#1 Introduction

What is a distributed system?

A set of nodes, connected by a network, which appear to its users as a single coherent system.

Why study distributed system?

- Partial Failures
- Concurrency

Atomic Broadcast

- A node broadcasts a message.
- If sender correct, all correct nodes deliver msg.
- All correct nodes deliver the same messages.
- Message delivered in the same order.

Atomic broadcast can be used to solve Consensus!

- Receive the proposal in same order, so they can decide the same value.

Consensus can be used to solve atomic broadcast!

Atomic broadcast equivalent to Consensus!

Models of Distributed Systems

- Timing assumptions in processes/network/clocks
- Failure assumptions in processes (crash, stop and byzantine)/network (drop messages)

The Asynchronous Systems Model

No bound-on time to deliver a message.

No bound-on time to compute.

Clocks are not synchronized.

(Consensus cannot be solved in asynchronous system if node crashes can happen)

The Synchronous Systems Model

Known bound on time to deliver a message(latency).

Known bound on time to compute.

Known lower and upper bounds in physical clock drift rate.

(Consensus can be solved in synchronous system with up to N-1 crashes)

We need accurate crash detection: every node sends a message to every other node. If no msg from a node within bound, node has crashed.

Partially synchronous system

Initially system is asynchronous

Eventually the system becomes synchronous

(Consensus can be solved in partially synchronous system with up to N/2 crashes)

Failure detectors

Consensus and atomic broadcast solvable with failure detectors.

Timed Asynchronous system

No bound-on time to deliver a message.

No bound-on time to compute.

Clocks have known clock-drift rate.

Byzantine faults

Only tolerate up to 1/3 Byzantine processes.

Non-Byzantine algorithms can often tolerate 1/2 nodes in the asynchronous model.

#2 Basic Abstractions

The event-based component model

Set of processes and a network.

- **Computation step**
- **Communication step**

Components are concurrent and access local state, each component receives messages through an input FIFO buffer. Events are handled by procedures called event handlers.

The event-based programming

Each program consists of a set of modules or component specification.

Specification of a Service

1. Interface (contract, API): requests, responses
2. Correctness properties: safety, liveness
3. Underlying model: assumptions on failures, assumptions on timing

Correctness Properties

Safety: properties that state that nothing bad ever happens.

Liveness: properties that state something good eventually happens.

Execution and Traces

An execution fragment of A is sequence of alternating states and events

$s_0, \varepsilon_1, s_1, \varepsilon_2, \ldots, s_r, \varepsilon_r, \ldots$

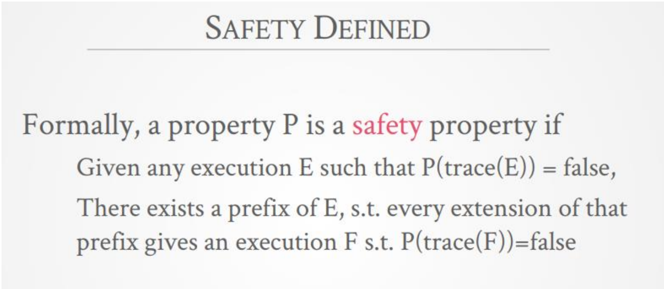
$(s_k, \varepsilon_{k+1}, s_{k+1})$ transition of A for $k \geq 0$

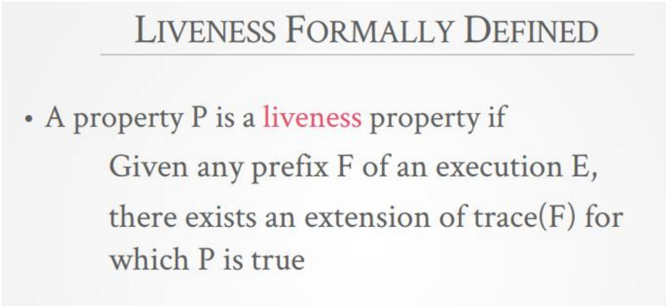
An execution is execution fragment where s_0 is an initial state

A trace of an execution E, trace(E)

The subsequence of E consisting of all external events

$\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_r, \ldots$





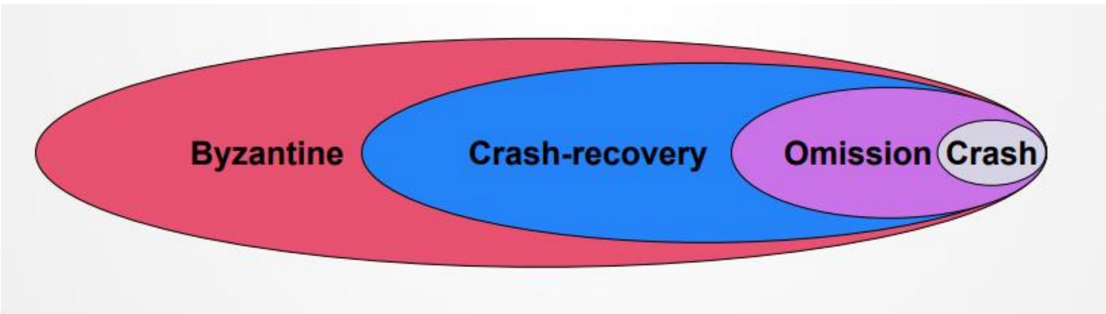
Process Failure Model

1. Process failures

- Crash-stop: process stops taking steps (no sending and no receiving).
- Omissions: process omits sending or receiving messages.
- Crash-recovery: 1. Process crashes and never recover. 2. Process crashes and recovers infinitely often(unstable).
- Byzantine / Arbitrary: A process may behave arbitrarily and maliciously.

Fault-tolerance Hierarchy

An algorithm that works correctly under a general form of failure, works correctly under a special form of failure.



2. Channel failures

- Fair-loss links: Channels delivers any message sent with non-zero probability (no network partitions)
- Stubborn links: Channels delivers any message sent infinitely many times.
- Perfect links: Channels that delivers any message sent exactly once.
- Logged perfect links: Channels delivers any message into a receiver's persistent store (message log)
- Authenticated perfect links: Channels delivers any message m sent from process p to process q, that guarantees the m is actually sent from p to q.

Fair-loss Properties:

1. Fair-loss: If **m is sent infinitely** often by pi to pj, and neither crash, then **m is delivered infinitely** often by pj.
2. Finite duplication: If a m is **sent a finite number of times** by pi to pj, then it **is delivered at most a finite number** of times by pj.
3. No creation: No message is delivered unless it was sent.

Stubborn links:

1. Stubborn delivery: if a correct process pi sends a message m to a correct process pj , then pj **delivers m an infinite number of times**.
2. No creation: if a message m is delivered by some process pj, then m was previously sent by some process pi.

Implementation: use the fair-loss link; sender stores every messages it sends and periodically resends all these messages.

Perfect links:

1. Reliable Delivery: **If pi and pj are correct**, then **every message** sent by pi to pj is **eventually delivered by pj**.
2. No duplication: Every message is delivered **at most once**.
3. No creation: No message is delivered unless it was sent.

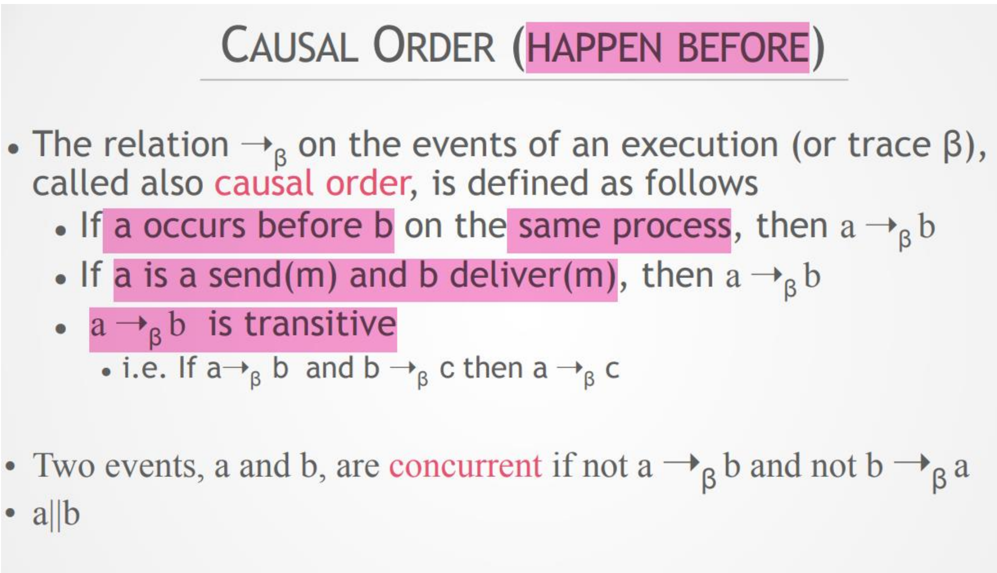
Implementation: use stubborn links and receiver keeps a log of all received messages, only deliver messages that weren't delivered before.

3. Timing assumptions

Asynchronous model and causality

No timing assumption and **reasoning model based on** which events may cause other event - **causality**.

Total order of event **not observable locally**, no access to global clocks



Computation theorem

- **Computation Theorem**:
  - Let E be an execution ($c_0, e_1, c_1, e_2, c_2, \ldots$), and V the trace of events ($e_1, e_2, e_3, \ldots$)
  - Let P be a permutation of V, preserving causal order
    - P=(f_1, f_2, $f_3 \ldots$) preserves the causal order of V when for every pair of events $f_i \rightarrow_V f_j$ implies f_i is before f_j in P
  - Then E is similar to the execution starting in c_0 with trace P

Equivalence of executions

If two executions F and E have the **same collection of events**, and their **causal order is preserved**, F and E are said to be **similar executions**, written F~E.

Two important results

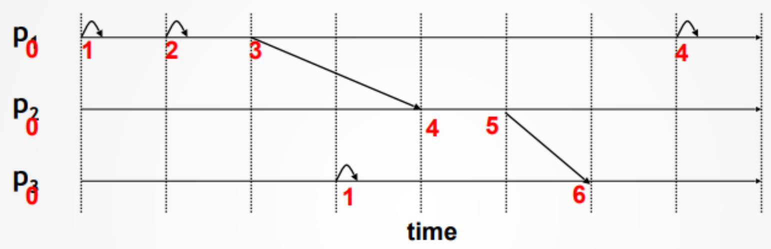
Result 1: no algorithm in the asynchronous system observes the order of the sequence of events for all executions.

Result 2: The computation theorem does not hold if the model is extended such that each process can read a local hardware clock

#3Logical Clocks

Lamport logical clocks

- A clock is function **t** from the events to a totally order set such that for events a and b
  - if $a \rightarrow b$ then $\mathbf{t}(a) < \mathbf{t}(b)$



Lamport logical clocks guarantee that:

$$\text{If } a \rightarrow_\beta b, \text{ then } \mathbf{t}(a) < \mathbf{t}(b),$$

$$\text{if } \mathbf{t}(a) \geq \mathbf{t}(b), \text{ then not } (a \rightarrow_\beta b)$$

Vector Clocks

- $v_p \le v_q$ iff
  - $v_p[i] \le v_q[i]$ for all i
- $v_p < v_q$ iff
  - $v_p \le v_q$ and for some i, $v_p[i] < v_q[i]$
- v_p and v_q are concurrent ($v_p \parallel v_q$) iff
  - not $v_p < v_q$, and not $v_q < v_p$
- Vector clocks guarantee
  - If v(a) < v(b) then a → b, and
  - If a → b, then v(a) < v(b)
    - where v(a) is the vector clock of event a

[3,0,0] ≤ [3,1,0]

[3,0,0] < [3,1,0]

[3,1,0] <> [4,0,0]

Summary

- the relation $\rightarrow_\beta$ on events in executions
  - Partial: $\rightarrow_\beta$ doesn't order concurrent events
- the relation < on Lamport logical clocks
  - Total: any two distinct clock values are ordered (adding pid)
- the relation < on vector timestamps
  - Partial: timestamp of concurrent events not ordered

Logical clock

If $a \rightarrow_\beta b$ then $t(a) < t(b)$ (1)

Vector clock

If $a \rightarrow_\beta b$ then $v(a) < v(b)$ (1)

If $v(a) < v(b)$ then $a \rightarrow_\beta b$ (2)

#4 Failure Detectors

Motivation

A failure detector can substitute(替代) timing assumption.

Spoiler alert: the accuracy of a FD relates to the strength of the underlying model.

Implementation idea

- Periodically exchange heartbeat messages
- Timeout based on worst case message round trip
  - If timeout, then suspect process
  - If received message from suspected node, revise suspicion and increase time-out

Completeness and accuracy

Failure detectors are **feasible only in synchronous and partially synchronous systems**

Strong Completeness: **Every crashed** process is **eventually** detected by **all correct** processes.

- Strong Completeness
  - Every crashed process is eventually detected by all correct processes

Weak Completeness: Every crashed process is **eventually** detected by **some correct** process.

- Weak Completeness
  - Every crashed process is eventually detected by some correct process

at least one

Strong accuracy: No correct process is ever suspected.

- Strong Accuracy
  - No correct process is ever suspected

Weak accuracy: There exists a correct process which is never suspected by any process.

- Weak Accuracy
  - There exists a correct process which is never suspected by any process

Eventual Strong Accuracy

After some finite time the FD provides strong accuracy

Eventual Weak Accuracy

After some finite time the detector provides weak accuracy

Class of failure detectors

Four detectors with strong completeness

- Perfect Detector (P)
  - Strong Accuracy
- Strong Detector (S)
  - Weak Accuracy

$\}$ Synchronous Systems

- Eventually Perfect Detector (◊P)
  - Eventual Strong Accuracy
- Eventually Strong Detector (◊S)
  - Eventual Weak Accuracy

$\}$ Partially Synchronous Systems

Four detectors with weak completeness

- Detector Q
  - Strong Accuracy
- Weak Detector (W)
  - Weak Accuracy

$\}$ Synchronous Systems

- Eventually Detector Q (◊Q)
  - Eventual Strong Accuracy
- Eventually Weak Detector (◊W)
  - Eventual Weak Accuracy

$\}$ Partially Synchronous Systems

具体内容看课件!!!

Leader elections

- Failure detection captures failure behaviour
  - Detect failed processes
- Leader election (LE) also captures failure behaviour
  - Detect correct processes (a single and same for all)
- Formally, leader election is a FD
  - Always suspects all processes except one (leader)
  - Ensures some properties regarding that process

LE with P (perfect failure detector)

INTERFACE OF LEADER ELECTION

Module:

Name: LeaderElection (le)

Events:

Indication: ⟨leLeader | p_i⟩

Indicate that leader is node p_i

they can disagree on the failed processes

Properties:

- *LE1 (eventual completeness).* Eventually every correct process trusts some correct process
- *LE2 (agreement).* No two correct processes trust different correct processes
- *LE3 (local accuracy).* If a process is elected leader by p_i, all previously elected leaders by p_i have crashed

说法更加 focus on 找哪些是 correct,以及对 correct 一致的 agreement.

Eventual LE Ω with ◊ P (eventual perfect failure detector)

INTERFACE OF EVENTUAL LEADER ELECTION

Module:

Name: EventualLeaderElection (Ω)

Events:

Indication (out): ⟨Ω, Trust | p_i⟩

Notify that p_i is trusted to be leader

Properties:

ELD1 (eventual completeness). Eventually every correct node trusts some correct node

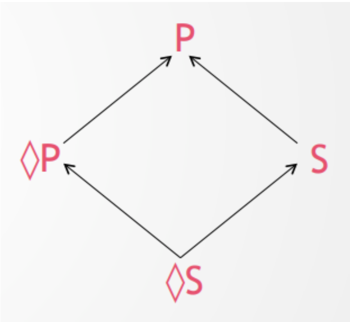
ELD2 (eventual agreement). Eventually no two correct nodes trust different correct node

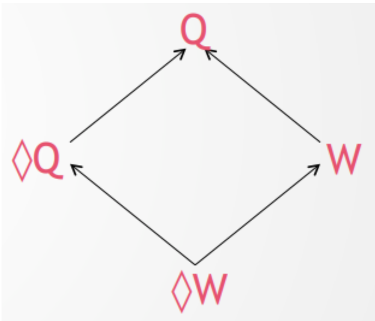
Reductions

We say X⩽Y if X can be solved given a solution of Y. Read X is reducible to Y.

- A relation ≼ is a preorder on a set A if for any x,y,z in A
  - x ≼ x (reflexivity)
  - x ≼ y and y ≼ z implies x ≼ z (transitivity)
- Difference between preorder and partial order
  - Partial order is a preorder with anti-symmetry
    - x ≼ y and y ≼ x implies x = y
- For preorder two different objects x and y can be symmetric
  - It is possible that x ≼ y and y ≼ x for two different x and y, (x ≠ y)

- We write X≃Y if
  - X≼Y and Y≼X
  - Problem X is equivalent to Y
- We write X≺Y if
  - X≼Y and not X≃Y
  - or equivalently, X≼Y and not Y≼X
  - Problem X is strictly weaker than Y, or
  - Problem Y is strictly stronger than X





COMPLETENESS "IRRELEVANT"

- Weak completeness trivially reducible to strong
- Strong completeness reducible to weak
  - i.e. can get strong completeness from weak
    - P≼Q, S≼W, ◊P≼◊Q, ◊S≼◊W,
  - They're equivalent!
    - P≃Q, S≃W, ◊P≃◊Q, ◊S≃◊W

Every process q broadcast suspicions Susp periodically.

Every crash is eventually detected by all correct p.

Accuracy

Completeness	Strong	Weak	Eventual Strong	Eventual Weak
Strong	P	S	◊P	◊S
Weak	Q	W	◊Q	◊W

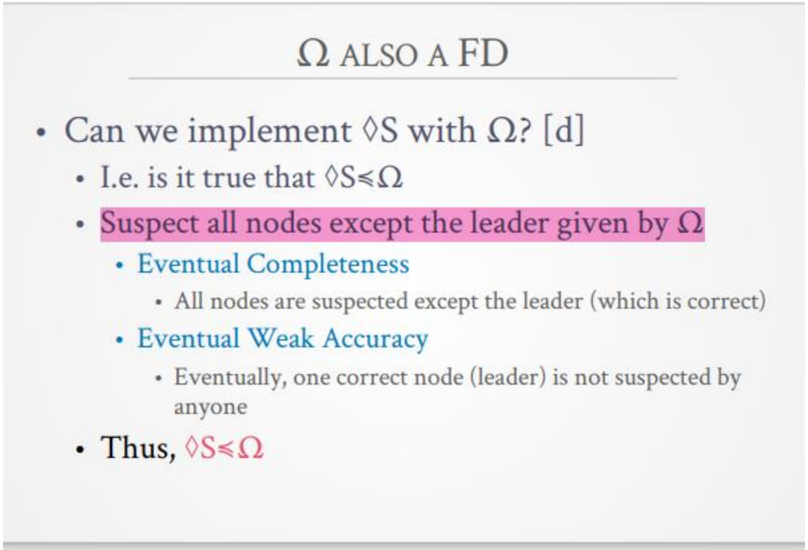
ID2203

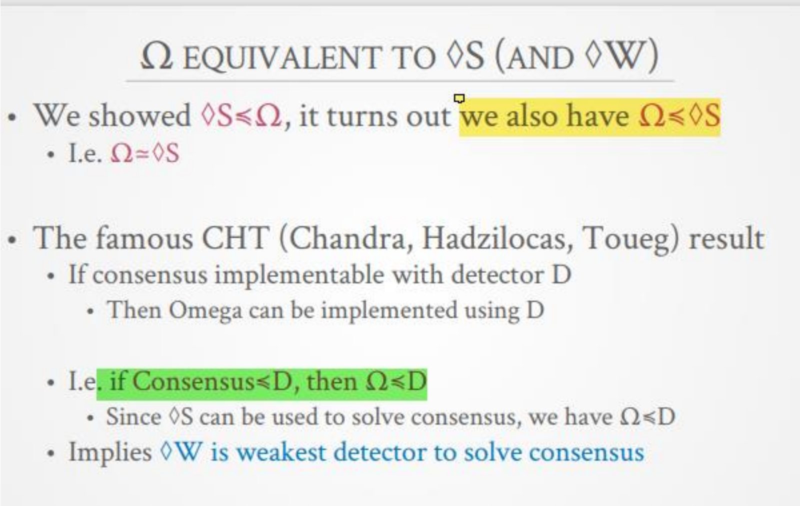
KTH

如何证明 trivially reducible 和 maintain accuracy 的细节去看 ppt.

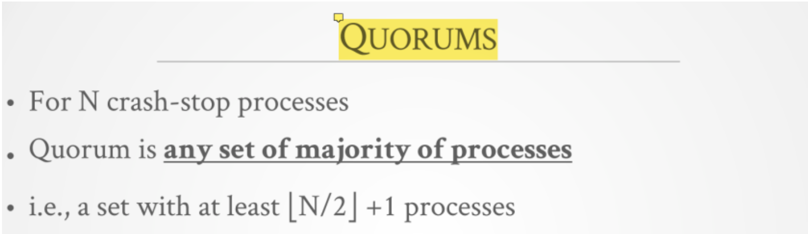
Ω also a FD

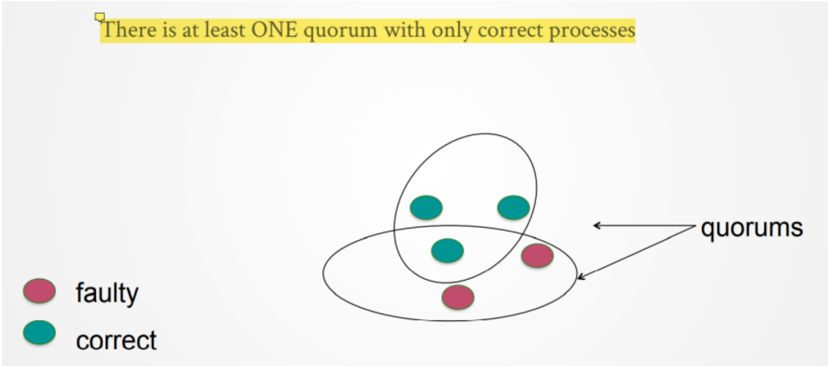
这部分看课件，有点复杂。

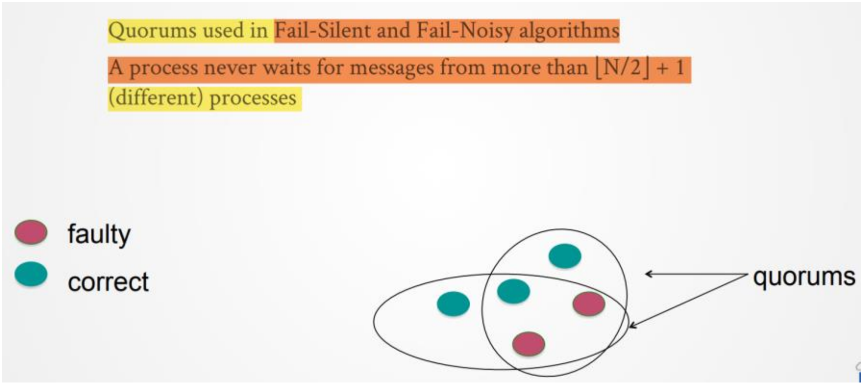




#5 Reliable Broadcast







Best-effort broadcast

Properties:

- **Best-effort-validity**

If p1 and p2 are correct, then any broadcast by p1 is eventually delivered by p2.

- **No duplication**

No message delivered more than once.

- **No creation**

No message delivered unless broadcast.

Reliable broadcast (if sender crashes, ensure all or none of the correct nodes get msg)

Properties:

Validity

If correct p1 broadcasts m, p1 itself eventually delivers m.

No duplication

No creation

Agreement

If a correct process delivers m, then every correct process delivers m.

Uniform reliable broadcast (**if a failed node delivers, everyone must deliver**, at least correct nodes)

RB validity

No duplication

No creation

Uniform agreement

For any message m, if a process delivers m, then every correct process delivers m.

Implementing BEB

-The bundles (fail-stop, fail-silent and fail-noisy) have to access perfect channel.

-We can just send m to all processes by perfect channel.

-The channel guarantees that if both sides are correct then the message is going to be delivered on the other side).

Implementing the Lazy RB in fail-stop model (synchronous model)

-We have perfect failure detector and perfect channel.

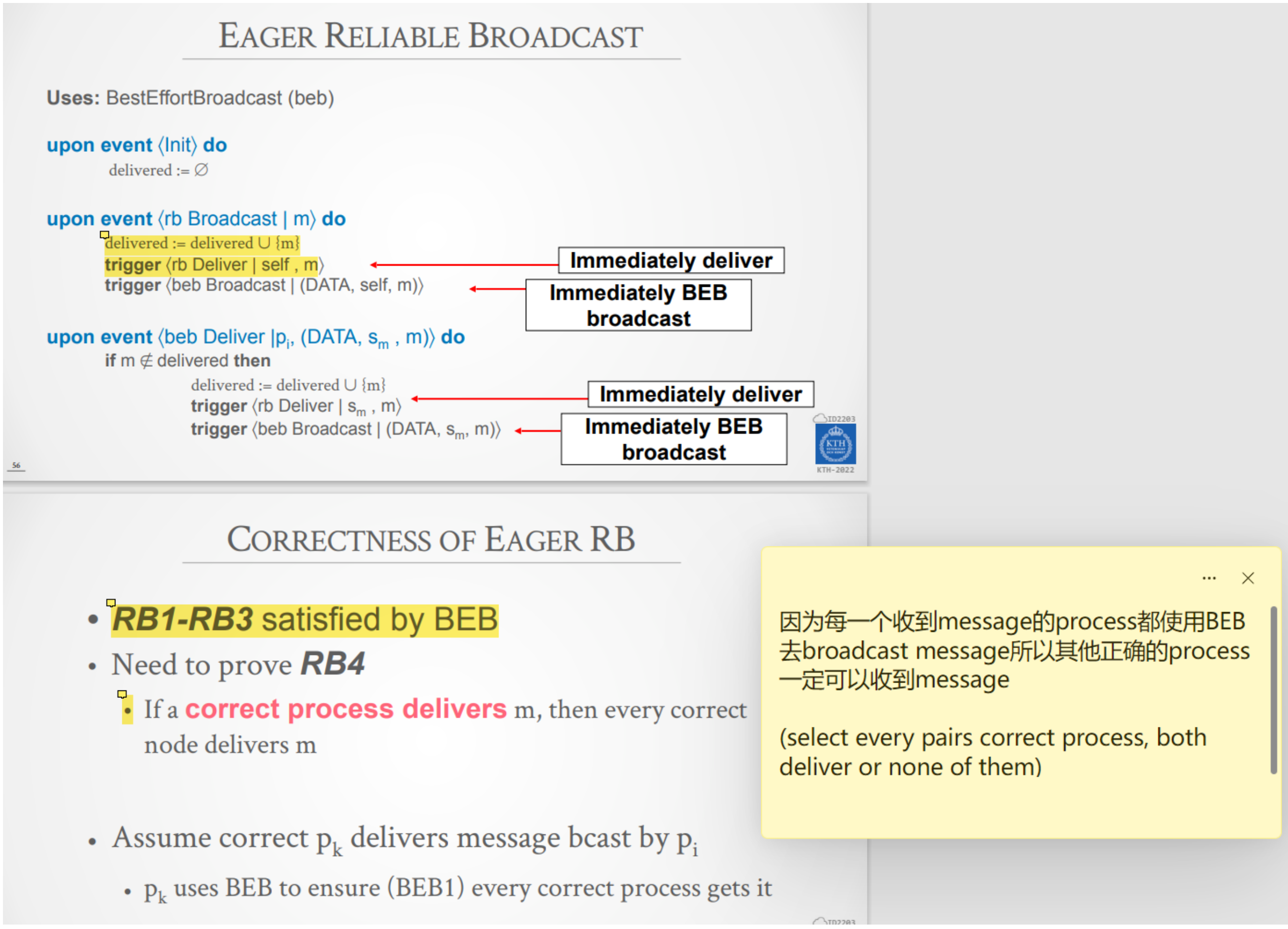
If sender s crashes, detect& relay(继续发送) msgs from s to all

-case 1: get m from s, detect crash s, redistribute m.

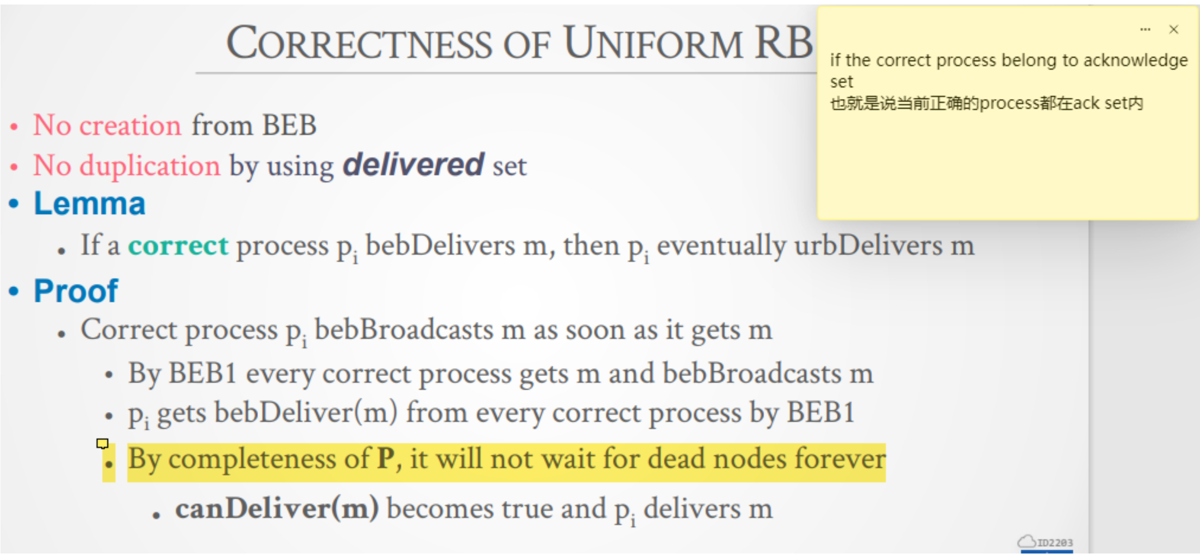
-case 2: detect crash s, get m from s, redistribute m.

Implementing the Eager RB in fail-silent model (asynchronous model)

Modify lazy RB to not use P?



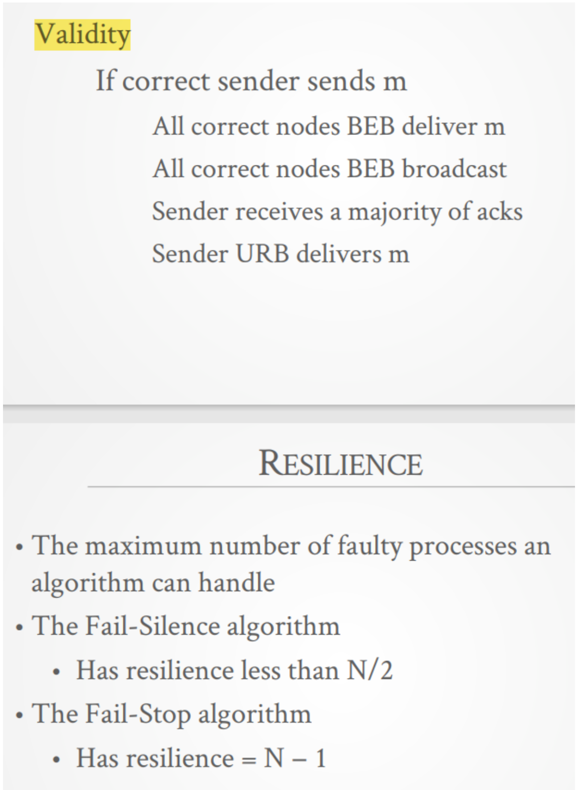
Uniform Reliable Broadcast



Uniform Reliable Broadcast – Fail-Silent

如果我们没有 perfect detector 怎么办?

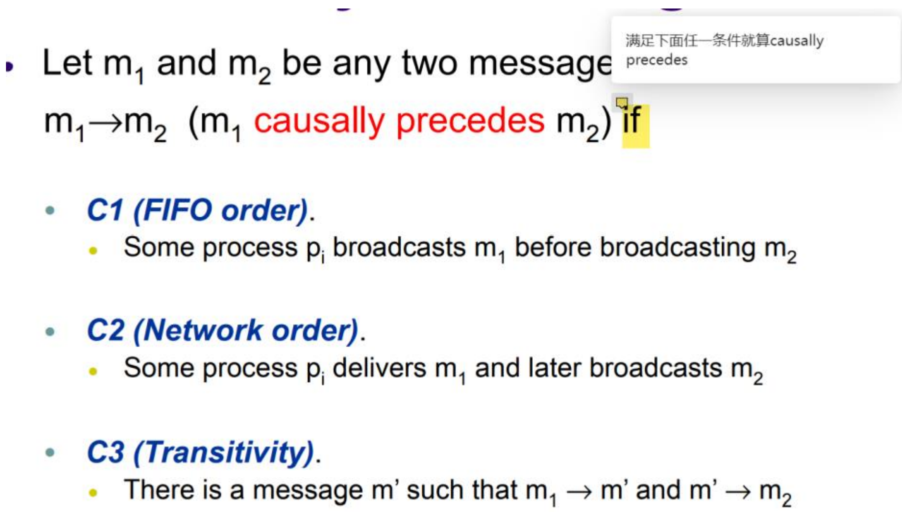
Majority is yyds!



也不是泛指，反正就是要一个 correct node 靠着 perfect link 广播出去了，all correct process 就一定能靠着 perfect kink BEB deliver 到，至于能不能全部 process 成功再广播不重要，只要满足有大多数的 process 是 correct 并成功 pending 并广播了这些信息就行,至于怎么保证大多数 node 是 correct,因为 fail-silent 的假设是大多数 node 正确，所以无需证明

相当于放宽了条件，因为只要满足大多数是 correct 就行，所以我们只要满足大多数 deliver 就行,同时也是意味着能实现 termination,无需无穷无尽地等待。

#6 Causal Broadcast



Implementation

Reuse RB for CB

- Use **reliable broadcast** abstraction to implement **reliable causal broadcast**
- Use **uniform reliable broadcast** abstraction to implement **uniform causal broadcast**

Reliable causal broadcast

- Main idea
  - Each broadcasted message carries a history
  - Before delivery, ensure causality
- First algorithm
  - History is set of all causally preceding messages

S. Haridi, KTHx ID2203.1x 23

Fail-Silent No-Waiting Causal Broadcast

- Each message m carries ordered list of causally preceding messages in **past$_m$**
- Whenever a node rb-Delivers m
  - co-Deliver causally preceding messages in **past$_m$**
  - co-Delivers m
    - Avoid duplicates using **delivered**

Correctness part 看课件

Reliable causal broadcast using FIFO Broadcast

#8 Paxos

Single value consensus properties

Validity:

Any value decided is a value proposed.

Agreement:

No two correct nodes decide differently.

Termination:

Every correct node eventually decides.

Integrity:

A node decides at most once.

Single value uniform consensus properties

Validity(safety):
Only proposed values may be decided.

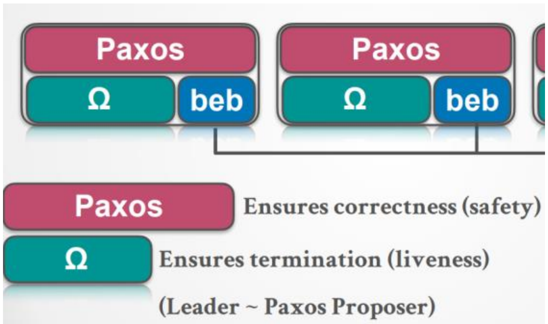
Uniform Agreement(safety):
No two nodes decide differently.

Termination(liveness):
Every correct node eventually decides.

Integrity(safety):
Each process can decide a value at most once.

- This consensus can be solved in Fail-Stop model with strong FD.
- Not solvable in the Fail-Silent model (asynchronous system model).
- Given a fixed set of deterministic processes there is no algorithm that solves consensus in the asynchronous model if one process may crash and stop
(FLP 理论证明了，在异步通信系统中，存在节点失效（即便只有一个），不存在一个可以解决一致性问题的确定性算法)

Abortable Consensus (Paxos)
Elect a single proposer using Ω,
but several processes might initially be proposers(contention).
For safety: might abort if there is contention.
For liveness: Ω ensures eventually 1 proposer succeeds.



Roles
- **Proposers:** will attempt imposing their proposal to set of acceptors.
- **Acceptors:** may accept values issued by proposers.
- **Learners:** will decide depending on acceptors acceptances.

Strawman solution

Centralized solution:
1. Proposer sends value to a central acceptor.
2. Acceptor decides first value it gets.
3. Problem: single point of failure.

Decentralized solution:

1. Proposers talks to set of acceptors.
2. Tolerate failures needs only a majority of acceptors surviving.
3. Proposers might fail (aborts) to impose their proposals.

(If majority of acceptors accept v, then v is chosen, the learners try to decide the chosen value)

P1. An acceptor accepts first proposal it received.

(This can provide the obstruction-free progress – if a single proposer executes without interference, it makes progress; ensure obstruction-free progress and validity. And the validity).

If competition exists, we should enable restarting by distinguish proposals with unique sequence number, ballot number, this number increases monotonically.

P2. If proposal (n, v) is chosen, every higher proposal chosen has value v.

(This can ensure agreement but how to implement it?)

P2a. If v is chosen, every higher proposal accepted has value v.

(Because the acceptor cannot use the knowledge they don't have, they cannot know what has chosen. We make it stricter - we accept only things that have been chosen in the past.)

P2b. If v is chosen, every higher proposal issued has value v.

(We cannot prevent an acceptor from accepting higher value proposal, because the acceptor cannot use the knowledge they don't have as same before, so we make it stricter – only issue that have been chosen.)

P2c. if any proposal (n, v) is issued, there is a majority set S of acceptors such that either

(a) no one in S has accepted any proposal numbered less than n. (nothing is chosen so far so we can pick any value we want)

(b) v is the value of the highest proposal among all proposals less than n accepted by acceptors in S. (something is probably chosen, and I use that as my value)这里是可能决定，因为在 majority 中存在的 highest proposal 可能是被决定的，也可能并不是被决定的。

How to implement P2c?

1. The value of the highest round number.
2. A promise that the state of S does not change until round n. (key of paxos)

---Prepare Phase

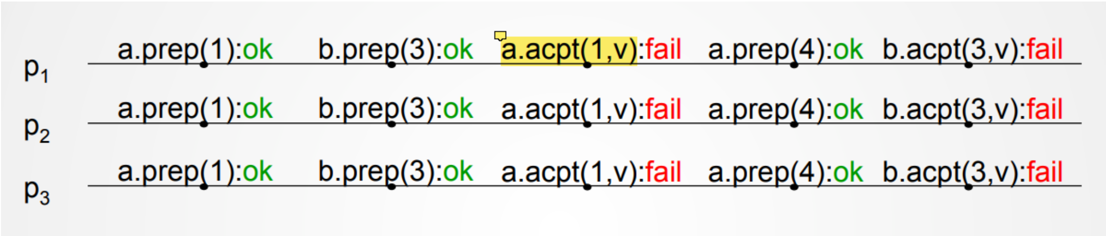
We need a prepare(n) phase before issuing prop (n, v)

-Extract a promise from a majority of acceptors not to accept a proposal less than n.

-Acceptor sends back its highest numbered accepted value.

FLP Ghost

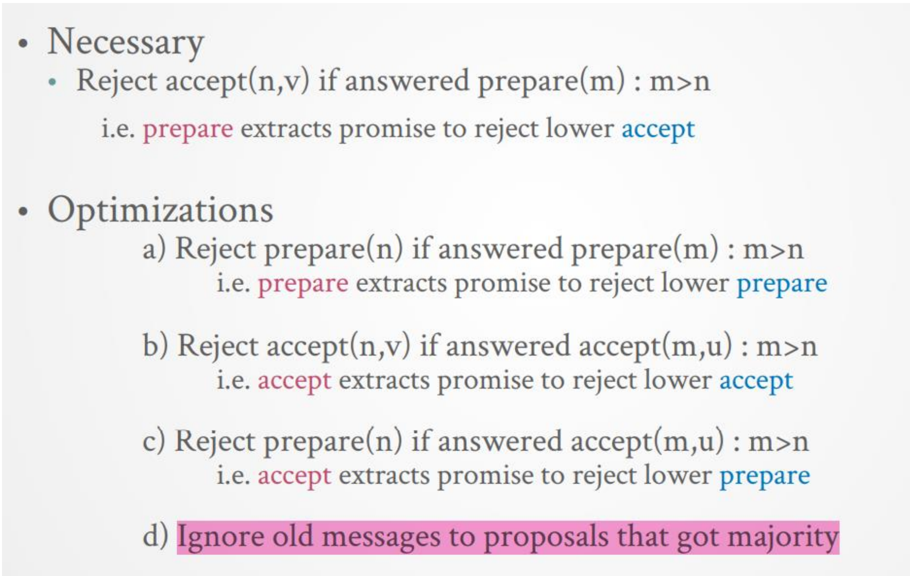
paxos 只保证 something that is chosen and then always will be decided（不论是在有无 stable leader 的情况下）



Proposers a and b forever racing…

- Eventually leader election ensures liveness
- Eventually only one proposer -> termination

Optimizations



State to remember

Each acceptor remembers:

-Highest proposal (n, v) accepted (needed when proposer ask prepare m)

-Highest prepare if has promised (ignore accept(m) with lower number)

Can be saved to stable storage for recovery

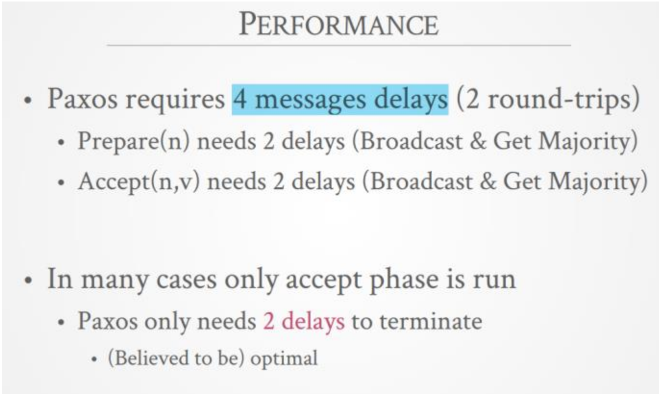
Performance

Paxos requires 2 roundtrips (with no contention)

-Prepare (n): prepare phase (read phase)

-Accept (n, v): accept phase (write phase)

(Improvement: Proposer skips the accept phase if a majority of acceptors return the same value v) 大多数人都告诉你他们的选择了，没必要再考虑其他人了！



#9 Replicated Logs and State Machines

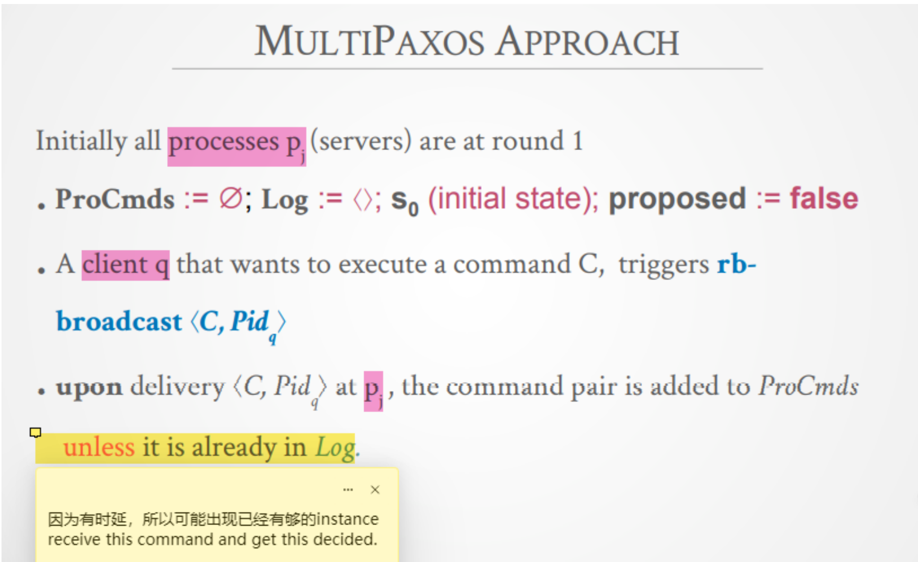
State Machine

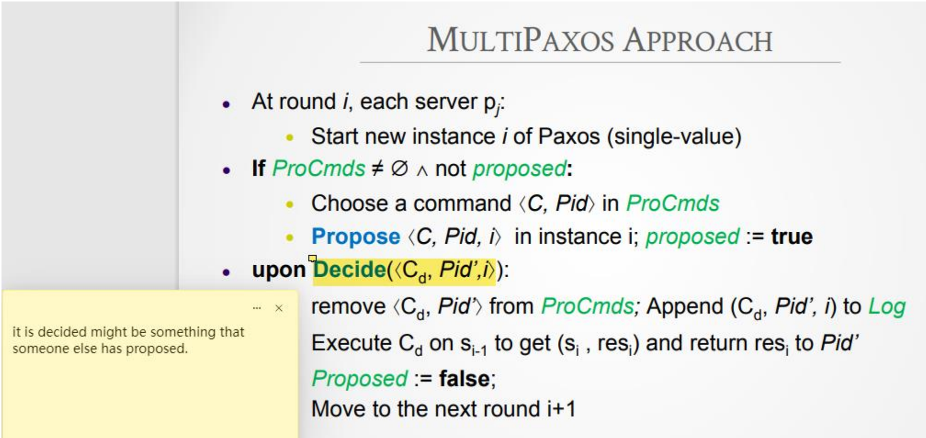
- Executes a sequence of command
- Transforms its state and may produce some output
- Commands are deterministic.
- Outputs of the state machine are solely determined by the initial state and by the sequence of commands

Replicated log

- Ensures state machines execute same commands in same order.
- Consensus guarantees agreement on command sequence in the replicated log

Multi-Paxos





Multi-Paxos can be a mess

1. In order to select a command at round I any process (learner) have to agree on the sequence of commands C1..Ci-1, we need to use Paxos every round takes 4 communication steps.
2. Not easy to pipeline proposals: holes in the log might arise.

Sequence Consensus

We not only agree on each command, but also the sequence of commands.
(Can decide again if old decided sequence is a prefix of the new one)

Sequence Consensus Properties