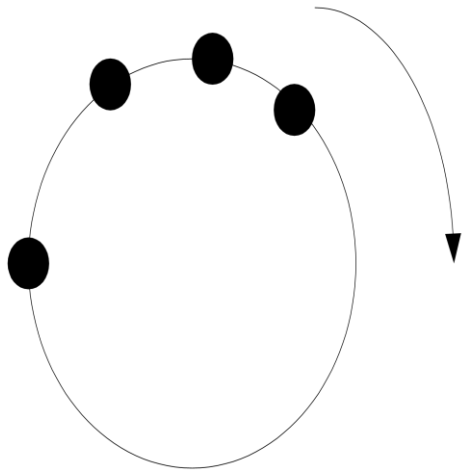
- 基本思想
  ① 每个处理器P_i发送一个msg(自己的标识符)到左邻居，然后等其右邻居的msg
  ② 当它接收一个msg时，检验收到的id_j，若$id_j>id_i$，则P_i转发id_j给左邻，否则没收id_j(不转发)。

§3.3.1 一个O(n²)算法

③ 若某处理器收到一个含有自己标识符的msg，则它宣布自己是leader，并发送一个终止msg给左邻，然后终止。

④ 当一处理器收到一个终止msg时，向左邻转发此消息，然后作为non-leader终止。

因为算法不依赖于n，故它是均匀的。



i—表示id 单向

§ 3.3.1 一个O(n^2)算法

Code for P_i

```
init var: asleep←true, id ←I
Begin
 While (receiving no message) do
  (1) if asleep do
      (1.1) asleep←false
      (1.2) send <id> to left-negihbor
    end if
 End while
 While (receiving <i> from right-neighbor) do
  (1) if id<<i> then send <i> to left-neighbor
       end if
  (2) if id=<i> then
      (2.1) send <Leader,i> to left-neighbor
      (2.2) terminates as Leader
    end if
 End while
 While (receiving <Leader,j> from right-neighbor) do
  (1) send <Leader,j> to left-neighbor
  (2) terminates as non-Leader
 End while
end
```

§3.3.1 一个O(n^2)算法

- 分析

① 正确性

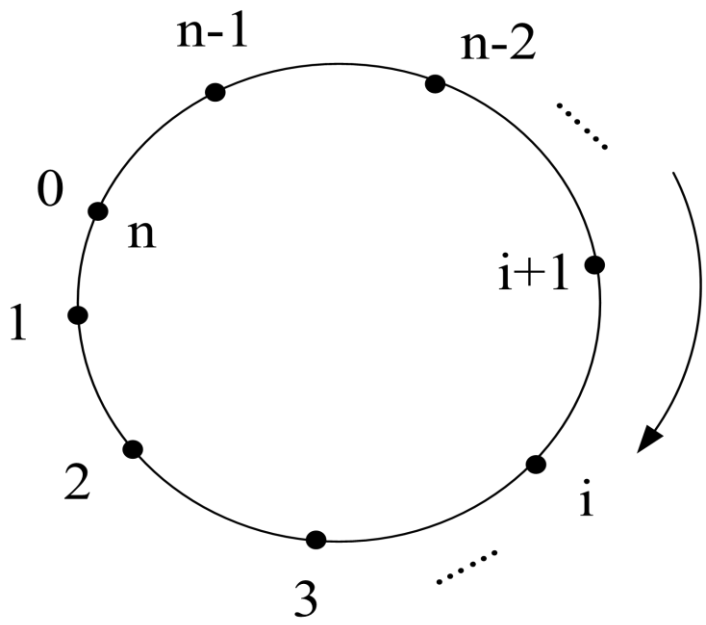
在任何容许执行里，只有最大标识符id_{max}不被没收，故只有具有id_{max}的处理器接受自己的标识符并宣布是leader，其他处理器不会被选中，故算法正确。

② msg复杂性

在任何容许执行里，算法绝不会发送多于O(n^2)个msgs，更进一步，该算法有一个容许执行发送O(n^2)个msgs：

§3.3.1 一个O(n²)算法

考虑处理器标识符为0，1，…，n-1构成的环，其次序如右图：



在这种配置里，id=i的处理器的msg恰好被发送i+1次，即发送到i-1,i-2,…,1,0，直到n-1时没收。因此，msg总数为：

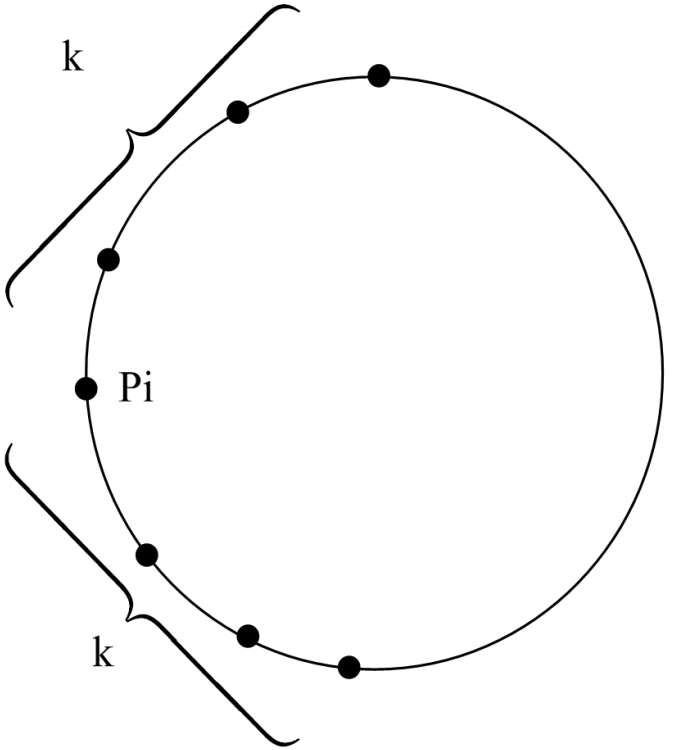
$$n+\sum_{i=0}^{n-1}(i+1)=\theta(n^2)$$ //前一项为终止msg

§3.3.2 一个O(*n*lg*n*)算法

仍然是绕环发送id，但使用更聪明的方法。保证最大id在环上周游且返回。

- k邻居

  一个处理器P_i的k邻居是一个处理器集合：该集合中的任一处理器与P_i在环上的距离至多是k，一个处理器的k-邻居集中恰用2k+1个处理器。



k=3，共有7个结点

§3.3.2 一个O(*n*lg*n*)算法

- 基本思想

  算法按阶段执行，在第*l*阶段一个处理器试图成为其2^l-邻接的临时leader。只有那些在*l*-th阶段成为临时领袖的处理器才能继续进行到(*l*+1)th阶段。因此，*l*越大，剩下的处理器越少。直至最后一个阶段，整个环上只有一个处理器被选为leader。

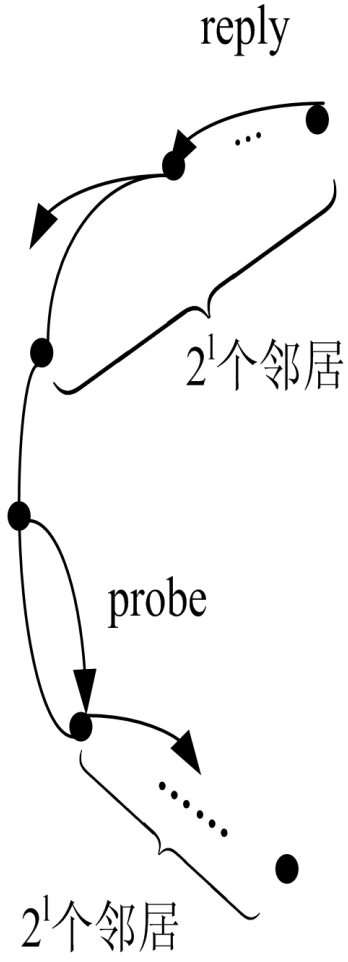
- 具体实现

  ① phase0: 每个结点发送1个probe消息(其中包括自己的id)给两个1-邻居，若接收此msg的邻居的id大于消息中的id，则没收此msg；否则接收者发回一个reply msg。若一个结点从它的两个邻居收到回答msg reply，则该结点成为phase0里它的1-邻居的临时leader，此结点可继续进行phase1。

§3.3.2 一个O(*n*lg*n*)算法

② phase *l*：在*l*-1阶段中成为临时leader的处理器P_i发送带有自己id的probe消息至它的2^l邻居。若此msg中的id小于左右两个方向上的$2*2^l$个处理器中任一处理器的id，则此msg被没收。若probe消息到达最后一个邻居而未被没收，则最后一个处理器发送reply消息给P_i，若P_i从两个方向均接收到reply消息，则它称为该阶段中2^l邻居的临时leader，继续进入下一阶段。

③ 终止：接收到自己的probe消息的结点终止算法而成为leader，并发送一个终止msg到环上。



§3.3.2 一个O(*n*lg*n*)算法

④ 控制probe msg的转发和应答

probe消息中有三个域：<prob, id, *l*, hop>

id-标识符

l-阶段数

hop-跳步计数器：初值为0，结点转发probe消息时加1.

若一结点收到的probe消息时，hop值为2^l，则它是2^l邻居中最后一个处理器。若此时msg未被没收也不能向前转发，而应该是向后发回reply消息。

§3.3.2 一个O(*n*lg*n*)算法

- 算法：Alg3.1 异步leader选举

```
var asleep init true;
upon receiving no msg:
   if  asleep then{
       asleep:=false;//每个结点唤醒后不再进入此代码
       send<probe, id, 0, 0> to left and right;
   }
upon receiving <probe, j, l, d> from left (resp, right):
   if(j=id) then //收到自己id终止，省略发终止msg
         terminate as the leader;
  if(j>id) and (d<2^l) then   //向前转发probe msg
                send <probe, j, l, d+1> to right (resp, left)
```

§3.3.2 一个O(*n*lg*n*)算法

if(j>id) and (d≥2^l)then//到达最后一个邻居仍未没收

send <reply, j, *l* > to left(resp, right) // 回答

//若j<id, 则没收probe消息

upon receiving <reply ,j , *l*> from left (resp, right):

if j≠id then

send<reply, j, l> to right (resp, left); //转发reply

else //j=id时，P_i已收到一个方向的回答msg

if already received <reply, j, *l*> from right (resp, left)

then//也收到另一方向发回的reply

send <probe, id, l+1, 0> to left and right;

//P_i是phase *l*的临时leader，继续下一阶段

§3.3.2 一个O(*n*lg*n*)算法

- 分析

① 正确性：因为具有最大id的处理器的probe消息是不会被任何结点没收的，所以该处理器将作为leader终止算法；另一方面，没有其他probe消息能够周游整个环而不被吞没。因此，最大id的处理器是算法选中的唯一的leader。

② msg复杂性（最坏情况下）

在phase l 里：

- 一个处理器启动的msg数目至多为：$4*2^l$
- 有多少个处理器是启动者呢？
  - l =0，有n个启动着（最多）
  - l≥1，在l-1阶段结束时成为临时leader的节点均是启动者

§3.3.2 一个O(*n*lg*n*)算法

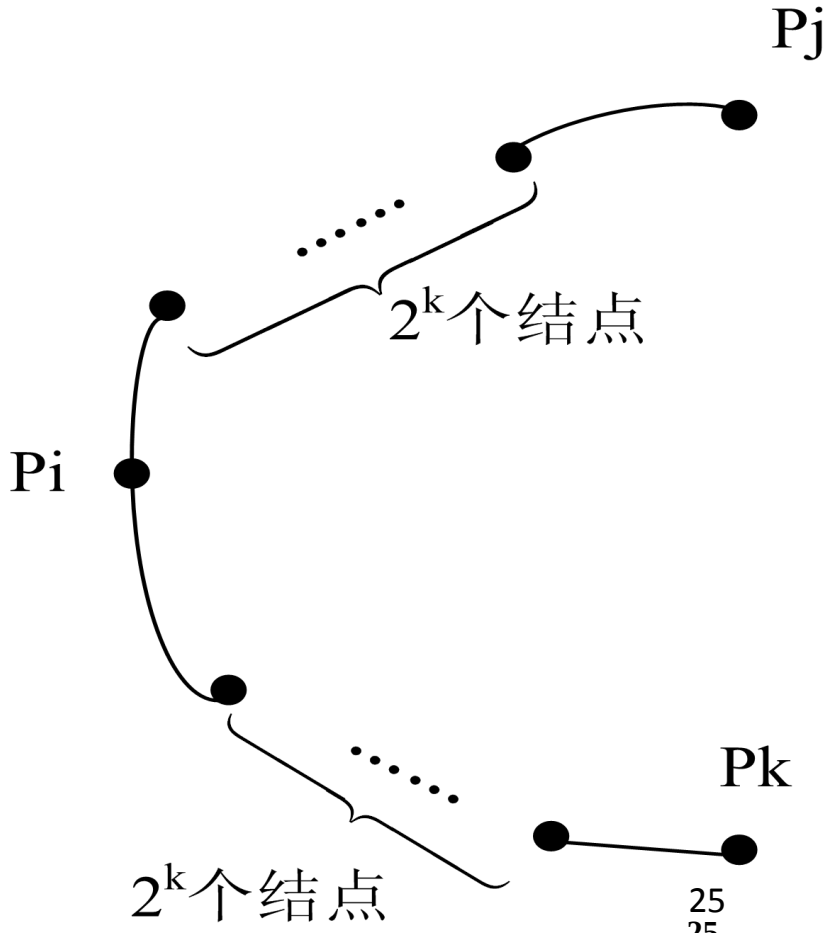
Lemma 3.3 对每个k≥1，在phase k结束时，临时leader数至多为 $n/(2^k+1)$.

pf: 若一结点P_i在k阶段结束时是一临时leader，则在P_i的2^k-邻居里每个结点的id均小于P_i的id。

在该阶段里，距离最近的两个临时leader P_i和P_j必满足：

P_i的2^k邻居的左边恰好P_j的2^k-邻居的右边，即P_i和P_j之间有2^k个处理器。

因此，在phase k里临时leader的最大数目必是以上述方式分布的，因为每2^k+1个结点至多有一个临时leader，所以leader数至多是$n/(2^k+1)$.



§3.3.2 一个O(*n*lg*n*)算法

Th3.4. 存在一个异步的leader选举算法，其msg复杂性为O(*n*lg*n*).

Pf: 由lemma3.3知，知道phase lg(n-1)时只剩下一个leader(最后的leader). msg总数：

$$4n + \sum_{l=1}^{\lg(n-1)} (4 \cdot 2^l) \frac{n}{2^{l-1}+1} + n \le 8n \lg n$$

i) phase 0: msg数为4n.

ii)终止msgs：n.

Note: 双向通信. 该msg复杂性的常数因子不是最优的.

下次继续！

第三章 环上选举算法

汪 炀

§3.3.3 下界Ω(*n*lg*n*)

现证明对于uniform算法，异步环里任何leader选举算法至少发送Ω(*n*lg*n*)个msgs。

我们的下界证明是针对leader选举问题的一个变种：

- 选中的leader必定是环上具有最大id的处理器。
- 所有处理器必须知道被选中leader的id，即每处理器终止前，将选中leader的id写入一个特殊变量。

- 基本思想。

设A是一个能解上述leader选举变种问题的均匀算法，证明存在A的一个允许执行，其中发送了Ω(*n*lg*n*)个msgs，证明采用构造法。

§3.3.3 下界Ω(*n*lg*n*)

对于大小为n/2的环构造算法的一个耗费执行(指msg的耗费)，然后将两个大小为n/2的不同环粘贴在一起形成一个大小为n的环，将两个较小环上的耗费执行组合在一起，并迫使θ(n)个附加msg被接收。

- 调度：前面定义过调度是执行中的事件序列，下面给出能够被粘贴在一起的调度。

这种扩展依赖于算法是一致的且对各种规模的环以相同的方式执行

- Def3.1 开调度

设σ是一个特定环上算法A的一个调度，若该环中存在一条边e使得在σ中，边e的任意方向上均无msg传递，则σ称为是open，e是σ的一条开边。

§3.3.3 下界Ω(*n*lg*n*)

Note：开调度未必是容许的调度，即它可能是有限的事件序列，环上的处理器不一定是终止的。

直观上，既然处理器不知道环的大小，我们能将两个较小的开调度粘贴为一个较大环的开调度，其依据是：算法是均匀的。

为简单起见，不放设n为2的整数次幂。

Th3.5 对于每个n及每个标识符集合(大小为n)，存在一个由这些标识符组成的环，该环有一个A的开调度，其中至少接收M(n)个消息，这里：

$$\begin{cases} M(2)=1 & n=2 \\ M(n)=2M(\frac{n}{2})+\frac{1}{2}(\frac{n}{2}-1) & n>2 \end{cases}$$

§3.3.3 下界Ω(*n*lg*n*)

显然递归方程的解为M(*n*)=θ(*n*lg*n*)，他蕴含了异步环选举问题消息复杂度下界。下面用归纳法证明之，其中

$$\begin{cases} \text{引理3.6是归纳基础}(n=2^1) \\ \text{引理3.7是归纳步骤}(n=2^i, i>1) \end{cases}$$

Lemma3.6 对每个由两个标识符构成的集合，存在一个使用这两个标识符的环R，R有A的一个开调度，其中至少有一个msg被接受。（归纳基础）

pf：假定R有两个处理器P_0和P_1，其标识符分别为x和y，不妨设x>y.

§3.3.3 下界Ω(*n*lg*n*)

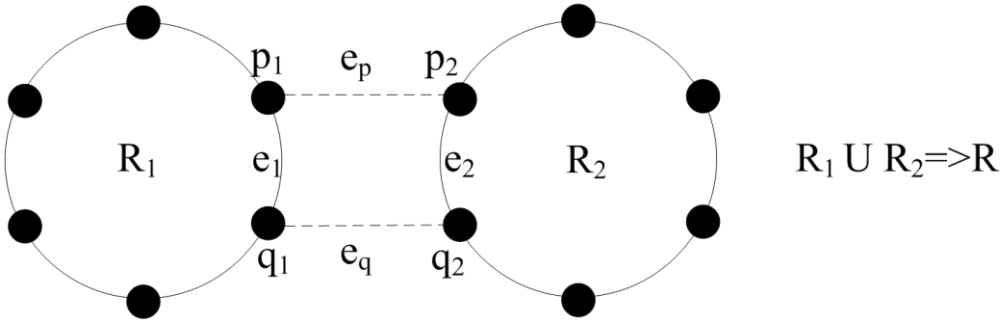
设α是A的一个容许执行，因为A是正确的，在α中，最终P_1定将P_0的标识符写入其中。因此，α中至少须接收一个msg，否则P_1不知道P_0的标识符为x.

设σ是α的调度的最短前缀：它包括第一个接受msg的事件。因为没有接收第一条msg的边是开的，因此σ中只有一个msg被接收且有一条开边，故引理成立。故σ是满足引理的开调度。

Lemma 3.7 选择n>2，假定对每个大小为n/2标识符集合，存在一个使用这些标识符的环，它有A的一个开调度，其中至少接收M(n/2)个msgs(归纳假设)，那么对于n个标识符的每个集合，存在一个使用这些标识符集的环，它有A的一个开调度，其中接收至少2M(n/2)+(n/2-1)/2个msgs(归纳步骤)。

§3.3.3 下界Ω(*n*lg*n*)

pf：设S是n个标识符的集合，将S划分为两个集合S_1和S_2，每个大小为n/2，由假设分别存在一个使用S_1和S_2中标识符的环R_1和R_2，它们分别有A的一个开调度σ_1和σ_2，其中均至少接收M(n/2)个msgs，设e_1和e_2分别是σ_1和σ_2的开边，不妨设邻接于e_1和e_2的处理器分别是p_1，q_1和p_2，q_2，将e_1，e_2删去，用e_p链接p_1和p_2，e_q链接q_1和q_2，即可将两个环R_1和R_2粘贴在一起形成环R。



现说明如何在R上构造一个A的开调度σ，其中至少有2M(n/2)+(n/2-1)/2个msg被接收。其想法是先让每个较小环分别执行“耗费”的开调度。

§3.3.3 下界Ω(*n*lg*n*)

1) $\sigma_1\sigma_2$构成R上A的一个开调度

考虑从R的初始配置开始发生的事件序列σ_1，因为R_1中的处理器由这些事件并不能区别R_1是一个独立的环还是R的一个子环，它们执行σ_1恰像R_1是独立的那样。考虑环R上后续事件序列σ_2(与上类似)，因为没有msg在e_p和e_q上传递，故R_2中处理器在σ_2中亦不能区别R_2是独立环还是R的子环。

因此，$\sigma_1\sigma_2$是一个调度，其中至少有2M(n/2)个msgs被接收。

2) 现说明如何通过连通e_p和e_q(但不是二者)来迫使算法接收(n/2-1)/2个附加的msgs。

考虑每个形式为$\sigma_1\sigma_2\sigma_3$的有限调度，因为$\sigma_1\sigma_2$中e_p和e_q均为开的，若σ_3中存在一边上至少有(n/2-1)/2个msg被接收，则$\sigma_1\sigma_2\sigma_3$是要找的开调度，引理被证。

假设没有这样的调度，那么存在某个调度$\sigma_1\sigma_2\sigma_3$，它导致相应执行中的一个“静止”配置。(配置：由全体结点状态构成)

一个处理器状态是“静止”的：若从该状态开始的计算事件序列中不send消息，即处理器接收一个msg之前不发送另一msg（即处理器的内部事件不引发send动作）

§3.3.3 下界Ω(*n*lg*n*)

一个配置是“静止”的(关于e_p和e_q)：若除开边e_p和e_q外，没有msgs处在传递之中，每个处理器均为静止状态。

不失一般性，假设R中最大id的处理器是在子环R_1中，因为没有msg从R_1传到R_2中，R_2中的处理器不知道leader的id，因此R_2里没有处理器能够在$\sigma_1\sigma_2\sigma_3$结束时终止。(在$\sigma_1\sigma_2\sigma_3$结束时，R_2里无结点终止)

我们断定在每个扩展$\sigma_1\sigma_2\sigma_3$的容许调度里，子环R_2里的每个处理器在终止前必须接收至少一个附加msg，因为R_2里每一处理器只有接收来自R_1的msg才知道leader的id。

上述讨论清楚地蕴含在环R上必须接收Ω(n/2)个msgs，但因为e_p和e_q是连通的，故调度未必是开的，即两边上均可能传递msg。

但若能说明e_p或e_q只有一个是连通的，迫使通过它接收Ω(n/2)个msgs，即可证明。这就是下一断言。

§3.3.3 下界Ω(*n*lg*n*)

Claim3.8 存在一个有限的调度片断σ_4，其中有 (n/2-1)/2个msgs被接收，$\sigma_1\sigma_2\sigma_3\sigma_4$是一个开调度，其中$e_p$或$e_q$是开的。

Pf: 设 使得$\sigma_1\sigma_2\sigma_3$ 是一个容许调度，因此所有的msgs在e_p和e_q上传递，所有结点终止。

因为R_2里，每个节点在σ_4''终止前必须收到一个msg，故在σ_4''终止前在 里至少接收n/2个msgs,设 是 里接收n/2-1个msg的最短前缀。

考虑 R_2''里在 中所有已接收msg的结点，σ_4' 因为我们是从一个静止位置开始的，其中只有在e_p和e_q上有msg在传输，故这些结点形成了 两个连续的结点集合P和Q：

P包含由于连通e_p而被唤醒的结点，故P至少包含p_1和p_2

Q包含由于连通e_q而被唤醒的结点，故Q至少包含q_1和q_2

§3.3.3 下界Ω(*n*lg*n*)

因为P∪Q中至少包含n/2-1个结点（由σ_4'决定），且又因它们中的结点是连续的，所以P∩Q=Φ。P和Q这两个集合中有一个集合，其中的结点至少接收(n/2-1)/2个msg,(因为P，Q中的结点共接收n/2-1个msg),不失一般性，假定这样的集合是P。

设σ_4是σ_4'的子序列，σ_4只包含在P中结点上发生的事件，因为σ_4里P中节点和Q中结点之间没有通信，故$\sigma_1\ \sigma_2\ \sigma_3\ \sigma_4$是一个调度。

因为σ_4里至少有 (n/2-1)/2各msg被接收，且由构造可知，e_q上无msg传递，因此$\sigma_1\ \sigma_2\ \sigma_3\ \sigma_4$是一个满足要求的开调度。

§3.3.3 下界Ω(*n*lg*n*)

总结： Th3.5的证明可分为3步：

1) 在 R1和R2 上构造2个独立的调度，每个接收2M(n/2)各msg: σ_1 σ_2

2) 强迫环进入一个静止配置： σ_1 σ_2 σ_3 （主要由调度片断σ_3 ）

3) 强迫(n/2-1)/2个附加msg被接收，并保持ep或eq是开的: σ_1 σ_2 σ_3 σ_4。

因此我们已构造了一个开调度，其中至少有2M(n/2)+ (n/2-1)/2个msg被接收。

§ 3.4 同步环

研究同步环上leader选举问题的上、下界。

- 上界： 提出两个msg复杂性为O(n)的算法，显然，这样的算法的msg复杂性是最优的。但运行时间并非只与环大小n有关，还与算法使用的非普通的id相关。(与id值相关)
- 下界： 讨论

  1）只基于标识符之间比较的算法

  2）时间受限（即若干轮内终止，轮数只依赖于环大小,不依赖于id值）的算法

  当算法受限于上述两个条件时，至少需要 $\Omega(n \lg n)$ 个msgs.

§ 3.4.1 上界 O(n)

上节已证明在异步环上leader选举问题的msg复杂度下界为 $\Omega(n \lg n)$，其算法的关键是msg延迟是任意长的。

因为同步环上

1) msg延迟是确定的，故同步模型是否有较好的结果呢?

2) 获取info不仅来自于接收msg,在某轮里的内附件也能获取info

本节提出两个算法，msg复杂性上界为O(n),针对单向环，但是用于双向环

1) 非均匀的：要求环中所有的结点开始于同一轮，标准的同步模型 (需知道n)

2) 均匀的： 结点可开始于不同轮，弱同步模型 (无需知道n)

§ 3.4.1 上界O(n)

Non-uniform Alg

- 基本特征

  选择环中最小id（各id互不相同）的结点作为leader，按Phase运行，每个阶段由n个轮组成。在Phase i (i ≥ 0)，若存在一个id为i的结点，则该结点为leader，并终止算法，因此，最小id的结点被选为leader

  显然，Phase数目取决于n个节点的标志符的取值。

- 具体实现

  Phase i包括轮：n·i+1, n·i+2, …, n·i+n

  在第i阶段开始，若一个结点的id是i，且它尚未终止，则该节点绕环发送一个msg后作为一个leader终止；若一结点的id不是i，且它收到一个msg，则它转发此msg后作为non-leader终止。

§3.4.1 上界O(n)

- 分析

正确性：显然，只有最小的标志符被选中作为leader

Msg复杂性：恰有n个msg被发送，故复杂性为O(n)。注意这n个msg均是在找到leader的那个Phase里发送的。

时间复杂性

依赖于环大小和环上最小标志符，不妨设环大小为n，最小标识符为i，则算法执行轮数为：n·(i+1)，不妨设i≥0 // 运行时间与环大小及标识符取值相关

缺点

必须知道环大小n和同步开始，下面算法克服了这些限制

①为什么id为i的结点要在phase i发msg

∵各结点互不知道彼此的id值

∴只能在第i phase，结点(id=i)发自己的id

②为什么每个phase要n轮

§ 3.4.1 上界O(n)

Uniform Alg

- 特点：①无须知道环大小，②弱同步模型

  一个处理器可以在任意轮里自发地唤醒自己，也可以是收到另一个处理器的msg后被唤醒

- 基本思想

  ① 源于不同节点的msg以不同的速度转发

  源于id为i的节点的msg，在每一个接收该msg的节点沿顺时针转发到下一个处理器之前，被延迟2^i-1轮

  ② 为克服非同时启动，须加一个基本的唤醒阶段，其中每个自发唤醒的结点绕环发送一个唤醒msg，该msg转发时无延迟

  ③ 若一个结点在算法启动前收到一个唤醒msg，则该结点不参与算法，只是扮演一个relay(转发)角色：即转发或没收msg

§3.4.1 上界O(n)

- 要点：在基本阶段之后，选举leader是在参与结点集中进行的，即只有自发唤醒的结点才有可能当选为leader。
- 具体实现

  ① 唤醒：由一个结点发出的唤醒msg包含该结点的id，该msg以每轮一边的正常速率周游，那些接收到唤醒msg之前未启动的结点均被删除(不参与选举)

  ② 延迟：当来自一个id为i的节点的msg到达一个醒着的节点时，该msg以2^i速率周游，即每个收到该msg的节点将其延迟2^i-1轮后再转发。

Note： 一个msg到达一个醒着的节点之后，它要到达的所有节点均是醒着的。一个msg在被一个醒着的节点接收之前是处在1st阶段 (唤醒msg，非延迟) ，在到达一个醒着的节点之后，它就处于2nd阶段，并以2^i速率转发(非唤醒msg，延迟)

§ 3.4.1 上界O(n)

③ 没收规则

a) 一个参与的节点收到一个msg时，若该msg里的id大于当前已看到的最小(包括自己)的id，则没收该msg；

b) 一个转发的节点收到一个msg时，若该msg里的id大于当前已看到的最小(不包括自己)的id，则没收该msg。

§ 3.4.1 上界O(n)

- 算法 Alg3.2 同步leader选举

```
var waiting : init  Φ;
    asleep  : init true;   //加上relay更好？     : init false;
1: 设R是计算事件中接收msg的集合
2: s:= Φ;// the msg to be sent
3: if asleep then {
4:    asleep:=false;
5:    if R = Φ then { // pi未收到过msg，属于自发唤醒
6:          min:=id;  //参与选举
7:       s:=s+{<id>}; // 准备发送
      }else{ //已收到过msg，但此前未启动，被唤醒故Pi不参与
8:        min:=∞; //选举，置min为∞，使其变为relay结点
         // relay:=true; ?
    }
  }
```

§3.4.1 上界O(n)

9: for each <m> in R do {// 处理完收到的m后相当于从R中删去
10: if m < min then { // 收到的id较小时通过
11: become not elected; // P_i未被选中
//可用relay控制使转发节点不延迟？
12: 将<m>加入waiting且记住m何时加入; //m加入延迟转发
13: min:=m;
} // if m > min then it is swallowed
14: if m=id then become elected; // P_i被选中
} //endfor
15: for each <m> in waiting do
16: if <m> 是在2^m-1轮之前接收的 then
17: 将<m>从waiting中删去并加入S
18: send S to left;

§3.4.1 上界O(n)

- 分析

下面证明，在第1个结点被唤醒之后的n轮，只剩下第二阶段的msg，只有参与的结点才有可能被选中。

(1) 正确性

对∀i∈[0,n-1]，设id_i是结点p_i的标识符，$<id_i>$是源于p_i的msg

Lemma 3.9 在参与的节点中，只有最小id的节点才能收回自己的id。

pf：①选中：设p_i是参与者中具有最小id的结点(Note：至少有1个结点须参与算法)，显然没有节点(无论是否参与)能没收$<id_i>$；另一方面，因为在每个节点上$<id_i>$至多延迟2^{id_i}轮，故p_i最终收回自己的id；

②唯一：除p_i外，没有别的节点p_j(j≠i)也收回自己的$<id_j>$。

若p_j收回自己的$<id_j>$，则$<id_j>$已通过p_i及其它所有结点，但$id_i < id_j$，因为p_i是一个参与者，它将不会转发id_j，矛盾！

该引理蕴含着：恰有一个结点收回自己的msg，故它是唯一声明自己是leader的结点，即算法正确。

§3.4.1 上界O(n)

(2) msg复杂性

在算法的一次容许执行里，发送的msg可分为三个类型：

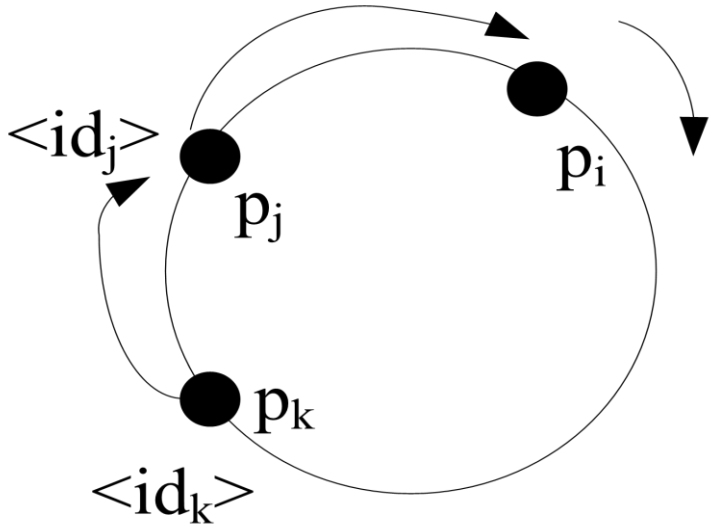
第一类：第一阶段的msg(唤醒msg)

第二类：最终leader的msg进入自己的第二阶段之前发送的第二阶段msg(其它结点发出的)

第三类：最终leader的msg进入自己的第二阶段之后发送的第二阶段msg(包括leader发出的)

Note：一个msg发送时，一开始是作为唤醒msg(非延迟)，当它到达的结点已唤醒时，msg就变为非唤醒msg(第二阶段，延迟msg)

§3.4.1 上界O(n)



① 第一类msg总数(第一阶段的msg)

Lemma 3.10 第一类msg总数至多为n

pf：只要说明每个节点在第一阶段至多转发一个msg即可。

反证：假设某结点p_i在其第一阶段转发两个msgs：一个来自p_j的$<id_j>$，一个来自p_k的$<id_k>$。不失一般性，p_j比p_k更靠近p_i(沿顺时针方向)。因此，$<id_k>$到达p_i之前先到达p_j。

若$<id_k>$是在p_j自发唤醒及发送$<id_j>$之后到达p_j，则$<id_k>$以2^{id_k}速度作为第二阶段msg继续前进；否则，p_j不是参与结点，不会发送$<id_j>$。

因此，或者$<id_k>$是作为第二阶段msg到达p_i，或者$<id_j>$未被发送，即：p_i最多收到一个第一阶段msg，矛盾！

§3.4.1 上界O(n)

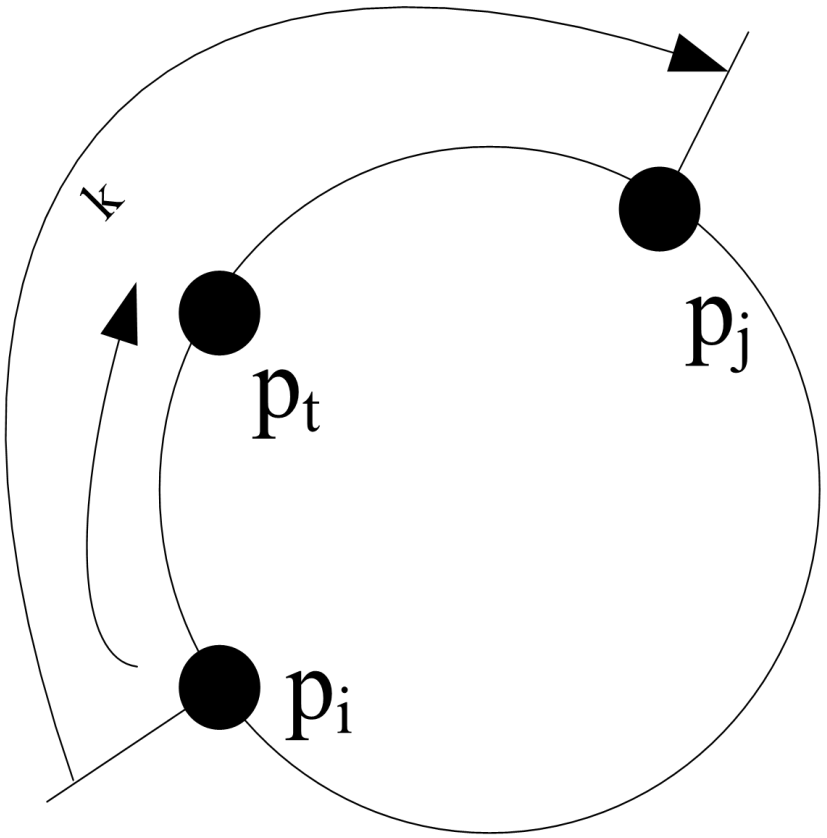
② 第二类msg总数 (最终leader发出的msg进入自己的第二阶段之前发送的第二阶段msg)

为了求得第二类msg总数，首先说明第一个开始执行算法的结点启动之后的n轮，所有的msg均在自己的第二阶段中。

设p_i是最早开始执行算法的结点中的某一个，其启动轮数为r。

Lemma 3.11 若p_j距离p_i为k(顺时针)，则p_j 接收的第一阶段的msg不迟于第r+k轮。

§3.4.1 上界O(n)



pf：对距离k归纳

基础：k=1，因为p_i的左邻居在第r+1轮接收到p_i的msg，故引理成立。

假设：设距离p_i为k-1的结点接收第一阶段的msg不迟于r+k-1轮。

步骤

若距离p_i为k-1的结点p_t(顺时针)接收第一阶段msg时已被唤醒，则p_t已发送了第一阶段msg给邻居p_j；

否则，p_t至迟在第r+k轮转发第一阶段msg到p_j。

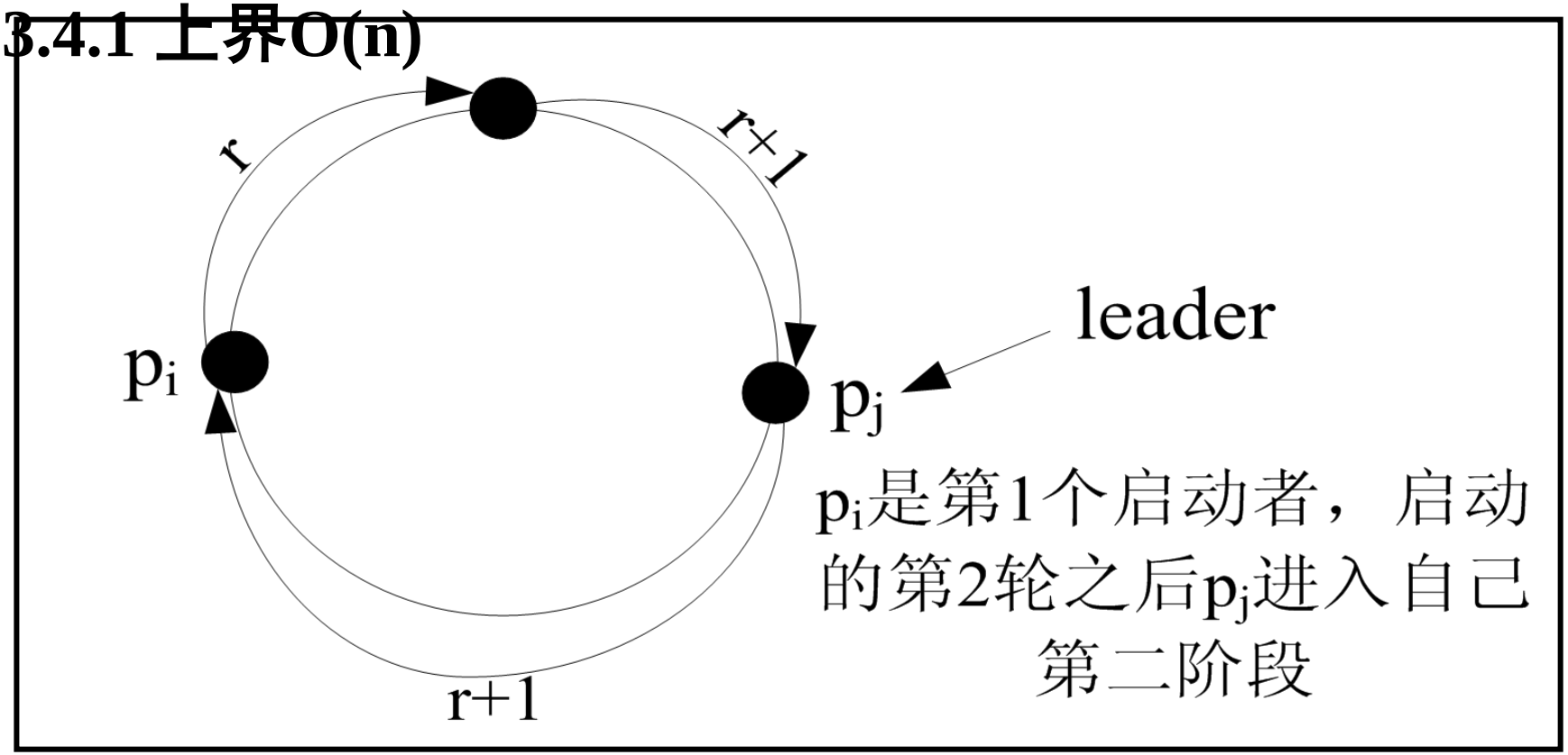
§3.4.1 上界O(n)

Lemma 3.12 第二类msg总数至多为n

pf：由引理3.10可知，在每边上至多只发送1个第一阶段的msg，又因为到第r+n轮，每边上已发送了一个第一阶段msg，故到第r+n轮之后，已无第一阶段的msg被发送。

Note：第一阶段msg是唤醒msg，即若在p_i(第一个启动结点)发出唤醒msg绕环一周回到p_i之前已有某结点启动，则该启动结点的msg在未收到p_i的msg之前已将自己的唤醒msg向前转发。

§3.4.1 上界O(n)



i) 第二类msg经历的总轮数：

由引理3.11知，最终的leader(不一定是首个启动者)的msg进入自己的第二阶段的时刻是：算法的第1个msg被发送之后至多n轮(前n轮)，故第二类被发送的msg必是在首个启动结点的n轮之中。

ii) 在这n轮中，第二类msg数目。即第二类msg是算法启动的前n轮中非唤醒msg的总数：

§ 3.4.1 上界O(n)

因为msg<i>在其第二阶段中，转发前须延迟2^i-1轮，所以若<i>是第二类msg，则它至多被发送$n/2^i$次。

因为较小id被转发的次数较多，故可这样构造以使msg数目最大：

所有结点均参与选举，标识符均尽可能小：0, 1, …, n-1（顺时针排列）。显然，因为id=0是leader，第二类msg中不包括leader的msg，故第二类msg总数至多是：

$$\sum_{i=1}^{n-1} \frac{n}{2^i} \le n$$

§3.4.1 上界O(n)

③第三类msg总数

(即：leader进入自己的第二阶段之后，所有非唤醒msg)

Lemma 3.13 在<id_i>返回p_i之后，没有msg被转发

pf：

设p_i是具有最小id的结点，当一结点转发<id_i>之后，该结点将不再会转发其它msg。

若<id_i>返回p_i，则所有结点均已转发过<id_i>，故再也没有其它msg被转发。

§ 3.4.1 上界O(n)

Lemma 3.14 第三类msg总数至多为2n

pf：设p_i是最终的leader，p_j是某个参与的结点，由引理3.9知，$id_i<id_j$，由引理3.13知，在p_i收回自己的id_i之后，环上不再有msg在传输。

i) 第三类msg经历的总轮数：因为在每个结点上，$<id_i>$至多延迟2^{id_i}轮(在唤醒结点上不延迟，故为至多)，所以$<id_i>$返回p_i至多经过$n\cdot 2^{id_i}$轮。

ii) 在这$n\cdot 2^{id_i}$轮中，第三类msg数目：第三类msg是在这$n\cdot 2^{id_i}$轮中发送的所有第二阶段msg(非唤醒msg)。在这$n\cdot 2^{id_i}$轮中，一个非唤醒的msg$<id_j>$被转发的次数至多为：

$$\frac{1}{2^{id_j}}\cdot n\cdot 2^{id_i}=n\cdot 2^{id_i-id_j}$$

§ 3.4.1 上界O(n)

故有：第三类msg总数至多为(包括leader)

$$\sum_{j=0}^{n-1} \frac{n}{2^{id_j - id_i}}$$

基于引理3.12的同样理由，当所有结点参与选举，及标识符为0, 1, …, n-1时有：

$$\sum_{j=0}^{n-1} \frac{n}{2^{id_j - id_i}} \le \sum_{k=0}^{n-1} \frac{n}{2^k} \le 2n$$

这里，$\forall\, id_i \in [0, n-1]$

Th3.15 存在一个同步的leader选举算法，其msg复杂度至多为4n

pf：由引理3.10，3.12及3.14立即可得。

§3.4.1 上界O(n)

(3) 时间复杂度

由引理3.13知，当leader接收到自己的id时，计算终止。这发生在第一个启动算法的节点之后的$O(n\cdot 2^i)$轮，其中i是leader的标识符。 // 当i=0时，为O(n)轮

//运行时间与环大小及标识符取值相关

(4) 思考

为何非唤醒msg要延迟2^i -1轮？

如何修改算法3.2来改善时间复杂性？

第三章 环上选举算法

汪 炀

§3.4.2 有限制算法的下界Ω(nlgn)

上节中的两个算法在同步环上最坏的msg复杂度为O(n)，但两算法的缺陷为：

① 它们用一种非同寻常的方式使用id，即id决定msg延迟多长；

② 在每个容许的执行中，执行轮数依赖于id，而id相对于n而言可能是巨大的。(更主要的)

本节将说明：

时间复杂度会表现很差

① 这些性质对于基于消息的有效算法而言是固有的；

② 若一个算法中的标识符仅用于比较，则它需要Ω(nlgn)个msgs；

③ 若一个算法中，限制轮数的上界，但独立于id，则它的msg复杂度亦为Ω(nlgn)。

§3.4.2 有限制算法的下界Ω(nlgn)

同步的下界不可能从异步的下界导出

因为上节中的算法表明：同步模型中的附加假定是必不可少的。

同步的下界对于非均匀和均匀算法均成立，但异步的下界只对均匀算法成立。

但是从同步导出的异步结果是正确的，并且提供了一个非均匀算法的异步下界。

异步通信模型中领导者选举问题所需的消息数下界为Ω(nlgn)且算法不依赖于比较的或者限时的

§3.4.2 有限制算法的下界Ω(nlgn)

基于比较的算法的概念和定义

为了得到下界，假定所有处理器在同一轮开始执行

一个环是由结点表按顺时针方向指定的，结点表开始于最小标识符。

在同步模型里，算法的容许执行完全由初始配置定义，这是因为msg延迟及节点步骤的相对次序均无选择的可能。

系统的初始配置完全由环决定，即由节点标识符列表按上述规则来决定。当算法的选择可以从上下文判断清楚时，则将由环R确定的容许执行表示为exec(R).

- 匹配：若环R_1上的结点p_i和R_2上的p_j在各自的环里具有相同的位置，则称p_i和p_j是匹配的，即：匹配的结点在各自环上距其最小id的结点的距离相同。

§3.4.2 有限制算法的下界Ω(nlgn)

- 基于比较的算法

直观上，若一个算法在两个相对次序相同的环上具有相同的行为，则该算法是基于比较的，形式定义：

1）序相同（order equivalent）

两个环$x_0, x_1,\ldots, x_{n-1}$和$y_0, y_1,\ldots,y_{n-1}$是(次)序等价的，若对每个i和j，$x_i<x_j$, 当且仅当$y_i<y_j$。

回忆一下环上的结点p_i的k-邻居是2k+1个结点的序列(下标是mod n)：

$p_{i-k}, \ldots, p_{i-1}, p_i, p_{i+1},\ldots, p_{i+k}$

序等价的概念很容易扩展到k-邻居集(所有索引按模n计算)

§3.4.2 有限制算法的下界Ω(nlgn)

2）何谓行为相同(similar)?

直观上：在序等价的环R_1和R_2上的容许执行里，发送同样的msg做同样的决策。但一般情况下，算法发送的msg包含结点的id，因此，R_1上发送的msg通常不同于R_2上发送的msg。然而，我们关注的是msg模式和决策。所谓msg模式(patten)是指：msg是何时何地发送的，而不是指其内容。

- 节点在第k轮里行为相似：考虑两个执行α_1，α_2和两个结点p_i，p_j，我们说p_i在α_1的第k轮里的行为相似于p_j在α_2的第k轮里的行为，若下述条件成立：

  ① p_i在α_1的第k轮里发送一个msg到其左(右)邻居当且仅当p_j在α_2的第k轮里发送一个msg到其左(右)邻居；

  ② p_i在α_1的第k轮里作为一个leader终止当且仅当p_j在α_2的第k轮里作为一个leader终止。

§3.4.2 有限制算法的下界Ω(nlgn)

- 节点的行为相似：若α_1里p_i和α_2里p_j在所有的$k\geq 0$轮里均相似，则称α_1里p_i和α_2里p_j的行为相似。
- 算法的行为相似：指每对匹配的结点行为相似

3）定义

Def 3.2 一个算法是基于比较的，若对于每对序等价的环R_1和R_2，每对匹配的节点在exec(R_1)和exec(R_2)里的行为均相似。

该定义说明，若一算法只与环上标识符之间的相对次序相关，而与具体id值无关，则该算法一定只是基于标识符的比较

§3.4.2 有限制算法的下界Ω(nlgn)

2. 基于比较算法的下界

设A是一个基于比较的leader选举算法，证明时考虑的环就序模式而言具有高度的对称性。即：环里存在很多次序等价的邻域。

直观上，只要两个节点有序等价的邻域，它们在A里就有同样的行为。我们将通过在一个高度对称的环里执行A来导出下界。讨论若一结点在某轮里发送一个msg，则具有序等价邻域的所有结点也在同一轮里发送一个msg。

证明的关键是：区别获得信息的轮及没有获得信息的轮.

Note：在同步环里，一个结点即使没有收到一个msg也会获得info，例如，在§3.4.1里的非均匀算法中，若第1到第n轮里未接收到msg，实际上蕴含着信息：环里没有标识符为0的结点。

§3.4.2 有限制算法的下界Ω(nlgn)

下面的证明关键是观察：

若某一轮r里不存在msg也对于结点p_i是有用的(指可获取info)，则仅当存在一个次序等价的不同环，在该环上的第r轮里已接收一个msg

例如，在非均匀算法中，若环上某一个结点的id为0，则在第1, 2, …, n轮里均已收到msg。但对于一个次序等价的不同环(设最小id>0)，则它在前n轮里就没有任何msg存在。但同样认为前n轮对每个结点是有用的。

因此，若某一轮在任何次序等价的环上均无msg发送，则该轮是无用的，而有用的轮被称为是主动的(active)。

§3.4.2 有限制算法的下界Ω(nlgn)

Def 3.3 在一个环R上的一个执行里，某轮r称为是active的，若该执行的第r轮里，某一个结点发送一个msg。当R是从上下文已知时，用r_k表示第k个active轮。

Note： 一旦环R确定，整个容许执行就确定(因为系统同步)

由于一个基于比较的算法在等价环上的行为相似，因此：

对于序等价的环R_1和R_2，一轮在exec(R_1)里是主动的当且仅当它在exec(R_2)里也是主动的。

因为消息中的信息在k个轮里只能在环上通过k个结点，所以一个结点在k轮之后的状态只依赖于它的k-邻居。

然而一个更强的性质是：一个结点在第k个主动轮之后的状态只依赖于它的k-邻居。这实际上告诉我们：信息只有在主动轮里才能获得。这一点在下面的引理中给出形式证明。

§3.4.2 有限制算法的下界Ω(nlgn)

Note：该引理无需结点匹配(否则立即从定义3.2中可得结论)，但要求它们的邻居是相同的(identical)。该引理要求假设两个环是次序等价的，原因是要确保考虑中的两个执行有相同的主动轮集合，因此r_k是良定义的。

Lemma3.16 设R_1和R_2是次序等价的两个环，设P_i和P_j分别是R_1和R_2上具有相同k-邻居的两个节点，那么在exec(R_1)的第1至第r_k轮里P_i所经历的转换序列和在exec(R_2)的第1至第r_k轮里P_j所经历的转换序列相同。

//该引理蕴含：在k个主动轮结束时，P_i和P_j的状态相同

Pf：非形式地，该证明说明在k个主动轮之后，一个结点可能只知道距离自己至多为k的那些结点。形式证明对k进行归纳。

§3.4.2 有限制算法的下界Ω(nlgn)

① 归纳基础：k=0，因为两个具有相同0-邻居的结点有同样的id，故它们的状态相同；

② 归纳假设：假定每两个具有相同(k-1)-邻居的结点在(k-1)个主动轮之后有相同的状态；

③ 归纳步骤：因为P_i和P_j有相同的k-邻居，故它们亦有相同的(k-1)-邻居。因此由归纳假设知，P_i和P_j在第(k-1)个主动轮之后处在相同的状态。又因P_i和P_j各自的邻居有同样的(k-1)-邻居，故由归纳假设知，它们各自的邻居在第(k-1)个主动轮之后也处在相同的状态。

两个主动轮之间可能有非主动轮。

因为在第(k-1)主动轮和第k主动轮之间的轮(若有的话)里，没有结点接收任何msg，故P_i和P_j及各自的邻居均处在相同的状态(Note: P_i在非主动轮里可能改变它的状态，但因为P_j有同样的转换函数，故它有同样的状态转换)。

§3.4.2 有限制算法的下界Ω(nlgn)

在第k个主动轮里：

i)若P_i和P_j均不接收msg，则它们在该轮结束时有相同的状态；

ii)因为P_i 和P_j的邻居状态相同，若P_i接收右(左)邻的一个msg，则P_j也接收右(左)邻同样的msg。因此，在第k个主动轮结束之后，P_i和P_j均处于相同的状态。□

下一引理将上述论断从具有相同k-邻居的结点扩展至具有次序等价的k-邻居的结点。这依赖于事实：A是基于比较的。

更进一步要求环R是有空隙的，即环R中，每两个id之间均有n个未使用的标识符。形式地，在大小为n的环上，若对于每一个标识符x，标识符x-1到x-n均不在环上，则该环称为有空隙的。

§3.4.2 有限制算法的下界Ω(nlgn)

引理3.16定义在两环上(P_i和P_j的k-邻居相同)，引理3.17是定义在同一个环上(P_i和P_j的k-邻居序等价)

Lemma3.17 设R是有空隙环，P_i和P_j是R上具有序等价k-邻居的两个结点。则P_i和P_j在exec(R)的第1到第r_k轮里有相似的行为。

Pf：我们构造满足下述条件的另一个环R’：

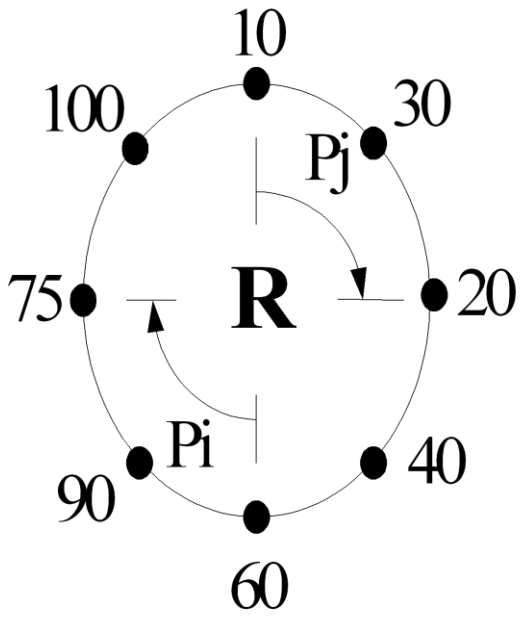
① R’中的P_j和R中P_i的有相同的k-邻居；

② R’中的标识符唯一

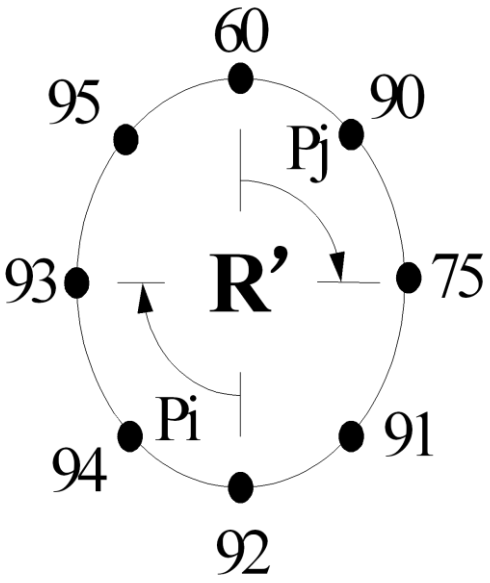
③ R’和R序等价，R’中的P_j匹配于R中的P_j。

因为R是有空隙环，故我们能够构造R’。例如，对于k=1和n=8有：

§3.4.2 有限制算法的下界Ω(nlgn)



R



R'

① P_i的1-邻居60，90，75　　P_j的1-邻居60，90，75

② R'中id唯一

③ R次序：　　R'次序：

10,30,20,40,60,90,75,100　　60,90,75,91,92,94,93,95

P_j与10距离为1，　　P_j与60距离为1

§3.4.2 有限制算法的下界Ω(nlgn)

(1) R上的P_i和R’上的P_j的前k个主动轮行为相似

因为R上P_i和R’上P_j的k-邻居相同，由引理3.16知，P_i和P_j在各自的exec(R)和exec(R’)的1到r_k轮里所经历的转换序列相同，所以P_i和P_j在各自的执行exec(R)和exec(R’)的1至r_k轮里的行为相似。 $P_i(R) \backsim P_j(R')$

(2)R上的P_j和R’上的P_j的前k个主动轮行为相似

因为算法是基于比较的，由定义3.2在两个序等价的环中，每对匹配的结点在各自执行中有相似的行为。而R里的P_j和R’里的P_j是匹配的，故R里的P_j在exec(R)的1至r_k轮里的行为相似于R’里的P_j在exec(R’)的第1至r_k轮里的行为。 $P_j(R') \backsim P_j(R)$

(3)R上两节点的k-邻居序等价，则其行为在前k个主动轮里相似

由(1)和(2)得：R里的P_i和P_j在exec(R)的1至r_k轮里的行为相似。$P_i(R) \backsim P_j(R)$ □

§3.4.2 有限制算法的下界Ω(nlgn)

Th3.18 对于每个n≥8(n是2的幂)，存在一个大小为n的环S_n，使得对每个基于比较的同步leader选举算法A，在S_n上A的容许执行里发送msg的数目为Ω(nlgn)

Pf：固定算法A。证明分2步：

(1) 构造1个高度对称(很多结点有很多序等价的邻居)的环S_n；

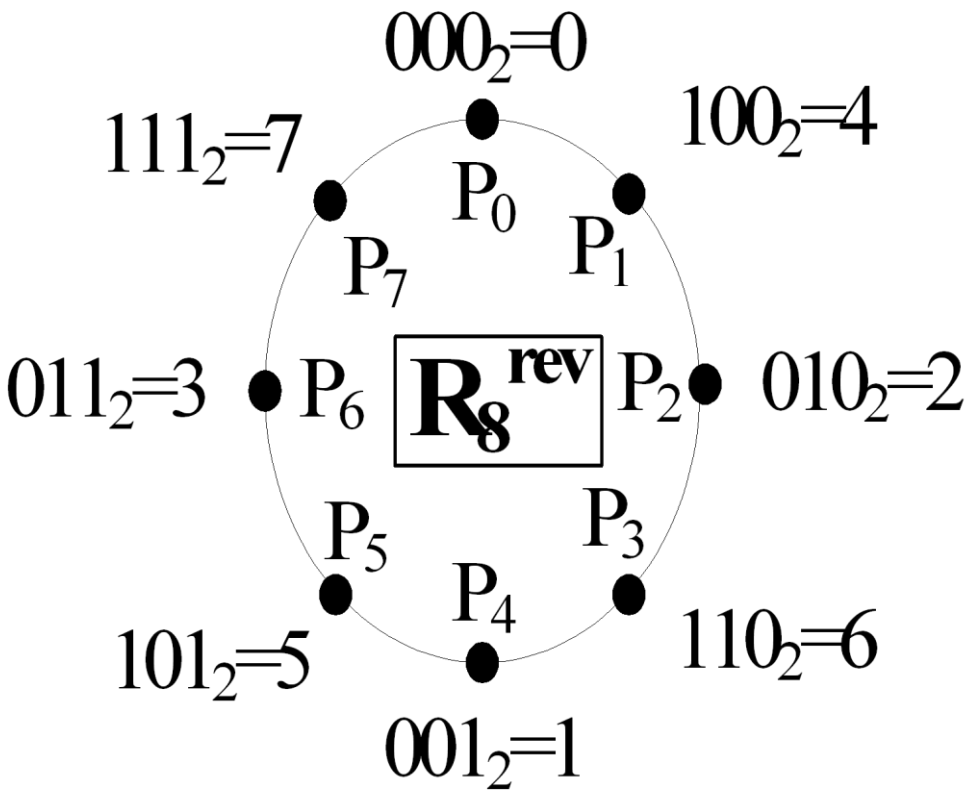
(2) S_n上发送的msg总数。

（1）构造S_n（分2步构造）

① 定义大小为n的环 R_n^{rev}：

对∀i∈[0，n-1]，设P_i的id为rev(i)，这里rev(i)是i的二进制表示的逆序列。

§3.4.2 有限制算法的下界Ω(nlgn)



例如，当n=8时有：

若将环 R_n^{rev} 划分为长度为j(j是2的方幂)的连续片断，则所有这些片断是序等价的(Ex3.9)。

片断数：n / j.

例如，4,2,6,1和5,3,7,0序等价

2,6,1,5和3,7,0,4序等价

§3.4.2 有限制算法的下界Ω(nlgn)

② 从R_n^{rev}构造S_n

将R^{rev}上每个id乘以(n+1)再加上n所获得的S_n是有空隙环。

但这种变化不改变片断的序等价性。

(2) S_n上发送的msg总数（分3步）

①求S_n中序等价的邻居集数目（引理3.19）

②由①证明算法里主动轮数目下界（引理3.20）

③由①求每个主动轮里发送msg数目的下界(引理3.21)

由②和③的组合即可获得算法的msg复杂性下界。

§3.4.2 有限制算法的下界Ω(nlgn)

Lemma3.19 对所有k<n/8以及每个S_n的k-邻居集N，在S_n中与N序等价的k-邻居集的个数(包括N本身)大于 $\frac{n}{2(2k+1)}$

Pf： N由2k+1个id构成，设j是大于2k+1的2的最小方幂。将S_n划分为n/j个连续片断，使某一片段包含N。

由S_n的构造可知，上述划分所得的所有片段均是序等价的。因此，至少有n/j个邻居集和N是序等价的。

设$j=2^i$，∵$2^{i-1}<2k+1<2^i$，∴$j<2(2k+1)$

故与N序等价的邻居集数目> $\frac{n}{j} > \frac{n}{2(2k+1)}$

§3.4.2 有限制算法的下界Ω(nlgn)

Lemma3.20 在exec(S_n)里，主动轮的数目至少为n/8。

Pf：设主动轮数目T<n/8。//反证法

设P_i是exec(S_n)里被选为leader的结点，由引理3.19知，与P_i的T-邻居集序等价的T-邻居集个数大于

$$\frac{n}{2(2T+1)} > \frac{n}{2(2n/8+1)} = \frac{2n}{n+4}$$

$$\because\ n \geq 8, \quad \therefore\ \frac{2n}{n+4} > 1$$

因此，存在某个异于P_i的结点P_j，P_j的T-邻居集与P_i的T-邻居集是序等价的。由引理3.17知，P_j与P_i的行为相似，故P_j亦被选为leader，这与A是正确的算法矛盾！□

§3.4.2 有限制算法的下界Ω(nlgn)

Lemma3.21 对于$\forall k \in [1, n/8]$，在exec(S_n)的第k个主动轮里，至少有 $n/2(2k+1)$ 个msg被发送。

Pf：考虑第k个主动轮，因它是主动的，故该轮里至少有一结点发送一个msg，不妨设P_i发送一个msg。

由引理3.19知，与P_i的k-邻居集序等价的k-邻居集个数大于 $n/2(2k+1)$，因为每个k-邻居集中至少有一个结点(k≥1)，这也就是说至少有 $n/2(2k+1)$ 个结点，其k-邻居集与P_i的k-邻居集序等价。

因此，由引理3.17知，这些结点与P_i的行为相似，即它们在第k个主动轮中发送msg。

§3.4.2 有限制算法的下界Ω(nlgn)

由引理3.20和3.21知，在exec(S_n) 里发送msg的总数至少为：

$$\sum_{k=1}^{n/8}\frac{n}{2(2k+1)} \geq \frac{n}{6}\sum_{k=1}^{n/8}\frac{1}{k} > \frac{n}{6}\ln\frac{n}{8}$$

即Ω(nlgn)，Th3.18得证。□

注意：为了使上述定理成立，要求标识符是取自集合{0,1,…,n^2+2n-1}。//该集合的势为n^2+2n。

原因是S_n中最小标识符为n，最大标识符为n^2+n-1 =(n+1)*rev(n-1)+n。

但是证明所用到的引理3.17要求算法在比S_n中最小标识符小n及最大标识符大n的所有标识符上是可以比较的。//有空隙环。

§3.4.2 有限制算法的下界Ω(nlgn)

3.时间受限算法的下界

下面的定义中，算法的时间不依赖于id，仅受限于环大小n，即使id可能是无界的(因为它们来自于自然数集合)。

Def3.4 若对任意的n，当标识符取自于自然数集合时，在所有大小为n的环上同步算法A的最坏时间是有界的，则称A为时间有界(或时间受限time-bounded)

证明时间受限的同步算法的msg复杂性的下界的基本思想是：

将时间受限算法映射为基于比较的算法。从而获得时间受限算法的msg复杂性下界Ω(nlgn)

§3.4.2 有限制算法的下界Ω(nlgn)

因为基于比较的算法的msg下界是针对n为2的方幂讨论的(虽然对所有n值成立)，故下面仍针对n为2的方幂情况来讨论。

为了将时间受限算法映射到基于比较的算法，需要定义在某一时间界限内算法的行为。

Def3.5 设R_1和R_2是任意两个大小为n的序等价的环，若每对匹配的结点在exec(R_1)和exec(R_2)的第1至t轮里有相似的行为，则同步算法A对于环大小为n的标识符集合S是基于t-比较的。

直观上，S上的一个基于t-比较的算法可看做这样一个算法：

它的行为与一个基于比较的算法的前t轮的行为相同，只要基于比较算法的标识符也选自于S；

若算法在t轮内终止，则它和S上基于比较的算法在所有轮上完全相同。

§3.4.2 有限制算法的下界Ω(nlgn)

首先要说明每个时限受限算法的行为和一个输入子集上的基于比较算法的行为相同(假设输入集合足够大)。为此须用Ramsey定理的有限版本。非形式地，Ramsey定理陈述：

设有一个集合S，若用t种颜色中的一种对每个大小为k的子集着色，则我们能够找到某个大小为 l 的子集使得它的所有大小为k的子集有相同的颜色。若将着色看做等价类划分(若两个大小为k的子集着相同颜色，则它们属于同一等价类)，则该定理说明存在一个大小为 l 的集合，其所有大小为k的子集是同一个等价类。

稍后，我们将对匹配结点行为相似的环着上相同颜色。

例：面试老师分为t组，每组有k个老师；面试学生集S中任何人可以到任何一组面试，则能找到某个 l 值，这 l 个学生中所有k个人的子集是在同一房间面试的。

§3.4.2 有限制算法的下界Ω(nlgn)

Ramsey's Theorem (finite version)

对所有正整数k，l 和t，存在一个整数 f(k,l, t)使得对每一个大小至少为 f(k,l, t)的集合S，对S的k-元子集的每一个t-着色，S的某一个l-元子集中所有k-元子集具有同样的颜色(l ≥k) 。（着色⇔等价类划分）

在下面的引理中，用Ramsey定理将任何时间受限算法映射到基于比较的算法上。

Lemma3.22 设A是一个运行时间为r(n)的时间受限的同步算法，则对于每个n，存在一个具有n^2+2n个id的集合C_n，使得A是C_n上的一个基于r(n)-比较的算法，这里n是环大小。

§3.4.2 有限制算法的下界Ω(nlgn)

Pf：固定n。设Y和Z是N(自然数集合)的任意两个n-元子集。

Y和Z称为等价子集，若对于每对序等价的环R_1和R_2(R_1和R_2中的标识符分别来自于Y和Z)，匹配结点在exec(R_1)和exec(R_2)的第1至r(n)轮里均有相似的行为。

该定义将N的n-元子集划分为有限多个等价类，因为行为相似仅指是否发送和接收msg，是否终止。我们对N的n-元子集着色使得两个n-元子集颜色相同当且仅当它们在同一等价类中。

由Ramsey定理，若设t是等价类(颜色)的数目，l 为n^2+2n，k为n，则因为N是无限集，存在一个势为n^2+2n的子集(N的子集)C_n，使得C_n的所有n-元子集属于同一个等价类。//N相当于定理中的S

§3.4.2 有限制算法的下界Ω(nlgn)

考虑大小为n，标识符来自C_n的两个序等价的环R_1和R_2，设Y和Z分别是R_1和R_2的标识符集合，Y和Z显然均是C_n的n元子集，所以它们属于同一个等价类。

由等价子集定义知：匹配的结点在exec(R_1)和exec(R_2)的第1至r(n)轮里均有相似的行为。因此，A是C_n(环大小为n)上的一个基于r(n)-比较的算法(由def3.5)□

Note：因为上述引理中算法A中的标识符是特定的，故还不能直接用Th3.18导出其msg复杂性为Ω(nlgn)。因此，须使用A来构造A’，使A’的复杂性与A相同，且ids来自于集合{0,1,…,n^2+2n-1}。因为基于比较的算法的ids来自于该集合。

§3.4.2 有限制算法的下界Ω(nlgn)

Th3.23 对每个同步的时间有界的leader选举算法A，以及每个n>8，n为2的方幂，存在一个大小为n的环R，使得A在R上的容许执行里发送Ω(nlgn)个msgs。

Pf：设A是运行时间为r(n)且满足定理假设的算法。固定n，设C_n是满足引理3.22的标识符集合，$c_0,c_1,\ldots c_{n^2+2n-1}$是$C_n$中按递增序排列的元素。

下面构造算法A'是基于比较的，它所执行的环大小为n，标识符集为$\{0,1,\ldots,n^2+2n-1\}$，它和A有同样的时间和msg复杂性。

§3.4.2 有限制算法的下界Ω(nlgn)

在算法A’中，一个标识符为i的结点执行算法就好像A在标识符C_i上执行一样。因为A在C_n上是基于r(n)-比较的且A在r(n)轮里终止，所以A’在大小为n的环上是基于比较的，且环上标识符来自于集合$\{0,1,\ldots,n^2+2n-1\}$。

由定理3.18，存在一个标识符来自于$\{0,1,\ldots,n^2+2n-1\}$，大小为n的环，算法A’在该环上发送的msg为Ω(nlgn)。

由A’的构造方法知，在大小为n、标识符来自于C_n的环上，存在A的一次执行，它发送的msg个数与A’相同。故定理得证。□

Ex3.9 若将环 R_n^{left} 划分为长度为j(j是2的方幂)的连续片段，则所有这些片段是次序等价的。

第4章
分布式系统中的计算模型

2006.12.15

概述

- 分布式系统计算模型的复杂性
  - 系统由并发执行部件构成
  - 系统中无全局时钟
  - 必须捕捉系统部件可能的失效
- 对策
  - 因果关系（Causality）
  - 一致状态（Consistent states）
  - 全局状态

§4.1 基本知识

- 协议（Protocol）
- 协议中的控制语句

1.Send（destination, action; parameters）

destination：处理器抽象。实用中是通信实体的地址：
机器名，机器的端口号(即1个socket地址)

action：控制msg，希望接收者采取的动作

parameters：参数集合

假定：

msg发送是无阻塞、可靠的（语义类似于TCP套接字）；

有时假定较弱的msg传递层（等价于UDP）。

TCP与UDP的区别：①基于连接与无连接②对系统资源的要求（TCP较多，UDP少）
③UDP程序结构较简单④流模式与数据报模式

●TCP保证数据正确性，UDP可能丢包

●TCP保证数据顺序，UDP不保证

§4.1 基本知识

2.接收msg

接收msg可推广至接收事件，引起事件的原因是：

外部msg、超时设定、内部中断

事件在处理前，一般是在缓冲区(如事件队列)中，若一处理器想处理事件，它必须执行一个声明处理这些事件的线程。

例如，一个节点通过执行下述代码等待事件$A_1, A_2,..., A_n$

waiting for $A_1, A_2,..., A_n$ ： //声明

A_1（Source; parameters）

Code to handle A_1

….

A_n（Source; parameters）

Code to handle A_n

当p执行send(q, A_1; parameters)且q执行上述代码时，q将最终处理由p发送的msg

3.超时

当怀疑远程处理器失效时，可通过超时检测来判定：

①当T秒后仍未收到P的类型为event的msg时，执行指定的动作

```
waiting until P sends (event; parameters), timeout=T
    on timeout
        timeout action
```

②仅当收到一个响应msg时才采取动作，超时不做任何动作

```
waiting until P sends (event; parameters), timeout=T
    on timeout;
if no timeout occurred
   { Successful response actions }
```

3.超时

③处理器等待响应T秒

若处理器在等待开始后T秒内没响应，则等待结束，协议继续

waiting up to T seconds for (event; parameters) msgs

Event: < msg handling code >

§4.2 因果关系

分布式系统为何缺乏全局的系统状态？

1.非即时通信

A和B同时向对方喊话

他们都认为是自己先喊话

C听到两人是同时喊话

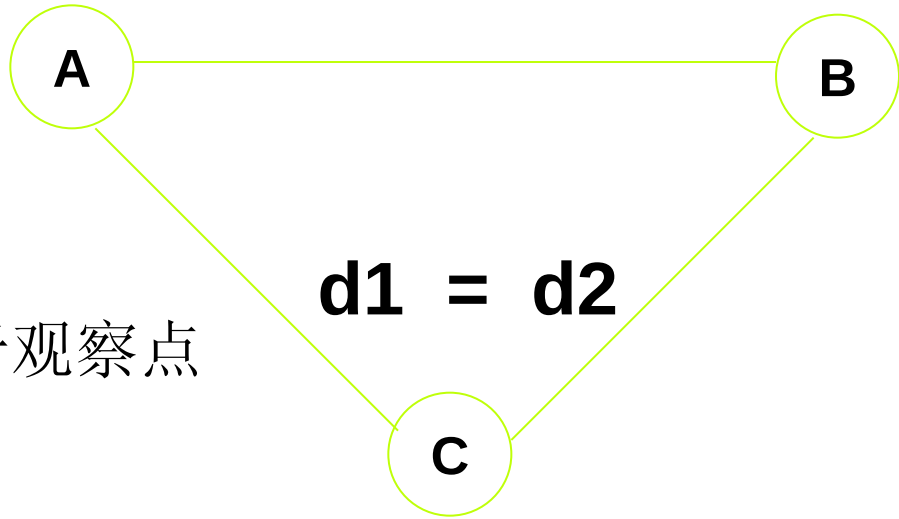
结论：系统的全局状态依赖于观察点

原因：

传播延迟

网络资源的竞争

丢失msg重发



§4.2 因果关系

2.相对性影响

假设张三和李四决定使用同步时钟来观察全局状态：

他们约定下午5点在某餐馆会面，张三准时到达，但李四在一个接近光速的日光系统中游览。

张三在等待李四1小时后离开餐馆，而李四在自己的表到达5点时准时达到餐馆，但他认为张三未达到。

因为大多数计算机的实际时钟均存在漂移，故相对速度不同，时钟同步仍然是一个问题。

结论：使用时间来同步不是一个可靠机制。

§4.2 因果关系

3.中断

假设张三和李四在同一起跑线上赛跑，信号为小旗，前两个问题可以忽略，但是...

即使可忽略其他影响，也不可能指望不同的机器会同时做出某些反应。因为现代计算机是一个很复杂的系统：CPU竞争、中断、页错误等，执行时间无法预料。

结论：不可能在同一时刻观察一个分布式系统的全局状态

必须找到某种可以依赖的性质：

- 时间回溯
- 因果相关

§4.2 因果关系

- 假设分布式系统构成：

  $P=\{P_1, P_2, \ldots, P_n\}$：处理器集合

  $\mathcal{E}$：全体事件的集合

  $\mathcal{E}_p \subseteq \mathcal{E}$，$\mathcal{E}_p$表示发生在p上的所有事件

- 次序 $e_1 < e_2$：事件e_1发生e_2在之前（亦记：$e_1 \rightarrow e_2$）

  $e_1 <_I e_2$：事件e_1发生e_2在之前，I为信息源

- 定序 有些$\mathcal{E}$中事件很容易定序：

  – 发生在同一节点p上的事件满足全序：

  若$e_1, e_2 \in \mathcal{E}_p$，则 $e_1 <_p e_2$ 或 $e_2 <_p e_1$ 成立

  –e_1发送消息m, e_2接收m，则$e_1 <_m e_2$

§4.2 因果关系

- Happens-before关系（$<_H$）

  该关系是节点次序和消息传递次序的传递闭包：

  - 规则1：若$e_1 <_p e_2$，则$e_1 <_H e_2$
  - 规则2：若$e_1 <_m e_2$，则$e_1 <_H e_2$
  - 规则3：若$e_1 <_H e_2$，且$e_2 <_H e_3$，则$e_1 <_H e_3$

  在集合 X 上的二元关系 R 的传递闭包是包含 R 的 X 上的最小的传递关系。

  $e_1 <_H e_3$表示存在1个事件因果链，使e_1发生e_3在之前

  Note：$<_H$是一种偏序关系，即

  存在e和e’，二者之间无这种关系

  - 并发事件：若两事件不能由$<_H$定序

§4.2 因果关系

- Happens-before关系（$<_H$）
  - 规则1表述的是同一处理器上两个事件之间的因果关系；
  - 规则2表述了不同处理器上两个事件之间的因果关系
  - 规则3阐述了传递律

Note:

Happens-Before关系完整表述了执行中的因果关系。如果一个执行中的事件按照其相对顺序重新进行排序，而不改变他们的happens-before关系的话，那么其结果仍是一个执行，并且对处理器而言，该执行与原执行并无区别。

§4.2 因果关系

- 举例

1) 因事件e1, e4和e7均发生在P1上，故：$e_1 <_{P1} e_4 <_{P1} e_7$

2) 因e1发送1个msg到e3，故：$e_1 <_m e_3$，类似地$e_5 <_m e_8$

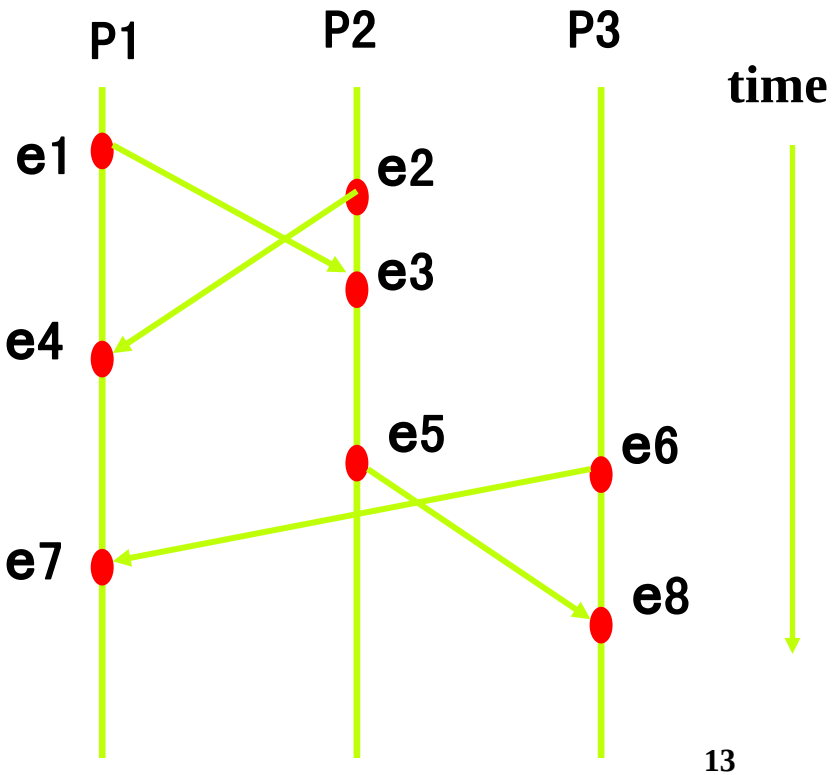
3) 应用规则1和2得：

$e_1 <_H e_4 <_H e_7$，$e_1 <_H e_3$，$e_5 <_H e_8$

4) 由$<_H$的传递闭包性质得：

$e_1 <_H e_8$

5) e1和e6是并发的：e1和e6之间无路径



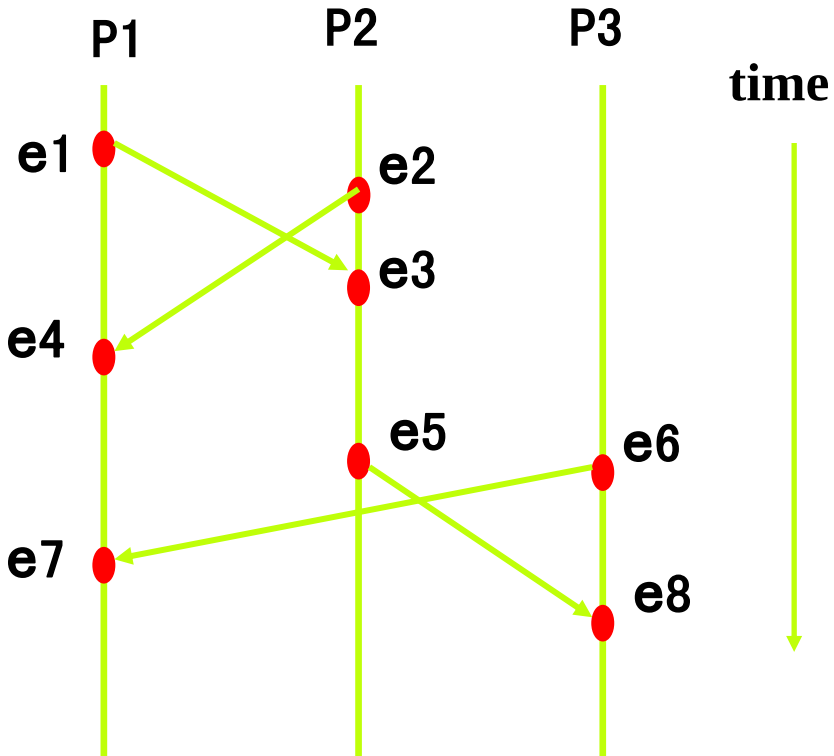
§4.2 因果关系

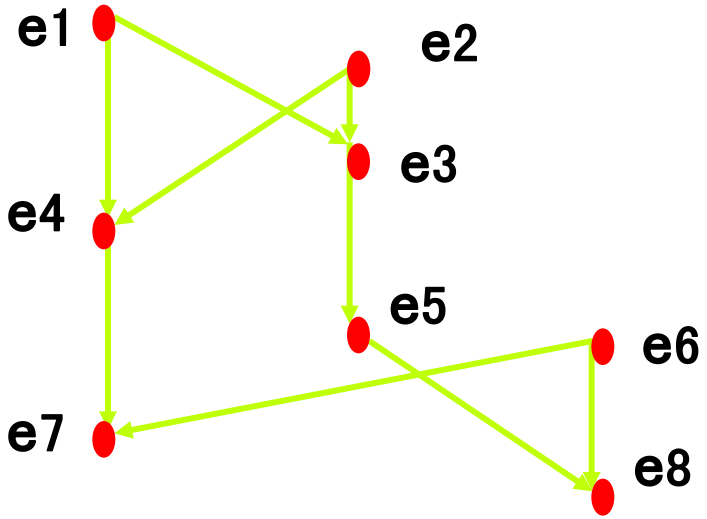
- H－DAG

  有时将Happens-before关系描述为一个有向无环图

  –顶点集V_H是事件集$\mathcal{E}$：$e \in V_H$ 当且仅当 $e \in \mathcal{E}$

  –边集E_H：若$(e_1, e_2) \in E_H$ 当且仅当$e_1 <_P e_2$或$e_1 <_m e_2$





§4.2.1 Lamport时间戳

- 系统有序性的重要性

  若分布式系统中存在全局时钟，则系统中的事件均可安排为全序。例如，可以更公平地分配系统资源。

- 全序对事件的影响和由H关系确定的偏序对事件的影响是一致的
- 如何通过H关系确定的偏序关系来建立一个“一致”的全序关系？
  - 在$<_H$的DAG上拓扑排序
  - On the fly：Lamport提出了动态即时地建立全序算法

§4.2.1 Lamport时间戳

- Lamport算法的思想

  每个事件e有一个附加的时戳：e.TS

  每个节点有一个局部时戳：my_TS

  每个msg有一个附加时间戳：m.TS

  节点执行一个事件时，将自己的时戳赋给该事件；

  节点发送msg时，将自己的时戳赋给所有发送的msg。

§4.2.1 Lamport时间戳

- Lamport算法的实现

Initially：my_TS=0;

On event e：

if (e是接收消息m) then

my_TS = max (m.TS, my_TS);

//取msg时戳和节点时戳的较大者作为新时戳

my_TS++;

e.TS＝my_TS; //给事件e打时戳

if (e是发送消息m) then

m.TS＝my_TS; //给消息m打时戳

§4.2.1 Lamport时间戳

- Lamport算法赋值的时戳是因果相关的

  若$e_1 <_H e_2$，则 $e_1.TS < e_2.TS$

  ∵若$e_1 <_P e_2$或$e_1 <_m e_2$，则的e_2时戳大于e_1的时戳

  ∴在因果事件链上，每一事件的时戳大于其前驱事件的时戳

- 问题：系统中所有事件已为全序？

  不同的事件可能有相同的时戳（并发事件）

- Lamport算法改进

  因为并发事件的时戳可以任意指定先后

  故可用节点地址作为时戳的低位

§4.2.1 Lamport时间戳

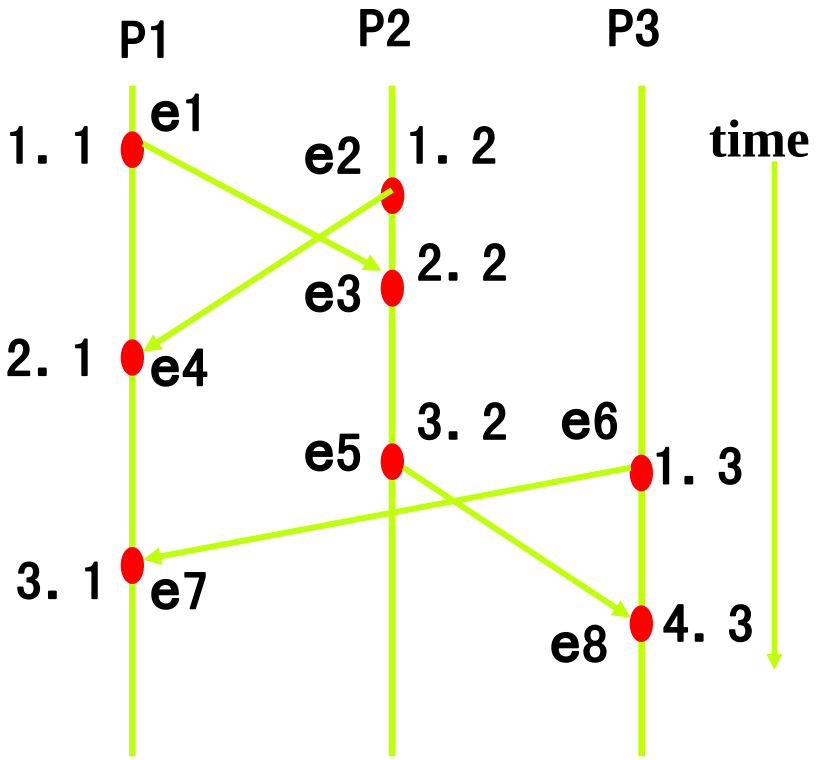
- 改进的Lamport时戳

事件标号：时戳.id

事件e8为4.3:

my_TS=max(m.TS,my_TS)

=max(3,1)=3



按字典序得全序：

1.1, 1.2, 1.3, 2.1, 2.2, 3.1, 3.2, 4.3

- 算法特点：分布、容错、系统开销小

- Lamport算法的迷人之处在于：任何进程在发送消息前，先将自己的本地计数器累加1；接收进程总是计算自己的本地计数器和接受到计数器中较大值加上1的结果

§4.2.2 向量时间戳

- Lamport时戳缺点

  若$e_1 <_H e_2$，则 $e_1.TS < e_2.TS$；反之不然。

  例如：1.3<2.1，但是$e_6 < e_4$不成立

  原因：并发事件之间的次序是任意的

  不能通过事件的时戳判定两事件之间是否是因果相关

- 判定事件间因果关系的重要性

  例子：违反因果关系检测

  在一个分布式对象系统中，为了负载平衡，对象是可移动的，对象在处理器之间迁移是为了获得所需的调用的进程或资源。如下图：

§4.2.2 向量时戳

1)P1持有对象O，决定迁移到P2

为获取资源，P1将O装配在消息M1中发送给P2

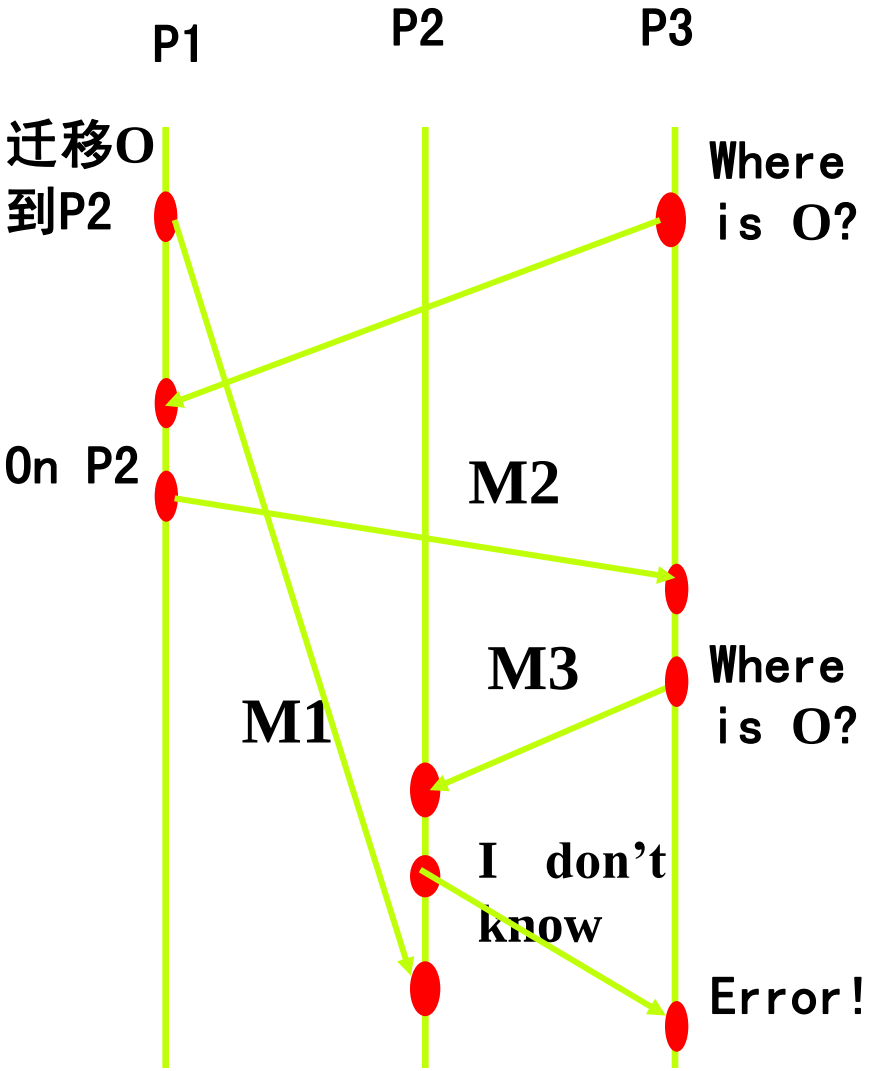
2)P1收到P3访问O的请求

P1将O的新地址P2放在消息M2中通知P3

3)P3在M3中请求访问P2的O

当M3达到P2时，O不可用，故回答一个出错消息。

- 问题：当debug该系统时，会发现O已在P2上，故不知错在哪？



§4.2.2 向量时间戳

- 错误原因：违反因果序

P3请求O是发生在从P1迁移到P2之后，但该请求被处理是在迁移达到P2之前。形式地，

设：s(m)是发送m的事件

r(m)是接收m的事件

若$s(m1)<_H s(m2)$，则称消息m1因果关系上先于m2，记做$m1<_C m2$

若$m1 <_C m2$，但 $r(m2) <_P r(m1)$，则违反“因果关系”：

即，若m1先于m2发送，但在同一节点P上m2在m1之前被接收

例如，在上例中有：

$M1<_C M3$，但 $r(M3) <_{P2} r(M1)$

§4.2.2 向量时间戳

- 违反因果序检测
  - 定义：若时戳VT具有比较函数$<_V$满足：

    $e1 <_H e2$ iff $e1.VT <_V e2.VT$

    则我们能够检测出是否违反因果关系
  - VT性质

    1）因$<_H$是偏序，故$<_V$也是偏序；

    2）因为必须知道在因果关系上每一节点中哪些事件是在事件e之前，故e.VT中必须包含系统中每一个其它节点的状态。

    这两个性质导致了向量时戳VT的引入

§4.2.2 向量时戳

- 向量时戳VT

VT是一个整数数组：

e.VT[i]=k表示在节点i（或P_i）上，因果关系上e之前有k个事件（可能包括e自己）。

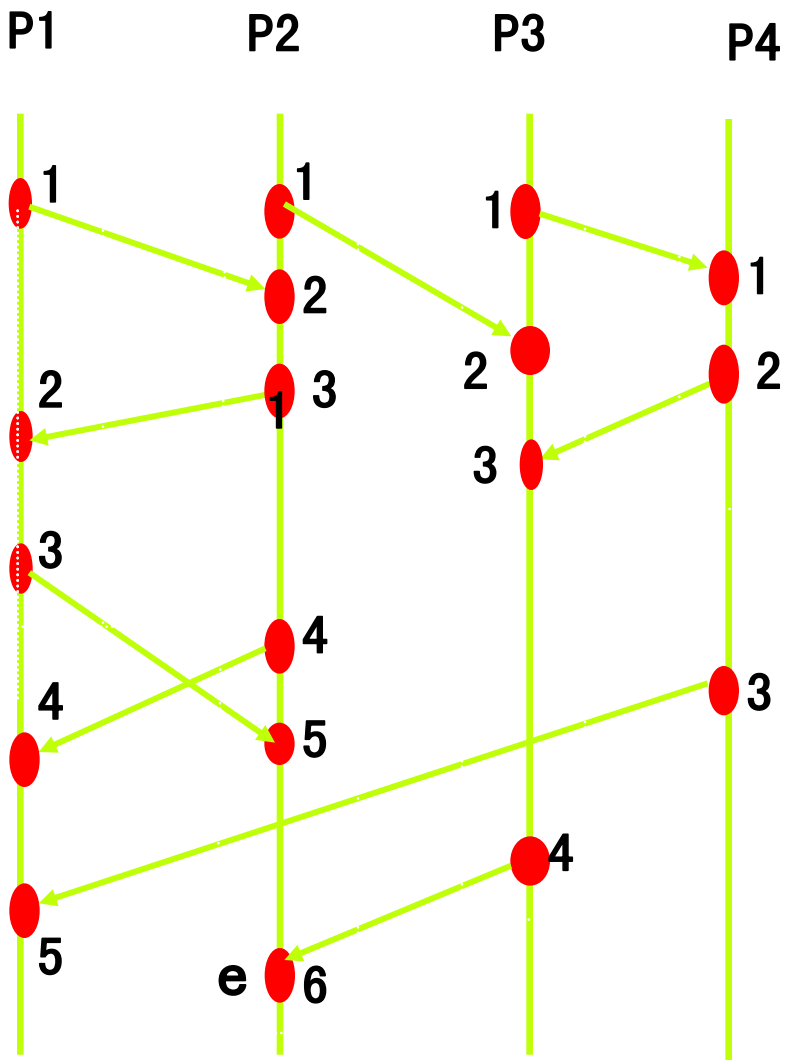
e.VT=(3,6,4,2)表示因果序：

在P1上，有3个事件须在e之前

在P2上，有6个事件须在e之前

在P3上，有4个事件须在e之前

在P4上，有2个事件须在e之前



§4.2.2 向量时间戳

- 向量时戳的意义

  在因果关系上，e1.VT $\leqslant_V$ e2.VT表示e2发生在e1及e1前所有的事件之后。更精确的说，向量时钟的次序为：

  e1.VT$\leqslant_V$ e2.VT iff e1.VT[i]$\leqslant$ e2.VT[i], i=1, 2, …, M

  e1.VT$<_V$ e2.VT iff e1.VT$\leqslant_{VT}$e2.VT且e1.VT$\neq$e2.VT

- 向量时戳算法

  my_VT：每个节点有局部的向量时戳

  e.VT：每个事件有向量时戳

  m.VT：每个msg有向量时戳

  节点执行一个事件时，将自己的时戳赋给该事件；

  节点发送msg时，将自己的时戳赋给所有发送的msg。

注意$\leq_V$, $\leq_{VT}$以及$<_V$之间的区别：V代表因果序，而VT代表时戳比较

§4.2.2 向量时间戳

- 算法实现

Initially：my_VT=[0,…,0];

On event e：

```
if  ( e是消息m的接收者 ) then
    for i=1 to M do //向量时戳的每个分量只增不减
        my_VT[ i ] = max( m.VT[i], my_VT[i] );
my_VT[ self ]++; //设变量self是本节点的名字
e.VT＝my_VT; //给事件e打时戳
if ( e是消息m的发送者 ) then
    m.VT＝my_VT; //给消息m打时戳
```

§4.2.2 向量时间戳

- 算法性质

1）若$e <_H e'$，则$e.VT <_{VT} e'.VT$

∵算法确保对于每个事件满足：

若$e <_P e'$或$e <_m e'$，则$e.VT <_{VT} e'.VT$

2）若$e \not<_H e'$，则$e.VT \not<_{VT} e'.VT$

pf：若e和e' 因果相关，则有$e' <_H e$，即$e'.VT <_{VT} e.VT$

若e和e‘ 是并发的，则在H-DAG上，从e到e’和从e’到e均无有向路径，即得：

$e.VT \not<_{VT} e'.VT$ 且 $e'.VT \not<_{VT} e.VT$

当且仅当e.VT和e’.VT是不可比时，称向量时戳是捕获并发的！

§4.2.2 向量时戳

- 向量时戳比较

e1.VT=(5,4,1,3)

e2.VT=(3,6,4,2)

e3.VT=(0,0,1,3)

1）e1和e2是并发的

∵e1.VT[1]>e2.VT[1]

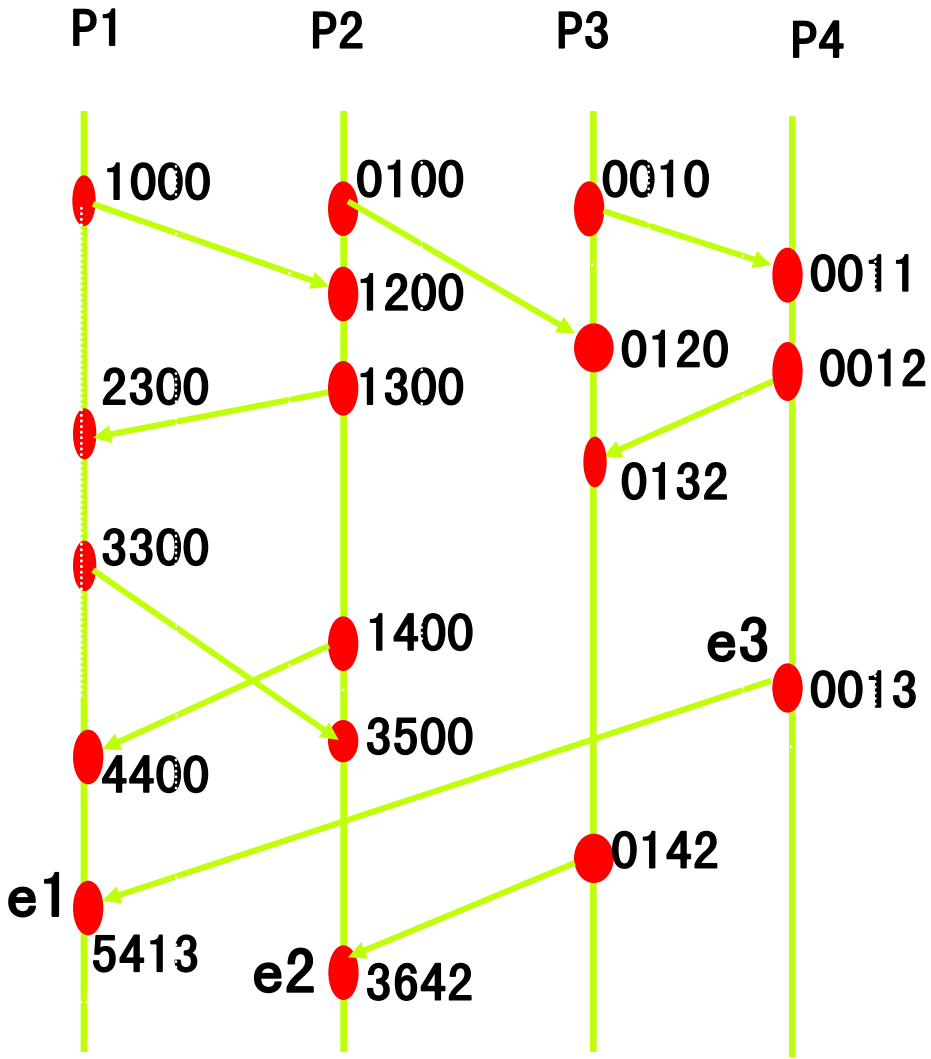
e1.VT[2]<e2.VT[2]

∴e1到e2及e2到e1均无路径

2）e3在因果序上先于e1

即：$e3.VT <_V e1.VT$

e1的前驱事件见方框



§4.2.2 向量时戳

- 因果序检测

1) 消息时戳间比较

在P2上，先到达的M3的时戳为（3,0,3），后到达的M1的时戳为（1,0,0）。但是：

∵ $(1,0,0) <_V (3,0,3)$

∴ M1在因果序上先于M3

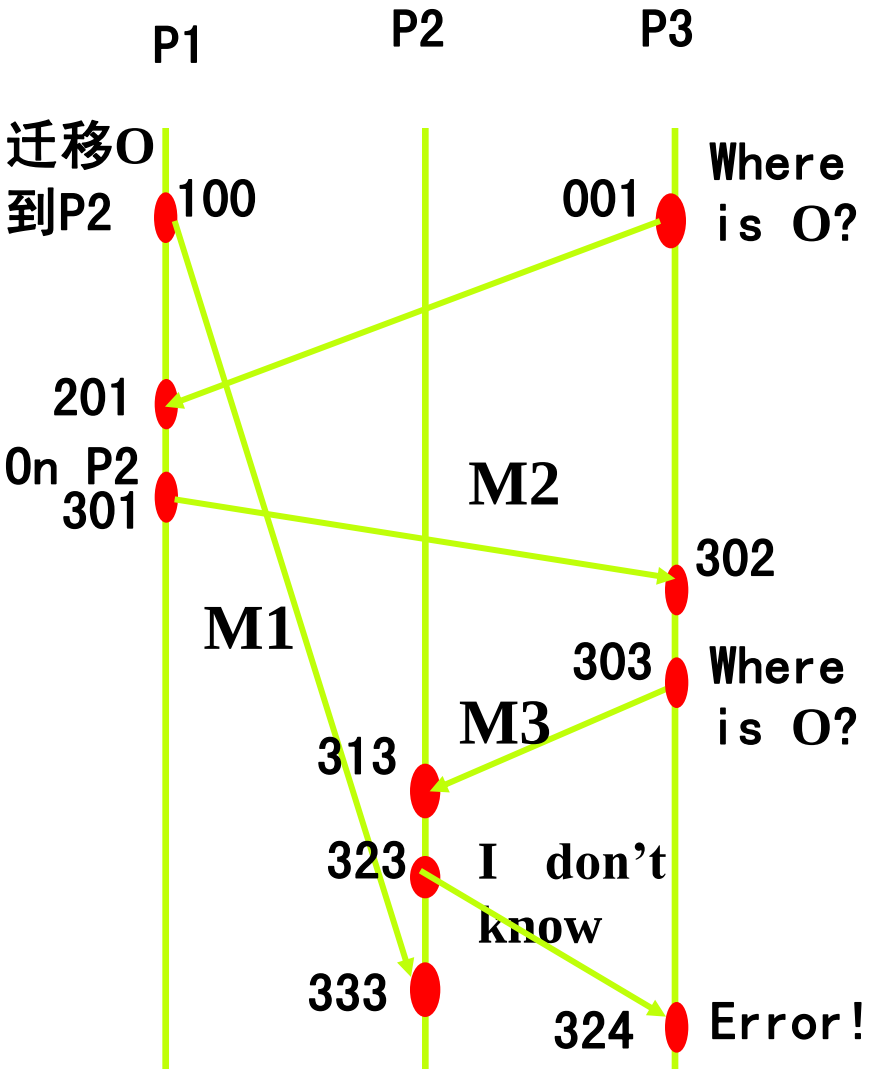
故M1后于M3到达违反因果序

2) 消息时戳和局部时戳比较

当时戳为(1,0,0) 的M1到达P2时，P2的时戳是(3,2,3)。但：

∵ $(1,0,0) <_V (3,2,3)$

∴ M1在因果序上应先于(3,2,3)对应的事件



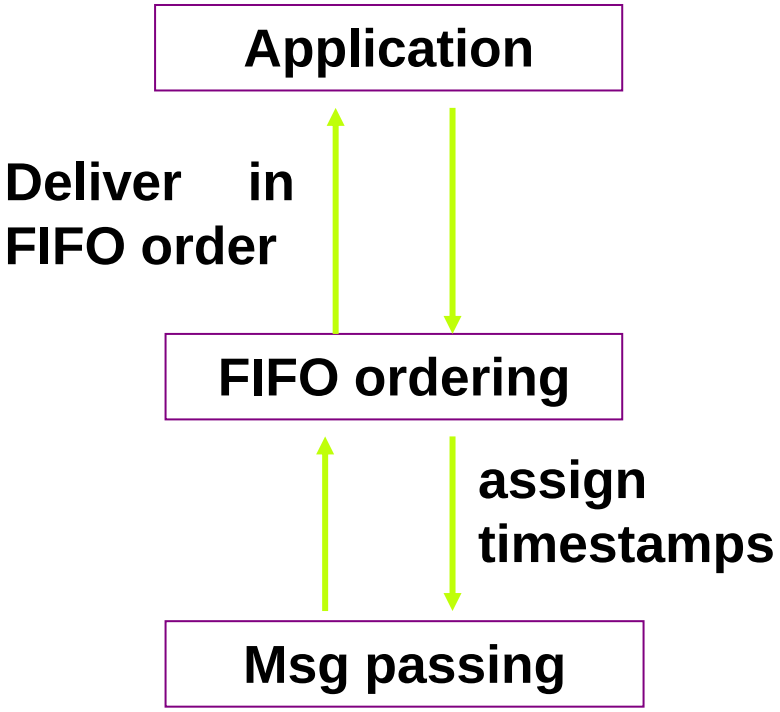
§4.2.3 因果通信

- 如何保证通信不违反因果关系？

  处理器不能选择msg达到的次序，但能抑制过早达到的msg来修正传递（指提交给应用）次序。

- FIFO通信（如TCP）

  由msg传递协议栈里的一层负责确保FIFO通信



§4.2.3 因果通信

- FIFO通信

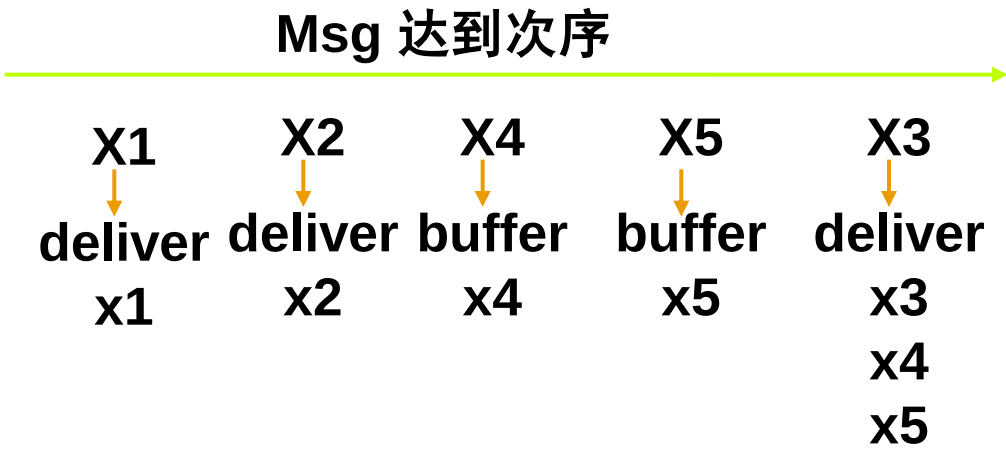
源处理器给每个发送的msg顺序编号，目的处理器知道自己所收到的msg应该有何顺序的编号，若目的处理器收到一个编号为x的msg但未收到较小编号的msg时，则延迟传递直至能够顺序传递为止。

- 因果通信

因果通信与FIFO通信类似

源：附上时戳

目的地：延迟错序msg



§4.2.3 因果通信

- 因果通信实现思想
  - 抑制从P发送的消息m，直至可断定没有来自于其它处理器上的消息m’，使m’ $<_V$ m.
  - 在每个节点P上：

earliest[1..M]：存储不同节点当前能够传递的消息时戳的下界
earliest[k]表示在P上，对节点k能够传递的msg的时戳的下界
blocked[1..M]：阻塞队列数组，每个分量是一个队列

- Alg. Causal Msg delivery

定义时戳1_k： 若使用Lamport时戳，则1_k＝1；
若用向量时戳，则1_k＝(0,…,1,0,…,0)，k^{th}位为1

初始化

1: earliest[k] =1_k，k=1,…,M

2: blocked[k] ={ }，k=1,…,M//每个阻塞队列置空

§4.2.3 因果通信

3:On the receipt of msg m from node p:

4: delivery_list={};

5: if （blocked[p]为空）then

6: earliest[p]=m.timestamp;

7: 将m加到blocked[p]队尾；//处理收到的消息

8: while（∃k使blocked[k]非空 and 对每个i=1,…,M(除k和self外),
not_earliest(earliest[i], earliest[k], i)）{//处理阻塞队列
//对非空队列k，若其他节点i上无比节点k更早的msg要达到本
//地，则队列k的队首可解除阻塞

9: 将blocked[k]队头元素m’出队，且加入到delivery_list;

10: if （blocked[k]非空）then

11: 将earliest[k]置为m’.timestamp；

12: else increment earliest[k] by 1_k }//end while

13: deliver the msgs in delivery_list; //按因果序

§4.2.3 因果通信

- 向量时戳比较

```
not_earliest( proc_i_vts, msg_vts, i ) {//前者不早于后者时为真
    if  ( msg_vts[i]<proc_i_vts[i] )
            return turn;
     else   return false;
  }
```

- 分析　使用向量时戳较好，不会假定序。

1）初始化：在本地节点（self）上，能够最早传递的来自于节点k的msg的时戳存储在本地的earliest[k]中，line1初始化正确

2）处理接收的消息：当本地节点接收来自于p的消息m时

行5,6：若blocked[p]中无msg，则earliest[p]被置为m的时戳，这是因为从p上不会有更早的msg达到本地，更早的已处理过。

§4.2.3 因果通信

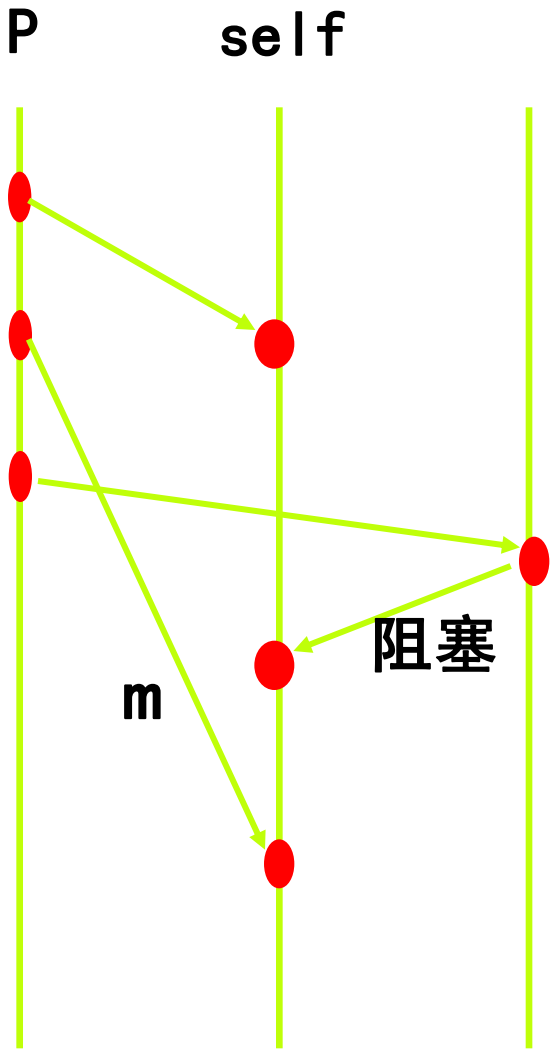
- 分析

行7：将m放入阻塞队列blocked[p]中，直到可安全传送为止。

3）处理阻塞：因m的达到能够将earliest[p]中的时戳更新（变大），故可能会使若干个阻塞的msg变为非阻塞；当然m本身可能是安全的，可直接传递给应用。

最终，一个阻塞msg变为非阻塞msg后，也可能使其他阻塞msg变为非阻塞msg。

行8：while循环检查阻塞队列，对于非空队列k进行处理。



§4.2.3 因果通信

- 分析

什么样的msg是非阻塞的？

当阻塞队列k（指blocked[k]）非空时，检测其能否解除阻塞：

设队头msg是m，在self上若其他节点i（除self和k之外）无更早时戳earliest[i](即比m.timestamp=earliest[k]更小)的msg可能被传递，则消息m是非阻塞的。此时，m可安全传递，m出队（行9）。

4）问题

上述算法可能会发生死锁：若一节点长时间不发送你要的msg，会发生死锁。

因此，上述因果通信算法通常被用于组播的一部分。

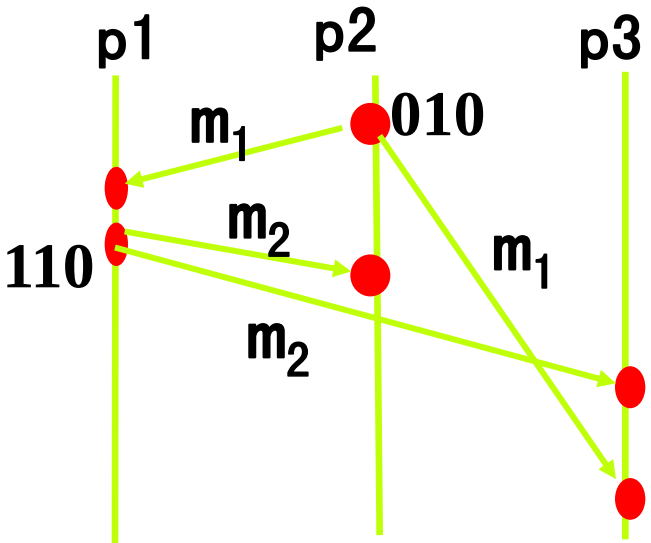
§4.2.3 因果通信

- 算法执行例子（Causal Multicast）

  因为协议只对发送事件的因果序感兴趣，故一节点只有发送msg时对向量时戳做增量操作。

①处理接收消息：当p3从p1接收m2时，修改earliest[1]为(1,1,0)，m2入阻塞队列blocked[1]；//line6，7

②处理阻塞：blocked[1]≠Φ，故while循环中k=1，self=3，i=2，not_earliest(earliest[2], earliest[1], 2)，前者早于后者，表示p2上有一个更早的msg还没有达到p3，返回假。m2被阻塞



在p3上

earliest[1]	earliest[2]	earliest[3]
(1,0,0)	(0,1,0)	(0,0,1)

接收m2　//k=1, i=2, 阻塞m2

m2		
（1,1,0） ≥	(0,1,0)	(0,0,1)

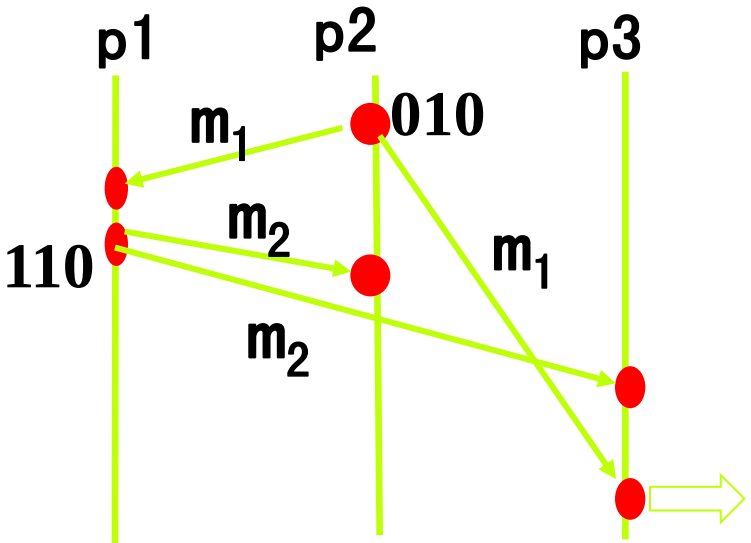
§4.2.3 因果通信

①处理接收消息：当p3从p2接收m1时，earliest[2]为(0,1,0)(不变)，m1入阻塞队列blocked[2]；//line6，7

②处理阻塞：while检测各阻塞队列是否有阻塞的msg。

k=2, blocked[2]≠Φ, 其中的m1不依赖其他事件，故放入传递表(行9)，m1出队后blocked[2]=Φ, earliest[2][2]+1(行12)后earliest[2] =(0,2,0);

k=1,blocked[1]≠Φ, self=3, i=2, not_earliest(earliest[2], earliest[1], 2)，前者不早于后者，表示p2上无更早的msg会达到p3，返回真。m2从blocked[1]出队入传递表(行9), blocked[1]=Φ，earliest[1][1]+1(行12)后earliest[1]=(2,1,0)。



接收m1

m2	m1	//k＝2, i=1
（1,1,0） >	(0,1,0)	(0,0,1)
deliver m1		
m2		//k＝1, i=2
（1,1,0） <	(0,2,0)	(0,0,1)
deliver m2		
（2,1,0）	(0,2,0)	(0,0,1)