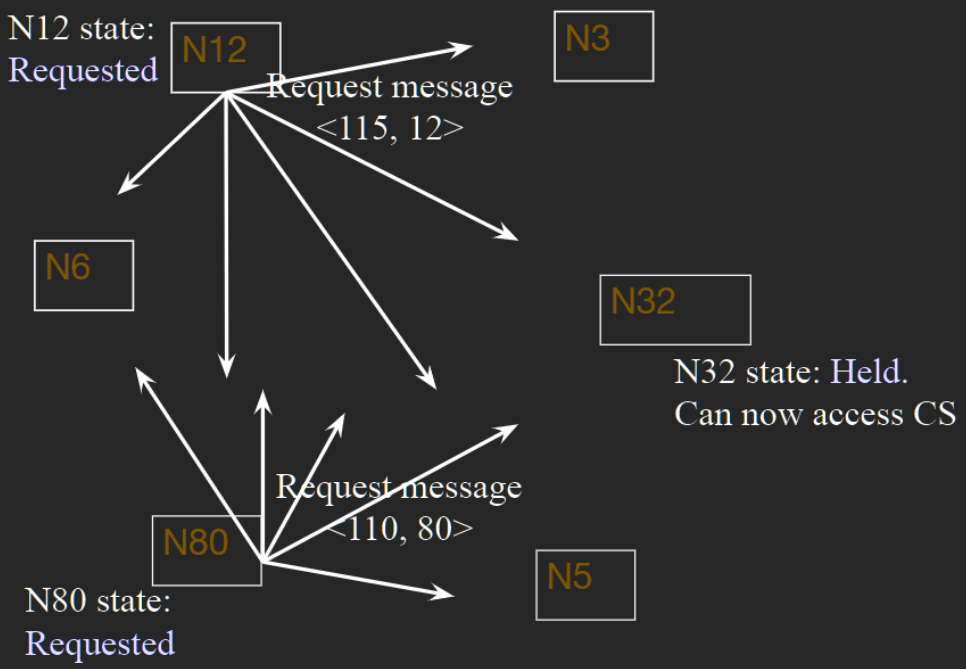
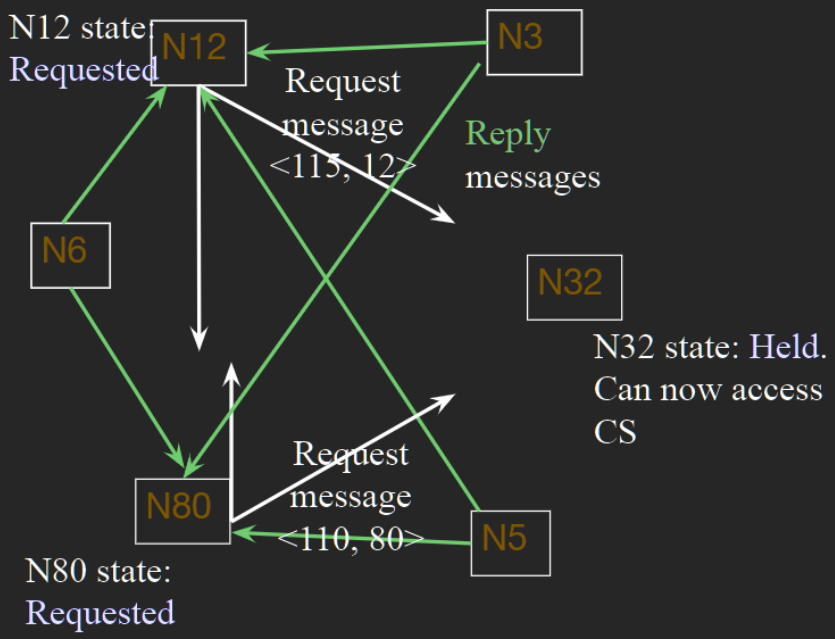
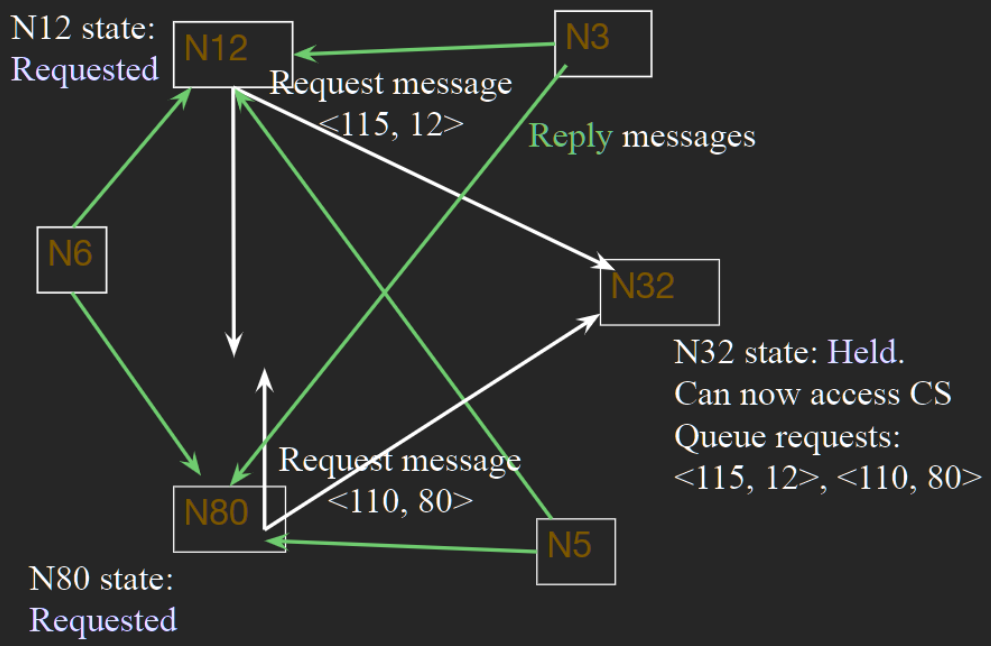
Example: Ricart-Agrawala Algorithm1



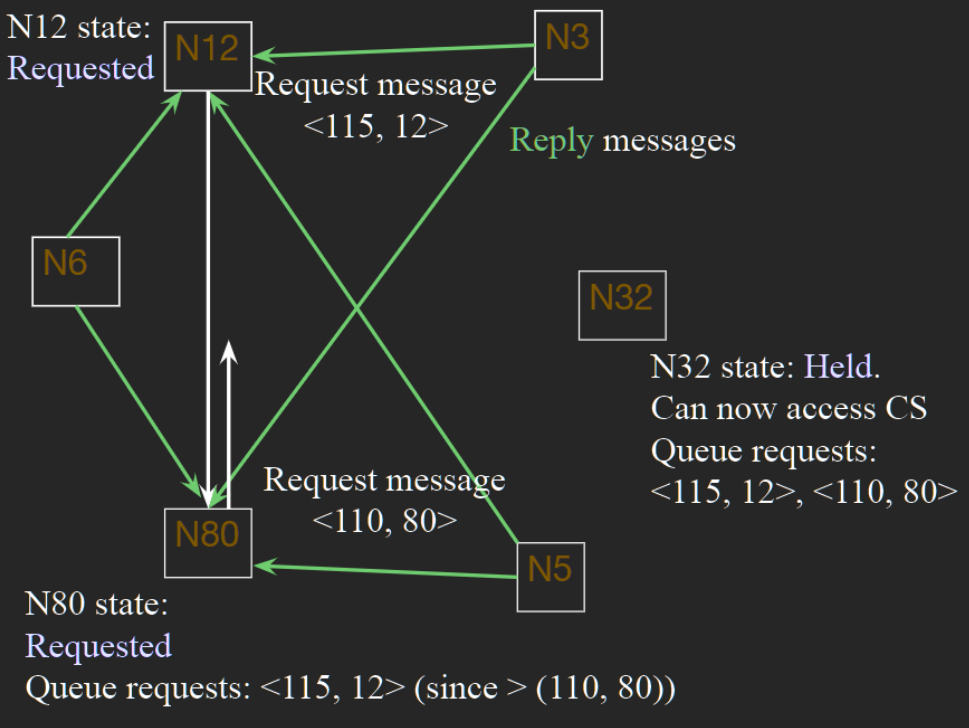
Example: Ricart-Agrawala Algorithm1



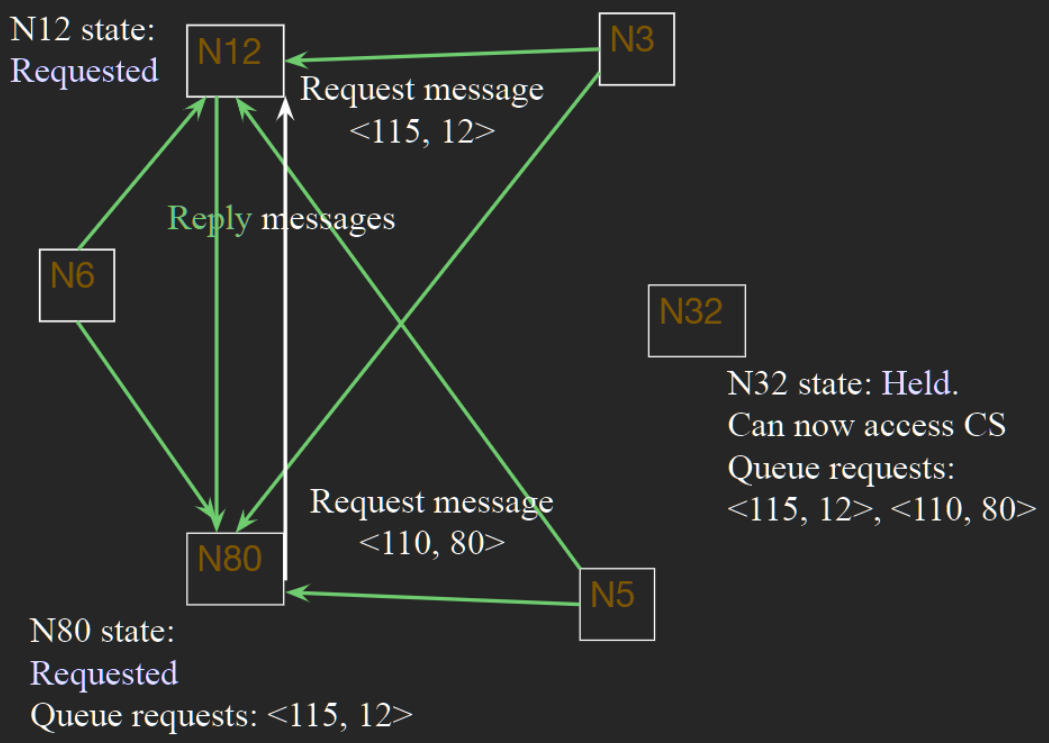
Example: Ricart-Agrawala Algorithm1



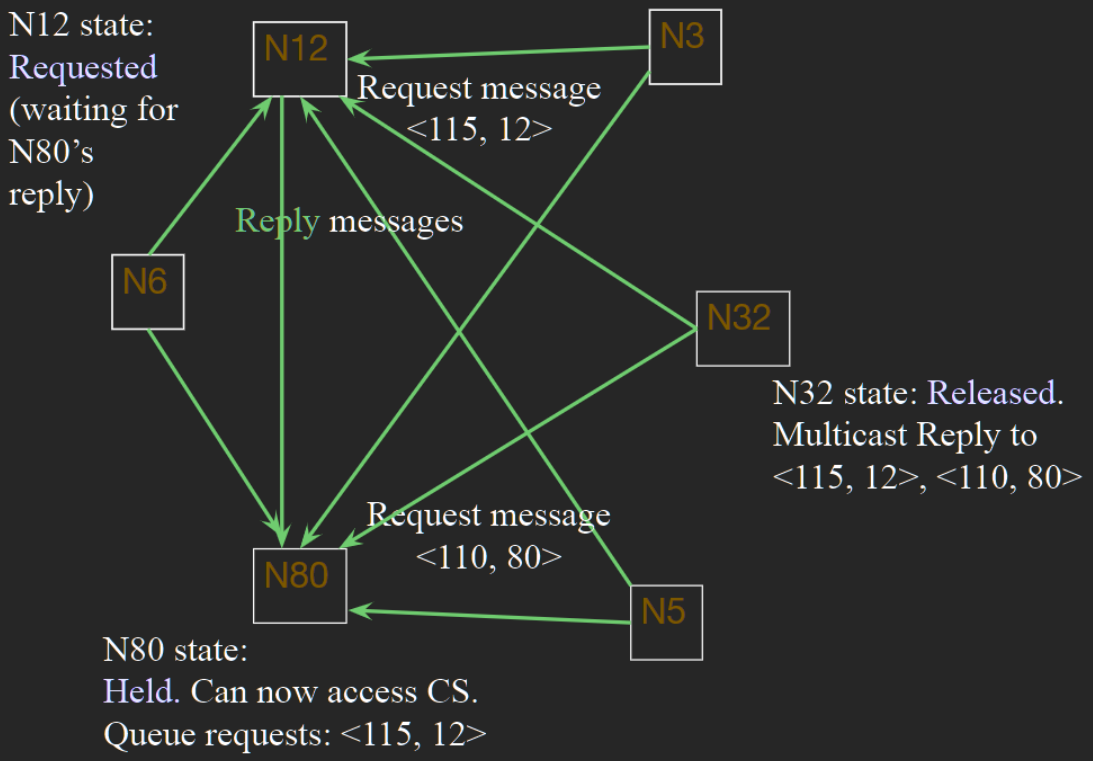
Example: Ricart-Agrawala Algorithm1



Example: Ricart-Agrawala Algorithm1



Example: Ricart-Agrawala Algorithm1



Token-Based Mutual Exclusion

基於Token的互斥

- Token Ring Algorithm

  令牌環算法
- Ricart-Agrawala Second Algorithm

  Ricart-Agrawara第二算法

Token Ring Algorithm

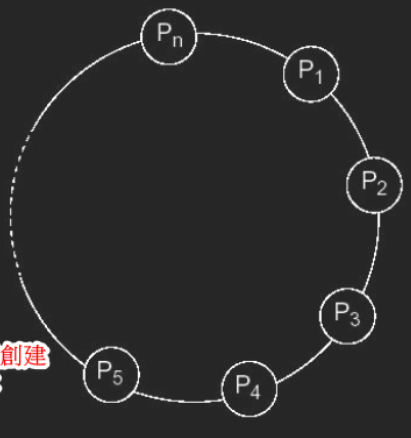
Token算法

- Simple - arrange the n processes in a logical ring.

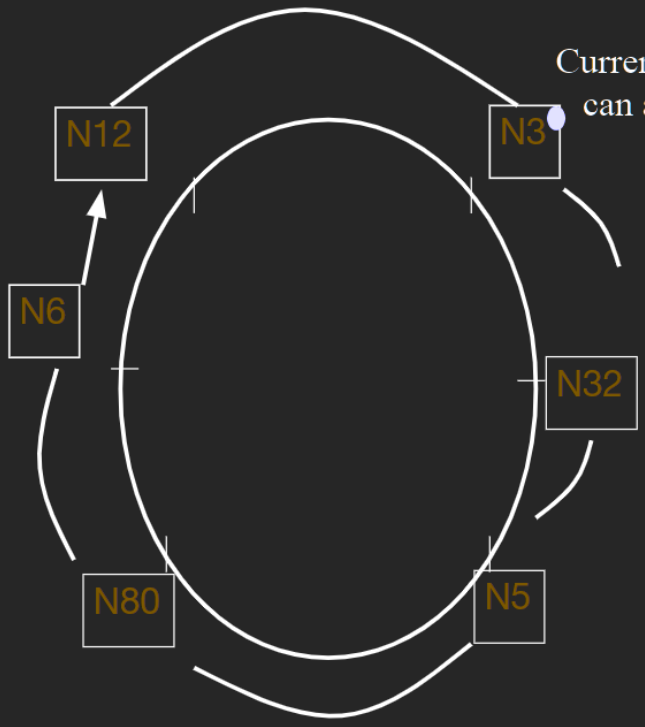
  簡單 - 安排 n 個進程在邏輯環
- The logical ring topology is created by giving each process the address of one other process which is its neighbor in the clockwise direction.

  邏輯環拓撲為通過給予每個過程創建另一個進程的地址這是它的鄰居順時針方向。
- Logical ring topology is unrelated to the physical interconnection between computers.

  邏輯環拓撲無關到物理互連計算機之間。

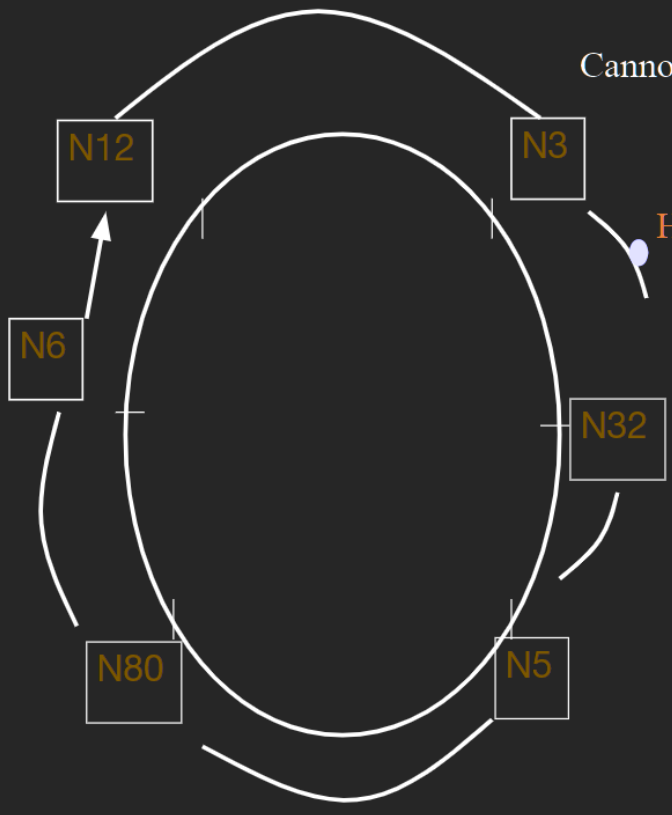


Token Ring-based Mutual Exclusion



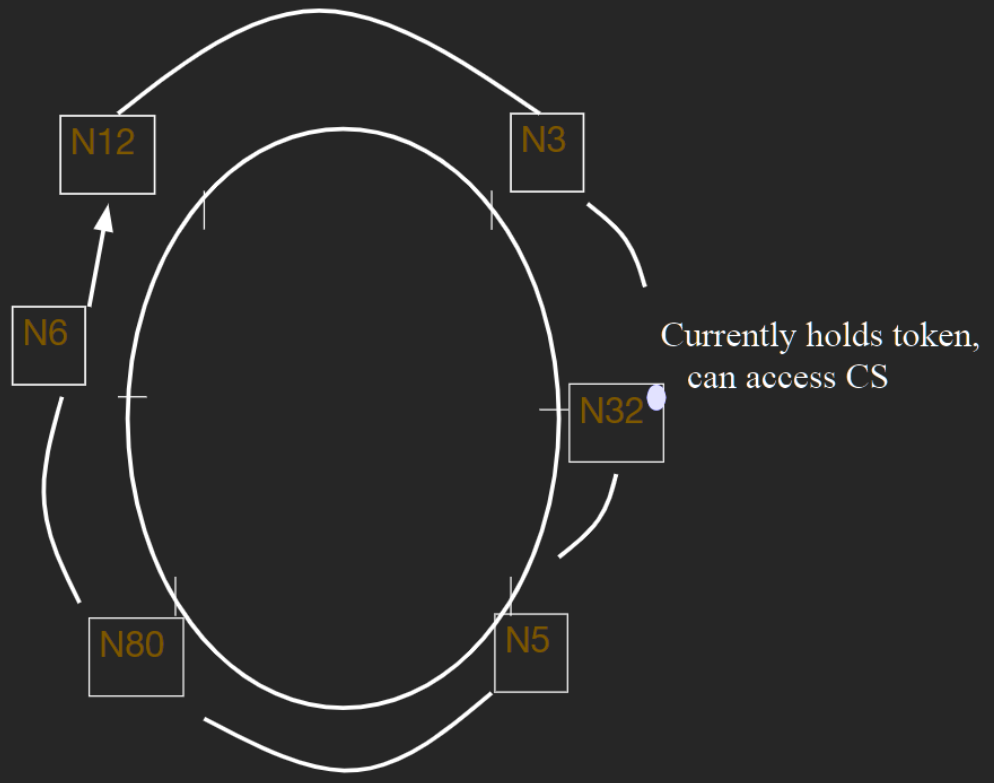
Token:

Token Ring-based Mutual Exclusion



Token:

Token Ring-based Mutual Exclusion



Token Ring Algorithm

Token 算法

- The token is initially given to one process.
  最初分配給一個進程
- The token is passed from one process to its neighbor around the ring.
  Token 從一個進程傳遞給它的
- When a process requires entry to the CS, it waits to receive a token from its neighbor (right) and retains it; enters the CS; when done, it passes the token to its left neighbor (clockwise).
  當一個進程需要進入 CS 時，它會等待從其鄰居（右）接收 Token 並保留它；進入 CS；完成後，它將 Token 傳遞給它的左鄰居（順時針）。
- When a process receives the token, but does not require entry to CS, it passes it along the ring.
  當一個進程收到 Token，但沒有需要進入 CS，它沿著環傳遞它。

Token Ring Algorithm

Token環算法

- It can take from 1 to n-1 messages to obtain a token. Messages are sent around the ring even when no process requires the token (additional load on network).
  獲取Token需要1到n-1條消息。消息即使沒有進程需要Token，也會在環上發送（額外的網絡負載）
- Works well in heavily loaded situations when there is a high probability that the process which gets the token wants to enter the CS. Works poorly in lightly loaded cases.
  在高負載情況下運行良好，獲取token的進程想要進入的概率CS。在負載較輕的情況下效果不佳。
- If a process fails → no progress can be made until a reconfiguration is applied to extract the process from the ring.
  如果一個過程失敗→無法取得進展，直到應用重新配置以從環中提取過程。
- If the process holding the token fails → a unique process has to be picked to regenerate token and pass it along the ring (election algorithm).
  如果持有Token的進程失敗→必須有一個唯一的進程選擇重新生成Token並將其沿環傳遞（選舉算法）。

Ricart-Agrawala Second Algorithm

Ricart-Agrawara第二算法

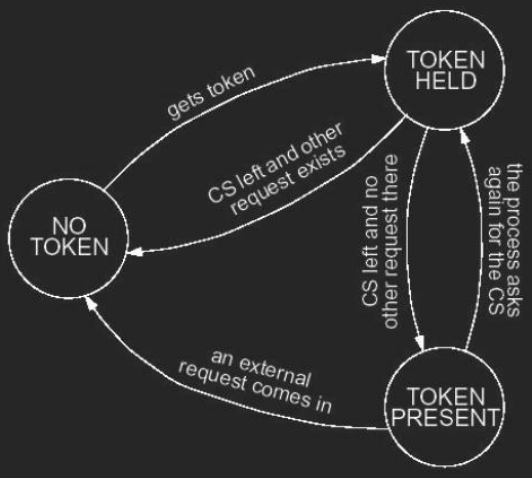
- A process is allowed to enter the critical section when it gets the token.
  - Initially the token is assigned arbitrarily to one of the processes.
    進程在獲得Token時被允許進入臨界區。最初，Token被任意分配給進程之一。
- In order to get the token it sends a request to all other processes competing for the same resource.
  - The request message consists of the requesting process' timestamp (logical clock) and its identifier.
    為了獲得Token，它向競爭同一資源的所有其他進程發送請求。請求消息由請求進程的時間戳（邏輯時鐘）及其標識符組成。
- When a process Pi leaves a critical section 當進程Pi離開臨界區時
  - it passes the token to one of the processes which are waiting for it; this will be the first process Pj, where j is searched in order [i+1, i+2, ..., n, 1, 2, ..., i-2, i-1] for which there is a pending request.
  - If no process is waiting, Pi retains the token (and is allowed to enter the CS if it needs); it will pass over the token as result of an incoming request.

  它將Token傳遞給正在等待它的進程之一；這將是第一個進程 Pj，其中 j 的搜索順序為 [i+1, i+2, ..., n, 1, 2, ..., i-2, i-1]，其中有一個待處理請求。如果沒有進程在等待，Pi 保留Token(如果需要，允許進入CS)；它將作為傳入請求的結果傳遞Token。
- How does Pi find out if there is a pending request?
  - Each process Pi records the timestamp corresponding to the last request it got from process Pj, in requestPi[j]. In the token itself, token[j] records the timestamp (logical clock) of Pj's last holding of the token. If requestPi[j] > token[j] then Pj has a pending request.

  Pi如何知道是否有待處理的請求？每個進程Pi在requestPi[j]中記錄它從進程Pj得到的最後一個請求對應的時間戳。在Token本身中，Token[j]記錄了Pj最後一次持有Token的時間戳（邏輯時鐘）。如果requestPi[j] > token[j]那麼Pj有一個待處理的請求。

Ricart-Agrawala Second Algorithm (cont'd)

- Each process keeps its state with respect to the token: NO-TOKEN, TOKEN-PRESENT, TOKEN-HOLD.



Ricart-Agrawala Second Algorithm (cont'd)

The Algorithm

- Rule for process initialization

/* performed at initialization */

[RI1]: $state_{Pi}$:= NO-TOKEN for all processes P_i, except *one single process* P_x for which $state_{Px}$:= TOKEN-PRESENT.

[RI2]: $token[k]$ initialized 0 for all elements $k = 1 .. n$. $request_{Pi}[k]$ initialized 0 for all processes P_i and all elements $k = 1 .. n$.

- Rule for access request and execution of the CS

/* performed whenever process P_i requests an access to the CS and when it finally gets it; in particular P_i can already possess the token */

[RA1]: **if** $state_{Pi}$ = NO-TOKEN **then**

P_i sends a request message to all processes; the message is of the form (T_{Pi}, i), where $T_{Pi} = C_{Pi}$ is the value of the local logical clock, and i is an identifier of P_i.

P_i waits until it receives the token.

end if.

$state_{Pi}$:= TOKEN-HELD.

P_i enters the CS.

Ricart-Agrawala Second Algorithm (cont'd)

☞ Rule for handling incoming requests

/* performed by P_i whenever it received a request $\langle T_{Pj}, j \rangle$ from P_j */

[RH1]: $request[j] := \max(request[j], T_{Pj})$.

[RH2]: **if** $state_{Pi}$ = TOKEN-PRESENT **then**

P_i releases the resource (see rule RR2).

end if.

☞ Rule for releasing a CS

/* performed by P_i after it finished work in a CS or when it holds a token without using it and it got a request */

[RR1]: $state_{Pi}$ = TOKEN-PRESENT.

[RR2]: **for** k = [i+1, i+2, ..., n, 1, 2, ..., i-2, i-1] **do**

if request[k] > token[k] **then**

$state_{Pi}$:= NO-TOKEN.

$token[i] := C_{Pi}$, the value of the local logical clock.

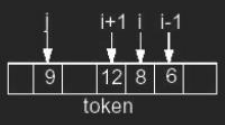
P_i sends the token to P_k.

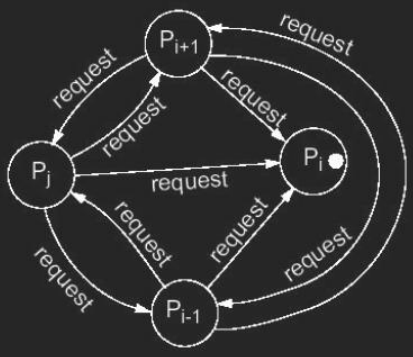
break. /* leave the for loop */

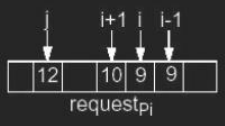
end if.

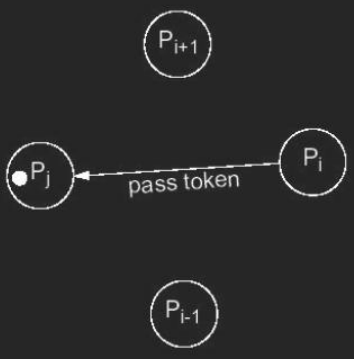
end for.

Ricart-Agrawala Second Algorithm (cont'd)









Ricart-Agrawala Second Algorithm (cont'd)

- The complexity is reduced compared to the (first) Ricart-Agrawala algorithm: it requires *n* messages for entering a CS: (*n*-1) requests and one reply.
- The failure of a process, except the one which holds the token, doesn't prevent progress.

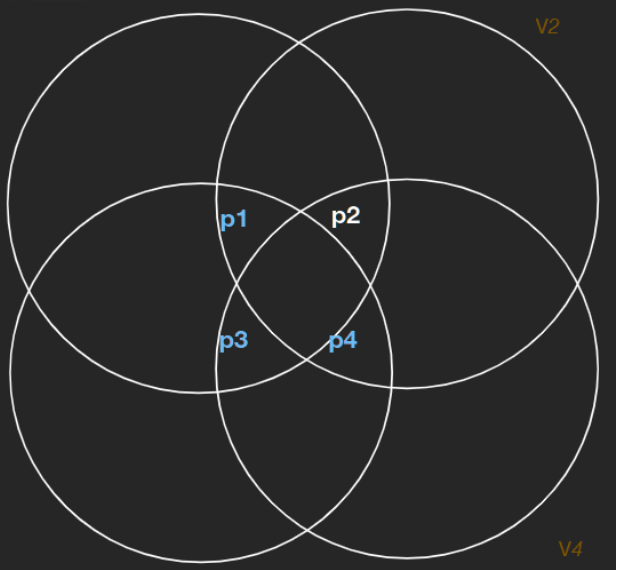
Maekawa's Algorithm: Voting Sets

Maekawa 的算法：投票集

- Ricart-Agrawala1 requires replies from *all* processes in group
- Instead, get replies from only *some* processes in group; but ensure that only one process is given access to CS at a time

- Ricart-Agrawala1 需要組內所有進程的回复
- 相反，僅從組中的某些進程中獲取回复；但確保只有一次允許一個進程訪問 CS

P1's voting set = V1



- Each process requests permission from only its voting set members(not from all)
- Each process (in a voting set) gives permission to at most one process at a time
  - Not to all

- 每個進程請求來自只有它的投票集成員（不是來自全部）
- 每個進程（在一個投票集中）給出一次最多允許一個進程時間
- 並非所有人

Maekawa's Voting Sets

Maekawa 的投票集

- Each process P*i* is associated with a *voting set* V*i* (of processes)
  每個進程 Pi 都與一個投票集 Vi（進程的）相關聯
- Each process belongs to its own voting set
  每個進程都屬於自己的投票集
- *The intersection of any two voting sets must be non-empty*
  任意兩個投票集的交集必須非空
  - *Same concept as Quorums!*
    與法定人數相同的概念！
- Each voting set is of size *K*
  每個投票集的大小為 K
- Each process belongs to *M* other voting sets
  每個進程屬於M個其他投票集
- Maekawa showed that $K=M=\sqrt{N}$ works best
  Maekawa 證明 K=M=√N 效果最好
- One way of doing this is to put N processes in a $\sqrt{N}$ by $\sqrt{N}$ matrix and for each P*i*, its voting set V*i* = row containing P*i* + column containing P*i*. Size of voting set = $2*\sqrt{N}-1$

  一種方法是將 N 個進程放入 √N × √N 矩陣中，並且對於每個 Pi，它的投票集 Vi = 包含 Pi 的行 + 包含 Pi 的列。投票集的大小 = 2*√N-1

Maekawa's Algorithm - Actions

Maekawa 的算法 - 動作

- state = Released, voted = false
- enter() at process P*i*:
  - state = Wanted
  - Multicast Request message to all processes in V*i*
  - Wait for Reply (vote) messages from all processes in V*i* (including vote from self)
  - state = Held
- exit() at process P*i*:
  - state = Released
  - Multicast Release to all processes in V*i*

- When P*i* receives a Request from P*j*:

if (state == Held OR voted = true)
    queue Request
else
    send Reply to P*j* and set voted = true

- When P*i* receives a Release from P*j*:

if (queue empty)
    voted = false
else
    dequeue head of queue, say P*k*
    Send Reply *only* to P*k*
    voted = true

Safety
安全

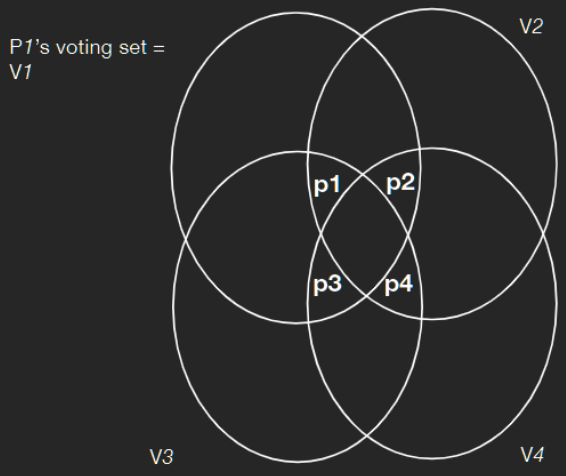
當進程Pi收到來自所有進程的回复時其投票集Vi成員，沒有其他成員
進程Pj可以收到回复來自其所有投票集成員Vj

- When a process P*i* receives replies from all its voting set V*i* members, no other process P*j* could have received replies from all its voting set members V*j*
  - V*i* and V*j* intersect in at least one process say P*k*
  - But P*k* sends only one Reply (vote) at a time, so it could not have voted for both P*i* and P*j*

- Vi和Vj在至少一個過程中相交說PK
- 但Pk一次只發送一個回复（投票），所以它不可能同時投票給Pi和Pj

Liveness
活力

- A process needs to wait for at most (*N-1*) other processes to finish CS
- But does not guarantee liveness
- Since can have a *deadlock*
- Example: all 4 processes need access
  - P1 is waiting for P3
  - P3 is waiting for P4
  - P4 is waiting for P2
  - P2 is waiting for P1
  - No progress in the system!
- There are deadlock-free versions



- 一個進程最多需要等待(N-1)個其他進程完成CS
- 但不保證活性
- 因為可以有死鎖
- 示例：所有4個進程都需要訪問
  - P1 is waiting for P3
  - P3 is waiting for P4
  - P4 is waiting for P2
  - P2 is waiting for P1
  - No progress in the system!
- 有無死鎖版本

Election Algorithms

選舉算法

- Many distributed algorithms require one process to act as a coordinator or, in general, perform some special role.
  許多分佈式算法需要一個進程作為協調器，或者通常執行一些特殊的角色。
- Examples with mutual exclusion 互斥的例子
  - Central coordinator algorithm 中央協調器算法
    - At initialization or whenever the coordinator crashes, a new coordinator has to be elected. 在初始化或協調器崩潰時，必須選舉新的協調器。
  - Token ring algorithm 令牌環算法
    - When the process holding the token fails, a new process has to be elected which generates the new token.
      當持有令牌的進程失敗時，必須選擇一個新進程來生成新令牌。

Election Algorithms

選舉算法

- It doesn't matter which process is elected. 選擇哪個進程並不重要。
  - What is important is that one and only one process is chosen (we call this process the coordinator) 重要的是只選擇一個進程（我們稱這個進程為協調器）
  - All processes agree on this decision. 所有流程都同意此決定
  - Let all processes know about this leader
    讓所有流程都知道這個領導者

What happens if the leader fails? 如果領導失敗了怎麼辦？

- Election is typically started after the failure occurs. 選舉通常在失敗發生後開始。
  - The detection of a failure (e.g. the crash of the current coordinator) is normally based on time-out. 故障的檢測（例如當前協調器的崩潰）通常基於超時。
    - a process that gets no response for a period of time suspects a failure and initiates an election process
      在一段時間內沒有得到響應的進程懷疑失敗並啟動選舉進程。
- An election process is typically performed in two phases:
  選舉過程通常分兩個階段執行：
  - Select a leader with the highest priority. 選擇具有最高優先級的領導者。
  - Inform all processes about the winner.
    通知有關獲勝者的所有流程。
- Assume that each process has a unique number (identifier).
  - In general, election algorithms attempt to locate the process with the highest number, among those which currently are up. 假設每個進程都有一個唯一的編號（標識符）。
    通常，選舉算法會嘗試在當前運行的進程中找到編號最高的進程。

Assumptions - System Model

假設 - 系統模型

- *N* processes.
  N 個進程。
- Each process has a unique id.
  每個進程都有一個唯一id。
- Messages are eventually delivered.
  消息是最終交付。
- Failures may occur during the election protocol.
  選舉協議期間可能會發生故障。
- Any process can call for an election.
  任何進程都可以調用選舉。
- A process can call for at most one election at a time.
  一個進程最多可以調用一次選舉。
- Multiple processes are allowed to call an election simultaneously.
  允許多個進程舉行同時選舉。
  - All of them together must yield only a single leader
    所有這些加在一起只能產生一個單一領導
- The result of an election should not depend on which process calls for it.
  選舉結果應不取決於哪個進程去呼叫它。

The Leader Election Problem

領袖選舉問題

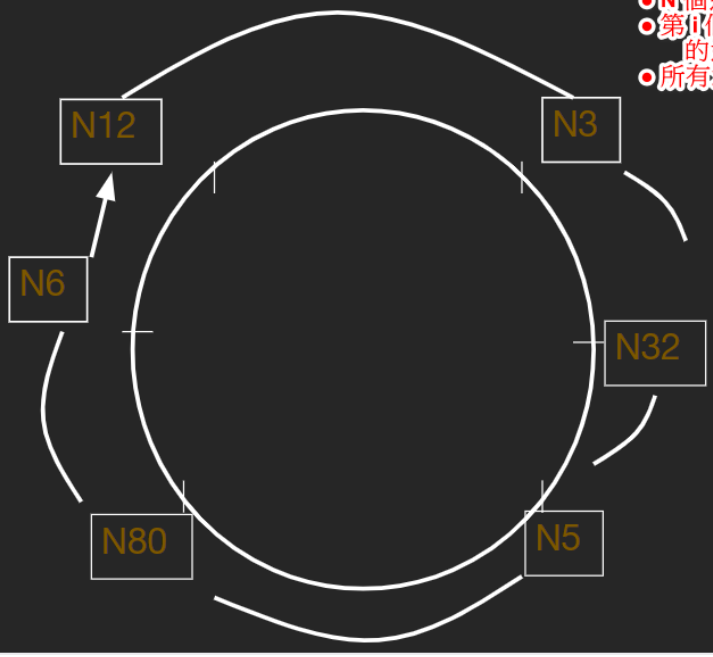
選舉算法的運行必須始終保證在結束是：

- A run of the election algorithm must always guarantee at the end:
  - **Safety**: For all non-faulty processes *p*: (*p*'s elected = (q: a particular non-faulty process with the best attribute value) or Null)
    安全性：對於所有非故障進程 p:(p 的當選 =(q:一個特定的具有最佳屬性值的非故障進程) 或 Null)
  - **Liveness**: For all election runs: (election run terminates) & for all non-faulty processes *p*: *p*'s elected is not Null
    活躍度：對於所有選舉運行：(選舉運行終止) & 對於所有非故障進程 p:p 的選舉不是 Null
- At the end of the election protocol, the non-faulty process with the best (highest) election attribute value is elected.
  在選舉協議結束時，非故障過程具有最佳（最高）選舉屬性值的選舉。
  - Common attribute : leader has highest id
    共同屬性：leader id 最高
  - Other attribute examples: leader has highest IP address, or fastest cpu, or most disk space, or most number of files, etc.
    其他屬性示例：leader 有最高的 IP 地址，或最快的 cpu，或最多的

The Ring Algorithm

- *N* processes are organized in a logical ring
  - *i*-th process p_i has a communication channel to $p_{(i+1) \bmod N}$
  - All messages are sent clockwise around the ring.

環算法
- N個進程被組織在一個邏輯環中
- 第i個進程pi有一個到p(i+1) mod N的通信通道
- 所有消息都圍繞環順時針發送。



The Ring Election Protocol

環狀選舉協議

- Any process p_i that discovers the old coordinator has failed initiates an "Election" message that contains p_i's own id:attr. This is the *initiator* of the election.

任何發現舊協調器失敗的進程pi都會啟動包含pi自己的id:attr的"選舉"消息。這是選舉發起人

- When a process p_i receives an "Election" message, it compares the attr in the message with its own attr.

當進程pi收到"選舉"消息時，它比較attr在消息中使用自己的attr。

  - If the arrived attr is greater, p_i forwards the message.

如果到達的attr更大，pi轉發消息

  - If the arrived attr is smaller and p_i has not forwarded an election message earlier, it overwrites the message with its own id:attr, and forwards it.

早些時候，它用自己的id:attr覆蓋消息，並轉發它。

如果到達的attr較小並且pi沒有轉發選舉消息

  - If the arrived id:attr matches that of p_i, then p_i's attr must be the greatest (why?), and it becomes the new coordinator. This process then sends an "Elected" message to its neighbor with its id, announcing the election result.

如果到達的id:attr與pi匹配，則pi的attr必須是最大的

（為什麼？），它成為新的協調者。這個過程然後發送一個"選舉"消息給它的鄰居，宣布選舉結果

- When a process p_i receives an "Elected" message, it
  - sets its variable $elected_i$ ☐ id of the message.

當進程pi收到"Elected"消息時，它
- 設置它的變量selectedi □消息的id。
- 轉發消息，除非它是新的協調者。

  - forwards the message unless it is the new coordinator.

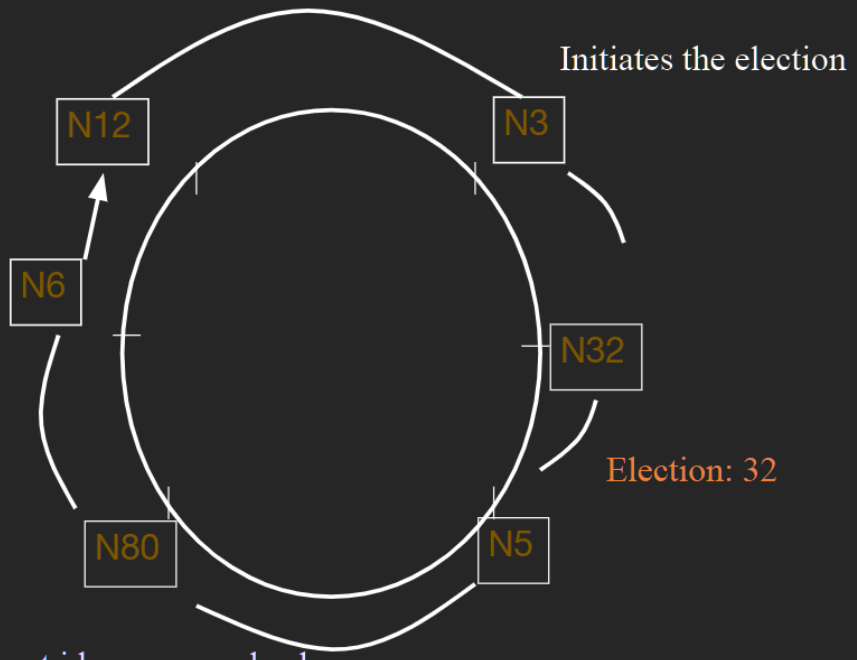
The Ring-based Algorithm

基於環的算法

我們假設進程被安排在一個邏輯環中

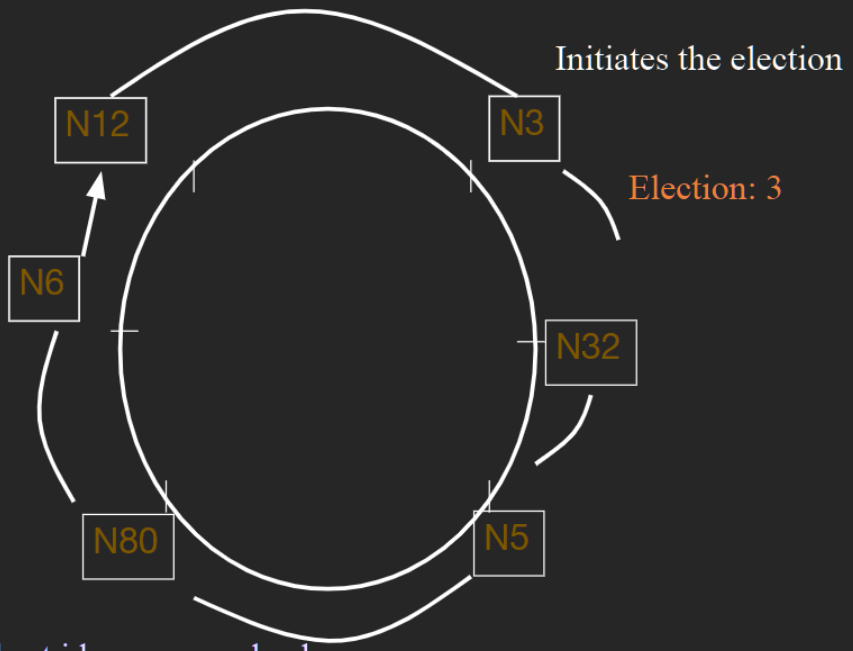
- We assume that the processes are arranged in a logical ring
  - Each process knows the address of one other process, which is its neighbor in the clockwise direction. 每個進程都知道另一個進程的地址，它是順時針方向的鄰居。
- The algorithm elects a single coordinator, which is the process with the highest identifier. 該算法選出一個協調器，它是具有最高標識符的進程。
- Election is started by a process which has noticed that the current coordinator has failed. 選舉是由注意到當前協調器失敗的進程開始的。
  - The process places its identifier in an election message that is passed to the following process. 該進程將其標識符放在傳遞給以下進程的選舉消息中。
  - When a process receives an election message 當進程收到選舉消息時
    - It compares the identifier in the message with its own. 它將消息中的標識符與其自己的標識符進行比較
    - If the arrived identifier is greater, it forwards the received election message to its neighbor 如果到達的標識符更大，則將收到的選舉消息轉發給它的鄰居
    - If the arrived identifier is smaller it substitutes its own identifier in the election message before forwarding it. 如果到達的標識符較小，則在轉發之前在選舉消息中替換它自己的標識符。
    - If the received identifier is that of the receiver itself, this will be the coordinator. 如果接收到的標識符是接收器本身的標識符，則這將是協調器。
- The new coordinator sends an elected message through the ring. 新的協調器通過環發送一條選舉消息。

Leader Election: Ring Algorithm Example



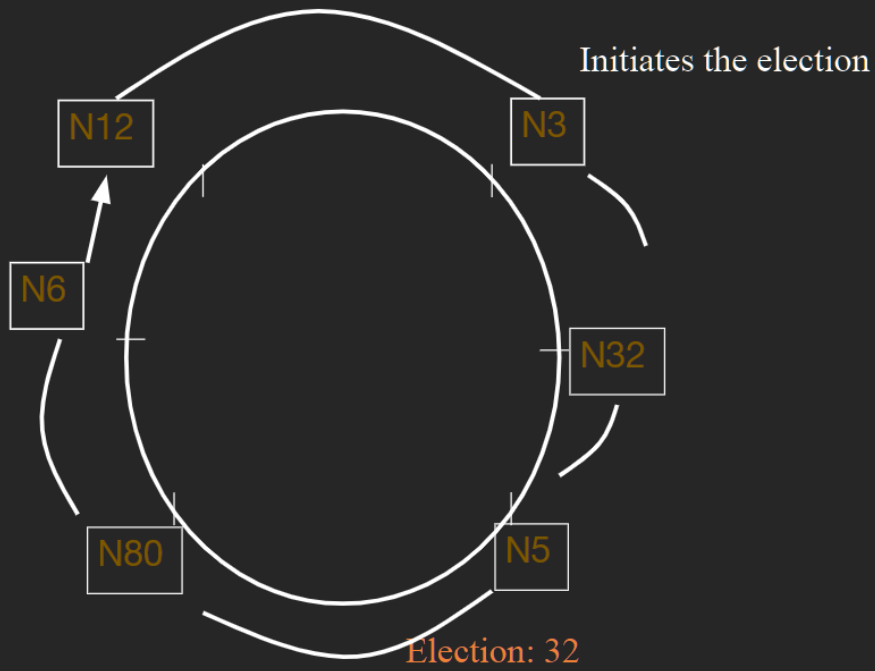
Goal: Elect highest id process as leader

Leader Election: Ring Algorithm Example



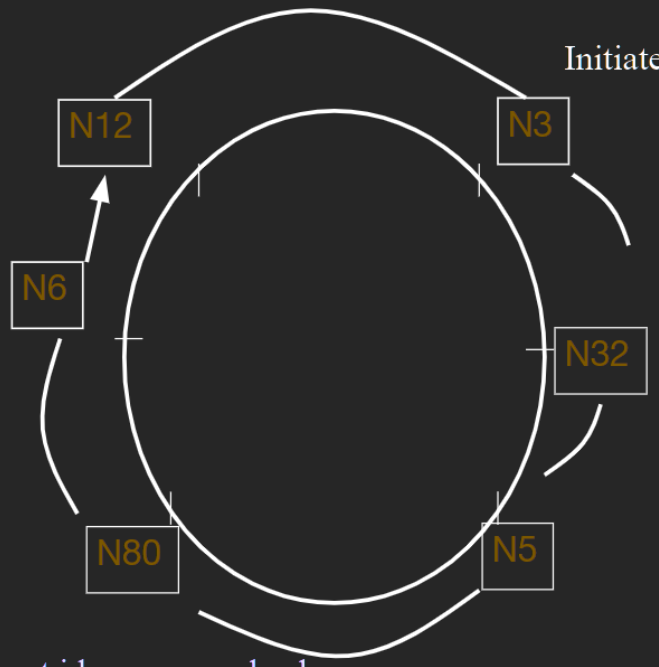
Goal: Elect highest id process as leader

Leader Election: Ring Algorithm Example



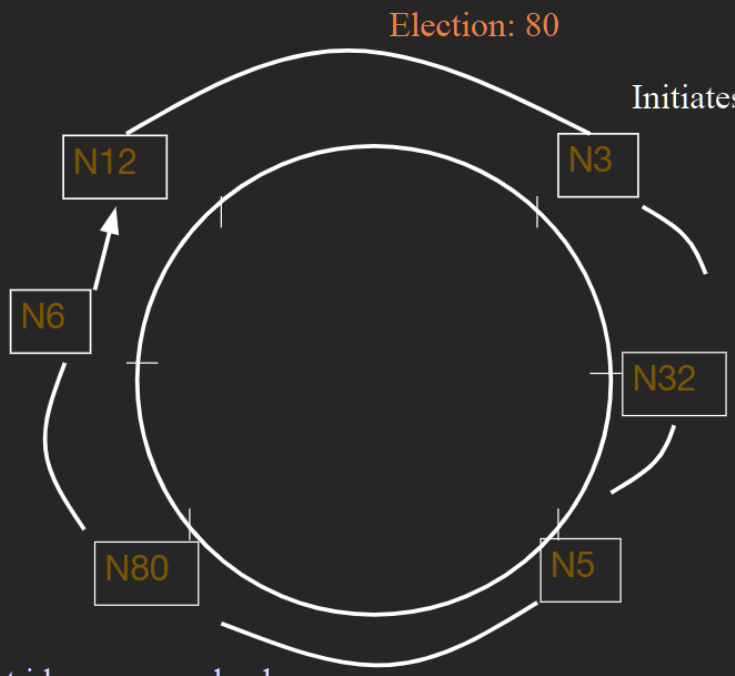
Goal: Elect highest id process as leader

Leader Election: Ring Algorithm Example



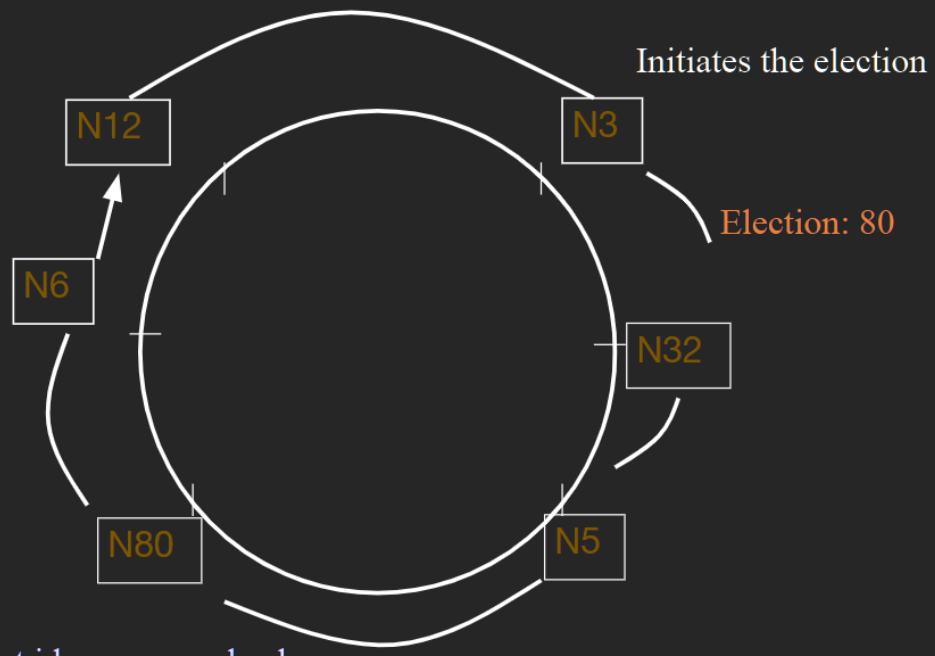
Goal: Elect highest id process as leader

Leader Election: Ring Algorithm Example



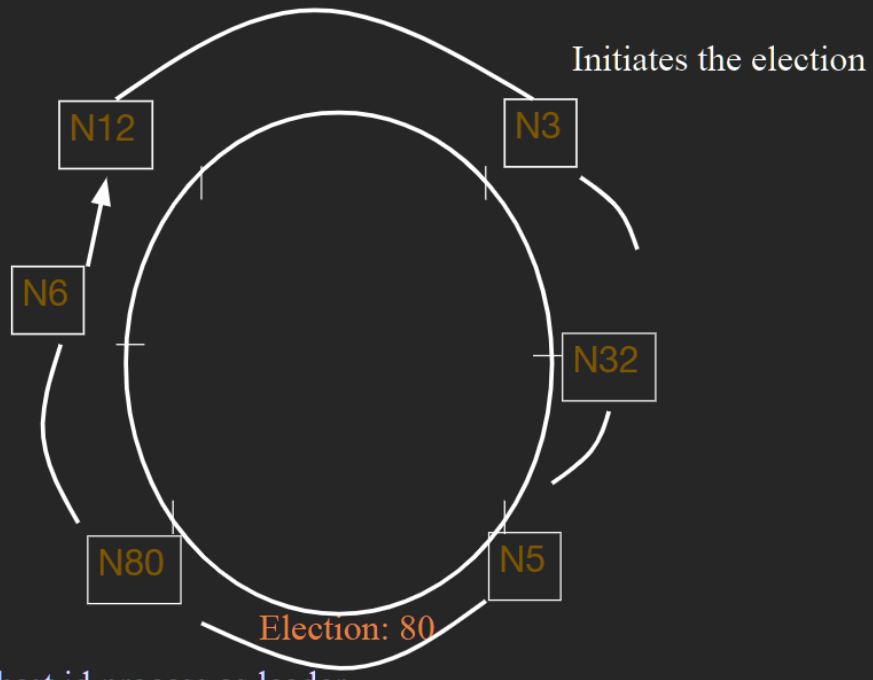
Goal: Elect highest id process as leader

Leader Election: Ring Algorithm Example



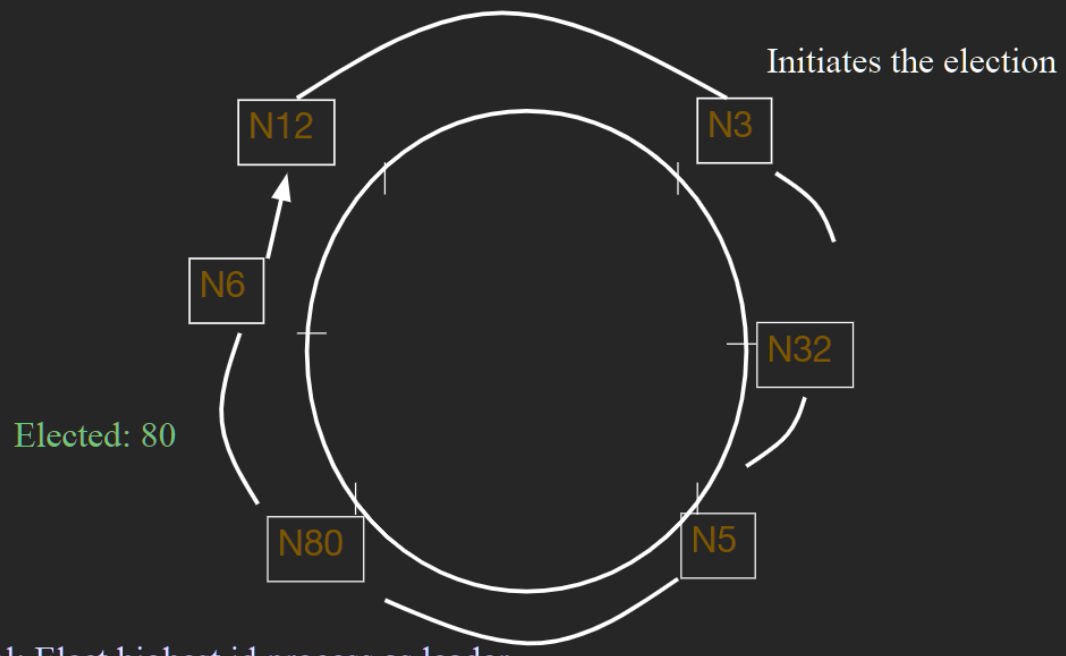
Goal: Elect highest id process as leader

Leader Election: Ring Algorithm Example



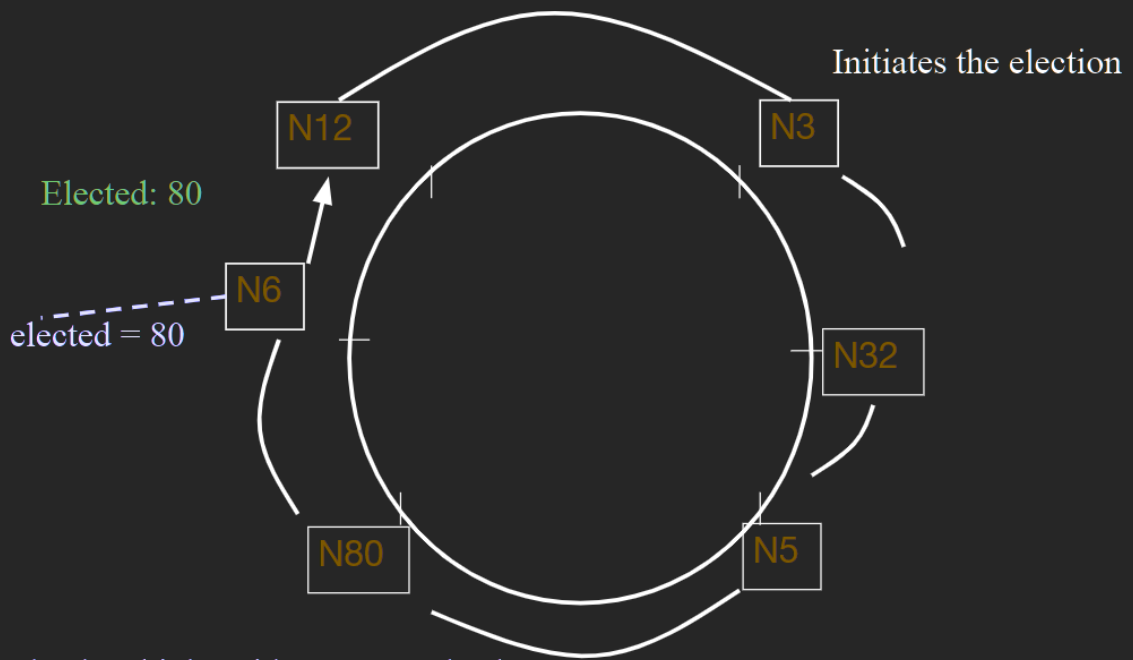
Goal: Elect highest id process as leader

Leader Election: Ring Algorithm Example



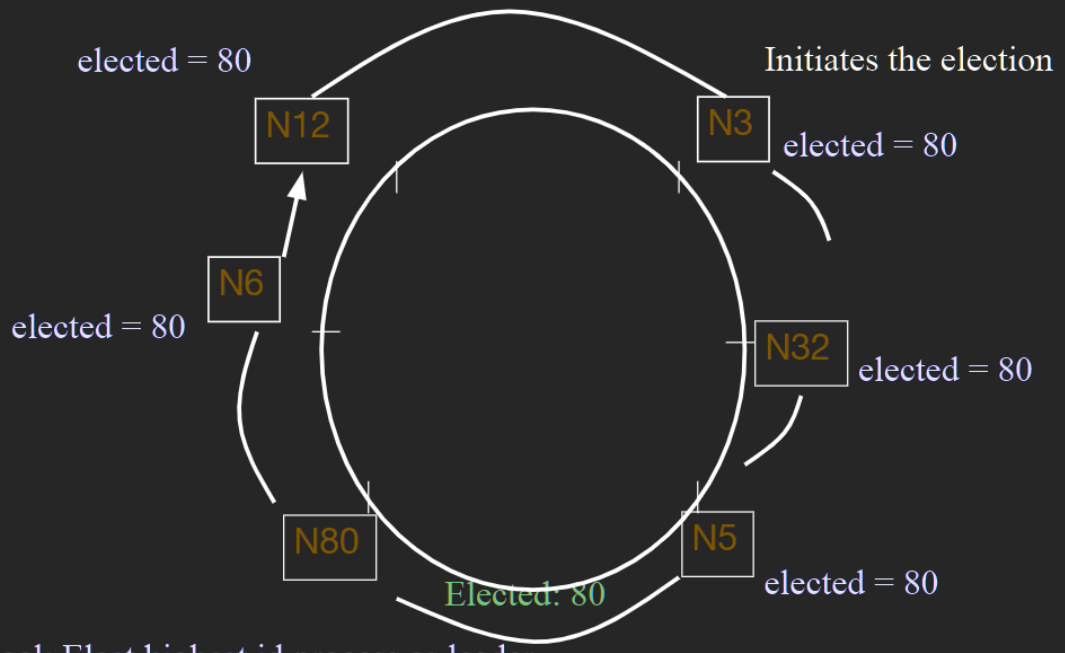
Goal: Elect highest id process as leader

Leader Election: Ring Algorithm Example



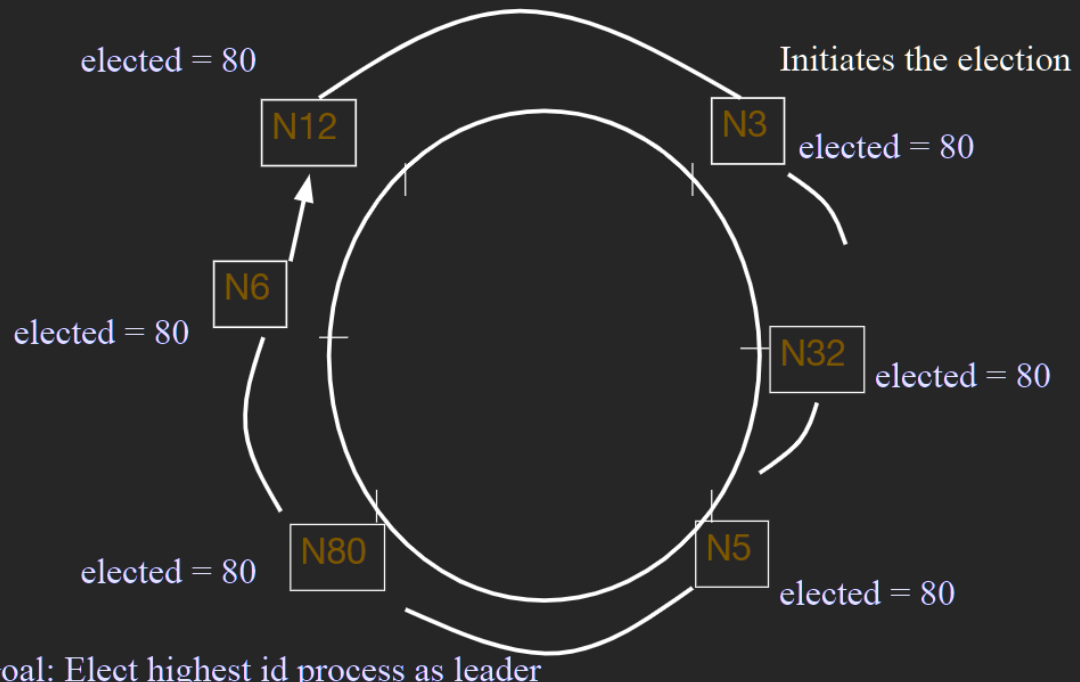
Goal: Elect highest id process as leader

Leader Election: Ring Algorithm Example



Goal: Elect highest id process as leader

Leader Election: Ring Algorithm Example



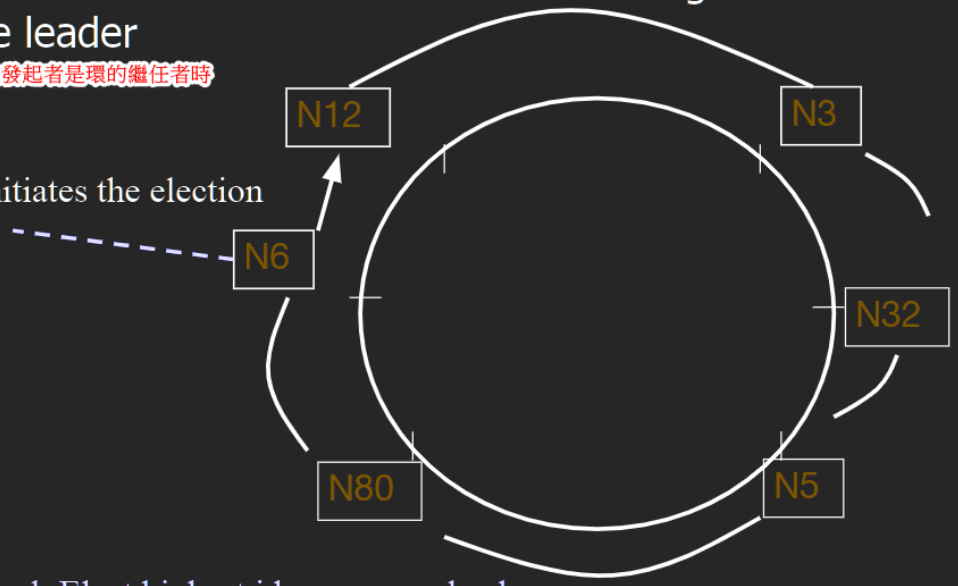
Goal: Elect highest id process as leader

Worst-case Analysis

最壞情況分析

- Let's assume no failures occur during the election protocol itself, and there are *N* processes 讓我們假設在選舉協議本身期間沒有發生故障。
- How many messages? 多少條消息?
- Worst case occurs when the initiator is the ring successor of the would-be leader

最壞的情況發生在發起者是環的繼任者時潛在的領導者



Goal: Elect highest id process as leader

Worst/Best/Avg Case Analysis

Worst-case Analysis 最壞情況分析

- (*N-1*) messages for Election message to get from Initiator to would-be coordinator (N-1)條用於選舉消息的消息從發起者到潛在的
- *N* messages for Election message to circulate around ring without message being changed N條消息用於選舉消息在沒有消息的情況下在環中循環
- *N* messages for Elected message to circulate around the ring N條消息為Elected消息在環上循環
- Message complexity: (*3N-1*) messages 消息複雜度：(3N-1)條消息
- Completion time: (*3N-1*) message transmission times 完成時間：(3N-1)條消息傳輸次數
- Thus, if there are no failures, election terminates (liveness) and everyone knows about highest-attribute process as leader (safety)

因此，如果沒有失敗，選舉終止（活性），每個人都知道關於作為領導者的最高屬性過程（安全）

Best Case Analysis

- Initiator is the would-be leader.
- Message complexity: *2N* messages
- Completion time: *2N* message transmission times

最佳情況分析
- 發起者是潛在的領導者。
- 消息複雜度：2N條消息
- 完成時間：2N消息傳輸次數

Average Case Analysis

平均情況分析
- (2N + N/2)條消息；(2N + N/2)完成時間

- *(2N + N/2)* messages; *(2N + N/2)* completion time

Leader Election:The Ring Algorithm Multiple Initiators 領導者選舉:環算法多個發起者

- Several elections can be active at the same time. 可以同時進行多個選舉。
  - Messages generated by later elections should be killed as soon as possible. 以後選舉產生的消息應該盡快被殺死。
- Processes can be in one of two states 進程可以處於兩種狀態之一
  - Participant or Non-participant. 參與者或非參與者。
    - Initially, a process is non-participant. 最初，進程是非參與者的。
- The process initiating an election marks itself participant. 發起選舉的過程將自身標記為參與者
- Rules 規則
  - Include initiators id with all messages 在所有消息中包含發起者ID
  - For a participant process, if the identifier in the election message is smaller than the own, does not forward any message (it has already forwarded it, or a larger one, as part of another simultaneously ongoing election). 對於一個參與者進程，如果選舉消息中的標識符小於自己的標識符，則不轉發任何消息（它已經轉發了它，或者更大的一個，作為另一個同時進行的選舉的一部分）。
  - When forwarding an election message, a process marks itself participant. 在轉發選舉消息時，進程將自己標記為參與者。
  - When sending (forwarding) an elected message, a process marks itself non-participant. 當發送（轉發）一個選擇的消息時，一個進程將自己標記為非參與者。

領導者選舉:環算法多個發起者

Leader Election:The Ring Algorithm Multiple Initiators

☞ By default, the state of a process is NON-PARTICIPANT

☞ Rule for election process initiator

/* performed by a process P_i, which triggers the election procedure */

[RE1]: $state_{Pi}$:= PARTICIPANT.

[RE2]: P_i sends an *election message* with *message.id* := *i* to its neighbour.

☞ Rule for handling an incoming *election message*

/* performed by a process P_j, which receives an *election message* */

[RH1]: **if** *message.id* > *j* **then**
  P_j forwards the received *election message*.
  $state_{Pj}$:= PARTICIPANT.
elseif *message.id* < *j* **then**
  if $state_{Pj}$ = NON-PARTICIPANT **then**
    P_j forwards an *election message* with *message.id* := *j*.
    $state_{Pj}$:= PARTICIPANT
  end if
else
  P_j is the coordinator and sends an *elected message* with *message.id* := *j* to its neighbour.
  $state_{Pj}$:= NON-PARTICIPANT.
end if.

☞ Rule for handling an incoming *elected message*

/* performed by a process P_i, which receives an *elected message* */

[RD1]: **if** *message.id* ≠ *i* **then**
  P_i forwards the received *elected message*.
  $state_{Pj}$:= NON-PARTICIPANT.
end if.

Failures?

失敗?

- One option: have predecessor (or successor) of would-be leader N80 detect failure and start a new election run
  - May re-initiate election if
    - Receives an Election message but times out waiting for an Elected message
    - Or after receiving the Elected:80 message
  - But what if predecessor also fails?
  - And its predecessor also fails? (and so on)

- 一種選擇：擁有潛在領導者的前任（或繼任者）N80檢測失敗並開始新的選舉運行
- 如果可重新發起選舉
  - 收到選舉消息但等待超時當選的消息
  - 或在收到Elected:80消息後
- 但如果前任也失敗了怎麼辦?
- 它的前身也失敗了？（等等）

Fixing for failures (2)

修復故障(2)

- Second option: use the failure detector
  - 第三種選擇：使用故障檢測器
- Any process, after receiving Election:80 message, can detect failure of N80 via its own local failure detector
  - If so, start a new run of leader election
- But failure detectors may not be both complete and accurate
  - Incompleteness in FD => N80's failure might be missed => Violation of Safety
  - Inaccuracy in FD => N80 mistakenly detected as failed
    - => new election runs initiated forever
    - => Violation of Liveness

- 任何進程，在收到Election:80消息後，都可以檢測N80通過其自身的本地故障檢測器發生故障
- 如果是，則開始新一輪的領導人選舉

- 但故障檢測器可能既不完整又不准確
- FD中的不完整性=>N80的故障可能會被遺漏=>違反安全
- FD不准確=>N80錯誤地檢測為失敗
  - =>新的選舉運行永遠啟動
  - =>違反Liveness

Why is Election so Hard?

選舉為何如此艱難？

- Because it is related to the consensus problem! 因為它關係到共識問題！
- If we could solve election, then we could solve consensus! 如果我們能解決選舉，那麼我們就能解決共識！
  - Elect a process, use its id's last bit as the consensus decision 選舉一個進程，使用其id的最後一位作為共識決定
- But since consensus is impossible in asynchronous systems, so is election!

但既然不可能達成共識異步系統，選舉也是如此！

Leader Election: The Bully Algorithm

領袖選舉：欺凌算法

- A process has to know the identifier of all other processes 一個進程必須知道所有其他進程的標識符
  - (it doesn't know, however, which one is still up); the process with the highest identifier, among those which are up, is selected. (然而，它不知道哪個還在運行)：在那些up的進程中選擇具有最高標識符的進程。
- Any process could fail during the election procedure. 在選舉過程中，任何過程都可能失敗。
- 當進程Pi檢測到故障並且必須選舉協調器時
  When a process Pi detects a failure and a coordinator has to be elected
  - It sends an election message to all the processes with a higher identifier and then waits for an answer message: 它向所有具有更高標識符的進程發送選舉消息，然後等待應答消息：
  - If no response arrives within a time limit 如果在時限內沒有回復
    - Pi becomes the coordinator (all processes with higher identifier are down) Pi成為協調器（所有具有更高標識符的進程都關閉）
    - it broadcasts a coordinator message to all processes to let them know. 它向所有進程廣播協調器消息，讓他們知道。
  - If an answer message arrives, 如果收到回復消息，
    - Pi knows that another process has to become the coordinator □ it waits in order to receive the coordinator message. Pi知道另一個進程必須成為協調器□，它等待接收協調器消息。
    - If this message fails to arrive within a time limit (which means that a potential coordinator crashed after sending the answer message) Pi resends the election message. 如果此消息未能在時間限制內到達（這意味著潛在協調器在發送應答消息後崩潰），Pi將重新發送選舉消息。
- When receiving an election message from Pi 當收到來自Pi的選舉消息時
  - a process Pj replies with an answer message to Pi and
  - then starts an election procedure itself(unless it has already started one) it sends an election message to all processes with higher identifier. 進程Pj用應答消息回復Pi和然後自己啟動一個選舉程序（除非它已經啟動了）它向所有具有更高標識符的進程發送選舉消息。
- Finally all processes get an answer message, except the one which becomes the coordinator. 最後，除了成為協調者的進程之外，所有進程都會收到一條應答消息。

The Bully Algorithm

- By default, the state of a process is ELECTION-OFF

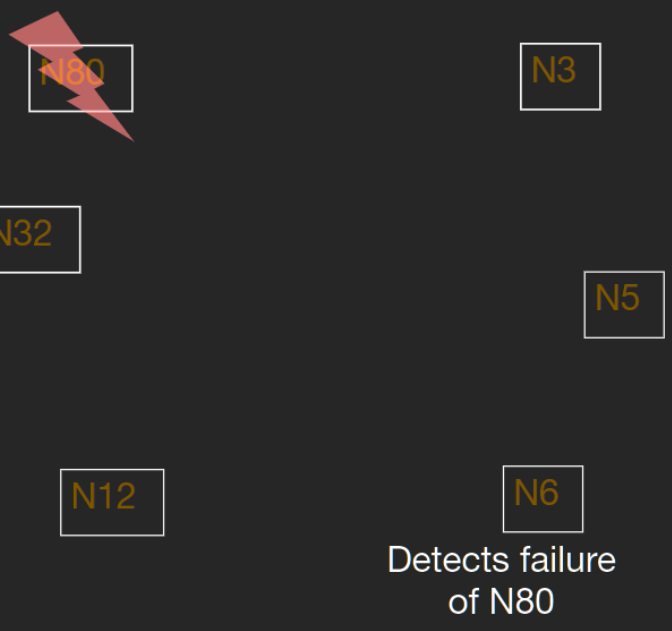
- Rule for election process initiator
  /* performed by a process P_i, which triggers the election procedure, or which starts an election after receiving itself an *election message* */

[RE1]: $state_{Pi}$:= ELECTION-ON.
P_i sends an *election message* to all processes with a higher identifier.
P_i waits for *answer message*.
if no *answer message* arrives before time-out **then**
P_i is the coordinator and sends a *coordinator message* to all processes.
else
P_i waits for a *coordinator message* to arrive.
if no *coordinator message* arrives before time-out **then**
restart election procedure according to RE1
end if
end if.

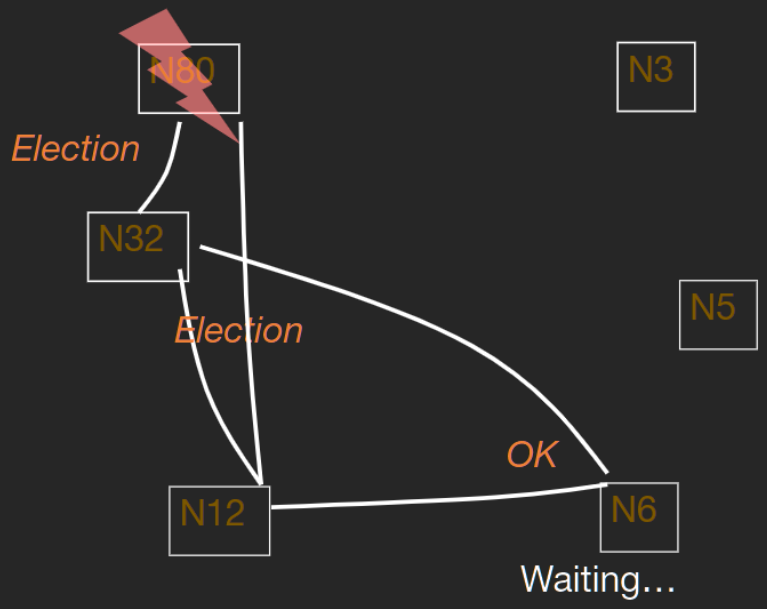
- Rule for handling an incoming *election message*
  /* performed by a process P_i at reception of an *election message* coming from P_j */

[RH1]: P_i replies with an *answer message* to P_j.
[RH2]: **if** $state_{Pi}$:= ELECTION-OFF **then**
start election procedure according to RE1
end if

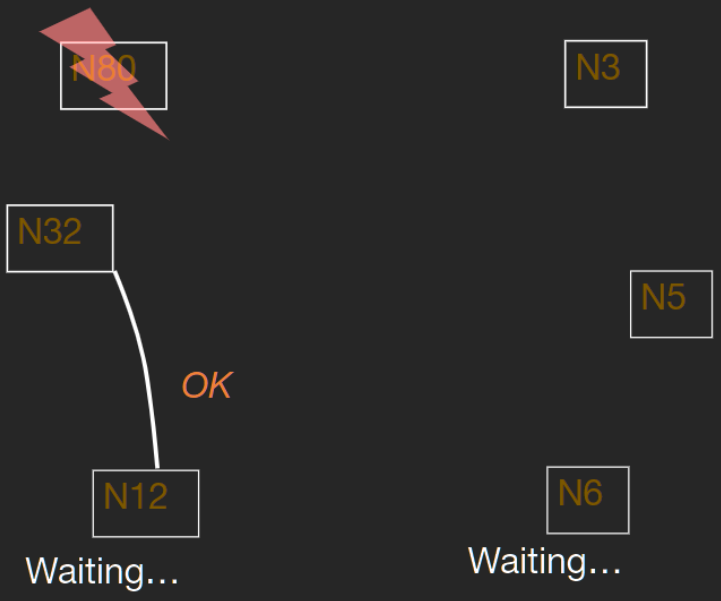
Bully Algorithm: Example



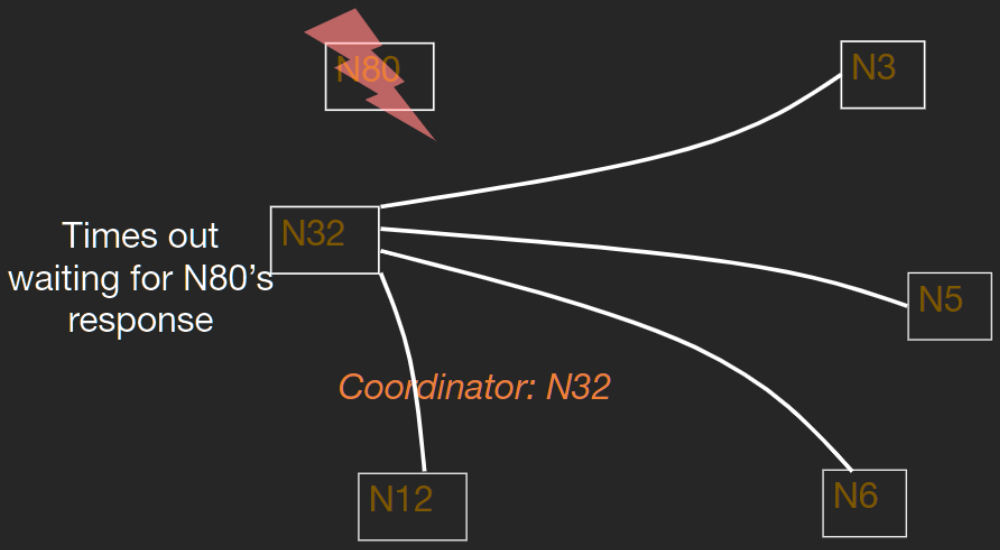
Bully Algorithm: Example



Bully Algorithm: Example



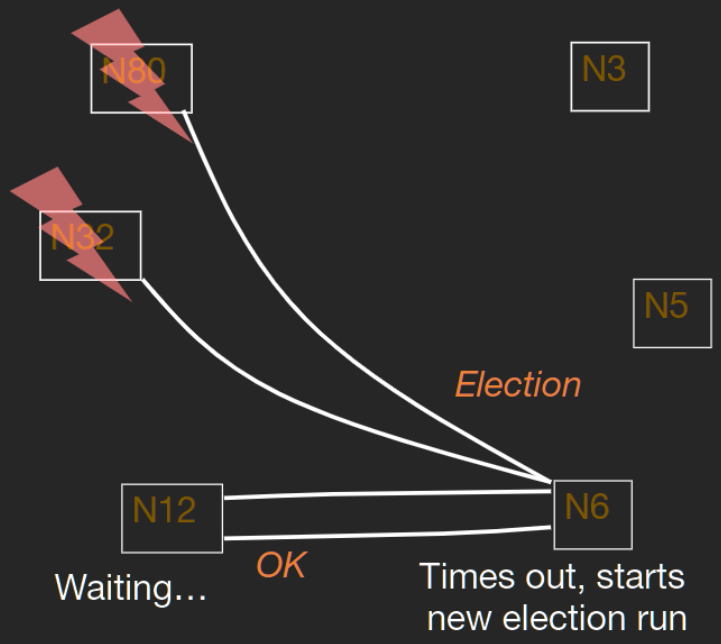
Bully Algorithm: Example



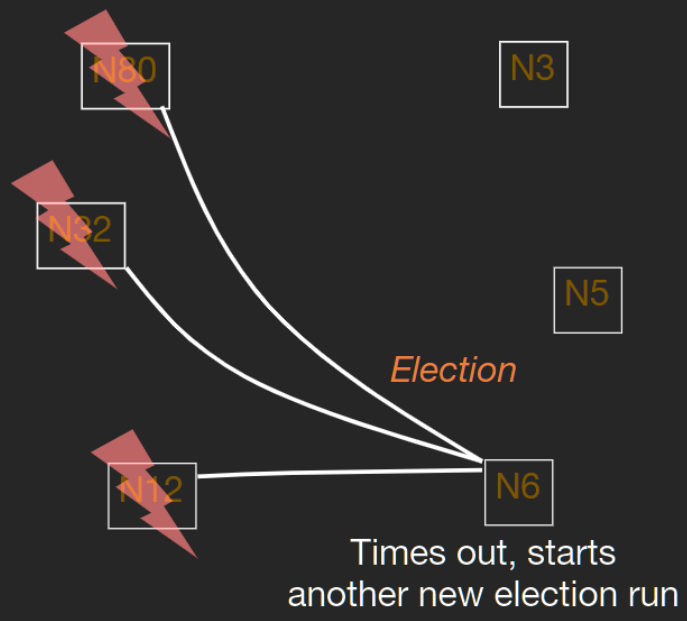
Election is completed

Failures during Election Run

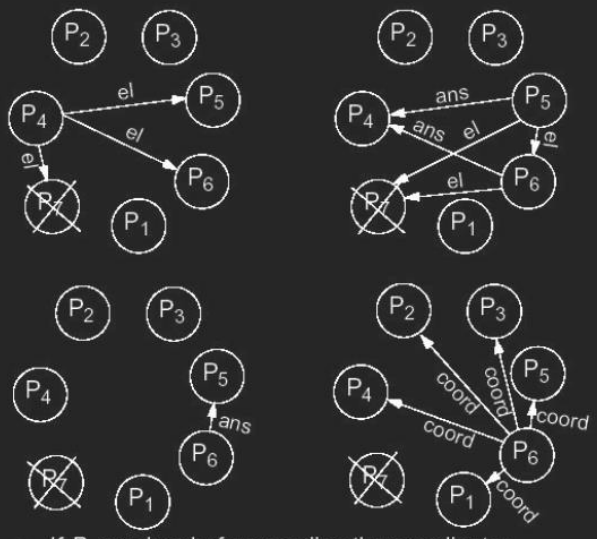
Bully Algorithm: Example



Bully Algorithm: Example



The Bully Algorithm



- If P_6 crashes before sending the coordinator message, P_4 and P_5 restart the election process.

☞ The best case: the process with the second highest identifier notices the coordinator's failure. It can immediately select itself and then send n-2 coordinator messages.

☞ The worst case: the process with the lowest identifier initiates the election; it sends n-1 election messages to processes which themselves initiate each one an election ⇒ $O(n^2)$ messages.

The Bully Algorithm: Analysis

欺凌算法：分析

- **Worst-case** completion time: 最壞情況完成時間
  - When the process with the lowest id in the system detects the failure. 當系統中id最小的進程檢測故障。
    - (*N-1*) processes altogether begin elections, each sending messages to processes with higher ids. 進程完全開始選舉。每個進程都向具有更高 id 的進程發送消息。
    - *i*-th highest id process sends (*i-1*) election messages 第 i 個最高 id 進程發送 (i-1) 個選舉消息
  - Number of Election messages
    $= N\text{-}1 + N\text{-}2 + \ldots + 1 = (N\text{-}1)*N/2 = O(N^2)$
    選舉訊息數
- **Best-case** 最好的情況
  - Second-highest id detects leader failure 第三高的 id 檢測到領導失敗
  - Sends (*N-2*) Coordinator messages 發送 (N-2) 個協調器消息
  - Completion time: 1 message transmission time 完成時間：1條消息傳輸時間

Impossibility?

不可能？

- Since timeouts built into protocol, in asynchronous system model: 由於超時內置於協議中，在異步系統模型：
  - Protocol may never terminate => Liveness not guaranteed 協議可能永遠不會終止=>活性不會保證
- But satisfies liveness in synchronous system model where 但滿足同步系統中的活性模型在哪裡
  - Worst-case one-way latency can be calculated = worst-case processing time + worst-case message latency 可以計算最壞情況的單向延遲=最壞情況處理時間+最壞情況消息延遲

Election in Industry

- Several systems in industry use consensus based approaches for election
  - e.g. Paxos is a consensus protocol (safe, but eventually live):
- Google's Chubby system
- Apache Zookeeper

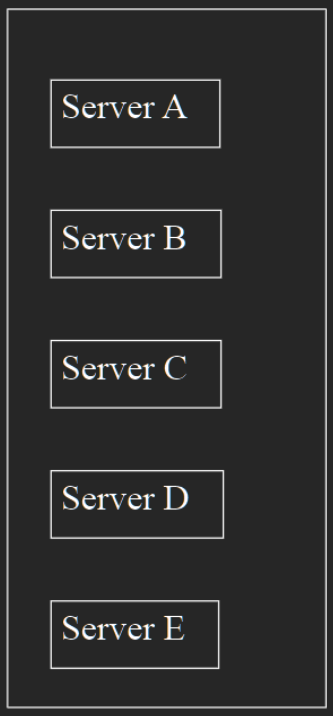
行業選舉
- 行業內多個系統使用共識基於選舉的方法
  - 例如Paxos是一個共識協議（安全，但最終live）：
- Google的Chubby系統
- Apache Zookeeper

Election in Google Chubby

在 Google Chubby 中選舉

- A system for locking
  一種鎖定係統
- Essential part of Google's stack
  谷歌堆的重要組成部分
  - Many of Google's internal systems rely on Chubby
    谷歌的許多內部系統依賴於 Chubby
  - BigTable, Megastore, etc.
    BigTable、Megastore 等
- Group of replicas
  副本組
  - Need to have a master server elected at all times
    任何時間都需要有一個主服務器

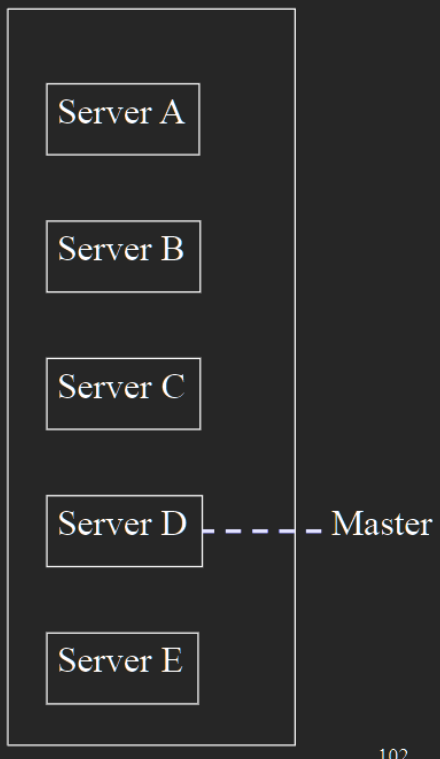
Reference: http://research.google.com/archive/chubby.html



Election in Google Chubby

在 Google Chubby 的選舉

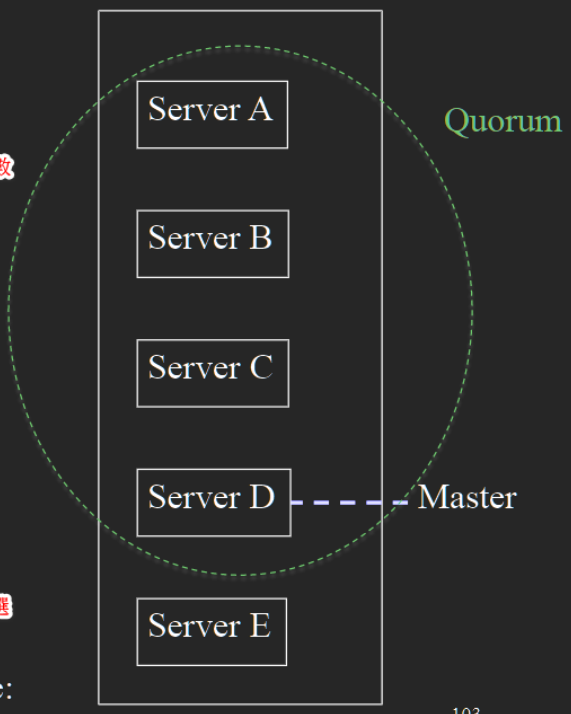
- Group of replicas
  - Need to have a master (i.e., leader)
- Election protocol
  - Potential leader tries to get votes from other servers
  - Each server votes for at most one leader
  - Server with *majority* of votes becomes new leader, informs everyone
    - 副本
    - 需要有一個 master（即，領導者）
    - 選舉協議
      - 潛在領導者試圖獲得選票來自其他服務器
      - 每台服務器最多投一票給一個領導者
      - 擁有多數票的服務器成為新領導，通知每個人



Election in Google Chubby

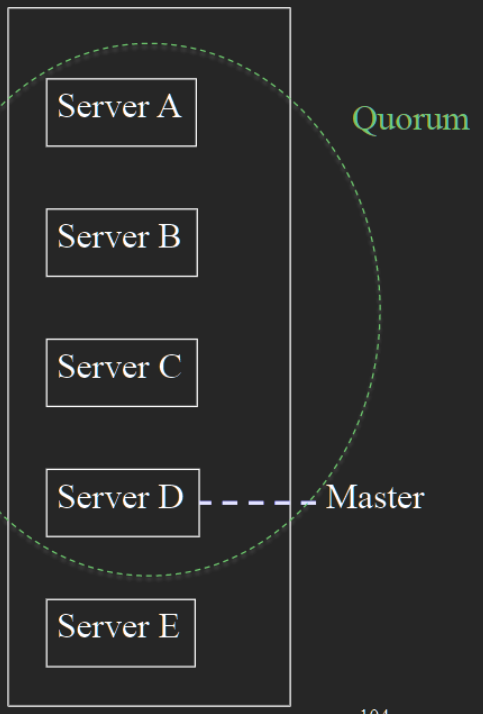
在Google Chubby的選舉

- Why safe? 為什麼安全?
  - Essentially, each potential leader tries to reach a *quorum*
    - 本質上，每個潛在的領導者都會嘗試達到法定人數
  - Since any two quorums intersect, and each server votes at most once, cannot have two leaders elected simultaneously
    - 由於任意兩個群體相交，並且每台服務器最多投票一次，不能同時選出兩名領導人
- Why live? 為什麼是live?
  - Only eventually live! Failures may keep happening so that no leader is ever elected
    - 只有最終live！故障可能繼續發生，所以沒有領導者曾經當選
  - In practice: elections take a few seconds. Worst-case noticed by Google: 30 s
    - 在實踐中：選舉在幾秒，谷歌注意到的最壞情況：30S



Election in Google Chubby

- After election finishes, other servers promise not to run election again for "a while"
  - 選舉結束後，其他服務器承諾不再為"a While"
  - "While" = time duration called "Master lease"
    - "While"=稱為"Master lase"的持續時間(主租約)
  - Set to a few seconds
    - 設置為幾秒
- Master lease can be renewed by the master as long as it continues to win a majority each time
  - 主租約可以更新,經由Master得到,只要那個同一個Master每次都贏
- Lease technique ensures automatic re-election on master failure
  - 租賃技術確保自動化主失敗時重選



Summary (Distributed Mutual Exclusion)

總結（分佈式互斥）

- In a distributed environment no shared variables (semaphores) and local kernels can be used to enforce mutual exclusion. Mutual exclusion based] on message passing.
  在分佈式環境中，沒有共享變量（信號量）和本地內核可用於強制互斥，基於消息傳遞的互斥。
- There are two basic approaches to mutual exclusion: non-token-based and token-based.
  互斥有兩種基本方法：非基於令牌的和基於令牌的。
- The central coordinator algorithm is based on the availability of a coordinator process which handles all the requests and provides exclusive access to the resource. The coordinator is a performance bottleneck and a critical point of failure. However, the number of messages exchanged per use of a CS is small.
  中央協調器算法基於處理所有請求並提供對資源的獨占訪問的協調器進程的可用性。協調器是性能瓶頸和故障的關鍵點。然而，每次使用CS交換的消息數量很少。
- The Ricart-Agrawala algorithm is based on fully distributed agreement for mutual exclusion. A request is multicast to all processes competing for a resource and access is provided when all processes have replied to the request. The algorithm is expensive in terms of message traffic, and failure of any process prevents progress.
  Ricart-Agrawala算法基於完全分佈式的互斥協議。一個請求被多播給所有競爭資源的進程，並且當所有進程都響應該請求時提供訪問。該算法在消息流量方面是昂貴的，並且任何進程的失敗都會阻止進展。
- Ricart-Agrawala's second algorithm is token-based. Requests are sent to all processes competing for a resource but a reply is expected only from the process holding the token. The complexity in terms of message traffic is reduced compared to the first algorithm. Failure of a process (except the one holding the token) does not prevent progress.
  Ricart-Agrawala的第二種算法是基於令牌的。請求被發送到所有競爭資源的進程，但只有持有令牌的進程才會收到回復。與第一種算法相比，降低了消息流量方面的複雜性。進程的失敗（持有令牌的進程除外）不會阻止進程。
- The token-ring algorithm very simply solves mutual exclusion. It is requested that processes are logically arranged in a ring. The token is permanently passed from one process to the other and the process currently holding the token has exclusive right to the resource. The algorithm is efficient in heavily loaded situations.
  令牌環算法非常簡單地解決了互斥問題。要求進程在邏輯上排列成一個環。令牌永久地從一個進程傳遞給另一個進程，當前持有令牌的進程對資源擁有獨占權。該算法在負載較重的情況下是有效的。

Summary (Leader Election)

總結（領袖選舉）

- For many distributed applications it is needed that one process acts as a coordinator. An election algorithm has to choose one and only one process from a group, to become the coordinator. All group members have to agree on the decision.
  對於許多分佈式應用程序，需要一個進程充當協調器。選舉算法必須從一組中選擇一個且僅一個進程，成為協調器。所有小組成員都必須就該決定達成一致。
- The bully algorithm requires the processes to know the identifier of all other processes; the process with the highest identifier, among those which are up, is selected. Processes are allowed to fail during the election procedure.
  Bully算法要求進程知道所有其他進程的標識符；在那些up的進程中選擇具有最高標識符的進程。在選舉過程中允許進程失敗。
- The ring-based algorithm requires processes to be arranged in a logical ring. The process with the highest identifier is selected. On average, the ring based algorithm is more efficient then the bully algorithm.
  基於環的算法需要將進程安排在邏輯環中。選擇具有最高標識符的進程。平均而言，基於環的算法比欺凌算法更有效。
- Industry based approaches - Google Chubby
  基於行業的方法 - Google Chubby

Distributed Deadlocks

分佈式死鎖

- Deadlocks is a fundamental problem in distributed systems.
  死鎖是分佈式系統中的一個基本問題。
- A process may request resources in any order, which may not be known a priori and a process can request resource while holding others.
  一個進程可以以任何順序請求資源，這可能不是先驗已知，進程可以在持有時請求其他資源。
- If the sequence of the allocations of resources to the processes is not controlled, deadlocks can occur.
  如果資源分配給進程的順序是不受控制，可能會發生死鎖。
- A deadlock is a state where a set of processes request resources that are held by other processes in the set.
  死鎖是一組進程請求資源的狀態由集合中的其他進程持有
- Conditions for a deadlocks
  死鎖的條件
  - Mutual exclusion, hold-and-wait, No-preemption and circular wait.
    互斥、保持等待、非搶占和循環等待。

Modeling Deadlocks

建模死鎖

- In addition to the standard assumptions (no shared memory, no global clock, no failures), we make the following assumptions:
  除了標准假設（沒有共享內存，沒有全局時鐘，無故障），我們做出以下假設：
- The systems have only reusable resources.
  這些系統只有可重用的資源。
- Processes are allowed to make only exclusive access to resources.
  進程只允許對資源進行獨占訪問。
- There is only one copy of each resource.
  每個資源只有一個副本。
- A process can be in two states: *running or blocked.*
  一個進程可以有兩種狀態：運行或阻塞
  - In the running state (also called *active state), a process* has all the needed resources and is either executing or is ready for execution.
    在運行狀態(也稱為活動狀態)，一個進程擁有所有需要的資源並且正在執行或準備執行。
  - In the blocked state, a process is waiting to acquire some resource.
    在阻塞狀態下，一個進程正在等待獲取一些資源
- The state of the system can be modeled by directed graph, called a *wait for graph (WFG).*
  系統的狀態可以用有向圖建模，稱為等待圖（WFG）。
  - In a WFG , nodes are processes and there is a directed edge from node P1 to mode P2 if P1 is blocked and is waiting for P2 to release some resource.
- A system is deadlocked if and only if there exists a directed cycle or knot in the WFG.
  在WFG中，節點是進程，並且存在從節點P1到模式P2如果P1被阻塞並正在等待P2釋放一些資源
  系統死鎖當且僅當存在有向循環或WFG中的結。