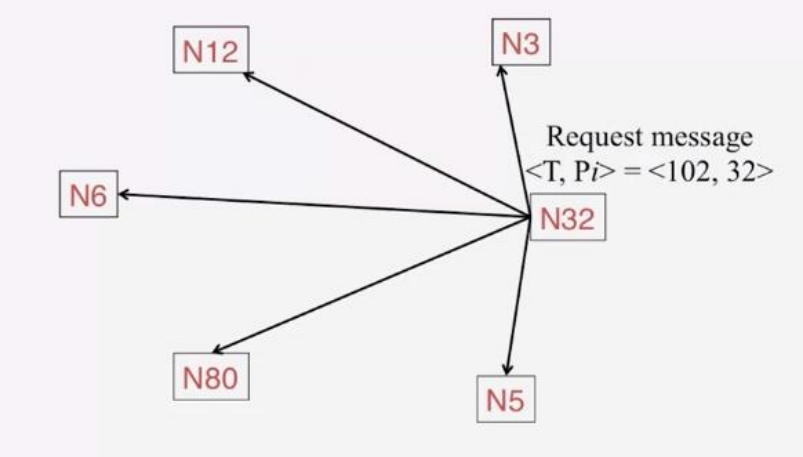
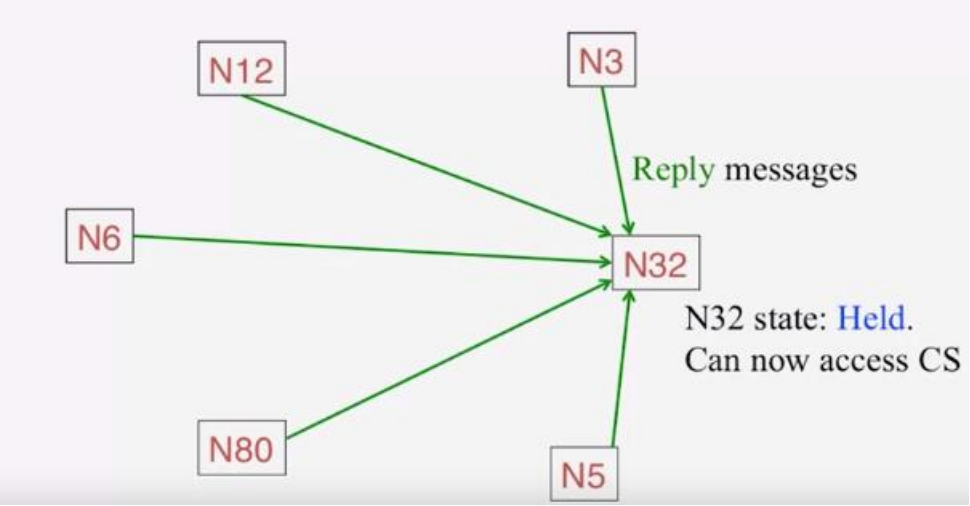
Ricart-Agrawala Algorithm

- enter() at process P*i*
  - set state to Wanted
  - multicast "Request" <T*i*, P*i*> to all processes, where T*i* = current Lamport timestamp at P*i*
  - wait until ***all*** processes send back "Reply"
  - change state to Held and enter the CS
- On receipt of a Request <T*j*, P*j*> at P*i* (*i* ≠ *j*):
  - **if** (state = Held) or (state = Wanted & (T*i*, *i*) < (T*j*, *j*))
    // lexicographic ordering in (T*j*, P*j*)
    add request to local queue (of waiting requests)
    else send "Reply" to P*j*
- exit() at process P*i*
  - change state to Released and "Reply" to *all* queued requests.

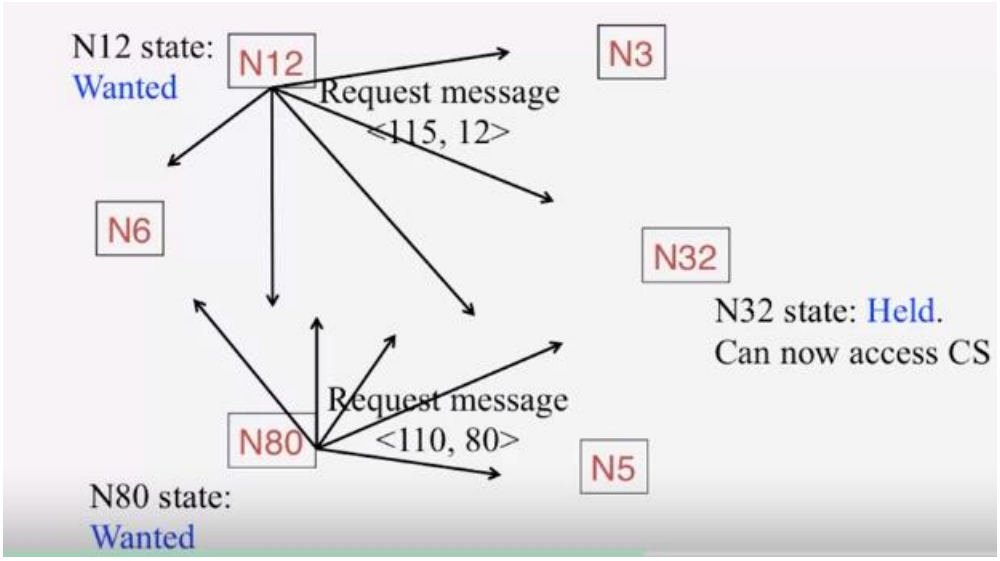
上面是公式， 下面是範例



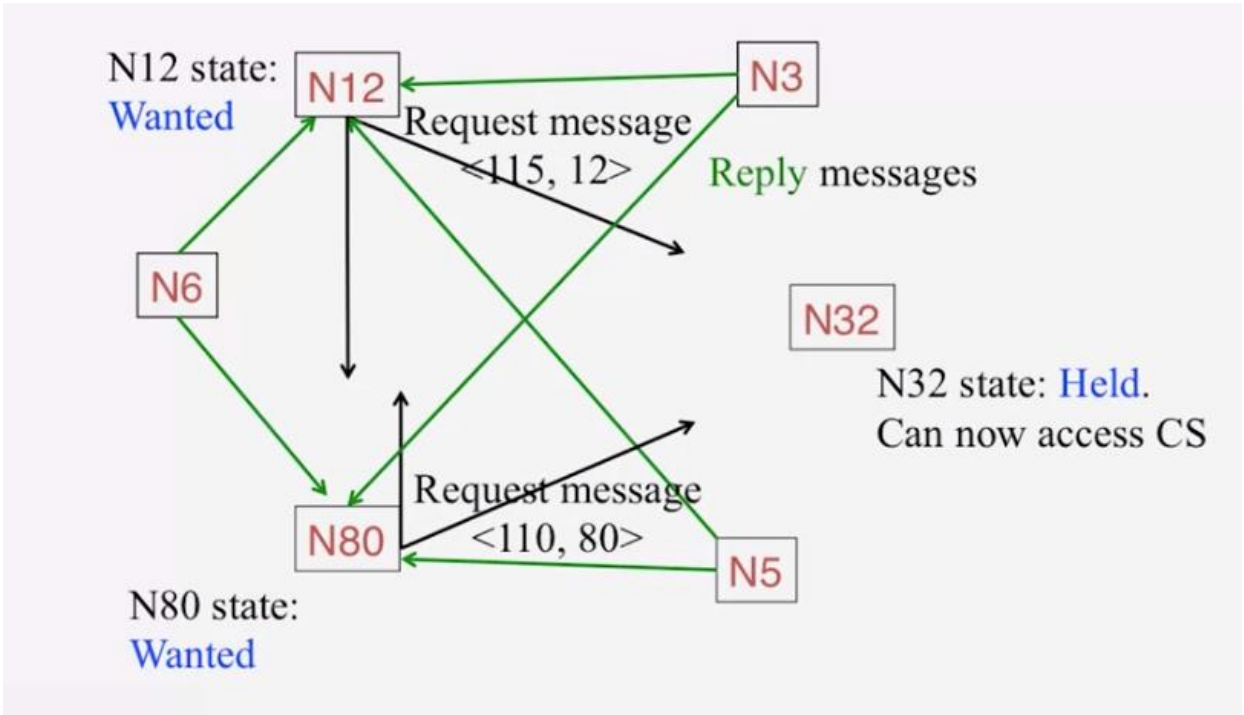
當 N32 有需求 Ti 是時間 Pi 是 ID



發出請求時其他點馬上就回應, 因只有 N32 有需求, 然後 N32 收到 N-1 個回复
所以進入 Held 可去存取 CS (critical section)

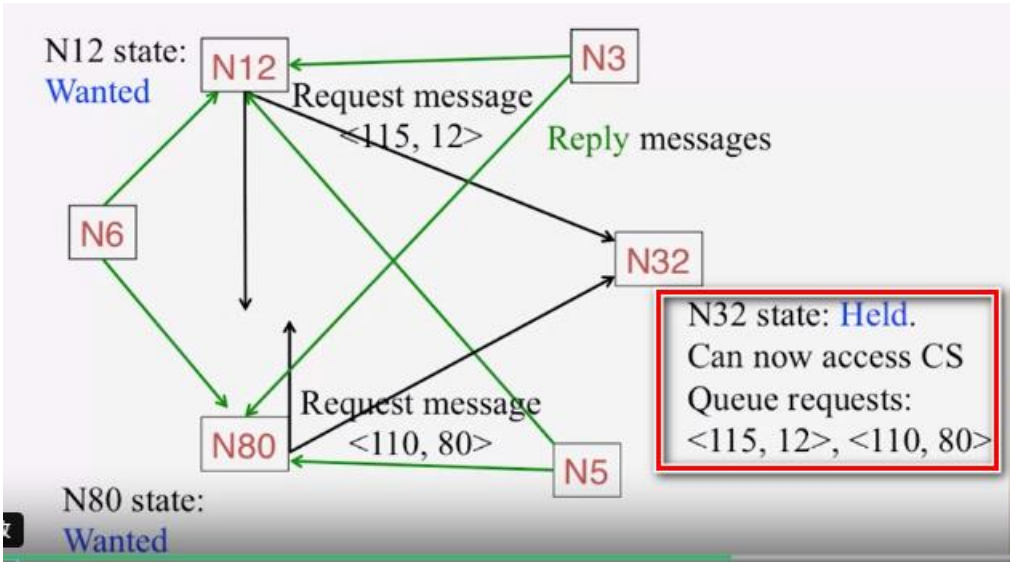


N12 , N80 發出請求
N12 時間 115 所以發出 (115 ,12)
N80 時間 110 所以發出 (110 ,80)



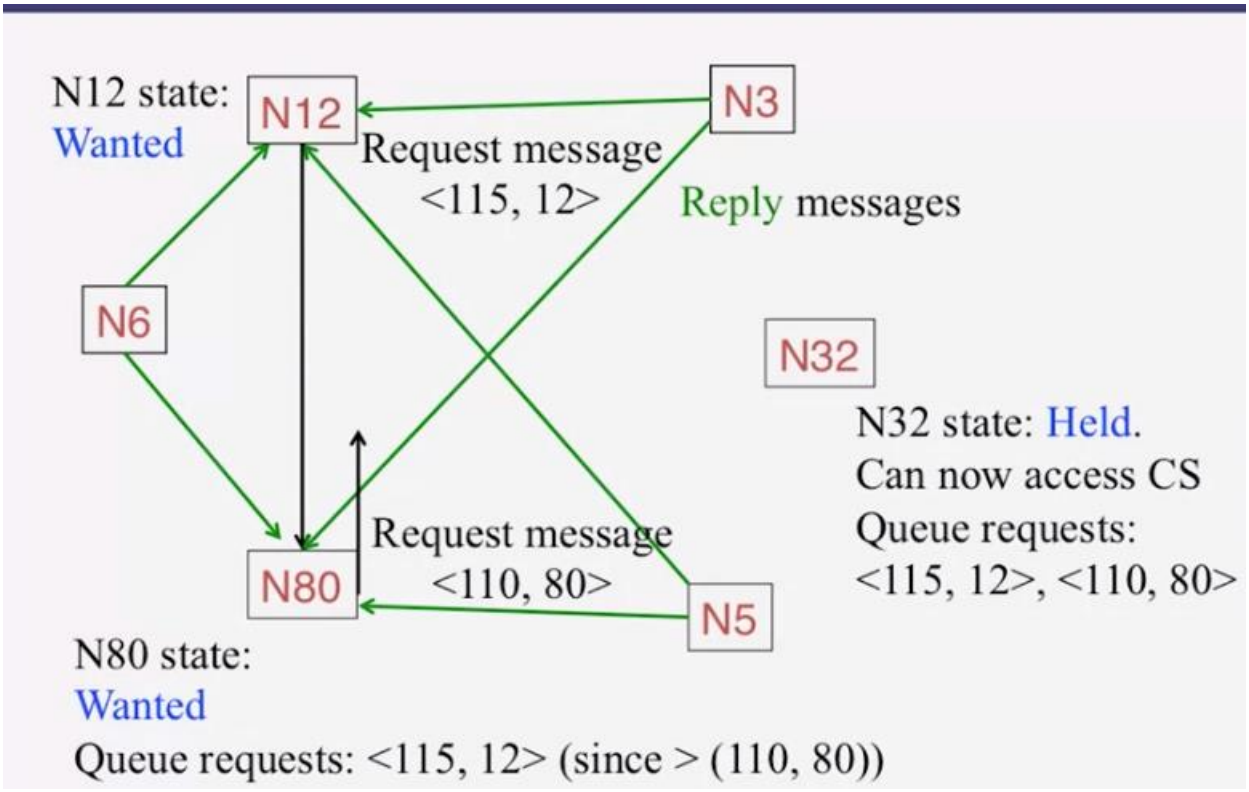
因為 N3、N6 和 N5 ，不是 Held 或 Wanted 所以會立即回复。
此時，N12 和 N80 各收到 3 條回复消息(N3、N6 和 N5 來的)
，但他們正在等待另外兩條回复消息。(N12 等 N32 與 N80 , N80 等 N32 與 N12)

來看 N32 , 因為 N32 狀態現在是 Held , 它會將這兩個請求排隊，包括時間戳和 ID



N32 會將收到的放到 Queue 裡面(順序不重要收到就放進去)

N32 不會回復 N12 與 N80 (**只要有放到 Queue 的就不會回復**)



假設 N80 首先收到 N12 的請求，且 N80 它的狀態是 Wanted

因此它會檢查傳入請求的時間戳是否低於它。

它看到傳入請求的時間戳為 115。**這是高於 N80 自己的請求。**

因此 N80 將 N12 放入 Queue 裡面。

N80 變成不會發送回復消息給 N12。 N12 要一直等待 N80 的回復消息。

就是 N80 會判斷 N80 的時間比 N12 優先所以不會回應

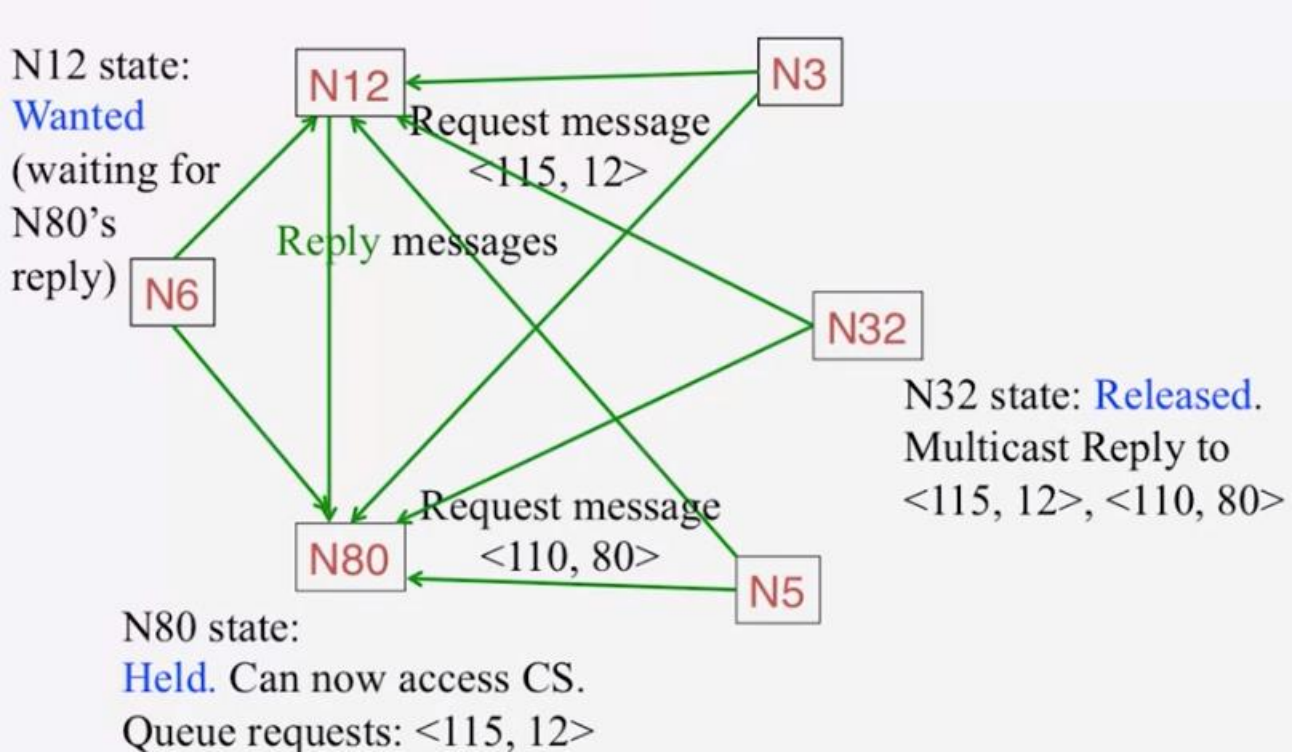
N80 會判斷 N80 的時間比 N12 優先所以不會回應 (反之不會放到 Queue 會回)

N12 收到 N80 的請求，且 N12 它的狀態是 Wanted

因此它會檢查傳入請求的時間戳是否低於它。

它看到傳入請求的時間戳為 110。這是低於 N12 自己的請求。

在這種情況下，N12 回復 N80(沒放到 Queue 會回)。



當 N32 完成了工作後 Held 變成 Released 會通知 N12 與 N80 (Queue 裡面取出)
N80 就收到 N-1 個回复了 就會變成 Held 去工作了
N12 收到 N-2 個回复 N80 還沒回)
最後 當 N80 完成了工作後 Held 變成 Released 會通知 N12 (Queue 裡面取出)
N12 就收到 N-1 個回复了 就會變成 Held 去工作了

Held 就是可以 CS (critical section)的許可 ….. 就是可以工作了

In case of equal timestamps, the process with the lower ID wins.
在時間戳相等的情況下，ID 較小的進程獲勝。

- Safety
  - Two processes P*i* and P*j* cannot both have access to CS
    - If they did, then both would have sent Reply to each other
    - Thus, (T*i*, *i*) < (T*j*, *j*) and (T*j*, *j*) < (T*i*, *i*), which are together not possible
    - What if (T*i*, *i*) < (T*j*, *j*) and P*i* replied to P*j*'s request before it created its own request?
      - Then it seems like both P*i* and P*j* would approve each others' requests
      - But then, causality and Lamport timestamps at P*i* implies that T*i* > T*j* , which is a contradiction
      - So this situation cannot arise

安全的 Pi 與 Pj 不會同時存取 CS
基於因果關係的 Lamport 時間戳生成的任何請求都將具有更高的時間戳。

- Liveness
  - Worst-case: wait for all other (*N-1*) processes to send Reply
- Ordering
  - Requests with lower Lamport timestamps are granted earlier

liveness 是正確的，因為在最壞的情況下，您的進程最終會等待所有其他 N-1 進程發送回復。
基於因果關係的 Lamport 時間戳生成的任何請求都將具有更高的時間戳。

OK, BUT ...

- Compared to Ring-Based approach, in Ricart-Agrawala approach
  - Client/synchronization delay has now gone down to O(1)
  - But bandwidth has gone up to O(N)
- Can we get *both* down?

因此，即使與基於環的方法相比，我們在客戶端/同步延遲方面做得更好。 帶寬增加了。 現在的問題是，我們能否在這兩種情況下做得更好。 我們將在下一講中看到這一點。