Distributed Systems Group
Department Software Technology
Faculty EEMCS
DELFT UNIVERSITY OF TECHNOLOGY

Distributed Algorithms (IN4150)

List of algorithms (2018-2019)

January 3, 2019

Below is a list of the algorithms in the order of treatment in the course, with the names of the original authors, if applicable.

Chapter 3: Synchronization

1. Alpha-, beta-, and gamma-synchronizer: Awerbuch
2. Causal message ordering (broadcast): Birman-Schiper-Stephenson
3. Causal message ordering (point-to-point): Schiper-Eggli-Sandoz
4. Total message ordering
5. Determining global states: Chandy-Lamport
6. Termination detection in a unidirectional ring
7. Termination detection in a general network
8. Deadlock detection for AND requests: Chandy-Misra-Haas
9. Deadlock detection for OR requests: Chandy-Misra-Haas
10. Deadlock detection for M-out-of-N requests (with/without instantaneous communication): Bracha-Toueg

Chapter 4: Coordination

1. Assertion-based mutual exclusion: Lamport
2. Assertion-based mutual exclusion: Ricart-Agrawala
3. Assertion-based mutual exclusion: Maekawa
4. Generalized assertion-based mutual exclusion

5. Token-based mutual exclusion: Suzuki-Kasami
6. Token-based mutual exclusion: Singhal
7. Token-based mutual exclusion (tree-based): Raymond
8. Detection of loss and regeneration of a token
9. Election in a synchronous unidirectional ring (non-comparison-based)
10. Election in a bidirectional ring: Hirschberg-Sinclair
11. Election in a bidirectional ring (enhanced version)
12. Election in a unidirectional ring: Chang-Roberts
13. Election in a unidirectional ring: Peterson
14. Election in a synchronous complete network: Afek-Gafni
15. Election in an asynchronous complete network: Afek-Gafni
16. Minimum-weight spanning trees: Gallager-Humblet-Spira

Chapter 5: Fault Tolerance

1. Agreement with stopping failures
2. Byzantine agreement with oral messages: Lamport-Pease-Shostak
3. Byzantine agreement with authentication: Lamport-Pease-Shostak
4. Single-value Paxos consensus: Paxos
5. Practical Byzantine Fault Tolerance: Castro and Liskov
6. Zyzzyva: Kotla et al.
7. Randomized agreement with crash failures: Ben-Or
8. Randomized Byzantine agreement: Ben-Or
9. Stabilizing mutual exclusion: Dijkstra
10. Stabilizing stop-and-wait datalink algorithm
11. Non-stabilizing and stabilizing sliding-window datalink algorithms

Chapter 3 Synchronization

Synchronous System v.s. Asynchronous System

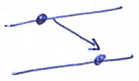
global common clock
bounded msg delay

x clock
finite & unbounded delay

Happen-Before Relation. →

1) local order

2) message exchange



send → receive

3) transitivity

$a \to b$. $b \to c$ $a \to c$

Concurrency Relation

all to

neither $a \to b$. $b \to a$.

not transitive

Logical clock.

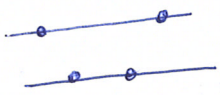
assign timestamp to events $C: E \to S$.
characterize HB relation.

Definition

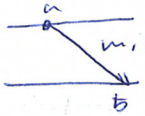
Implement: 1) **Scalar clock. / Lamport clock**

integer counter

Defin. & rules { internal c_i++.
send & receive



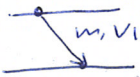
Problem..?
concurrent processes do not have equal clock value)



$b = c_j = \max(C(a)+1, c_j+1)$

2) **Vector clock**
vector.

rules { internal
send & receive



$\max(V_j + e_j, V(a))$

definition

meaning.

achieve concurrent?

characterize HB relation.

Problem in Synchronization.

- time
- different time granularity
- achieved by msg

Synchronous Communication vs. Asynchronous communication

occur simultaneously
a single msg transfer event
sending blocked.

separate

sending do not block
receiving block/interrupted.

Synchronizer

Simulation of SS on AS

Synchronizer

purpose: run SA on asynchronous system

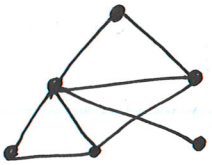
problem: when issue clock pulse?

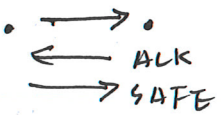
idea: sender figure out all msg received.
notify receiver.

α - Synchronizer

idea: ALK.
safe node?
when to proceed?

complexity: communication $O(N^2)$
time $O(1)$

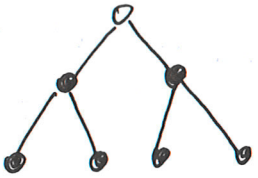


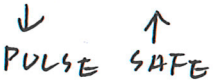


β - Synchronizer

idea: spanning tree
PULSE. SAFE

complexity: communication $O(N)$
time $O(N)$

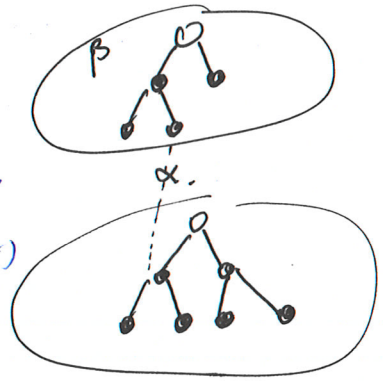




γ - Synchronizer

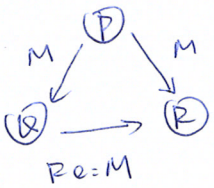
idea: initialization: preferred-link
PULSEs and SAFEs
CLUSTER-SAFEs and READYs

complexity: communication $O(E)$
time $O(H)$



Message Ordering

Message Order Violation



Problem

requirement on the order of msg received

e.g. see reply later than origin

replicated database (process update)

Message Destination types

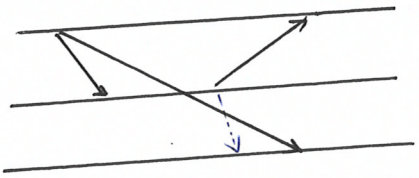
Dest(m)

1) point-to-point $|Dest(m)| = 1$
2) multicast > 1
3) broadcast # processes

Causal message Ordering

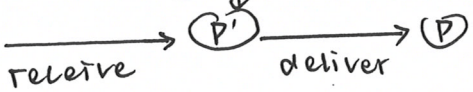
$m(m_1) \rightarrow m(m_2)$

$di(m_1) \rightarrow di(m_2)$



Implement of MO

buffer



Causal Ordering: Broadcast

NO FIFO Required

Idea
- vector clock.
- message = content + timestamp
- number broadcast msg.
- receiving process → can't deliver. : buffer
  → deliver this msg & check buffer

$D_j(m) = V + e_j \geq V_m$

Implement
- broadcast
- receive msg
- deliver msg

$V = V + e_j$

Causal Ordering: point-to-point

NO FIFO Required

Idea
- vector clock.
- local buffer $S = \{(P_j, V_j) \dots\}$
- message = content + local buffer + timestamp
- deliver condition $\begin{cases} no(i,v') \\ exist(i,v')\ v' \leq v \end{cases}$ → delivered. → buffer merge sm&s.

Implement
- send: ① send msg ② update local buffer
- receive: check deliver → ① deliver msg ② check pending buffer; → ③ pend
- deliver: merge local buffer

Total message ordering

all processes receive in same order

Simple solution

sequencer
- number msg
- broadcast
- record history

do broadcast → send to sequencer

Sophisticate algorithm (FIFO required)

scalar clock (with process ID as tie breaker)

ordered message queue

ACK msg

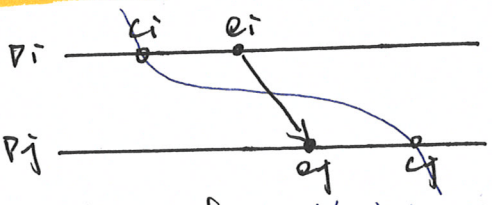
deliver condition
- head of queue
- all ACK received

Global State

Problem: global state in asynchronous system

Application: ① debug ② detect e.g. bank.

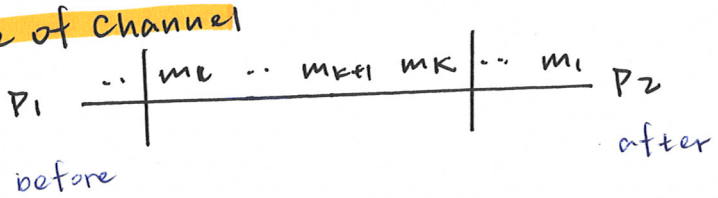
Consistent cut: internal. one → every



vector time of c_i: $V(c)[i] = \max\{V(c_1)[i], \dots V(c_n)[i]\}$

$V(c) = (V(c_1)[1], V(c_2)[2], \dots V(c_n)[n])$

State of Channel



Chandy's and Lamport's Algo (FIFO)

Idea: record state → ① record own ② marker send

receive marker { first: ① state channel ① record own ② marker back ④ create buffer

later: ① state channel

Termination Detection

Problem: distributed computation terminate

active → passive

passive → active

Termination Detection in unidirectional ring (FIFO)

Idea:

Naive solution: P_0: special

token = knowledge

receive / passive

token \ active

Problem: reactivate

Colored solution: introduce color in processes and token

process → turn black: msg sport passiveness

→ turn white: forward black token

token — initial white

— turn black: arrive black

Final decision → token → black: ✗

→ token = white + P_0 passive

General Termination Detection.

Idea: special P

weights for processes and msgs

Initially, $w(P)=1$ $w(P_i)=0$.

then $w(P_i)=\frac{1}{n}$

if want → send msg: $w(P_r)=\frac{w_i}{2}$.

$w(msg)=\frac{w_m}{2}+w_i$

receive msg: $w(P_i)=\frac{w_i}{2}+w_m$

terminates: $w(P_i)=0$ ~~$w(P)=$~~

$w(msg)=w(P_r)$

Final decision: $w(P)=1$

Deadlock Detection.

Problem: each request, wait

Wait-for-Graph: (WFG) nodes
links

Resource Model: resource RA
process request
resource granted
release resource
Condition: cycle

Communication Model: block and wait, from dependent set (1 msg)
→ active. receive single msg.
Condition-Knot { a path
no edge in → out.

Deadlock detection for AND request

Idea: suspect deadlock → send PROBE
propogate.
Condition: receive own PROBE
Pi dependent. Pj: path.
Condition: dep(i) = true.

Deadlock detection for OR request

Idea: suspect deadlock → send QUERY
→ propogate → construct tree
dependent set

receive QUERY ⇒ 1st engaged.
later send REPLY back
all REPLY received

condition: every QUERY → REPLY

Deadlock detection for N-out-of-M Requests

processes — active
blocked.

msg: REQUEST, REPLY, RELINQUISH

With instantaneous communication

Idea: OUT. IN → OUT=IN

2 phase { ① notify ② simulate.

① notify() → spanning tree

if no request pending → free=true. grant()

ends receiving DONE

② GRANT simulate REPLY

sufficient GRANT → active

condition: initiator. free=false

With messages in transit

Idea: color represent link. { grey, black, white, translucent

(optimistic) grey, white, translucent disapply → apply algo.

grey. white edges resource granted

active : nv ≤ # grey white

know color of edge → send color msg to IN&OUT set with

Chapter 4 Coordination

Mutual Exclusion

Problem: grant exclusive privilege

Algo { token-based: possess = execute CS
assertion-based: request permission from all/part

reply → conclude only one

Lamport's mutual-exclusion algorithms: Assertion-based (FIFO)

Idea: msg = scalar clock + pid. ordered request queue.

want CS: → REQUEST to all (+itself).

receive REQUEST → ① enter queue.
② send REPLY

enter CS if ① all REPLY ② head of queue.

leave CS: send RELEASE to all
↑ remove request

Complexity $3(n-1)$.

Optimization: suppress unnecessary REPLY.
× needed if older REQUEST.

Assertion based: Ricart's and Agrawala's

Idea: combine REPLY and RELEASE RELEASE not needed.

REPLY { immediately request higher > own
defer

enter CS: all REPLY.

leave CS: send deferred REPLY.

Complexity $2(n-1)$.

Optimization reduce request set.

Maekawa's Assertion-based algo:

Requireset Properties
1) any two non-empty intersection (must)
2) every process in own
3) request size same size
4) every process in same number request set

Lemma minimum size of request set is $O(\sqrt{n})$

Idea : request set.

want CS? → send REQUEST to R.

enter CS : receive GRANT from R.

leave CS : send RELEASE to R

Handling deadlock

granted + receive REQUEST → compare timestamp

if new ~~older~~ later : ① enter queue ② send POSTPONE

if old ~~older~~ later : send INQUIRE to old.

receive INQUIRE: wait until

/ receive GRANT from all : ① complete CS ② send RELEASE

\ receive one POSTPONE : ① send RELINQUISH

receive RELINQUISH : ① enqueue request

② send GRANT to oldest

A generalized mutual-exclusion algorithm

Token based: Suzuki's and Kasami's

Idea: token = permission to CS.

want CS: ① number request.

② send REQUEST to all (+ itself).

Process maintain array N: for every p last request knows

TN: .... last request granted.

Token

compare N & TN.: forward token

avoid starvation: check own index

leave CS: update TN.

send token to first N[i] > TN[i]

else wait

Variation (no starvation): In token maintain queue = pId should receive token.

leave CS: ① add Queue

② remove itself

③ send token to head.

Optimization: reduce REQUEST

Token based: Singhal's

Optimization: send REQUEST to may have token.

Idea: Process state { R requesting. E executing CS H holding token O other

Process maintain arry N: request number

S = states

Token: TN, TS.

S initialization:

P_0: $S[0]=H$ $S[j]=O$ ...

P_i if $S[j]=R$. $(j<i)$ $S[j]=O$. $(j>i)$

want CS: ① set own requesting ② send REQUEST to R.

receive REQUEST: ① update request number.

② if local state

- E, O: record requesting
- R: send REQUEST
- H: send token.

receive token: update TN. N. S. TS.

Token based: Raymond let a single msg represent all request (FIFO)

Idea assume spanning tree is created.

root = hold token. change ↻.

every process maintains holder ─ own id
                          └ neighbor id on the path to root

request queue ─ own id
              └ neighbor id.

want CS: ① append own id in queue ② send REQUEST in direction root

receive REQUEST: ① append neighbor id

② send REQUEST to parent

if is root ① send token to head. ② dequeue

③ reassign parent

receive token: if is head: enter CS

else: ① propagate to head ② send REQUEST if queue not empty

dequeue.

assign parent

Election

Problem: single process get privilege
elect with largest id

Anonymous Network ~~no~~ no id

Election in a synchronous unidirectional ring (non-comparison based)

Idea: elect minimum
ring size = n. known to all.
In round i, inspect id = i. → send id to neighbor.
relay.
n, no msg → doesn't exist!

In $n+1$ $= 2$.
$(k-1)n+1$ k.

Complexity $O(n)$

Election in bidirectional ring: Solution 1

Idea: check if largest id in ever larger segment (size $= 2^k + 1$)
In round k, check 2^{k-1} both side.
PROBE msg = id + hop counter
receive PROBE: if own larger: discard.
else : forward. hop --.
if hop count = 0. send OK.

initiate next round if receive 2. OK.
Elect Condition: if not known size = 2OK from wrong side.
/: process receive 2 PROBE.

Election in bidirectional ring: Solution 2

Idea. neighbour bounded by active processes.
In round i, exchange msgs.
If own = largest → active.
else passive

Subsequent phase: execute in virtual ring
Elect Condition: receive own id

Election in unidirectional ring: Chang Robert

Idea: send msg with id to neighbour.

compare = elected.

\ own larger. discard

\ smaller. relay

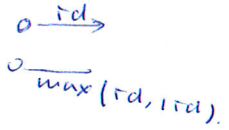
Election in unidirectional ring: Peterson

Idea simulation of bi sol 2.

send id to right. maxpid, received from left → right

if 1st = largest → active

else → passive



Elect condition: receive own id.

Election in complete network: Afek and Gafni (synchronous)

Idea send id to ever larger set and wait for ACK.

candidate process

if receive larger: ① send ACK ② adopt id.

if receive smaller msg: killed.

size is twice

Elect condition: receive ACK from all

node consist { candidate process = level
ordinary process: ① receive ② order. ③ send ACK.

msg = level + id

elected = first start

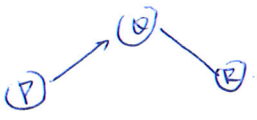
Election in asynchronous network: Afek and Gafni

Idea no make sense: wait for all ACK.

react if receive single msg

msg = level + id level = captured process number.

① P capture P (owned by R) R also needed captured.



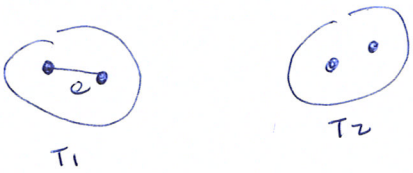
node keep links { father
potential father

In asynchronous system, previous owner might be killed.

Minimum spanning Tree

Application: broadcast in minimum cost

Lemma1: A weighted connected graph in which all weights are different has a unique MST.



some edges in T_1 ~~or~~ but not T_2.

e: lowest weight.

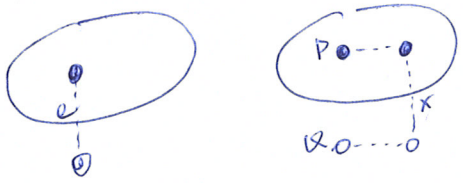
add e to T_2 → cycle

(∵ e' in T_2 not in T_1)

w(e') > w(e)

throw e'. → lower weight

Lemma2: F is fragment and e is MOE, F with e and node incident with e



Suppose not

final MST must have a P→R.

∴ at least one edge outgoing edge

∵ MOE w(e) < w(x)

fragment. subtree

outgoing edge

MOE (minimum-weight outgoing edge)

Idea: build from fragment, start with single node.

Fragment has level. 1 node = level 0.

core, name

nodes cooperate find MOE. fragment connect along MOE

merge: new fragment.

Absorb same level, core

L > L' postpone.

any start spontaneously. receive → start

Byzantine Agreement with authentication

Idea: msg carry signature. ×forged
loyal could verify
msg = initial value + signature $(V_0 : S_0 : \cdots S_{i_k})$
lieutenant wait until length = f+1.
maintain. set V = received order.
execute same choice function

Randomized ~~Byzantine~~ Agreement with crash failure

Idea: every start with v. $f < \frac{n}{2}$

rounds consist 3 phases

{ notification phase : ① broadcast NOTIFICATION. ② wait for $(n-f)$ M.
proposal : ① if enough notification for one. broadcast it
$> \frac{n}{2}$ else broadcast ~~random~~?
decision if decided : stop
else wait $(n-f)$ P.

① if ≥ 1 P. ~~de~~adopt value
$\geq f$ P. decide that
else new round with random v.

Randomized Byzantine Agreement

Idea: every start with v. $n > 5f$.

notification phase ① broadcast. ② wait $(n-f)$ M.

proposal phase if enough support $> \frac{n+f}{2}$: propose that.
?
else
if decide : stop
else wait $(n-f)$ P.

decision phase if $> f$ Propose : adopt value
$> 3f$: decide

else. new round with random

Chpter 5 Fault Tolerance

Consensus: agree on same value

Application: 1) commit in distributed DB.

2) modify reword in DB with replica

Fault classification { fail-stop, omission, performance, Byzantine

Byzantine Agreement

Problem: Goal: non-faulty processes agree on same

source/commander start

lieutenant decide

Agreement = agree on same

Validity = source is non-faulty, decide initial

Termination = decide with finite time

Agreement with stopping failure

Idea: f=

Every process start with v.

maintain set W. $W=\{v\}$ initially

f+1 round: ① broadcast ② receive W_j.

$W = W \cup W_j$

Decide. W single element. → decide

→ default

Byzantine Agreement with Oral Messages

Idea: f+1 levels.

Bottom case. OM(0) f=0.

① commander broadcast.

② decide that value.

OM(f) f>0.

① command broadcast

② each as commander. execute OM(f-1)

Lieutent decide on majority {commander, lieutenant commander}

Stabilization

Problem: ~~transparent~~ fault
transient.
bring incorrect → correct state

Stabilizing Mutual-Exclusion in unidirectional ring.

Legal state $\{i \mid in\_CS_i = 1\} \leq 1$

Idea
In a shared memory model: inspect own & predecessor
composite atomicity: R&W single step
Pi maintain single integer modulo K.
Ordinary process: ① compare
② unequal. → copy

Special process: ① compare
② equal → increment by 1

legal: K, K ... K, K-1, K-1 ....

Stabilizing stop-and-wait datalink algorithm

Idea
Sender & receiver maintain counter
Send msg → receive Ack then send next.
timeout: resend.

A non-stabilizing datalink algorithm

Idea
Sender maintain { w, ns, na.

receiver maintain nr.
msg = msg number.
receive msg: ① nr ++ ② send Ack.
receive Ack: ① if Ack higher = set na.
timeout: resend unacked

predicate $(na \leq nr.) \;\&\; (nr \leq ns) \;\&\; (ns \leq na + w)$
$\&$
$(msg, i) \quad i < ns$
$\&$
$(ack, i) \quad i < nr.$

A stabilizing datalink algorithm

Idea: msg carry 2 integer. 2nd=na. → give catch up with sender.

predicate

↑snr. ↑rns

State Machine Replication

Problem: execute client request in same order

Single Value Paxo

Idea: Goal: decide a value

proposer. acceptor. learner.

Algo proceed in round. int

round num assign to proposer.

propers ignore msg with smaller round.

phase1: build majority acceptor ← propose.

receive: if newest, reply latest vote & round number

phase2: propose latest vote to majority
(if not voted, random)

acceptor vote positive / abstain to learner.

~~Part~~. Practical Byzantine Fault Tolerance

View: primary backup. view num

client send request to primary

primary assign number, multicast to backup.

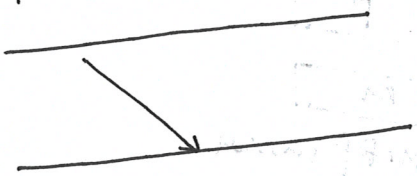
replica execute & reply to client

client wait for F+1 same result

if no. client send all req to replica.

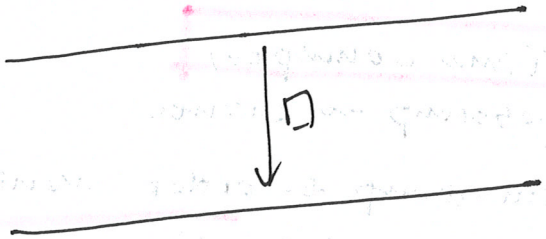
17/11/15 02 Modeling distributed system

Asynchronous communication (AC) / Non-blocking course



just send, doesn't check whether receive it / wait

Synchronous communication (SC)



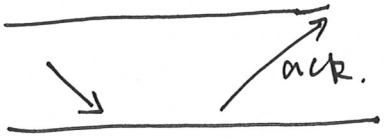
send message and wait until receive it. process receiving block

no message in transit

e.g. ~~email~~ tele

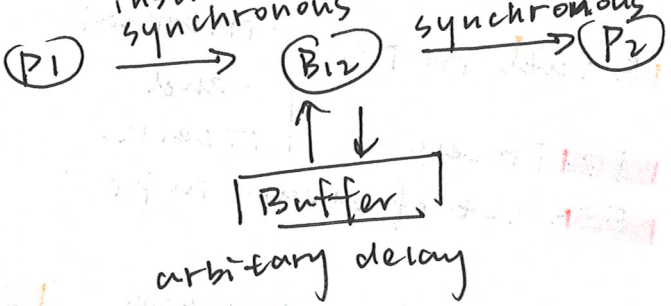
simulate

AC → SC



SC → AC

instantaneous synchronous



arbitary delay

Link Properties

FIFO. receive message in order they sent

Buffer (un)bounded. store all mess in transit

Mess delay finite finite ∧ unbounded receiver don't receive for a long time, he is never be sure whether mes will arrive

Mess delay bounded if process expects msg, after time expire it will sure that it wonce receive it in future } synchronous

Msg may not lost

Msg may not be damaged.

Asynchronous System

no common clock

finite ∧ unbounded msg delay

event-driven. (internal / reception event)

upon receipt of msg — when condition.

Synchronous System

a common clock

bounded msg delay

time-driven

round { receive, do, send }

(in reality x exist)

Simulation

① Local simulation

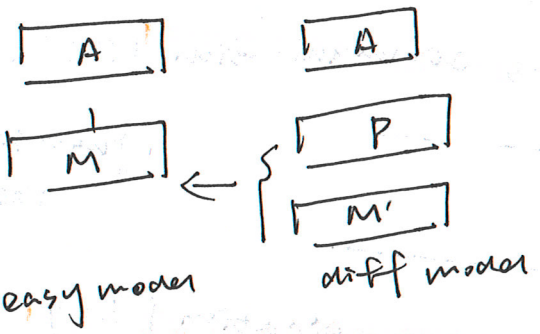
[M', P] looks like M

^ from outside differ

② Global simulation.

to outside observer

there is no difference



Chapter 3 Synchronization. (Time Concepts)

Time is needed to — assign timestamp to events

— compare timestamp to order events

(logical time > physical)

in AS, synchronization has to be achieved through msg.

Events in DS { internal, send, receive

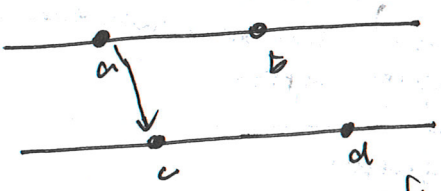
Pi: Process

Ei: set of events in Pi

Happen-Before Relation / Causality Relation

local order: if a, b in Ei, a occurs before b.

then $a \to b$



$a \to b$ ①

$c \to d$ ①

$a \to c$ ②

$a \to d$ ③

message exchange: if a in Ei is sending ~~msg~~ msg.

b in Pj is receiving msg.

then $a \to b$

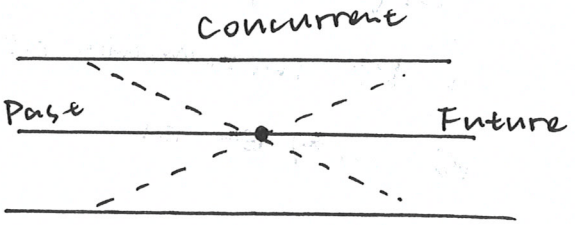
transitivity: if $a \to b$, $b \to c$

then $a \to c$

Concurrency: if neither $a \to b$ nor $b \to a$
then a and b are concurrent ($a \| b$)

Casual past (history) of event a.

$P(a) = \{ b \text{ in } E \mid b \to a \}$



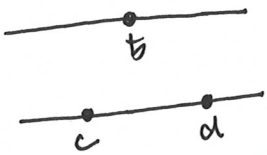
Events concurrent with a:

$C(a) = \{ b \text{ in } E \mid a \| b \}$

Casual Future of an event a:

$F(a) = \{ b \text{ in } E \mid a \to b \}$

concurrency isn't transitive



$b \| c$. $b \| d$

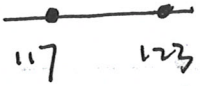
$\not\Rightarrow c \| d$

Logical Clocks

S: partially ordered set (e.g. natural numbers)

logical clock: $C: E \to S$

if $a \to b$. $C(a) < C(b)$



converse may not hold!

logical clock characterize HB:

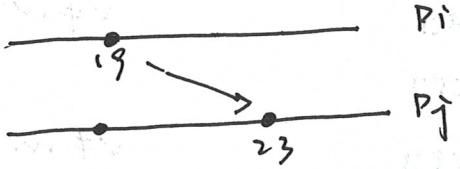
$C(a) < C(b)$ iff $a \to b$

Conditions for logical clock

1) single P. a, b in E_i. if $a \to b$. then $C_i(a) < C_i(b)$

2) transitivity ✓

3) msg ex.



how receiving later than sending?

Scalar Clock Implementation

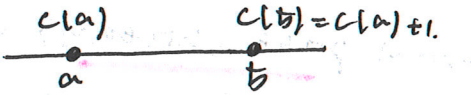
1) a in Ei. a isn't msg receive event.

~~then~~

Ci (integer counter in Pi)

① Pi increments ci

② C(a) ← ci

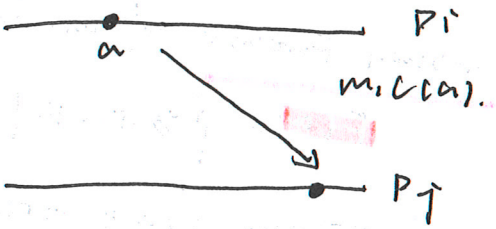


2) for send event

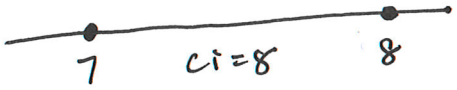
① Pi send C(a) along with m

② Pj assign C(j) max(Cj+1, C(a)+1)

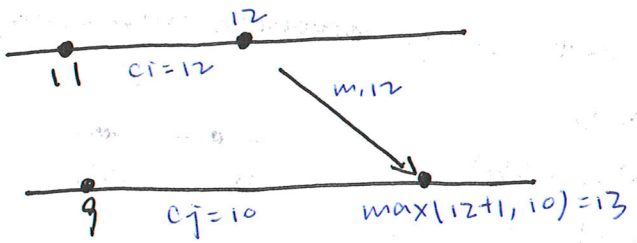
③ Pj assign C(b) ← C(j)



e.g. 1)

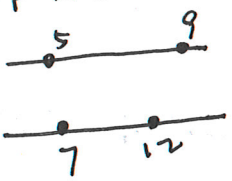


2)



Problem: X characterize HB relation.

e.g. 2P x communicate



5<7

but a ~~<~~ b

Vector Clock

Comparison of 2 vector v, w of length k

v=w iff v[i]=w[i] i=1,2,... k

v≤w iff v[i]≤w[i]

v<w iff v[i]<w[i] v≠w.

v≥w iff v[i]≥w[i]

v>w iff v[i]≥w[i] v≠w.

max(v,w)[i] = max(v[i], w[i])

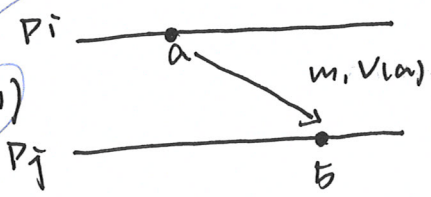
Vector Clock Implementation

Initial: $V_i = (0, 0 \cdots 0)$

1) if a in E_i. a is internal event.
   - ① P_i increment $V_i[i]$ +1
   - ② $V(a) \leftarrow V_i$

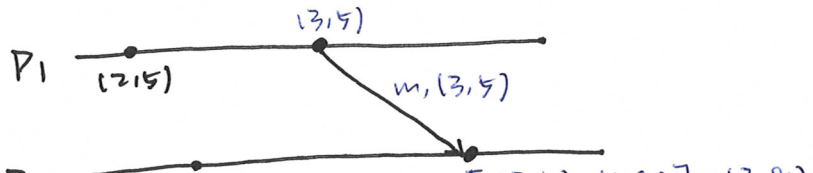
2) if a is sending event in P_i
   b receiving event in P_j
   - ① P_i send $V(a)$ along with m
   - ② P_j assign V_j max(V_j to e_j, $V(a)$)
   - ③ P_j sets $V(b) \leftarrow V_j$

double change!



e.g. 1)

2)



$V_2 \leftarrow V_2 + e_2$
$\parallel$
(1,8)

↑ most up-to-date knowledge for P_1

Meaning of Vector Clock

1) $V_i[j] = \begin{cases} \text{local event} & i = j \\ \text{the last event in } P_j \text{ that } P_i \text{ knows} & i \neq j \end{cases}$
   either directly or indirectly

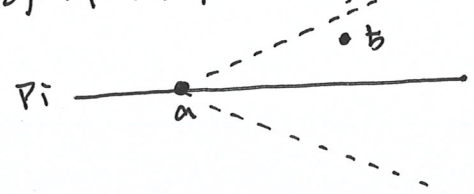
2) a || b are concurrent
   if $V(a)[i] > V(b)[i]$
   $V(a)[j] < V(b)[j]$

Vector Clock characterize HB

$V[a] < V(b)$ iff $a \rightarrow b$

1) if $a \rightarrow b$. then $V(a) < V(b)$

2) if $V(a) < V(b)$:



< a, b in Pi. $V_a[i] < V(b)[i]$
  a in Pi. b in Pj
  b has at least same knowledge about Pi as at event a

Theorem:

For k-dimensional vector clock to characterize HB relation, with n processes, we need $k \geq n$.

Proof for $n=2$.

Suppose $k=1$. Take 2 p don't communicate.

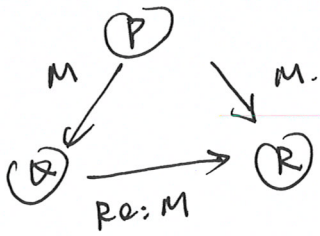
x characterize ~~it~~ interval events.

$\therefore k \geq 2$.

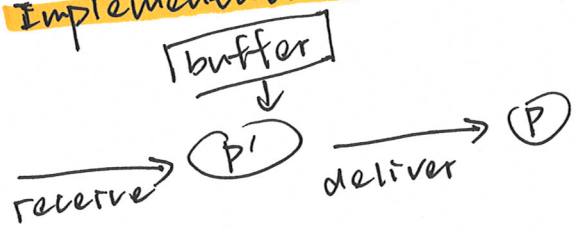
Proof for $n>2$:

Chapter 3 Message Ordering

use clock to enforce ordering of msg.



Implementation



incoming msg → buffer

P' check if it can be delivered

- ✓ deliver
- ✗ postpone delivery.

Dest(m): set of destinations

point-to-point: $|Dest(m)| = 1$

Multicast: $|Dest(m)| > 1$

Broadcast: $|Dest(m)| = \#$ process

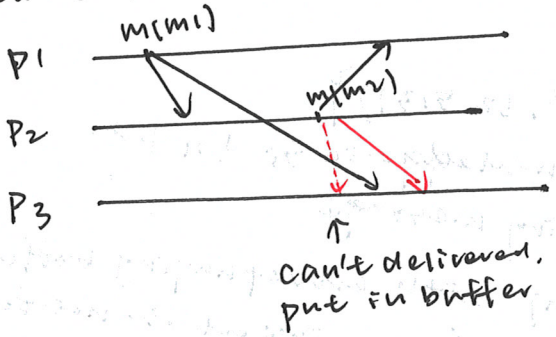
m(m): event of multicasting a msg m.

di(m): event of delivering a msg m to Pi

Casual Message Ordering

if $m(m_1) \rightarrow m(m_2)$

then $di(m_1) \rightarrow di(m_2)$ for all i in $Dest(m_1)$ and $Dest(m_2)$



Birman-Schiper-Stephenson Algorithm (Broadcast msg)

Idea:

1) receive broadcast msg from same process, inspect number check if miss intermediate msg.

2) msg contains knowledge of process at the point of sending
   - check whether missing msg from other processes

Implementation:

use vector clock.

send m along with Vm (local vector clock)

condition for delivering: $P_i \xrightarrow{m} P_j$

$D_j(m) = (V_j[i] = V_m[i] - 1) \wedge (V_j[k] \geq V_m[k])$ for $k \neq i$

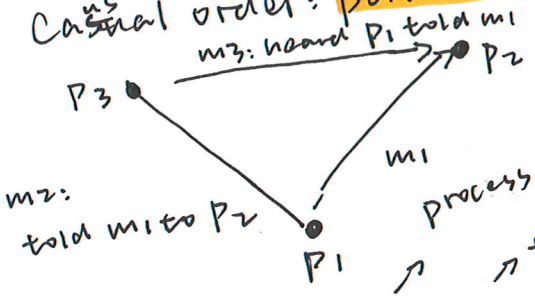
P_i was at least as up to date as P_j when it sent message

Correctness

1) same p.

2) diff p.

Causal order: point-to-point : Schiper-Eggli-Sandoz



buffer $S = \{(P_j, V_j); \ldots\}$

(local buffer: knowledge about all other process (process at least should be that point)

send along with msg: (use for yourself)

update knowledge for receiving process

eg. in P_1: $S_1 = \{(2,[3,1,1,4]); (3,[2,5,1,3])\}$

after sending msg, replace knowledge in S_i for P_j.

by the timestamp of sending message

Condition for delivering msg: $D_i(m)$ with accompanying buffer S_m in P_i

$(m, S_m) \rightarrow (P_i)$

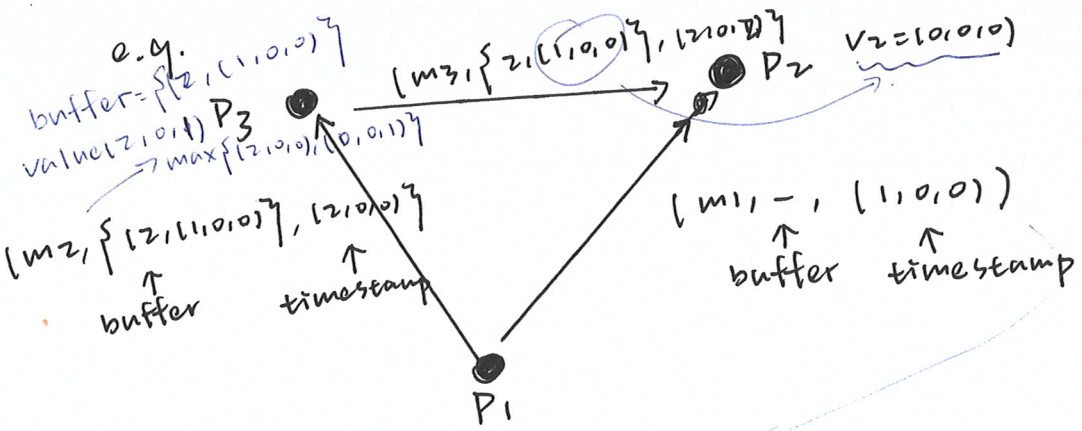
① (i, V) x exist in S_m inspect else not for him

i.e. sending process doesn't have any clue at all

the receiving processing should be

② (i, V) exist in S_m and $V \leq V_i$

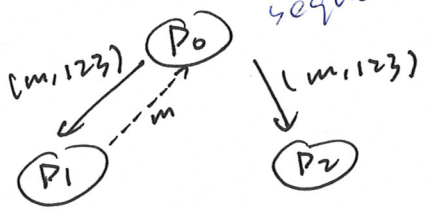
i.e. at least has up-to-date knowledge for all components



Total Message Ordering

All processes receive all messages in the same order.

(don't care about order. not necessarily causal order)



if one p want to broadcast.

① send msg to P_0

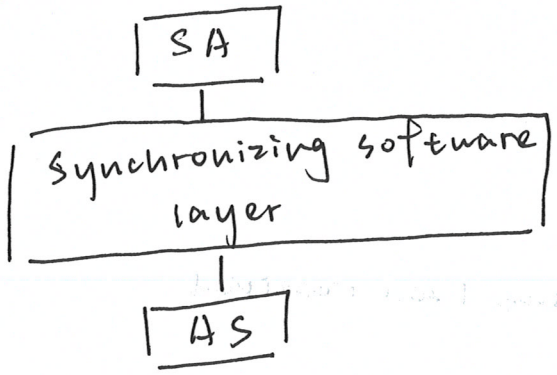
② P_0 numbers all msg sequentially broadcast to all processes

③ P_0 keep a history

17/11/20

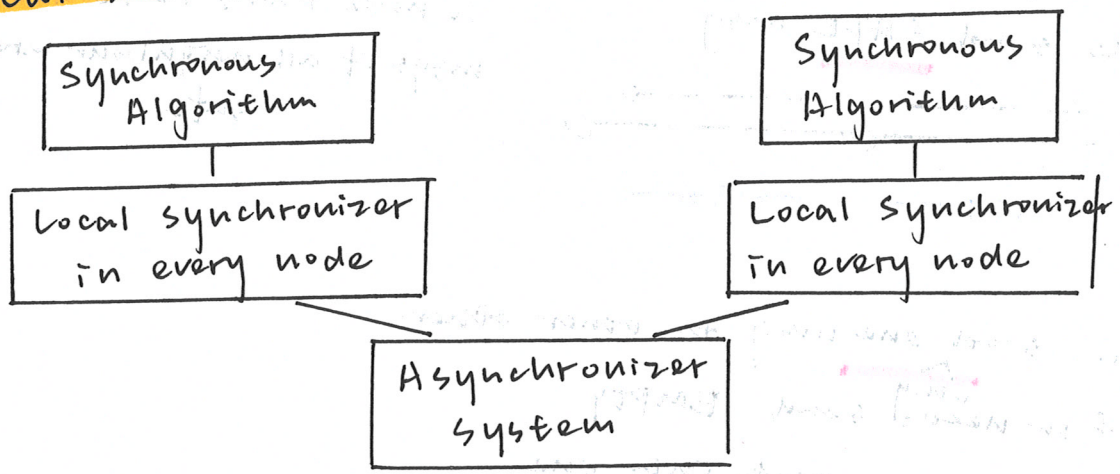
Synchronous systems more powerful than asynchronous system

Global simulation



outsider see a synchronized system

Local simulation



every process locally is a synchronous system
outsider see an asynchronous system

Synchronizer: make AS looks like SS

Problem of AS: no idea when msg arrive (finite ^ unbounded)

Basic Idea: processes run in rounds
p receive all msg it should receive in current round
then a clock pulse

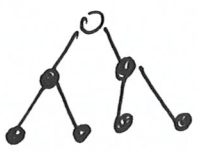
Types of synchronizer

α - Synchronizers

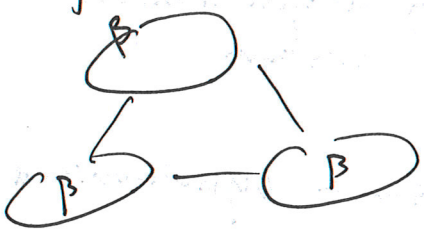
nodes have point-to-point connections (not necessarily complete)

synchronizer take all msg (n^2).
communicatin - inefficient
time - efficient

β - Synchronizers ⟩ cut msg
create spinning tree
communication - efficient
time - inefficient (msg travel ↑↓)



γ - Synchronizer



split network into cluster

intra-cluster: β | spining tree

inter-cluster: α

Generating a pulse

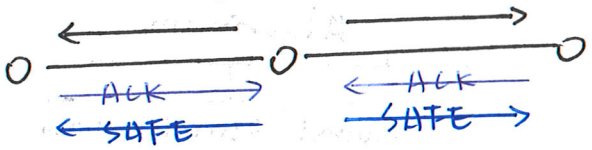
x wait for all msgs

(-) acknowledge

node is safe: all msgs it sent has been received

detect by: reception of ACK.

safe node send SAFE msg



a node knows it receives all msgs if all neighbour are safe

(=) tricky

every node send one (only) msg to other other.

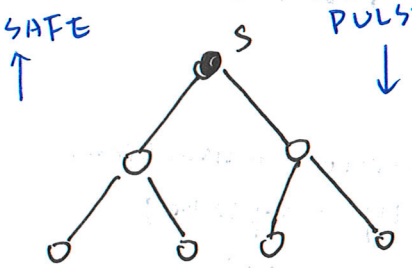
- if no need: send EMPTY
- if multiple: pack into one

a node should in every pulse receives single msg from neighbor

β - Synchronizers Impl

Initial: ① elect a leader

② creat spanning tree



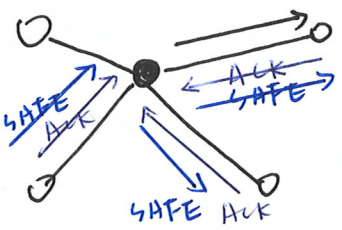
- when a node finds itself & all descendants safe
  - if sends SAFE to its parent
- when root receives SAFE, send PULSE down
- when node receives PULSE, it can go to next round

Communication: $C(\beta) = O(|V|)$

Time: $T(\beta) = O(|V|)$ ~~bad~~ (depends on depth of trees)

balanced tree: $O(\log n)$

α-Synchronizer Impl



A node:

find itself safe if receives all Ack

send SAFE too all neighbors

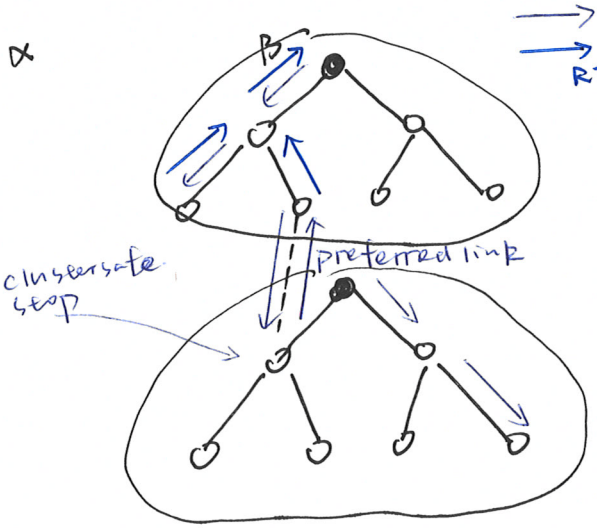
receive SAFE from neighbors

generate local pulse

Complexity: communication: $C(\alpha) = O(|V|^2)$

time: $T(\alpha) = O(1)$ [3 steps = constant times]

γ(gamma)-Synchronizer Impl



Initial:

① partition nodes into clusters

② select leader in cluster

③ create spanning tree in cluster

④ create preferred link between clusters

β-synchronizer: (1st wave)

① every node broadcast PULSE down

② convergecast SAFE upward

α-synchronizer: (2nd wave)

① when root knows whole tree is SAFE

broadcast CLUSTER_SAFE down — whole tree / preferred link

② node converge cast READY upwards

when — receive READY from all decendants

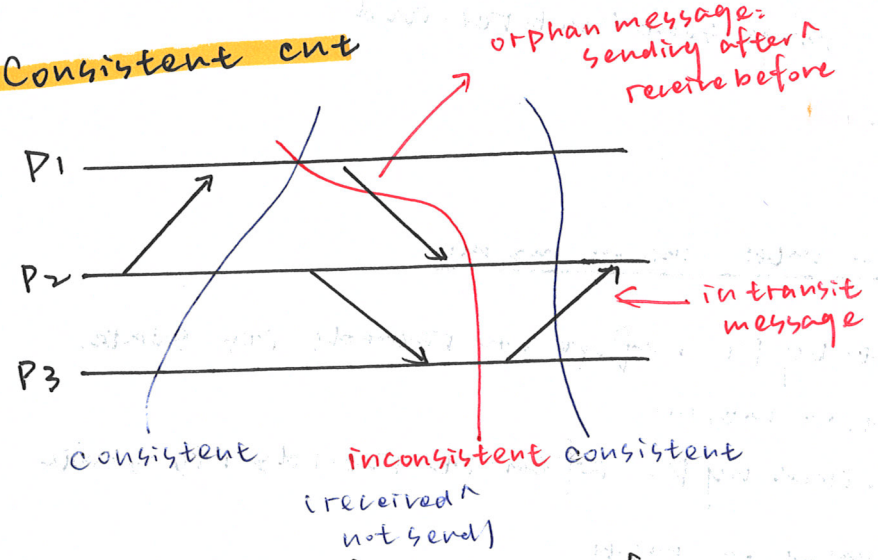
\ and cluster_SAFE along all preferred links

17/11/22 Global State

Problem: determine a global state in asynchronous system
(processes, links, msgs)

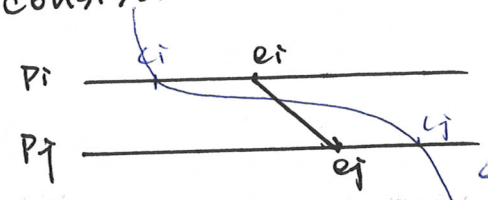
Applications: debug applications (distributed)
detect local stable properties (deadlock)
eg. transferring money in bank account
freeze the money in your account ∧ in transit

Consistent cut



cut is a set of one event for every process $\{c_1, c_2, \dots c_n\}$

consistent cut:



no e_i in P_i & e_j in P_j
that $e_i \to e_j$ & $c_i \to e_i$
& $e_j \not\to c_j$

use vector clock $V(c_i)$ of event c_i

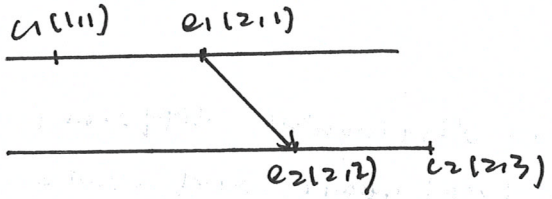
$V(c)[i] = \max(V(c_1)[i], V(c_2)[i], \dots V(c_n)[i])$ component-wise

↑ time of cut

C is consistent iff $V(c) = (V(c_1)[1], V(c_2)[2], \dots V(c_n)[n])$

↙ local component is maximum

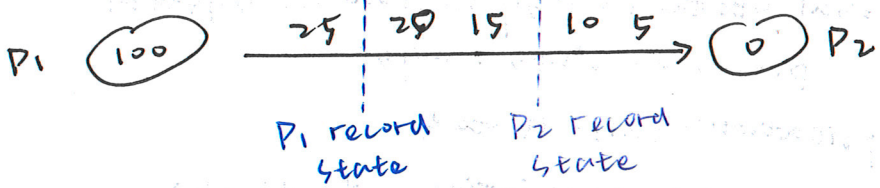
e.g. violation



$V(c) = \max((1,1),(2,3)) = (2,3)$

$(V(c_1)[1], V(c_2)[2]) = (1,3)$

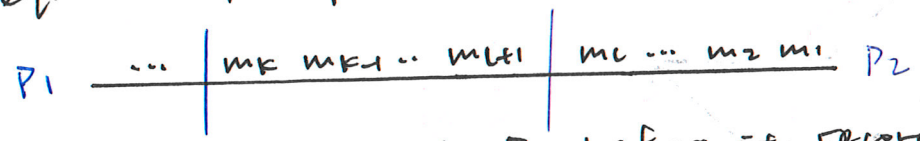
e.g. money transfer



Recorded states { P1: 50
P2: 15
channel: 20, 15
↑ msgs sent ^ not received

State of channel (FIFO)

a sequence of msgs



state of c = msgs sent by P1 before it records its state.

mk, mk-1, ... m2, m1

= msgs received by P2 after it records its state

... mk, mk-1 ... ml+1

{ k > l : state of c = [mk, mk-1, ... ml+1]
k = l : empty
k < l : inconsistent

Chandy's and Lamport's

record local state — blue line

① record local state ↑

② for outgoing channel: send #

incoming :

create buffer: msg after recording state & before receiving marker

wait for marker on incoming channel

Assumption

channel is FIFO

may spontaneously start

append (pid, seq-id) to marker to distinguish different invocation

After algorithm, each process have local states and state for incoming channel

Global state can be collected

Recorded state

Q: what is the use of state if it hasn't occurred?

Might occur if sequence of concurrent event is different.

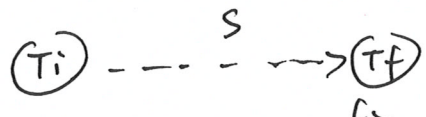
event — post-recording — occurs after process record state

event — pre-recording — before

pre-recording | recording ~~process~~ of state | post-recording

If want to record global state

S = sequence of events



S = e e e e_p e_p e e_p | e e e_p e_p

↑ pre-recording ↑ post-recording

For multiple ~~event~~ processes, ~~subsequence~~ might be out of order.

Reorder events

post pre

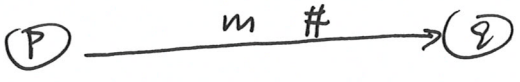
S = e e e e e e_p e_q e e e

① e_p. e_q both send/receive events

record ✓.

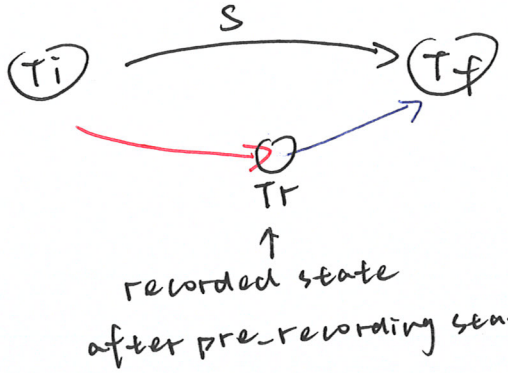
② e_p send. e_q receive. → can't interchange.

but it's impossible



~~e_q~~ e_q = post-recording (x)

reordered states of processes are still same (∵ same events)



S: original → used to check stable properties

properties in Tr also holds in Tf.

eg. if deadlocked in Tr.

→ Tf is also deadlocked

17/11/22

~~Global State~~

Termination Detection

Problem: detect the application is terminated

A distributed application consists process in different processor.

Some p finish spontaneously as done all local work.

active → passive

Passive nodes may become active again if receive msg carry additional work.

problem: some messages are still in transit

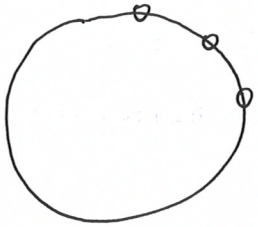
may re-activate process

Termination Detection in a unidirectional ring. (FIFO links)

Naive Solution:

Token: travel along ring

capture the knowledge active/passive /re-activated ?



when token hits process

- passive. → continue
- active → wait until becomes passive

when come back to originating process, report termination

Problem: process can be re-activated

Solution:

introduce color (black, white) for token and process

Process turn black if send message to a ~~lower~~ higher numbered process (send from left to right)

→ spot the passiveness

Token: start with white

turn black if arrives at black process (continue black to initiating process)

when return
- black = no conclusion
- white

Correctness

loc t = location of token

loc t = i if { token in Pi / token in transit. Pi is the last process token is presen

Predicates:

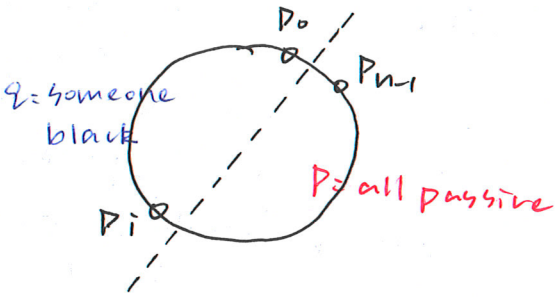
P: for all i. $1 \le i < n$, state_i = passive

Q: There exist an i, $0 \le i \le n$,

color_pi i = black.

r: color_t = black.

At least 1 predicate is true.



Proof: P || Q || r is invariant.

① If token in location 0. (white)

Queue = p0.

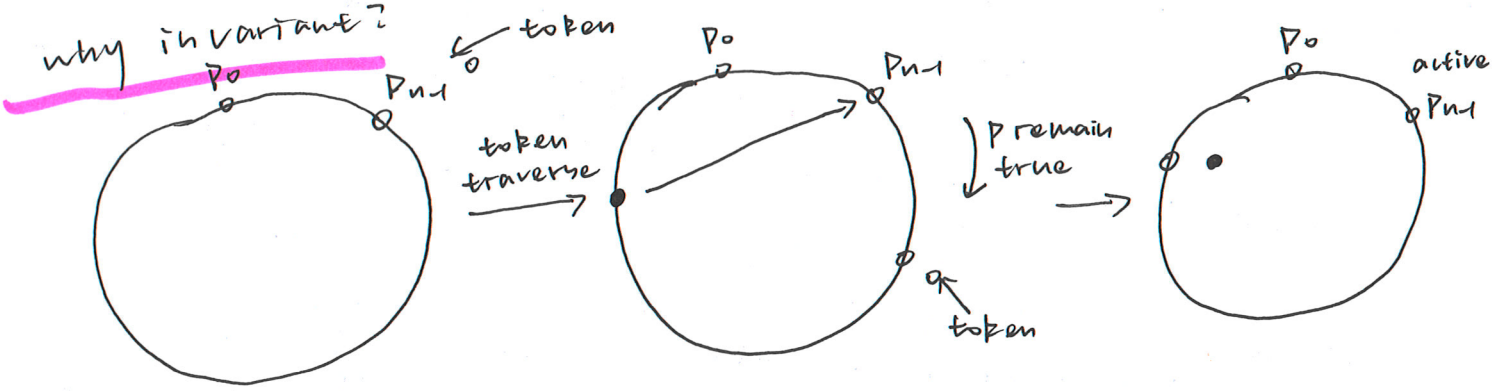
∴ Q x

r x

P must hold (means all process on the ring is passive)

+ P0 = passive ⇒ conclude termination

why invariant?



P: $1 \le i < n$ (n−1) . ~~empty set~~

empty set is always true

P: x

Q: ✓

(token travel and pass P0).

P0 also know about sport

P: x

Q: x

r: ✓

General termination detection

No special structure for network.

All processes and messages have non-negative weights

One special process

Initial weight { special process = 1, all other = 0.

initialization → $w(P_i) = \frac{1}{n}$

when send a message

P_i → P_j

before w_i

after $\frac{w_i}{2}$

w_j

$w_j + \frac{w_i}{2}$

when process terminates

P_i (P)

before w_i → $m(w_i)$ → w

after 0 → $w + w_i$

Termination Condition: $w(P) = 1$

17/11/27 Deadlock Detection

Problem:

Each process request one resource.
They are waiting for each other because of resource request.

Resource model

Every resource is represented by process = process R_A represents resource A.

Processes are waiting for each other.

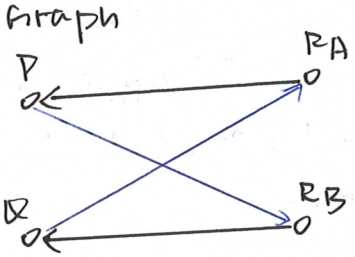
Process P requests Resource A

After request has been granted:

RA is waiting to get it back.

After resource has been released:

Wait For Graph



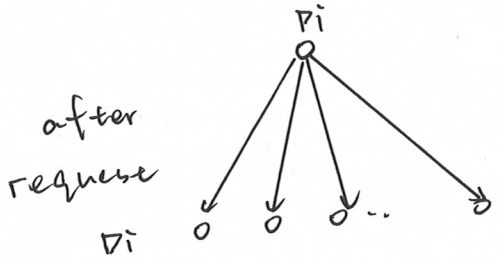
cycle → deadlock

Communication Model

Idea: Process blocks and waits for one message from bunch of msgs.
Pi

Pi: dependent set

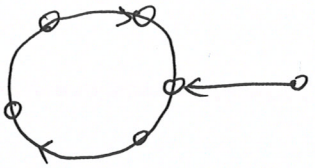
A Process ~~trans~~ turns active when receives a single ~~process~~ message from any message in its dependent set.



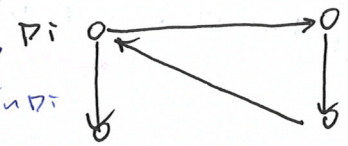
After reception of a single message Pi

Deadlock and WFG graph

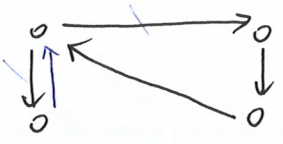
resource model: a cycle

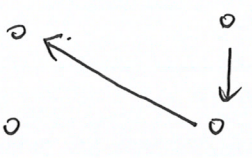


communication model:
a cycle doesn't necessarily indicate a deadlock

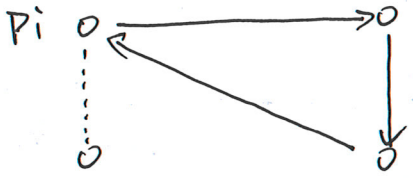


~~not wait~~
if receive 1 msg in Pi
all arrow gone.





Knot: a set of nodes in directed graph
with a path from every process to every other process (not needed complete graph.)
no edge from a node in S to a node not in S

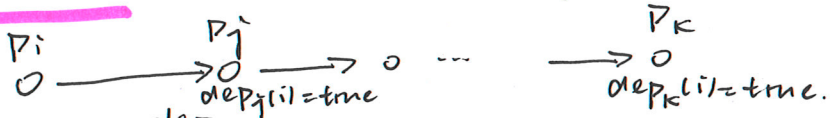


There is a deadlock if there is a knot in WFG.

Chandy-Misra-Haas: Resource Model

Idea: when a process suspects deadlock (wait for long time),
it sends a special message (PROBE) to all processes it wait for.
other processes propogate message
Deadlock condition: if receives one of own messages

Pi is dependent on Pj if there is a ~~graph~~ path in WFG from Pi to Pj

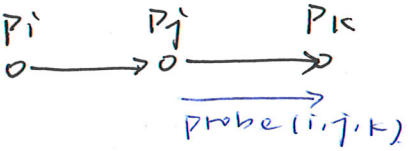


dep_i(j) = ~~dep~~ true : Pi knows Pj depends on it.

Pi is deadlock if dep_i(i) = true

Implement.

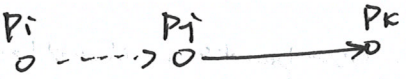
Probe msg: Pj sends probe(i,j,k)
if Pj is blocked
Pj waits for Pk
Pj knows Pi is dependent on it.



Chandy-Misra-Haas: Communication Model

Idea:
when a process suspects deadlock, it sends a special message QUERY
to all processes it is waiting for
those processes propagate these msgs
when a process already receive the msg, send reply immediately.
when a process exhaust all links, sends back a reply.
Deadlock if receives a reply from all processes it send query to

Query / Reply msg (i, m, j, k)



i: initiator of the query

m: query sequence number

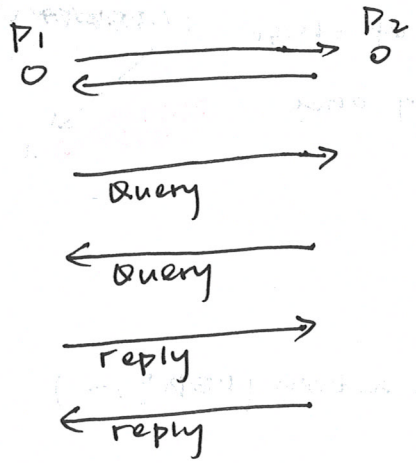
Pj sends msg to Pk

Engager: the node from which the process receives first Query msg.

whenever a node turns active (resources are granted), the whole algorithm stops

A process initiates a query is deadlocked if and only if it receives a REPLY to every Query it sends.

Example



Bracha and Toueg: Requests

N-out-of-M request

A request is targeted at M processes.

A process can be satisfied with a lower number N. reply.

OR: N=1

AND: N=M

Example

1) AND: in distributed database, transaction runs on different location

when commit a transaction, needs agreement from all component.

2) N-out-of-M: quorum-based replication $(N > \frac{M}{2})$

Idea:

Any process that suspects a deadlock can initiate the algo.

Algo consists of 2 phases

{ notify — start of algo (wave-out)
simulate — granting of resources

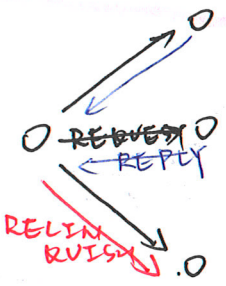
when initiator remains blocked after exe, it is deadlocked

Processes — active — could request and then blocked
blocked — wait for request to be satisfied

Active process can do N-out-of-M request

1) send a REQUEST to all M
2) become blocked
3) wait for postive REPLY from N of those
4) send RELINQUISH to remaining ones
5) become active

2-out-of-3 request



FIFO Channel

WFG

A change in WFG correspond on an action (REPLY, ...)

Schedule: sequence of operations

Schedules applied to G : G -|* G'

A node is deadlocked in G if there is no schedule that takes G to G' in which v is active

Static system: msg are instantaneous

In set = OUT set (∵ no msg in transit)

Transformation in WFG

① Add k outgoing msgs
→ r-out-of-k Request

② Receive REPLY → delete an edge

B

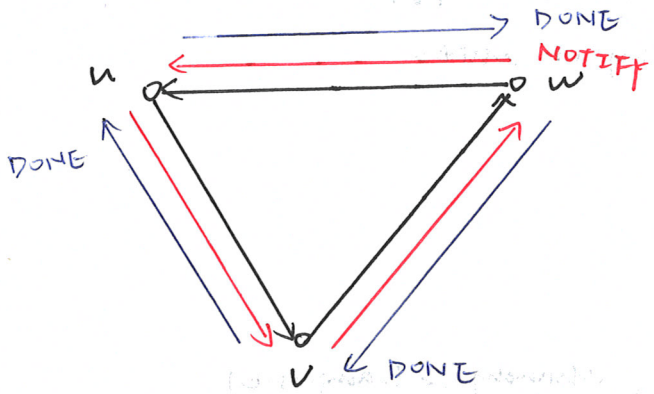
Bracha and Toueg: With instaneous messages

- NOTIFY
  - GRANT
  - ACK
- DONE

Initiator is not deadlocked if ~~free~~ = true

(in other algo. one is deadlocked → others also deadlocked in communication model)

Example



Bracha and Toueg with messages in transit

use color to represent links

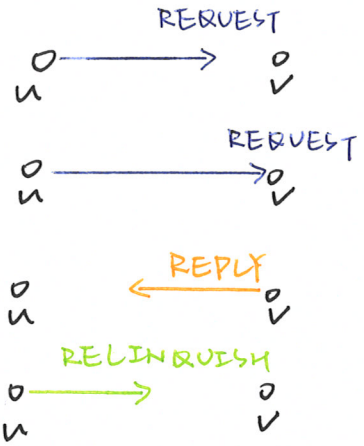
color(u,v):

grey:

black:

white:

translucent



grey eages will eventually become black

Idea: ignore grey

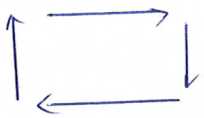
white & translucent will disappear if msg is received

map colored WFG to colorless WFG

A node is active if u ≤ # grey write

REQUEST

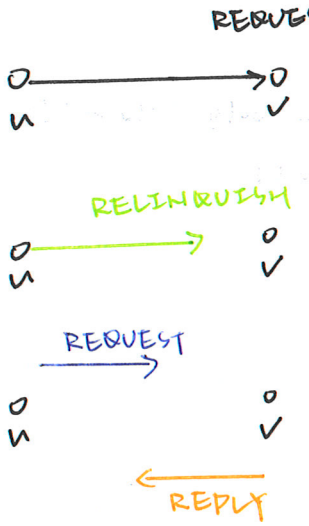
can detect a deadlock about to happen



Nodes need to know color of edges

by exchanging COLOR messages containing IN & OUT set

compare IN & OUT set → figure out color



u in IN$_v$ & v in OUT$_{(u)}$

u in IN$_v$ & u not in OUT$_u$

u not in IN$_v$ & v in OUT$_u$.

(u,v) is grey or white

Dynamic Systems

No static WFG

Trick: detect a global state. (Chandy & Lamport).

apply algo for static system

17/11/29 Mutual Exclusion

Problem: grant exclusive privilege

Requirements: no deadlock
no starvation
fairness

Most algorithms are centralized
- token-based: special msg allows enter into critical section
- assertion-based: msg exchange, figure out it can enter CS based on msg receive

Lamport's mutex algorithm

Channels are FIFO. use scalar clock (+ process id)
Each process maintains request queue (ts ordered)

Pi request access to CS
broadcast (REQUEST, ts, i) to all p (pi incl)

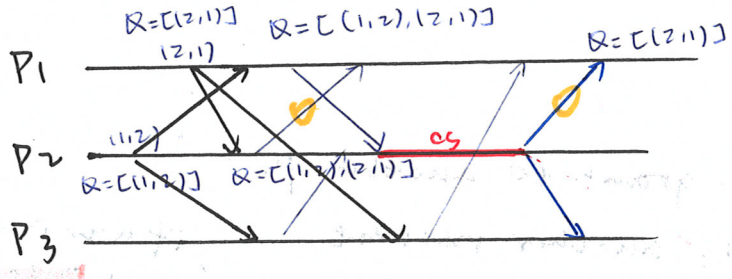
Pj receive request
send (REPLY, ts')
put (ts, j) in request queue

enter CS condition
- receive REPLY from all
- request at head

Pi Release CS
send (RELEASE, r) to all (Pi incl)

Receive Release from Pj
remove (ts, j) from request queue

RQ is equal



3(n-1)
suppress unnecessary REPLY

links FIFO needed

Ricart-Agrawala's mutex algorithm

Optimization of Lamport: combine RELEASE and REPLY
defer REPLY when request has lower priority

links FIFO needed?

Maekawa's mutex algorithm

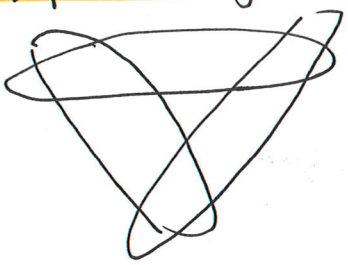
Idea: process sends request to subset of msg
↙
request set

reply: give exclusive access to CS
don't give another REPLY to other process unless receive RELEASE

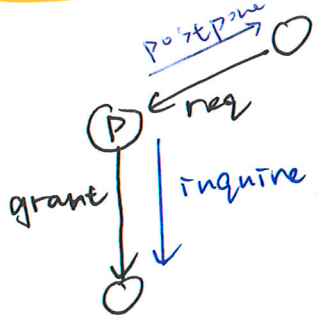
Feature of request sets:
request sets has same sets
every p is contained in the same number of request sets (share the load)
any two request sets have a non-empty intersection (grant permission to only one p)

every p in our request set

Problem: possibility of deadlock



Idea of handling deadlock



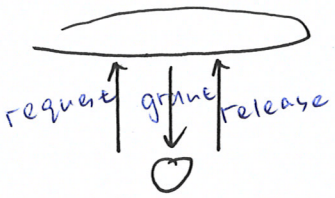
if give grant to another p.
then gets another request

ignore → DEADLOCK.

give grant to older request
by comparing ts
- if new is later. send POSTPONE
- withdraw grant by INQUIRE
- wait for RELINQUISH

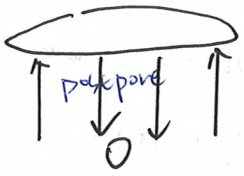
Complexity:

Low Load



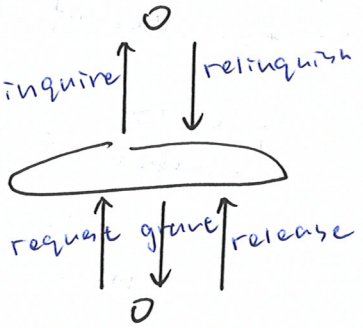
3n
↑
size of RS

Heavy load



4n

old request



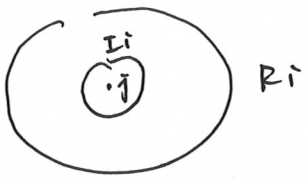
5n

Generalized mutex algorithm

common structure for assertion-based algorithm

each process maintains 3 sets:

- Request set
- inform set:
- status set:



RELEASE: not sent to everyone in Ri.
to a smaller set Ii

Pi inform Pj

Pj records status of Pi

Lemma: minimum size of request set is $\Omega(\sqrt{n})$

K: size of request set

D: the time of every process appear in request set

$$\text{process number} = \frac{\text{number of req set} \times \text{size}}{\text{number of appearance}}$$

$$n = n \times \frac{K}{D} \quad K = D$$

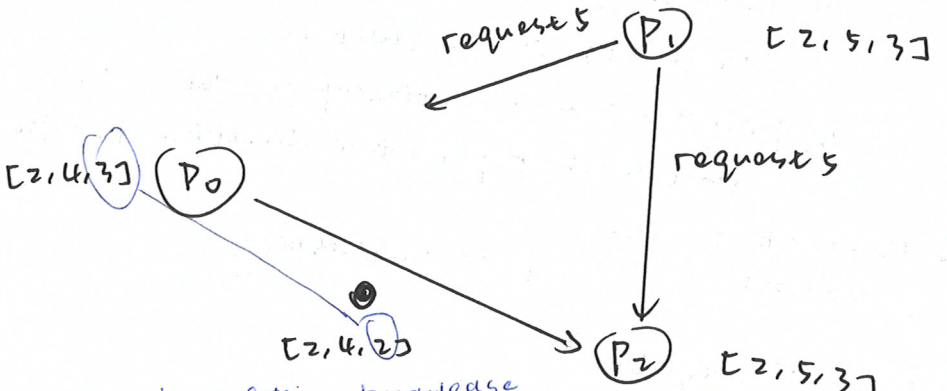
Suzuki-Kasami's mutex algorithm

token-based: a p can enter CS when possesses token

Idea: number request sequentially

broadcast request

token transfer ~~to~~ knowledge to req has been satisfied



token can carry knowledge for latest request has been satisfied

↑ latest knowledge for P0

P0 → P2 token

token satisfies request No.2 of P2.

P0 knows request No.3

Problem: process with high number may get starvation as token may circulated in low number of processes

Variation: In token maintains a queue

Singhal's mutex algorithm

Improvement: not send request to all processes

send to p which may possess token

Implement: each p maintain state array S[1...n]

possible states:
- R = requesting token
- E: executing CS
- H: keeping token & outside CS (don't know where to send)
- O: other

Initialization:

suppose token is located in P1

in P1: S[1] = H.

S[j] = O. j = 2...n.

in Pi: S[j] = R. j = 1...n-1

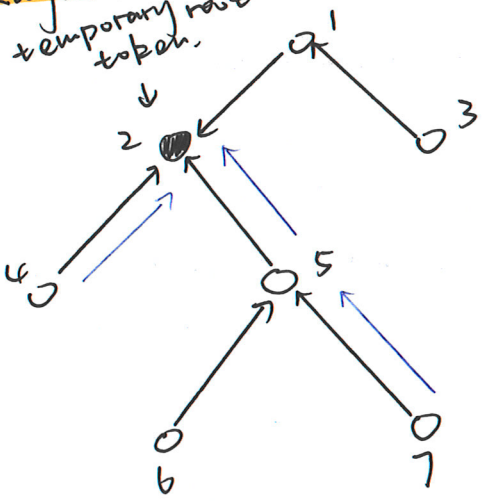
S[j] = O. j = i, i+1...n

token carries TN and TS

TN[i] = 0. TS[i] = O

↑ request num ↑ process states

Raymond's token-based algorithm



Idea: 1) suppose a spanning tree

2) process holding the token is root

3) process sends request in the direction towards root.

4) let a single process represent all requests in a subtree

5) process has a notion of holder, which is the id of p in the direction of token.

6) each process maintain a request queue
   - own id
   - id of neighbor

   p is requesting

Msg complexity: depends on depth of tree

token does a tree traversal. : travel each link twice.

every token step is preceded by a request step

17/2/4 Election

Problem: every process has an integer.
Elect a process with largest/smallest integer.

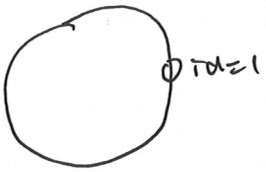
Trivial solution: every process sends id to every other process

Anonymous network: nodes initially don't have ids
Election is impossible in anonymous network

Comparison-based algorithm

Non-comparison-based algo in synchronous ring

all process know current round number



round 1: every process inspect own id=1
if so. send msg along ring
if no. in first n round, no msgs

round n+1: id=2?
:

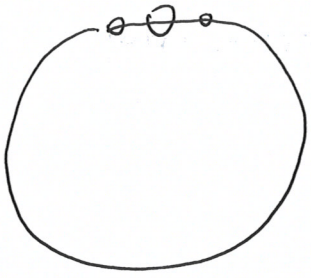
msg/time complexity: O(n).

Compare round number & id.

Bidirectional ring (asyn) O(nlogn)

Idea: a process check if its id is largest among two neighbors. — if so. survive
no. stop

in round k, check 2^{k-1} process on both sides



1) own id larger
discard

2). own id smaller

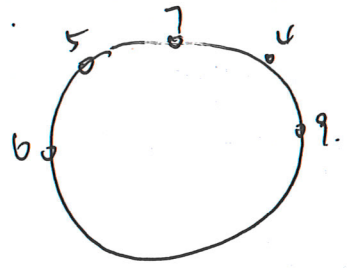
3) hop count > 0.
propogate msg. with hop--

4) reply ok.

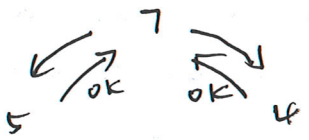
A process starts phase k+1. if receives 2 phase-k ok msgs.

A process is elected. — don't know size of ring.
know: ① receive own probe msg.
② process receives same id from either side
stop at k round, k is $2^{k+1} \geq n$. smallest

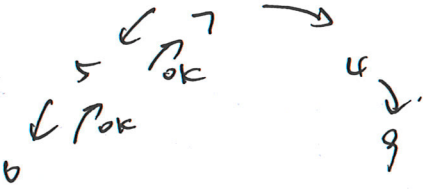
e.g.



four phase1:



phase2:



Bidirectional = solution 2

in first round:
exchange ids with 2 neighbors

a process remain active if own id is larger
else passive.

every next round: repeat round 1 in virtual ring.

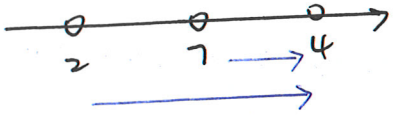
A process is elected if receives own id.

msg complexity: $2n \log(n)$.

Unidirectional ring: Peterson's

A simulation in unidirectional ring of solution 2 in bidirectional ring.

tid = id.
ntid: upstream neighbor
nntid: max(tid, ntid)
if ntid ≥ tid, ntid ≥ nntid, remains active
tid = ntid
else turn passive



every subsequent round, repeat round 1 in virtual ring

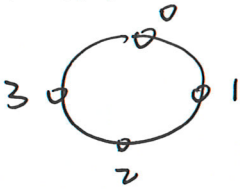
A process is elected if receives own id.

Active nodes will cut at least half every round,

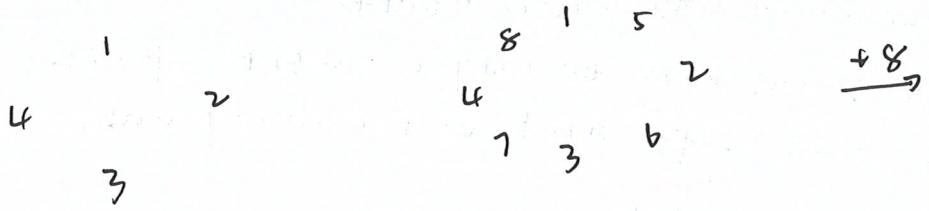
msg complexity: $2n \log(n)$ (at most)

$2n$: in every round 2 msgs sent along one link
$\log n$: number of rounds at most

Q1: when algo terminates after one round?
increasing id. along ring.
/decreasing



Q2: ring of size $n=2^k$ use k rounds?



bit-reversal (list 2^k-1)

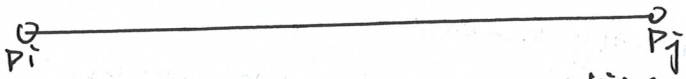
0 1 2 3

00 01 10 11

00 ↗ 10 ↘ 01 ↗ 11 ↘.

17/12/06

Assertion:



If id of Pi still survives in some active process Pj, all processes between Pi and Pj are passive.

Time complexity: $2n-1$

Complete Network Synchronous

straightforward: every process sends its id to everybody else.

n^2 msgs.

Afek's and Gafni's

Idea: successively send Id to ever larger sets of processes and wait for Ack. from all of them

If a process receives an ID larger than its own, send Ack.

A process is killed if it doesn't receive all Acks it expects.

A process adopts the largest id it sees.

Syn

Alive candidate process send msg to larger subsets in successive rounds.

Node { candidate process
ordinary process : awake by reception of msg (and doesn't want to be elected.)

spawn only spawns ordinary process

candidate: send msgs to remaining P.

ordinary : receives msgs from candidate process and compare.

msg: (level, id)

level = the number of rounds since start

candidate process sends msg to larger subsets of sizes powers of 2, and keeps track of remaining links

msg compared in lexicography:

(level, id) (2, 5) > (1, 3001)

A node with largest id among those start earliest wins.

A node is elected if receives Ack from all other processes

levels are not strictly needed.

Q: ~~distance~~ disadvantage of not using levels?

time may ↑.

Time complexity: $\log(n)$

Msg : $2n\log n$

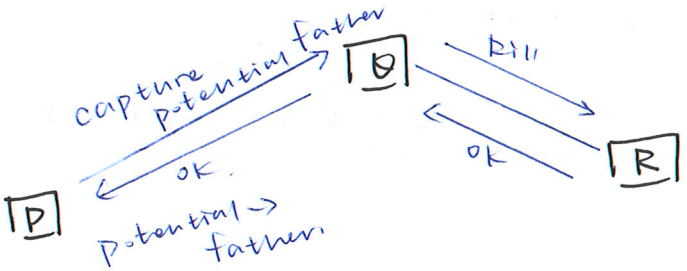
Afek's and Gafni's asynchronous algorithm

It doesn't make sense for a node to wait for all msgs.

It reacts once receives a single msg.

If one node captures a node previously captured by another node, that node has also to be captured.

A node keeps two special link { father : link to owner; potential father : to potential new owner



Time complexity: $O(n)$

Msg : $n\log n$

Exercise Afek's and Gafni's

d. all processes start algorithm in round 1.

1st round.

$8 \to 0$

$9 \to 1$

$\vdots$

$15 \to 7$.

2nd round

$12 \to 8$ $\{8, 0\}$

$13 \to 9$ $\{9, 1\}$

$\vdots$

$15 \to 11$ $\{11, 3\}$.

3rd round.

$14 \to 12$ $\{0, 8, 4, 12\}$

$15 \to 13$ $\{1, 5, 9, 13\}$

4th round

$15 \to 16$ $\{0, 2, 4, 6, 8, 10, 12, 14\}$

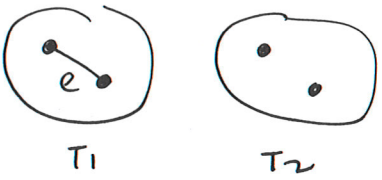
Ch4. Minimum Weight Spanning Trees

Problem: a weighted undirected graph $G=(V, E)$
make a selection of links that connect the network completely
and total weight of links is smallest among all possibilities

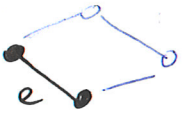
Difficulty: compute in distributed way

Lemma: There is a unique MST.
Assume all weights are difficult

Proof: suppose 2 MSTs: T_1 and T_2
there is some edge that is part of T_1 but not T_2



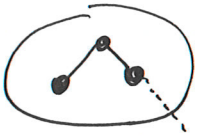
take one with lowest weight: e
add edge to T_2 then gets a cycle
T_1 has no cycles. $\Rightarrow$ T_2 contains an edge e' that not in T_1.



T_2 with e.

$w(e') > w(e)$
throw e' and keep e. T_2 has a lower sum of weight than original T_2.

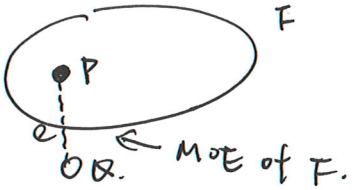
Notion:
fragment: subtree of final MST.
outgoing edge: one node in fragment and one node outside.



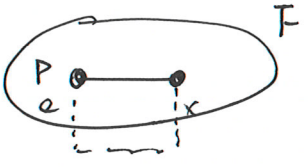
MOE (Minimum-weight outgoing edge)

Lemma2: If fragment F. MOE. e:
F with e and node incident with e is also a fragment.

Proof: suppose not.
$\Rightarrow$ in final MST there must be path connects P and R.



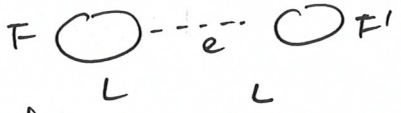
$\Rightarrow$ there is a cycle if add e to final MST.



- at least one edge x is also outgoing edge of F.
∴ w(x) > w(e).
replace x by e yield an lower weight MST.
∴ contradiction.

Notion:
level: fragment has level.
Fragment with single node has level 0.

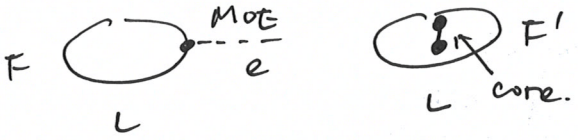
Merge = two fragments with **same** level.
have common MOE, level
yield a new fragment whose name is the core ↑ last edge used for merge
weight of.



level increased by 1.

Absorb = a fragment of a **lower** level into fragment of a higher level

level stay same.
core same
name same



Q: Minimum size of fragment level k?

binary tree

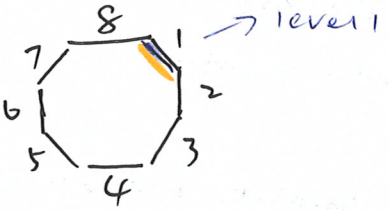
2^k.

2 level 0 make level 1

for level 2, needs 2 level 1, 4 nodes.

Q: Is it possible for MST have level 1?

First step is to merge.
Reset all absorb. (link weight as a clock)
increase / decrease



17/12/11

Data Structure

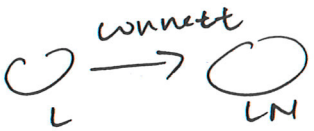
edge_state { ? in-MST / in-MST / not-in-MST

node_state { sleeping (initial state) / find / found

Message type { initiate — from core find MOE
report — report candidate MOE to core
test — test an edge for being candidate MOE
accept
reject
change-root — change direction for core
connect

Algorithm

wakeup().
check MOE.
MOE in MST
connect(MOE)



receive connect
if sleeping
wakeup().

if L < LN.
In-MST.
Send initiate msg.

find_count.

else
if - ? in-MST
append queue

else
send initiate message

Complexity

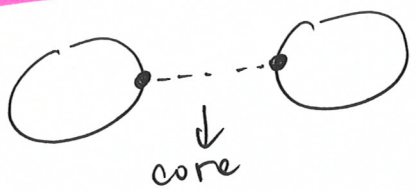
Reason absorb: merge takes a lot effort from fragment to find MOE

absorb only takes small fragment searching, doesn't take large one.

Message complexity: N nodes and E edges

$2E + 5N \log N$

Election in MST



pick node in core with highest / lowest weight

Q: Is the ① core of final MST always same if build on same network several times?

② same edges for merges & absorbs? multiple

Exercise MST.

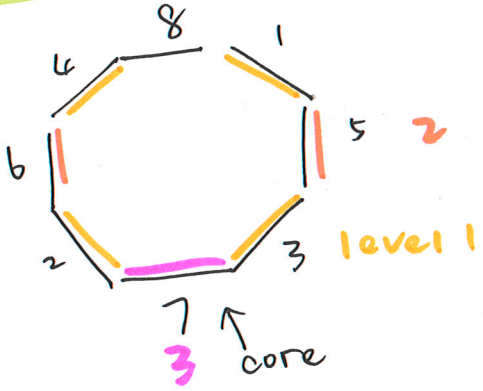
d. final result of MST has level 1?

start with merge, and then all absorbs

e. Maximum level of MST with $n = 2^k$ nodes.

k.

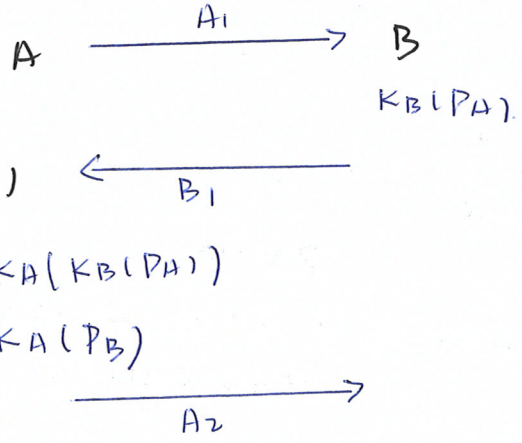
an example when n=8?



Chap 5. Fault Tolerance and Consensus

Example in real world,

Alice and Bob want to make an appointment.



Alice sends Msg A1.

Bob receives A1. knows PA: KB(PA)

Bob sends back B1

Alice receives B1.

⋮ ⋮

Problem: messages have arbitrary delay
and may get lost.

Assume the problem has a solution of minimum number of exchange
last msg may be lost

Common knowledge:
(everyknows)(everyone knows))

Fault Tolerance & Consensus
processors decide on the value of single bit (Yes or No).
without errors, exchange value → majority →
failure in system. eg. x tell proposal
tell different value to diff proc

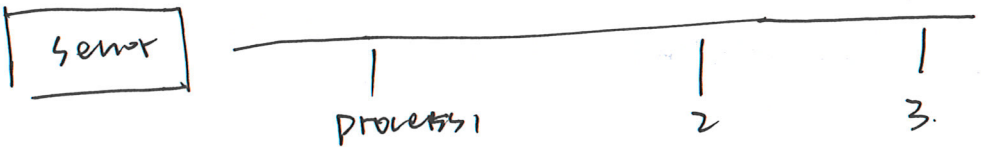
Application

1) transaction in distributed DB
all sites agree on completion of transaction.
2) replicated DB
if want to modify data
in favor of changing?

Original Motivation 1980s.

compute direction based on temperature

use redendancy to deal with faults



senor give all same value?

↘ to agree on input.

Fault classification

fail-stop : just stop

omission: : skip a msg

performance: slow

byzantine: execute random behavior.

Model properties

synchronous v.s. asynchronous

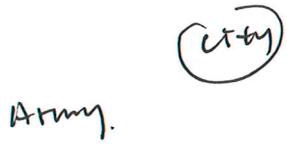
↘ differentiate delay & failure

easy.

authentication : processor sign msgs.

Network connectivity: complete network

Byzantine



Army want to attack the city simultaneously.

Need to reach an agreement

attack / not attack?

17/12/13

Byzantine agreement problem

source/commander.
→ 0 (0/1).

① agreement.

② validity: correct processes agree on correct initial value — start with binary value

③ termination: within finite time

Variantion of problem

All processes starts with value vi

1) agree on a vector with a value for all
(v1, v2.. vn)

2) single value

majority (v1, v2.. vn)

Simple solution for stopping failure

Every process starts with value v.

collects all decision values received in set W.

W = {v} initially

In each round (f+1), broadcast W to all other p.

received Wj.

W = Union (Wj).

if p fail., its own value circulate in system.
(achiere validity)

Finally < if w contain single v. → v.
default / majority?

Q: why need round? f+1 round. f processors.

At least 1 round without failure.

Optimization.

1) |w|=1 or |w|>1?

→ agree on default.

only broadcast { first initial
first different value → decide default

Byzantine Solution.

number of processes: n.

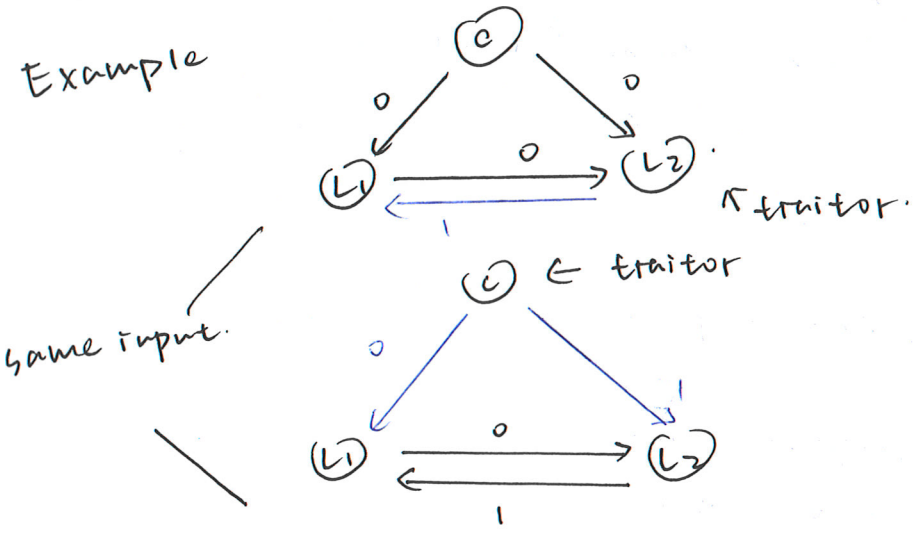
possible failing processes: f.

$f < \frac{n}{3}$

Minimal number of rounds: f+1
in deterministic solution.

Some randomized solution have lower expected number of rounds.

Example



Synchronous Byzantine Solution

recursive

Oral Message: No signature

Bottom case = OM(0)
no failure.
commander broadcast initial value
every other p accepts

OM(f). f>0
commander broadcast initial
number process: commander=0
lieutenant 1, .. n-1.

vi: received from commander

Recursive.
every p executate O(f-1). act as commander
exclude c.
until O(0).
decides on majority (v1, .. vi) .. vn-1)
from i
when Ln-1 is commander
decision take
majority( )

Number of executions: exponential

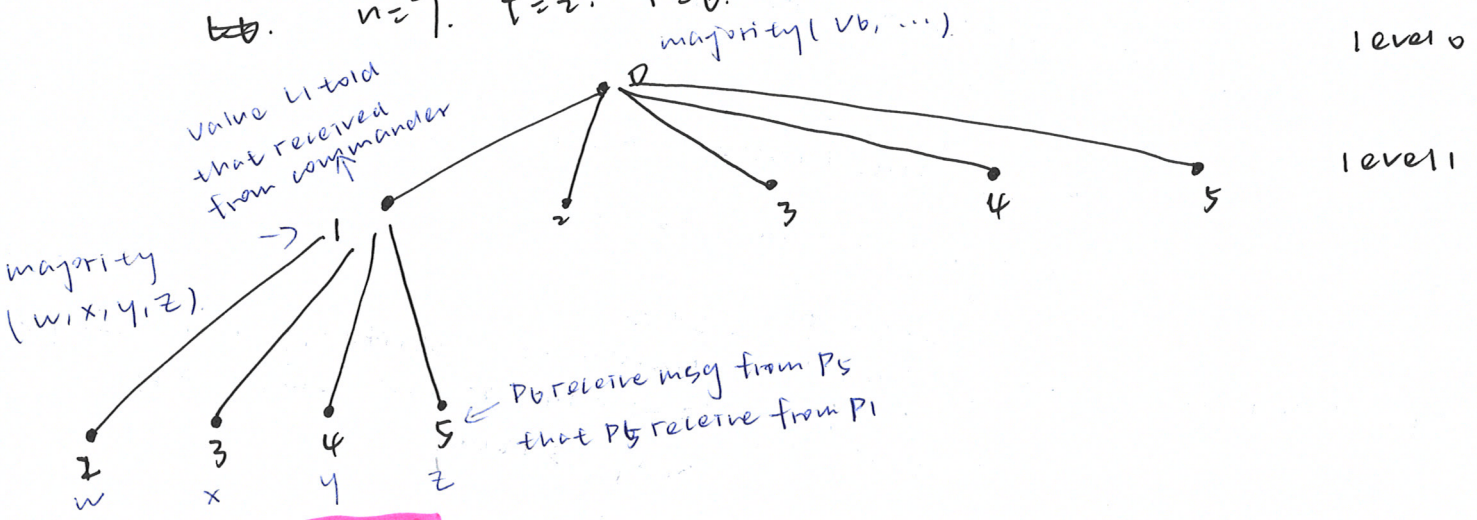
OM(f) = 1

OM(f-1) = (n-1)

OM(K) : (n-1)(n-2) ·· (n-f+K) times

Example. Lb local tree

~~Lb~~. n=7. f=2. i=6.



Decision Taken:

Leaf level: decide on value received OM(0)

higher level: take the majority of decisions made in child node

level 0: final decision include value received from commander.

Byzantine agreements with authentication.

Each msg carry a signature, which cannot be forged

P0 is commander.

Allow any number of traitor. f.

structure of msg: (V: S0: Sr1: ·· Sik)

Every process maintain a set of order V. for deciding (majority ··)

↑ all potential value in msg received

Commander: send (V: S0) to all other.

Lieutenant: Append signature. Union V with v.

Send msg to all processes do not received.

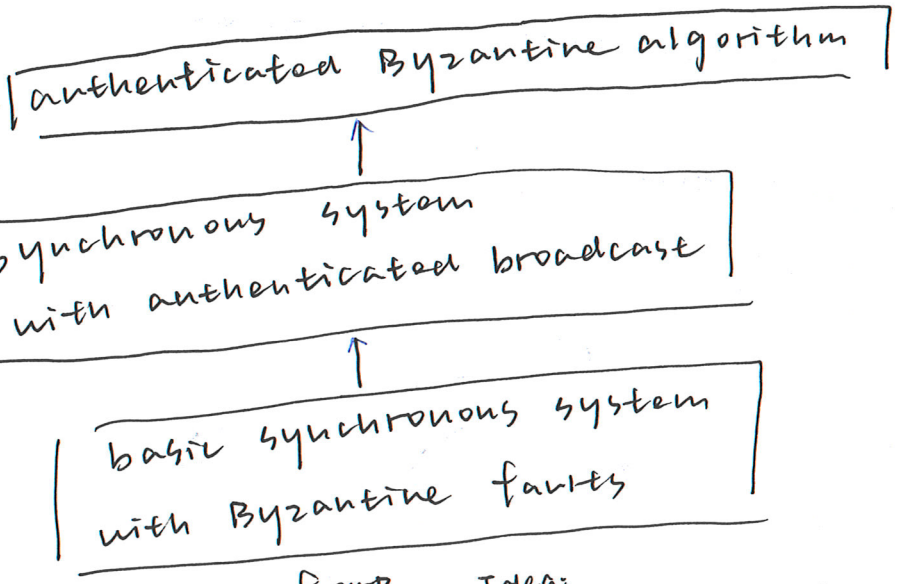
Example: ① ← traitor

1:0 0:0

Efficient solution to Byzantine Agreement

Polynomial

Make communication look authenticated by authenticated broadcast / consistent broadcast



Broadcast messages (i, m, r) ← from p. ; r: round

Idea:
receive msg → verify signature → accept
(faulty process accept much later)
accept → relay

Properties of authenticated broadcast:

1) correctness: correct process accept msg from .. .. in same round.

2) unforgeability: if a process is correct and doesn't broadcast (i, m, r), no correct process will accept it.

3) relay: if a correct p receives msg in round $r' \geq r$, every other accepts it ~~in later~~ round. at latest the next

Algorithm

commander p_0 start with v

All other p start with $v=0$

Only $v=1$ are sent

if $v=1$ & never do broadcast before

broadcast $(i, 1, r)$

if in previous round received 1 from p_0 and received 1 from other processes $r-1$ times

set $v=1$

decide (v)

Randomized Byzantine Agreement (exponential)

Solution for asynchronous & synchronous system

$n > 5f$

Every process starts with v.

Algo proceeds in rounds consisting 3 phases.

- notification phase : propogate msg
- proposal phase
- decision phase : check enough proposal for final decision
  → if no. flip coin to decide v start in next round

If wait for msgs from everybody else, stop after receiving $n-f$ msgs.

At least $n-2f$ correct.

Implement:

broadcast NOTIFICATION to anyone else. — notification phase

WAIT for $(n-f)$ Notify msg.

→ If gather enough notifications for one value. → $> \frac{n+f}{2}$

broadcast that value

→ If not.

broadcast ? (don't have proposal)

$\frac{n+f}{2} - f = \frac{n-f}{2}$ = half reliable process — proposal phase.

→ If decided in previous round.

STOP

→ wait $(n-f)$ proposal msgs

If receive support $> f$ (at least 1 correct supports)

$v \leftarrow w$. (adopt proposal)

if $> 3f$ msgs. — decision phase

take decision on w

$v \leftarrow$ random.

Lemma1: No simultaneous contradicting proposals by correct processes in same round.

Proof: receive $\frac{n+f}{2}$ proposal msgs.

more than $\frac{n+f}{2} - f = \frac{n-f}{2}$ from correct processes

which is majority

Lemma2: when all correct processes have same value, immediate decision.

if all correct start with v in round r, all decide v in round r.

Proof: receive $n-f$ notification. at lease $n-2f$ from ~~correct~~ correct.

$\because n > 5f \quad \therefore n-2f = \frac{n}{2} + \frac{n}{2} - 2f$

$> \frac{n}{2} + \frac{5f}{2} - 2f = \frac{n-f}{2}$

↑ correct processes propose v

$\because n > 5f \quad \therefore n-2f > 3f.$

Lemma3: If a correct process decides v in round r, all ... r+1.

Proof: from Lemma2, all correct process proposal v in round r+1.

if decides v in r, must receive $>3f$ proposals on v.

m of which is correct. $m > 2f$.

Correct processes receive $m-f > f$ proposals on v,

which is the condition for adopting value.

2017/11/8 Stabilization

A stabilizing algorithm:

deal with transient faults in an implicit way

(operate as usual, if meet some errors, resolve it)

can be started in any configuration

Real world problem:

orchestra play without constructor

player listen what neighbor playing

~~restart~~ if find play at wrong page, restart..

Types of transient fault:

possible causes: 1) no connection

2) memory crashes

3) transmission errors

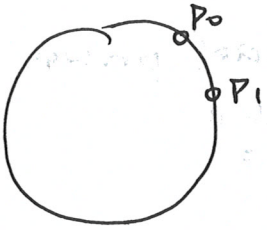
Definition

stable predicate P.

legal states is defined by predicate.

First stabilizing Algorithm: Dijkstra.

Initial problem: mutual exclusion in unidirectional ring



shared memory, Pi can inspect value in P0 & its own

from predecessor

composite atomicity: reading and writing in

one atomic/single step

Pi has an integer Xi modulo k.

Idea:

Ordinary process Pi: compare value with predecessor

Pi-1 Pi

0 → 0

8 8

8

if different → copy it.

Special process P0: compare value with predecessor

if equal → increment its own value

$X0 = (X0 + 1) \bmod k$

Pi-1 Pi

0 0

8 8

8 9

Q: what is the maximum number of steps before only one process is enabled?

$$(N-1) + (N-2) + \cdots + 2 + 1 = \frac{(N-1)N}{2}$$

Q: what is legal states of ring?

↑ only one process can enter critical section

K, K, K ... , K, K (P0 can enter)

K, K, K, .. K, K-1, K-1, K-1 .. (P1 can enter)

Correctness:

Idea: missing label concept:

All processes except P0 copy value.

P0 introduces new value

if K > N, P0 go through 0, 1, 2 .. K-1

∴ sometime introduce new value in system

if K = N

if K = N-1

if K is smaller, don't allow a lot different values in system, don't work. (ex. 3)

Remark:

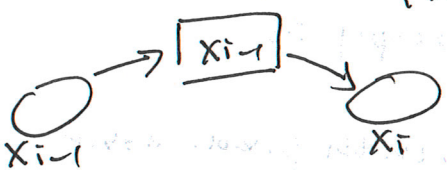
If number of processes is not a prime, a special process is needed.

If shared memory with read/write atomicity:

in one step can only read or write

local integer + intermediate process

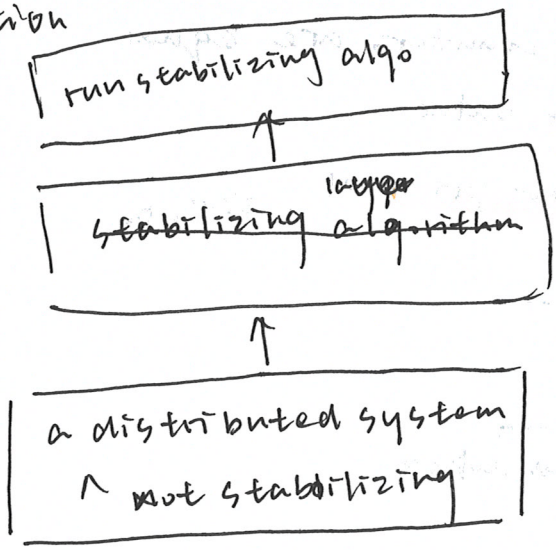
(send by one & receive by other)



K > 2N-2

requires stabilizing communication

simulation



make it stabilizing

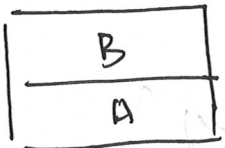
Fair protocol composition stabilizing algorithm for communication

Building a stabilizing algo on two different layers

If have a combined protocol executing both,

do one protocol alternatively, ~~ads~~

would get stabilizing algo.



A is stabilizing layer.

B.

P is also stabilizing layer by combining 2.

state transition functions:

for P_1: $f: A \to A$

for P_2: $f: A \times B \to B$

↑ can access lower level

↑ only modify B

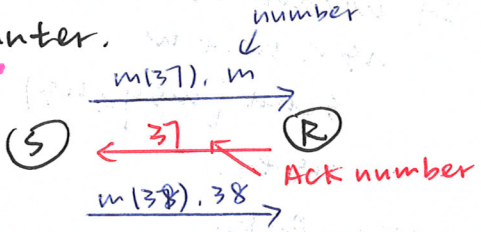
A protocol P is a fair composition/combination is executing states transitions in alternate fashion.

A stabilizing datalink algo: stop-and-wait

Idea: send a msg, and wait for acknowledge to send next msg.

sender and receiver maintain counter.

send msg with msg number.



Whenever a receiver receive a msg,

if is a new msg, adopt value in msg,

then send Ack back.

TimeOut = If sender doesn't receive Ack.

resend msg

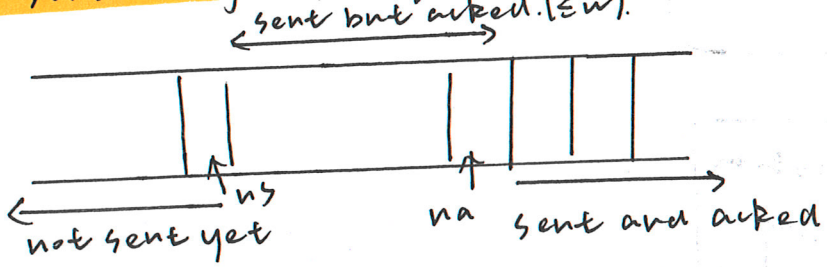
Legal state: all counters are identical. (sender, receiver, msg)

Remark:

System in legal state if all counters are equal.

Sender introduce new higher value

A stabilizing algo: sliding-window protocol TCP like



sender { w: window size
ns: the number of next msg to send
na: first non-acked msg

receiver: nr: next msg to be received

Stable predicate P

((na ≤ nr) and (nr ≤ ns) and (ns ≤ na + w))

can't have Ack if don't received — can't received if don't send — window has size

and

for each (message, i) in channel SR, i < ns

and

for each (ack, i) in channel RS, i ≤ nr

can't have Ack if no received

Implement (Not stabilizing version)

Send msg:

If window is not full:

ns < na + w

send (msg, ns)

ns++.

Receive Ack.

If is new Ack ← possibly not FIFO

adapt na = i

Timeout in sender

resend all msgs in window unacked

Receiving a msg:

check if it's the next msg expected

adapt nr.

send Ack. (whether expected or not)

(receive all previous one)