文章出處:

http://www.evanlin.com/moocs-coursera-cloud-computing/

- Data Center PUE (Power usage effectiveness)
  - 通常這個數字一定會大於 1 (業界大概是 1.4~1.5)

$$\text{PUE} = \frac{\text{Total Facility Energy}}{\text{IT Equipment Energy}}$$

- 分散系統中有談到 Map Reduce 要如何運作:
  - YARN 是透過 RM (Resource Manager) 來控制全部的系統工作分配．
  - MapReduce
    - Map 分配工作
    - Reduce 根據 key 將重複的結果合併起來．

Multiple-Cast:

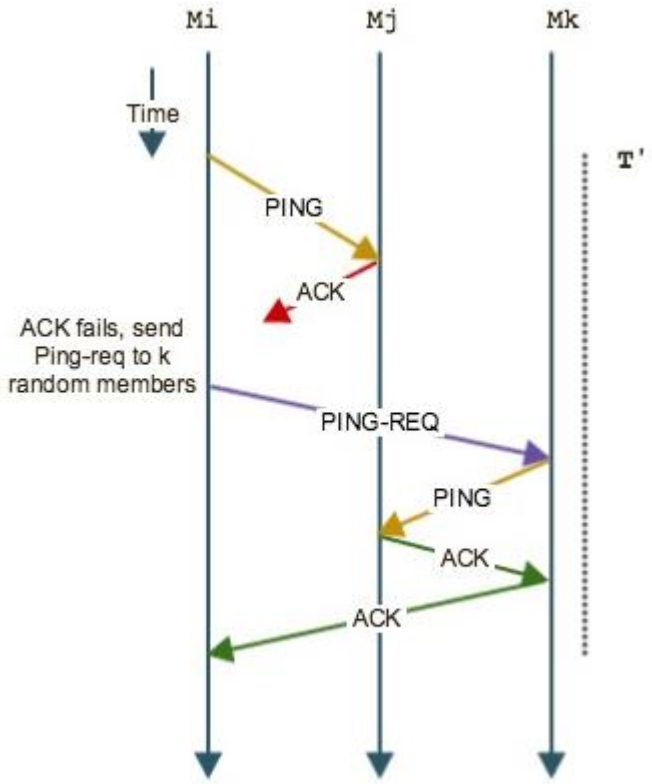
- 單節點向所有節點廣播
- 透過 ACK (Acknowlegement) 與 NACK (negative acknowledgements) 來表示有收到或是沒有收到．
- 時間複雜度: $O(n)$O(n)

Gossip

- Gossip broacast (UDP)
- There are two mode about broadcast:
  - Push
    - 時間複雜度 $O(\log_n)$O(logn)
  - Pull
    - 時間複雜度 $O(\log(\log_n))$O(log(logn))
- Fault Torrent:
  - Packet Lost: 50% 封包遺失，依舊可以正常運作．
  - Node Failure: 可以有一半左右的節點失效，還是可以正常運作．
- Gossip-Style Failure Detection (Heartbit)

- 每個節點會針對 member list (各自維護一份） 來做定期的 heartbit · （該時間為 T_{gossip}Tgossip)
- 每次收到之後，local member list 會更新並且把最新收到的時間更新進去 ·
- 只要時間超過了 T_{fail}Tfail 之後，就會被當成是無效節點 · 並且在 $T_{cleanup}$Tcleanup 之後來清理掉 ·
  - 為何需要兩個時間？ 有一種可能是，節點被認為失效了 · 但是過了 T_{fail}Tfail 忽然又活起來的話，就可以被加入回去 ·
- 結論: 如果 T_{fail}Tfail 與 $T_{cleanup}$Tcleanup 越大，越容易 false positive · 但是可以節省流量 ·

Probabilistic Faulure Detection (SWIM Gossip)



當 M_iMi 與 M_jMj 無法直接連接的時候，節點 M_iMi 會隨機發送到第 k 個節點 (M_kMk) 來做另一個方面的確認 · 如此一來，如果 M_kMk 會嘗試去 ping M_jMj ，如果取得正常的反應 · 就代表節點 M_jMj 依舊是活著，可能只是 toponology M_i−>M_jMi−>Mj 有問題發生 ·
當有 n 個節點，要能夠透過傳遞來知道一個節點壞掉需要透過 $O(\log(N))$O(log(N)) 的時間 ·

Dissemination and suspicion

- 傳播(Dissemination) Member List 的途徑:
  - Multicast (Machine-IP) 比較不可靠
  - TCP/UDP 比較可靠 (不過 UDP 也沒有 handshake)
  - 也可以透過 Piggybacked 就是 ping 夾帶 member list 方式來傳遞 member list

Napster

伺服器不存檔案，但是存每個節點的資料與檔案清單．並且也存放節點網路狀況．

- Server maintain <file, ip_address> tuple
- How client search:
    - Send key word to server
    - Server search tuple list, return ip list
    - Client ping each node to find transfer rate.
    - Client fetch file from best host.
- Using TCP

Problem:

- Centralize server
- Server cannot fixed SPOF
- No security (plaintext)

Gnutella

下一代的 Napster ，主要針對 Napster 集中式伺服器的問題來解決．Client act as server call **Servents**

Gnutella 有五種訊息種類:

- Query (Search)
- QueryHit (Reponse for Search)
- Ping (Hearbit)
- Pong (Response for Heartbit)
- Push (Init for transfer)

Message Format:

- Descriptor ID
- Payload Descriptor (Messavge Type)
- TTL (Time To Life)
- Hops (Increase by each node (hop))
- Payload Length

P.S.:

- TTL only use for QueryHit to provide distance of overlay network.
- 透過 HTTP 傳遞檔案
- 具有 Reverse-Routed 功能的只有 QueryHit ，因為他是回應 Query 的答案，需要具有回傳的的功能．

避免過多的 Query 流量

- 每個節點記住傳過的清單

- 每個節點只會轉達“一次”相同來源的 Query
- 重複的 Qeury 會被 drop

Problem:

- 太多的 Ping/Pong 網路流量．(約莫 50%)
    - 解法: 透過 cache 解決 heartbit 流量．

Fasttrack and Bittorent

Fasttrack 的特點

- 混合 Gnutella 與 Napster 的優點
- 某些節點(node)會變成 Supernode 也就是可以作為目錄的節點．

Bittorent

流程:

- 下載 .torrent 檔案
- 讀取裡面所有的 tracker (所謂的 tracker 就是管理所有 peer 的清單的伺服器）
- 分別到 tracker 去下載相關的 peer 清單
- 跟每一個 peer 去要求下載自己所需要的 block

選取檔案策略

- 採取 Local Rarest First 也就是會先找重複性最少的 block 來下載
- 如此一來可以增加檔案的健康度

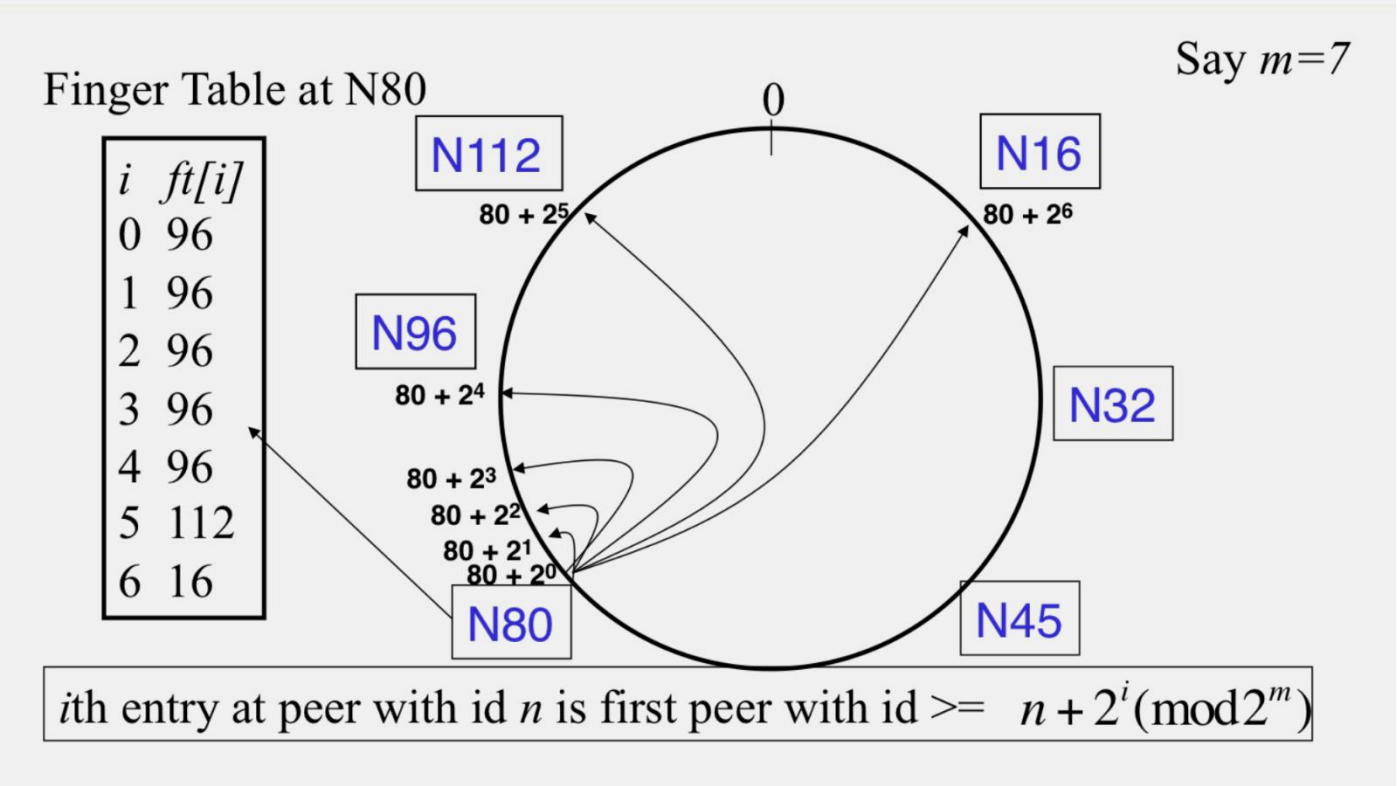
Chord

效能比對

	Memory	Lookup Latency	Message for lookup
Napster	O(1)	O(1)	O(1)
Gnutella	O(n)	O(n)	O(n)
Chord	O(log(n))	O(log(n))	O(log(n))

What is Chord

- A DHT (Destributed Hash Table)
- Using consistent hashing



Finger Table 計算方式

根據上圖

- 如果 m = 7
- 那麼如果要找出 N80 的 ft(finger table of N80) 就會是:
  - 80 + 2^0 = 80+ 1 = 81 > 80 –> 96
  - 80 + 2^1 = 80+ 2 = 82 > 80 –> 96
  - 80 + 2^2 = 80+ 4 = 84 > 80 –> 96
  - 80 + 2^3 = 80+ 8 = 88 > 80 –> 96
  - 80 + 2^4 = 80+ 16 = 96 = 96 –> 96
  - 80 + 2^5 = 80+ 32 = 112 > 96 –> 112
  - 80 + 2^6 = 80+ 64 = 144 mod 128 = 16 = 16 –> 16

How to handle Chord Failure?

尋找繼承者節點 (successor)

- N32
  - successor N45
  - presuccessor N16
- N112
  - successor N16
  - presuccessor N96

備份方式:

通常會有 r 份備份$r=2\log(N)$r=2log(N)

新增節點

如果增加新節點 N40 在 N32 與 N45 之間，那麼**有經過這兩個的上的 FT**(並不是代表這兩個節點..)就需要重新計算・

Pastry

特色

- 相當類似 Chord 使用 consistent hashing table
- 每個葉節點 (Leaf Node) 會知道自己前一個(Predecessor) 葉節點跟後一個(Successor)葉節點
- Routing Table 採取 **Prefix Matching**
    - 時間複雜度就是 $\log(N)$log(N)
    - 由第一個位元開始比對，相同 prefix 最多的就是 “best next-hop”

ex:

10110110111

的 best next-hop 就是 101101101”0”1

Cassandra

Replica Strategy (備份的策略)

Simple Strategy:

就是簡單地透過 Partition 來在同個地方備份多份資料・ 這邊有兩種方式:

- Random Partition: 類似 Chord 的 Hashing (Consistent Hashing Ring)
- ByteOrderedPartitioner: 直接給予一個範圍的來做切割

Network Topology Strategy:

如果你的 Cassandra 是跨多個 DC(Data Center) 的話，你就必須要參考這樣的備份方式・ 可能是一個資料中心 (DC) 有 2~3 份的備份・

NetworkTopologyStrategy:

- 會不斷的尋找 replica 直到不同 rack 為止。
- 舉例: Clockwise N1 ~ N6. N1, N2 in Rack1. N3 N4 in Rack2.. N5, N6 in Rack 3.
    - 如果第一個 Replica 在 N3 ，則下一個 Replica 會出現在 N5。 因為要透過 clockwise 尋找出不同 Rack 的機器。 N4 在同一個 Rack 所以不選。要選下一個 N5 。

對於 Network Topology 方式而言， **Snitches** 提供一個方式可以針對資料中心 (DC) 以及機架 (Rack) 來辨識的方式。 提供以下方式，細節可以看文件:

- Simple Snitch: 不在意各種網路架構（連 Rack 也不在意)
- RackInferring: 假設分類與你的 IP 有關:
    - `102.103.104.105 = X.<DC>.<Rack>.<Node>`
    - 舉例而言:
        - 同個 Rack : 102.103.104.122, 102.103.104.123
        - 同個 DC 不同 Rack: 102.103.**104**.122, 102.103.**112**.123
- PropertyFile Snitch: 透過設定檔
- EC2 Snitch: AWS EC2 的區域來判別 DC, Zone-> Rack
    - Eg: `X.<EC2 Region>.<Avaliable Zone>.<Node>`

讀與寫的方式

Write:

- 如果某個 replica 斷線， Coordinator 會先寫在自己這邊等待恢復
- 如果全部的 replica 都斷線， Coordinator 會本地端暫存一下 (buffer)

當一個 Replica 收到 Write 的指令:

- 先寫 commit log file
- 寫在 MemTable
- 記憶體滿的話，就 flush 到 SSTable (Sort String Table)
- 透 Bloom Filter 來尋找有沒有存放該資料

刪除(Delete)

- 不會馬上刪除，會加上一個 tombstone (墓碑)
- tombstone 的資料再 Compaction (SSTable 滿了需要壓縮與精簡) 發生的時候就會刪除

READ:

- 任何命令都會發送給 Coordinator ，然後尋找真正資料儲存的 Partition
- 發送查詢到所有的 replica ，等到"特定個數 X "的 replica 回覆就回答給查詢的人
- 收到各個 replica 的資料會比對，如果有不同會做一個 read repair 的動作來更新錯誤的 replica

Suspicion Mechanisms

Cassandra 透過 suspicion mechanism 來處理斷線或是結點出問題．

PHI 代表一個 heartbeat 變異數，也就是 timeout 的間隔．

`Eg: PHI=5, timeout 10 ~ 15`

Note: This already deprecated by Cassandra

CAP Theorem

資料庫的三大定理:

- Consistency: 所有節點都要能在同一個時間讀到相同資訊
- Availibility: 系統要在任何狀況下都要能夠運作，並且快速回覆．
- Partition-Tolerance: 系統即使被切割的狀況下，要能夠繼續運作．

在一般的分散式系統中，通常只能有兩個能夠滿足．或是應該說三個只能有兩個被完全滿足，第三個可能會部分滿足．

Eg:

- Cassandra:
  - Eventually (weak) consistency, Availiability, Partition-tolerance.
- RMDBSs:
  - Strong consistency, Availiability, no Partition-tolerance.

BASE (Basically Available Soft-state Eventual consistency)

Eventually Consistency:

If all writes stop all its values will converge eventually.

Quorem

Quorem 就是選舉 Leader 的機制，而對於參加選舉的主機

R: 具有讀取的主機數 N: 所有的主機數 W: 具有寫入權限的主機數

必須滿足以下的格式：

- `W + R > N`
- `W > N/2`

Consistency 系列

- Strong Consistency (RMDBs)
    - 就一般的強一致性
- CRDTs
    - 只允許每次加一的變更數值。
- Probabilistic
- Red-Blue
    - 分成藍色指令跟紅色指令，紅色必須要在同個 DC 中保持特定順序，藍色則不需要。
- Per-key sequential
- Causal
- Eventual
    - 所有寫入動作停止後，資料就全部會一致

HBase

Feature:

- Yahoo 開源
- Facebook 內部使用
- API:
    - Get/Put (row)
    - Scan (row range filter)
    - MultiPut
- 比較重視 Consistency

架構:

- 切割成不同區域 (regions) 分散在不同的備份主機上
- ColumnFamily 就是一群的欄位 (column)
- Store:
    - 就是一個 ColumnsFamily + Region
    - MemoryStore 放在記憶體中的 Store

HFile 結構:

Refer to [Cloudera Blog: Apache HBase I/O – HFile]

- 主要都是 key/value 架構，一個 HFile 包含多個 key/value pair
- 每一個 key/value 內容包含著
    - Key length
    - value length
    - row id
    - col family length
    - col family
    - ts
    - key type
    - value

如何達到 Strong Consistency : Hbase Write-Ahead Log

流程:

- client 寫入數個資料 k1, k2, k3, k4
- 透過 HRegionServer 查到 k1, k2 在 region 1 而 k3, k4 在 region 2
- 透過 HRegion 找到相關的 HFile
- 這時候先將 log 寫到 HLog ，可以寫入失敗的時候可以再度重做
- [預防資料遺失] 先將資料寫入 Hlog 然後才會去修改 MemStore
- 透過 Store 裡面的 MemStore 將 HFile 裡面的數值修改

Time and Ordering

Introduction

時間 (time) 指的是各個系統中用來同步的 clock ，在單機上面都是使用 CPU 的時脈作為所有內部軟體的時間資訊，來同步之用．

但是在分散式系統下，時間就變得難以同步．而每一個網路中的動作都需要的 ts 也就難以同步． 困難的地方有：

- 每一台機器有自已的 CPU 時脈
- 如果時間沒有同步 Message Delay 跟 Process Delay 就無法正確的限制

這裡有兩個名詞:

- **Clock Skew**: 指的是兩個時間 (clock) 在速度上相同但是有起始點的差異

- **Clock Drift**: 指的是兩個時間 (clock) 雖然起始點相同，但是在速度上不同

所以相同速度，不同起始時間的兩個 clock 有著 non-zero clock skew but **zero** clock drift

多久需要同步一次兩個 clock ?

如果最多能夠忍受時間相差 M 分鐘 (Clock Skew M) 那麼 `M/(2 * MDR)` 就需要同步一次 .

參考:

- HBase – 存储文件 HFile 结构解析

Network Time Protocol (NTP)

NTP 為一個樹狀結構的方式來同步時間

根據以上的圖形

- offset = ((t1-t0) + (t2-t3))/2
- round-trip delay = (t3-t0) - (t2-t1)

Lamport Timestamps

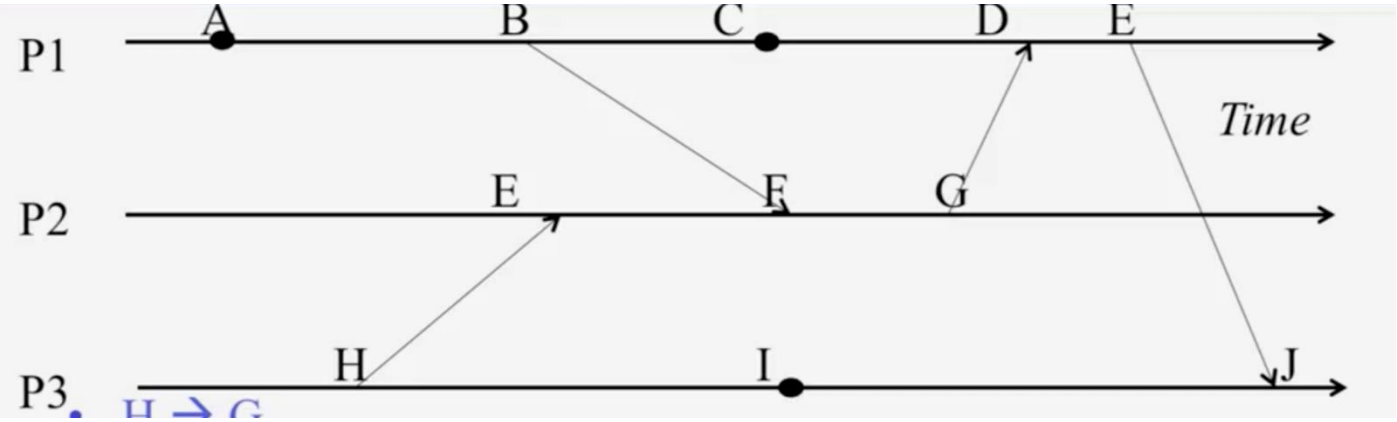
這個定理就是由 Paxos 的作者提出的，當初他就是在做 Lamport Timestamps 的時候想到利用類似的方是可以解決 Consensus Problem 的方法 .

基礎定理與標記

–> : 代表的是 Happen Before ，也就是左方的事件一定比右方的事件還早發生，不論雙方的時間究竟有沒有同步 .

- a –> b : time(a) < time(b) 同步過的時間必定 time(a) < time(b)
- send(m) -> receive(m) : 因為傳送必定有網路需要傳遞的時間，所以開始傳送的時間必定比接受到的時間還前面 .
- 遞移律 a –> b, b –> c 則必定 a –> c

透過一張圖來講解更多關於 Lamport Timestamp



針對這張圖，稍微講解:

- P1, P2, P3 不一定是具有同步的 timestamp
- P1 左到右是直線的，具有因果關係，也就是 A –> B (A happen before B)
- 有向的箭頭代表著某人傳訊息給另外一方， B ->(箭頭) F 代表著是 B 傳訊息給 F ，由於基礎定理 send(b) -> receive(f)，所以 B –> F (B happen before F)

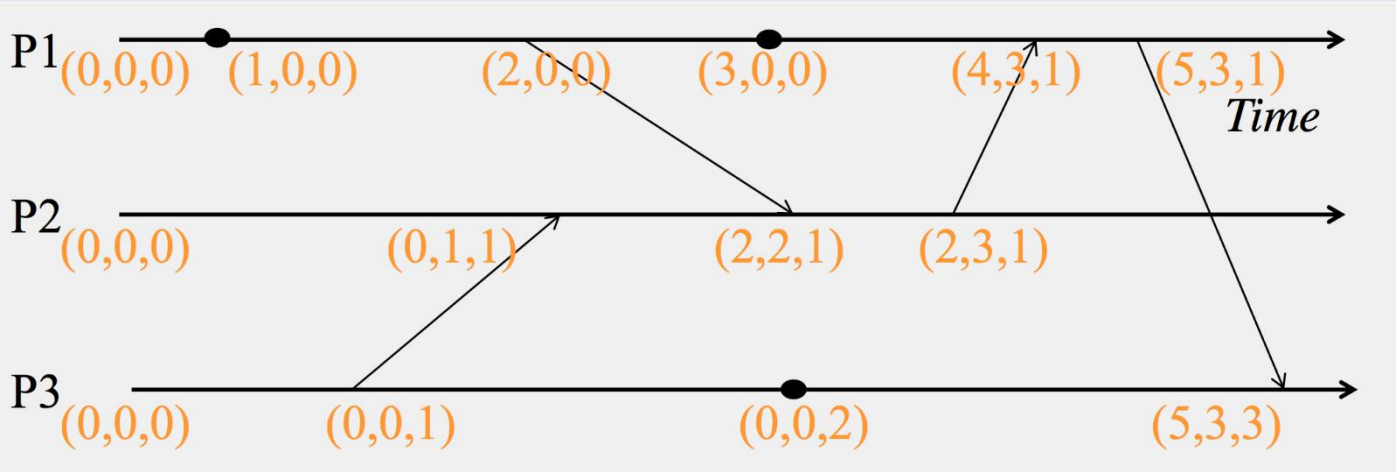
幾個範例：

- F –> G
- F –> J
- H –> J
- C –> J
- A –> F : A–> B ; B –> F ; A –> F

針對 Lamport Timestamp 計算時間上，如果 send(b) -> receive(f) ，則透過時間算法為:

```
max(local clock, message timestamp) + 1
```

Vector Timestamps



這邊會有三個資料 (x1, x2, x3) ，其中 x1 代表循序的 P1 timestamps, x2 代表 P2 的 timestamps …

而傳訊息的時候，就會把其他兩個傳給對方．舉例而言， P1 （2, 0, 0) -> P2 原本前面是 (0, 1, 1) 本來應該是 (0, 2, 1) 但是由於 (2, 0, 0) –> (2, 2, 1) 就是 `(max(x1, y1), max(x2, y2), max(x3, y3))`

Lamport Timestamp v.s. Vector Timestamps

	Lamport Timestamps	Vector Timestamps
Timestamp Data	Single Integer	Tuple (x1, x2, …)
Causality	obey	obey
Identify Concurrent Events	No	Yes

Snapshots

Global Snapshot

Global Snapshot =

- Global State =Individual state + communication channel

時間不同步的時候所造成 Global Snapshot 會失敗的原因

- 時間不正確
- 無法抓到溝通的狀態

任何造成 Global Snapshot 變動的原因:

- Process send/receive message
- Process move one step

以一個範例來解釋演算法

基本定義:

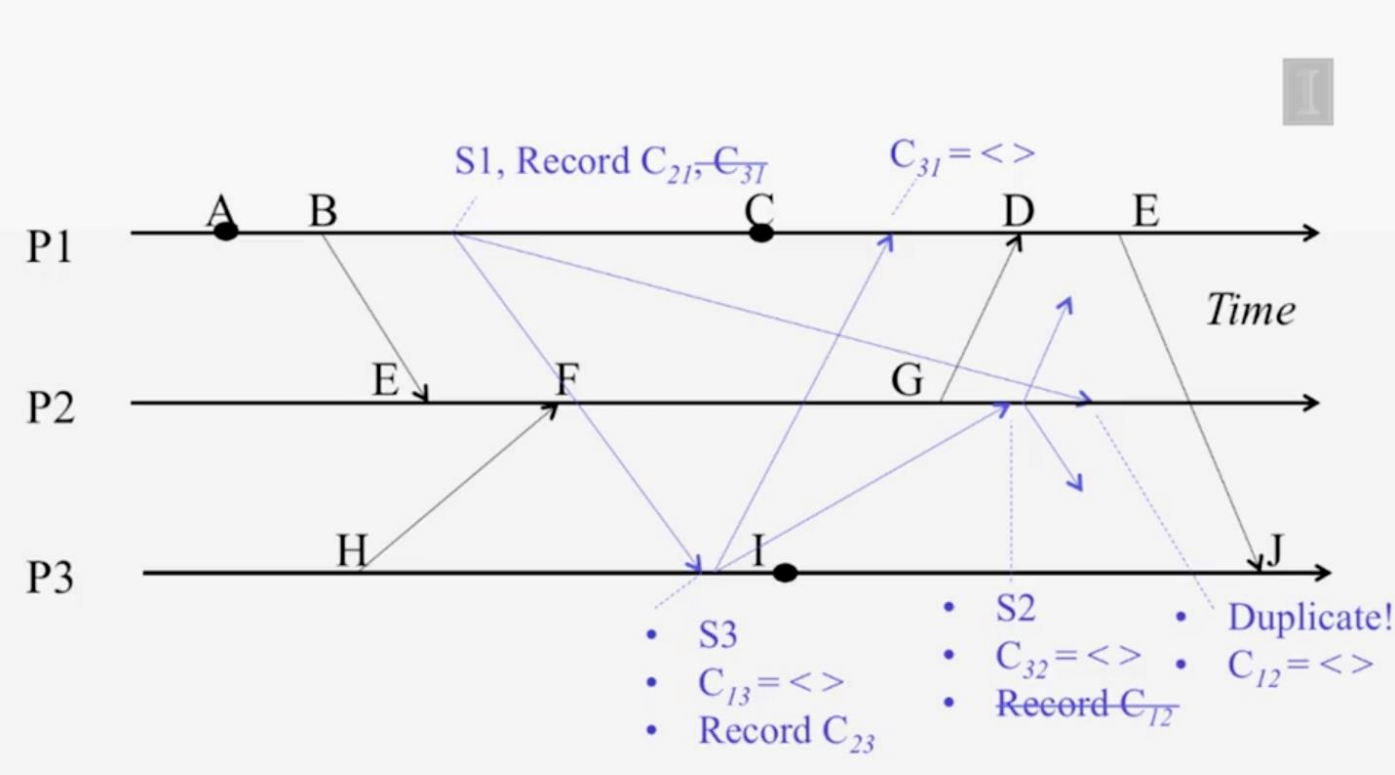
- There are no failures and all messages arrive intact and only once
- The communication channels are unidirectional and FIFO ordered
- There is a communication path between any two processes in the system
- Any process may initiate the snapshot algorithm
- The snapshot algorithm does not interfere with the normal execution of the processes
- Each process in the system records its local state and the state of its incoming channels

Chandy-Lamport Global Snapshot Algorithm

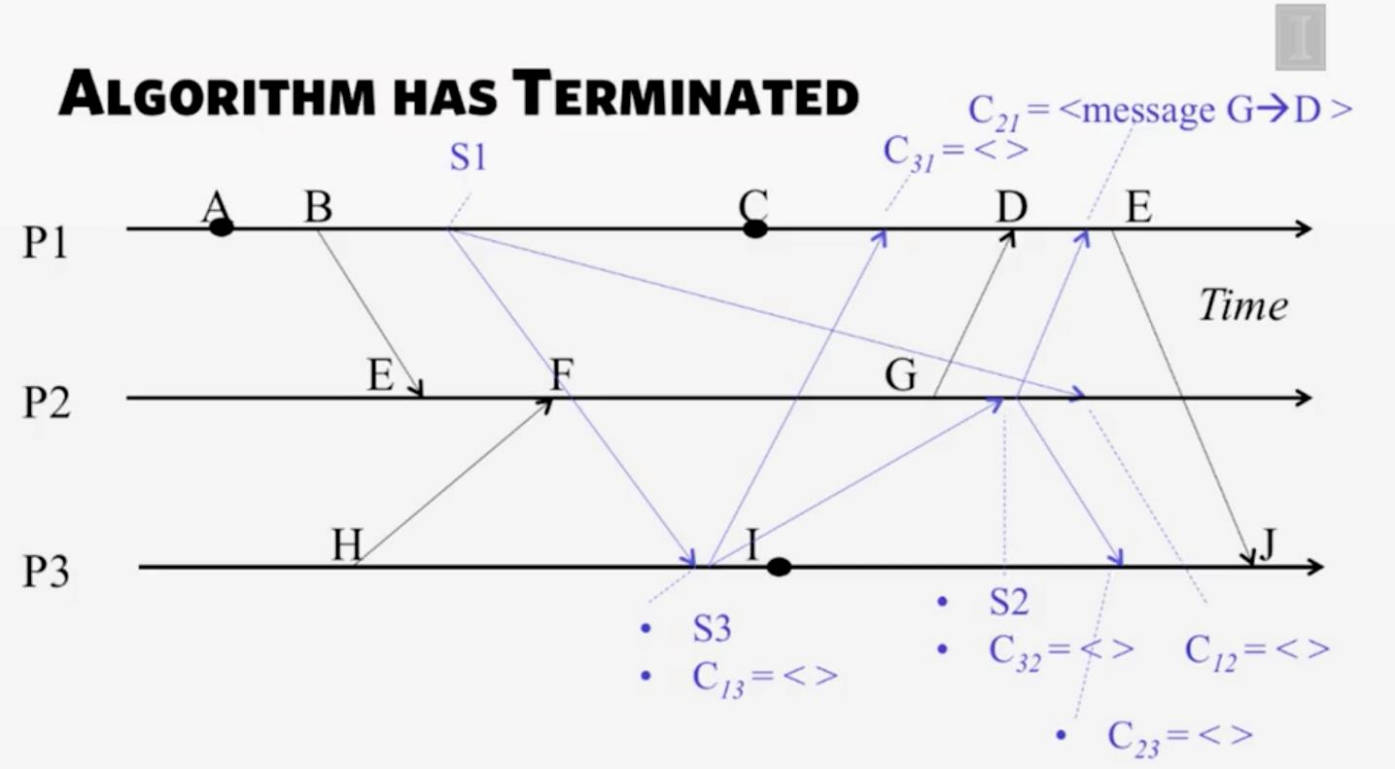
Wiki

- 隨便挑一個 Process (PiPi) 來對所有其他的 Process 傳送一個 Marker 訊息 CikCik)
    - 並且開始記錄所有進來的 Message (InComing Msg)
- 如果收到的 Process PjPj 並沒有收過 CijCij
    - 先將該訊息 CijCij 的 State 標示為 “empty” 一樣開始傳送 PjkPjk Marker 給其他 Process
- 如果已經收過了
    - 代表所有 Process 已經開始在傳遞 Marker 而其他訊息已經完整的到了．
- 停止條件:
    - 當所有節點都收到 Marker 代表儲存的資料已經完成．
    - 當所有 Process 收到其他訊息都是 Marker (需要有 N - 1 個)

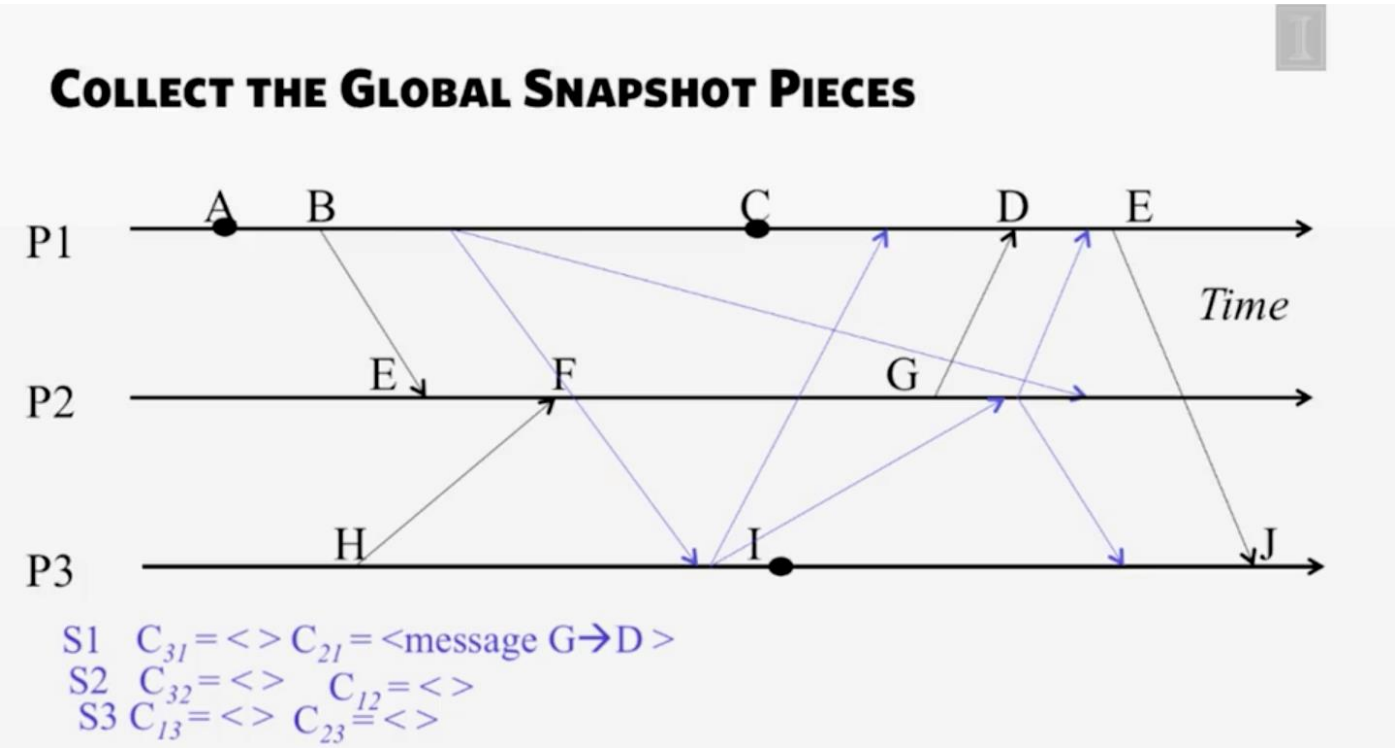
最後還要把各個分開的 Snapshot 組合成 Global Snapshot



停止條件，所有節點都收到 Marker 外，所有等待外部回來的資料 (marker) 也都收到．



將 Snapshot 統一回收整理成一個大的 snapshot



P.S. 這個演算法並不記錄對外的資料，因為由收到資料的人統一來記錄結果。

Consistent Cut

講解到 Snapshot ，不免俗就要講到擷取 snapshot 的時間點 (cut)。

Consistent Cut

- A cut C is a consistent cut if and only if: for (each pair of events e, f in the system)

也就是說每一個 pair f->e (f happen before e) 如果有 e in the cut 那麼必須 f 也必須要 in the cut． 反之不需要滿足．

> 透過 Chandy-Lamport Global Algorithm 截取的 Snapshot 都滿足 Consistent Cut

證明: e_j **in the cut (透過** P_j **記錄到)** 那麼 e_i **(透過** P_i **記錄到)** 必定也在 **snapshot** 之中．

- We already know $e_i -> e_j$ $e_i -> e_j$
- We also know e_j in the cut but e_i not in the cut
- Because $e_i -> e_j$ $e_i -> e_j$ and P_i already record state for e_i
- Since e_i happen before e_j, if e_i not in the cut so e_j must not in the cut too.
- Contracdition.

Safety and Liveness

Conrectness => Liveness or Safety

Definition:

- **Liveness**: something good *eventually* happen
- **Safety**: Guarantee something bad *never* happen

Chandy-Lamport Algorithm 與 Safety, liveness 的關係:

- 一旦 Stable 就會一直保持 Stable
- 只能滿足 Liveness but non-safety

Multicast

Communication 的種類:

- Unicast: 訊息從一個傳送者送到一個接收者
- Multicast: 訊息寄送給一群的人
- Broadcast: 訊息寄送給全部的人

Multicast Ordering

探討順序的時候，有以下三個方式的 multicast :

- FIFO Ordering
- Causal Ordering
- Total Ordering

FIFO Ordering:

這邊的 FIFO 代表的是，先發送的人就會被收到．（同一個 Process 上):

- 先發送的人，會先被人收到． (同一個發送的人，先後順序在某個節點上應該會保持一樣的順序．)
- Ex:
  - P_i−>P_j,P_k,P_lPi−>Pj,Pk,Pl, P_m−>P_j,P_k,P_lPm−>Pj,Pk,Pl
  - 根據 FIFO 原理
  - P_jPj 收到順序也會是 $P_i(P_ij)$,$P_m(P_mj)$Pi(Pij),Pm(Pmj)

Causal Ordering:

Causal Ordering 代表的是因果的關係，也又是接受到的人絕對比傳送的人還晚． （因為傳送過程的 latency) 這邊指的是跨 Processes ，的因果關係．

```
Causal Ordering -->(imply) FIFO Ordering
```

由於 Causal Ordering 可以跨 processes 跟同個 process ． 如果是在同個 Process 的話，就是指的是先入先出 (FIFO) 因為同一個的先後順序必定影響該 multicast 的順序．

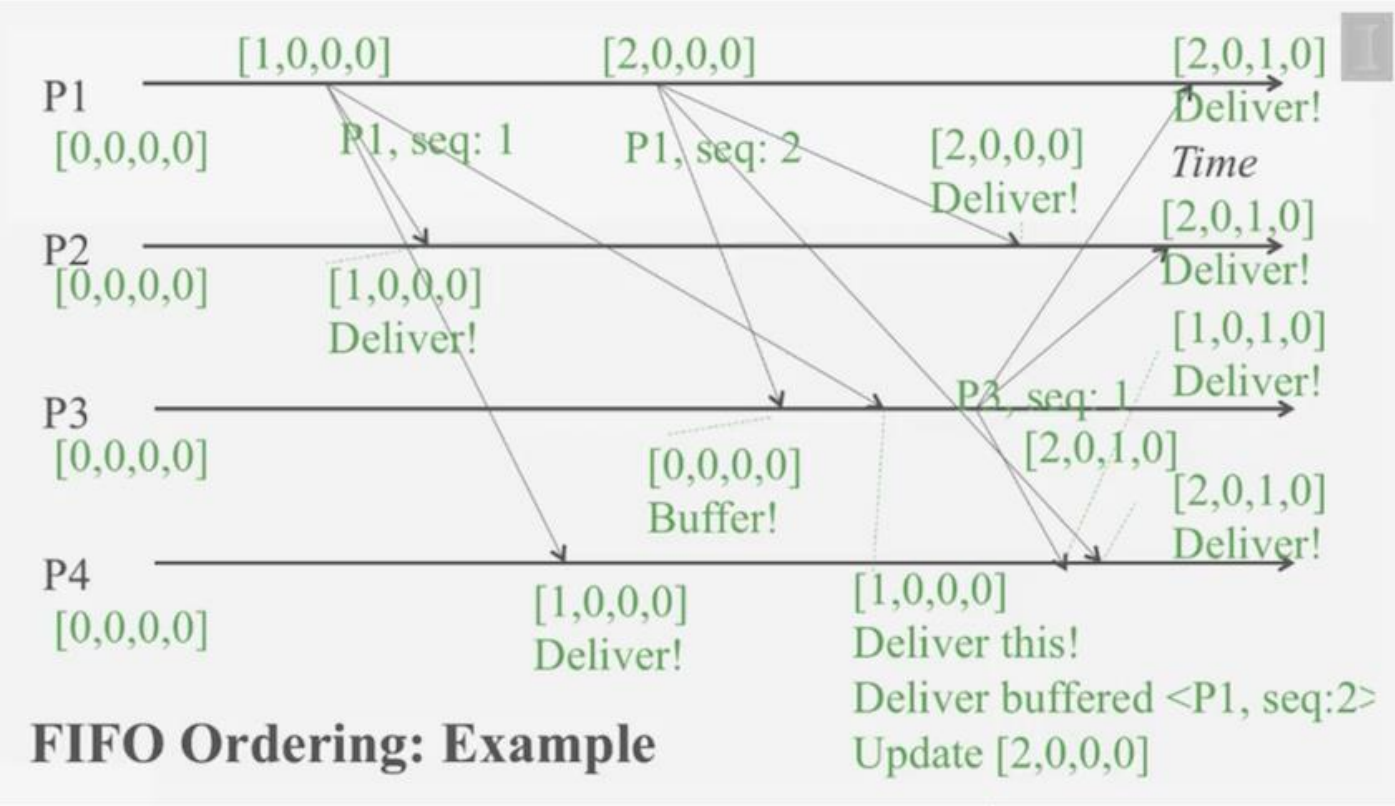
Total Ordering:

所有 process 收到訊息的順序都相同．

Hybrid ordering

- FIFO-Total:
- Causal-Total:

Implement of multicasting



Implement FIFO Ordering

拿這張圖來解釋如何讓 FIFO Ordering 能夠滿足:

- P1 有兩個 multicasting [1,0,0,0] 與 [2,0,0,0]
- 會看到在 P3 那邊 [2,0,0,0] 比起 [1,0,0,0] 先到
- 所以後面的需要先等一下 (buffered) 改成 [0,0,0,0] 等到後面到了才能改・
- 但是 P1, seq: 2 與 P3 seq: 1 不會有這樣的問題・因為 FIFO Ordering 並不局限於跨線程

About total ordering

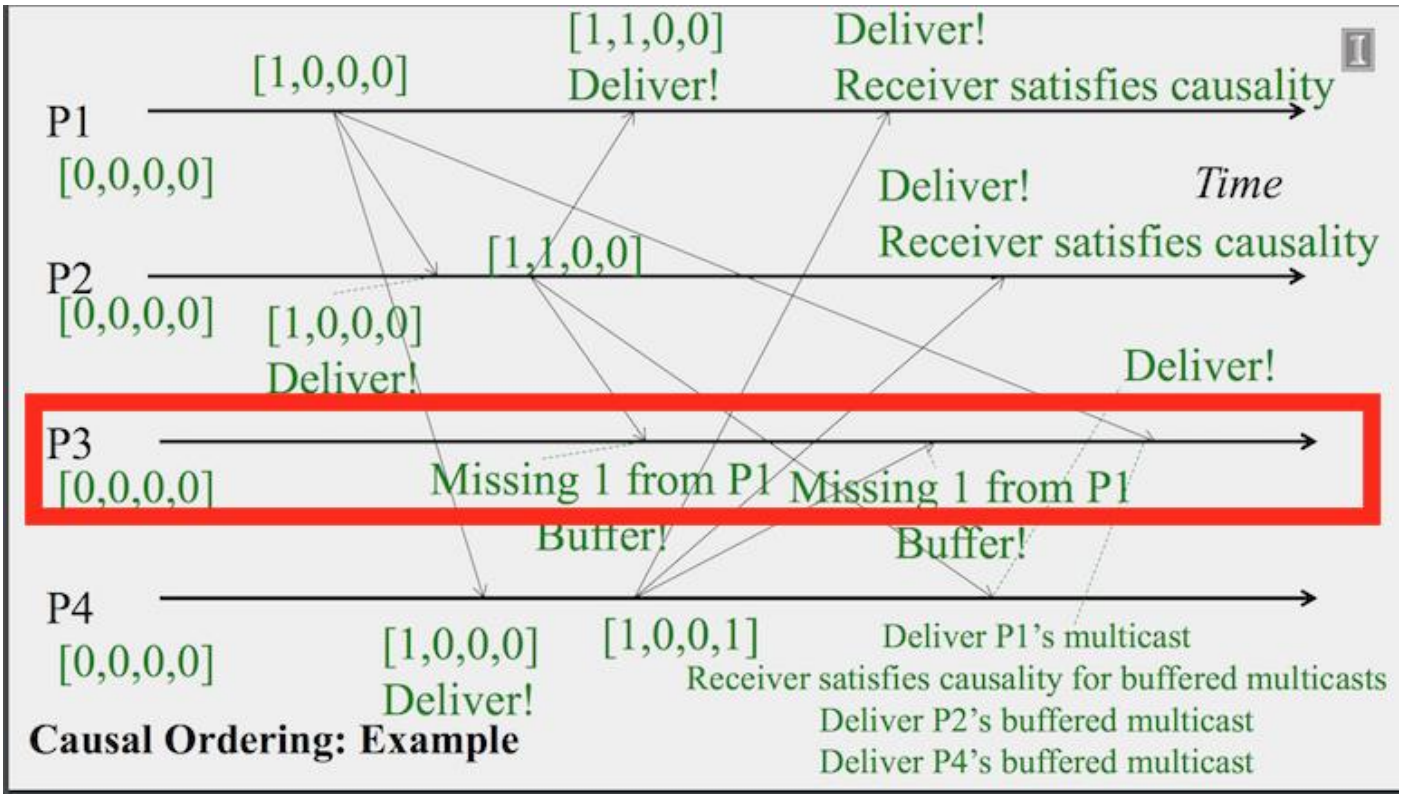
就是計算每個人收到的順序必須要跟開始發送的順序一樣（可跨程序） 就算是・計算的方式就是去累進收到的順序・

Implement Causal Ordering

滿足條件:

> 所有的接收者都必須有著相同的接受順序（跨程序）・

```
If multicast(g,m) -> multicast(g,m’)
then any correct process that
delivers m’ would already have
delivered m.
```



做法:

- P1 與 P2 原本就滿足 Causal Ordering ，故不提。
- P3 先收到 P2 msg ，這時候就知道要等待 P1 msg 因此就 buffer
- P3 後來又收到 P4 msg ，又必須要等待 P1 msg 所以就 buffer
- P3 要一直收到 P1 msg 之後，才會把 P2 與 P4 msg 收起來
- 如此一來就滿足 Causal Ordering

Reliability of multicast

```
Reliability = Correctness = Ordering
```

在此討論的 Reliability 都是基於 non-Faulty (沒有錯誤發生的) process 上面來討論。

現實中的實現方式

在現實中如何透過實現類似 multicast 的確保方式？

每個 process 收到之後，在發送 multicast 給其他 process 作為標記

透過這樣的方式，才能確保每個 process 收到的 msg 順序是正確的，因為順序需要有外部的資訊來紀錄．

Virtual Synchrony

View

每個 Process 都維護一套自己的 member list ，那樣的 member list 就稱為 “View”

View Change

當 Process 發生了 “新增”, “離開”, “故障” .. 等等影響 member list 的事情就稱為 View Change

Virtual synchrony

Virtual Synchrony 保證所有的 View Change 都會發生於相同的順序上．

ex:

- P1 join
- {P1, P2}, {P1, P2, P4} , {P1, P2, P3} are all view change
- 所以 P2 會收到 一樣順序 {P1, P2}, {P1, P2, P4} , {P1, P2, P3}

Paxos

Consensus Problem

問題釐清:

為何 SLA 永遠都是 five-9' 獲釋 seven-9' 永遠不能夠 100%?

解答:

所謂的錯誤，不光是任何人員控管與公司流程上的錯誤．當然也不在於機器本身的錯誤． 而是在於是否能夠達到一致性(Consensus)．

Consensus Statement:

Consensus Constraints:

- **Validity**: 需要全部人提議相同數值才能決定，
- **Integrity**: 最後決定的數值必定是某個 process 提議出來（不是無中生有)
- **Non-triviality**: 至少有一個初始狀態，而不是常見的 all-0, all-1

Consensus In Synchronous Systems

Synchronous Systems 的特性:

- 訊息有時間限制
- Process 間交換訊息有時間限制 (round: f, upper bound)

如果能夠解決 Synchronous Systems 的一致性 (Consensus) 問題 ，那麼是否能夠解決 Asynchronous System

–> 將訊息交換的時間限制延長到 N

Paxos

關於 Paxos

並沒有完全解決 Consensus 的問題，

但是可以提供兩種主要特性:

- Safety:
    - 不違背一制性
- Evantual liveness:
    - 如果有問題發生，還是可以達成一制性， (但是不保證，原因後續）

並且在以下系統都有使用:

- zookeeper
- Chubby

Paxos 三個階段:

Election

- 每個節點 (node) 會送出 election request

- 需要達到大多數 (majority) (或是稱為 Quorum)
- Paxos 允許 multiple leader (這裡先不討論那麼複雜)

產出會有一個(或多個) Leader

Bill

- Leader propose value (v)
- 如果節點同意會回覆

Law

- 如果 Leader 獲得大多數的同意
- 會發出 Law (or Learn) request 給每個節點，確認數值的紀錄