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)

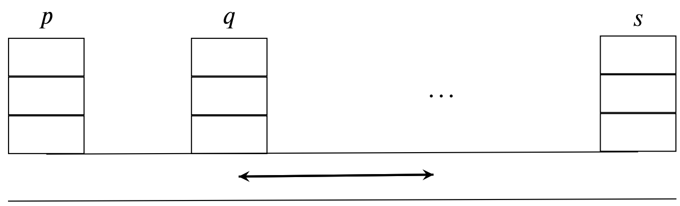
1.1.1 How to make systems tolerate faults

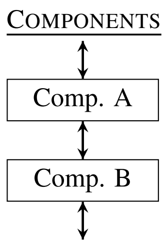
- PREVENTION
- TOLERANCE
  - Replication/Redundancy
  - Recovery
- REMOVAL
- FORECASTING/PREDICTION

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





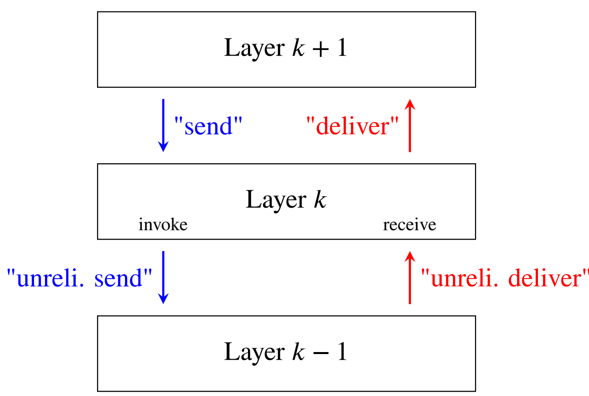
EVENTS for Component c:

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

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

upon $\langle b, domore \mid p \rangle$ 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 ...

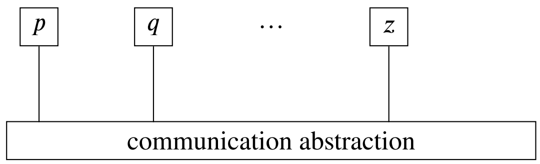
Implementation (asynchronized) JOBHANDLER
State
  $buf \leftarrow \emptyset$
upon $\langle$*jh, handle* | *job*$\rangle$ do
  $buf \leftarrow buf \cup \{job\}$
  trigger $\langle$*jh, confirm* | *job*$\rangle$

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

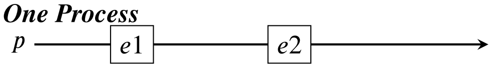
Events:
$\langle sl.send \mid q, m \rangle$
$\langle sl.deliver \mid p, m \rangle$
Properties:
Reliable delivery:
If a correct process sends a message m to a correct process q then q eventually delivers m
No creation:
If process q delivers some m from process p then p has sent m to q
At-most-once delivery: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

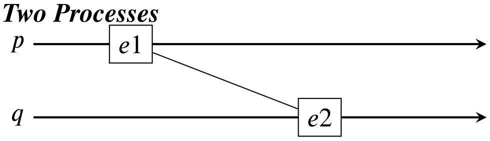
2.4 Timing Assumptions

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



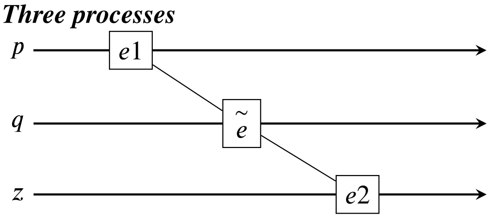
If $e2$ happened after $e1$ in one process, we know the sequence of events.

  - ***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

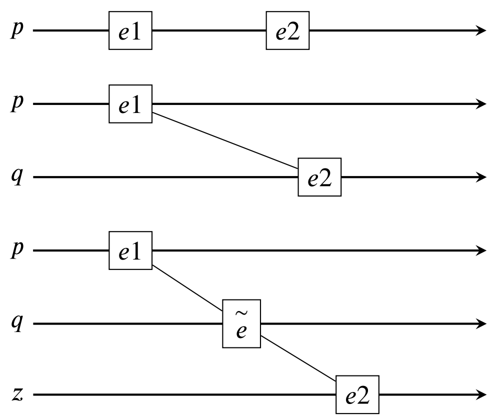
3 Timing Assumptions - March 3, 2020

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

EITHER:

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

OR:

- 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: $\langle \mathbb{P}.Crash \mid p \rangle$ 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:
$\exists t : \forall p \in \mathbb{F}, \forall q \in \mathbb{C} : \exists t' \geq t : \langle \mathbb{P}.Crash \mid p \rangle$ occurs on process q at time t'.
Strong Accuracy:
$\forall q \in \mathbb{C}$ if $\langle \mathbb{P}.Crash \mid p \rangle$ 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 $\leftarrow$ v   return ACK

Module Register (r):

Events:

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

$\langle r,$ 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()

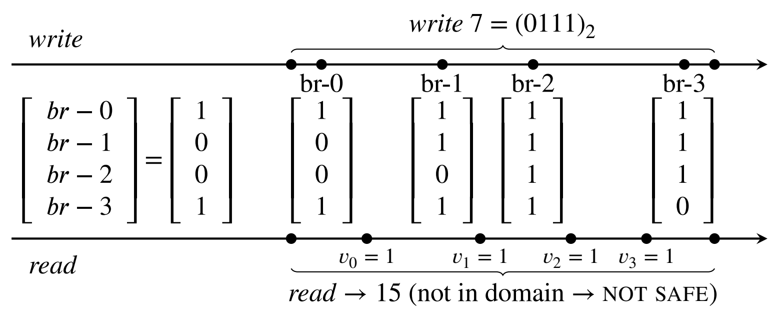
MVR

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)*

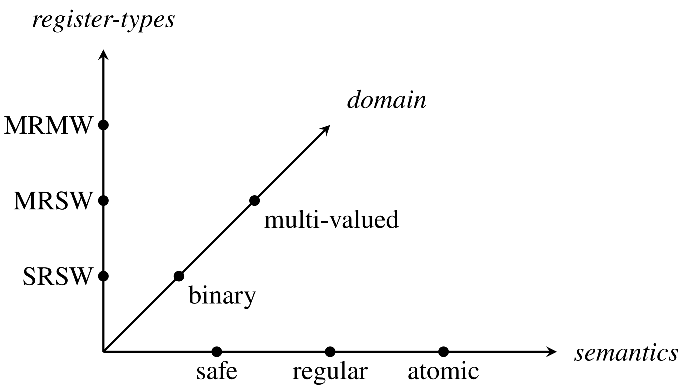
Safety:

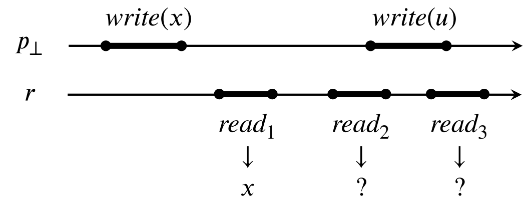
(1) -"-

(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.





	$read_1$	$read_2$	$read_3$
safe	x	*any*	*any*
regular	x	x	x
		x	u
		u	x
		u	u
atomic	x	x	x
		x	u
		u	u

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 5th Lecture - 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 ⟨*onrr-Write* | v⟩ do
  send message [WRITE, v] to all $p \in \Pi$ //*best-effort broadcast*
  wait for receiving message [ACK] from all processes in *correct*
  trigger ⟨*onrr-WriteResponse*⟩

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

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

upon ⟨*onrr-Read*⟩ do
  trigger ⟨*onrr-ReadReturn* | *val*⟩

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* $\mid v\rangle$ 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 6th Lecture - March 19, 2020

6.1 sub