1 Introduction - February 19, 2020

1.1 Defining Dependable Systems

QUOTES:

A distributed system is a system where a computer of which you did not know it exists can prevent you from getiing your job done. - Leslie LAMPORT

There is perhaps a market for maybe five computers in the world. - TJ WATSON

FAULT → ERROR → FAILURE
- Train delayed because of tree has fallen on the tracks
- Travelers reach destination too late
- Alice misses her exam

	FAULT	ERROR	FAILURE
Train:	Tree fallen	no train	delay for passengers
Journey:	Train delay	delay	reached destination 2h after intention
Exam:	arrival 2h late	missed time-slot	repeat exam

FAULT: cause of failure
ERROR: internal state of system, not according to specification
FAILURE: observable deviation of specification

FAULT examples:
- timing
- cables
- power supply
- messages lost
- data loss (solved with RAIDs)

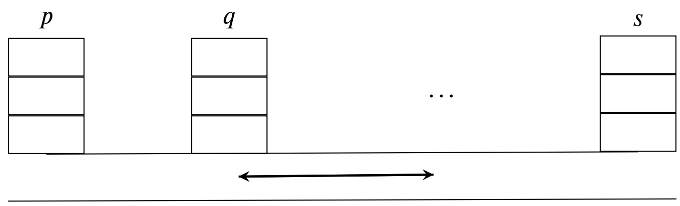
1.1.1 How to make systems tolerate faults

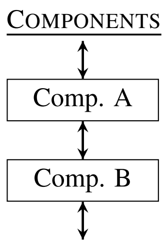
- PREVENTION
- TOLERANCE
  - Replication/Redundancy
  - Recovery
- REMOVAL
- FORECASTING/PREDICTION

SAFETY $\neq$ SECURITY
SAFETY is connected to loss of live/material due to accidents
SECURITY is connected to malicious intent

1.1.2 Defining distributed computation

Processes $\Pi = \{p, q, r, s \dots\}$
$\mid \Pi \mid = N$





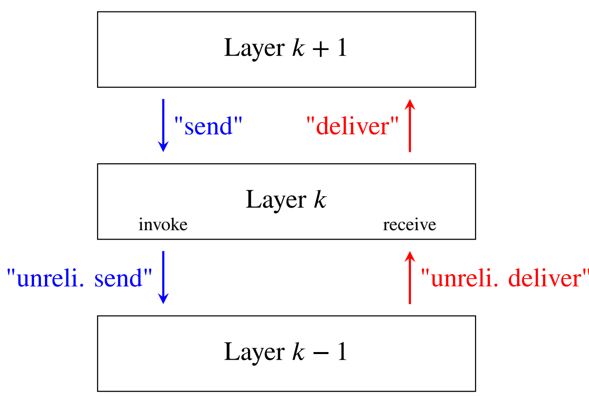
EVENTS for Component c:

$$\langle c, event \mid param_1, param_2 \ldots \rangle$$

upon $\langle c, ev_1 \mid param_1 \rangle$ do
  do something
  trigger $\langle b, \textit{domore} \mid p \rangle$

upon $\langle b, \textit{domore} \mid p \rangle$ do

1.1.3 Layered modules



Events either travel:
- upwards (red): indication
- downwards (blue): request

Events on a given layer may be:
- input events (IN)
- output events (OUT)

1.1.4 Module Jobhandler

Events:
  Request: $\langle$*jh, handle* | *job*$\rangle$
  Indication: $\langle$*jh, confirm* | *job*$\rangle$
Properties:
  Every job submitted for handling is eventually confirmed.

Implementation (synchronized) JOBHANDLER
State
  ...
upon $\langle$*jh, handle* | *job*$\rangle$ do
  "process job"
  trigger $\langle$*jh, confirm* | *job*$\rangle$

upon ...
upon ...

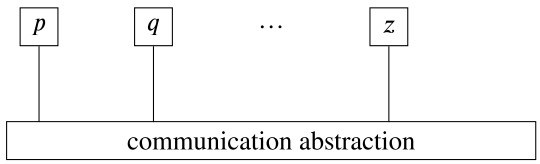
Implementation (asynchronized) JOBHANDLER
State
  $buf \leftarrow \emptyset$
upon $\langle$*jh, handle* | *job*$\rangle$ do
  $buf \leftarrow buf \cup \{job\}$
  trigger $\langle$*jh, confirm* | *job*$\rangle$

upon $buf \neq \emptyset$ do
  $job \leftarrow$ some element of buf
  "process job"
  $buf \leftarrow buf \setminus \{job\}$

1.2 Concurrency and Replication in Distributed Systems

2 Models and Abstructions - February 26, 2020

2.1 Processes and Protocols



- Set of Processes Π
  $|\ \Pi\ |= N$
- A process is an automaton
- A protocol is a set of processes

2.1.1 Execution

- Each computation step and every step of sending a message or receiving a message is an event
- An execution (history) is a sequence of all events of the processes as seen by a (hypothetical) global observer
- trace = execution

2.1.2 Properties

Used for specifying the abstractions:

- **Safety properties** (*something "bad" has not happened*)
  If a property P has been violated in some execution E, then there exists a prefix E' of E such that in every extension of E', property P is violated
- **Liveness properties** (*something "good" will happen in the future* [EVENTUALLY])
  Property P can be satisfied by some extension $\widetilde{E}$ of a given execution E

Safety or *Liveness* alone is not very useful. Only combination of both properties.

2.1.3 Process Failures

A process consists of different modules - if one of them fails the entire thing fails at once.

★ ***Crashes***

- *Omission failures* (*message sending and receiving events are omitted*)
- *Crash-Recovery Failure*
  - store(-) operation to write to stable storage
  - upon recovery, one can restore(-) data from this stable storage
- *Eavesdropping Fault*

★ ***Arbitrary Fault (Byzantine Fault)***

2.2 Cryptographic Abstraction

- ***Hash functions*** (SHA-256)
  $H : 0, 1^{\star} \rightarrow \{0, 1\}^{k}$
  - collision-free: difficult to find x, x' with $x \neq x'$ and $H(x) = H(x')$
- ***Message-Authentication-Code (MAC)*** (HMAC-SHA256)
  - *authentication*$(p, q, m) \rightarrow a$
  - *verifyAuth*$(p, q, m, a) \rightarrow$ YES/NO
- ***Digital Signatures*** (RSA, (EC)DSA)
  - *sign*$(p, m) \rightarrow s$
  - *verifySign*$(p, m, s) \rightarrow$ YES/NO
  ★ *Correctness*:
  $\forall m, p :$ *verifySign*$(p, m, sign(p, m))$ = YES
  ★ *Security*:
  $\forall m, p, s :$ *verifySign*(p, m, s) = NO, unless p has executed $sign(p, m) \rightarrow s$

2.3 Communication Abstraction

Every process can send messages to every other process.

2.3.1 Stubborn point-to-point links

Events:
$\langle sl.send \mid q, m \rangle$ { *send message m to process q*
$\langle sl.deliver \mid p, m \rangle$ { deliver a received message m from process p
Properties:
Stubborn delivery:
If a process sends a message m to process q, then m is infinitely often delivered at q.
No creation:
If some process q delivers some message m from p then process p has previously sent m to q.

2.3.2 Perfect point-to-point links

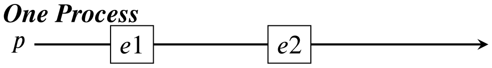
Events:
$\langle sl.send \mid q, m \rangle$
$\langle sl.deliver \mid p, m \rangle$
Properties:
Reliable delivery:
If a correct process sends a message m to a correct process q then q eventually delivers m
No creation:
If process q delivers some m from process p then p has sent m to q
At-most-once delivery:
Every message m is delivered at most once from p to q.

2.3.3 Alg. impl. perfect links (pl) from stubborn links (sl)

INIT:
$\mathbb{D} \leftarrow \emptyset$
upon $\langle pl.send \mid q, m \rangle$ do
  trigger $\langle sl, send \mid q, m \rangle$
  upon $\langle sl.deliver \mid p, m \rangle$ do
    if $(p, m) \notin \mathbb{D}$ then
      $\mathbb{D} \leftarrow \mathbb{D} \cup \{(p, m)\}$
      trigger $\langle pl.deliver \mid p, m \rangle$
      …

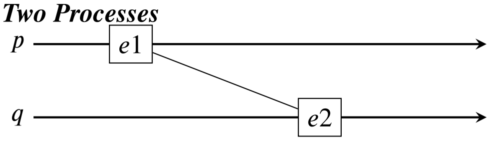
2.4 Timing Assumptions

- **Asynchronous model** (*Logical Timing*)
  - ***One Process***



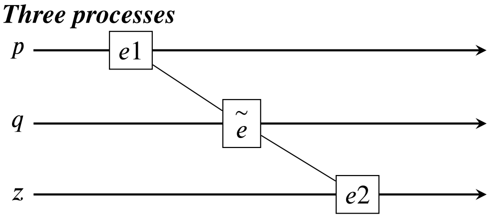
If $e2$ happened after $e1$ in one process, we know the sequence of events.

  - ***Two Processes***



If we know that $e1$ caused $e2$, we know that $e2$ happened after $e1$.

  - ***Three processes***



Transitivity holds across processes, so if $e1$ caused $\tilde{e}$ which cause $e2$, $e2$ happened after $e1$.

- Other time models exist

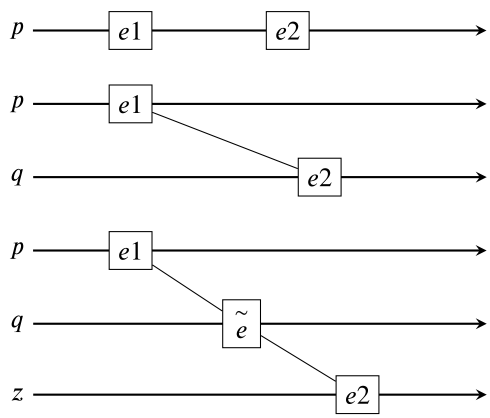
3 Timing Assumptions - March 3, 2020

3.1 Asynchronous System

Logical clock creates a logical time

- Each process p keeps a logical clock lp (initially 0)
- When an event e on p occurs, then $lp \leftarrow lp + 1$
- When p sends a message m to q, then p attaches a timestamp $ts(m) = lp$ to m
- When p receives a message m' with $ts(m')$, then p sets $lp \leftarrow max\{lp, ts(m')\} + 1$

3.1.1 Happens-before relation



In each of these we can say that $e1$ happens before $e2$

3.1.2 Lemma

$e1$ occurs at p at lp
$e2$ occurs at q at lq
$\Rightarrow e1 \rightarrow e2$, then $lp < lq$, but not the other way round!

3.2 Synchronous System

<u>**EITHER:**</u>

- Assume every process has access to a real-time clock (***RTC***)

<u>**OR:**</u>

- Synchronous computation (bounds on computation time)
- Synchonous communication (bounds on message-transmission time)

CAREFUL! when synchrony, assumptions are needed for safety properties

3.3 Partially Synchronous Model

- Synchronous most of the time
- When asynchronous, must not violate safety
  Formally captured by abstraction of an eventually synchonous system.
- Initial period of asynchrony
- After some point in time (unknown to algorithm), system is synchonous

NOTE: Abstract model will remain synchronous forever after sync-point. In practice, periods of synchrony and asynchrony alternate.

3.4 Abstracting Time

DEFINITION: Perfect Failure Detecture $\mathbb{P}$
EVENT: $\langle \mathbb{P}.\mathit{Crash} \mid p \rangle$ denotes that process p has crashed.
PROPERTIES:

STRONG COMPLETENESS:
Eventually every process that has crashed is detected by all correct processes.

STRONG ACCURACY:
For any process p, if p detects that q crashed, then q has crashed.

Formally, all processes are either alive forever or they crash and stop.
Suppose a notion of time in $\mathbb{N}$:

$C : \mathbb{N} \to \Pi$, $C(t)$ denotes the processes that are live at time t.
$F : \mathbb{N} \to \Pi$, $F(t)$ denotes the proceses that are faulty (crashed) at time t.

$p \in F(t)$, then $\forall t' \geq t : \ p \in F(t')$ (crashes are irreversible)

$\mathbb{F} \ = \ \bigcup_{t \geq 0} F(t)$, set of all faulty processes
$\mathbb{C} \ = \ \Pi \setminus \mathbb{F}$, set of all correct processes

Strong Completeness:
$\exists t : \forall p \in \mathbb{F}, \forall q \in \mathbb{C} : \exists t' \geq t : \langle \mathbb{P}.\mathit{Crash} \mid p \rangle$ occurs on process q at time t'.
Strong Accuracy:
$\forall q \in \mathbb{C}$ if $\langle \mathbb{P}.\mathit{Crash} \mid p \rangle$ occurs on process q at time t then $p \in F(t)$.

3.4.1 Implementing $\mathbb{P}$

Initialization:
start timer Δ
alive $\leftarrow \Pi$
detected $\leftarrow \emptyset$

upon timeout do for all $p \in \Pi$ do
if $p \notin alive \wedge p \notin detected$ then trigger $\langle \mathbb{P}.Crash \mid p \rangle$
detected $\leftarrow$ detected $cup\{p\}$
start timer with Δ
alive $\leftarrow \emptyset$
send msg [PING] to all $p \in \Pi$

upon receive msg. [PING] from p do
send msg [PONG] to p

upon receiving [PONG] from p do
alive $\leftarrow$ alive $\cup\{p\}$

DEFINITION: Leader Election
EVENT: $\langle le.leader \mid p \rangle$, elects p to be leader
PROPERTIES (Eventual Leadership):
Eventually, some process l is elected leader by every correct process

ACCURACY:
If a process is elected leader then all previously elected leaders have crashed.

DEFINITION: Eventually Perfect Failure Detector
EVENTS:
$\langle \diamond\mathbb{P}.Suspect \mid p \rangle$, process p is suspected.
$\langle \diamond\mathbb{P}.Restore \mid p \rangle$, process p is thought to be alive.

PROPERTIES

STRONG COMPLETENESS:
Eventually, every process that has crashed is suspected by every correct process

EVENTUAL STRONG ACCURACY:
Eventually, every process that has crashed is suspected permanently by every correct process.

Model	Processes	Timing	
fail-stop	crash-stop	synchronous	$\langle \mathbb{P} \rangle$
fail-noisy	crash-stop	partially synchronous	$\langle \diamond\mathbb{P} \rangle$, $N > 2F$
fail-silent	crash-stop	asynchronous	$N > 2F$

4 System Models - March 11, 2020

CGR11	processes	timing assumption	assumption	other names
fail-stop	crash	$\mathbb{P}$	-	synchronous
fail-noisy	crash	$\diamond\mathbb{P}, \Omega$	$N > 2\mathbb{F}$	eventually synchronous
fail-silent	crash	-	$N > 2\mathbb{F}$	asynchronous
fail-silent randomized	crash	-	$N > 2\mathbb{F}$, randomness	asynchronous randomized
fail-revocery	crash-recovery			
fail-arbitrary-noisy	fail-arbitrary	Byz. leader detector	$N > 3\mathbb{F}$	"BFT" (PBFT)
fail-arbitrary-silent	-"-	-	$N > 3\mathbb{F}$	asynchronous Byzantine
fail-arbitrary randomized	BYZANTINE	-	$N > 3\mathbb{F}$	randomized Byzantine fault model

4.1 Chapter 3: Distributed Storage and Shared Memory

- Storage abstraction provided by distributed processes
- Here: simplified model where $\Pi = \mathbb{C}$, designated processes act as writing/reading clients

4.1.1 Main Abstraction

Shared Read-/Write-Register:

Operations:

read() $\rightarrow v$
write(v) $\rightarrow$ ACK

Sequential implementations:

state:
val, initially NULL
function read()
return val
function write(v)
val ← v   return ACK

Module Register (r):

Events:

$\langle r,$ READ$\rangle$
$\langle r,$ READRESP | $v\rangle$
$\langle r,$ WRITE | $v\rangle$

$\langle r,$ WRITERESP$\rangle$ (acknowledgement)

Liveness:
every operation eventually returns a response

Safety:
Every read operation returns the value written by the "last write" operation, when no concurrent operation.

Operations:
every operation modeled by two events
- Invocation event
- Completion event

4.1.2 Definition (Preceding)

Operation o_1 precedes operation o_2 if o_1 completes before o_2 is invoked.

4.1.3 Definition (Sequential)

Operations o_1 and o_2 are sequential if o_1 precedes o_2 or o_2 precedes o_1.

4.1.4 Definition (Concurrent)

Operations o_1 and o_2 are concurrent if they are not sequential.

4.1.5 Register Example

Register Domain
- binary register $\{0, 1\}$
- multi-valued register

Register Types
- (1,1) 1 writer, 1 reader (SRSW register (single-writer-single-reader))
- (1,N) 1 writer, N readers (MRSW register (multi-writer-single-reader))
- (N,N) N writers, N readers (MRMW register (multi-writer-multi-reader))

Semantics:
Safe:
A read() not concurrent with a write returns the value written by the most recent write() operation (a safe register can return any object from the domain)

4.1.6 An unsafe register

Implement a multi-valued register (mvr) from (many) binary registers.
Domain $\mathbb{D} = [0, 11]$
4 binary registers $br - 0, br - 1, br - 2, br - 3$
Notation mit function calls:
br-0.write(1)
mvr.read()

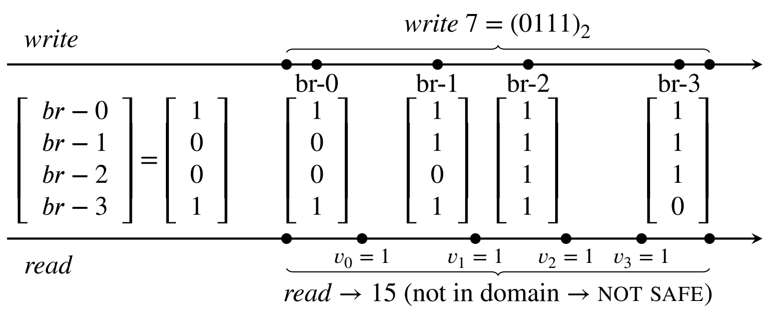
MVR

state
br-0, br-1, br-2, br-3 initially 0

function mvr.write(v)
$(b_3 b_2 b_1 b_0)_2 \leftarrow v$
for $i \leftarrow 0, ..., 3$ do
$br\text{-}i$.write(b_i)
return ACK

function mvr.read()
for $i \leftarrow 0, ..., 3$ do
$v_i \leftarrow br\text{-}i$.read()
return $(v_3 v_2 v_1 v_0)_2 z$

Execution: *initially mvr stores 9* = $(1001)_2$



Regular Semantics:

Only single-writer registers

Safety:

A read(), not concurrent with a write(), returns the most recently written value.
Otherwise read() returns the most recently written value or the concurrently written values.

Atomic Semantics: *(assume values written are unique)*

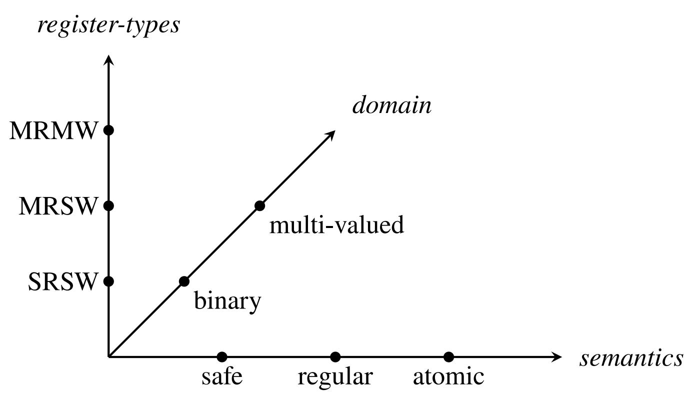
Safety:

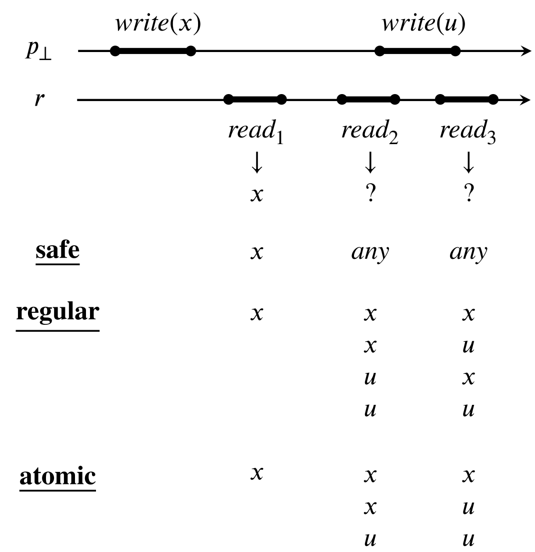
(1) -"-

(2) If read() → v and a subsequent read() → w, then write(v) preceds write (w) or write(v) is concurrent to write(w).

Alternative characterization with linearizability

Collaps each operation to its linearization point, which must occur between invocation and response, and values returned satisfy the sequential specifications of the object.





4.1.7 Implementation of an (1,N) Regualar Register in Fail-Silent Mode

Majority-Voting

state:
val
ts
wts ← 0 //writer only

function rr.write(v)
wts ← *wts* + *1*
send message [WRITE, wts, v] to all $p \in \Pi$
wait for message [WRITE-ACK] from $> N/2$ processors
return ACK

upon *receive* message [WRITE, ts', v] from w do
$(val, ts) \leftarrow (v, ts')$
send message [WRITE-ACK] to w

upon *receive* message [READ] from r do
send message [READVAL, ts, val] to r

function rr.read()
send message [READ] to all $p \in \Pi$
wait for message [READVAL, ts', val'] from $> N/2$ processors
let v be the value val' among the received pairs with the highest timestamp
return v

5 Implementations of Registers - March 18, 2020

5.1 REGULAR register implementation in *fail-stop* model

- synchronous
- Perfect Failure Detector $\mathbb{P}$

(1,N) regular register (*onrr*)

tikz

Init:
$val \leftarrow 1$
$correct \leftarrow \Pi$

upon $\langle$*onrr-Write* | $v\rangle$ do
send message [WRITE, v] to all $p \in \Pi$ //*best-effort broadcast*
wait for receiving message [ACK] from all processes in *correct*
trigger $\langle$*onrr-WriteResponse*$\rangle$

upon receive message [WRITE, v'] from process w do
$val \leftarrow v'$
send message [ACK] to w

upon $\langle\mathbb{P}$-*Crash* | $c\rangle$ do
$correct \leftarrow correct \setminus \{c\}$

upon $\langle$*onrr-Read*$\rangle$ do
trigger $\langle$*onrr-ReadReturn* | $val\rangle$

5.2 REGULAR register implementation in *fail-silent* model

- asynchronous

(1,N) regular register with $N > 2F$

Init:
$(ts, val) \leftarrow (o, \perp)$
$wts \leftarrow 0$
$rid \leftarrow 0$

upon $\langle$*onrr-Write* | $v\rangle$ do
$wts \leftarrow wts + 1$
send message [WRITE, wts, v] to all $p \in \Pi$
wait for receiving message [ACK, ts'] s.t. $ts' = wts$ from $> \frac{N}{2}$ processes
trigger $\langle$*onrr-WriteResponse*$\rangle$

upon receive message [WRITE, ts', v'] from process w do
if $ts' > ts$ then
$(ts, val) \leftarrow (ts', v')$
send message [ACK, ts'] to w

upon $\langle$*onrr-Read*$\rangle$ do
$rid \leftarrow rid + 1$
send message [READ, rid] to all processes in Π
wait for receive message [VAL, r, ts', v'] s.t. $r = rid$ from $> \frac{N}{2}$ processes
$\overline{v} \leftarrow$ value v in the message with the highest timestamp ts'
trigger $\langle onrr - ReadReturn \mid \overline{v}\rangle$

upon receiving message [READ, r] from process p do
send message [VAL, r, ts, val] to p

5.3 Example execution

tikz

5.4 Make Algorithm (ABOVE) (Alg. 4.2) ATOMIC

(1,N)-ATOMIC register (*onar*)

upon $\langle onar - Read \rangle$ do
 $rid \leftarrow rid + 1$
 send message [READ, rid] to all processes in Π
 wait for receive message [VAL, r, ts', v'] s.t. $r = rid$ from > $\frac{N}{2}$ processes
 $(rts, rval) \leftarrow ts', v'$-pair from VAL message with highest ts'
 send message [RWRITE, rts, $rval$] to all $p \in \Pi$
 wait for receiving message [RACK, rts'] s.t. $rts' = rts$ from > $\frac{N}{2}$ processes
 trigger $\langle$*onrr-ReadResponse* | $rval\rangle$

upon receive message [RWRITE, ts', val' from r do
 if $ts' > ts$ then
  $(ts, val) \leftarrow (ts', v')$
 send message [RACK, ts'] to r

start: (1,N) REGULAR register
intermediate: (1,1) ATOMIC register
goal: (1,N) ATOMIC register

5.5 From (1,1) ATOMIC to (1,N) ATOMIC register

tikz

Transformation:
implements: (1,N) ATOMIC register (*onar*)
uses: (1,1) ATOMIC register (u^2 : *ooar.i.j*)
 Init: $ts \leftarrow 0$

operation *onar*-WRITE(v) is
 $ts \leftarrow ts + 1$
 for $p \in \Pi$ do
  ooar.p.w-WRITE((ts, v))
 return ACK

operation *onar*-READ() is
 readList ← []
 for $p \in \Pi$ do
  readList[p] ← ooar.self.p-READ()
 (maxts, maxval) ← highest(readList)
 for $p \in \Pi$ do
  ooar.p.self-WRITE((maxts, maxval))
 return maxval

5.6 From (1,N) ATOMIC to (N,N) ATOMIC register

tikz

- writer uses highest timestamp that it reads
- timestamps become (ts, *index*) duples (index of process)

5.7 Register Implementation in BYZANTINE Model ($N > 3F$)

tikz01
tikz02

- relax the specification
- introduce data authentication using digital signature

6 Byzantine Distributed Storage - March 25, 2020

6.1 Specification

tikz

6.1.1 $(1, N)$-REGULAR Register

IN: $\langle Read \rangle$ OUT: $\langle ReadReturn \mid v \rangle$
IN: $\langle Wirte \mid v \rangle$ OUT: $\langle WriteReturn \rangle$

- **Termination:**
  Every operation eventually terminates
- **Validity:**
  every *read* returns either the concurrently written value or the most recently written value

Init:
$(ts, val, sig) \leftarrow (\bot, \bot, \bot)$
$wts \leftarrow 0$

upon *write*(v) do
$wts \leftarrow wts + 1$
$\sigma \leftarrow sign(\text{WRITE}\|wts\|v)$
send message [WRITE, wts, v, σ] to all processes
wait for messages [ACK, ts'] s.t. $ts' = wts$ from $> \frac{n+f}{2}$ processes (BYZANTINE quorum)

upon receive message [WRITE, ts', v', σ'] do
if $ts' > ts$ then
$(ts, val, \sigma) \leftarrow (ts', v', \sigma')$
send message [ACK, ts'] to writer w

upon *read()* do
send message [READ] to all processes
wait for message [VALUE, ts', v', σ'] from $> \frac{n+f}{2}$ with *verifySign*(σ', WRITE $\|ts'\|v'$) = TRUE
let v be the received value v' from message with the highest timestamp ts'
return v

upon [READ] ...

- **Termination**
  n replicas and f faulty $\Rightarrow \geq n-f$ responses
  <u>Show:</u> $n-f > \frac{n+f}{2}$, if $n > 3f$

$$n - f > \frac{n+f}{2}, \textit{ if } n > 3f \ldots$$

- **Validity:**
  tikz
- **Safety:**
  Q_w quorum used by writer
  Q_r quorum used by reader
  $\mid Q_w \mid > \frac{n+f}{2}$
  $\mid Q_r \mid > \frac{n+f}{2}$
  <u>Show:</u> There exists at least one correct process in $Q_w \cap Q_r$ Suppose not:
  Count number of distinct correct processes in $Q_w \cap Q_r$
  number is $\geq \mid Q_w \mid -f + \mid Q_r \mid -f > \frac{n+f}{2} - f + \frac{n-f}{2} - f = n - f$
  $\Rightarrow$ So there exists at least one correct process in $Q_w \cap Q_r$

6.1.2 Can we implement this w/o signatures?

tikz

- could return c default value
- this idea can be turned into for emulating < <u>safe</u> BYZANTINE registers assuming $n > 4f$ processes
- (2-round protocol with $n > 3f$ exists...)

6.1.3 Practical Leaderless Replication

- clients send request directly to replicas
- possibly via some coordinator
- tikz
- Consistency of the stored data
  - auxiliaty service within storage system will eliminate differences - "*anti-entropy*"
  - *read-repair* by reading clients that detect inconsistent clients
- Quorums for reading and writing
  r - replicas for reading
  w - replicas for writing
  so far: r = w > $\frac{n}{2}$
  $r = 1, w = n$ read-one write-all
  $r = n, w = 1$ read-all write-one
- **Strong Consistency** iff $r + w > n$
  Some systems use $r + w \leq n$

6.2 Key-Value Store (KVS)

tikz

- Replica set usually differs for each pair
- Semantics is not formally specified
  … too expensive
  … atomic problematic because the reader would write
- Conflicts?
  - resolved using heuristic methods
  - last-write wins policy
    … data loss is possible
- Practical system ensure usually a notion of eventual consistency
- Some systems let clients merge conflicting written values
  - often easy for applications
  - very difficult in general
  … AMAZON Dynamo
    tikz

7 CHAPTER 4: Reliable Broadcast - April 01, 2020

7.1 Definition

(IN) $\langle$BROADCAST | $m\rangle$ "*broadcast a message m*"
(OUT) $\langle$DELIVER | $p, m\rangle$ "*delivers a message m from sender p*"

tikz

7.2 Best-Effort Broadcast (UNRELIABLE)

7.2.1 Definition (VALIDITY):

If a *correct* process broadcasts a message m, then every *correct* process eventuall delivers m.

7.2.2 Definition (NO DUPLICATION):

No message is delivered more than once.

7.2.3 Definition (NO CREATION):

If a process delivers a message m with sender s, then m was previously broadcasted by process s.

7.3 Messages are UNIQUE

How do we ensure this in practice? . . .

7.4 Reliable Broadcast

7.4.1 Definition:

Same properties as (unreliable) best-effort broadcast.

Agreement:
If a message m is delivered by a *correct* process then every *correct* process eventually delivers m.

tikz

7.5 EAGER Reliable Broadcast

Init: *delivered* $\leftarrow \emptyset$

upon $\langle rb\text{-}\mathrm{BROADCAST} \mid m\rangle$ do
  send message [DATA, *self*, m] to all $p \in \Pi$

upon receive message [DATA, s, m] from q do
  if $m \notin delivered$ then
    delivered $\cup \leftarrow \{m\}$
    trigger $\langle rb\text{-}\mathrm{DELIVER} \mid s, m\rangle$
    send message [DATA, s, m] to all $p \in \Pi$

7.5.1 Implementation:

tikz

A process that crashes:
All modules crash simultaneously.

7.5.2 How adequate is this (basic) reliable broadcast?

tikz

7.6 UNIFORM Reliable Broadcast

7.6.1 Definition:

Same properties as regular reliable broadcast.

Uniform Agreement:
If a process delivers a message m, then every correct process eventually delivers m.

tikz

7.6.2 Implementation of URB in async. netw. with f crashes and $N > 2f$

$pending \leftarrow \emptyset$
$ack[] \leftarrow []$
$delivered \leftarrow \emptyset$
upon $\langle urb\text{-}\textsc{Broadcast} \mid m \rangle$ do
  $pending \cup \leftarrow \{(self, m)\}$
  send message [DATA, $self$, m] to all $p \in \Pi$

upon receive message [DATA, s, m] from q do
  $ack[m] \leftarrow ack[m] \cup \{q\}$
  if $(s, m) \notin pending$ then
    $pending \leftarrow \{(s, m)\}$
    send message [DATA, s, m] to all $p \in \Pi$

upon $\exists (s, m) \in pending \&\& m \notin delivered$ s.t. $\mid ack[m] \mid > \frac{N}{2}$ do
  $delivered \cup \leftarrow \{m\}$
  trigger $\langle urb\text{-}\textsc{Deliver} \mid s, m \rangle$

tikz

- If q delivers m, then it has received the DATA messages from $> \frac{N}{2}$ processes
- Since $f < \frac{n}{2}$, at least one DATA message was sent by a correct process, this process has sent m ro all other processes
- All correct processes eventually send DATA message containing m
- all correct processes eventually deliver m

7.7 Order

tikz

7.7.1 FIFO-Order Broadcast

per-sender order

Interface
Same as reliable broadcast

Properties
- Validity
- No Duplication
- No creation
- Agreement
- FIFO-Order (**if** bc. $m_1 \rightarrow m_2$ **then** del. $m_1 \rightarrow m_2$)
  If process broadcasts m_1 and subsequently broadcasts m_2, then no process delivers m_2 unless it has also delivered m_1 before.

Implementation

- Each process adds a (local) sequence number to every payload message
- For each sender, every receiver delivers payload messages according to the sequence number (requiring buffering)

7.7.2 Causal Order (broadcast):

tikz

Prevent such violations of causal order:

- keep track of complete history of past delivered messages

tikz

8 CHAPTER 6: Total-Order Broadcast - 08.04.2020

8.1 Notion of Causality

8.1.1 Events

- BROADCAST(m)
- DELIVER(m)

8.1.2 Causality Relation

$m_1 \rightarrow m_2$
when one of the 3 conditions apply:

1. process p broadcasts m_1 before it broadcasts m_2
2. some process delivers m_1 and later broadcasts m_2
3. some message m_3 exists s.t. $m_1 \rightarrow m_3$ and $m_3 \rightarrow m_2$

8.2 Causal-Order Broadcast using *Vector Clocks*

<u>Init:</u>
$V \leftarrow [0]^n$
$lsn \leftarrow 0$
$pending \leftarrow \emptyset$

<u>upon</u> $\langle crb - \text{BROADCAST} \mid m\rangle$ <u>do</u>
$w \leftarrow V \quad V[rank(self)] \leftarrow lsn$
$lsn \leftarrow lsn + 1$
<u>trigger</u> $\langle rb - \text{BROADCAST} \mid [W, m]\rangle$

<u>upon</u> $\langle rb - \text{DELIVER} \mid p, [W, m]\rangle$ <u>do</u>
$pending \leftarrow pending \cup \{[W, m]\}$
<u>if</u> $\exists(\overline{p}, \overline{W}, \overline{m} \in pending \wedge \overline{W} \leq V$ <u>then</u>
$pending \leftarrow pending \setminus \{[\overline{p}, \overline{W}, \overline{m}]\}$
$V[rank(\overline{p})] \leftarrow V[rank(\overline{p})] + 1$
trigger $\langle crb - \text{DELIVER} \mid \overline{p}, \overline{m}\rangle$

8.2.1 Example

tikz

8.3 Total-Order Broadcast

…is a reliable broadcast with the following additional <u>total-order property</u>:

For two messages m_1 and m_2 such that processes p and q have both delivered m_1 and m_2, then p delivers m_1 before m_2 iff q delivers m_1 before m_2.
Any delivery sequence of a process is a prefix of another process which has more messages delivered.

8.4 Consensus

8.4.1 Events

(IN) $\langle c - \text{PROPOSE} \mid v \rangle$ - ... proposes v ...
(OUT) $\langle c - \text{DECIDE} \mid v \rangle$ - ... decides for v ...

8.4.2 Properties

Termination:
Every correct process eventually decides some value.

Validity:
If a process decides a value v, then v was proposed by some process

Integrity:
A process decides at most once.

Agreement:
No two (correct) processes decide different values.

8.5 From Consensus to Total-Order Broadcast

Init:
$unordered \leftarrow \emptyset$
$delivered \leftarrow \emptyset$
$r \leftarrow 1$
$wait \leftarrow$ FALSE ... consensus is running?

upon $\langle tob$ – BROADCAST | $m\rangle$ do
trigger $\langle rb$ – BROADCAST | $m\rangle$

upon $\langle rb$ – DELIVER | $p, m\rangle$ do
if $m \notin delivered$ then
$unordered \leftarrow unordered \cup \{(p, m)\}$

upon $unordered \neq \emptyset \wedge \neg wait$ do
$wait \leftarrow$ TRUE
initialise consensus instance $c.r$
trigger $\langle c.r$ – PROPOSE | $unordered\rangle$

upon $\langle c.r$ – DECIDE | $d\rangle$ do
for $(s, m) \in d$ in some fixed order do
trigger $\langle tob$ – DELIVER | $s, m\rangle$
$delivered \leftarrow delivered \cup \{m\}$
$unordered \leftarrow unordered \setminus \{(s, m)\}$
$r \leftarrow r + 1$
$wait \leftarrow$ FALSE

8.6 From Total-Order Broadcast to Consensus

tikz

8.7 State-Machine Replication

tikz

8.7.1 Events

(IN) $\langle rsm$ – EXECUTE | $command\rangle$ - ... proposes v ...
(OUT) $\langle rsm$ – OUTPUT | $response\rangle$ - ... decides for v ...

8.7.2 Properties

Termination:
If a correct process executes a command, then the process eventually also outputs a response (for that command).

Agreement:
All correct processes output the same sequence of responses

8.8 From Total-Order Broadcast to Replicated State-Machine

Init:
$state \leftarrow \perp$

upon $\langle rsm - \text{EXECUTE} \mid cmd \rangle$ do
trigger $\langle tob - \text{BROADCAST} \mid cmd \rangle$

upon $\langle tob - \text{DELIVER} \mid p, cmd \rangle$ do
$(r, state) \leftarrow F(cmd, state)$
trigger $\langle rsm - \text{OUTPUT} \mid r \rangle$

9 CONSENSUS - April 22, 2020

9.1 Definition

9.1.1 Events

(IN) propose (v)
(OUT) decide (v)

9.1.2 Properties

- **Termination:**
  ... terminates
- **Validity:**
  decision value v was proposed
- **Integrity:**
  decide at most once
- **Agreement:**
  No two processes decide different values

9.2 Flooding Uniform CONSENSUS

<u>init:</u>
  correct $\leftarrow \Pi$
  round $\leftarrow 1$
  decision $\leftarrow \bot$
  proposals $\leftarrow \{\}$

<u>upon</u> *uc*- PROPOSE(v) <u>do</u>
  $proposals \leftarrow proposals \cup \{v\}$
  trigger $\langle beb\text{-BROADCAST} \mid [\text{PROP}, 1, proposals]\rangle$

<u>upon</u> $\langle \mathbb{P} - \text{CRASH} \mid p\rangle$ <u>do</u>
  $correct \leftarrow correct \setminus \{p\}$

<u>upon</u> $\langle beb\text{-DELIVER} \mid p, [\text{PROP}, r, ps]\rangle$ <u>such that</u> $r = round$ <u>do</u>
  $received \leftarrow received \cup \{p\}$
  $proposals \leftarrow proposals \cup ps$

<u>upon</u> $correct \subseteq received$ <u>do</u>
  <u>if</u> $round < N$ <u>then</u>
    $round \leftarrow round + 1$
    $received \leftarrow \emptyset$
    trigger $\langle beb\text{-BROADCAST} \mid [\text{PROP}, round, proposals]\rangle$
  <u>else</u> // $round = N$
    $decision \leftarrow min(proposals)$
    DECIDE($decision$)

9.3 CONSENSUS Protocol with Eventual Synchrony

- Consensus mechanism inside Paxos, Viewstamp Replication, ZooKeeper, Raff
- more complex than the *fail-stop* protocol (with $\mathbb{P}$) because it uses Ω ($\diamond\mathbb{P}$)

* **Leader-Driven CONSENSUS**
tikz
- every epoch has a leader

9.3.1 Epoch-Change

Event: $\langle$*ec-Start Epoch* | $ts,\ l$

Properties:

Monotonicity:
If a process start epoch $(ts,\ l)$ and subsequently start epoch $(ts',\ l')$, then $ts' > ts$

Consistency:
If a process starts epoch $(ts,\ l)$ and also another process starts epoch $(ts',\ l')$ and $ts = ts'$ then $l = l'$

Leadership:
- There is a last epoch that is started at every process
- the leader of this last epoch is correct

9.3.2 Epoch-CONSENSUS

Events:

IN $\langle$*ep*-PROPOSE | $v\rangle$ // only leader
OUT $\langle$*ep*-DECIDE | $v\rangle$
IN $\langle$*ep*-ABORT$\rangle$
OUT $\langle$*ep*-ABORTED | *state*$\rangle$

Properties:

Validity:
If a process *ep*-DECIDES v, then v was *ep*-PROPOSED (by the leader) in some epoch $ts' \leq ts$

Agreement:
(as before)

Integrity:
(as before)

Termination:
If the leader l of epoch ts is correct, and w process aborts, then every correct process eventually *ep*-DECIDES

Lock-In:
If a process has *ep*-DECIDED v in epoch $ts' < ts$, then w

Abort Completeness: …

9.4 Leader-Driven CONSENSUS

Init:
$val \leftarrow \perp$
$proposed \leftarrow F$
$decided \leftarrow F$
Init *ep*-CONSENSUS with $(0, l_0)$ $[ep.0]$
$(ets, l) \leftarrow (0, l_0)$

upon $\langle uc$-PROPOSE $\mid v\rangle$ do
$val \leftarrow v$

upon $\langle ec$-STARTEPOCH $\mid nts, nl\rangle$ do
$(newts, newl) \leftarrow (nts, nl)$
abort the current epoch (ets, l) …

upon $\langle ep.ts$-ABORTED $\mid state\rangle$
$(ets, l) \leftarrow (newts, newl)$
Initialize *ep*-CONSENSUS with *state* and (ets, l)
upon $val \neq \perp \wedge l = self \wedge proposed = F$ do
$proposed \leftarrow T$
trigger $\langle ep.ets$-PROPOSE $\mid val\rangle$

upon $\langle ep.ts$-DECIDE $\mid v\rangle$ such that $ts = ets$ do
if $\neq decided$ then
$decided \leftarrow T$
trigger $\langle uc$-DECIDE $\mid v\rangle$

9.4.1 Why Agreement?

- if in same epoch, then *ep*-CONSENSUS agreement property
- otherwise, locking property ensures agreement

10 10th Lecture - April 29, 2020

10.1 sub

10.1.1 subsub

11 Byzantine Consensus - May 6, 2020

11.1 Weak Byzantine Consensus

11.1.1 Events

- propose(v)
- decide(v)

11.1.2 Properties

- **Termination**
  Eventually correct process eventually decides
- **Weak Validity**
  If all proceeses are correct, and all propose v, then no correct process decides a value different from v; if all processes are correct, and some correct process decides v, then v was proposed by some process.
- **Agreement**
  No two correct processes decide on different values

11.2 Strong Byzantine Consensus

Same as waek Byzantine consensus, but with...

11.2.1 Properties

- **Strong Validity**
  If all correct processes propose the same value v, then every correct process decides v; otherwise, a correct process must decide a special value $\square$ (*blank*), or a value proposed by a correct process.

11.3 *Special Case:* Binary Consensus

11.3.1 Properties

- **Strong Validity**
  The decision value was proposed by a correct process.

Simplification because there are only two decision values.

11.4 Fault-Tolerance

Byzantine Consensus requires a strong majority of correct processes.

11.4.1 Theorem

Every strong Byzantine Consensus protocol tolerates at most $f < \frac{N}{3}$ faulty processes.

11.4.2 Proof

Suppose not.
Consider f = 1. Then there is a Byzantine consensus protocol for N = 3 processes.

Execution A

tikz

Execution B

tikz

Execution C

tikz

One can extend this argument to $N > 3$ processes, as long as $N \leq 3f$.

11.4.3 Intuition

- processes can wait for at most $N - f$ others
- among these $N - f$, a majority must be correct

11.5 Reliable Broadcasts with Byzantine Faults

11.5.1 Events

- broadcast(m)
- deliver(m)

Byzantine broadcasts (here) can broadcast and deliver one payload message; with identified sender s (to broadcast multiple payloads, run many such protocols in parallel).

11.5.2 Properties

- **Validity**
  If a correct sneder s broadcasts m, then every correct process eventually delivers m
- **No Duplication**
- **Integrity**
- **Consistency**
  If a correct process delivers m, and another correct process delivers m', then $m = m'$.

11.6 Signed Echo Broadcast

tikz

Init: *sentecho* $\leftarrow F$
sentfinal $\leftarrow F$

upon *cb-broadcast*(n) do
send messgae [SEND, m] to all processes

upon receiving message [SEND, m] from s and $\neq$*sentecho* do
$\sigma \leftarrow sign(\text{ECHO} \,\|identifier\|m)$
send message [ECHO, m, σ'] from $> \frac{N+f}{2}$ distinct processes and $\neg$*sentfinal* do
$\Sigma \leftarrow$ list of received sig. σ
send message [FINAL, m, Σ]
sentfinal $\leftarrow T$

upon receive message [FINAL, m, Σ] do
if verify that Σ contains $> \frac{N+f}{2}$ signatures on ECHO $\|identifier\|m$ **then**
deliver(m)

11.6.1 Why Consistent?

some p has delivered m, then it has $> \frac{N+f}{2}$ sig. on m
some p' has delivered m', then it has $> \frac{N+f}{2}$ sig. on m'
how many distinct correct processes have signed?
p has seen $> \frac{N-f}{2}$ signatures (on m) from correct processes
p' has seen $> \frac{N-f}{2}$ signatures (on m') from correct processes
This means, p and p' have together seen $> N - f$ signatures from correct processes.
But there are only $N - f$ correct processes
$\Rightarrow m = m'$

This protocol uses signatures.
An alternative protocol uses pt-to-pt messages, where signatures are replaced by messages:

tikz

11.7 Byzantine Reliable Broadcasts (BRACHA)

11.7.1 Definition

A protocol for Byzantine Reliable Broadcast is a Byzantine consistent broadcast that satisfies also:

- **Totality**
  If a correct process delivers some message, then every correct process eventually delivers some message.

Totality + Consistency = Reliability

11.7.2 Implementation (by BRACHA)

Authenticator Double-Echo Broadcast

tikz

- All correct processes sned [READY, m] upon receiving $> \frac{N+f}{2}$ ECHO messages with m
- All correct processes **also** send [READY, m] after receiving $> f$ READY messages with m

$\Rightarrow$ and a correct process delivers m upon receiving $> 2f$ messages [READY, m]

11.7.3 Why does this ensure totality?

Suppose p that has delivered m
$\Rightarrow$ p has received more than $2f$ READY messages with m
$\Rightarrow$ more than f READY messages from correct processes
$\Rightarrow$ every correct process eventually receives $> f$ READY messages with m
$\Rightarrow$ every correct process eventually sends READY with m
$\Rightarrow$ every correct process eventually receives $> 2f$ READY, because $N - f > 2f$ assuming that $N > 3f$

11.7.4 How can we croadcast many payload messages?

- Run one instance of basic broadcast for each process as a sender
- After one instance delivers a message m, just start another one

$\Rightarrow$ Byzantine $\{{}^{Reliable}_{Consistent}\}$ Broadcast Channel

11.8 Byzantine Total Order Broadcasts

- Analogous to model with crash failures
- In principle, implement from one consensus instance for each round, as before
- In practice, more complex protocols are used which are efficient.

12 12th Lecture - May 13, 2020

12.1 Group Membership

- Group composition changes
- Faults occur ... process should be removed
- New processes may join (also processes which have previously failed and were "born again")

12.1.1 Formalization of group membership

- **Event (Indication):**
  $\langle gm\text{-}\textsc{View} \mid V \rangle$, $V = (id, M)$, $M \subseteq \mathbb{P}$ is a set of processes
  "process installs view V"
- **Properties:**
  - *Monotonity:*
    If process p installs view $V = (id, M)$ and later installs another view $V' = (id', M')$ then $id' > id$
  - *Agreement:*
    If process p installs view (id, M) and process q installs view (id', M') and $id = id'$, then $M = M'$
  - *Completeness:*
    If a process p crashes (or if p is added), then every correct eventually instakks a view $V = (id, M)$ s.t. $p \notin M$ (respectively, $p \in M$)
  - *Accuracy:*
    If some process installs a view $V = (id, M)$ and :
    - $p \notin M$ (but p was included in an earlier view), then p has crashed
    - $p \in M$ (but p was not included in an earlier view), then p was added

12.1.2 Implementation

1) Implementation with CONSENSUS (see CGR11)
2) Implementation from a failure detector (in practice)

12.2 Group Communication Systems (View Synchrony)

tikz

12.2.1 Popular Implementations

1. JGroups (Java implementations)
2. ISIS, ...

12.3 Probabilistic Broadcast

1. Quorum-based protocols do not scale to large numbers of processes (only $\approx 100s$ instead of $1000s$)
2. Reason is that consistency implies lower bounds on messages (communcation bits) of $\Omega(N^2)$
3. Probabilistic protocols ensure (agreement, ...) their properties only for a large fraction of nodes

12.3.1 Example Probabilistic Reliable Broadcast

- similar to reliable broadcast with crash failures
- but validity only holds with high probability:
  - VALIDITY: If a process broadcasts a message m, then with probability at least $1 - \epsilon$, every correct process eventually delivers m.

12.3.2 Protocols for Probabilistic Broadcast

Eager Probabilistic Broadcast (PUSH-Style Gossip

Init:
  delivered ← ∅

upon ⟨*pb*-BROADCAST | *m* do
  delivered ← *delivered* ∪{*m*}
  pb-DELIVER(*self*, *m*)
  gossip([GOSSIP, *self*, *m*, *t*]) // t = O(log N)

function *gossip*(*m*)
  for $p \in$ set of k randomly chosen processes in Π **do**
    lossy-SEND messages *m* to *p*

upon receiving [GOSSIP, *p*, *m*, *r*] do
  if $m \notin$ *delivered* then
    delivered ← *delivered* ∪{*m*}
    pb-DELIVER(*p*, *m*)
  if $r > 1$ then
    gossip([GOSSIP, *p*, *m*, $r - 1$])

Efficiency
With $t = O(log\ N)$ and constant fan-out k, this reaches all processes with high probability:

tikz

⇒ Push-only protocol has this limitations; Therefore in practice protocols use PUSH and PULL

PULL-Phase in PUSH-PULL Protocols

- If a process p learns about the existence of some message m that it has not delivered yet, then it "pulls" the message
  → send out a request for the contents of m using PUSH-Style Gossip
  if a process has m, then it sends m in pt-to-pt way to p
- Processes store a fraction of the delivered messages

12.4 Blockchain Consensus

- *Bitcoin* introduced a novel type of consensus protocol (called "NAKAMOTO-Consensus")
- Model without knowledge of participants
- But there are Byzantine participants
- How to "vote" in an open system

12.4.1 Proof of Work

- Moderately hard puzzles
- Puzzles can be produced easily and all puzzles must be equally difficult
- Control difficulty of puzzles (:= T)

tikz

Properties

1) P has no shortcut, essentially must invest $\approx 2^T steps$
2) many instances can be generated easily by V (all of difficulty T)
3) V must be able to check solution t efficiently

PoW by Hashing:

- Cryptographic hash function ($H : \{0,1\}^\star \to \{0,1\}^k$)
- Difficulty is a number of bits (leading) that must be 0
- Challenge: $x \in \{0,1\}^\star$
- solution t must satisfy $H(x\|t) < 2^{k-T}$
- Prover has no other strategy than to try O(2^T) input t

13 13th Lecture - May 21, 2020

13.1 sub

13.1.1 subsub