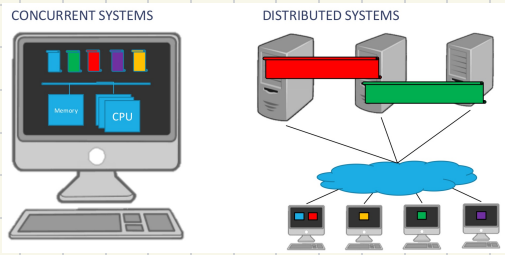
DISTRIBUTED SYSTEMS

LESSON 1

A DISTRIBUTED SYSTEM IS A SET OF AUTONOMOUS ENTITIES, EACH WITH COMPUTATIONAL POWER, CAPABLE OF COMMUNICATING, COORDINATING AND SHARING RESOURCES TO ACHIEVE A COMMON GOAL. IT'S PERCEIVED AS A SINGLE AND COHERENT SYSTEM BY USERS.



THE FAILURE OF A REMOTE NODE CAN NEGATIVELY IMPACT OTHER NODES, HIGHLIGHTING THE COMPLEXITY OF MANAGING FAILURES.

WHY DISTRIBUTED SYSTEMS?

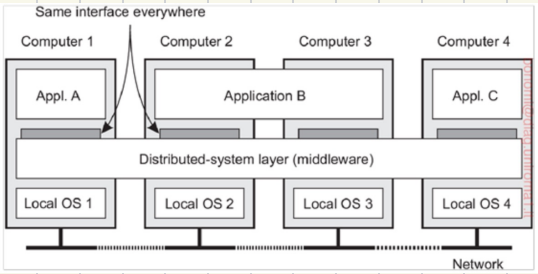
- **INCREASED PERFORMANCE**: TO HANDLE INCREASED USER DEMAND AND REDUCE LATENCY.
- **BUILDING DEPENDABLE SERVICES**: TO COPE WITH FAILURES.

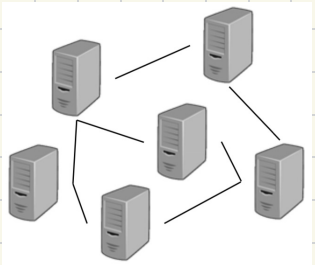
CHARACTERISTICS OF DISTRIBUTED SYSTEMS

- **AUTONOMOUS CALCULATION ELEMENTS**: EACH NODE IS INDEPENDENT AND THERE IS NO GLOBAL CLOCK.
- **SINGLE COHERENT SYSTEM**: THE ENTIRE SYSTEM MUST BEHAVE IN A PREDICTABLE WAY FOR USERS, EVEN IN FAULTS PRESENCE.

MIDDLEWARE AND PRIMARY GOALS

MIDDLEWARE IS ESSENTIAL TO HIDE HARDWARE OR SOFTWARE DIFFERENCES AND ALLOW COMMUNICATION BETWEEN PROCESSES.





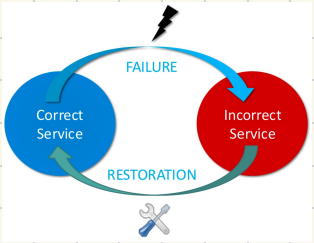
KEY GOALS INCLUDE OVERCOMING LIMITATIONS OF CENTRALIZED SYSTEMS, SUCH AS THE LACK OF A GLOBAL CLOCK AND MANAGING UNPREDICTABLE LATENCIES.

TRENDS

FOUR TRENDS INFLUENCE THE DEVELOPMENT OF DISTRIBUTED SYSTEMS:

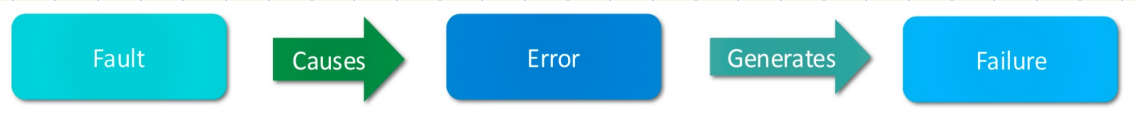
- **PERVASIVE NETWORKING**: A GLOBAL NETWORK WITHOUT TEMPORAL OR SPATIAL LIMITS.
- **MOBILE AND UBIQUITOUS COMPUTING**: DEVICES EVERYWHERE TO SUPPORT USERS ON THE GO
- **DISTRIBUTED MULTIMEDIA**: HANDLING MEDIA SUCH AS AUDIO AND VIDEO.
- **CLOUD COMPUTING**: MODELS SUCH AS IAAS, PAAS AND SAAS.

DEPENDABILITY



DEPENDABILITY REFERS TO THE ABILITY OF A SYSTEM TO AVOID FAILURES AND PROVIDE A SERVICE THAT CAN RIGHTLY BE CONSIDERED TRUSTED.

HAVING A SERVICE FAILURE MEANS THAT EXISTS AT LEAST A DEVIATION OF THE SYSTEM BEHAVIOUR FROM THE CORRECT SERVICE STATE. THE DEVIATION IS CALLED AN ERROR, AND THE HYPOTHESIZED CAUSE OF AN ERROR IS CALLED A FAULT (INTERNAL OR EXTERNAL).



HOW TO DESIGN, DEVELOP AND DEPLOY A DEPENDABLE SYSTEM?

- **AVAILABILITY:** READINESS OF CORRECT SERVICE.
- **RELIABILITY:** CONTINUITY OF CORRECT SERVICE.
- **SAFETY:** ABSENCE OF CATASTROPICH CONSEQUENCES.
- **INTEGRITY:** ABSENCE OF IMPROPER SYSTEM ALTERATIONS.
- **MAINTAINABILITY:** EASE OF REPAIRS AND MODIFICATIONS.

DEPENDABLE DISTRIBUTED SYSTEMS

HETEROGENEITY: IT'S ABOUT THE VARIETY OF COMPONENTS IN A DISTRIBUTED SYSTEM. AS A SOLUTION TO THE PROBLEM OF COMPATIBILITY BEETWEEN DIFFERENT COMPONENTS WE CAN USE A MIDDLEWARE.

OPENNESS: THE CAPABILITY OF A SYSTEM TO BE EXTENDED AND RE-IMPLEMENTED. SERVICES AND INTERFACES MUST BE CLEARLY SPECIFIED TO ENSURE INTEROPERABILITY, PORTABILITY AND FLEXIBILITY.

SECURITY: A DISTRIBUTED SYSTEM MUST PROTECT INFORMATION AND SERVICES AGAINST UNAUTHORIZED ACCESS AND TAMPERING BY GUARANTEEING THEM TO AUTHORIZED USERS. TO DO THIS WE USE ENCRYPTION (e.g. SSL/TLS) AND USER AUTHENTICATION AND AUTHORIZATION.

SCALABILITY: A DISTRIBUTED SYSTEM IS SCALABLE IF IT CAN MAINTAIN ADEQUATE PERFORMANCE AS THE NUMBER OF USERS OR RESOURCES INCREASES. THE SOLUTION IS TO DUPLICATE DATA AND SERVICES ACROSS MULTIPLE NODES.

CONCURRENCY: IN DISTRIBUTED SYSTEMS, MULTPLE USERS CAN ACCESS THE SAME RESOURCES SIMULTANEOUSLY. WE NEED TO MANAGE CONCURRENT READS AND WRITE VIA SYNCHRONIZATION (e.g. SEMAPHORES).

TRANSPARENCY. LOCAL AND REMOTE RESOURCES ARE ACCESSIBLE EQUALLY WITHOUT KNOWING THEIR LOCATION. THE SYSTEM MANAGES CONCURRENCY AND FAILURES. FURTHERMORE, USERS AND RESOURCES CAN MOVE WITHOUT AFFECTING THE SERVICE.

MODELLING DISTRIBUTED SYSTEMS

LESSON 2-3

BASED ON:

- **PROCESSES:** THEY ARE STATIC, WITH UNIQUE IDENTITIES AND A RANKING FUNCTION FOR INDEXING. THEY COMMUNICATE VIA UNIQUE MESSAGES, IDENTIFIED BY SEQUENCE NUMBERS OR LOCAL CLOCKS.
- **ALGORITHMS:** DISTRIBUTED COLLECTION OF AUTOMATA, ONE PER PROCESS, THAT REGULATE THE EXECUTION OF OPERATIONS.
- **SAFETY:** THE ALGORITHM SHOULD NOT DO ANYTHING WRONG.
- **LIVENESS:** ENSURE THAT EVENTUALLY SOMETHING GOOD HAPPENS

FAILURE MODELS

- **CRASH:** THE PROCESS STOPS AND DOES NOT RESUME.
- **OMISSION:** MESSAGES NOT SENT/RECEIVED.
- **CRASH-RECOVERY:** THE PROCESS CAN RECOVER AFTER A CRASH, BUT MAY LOSE ITS INTERNAL STATE.
- **EAVESDROPPING:** A PROCESS LEAKS INFORMATION OBTAINED IN AN ALGORITHM TO AN OUTSIDE ENTITY.
- **BYZANTINE:** ARBITRARY OR MALICIOUS BEHAVIOR.

TIMING ASSUMPTIONS

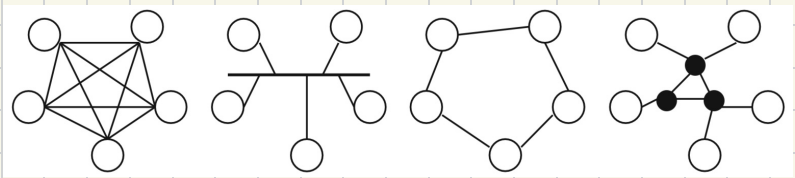
- **SYNCHRONOUS SYSTEM:** FEATURING LIMITS FOR OPERATION EXECUTION TIME, COMMUNICATION DELAY, AND PHYSICAL CLOCKS DRIFT.
- **ASYNCHRONOUS SYSTEM:** NO ASSUMPTIONS ABOUT PROCESSING OR COMMUNICATION TIMES. TIME IS BASED ON LOGICAL CLOCKS AND MESSAGES.
- **PARTIALLY SYNCHRONOUS:** SYNCHRONIZATION IS GUARANTEED ONLY AFTER AN UNKNOW TIME τ

ABSTRACTING COMMUNICATIONS

ABSTRACTIONS ALLOWS TO CAPTURE COMMON PROPERTIES TO MULTIPLE SYSTEMS, DISTINGUISH BETWEEN FUNDAMENTAL PROBLEMS AND SECONDARY DETAILS, AND AVOID REPEATEDLY SOLVING THE SAME PROBLEMS.

LINK ABSTRACTION

THE ABSTRACTION OF A LINK IS USED TO REPRESENT THE NETWORK COMPONENTS OF THE DISTRIBUTED SYSTEM. EVERY PAIR OF PROCESSES IS CONNECTED BY A BIDIRECTIONAL LINK.



- **FAIR-LOSS:** MESSAGES MAY BE LOST, BUT THEY MAY STILL ARRIVE.
- **STUBBORN:** THE SYSTEM GUARANTEES DELIVERY, BUT THERE MAY BE DUPLICATES.
- **PERFECT:** MESSAGES ALWAYS DELIVERED, ONLY ONCE AND IN ORDER.

MESSAGES ARE RETRANSMITTED TO COMPENSATE FOR LOSSES, AND CRYPTOGRAPICH PROTECTIONS ARE USED TO ENSURE INTEGRITY AND SECURITY.

TIME IN DISTRIBUTED SYSTEMS

LESSON 4

PROCESSES IN DS OPERATE ON DISTINCT NODES AND COMMUNICATE ONLY VIA MESSAGES. TIME IS AN ESSENTIAL QUANTITY FOR MANY APPLICATIONS AND ALGORITHMS. TIME SYNCHRONIZATION IS ESSENTIAL TO ORDER EVENTS AND ENSURE CORRECT FUNCTIONING.

EACH EVENT IS LABELED WITH A TIMESTAMP USING A LOCAL PHYSICAL CLOCK. IT'S SIMPLE TO SORT LOCAL EVENTS, BUT COMPLEX ACROSS DIFFERENT PROCESSES DUE TO NETWORK DELAYS AND LACK OF A GLOBAL CLOCK

HARDWARE CLOCKS (OSCILLATORS) GENERATE SOFTWARE CLOCKS.

CLOCK SYNCHRONIZATION

PARAMETERS AFFECTING THE ACCURACY:

- **SKEW.** THE DIFFERENCE IN TIME BETWEEN TWO LOCAL CLOCKS $|C_i(t) - C_j(t)|$.
- **DRIFT RATE:** GRADUAL DRIFT OF WATCHES CAUSED BY IMPERFECTIONS.

THERE ARE TWO TYPES.

- **EXTERNAL:** LOCAL CLOCKS ARE SYNCHRONIZED WITH AN AUTHORITATIVE SOURCE (e.g. UTC), WITH AN ERROR LIMIT D.
- **INTERNAL:** LOCAL CLOCKS ARE SYNCHRONIZED WITH EACH OTHER, WITH A MAXIMUM ERROR D.

INTERNALLY SYNCHRONIZED CLOCKS MAY NOT BE EXTERNALLY SYNCHRONIZED, WHILE EXT SYNC CLOCKS ARE AUTOMATICALLY INT SYNC (MAXIMUM 2D ERROR).

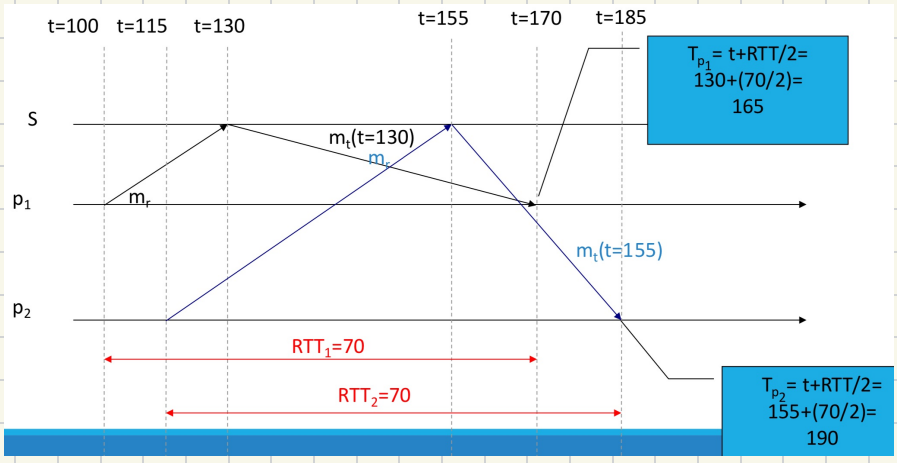
SYNCHRONIZATION ALGORITHMS

CHRISTIAN'S ALGORITHM: IS A EXT SYNC ALG. A PROCESS REQUESTS TIME FROM THE SERVER, CALCULATING AN ESTIMATED TIME AS

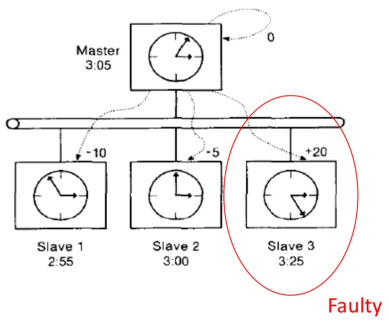
$$T = t + RTT/2$$

ACCURACY → $\pm (RTT/2 - t_{min})$

t_{min} IS THE MINIMUM TRASMISSION DELAY.



BARKELEY'S ALGORITHM: IS A INT SYNC ALG. THE MASTER PROCESS COLLECTS LOCAL CLOCKS, AVERAGES THE DIFFERENCES (Δi), AND SENDS CORRECTIONS TO THE PROCESSES.



Measuring the differences

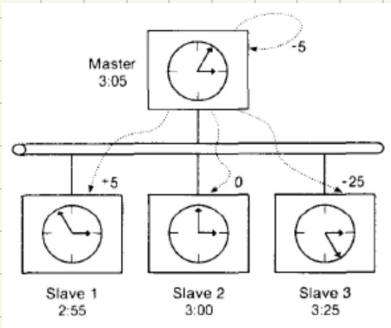
- $\Delta p_m = 3:05 - 3:05 = 0$
- $\Delta p_1 = 3:05 - 2:55 = -10$
- $\Delta p_2 = 3:05 - 3:00 = -5$
- $\Delta p_3 = 3:05 - 3:25 = 20$

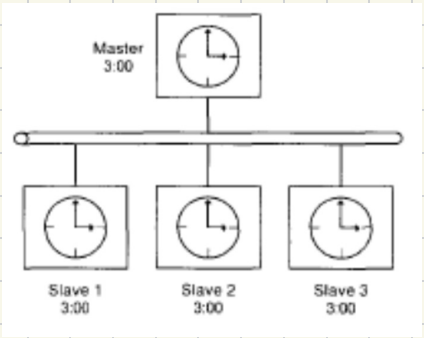
Computing the average

- Avg = (0 -10 -5) /3 = -5

Compute and send the Correction

- $Adj_m = Avg - \Delta p_m = -5 - 0 = -5$
- $Adj_1 = Avg - \Delta p_1 = -5 - (-10) = 5$
- $Adj_2 = Avg - \Delta p_2 = -5 - (-5) = 0$
- $Adj_3 = Avg - \Delta p_3 = -5 - 20 = -25$





NETWORK TIME PROTOCOL (NTP): SYNCHRONIZE CLIENTS WITH UTC VIA A SERVER HIERARCHY:

1. PRIMARY SERVER CONNECTED TO UTC SOURCES.
2. SECONDARY SERVER SYNCHRONIZED WITH PRIMARY ONES.

SYNC MODE:

- MULTICAST: LOW ACCURACY, USEFUL FOR FAST LANS.
- PROCEDURE CALL. HIGH ACCURACY, SIMILIAR TO CHRISTIAN'S ALG.
- SYMMETRICAL: TO SYNCHRONIZE SERVERS AT HIGHER LEVELS.

LOGICAL CLOCK

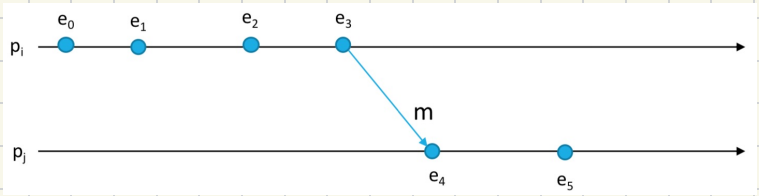
THE LOGICAL CLOCK IS A SW COUNTER THAT MONOTONICALLY INCREASES ITS VALUE. A LOGICAL CLOCK L_i CAN BE USED TO TIMESTAMP EVENTS.

HAPPENED-BEFORE RELATION

IN MANY APPLICATIONS IT DOESN'T MATTER WHEN AN EVENT HAPPENED, BUT IN WHAT ORDER AND THE HAPPENED-BEFORE RELATION CAPTURES THESE CASUAL DEPENDENCIES WITHOUT SYNCHRONIZATION OF PHYSICAL CLOCKS.

$e \rightarrow e'$: EVENT e OCCURS BEFORE e' IF:

- BOTH HAPPEN IN THE SAME PROCESS, AND e PRECEDES e'.
- e IS SENDING A MESSAGE, AND e' IS THE RECEPTION OF THE SAME.
- THE RELATIONSHIP IS TRANSITIVE.

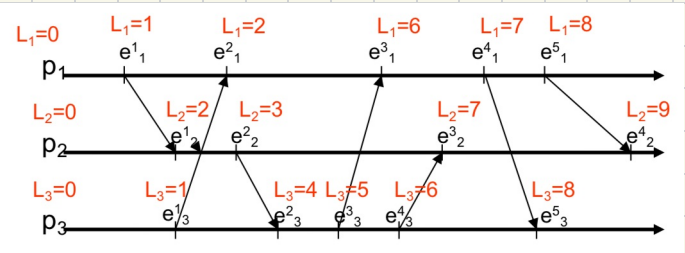


UNCORRELATED (CONCURRENT) EVENTS ARE REFERRED AS $e_i \| e_j$.

SCALAR LOGICAL CLOCK

EACH PROCESS p_i INITIALIZES ITS LOGICAL CLOCK $L_i = 0$ AND INCREASE L_i OF 1 WHEN IT GENERATES AN EVENT ($L_i = L_i + 1$). IN PARTICULAR, WHEN p_i SENDS A MESSAGE m, INCREASES L_i AND TIMESTAMP m WITH $t_s = L_i$. INSTEAD, IF p_i RECEIVES A MESSAGE THE LOGICAL CLOCK IS UPDATED $L_i = \max(t_s, L_i)$.

ENSURES $e \rightarrow e' \Rightarrow L(e) < L(e')$, BUT NOT THE OTHER WAY AROUND.



- $e^1_1 \rightarrow e^2_1$ and timestamps reflect this property
- $e^1_1 \| e^1_3$ and respective timestamps have the same value
- $e^1_2 \| e^1_3$ but respective timestamps have different values

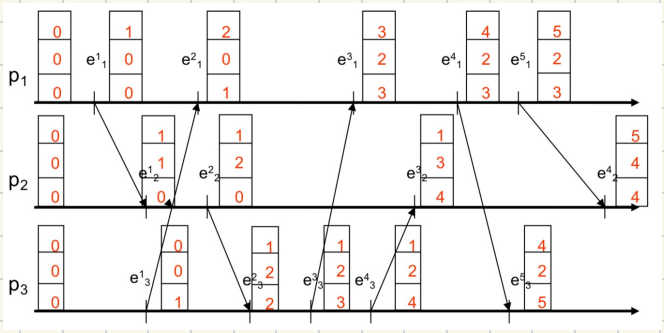
IT DOESN'T DISTINGUISH CONCURRENT FROM CORRELATED EVENTS.

SCALAR LOGICAL CLOCK IS SUITABLE FOR PROBLEMS SUCH AS DISTRIBUTED MUTUAL EXCLUSION (LAMPORT'S ALG).

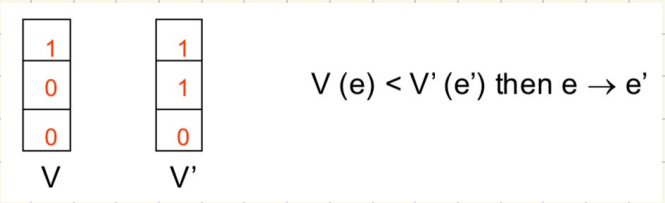
VECTOR CLOCK

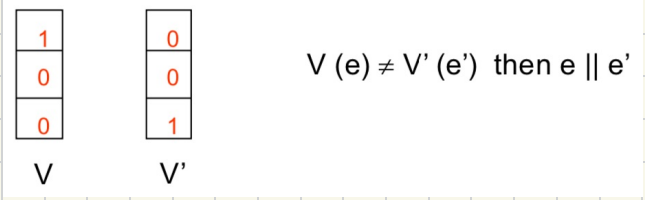
FOR A SET OF n PROCESSES, THE VECTOR CLOCK IS COMPOSED BY AN ARRAY OF n INTEGERS. SO EACH PROCESS p_i MAINTAIN A VECTOR CLOCK V_i AND TIMESTAMPS EVENTS BY MEAN OF ITS VECTOR CLOCK.

WHEN A TIMESTAMPED MESSAGE IS RECEIVED: $V_i[j] = \max(V_i[j], V[j])\ \forall j$.



IT DISTINGUISH IF $e \rightarrow e'$ OR $e \| e'$





VECTOR CLOCK IS USEFUL FOR ENSURING CASUAL ORDER.

DISTRIBUTED MUTUAL EXCLUSION

IT'S NECESSARY TO GUARANTEE THAT, IN A DISTRIBUTED SYSTEM, AT MOST ONE PROCESS ACCESSES A SHARED RESOURCE (CRITICAL SECTION, CS) AT A TIME.

IT IS BASED ON:

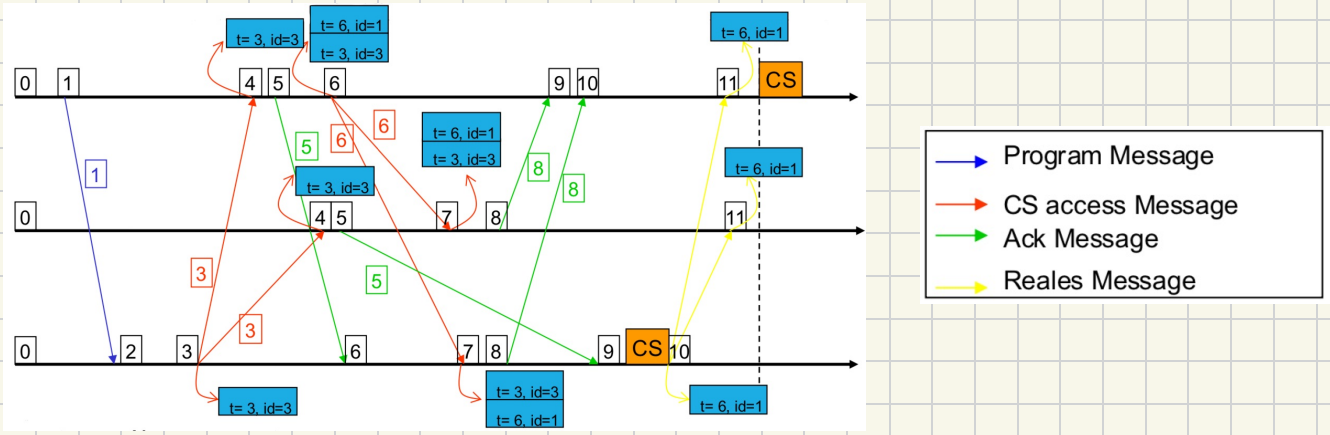
- MUTUAL EXCLUSION: AT MOST ONE PROCESS IN CS.
- NO-DEADLOCK: AT LEAST ONE PROCESS CAN ENTER CS.
- NO-STARVASION: ALL REQUESTS ARE EVENTUALLY SATISFIED.

LAMPORT'S ALGORITHM

EACH PROCESS SENDS A CS ACCESS REQUEST WITH ITS OWN TIMESTAMP TO THE OTHER PROCESSES, AND MAINTAINS AN ORDERED QUEUE FOR REQUESTS.

A PROCESS ENTERS CS IF ITS REQUEST IS THE OLDEST IN THE QUEUE AND IT HAS RECEIVED ACKNOWLEDGMENTS FROM ALL OTHER PROCESSES.

AFTER FINISHING, THE PROCESS SENDS A RELEASE MESSAGE TO OTHERS, WHO REMOVE THE REQUEST FROM THE QUEUE.



LAMPORT'S ALG NEEDS 3(N-1) MESSAGES FOR THE CS EXECUTION:

N-1 REQUEST, N-1 ACKS, N-1 RELEASES

RICART-AGRAWALA'S ALGORITHM

REDUCES THE NUMBER OF MESSAGES NEEDED TO 2(N-1):

N-1 REQUEST, N-1 ACKS

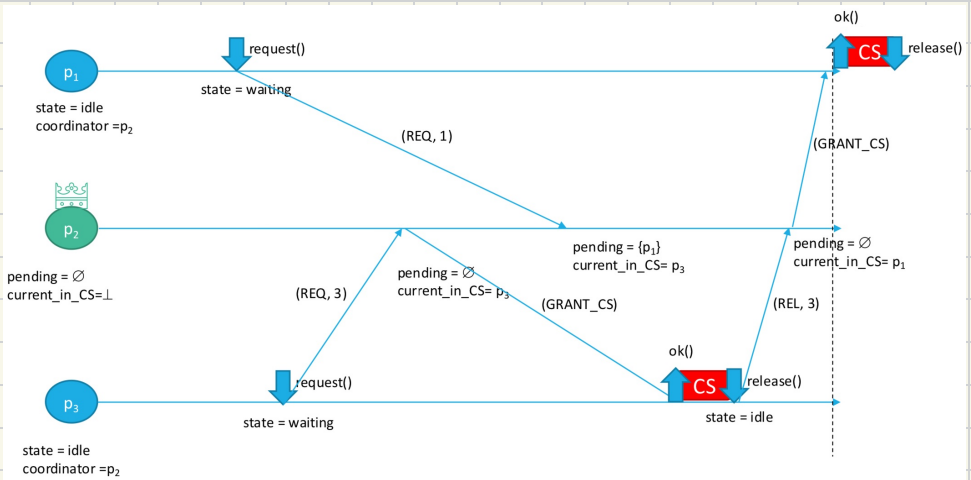
EACH PROCESS MAINTAINS A STATE (REQUEST, CS, NOT-CS) AND A QUEUE OF PENDING REQUESTS.

A PROCESS RESPONDS TO A REQUEST IF:

- IT'S NOT IN CS.
- DOESN'T ALREADY HAVE A REQUEST WITH AN OLDER TIMESTAMP.

COORDINATOR BASED

THERE IS A SPECIAL PROCESS (COORDINATOR) THAT RECEIVES REQUESTS FROM PROCESSES, ENSURES ACCESS TO THE CS BY SENDING A CONFIRMATION MESSAGE, AND RELEASE THE CS FOR OTHER PROCESSES AFTER RELEASE. THE COORDINATOR ALWAYS MAINTAIN A QUEUE OF PENDING REQUESTS.

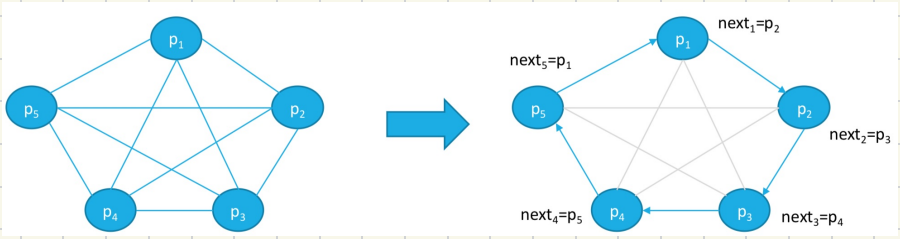


REQUIRES 2 MESSAGES TO ACCESS THE CS, AND 1 TO RELEASE IT.

TOKEN BASED

A UNIQUE TOKEN IS CREATED AND CIRCULATES IN A LOGICAL RING OF PROCESSES. ONLY THE PROCESS THAT OWNS THE TOKEN CAN ACCESS THE CS.

A LOGICAL RING IS CONSTRUCTED BY EXPLOITING POINT-TO-POINT COMMUNICATION CHANNELS. A TOKEN IS GENERATED AND INITIALLY ASSIGNED TO A PROCESS. WHEN A PROCESS REQUIRES CS, IT WAITS TO RECEIVE THE TOKEN AND ACCESSES THE CS, FINALLY SENDING THE TOKEN TO THE NEXT ONE IN THE RING. IF A PROCESS RECEIVES THE TOKEN BUT DOESN'T NEED THE CS, IT FORWARDS IT.



IT CONSUMES ONLY N MESSAGES PER ROUND BUT THE TOKEN CIRCULATES EVEN WHEN NO PROCESS REQUIRES THE CS.

QUORUM BASED

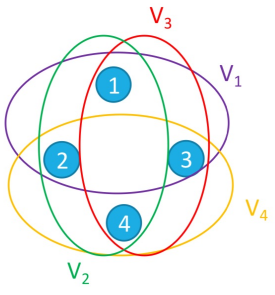
EACH PROCESS OBTAINS PERMISSION FROM A SUBSET OF PROCESSES (VOTING SET) TO ACCESS THE CS. VOTING SETS OVERLAP TO ENSURE CONFLICT AND MUTUAL EXCLUSION.

EACH PROCESS BELONGS TO ITS OWN VOTING SET AND TO M VOTING SETS. EACH PAIR OF VOTING SETS HAS AT LEAST ONE MEMBER IN COMMON. ALL VOTING SETS HAVE THE SAME $|V_i| = K$ DIMENSION.

A PROCESS REQUESTS PERMISSION TO ITS VOTING SET, RECEIVES ACKS FROM MEMBERS OF THE VOTING SET AND ENTERS THE CS. AFTER RELEASE, SENDS REL MESSAGES TO NOTIFY THE END OF ACCESS.

p1	p2
p3	p4

Process id	V
1	p_1, p_2, p_3
2	p_1, p_2, p_4
3	p_1, p_3, p_4
4	p_2, p_3, p_4



THIS ALGORITHM IS NOT DEADLOCK-FREE

FAILURE DETECTION

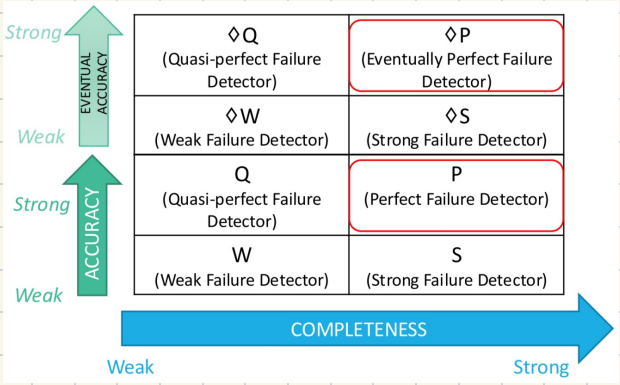
LESSON 7

A FAILURE DETECTOR (FD) IS A SW MODULE THAT PROVIDES INFORMATION ON THE FAILURE STATUS OF PROCESSES. A FD ENCAPSULATES THE TEMPORAL ASSUMPTIONS OF SYNCHRONOUS OR PARTIALLY SYNCHRONOUS SYSTEMS, AND ITS PRECISION DEPENDS ON THE STRENGTH OF THE TEMPORAL ASSUMPTIONS.

CLASSIFICATION

A FD IS DESCRIBED BY:

- **ACCURACY:** THE ABILITY TO AVOID MISTAKES IN THE DETECTION.
- **COMPLETENESS:** THE ABILITY TO DETECT ALL FAILURES.



- **W COMPL:** EVERY PROCESS THAT CRASHES IS PERMANENTLY SUSPECTED BY SOME CORRECT PROCESS.
- **S COMPL:** EVERY PROCESS THAT CRASHES IS PERMANENTLY SUSPECTED BY EVERY CORRECT PROCESS.
- **W ACC:** SOME CORRECT PROCESS IS NEVER SUSPECTED.
- **S ACC:** CORRECT PROCESSES ARE NEVER SUSPECTED.
- **EV W ACC:** THERE IS A TIME AFTER WHICH SOME CORRECT PROCESS IS NOT SUSPECTED.
- **EV S ACC:** THERE IS A TIME AFTER WHICH CORRECT PROCESSES ARE NOT SUSPECTED.

Module 2.6: Interface and properties of the perfect failure detector

Module:

Name: PerfectFailureDetector, **instance** $\mathcal{P}$.

Events:

Indication: $\langle\ \mathcal{P}, Crash \mid p\ \rangle$: Detects that process p has crashed.

Properties:

PFD1: *Strong completeness:* Eventually, every process that crashes is permanently detected by every correct process.

PFD2: *Strong accuracy:* If a process p is detected by any process, then p has crashed.

PERFECT FAILURE DETECTOR IMPLEMENTATION

Algorithm 2.5: Exclude on Timeout

Implements:
PerfectFailureDetector, **instance** $\mathcal{P}$.

Uses:
PerfectPointToPointLinks, **instance** pl.

upon event $\langle\ \mathcal{P}, Init\ \rangle$ **do**
  $alive := \Pi$;
  $detected := \emptyset$;
  $starttimer(\Delta)$;

upon event $\langle\ Timeout\ \rangle$ **do**
  forall $p \in \Pi$ **do**
    if $(p \notin alive) \land (p \notin detected)$ **then**
      $detected := detected \cup \{p\}$;
      trigger $\langle\ \mathcal{P}, Crash \mid p\ \rangle$;
    trigger $\langle\ pl, Send \mid p, [\text{HEARTBEATREQUEST}]\ \rangle$;
  $alive := \emptyset$;
  $starttimer(\Delta)$;

upon event $\langle\ pl, Deliver \mid q, [\text{HEARTBEATREQUEST}]\ \rangle$ **do**
  trigger $\langle\ pl, Send \mid q, [\text{HEARTBEATREPLY}]\ \rangle$;

upon event $\langle\ pl, Deliver \mid p, [\text{HEARTBEATREPLY}]\ \rangle$ **do**
  $alive := alive \cup \{p\}$;

LEADER ELECTION

ELECTING A LEADER CONSISTS OF IDENTIFYING A PROCESS THAT ACTS AS A COORDINATOR BETWEEN DISTRIBUTED PROCESSES. A LEADER CAN MANAGE EXCLUSIVE ACCESS TO A SHARED RESOURCE.

IMPLEMENTATION WITH P

THE GOAL IS TO IDENTIFY A LEADER AMONG THE CORRECT PROCESSES WITH THE HIGHEST IDENTIFIER IN A DISTRIBUTED SYSTEM, EXPLOITING A PERFECT FAILURE DETECTOR (P) THAT GUARANTEES THE DETECTION OF ALL FAILED PROCESSES

Module 2.7: Interface and properties of leader election

Module:

Name: LeaderElection, **instance** *le*.

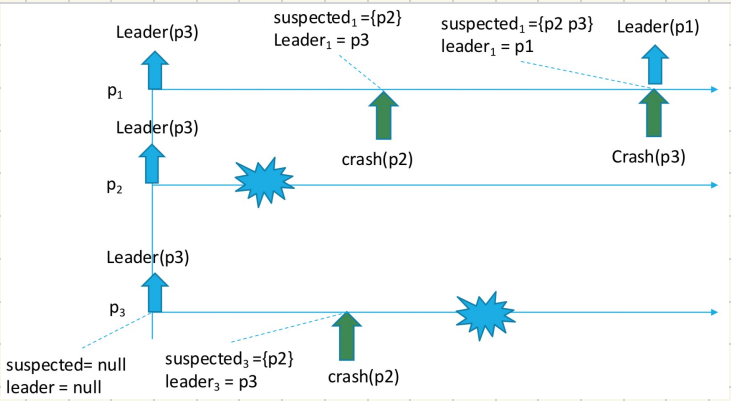
Events:

Indication: ⟨ *le*, *Leader* | *p* ⟩: Indicates that process *p* is elected as leader.

Properties:

LE1: *Eventual detection:* Either there is no correct process, or some correct process is eventually elected as the leader.

LE2: *Accuracy:* If a process is leader, then all previously elected leaders have crashed.



Algorithm 2.6: Monarchical Leader Election

Implements:
LeaderElection, **instance** *le*.

Uses:
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *le*, *Init* ⟩ **do**
suspected := ∅;
leader := ⊥;

upon event ⟨ $\mathcal{P}$, *Crash* | *p* ⟩ **do**
suspected := *suspected* ∪ {*p*};

upon *leader* ≠ maxrank(Π \ *suspected*) **do**
leader := maxrank(Π \ *suspected*);
trigger ⟨ *le*, *Leader* | *leader* ⟩;

EVENTUAL LEADER ELECTION

IT'S AN ABSTRACTION THAT ENSURE THAT, EVENTUALLY, ALL CORRECT PROCESSES ELECT THE SAME CORRECT LEADER.
DURING THE TRANSITIONAL PERIOD, MORE LEADERS MAY BE ELECTED. ONCE STABILIZED, THE LEADER NO LONGER CHANGES.

Module 2.9: Interface and properties of the eventual leader detector

Module:

Name: EventualLeaderDetector, **instance** Ω.

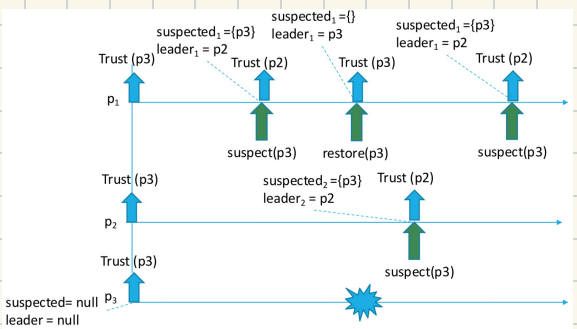
Events:

Indication: ⟨ Ω, *Trust* | *p* ⟩: Indicates that process *p* is trusted to be leader.

Properties:

ELD1: *Eventual accuracy:* There is a time after which every correct process trusts some correct process.

ELD2: *Eventual agreement:* There is a time after which no two correct processes trust different correct processes.



Algorithm 2.8: Monarchical Eventual Leader Detection

Implements:
EventualLeaderDetector, **instance** Ω.

Uses:
EventuallyPerfectFailureDetector, **instance** $\Diamond\mathcal{P}$.

upon event ⟨ Ω, *Init* ⟩ **do**
suspected := ∅;
leader := ⊥;

upon event ⟨ $\Diamond\mathcal{P}$, *Suspect* | *p* ⟩ **do**
suspected := *suspected* ∪ {*p*};

upon event ⟨ $\Diamond\mathcal{P}$, *Restore* | *p* ⟩ **do**
suspected := *suspected* \ {*p*};

upon *leader* ≠ maxrank(Π \ *suspected*) **do**
leader := maxrank(Π \ *suspected*);
trigger ⟨ Ω, *Trust* | *leader* ⟩;

CRASH RECOVERY

THE NUMBER OF CRASHES AND RECOVERIES FOR EACH PROCESS IS TRACKED. THE LEADER IS CHOSEN BASED ON THE NUMBER OF TIMES A PROCESS HAS BEEN RECOVERED (EPOCHS), AND THE LOWEST IDENTIFIER AMONG PROCESS WITH THE SAME NUMBER OF EPOCHS.

Algorithm 2.9: Elect Lower Epoch

Implements:
EventualLeaderDetector, **instance** Ω.

Uses:
FairLossPointToPointLinks, **instance** *fll*.

keeps track of how many times the process crashed and recovered

upon event $\langle\ \Omega, Init\ \rangle$ **do**
epoch := 0;
store(*epoch*);
candidates := $\emptyset$;
trigger $\langle\ \Omega, Recovery\ \rangle$;

upon event $\langle\ \Omega, Recovery\ \rangle$ **do**
leader := maxrank(Π);
trigger $\langle\ \Omega, Trust \mid leader\ \rangle$;
delay := Δ;
retrieve(*epoch*);
epoch := *epoch* + 1;
store(*epoch*);
forall $p \in \Pi$ **do**
trigger $\langle\ fll, Send \mid p, [\text{HEARTBEAT}, epoch]\ \rangle$;
candidates := $\emptyset$;
starttimer(*delay*);

Algorithm 2.9: Elect Lower Epoch

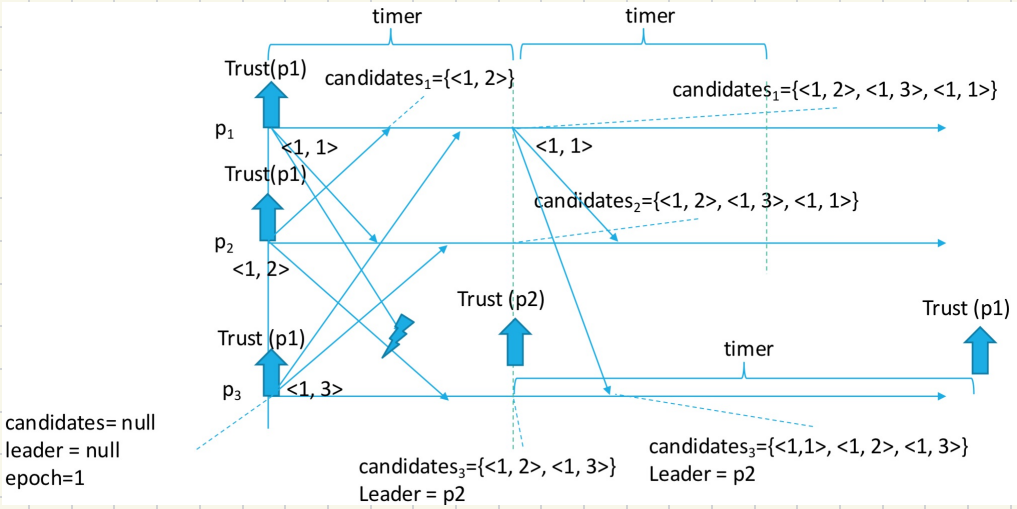
Implements:
EventualLeaderDetector, **instance** Ω.

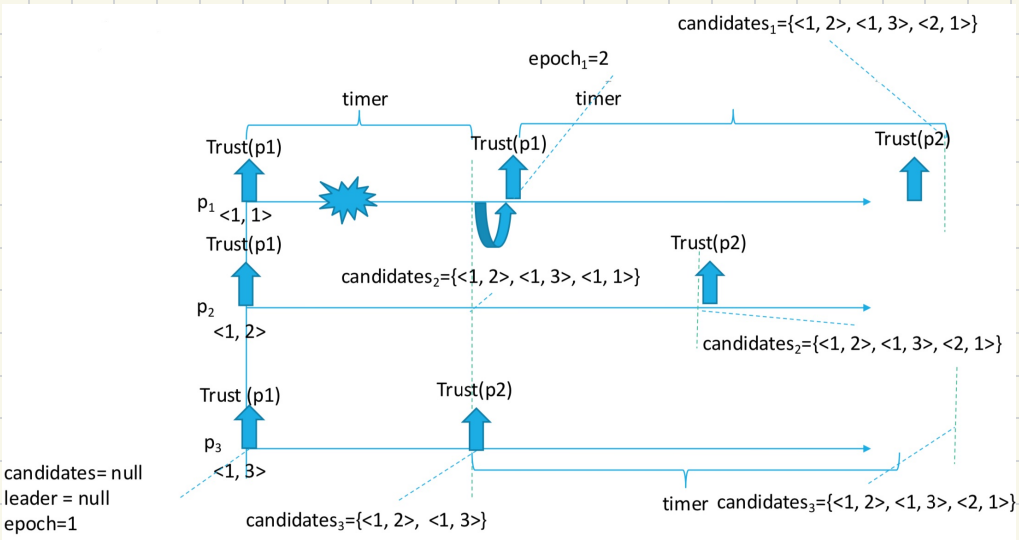
Uses:
FairLossPointToPointLinks, **instance** *fll*.

deterministic function returning one process among all candidates (i.e., process with the lowest epoch number and among the ones with the same epoch number the one with the lowest identifier)

upon event $\langle\ Timeout\ \rangle$ **do**
newleader := select(*candidates*);
if *newleader* $\neq$ *leader* **then**
delay := *delay* + Δ;
leader := *newleader*;
trigger $\langle\ \Omega, Trust \mid leader\ \rangle$;
forall $p \in \Pi$ **do**
trigger $\langle\ fll, Send \mid p, [\text{HEARTBEAT}, epoch]\ \rangle$;
candidates := $\emptyset$;
starttimer(*delay*);

upon event $\langle\ fll, Deliver \mid q, [\text{HEARTBEAT}, ep]\ \rangle$ **do**
if exists $(s, e) \in candidates$ such that $s = q \wedge e < ep$ **then**
candidates := *candidates* $\setminus \{(q, e)\}$;
candidates := *candidates* $\cup\ (q, ep)$;





BROADCAST COMMUNICATIONS

LESSON 9, 10

SO FAR WE HAVE FOCUSED ON THE INTERACTION BETWEEN TWO PROCESSES (CLIENT/SERVER). NOW LET'S BROADEN THE DISCUSSION TO BROADCAST, i.e. COMMUNICATION IN A GROUP OF PROCESSES.

BEST EFFORT BROADCAST (BEB)

Module 3.1: Interface and properties of best-effort broadcast

Module:

Name: BestEffortBroadcast, **instance** *beb*.

Events:

Request: ⟨ *beb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *beb*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p.

Properties:

BEB1: *Validity:* If a correct process broadcasts a message m, then every correct process eventually delivers m.

BEB2: *No duplication:* No message is delivered more than once.

BEB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

Algorithm 3.1: Basic Broadcast

Implements:
BestEffortBroadcast, **instance** *beb*.

Uses:
PerfectPointToPointLinks, **instance** *pl*.

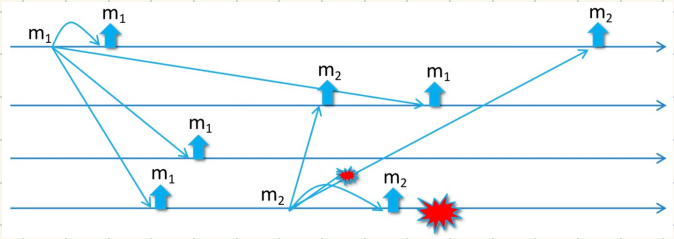
upon event ⟨ *beb*, *Broadcast* | m ⟩ **do**
forall $q \in \Pi$ **do**
trigger ⟨ *pl*, *Send* | q, m ⟩;

upon event ⟨ *pl*, *Deliver* | p, m ⟩ **do**
trigger ⟨ *beb*, *Deliver* | p, m ⟩;

VALIDITY: IF ONE CORRECT PROCESS TRANSMITS A MESSAGE, ALL CORRECT PROCESSES RECEIVE IT

NO DUPLICATION: EACH MESSAGE IS DELIVERED ONLY ONCE.

NO CREATION: ABSENCE OF CREATION OF NON-EXISTENT MESSAGES.

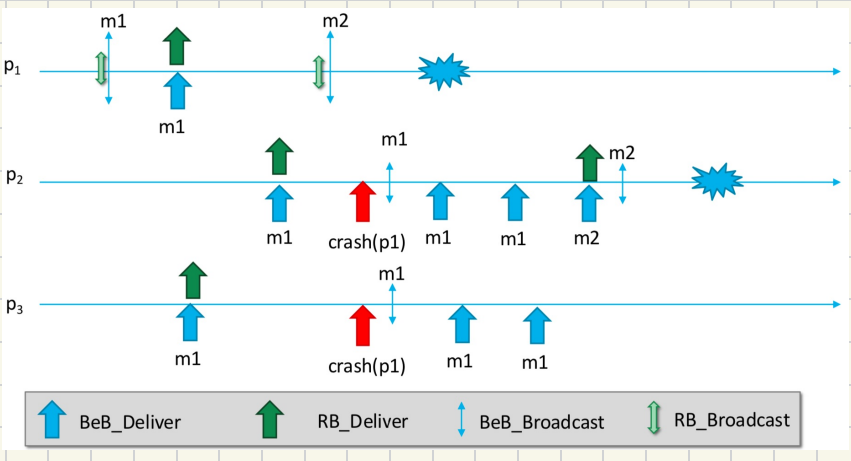


BEB GUARANTEES MESSAGE DELIVERY UNTIL THE SENDER FAILS. IF THE SENDER CRASHES DURING TRANSMISSION, SOME PROCESSES MAY NOT RECEIVE THE MESSAGE, CAUSING DISAGREEMENTS.

RELIABLE BROADCAST (RB)

RB IS BASED ON BEB, USES A PFD AND PROVIDES AN ADDITIONAL PROPERTY:

AGREEMENT: IF ONE CORRECT PROCESS DELIVERS A MESSAGE, ALL CORRECT PROCESSES DELIVER IT.



Module 3.2: Interface and properties of (regular) reliable broadcast

Module:

Name: ReliableBroadcast, **instance** *rb*.

Events:

Request: ⟨ *rb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *rb*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p.

Properties:

RB1: *Validity:* If a correct process p broadcasts a message m, then p eventually delivers m.

RB2: *No duplication:* No message is delivered more than once.

RB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

RB4: *Agreement:* If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

Algorithm 3.2: Lazy Reliable Broadcast

Implements:
ReliableBroadcast, **instance** *rb*.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *rb*, *Init* ⟩ **do**
correct := Π;
from[p] := $[\emptyset]^N$;

upon event ⟨ *rb*, *Broadcast* | m ⟩ **do**
trigger ⟨ *beb*, *Broadcast* | [DATA, *self*, m] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [DATA, s, m] ⟩ **do**
if $m \notin$ *from*[s] **then**
trigger ⟨ *rb*, *Deliver* | s, m ⟩;
from[s] := *from*[s] $\cup \{m\}$;
if $s \notin$ *correct* **then**
trigger ⟨ *beb*, *Broadcast* | [DATA, s, m] ⟩;

upon event ⟨ $\mathcal{P}$, *Crash* | p ⟩ **do**
correct := *correct* $\setminus \{p\}$;
forall $m \in$ *from*[p] **do**
trigger ⟨ *beb*, *Broadcast* | [DATA, p, m] ⟩;

THE ALGORITHM IS "LAZY" BECAUSE MESSAGES ARE RETRASMITTED ONLY WHEN NECESSARY.

THE EAGER ALGORITHM RETRANSMITS EVERY MESSAGE.

Algorithm 3.3: Eager Reliable Broadcast

Implements:
ReliableBroadcast, **instance** *rb*.

Uses:
BestEffortBroadcast, **instance** *beb*.

upon event ⟨ *rb*, *Init* ⟩ **do**
delivered := $\emptyset$;

upon event ⟨ *rb*, *Broadcast* | m ⟩ **do**
trigger ⟨ *beb*, *Broadcast* | [DATA, *self*, m] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [DATA, s, m] ⟩ **do**
if $m \notin$ *delivered* **then**
delivered := *delivered* $\cup \{m\}$;
trigger ⟨ *rb*, *Deliver* | s, m ⟩;
trigger ⟨ *beb*, *Broadcast* | [DATA, s, m] ⟩;

UNIFORM RELIABLE BROADCAST (URB)

URB ENSURES THAT GOOD AND BAD PROCESSES AGREE ON DELIVERED MESSAGES:

UNIFORM AGREEMENT: IF A MESSAGE IS DELIVERED BY ONE PROCESS (CORRECT OR FAILED), IT MUST BE DELIVERED BY ALL CORRECT PROCESSES.

Module 3.3: Interface and properties of uniform reliable broadcast

Module:

Name: UniformReliableBroadcast, **instance** *urb*.

Events:

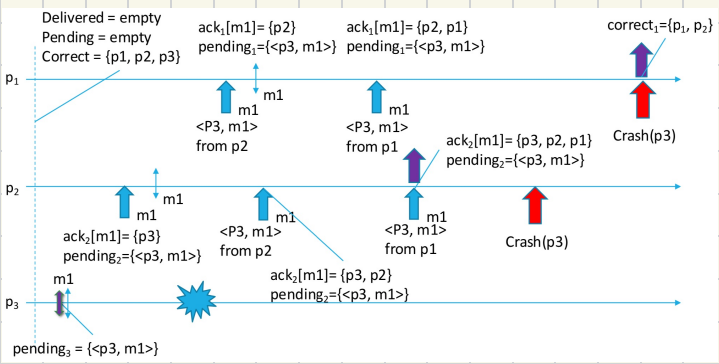
Request: ⟨ *urb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *urb*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p.

Properties:

URB1–URB3: Same as properties RB1–RB3 in (regular) reliable broadcast (Module 3.2).

URB4: *Uniform agreement:* If a message m is delivered by some process (whether correct or faulty), then m is eventually delivered by every correct process.



Algorithm 3.4: All-Ack Uniform Reliable Broadcast

Implements:
UniformReliableBroadcast, **instance** *urb*.

Uses:
BestEffortBroadcast, **instance** *beb*.
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *urb*, *Init* ⟩ **do**
delivered := $\emptyset$;
pending := $\emptyset$;
correct := Π;
forall m **do** *ack*[m] := $\emptyset$;

upon event ⟨ *urb*, *Broadcast* | m ⟩ **do**
pending := *pending* $\cup \{(self, m)\}$;
trigger ⟨ *beb*, *Broadcast* | [DATA, *self*, m] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [DATA, s, m] ⟩ **do**
ack[m] := *ack*[m] $\cup \{p\}$;
if $(s, m) \notin$ *pending* **then**
pending := *pending* $\cup \{(s, m)\}$;
trigger ⟨ *beb*, *Broadcast* | [DATA, s, m] ⟩;

upon event ⟨ $\mathcal{P}$, *Crash* | p ⟩ **do**
correct := *correct* $\setminus \{p\}$;

function *candeliver*(m) **returns** Boolean **is**
return (*correct* $\subseteq$ *ack*[m]);

upon exists $(s, m) \in$ *pending* such that *candeliver*(m) $\wedge$ $m \notin$ *delivered* **do**
delivered := *delivered* $\cup \{m\}$;
trigger ⟨ *urb*, *Deliver* | s, m ⟩;

PROBABILISTIC BROADCAST

IT ENSURES THAT MESSAGES ARE DELIVERED WITH A HIGH PROBABILITY (NOT FULLY RELIABLE). IT'S USEFUL FOR LARGE AND DYNAMIC GROUPS.

IN SYSTEM WITH MANY PROCESSES EACH PROCESS SENDS AND RECEIVES TOO MANY ACK MESSAGES, OVERLOADING THE SYSTEM. A POSSIBLE SOLUTION IS TO USE A HIERARCHICAL STRUCTURE TO COLLECT ACKS (e.g. BINARY TREE).

Module 3.7: Interface and properties of probabilistic broadcast

Module:

Name: ProbabilisticBroadcast, **instance** *pb*.

Events:

Request: ⟨ *pb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *pb*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p.

Properties:

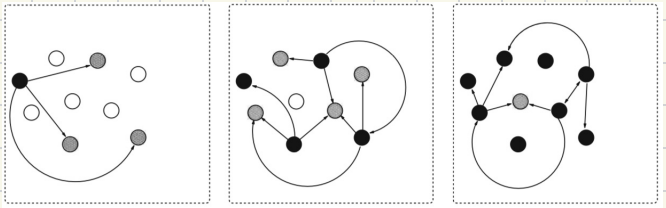
PB1: *Probabilistic validity:* There is a positive value ε such that when a correct process broadcasts a message m, the probability that every correct process eventually delivers m is at least $1 - \varepsilon$.

PB2: *No duplication:* No message is delivered more than once.

PB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

GOSSIP DISSEMINATION

INSPIRED BY THE SPREAD OF EPIDEMICS, EACH PROCESS SENDS A MESSAGE TO K RANDOMLY CHOSEN PROCESSES, WHICH FORWARD THE MESSAGE IN TURN.



(a) round 1 (b) round 2 (c) round 3

Algorithm 3.9: Eager Probabilistic Broadcast

Implements:
ProbabilisticBroadcast, **instance** *pb*.

Uses:
FairLossPointToPointLinks, **instance** *fll*.

upon event ⟨ *pb*, *Init* ⟩ **do**
delivered := ∅;

procedure *gossip*(*msg*) **is**
forall $t \in$ *picktargets*(k) **do trigger** ⟨ *fll*, *Send* | t, *msg* ⟩;

upon event ⟨ *pb*, *Broadcast* | m ⟩ **do**
delivered := *delivered* ∪ {m};
trigger ⟨ *pb*, *Deliver* | *self*, m ⟩;
gossip([GOSSIP, *self*, m, R]);

upon event ⟨ *fll*, *Deliver* | p, [GOSSIP, s, m, r] ⟩ **do**
if $m \notin$ *delivered* **then**
delivered := *delivered* ∪ {m};
trigger ⟨ *pb*, *Deliver* | s, m ⟩;
if $r > 1$ **then** *gossip*([GOSSIP, s, m, $r - 1$]);

function *picktargets*(k) **returns** set of processes **is**
targets := ∅;
while #(*targets*) < k **do**
candidate := random($\Pi \setminus$ {*self*});
if *candidate* $\notin$ *targets* **then**
targets := *targets* ∪ {*candidate*};
return *targets*;

IT DOESN'T GUARANTEE THAT ALL PROCESSES RECEIVE THE MESSAGE.

CONSENSUS

A GROUP OF PROCESSES MUST AGREE ON A SINGLE VALUE PROPOSED BY ONE OR MORE PROCESSES. IT IS BASED ON:

- **VALIDITY:** IF ALL PROCESSES PROPOSE THE SAME VALUE, THE VALUE DECIDED MUST BE THE ONE PROPOSED.
- **INTEGRITY:** EACH PROCESS DECIDES AT MOST ONCE, AND THE DECIDED VALUE MUST BE ONE OF THE PROPOSED VALUES.
- **AGREEMENT:** NO CORRECT PROCESS DECIDES A DIFFERENT VALUE THAN ANOTHER CORRECT PROCESS.
- **TERMINATION:** EVERY CORRECT PROCESS EVENTUALLY DECIDES SOME VALUE.

Module 5.1: Interface and properties of (regular) consensus

Module:

Name: Consensus, **instance** c.

Events:

Request: $\langle\ c, Propose \mid v\ \rangle$: Proposes value v for consensus.

Indication: $\langle\ c, Decide \mid v\ \rangle$: Outputs a decided value v of consensus.

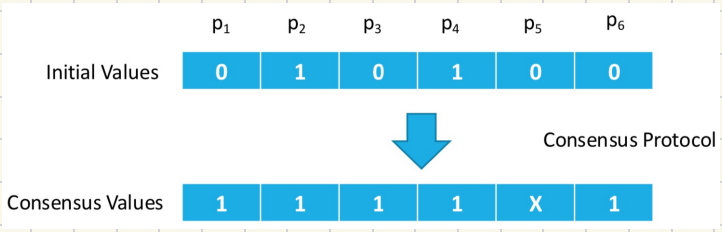
Properties:

C1: *Termination:* Every correct process eventually decides some value.

C2: *Validity:* If a process decides v, then v was proposed by some process.

C3: *Integrity:* No process decides twice.

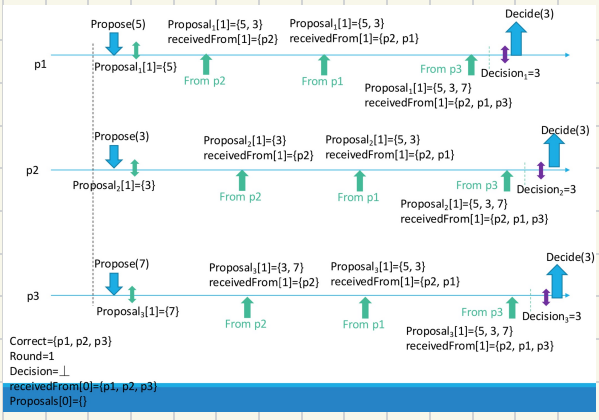
C4: *Agreement:* No two correct processes decide differently.



IT'S IMPOSSIBLE TO GUARANTEE CONSENSUS IN AN ASYNCHRONOUS SYSTEM, EVEN WITH A SINGLE CRASH FAILURE.

FLOODING CONSENSUS

PROCESSES EXCHANGE THEIR INITIAL PROPOSALS. ONCE ALL PROPOSALS HAVE BEEN RECEIVED FROM THE CORRECT PROCESSES, A SINGLE VALUE IS SELECTED. IF A PROCESS CRASHES DURING COMMUNICATION, SOME PROPOSALS MAY BE LOST, SO A VALUE IS DECIDED ONLY WHEN ALL PROPOSALS FROM THE CORRECT PROCESSES ARE AVAILABLE.



Algorithm 5.1: Flooding Consensus

Implements:
Consensus, **instance** c.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event $\langle\ c, Init\ \rangle$ **do**
$correct := \Pi$;
$round := 1$;
$decision := \bot$;
$receivedfrom := [\emptyset]^N$;
$proposals := [\emptyset]^N$;
$receivedfrom[0] := \Pi$;

upon event $\langle\ \mathcal{P}, Crash \mid p\ \rangle$ **do**
$correct := correct \setminus \{p\}$;

upon event $\langle\ c, Propose \mid v\ \rangle$ **do**
$proposals[1] := proposals[1] \cup \{v\}$;
trigger $\langle\ beb, Broadcast \mid [\text{PROPOSAL}, 1, proposals[1]]\ \rangle$;

upon event $\langle\ beb, Deliver \mid p, [\text{PROPOSAL}, r, ps]\ \rangle$ **do**
$receivedfrom[r] := receivedfrom[r] \cup \{p\}$;
$proposals[r] := proposals[r] \cup ps$;

upon $correct \subseteq receivedfrom[round] \wedge decision = \bot$ **do**
if $receivedfrom[round] = receivedfrom[round-1]$ **then**
$decision := \min(proposals[round])$;
trigger $\langle\ beb, Broadcast \mid [\text{DECIDED}, decision]\ \rangle$;
trigger $\langle\ c, Decide \mid decision\ \rangle$;
else
$round := round + 1$;
trigger $\langle\ beb, Broadcast \mid [\text{PROPOSAL}, round, proposals[round-1]]\ \rangle$;

upon event $\langle\ beb, Deliver \mid p, [\text{DECIDED}, v]\ \rangle$ **such that** $p \in correct \wedge decision = \bot$ **do**
$decision := v$;
trigger $\langle\ beb, Broadcast \mid [\text{DECIDED}, decision]\ \rangle$;
trigger $\langle\ c, Decide \mid decision\ \rangle$;

UNIFORM CONSENSUS

IT ADDS A STRONGER AGREEMENT PROPERTY, THAT EVEN IF ONE PROCESS CRASHES, THE CORRECT PROCESSES MUST STILL AGREE ON THE MESSAGE DELIVERED.

Module 5.2: Interface and properties of uniform consensus

Module:

Name: UniformConsensus, **instance** *uc*.

Events:

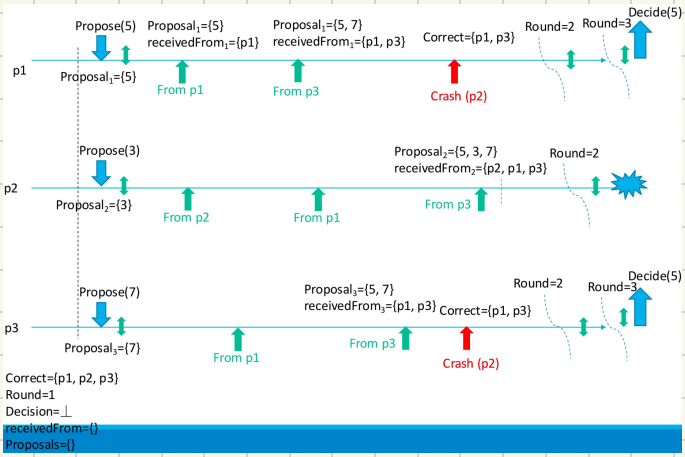
Request: ⟨ *uc*, *Propose* | *v* ⟩: Proposes value *v* for consensus.

Indication: ⟨ *uc*, *Decide* | *v* ⟩: Outputs a decided value *v* of consensus.

Properties:

UC1–UC3: Same as properties C1–C3 in (regular) consensus (Module 5.1).

UC4: *Uniform agreement:* No two processes decide differently.



Algorithm 5.3: Flooding Uniform Consensus

Implements:
UniformConsensus, **instance** *uc*.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *uc*, *Init* ⟩ **do**
correct := Π;
round := 1;
decision := ⊥;
proposalset := ∅;
receivedfrom := ∅;

upon event ⟨ $\mathcal{P}$, *Crash* | *p* ⟩ **do**
correct := *correct* \ {*p*};

upon event ⟨ *uc*, *Propose* | *v* ⟩ **do**
proposalset := *proposalset* ∪ {*v*};
trigger ⟨ *beb*, *Broadcast* | [PROPOSAL, 1, *proposalset*] ⟩;

No more related to the round

upon event ⟨ *beb*, *Deliver* | *p*, [PROPOSAL, *r*, *ps*] ⟩ **such that** *r* = *round* **do**
receivedfrom := *receivedfrom* ∪ {*p*};
proposalset := *proposalset* ∪ *ps*;

upon *correct* ⊆ *receivedfrom* ∧ *decision* = ⊥ **do**
if *round* = *N* **then**
decision := min(*proposalset*);
trigger ⟨ *uc*, *Decide* | *decision* ⟩;
else
round := *round* + 1;
receivedfrom := ∅;
trigger ⟨ *beb*, *Broadcast* | [PROPOSAL, *round*, *proposalset*] ⟩;

Decision only at the end

Cleaned at the beginning of each round

CONSENSUS PAXOS

LESSON 12

IT'S IMPOSSIBLE TO GUARANTEE CONSENSUS IN AN ASYNCHRONOUS SYSTEM, EVEN WITH A SINGLE CRASH FAILURE. A POSSIBLE SOLUTION IS THE PAXOS FAMILY OF ALGORITHM WHICH ALWAYS GUARANTEES SAFETY AND LIVENESS, THE LATTER ONLY IN PERIODS OF PARTIALLY SYNCHRONY.

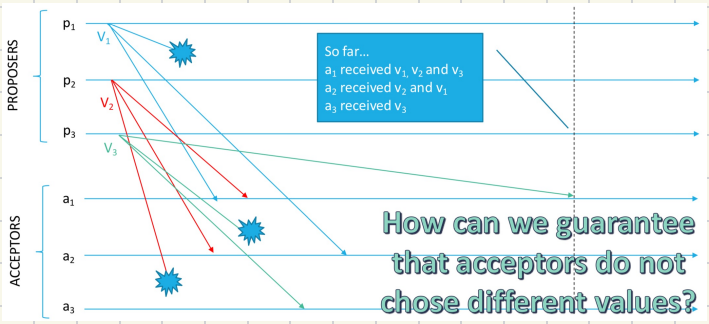
AGENTS OPERATE AT ARBITRARY SPEED, MAY FAIL BY STOPPING (AND MAY RESTART), AND MUST REMEMBER INFORMATION BETWEEN CRASHES AND RECOVERIES. MESSAGES CAN TAKE ARBITRARLY LONG TO BE DELIVERED, CAN BE DUPLICATED, AND CAN BE LOST, BUT THEY ARE NOT CORRUPTED.

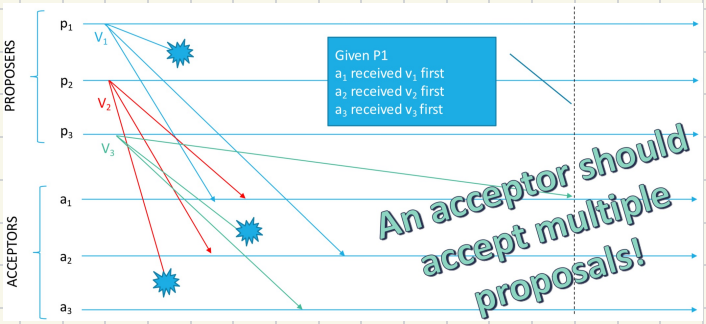
THERE ARE SOME ACTORS:

- **PROPOSERS:** PROPOSE A VALUE.
- **ACCEPTORS:** THEY DECIDE ON A PROPOSED VALUE.
- **LEARNERS:** THEY OBSERVE THE PROCESS AND LEARN THE DECIDED VALUE.

CHOOSING A VALUE

A SINGLE ACCEPTOR COULD ACCEPT THE FIRST PROPOSED VALUE AND NOTIFY EVERYONE BUT IF IT FAILS, CONSENSUS IS NOT POSSIBLE. A MAJORITY OF ACCEPTORS MUST BE INVOLVED TO ENSURE THAT A VALUE IS CHOSEN EVEN IN THE EVENT OF FAILURES.

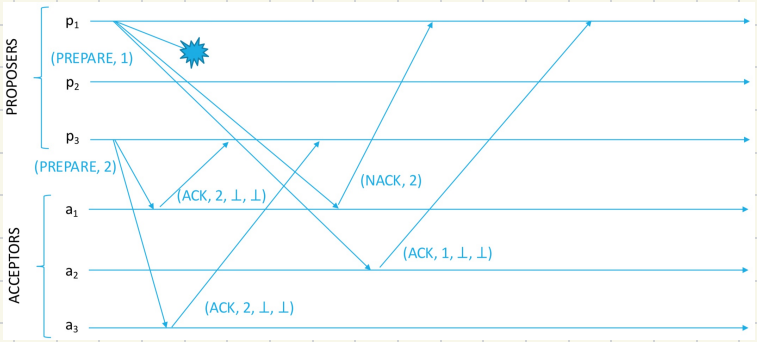




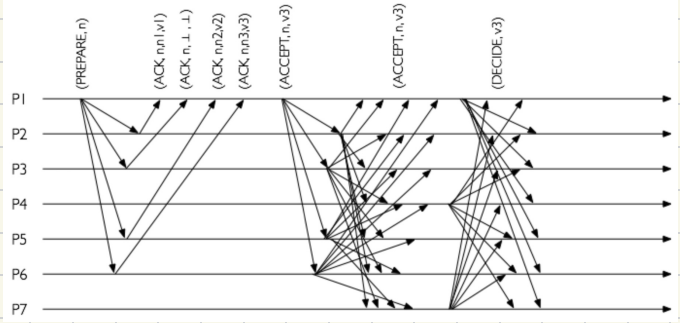
EACH PROPOSAL HAS A UNIQUE NUMBER m WHICH DETERMINES THE ORDER. THE ASSUMPTION THAT m HAS BEEN ACCEPTED IMPLIES THAT EVERY ACCEPTORS IN C (SET OF THE MAJORITY ACCEPTORS) HAS ACCEPTED A PROPOSAL WITH NUMBER IN [m, n-1] AND EVERY PROPOSAL IN THIS RANGE ACCEPTED HAS VALUE v.

PROTOCOL PHASES

PHASE 1. A PROPOSER CHOOSES A PROPOSAL NUMBER n AND SENDS A PREPARE(n) TO A MAJORITY OF ACCEPTORS.
EACH ACCEPTOR RESPONDS WITH A PROMISE NOT TO ACCEPT PROPOSALS NUMBERED LESS THAN n, AND THE VALUE v' OF THE HIGHEST-NUMBERED PROPOSAL IT ACCEPTED (IF IT EXISTS).



PHASE 2. IF THE PROPOSER RECEIVES RESPONSES FROM A MAJORITY OF ACCEPTORS, IT DETERMINES THE PROPOSAL VALUE v AND SENDS AN ACCEPTANCE REQUEST ACCEPT(n, v) TO THE ACCEPTORS.
THE ACCEPTORS ACCEPT THE PROPOSAL (n,v) IF THEY HAVE NOT RESPONDED TO A PREPARE WITH A NUMBER GREATER THAN n.



WHEN A MAJORITY OF ACCEPTORS ACCEPT A PROPOSAL, THEY NOTIFY THE LEARNERS, WHO LEARN THE VALUE v AND CONSIDER IT DECIDED.

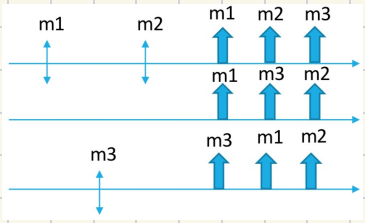
ORDERED COMMUNICATION

LESSON 13, 14

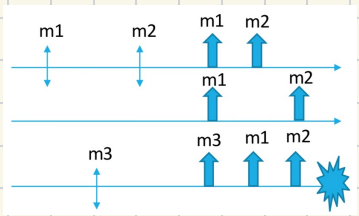
ORDERED COMMUNICATION PROVIDES GUARANTEES ABOUT THE ORDER OF DELIVERY OF MESSAGES IN A GROUP OF PROCESSES. THIS IS CRUCIAL TO AVOID ANOMALIES IN DS.

FIFO BROADCAST

IT ENSURES THAT ONE PROCESS DELIVERS MESSAGES SENT BY ANOTHER IN THE SAME ORDER IN WHICH THEY WERE TRANSMITTED.



FIFO URB: ALL CORRECT PROCESSES DELIVER MESSAGES IN THE SAME FIFO ORDER.



FIFO RRB: SOME PROCESSES MAY DELIVER MESSAGES IN A DIFFERENT ORDER IN CASE OF FAILURES.

Module 3.8: Interface and properties of FIFO-order (reliable) broadcast

Module:

Name: FIFOReliableBroadcast, **instance** *frb*.

Events:

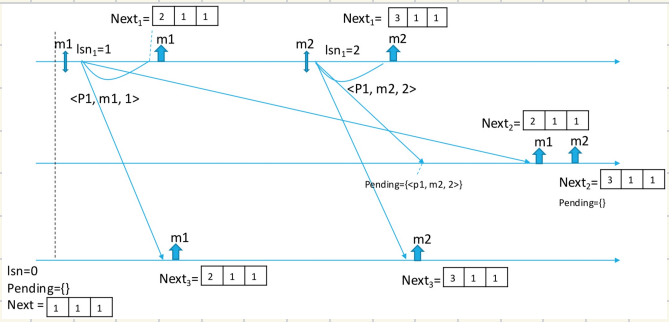
Request: ⟨ *frb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *frb*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p.

Properties:

FRB1–FRB4: Same as properties RB1–RB4 in (regular) reliable broadcast (Module 3.2).

FRB5: *FIFO delivery:* If some process broadcasts message m_1 before it broadcasts message m_2, then no correct process delivers m_2 unless it has already delivered m_1.



Algorithm 3.12: Broadcast with Sequence Number

Implements:
FIFOReliableBroadcast, **instance** *frb*.

Uses:
ReliableBroadcast, **instance** *rb*.

upon event ⟨ *frb*, *Init* ⟩ **do**
$lsn := 0$;
$pending := \emptyset$;
$next := [1]^N$;

upon event ⟨ *frb*, *Broadcast* | m ⟩ **do**
$lsn := lsn + 1$;
trigger ⟨ *rb*, *Broadcast* | [DATA, *self*, m, lsn] ⟩;

upon event ⟨ *rb*, *Deliver* | p, [DATA, s, m, sn] ⟩ **do**
$pending := pending \cup \{(s, m, sn)\}$;
while exists $(s, m', sn') \in pending$ such that $sn' = next[s]$ **do**
$next[s] := next[s] + 1$;
$pending := pending \setminus \{(s, m', sn')\}$;
trigger ⟨ *frb*, *Deliver* | s, m' ⟩;

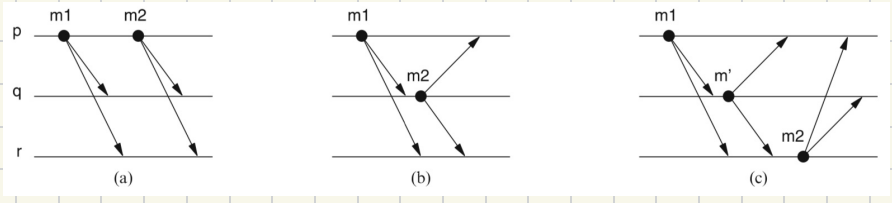
CASUAL ORDER BROADCAST

IT'S AN EXTENSION OF THE HAPPENED-BEFORE RELATION AND ENSURES THAT MESSAGES ARE DELIVERED ACCORDING TO THE CAUSE-EFFECT RELATIONSHIPS.

A MESSAGE m_1 CAUSE m_2 IF.

a) A PROCESS P SEND m_1 BEFORE m_2.
b) A PROCESS P DELIVERY m_1 AND THEN SEND m_2.
c) THERE IS AN INTERMEDIATE MESSAGE CONNECTING m_1 TO m_2.

CASUAL ORDER ⇒ FIFO ORDER
FIFO ORDER ⇏ CASUAL ORDER



Module 3.9: Interface and properties of causal-order (reliable) broadcast

Module:

Name: CausalOrderReliableBroadcast, **instance** *crb*.

Events:

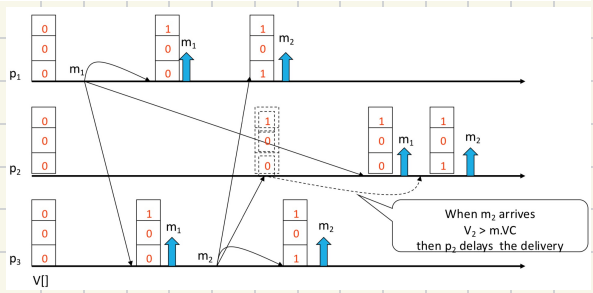
Request: ⟨ *crb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *crb*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p.

Properties:

CRB1–CRB4: Same as properties RB1–RB4 in (regular) reliable broadcast (Module 3.2).

CRB5: *Causal delivery:* For any message m_1 that potentially caused a message m_2, i.e., $m_1 \rightarrow m_2$, no process delivers m_2 unless it has already delivered m_1.



Algorithm 3.15: Waiting Causal Broadcast

Implements:
CausalOrderReliableBroadcast, **instance** *crb*.

Uses:
ReliableBroadcast, **instance** *rb*.

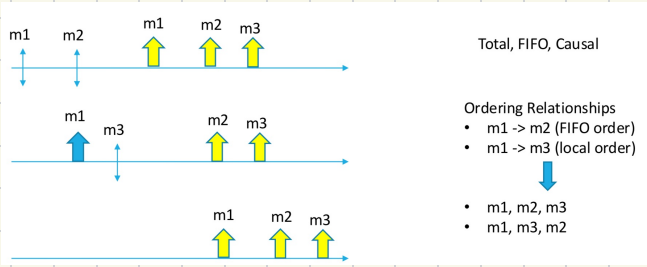
upon event ⟨ *crb*, *Init* ⟩ **do**
$V := [0]^N$;
$lsn := 0$;
$pending := \emptyset$;

upon event ⟨ *crb*, *Broadcast* | m ⟩ **do**
$W := V$;
$W[rank(self)] := lsn$;
$lsn := lsn + 1$;
trigger ⟨ *rb*, *Broadcast* | [DATA, W, m] ⟩;

upon event ⟨ *rb*, *Deliver* | p, [DATA, W, m] ⟩ **do**
$pending := pending \cup \{(p, W, m)\}$;
while exists $(p', W', m') \in pending$ such that $W' \leq V$ **do**
$pending := pending \setminus \{(p', W', m')\}$;
$V[rank(p')] := V[rank(p')] + 1$;
trigger ⟨ *crb*, *Deliver* | p', m' ⟩;

TOTAL ORDER BROADCAST (TO)

IT ENSURES THAT ALL PROCESSES DELIVER MESSAGES IN THE SAME GLOBAL ORDER. UNLIKE CASUAL ORDER, IT DOESN'T CONSIDER CASUALITY, BUT REQUIRES THAT ALL PROCESSES AGREE ON A SINGLE ORDER. IT'S ORTHOGONAL WITH RESPECT TO FIFO AND CASUAL ORDER.



TO SPECIFICATIONS ARE USUALLY COMPOSED BY FOUR PROPERTIES:

- VALIDITY (V): GUARANTEES THAT MESSAGES SENT BY CORRECT PROCESSES WILL EVENTUALLY BE DELIVERED.
- INTEGRITY (I): GUARANTEES THAT NO DUPLICATE MESSAGES ARE DELIVERED.
- AGREEMENT (A): IF ONE PROCESS (CORRECT OR FAILED) DELIVERS A MESSAGE m, THEN ALL CORRECT PROCESSES MUST DELIVER m.
- ORDER (O): ALL CORRECT PROCESSES DELIVER MESSAGES IN THE SAME ORDER.

STRONG UNIFORM TOTAL ORDER (SUTO): IF SOME PROCESS TODELIVERS SOME MESSAGE m BEFORE MESSAGE m', THEN A PROCESS TODELIVERS m' ONLY AFTER IT HAS TODELIVERED m

WEAK UNIFORM TOTAL ORDER (WUTO): IF PROCESS p AND PROCESS q BOTH TODELIVER MESSAGES m AND m', THEN p TODELIVERS m BEFORE m' IF AND ONLY IF q TODELIVERS m BEFORE m'.

Algorithm 6.1: Consensus-Based Total-Order Broadcast

Implements:
TotalOrderBroadcast, **instance** *tob*.

Uses:
ReliableBroadcast, **instance** *rb*;
Consensus (multiple instances).

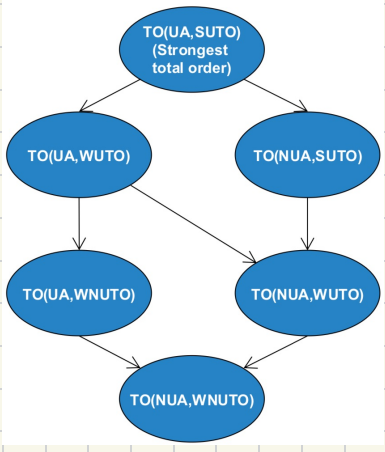
upon event ⟨ *tob*, *Init* ⟩ **do**
$unordered := \emptyset$;
$delivered := \emptyset$;
$round := 1$;
$wait :=$ FALSE;

upon event ⟨ *tob*, *Broadcast* | m ⟩ **do**
trigger ⟨ *rb*, *Broadcast* | m ⟩;

upon event ⟨ *rb*, *Deliver* | p, m ⟩ **do**
if $m \notin delivered$ **then**
$unordered := unordered \cup \{(p, m)\}$;

upon $unordered \neq \emptyset \wedge wait =$ FALSE **do**
$wait :=$ TRUE;
Initialize a new instance $c.round$ of consensus;
trigger ⟨ $c.round$, *Propose* | $unordered$ ⟩;

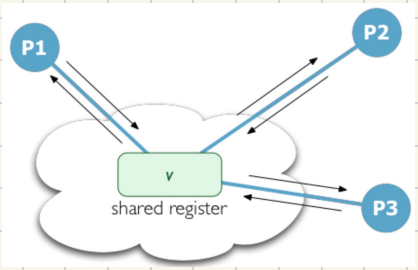
upon event ⟨ $c.r$, *Decide* | $decided$ ⟩ **such that** $r = round$ **do**
forall $(s, m) \in sort(decided)$ **do**
trigger ⟨ *tob*, *Deliver* | s, m ⟩;
$delivered := delivered \cup decided$;
$unordered := unordered \setminus decided$;
$round := round + 1$;
$wait :=$ FALSE;



REGISTERS

LESSON 15,16

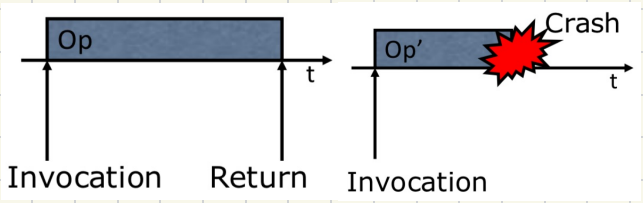
A REGISTER IS A SHARED VARIABLE ACCESSED BY PROCESSES THROUGH READ AND WRITE OPERATIONS.

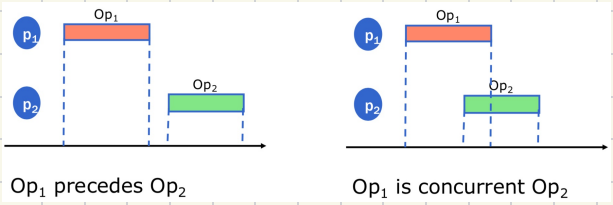


READ()→V: RETURNS THE CURRENT VALUE v OF THE REGISTER WITHOUT MODIFYING ITS CONTENT

WRITE(v): WRITES THE VALUE v IN THE REGISTER AND RETURNS TRUE AT THE END OF THE OPERATION.

(X,Y) DENOTES A REGISTER WHERE X PROCESSES CAN WRITE AND Y CAN READ.





GIVEN TWO OPERATIONS O_1 AND O_2, O_1 PRECEDES O_2 IF THE RESPONSE EVENT OF O_1 PRECEDES THE INVOCATION EVENT OF O_2. IF IT'S NOT POSSIBLE TO DEFINE A PRECEDENCE RELATION BETWEEN TWO OP, THEY ARE SAID TO BE CONCURRENT.

REGULAR REGISTER

IF A READ() IS NOT CONCURRENT WITH A WRITE(), IT RETURNS THE LAST WRITTEN VALUE INSTEAD, IF A READ() IS CONCURRENT WITH A WRITE(), IT CAN RETURN THE LAST WRITTEN VALUE OR THE VALUE OF THE CONCURRENT WRITE.

Module 4.1: Interface and properties of a $(1, N)$ regular register

Module:

Name: $(1, N)$-RegularRegister, **instance** *onrr*.

Events:

Request: ⟨ *onrr*, *Read* ⟩: Invokes a read operation on the register.

Request: ⟨ *onrr*, *Write* | *v* ⟩: Invokes a write operation with value *v* on the register.

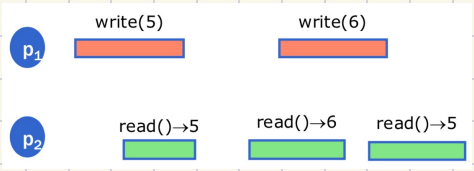
Indication: ⟨ *onrr*, *ReadReturn* | *v* ⟩: Completes a read operation on the register with return value *v*.

Indication: ⟨ *onrr*, *WriteReturn* ⟩: Completes a write operation on the register.

Properties:

ONRR1: *Termination:* If a correct process invokes an operation, then the operation eventually completes.

ONRR2: *Validity:* A read that is not concurrent with a write returns the last value written; a read that is concurrent with a write returns the last value written or the value concurrently written.



FAIL-STOP ALGORITHM: PROCESSES THAT FAIL STOP PARTICIPATING IN THE COMPUTATION (THEY ARE IGNORED) AND THE FAILURE IS DETECTABLE. IN THE **READ-ONE-WRITE-ALL** IMPLEMENTATION EACH PROCESS MAINTAINS A LOCAL COPY OF THE REGISTRY. WRITING UPDATES ALL COPIES OF THE CORRECT PROCESSES, AND READING USES ONLY THE LOCAL COPY.

Algorithm 4.1: Read-One Write-All

Implements:
$(1, N)$-RegularRegister, **instance** *onrr*.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *onrr*, *Init* ⟩ **do**
val := ⊥;
correct := *Π*;
writeset := ∅;

upon event ⟨ $\mathcal{P}$, *Crash* | *p* ⟩ **do**
correct := *correct* \ {*p*};

upon event ⟨ *onrr*, *Read* ⟩ **do**
trigger ⟨ *onrr*, *ReadReturn* | *val* ⟩;

upon event ⟨ *onrr*, *Write* | *v* ⟩ **do**
trigger ⟨ *beb*, *Broadcast* | [WRITE, *v*] ⟩;

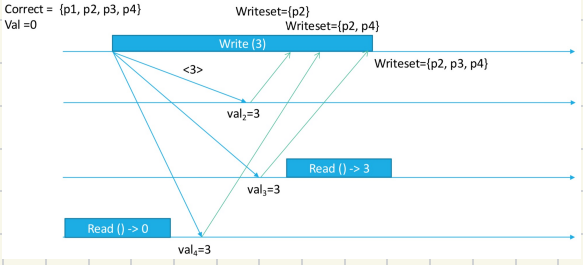
upon event ⟨ *beb*, *Deliver* | *q*, [WRITE, *v*] ⟩ **do**
val := *v*;
trigger ⟨ *pl*, *Send* | *q*, ACK ⟩;

upon event ⟨ *pl*, *Deliver* | *p*, ACK ⟩ **do**
writeset := *writeset* ∪ {*p*};

upon *correct* ⊆ *writeset* **do**
writeset := ∅;
trigger ⟨ *onrr*, *WriteReturn* ⟩;

READ

WRITE



FAIL-SILENT ALGORITHM: FAILURES ARE UNDETECTABLE AND PROCESSES CAN BEHAVE ARBITRARILY. IN THE MAJORITY VOTING IMPLEMENTATION EACH WRITE OPERATION ASSIGNS A TIMESTAMP TO THE WRITTEN VALUE. READS AGGREGATE DATA FROM A MAJORITY OF PROCESSES (QUORUM) AND RETURN THE MOST RECENT VALUE (BASED ON TIMESTAMP).

Algorithm 4.2: Majority Voting Regular Register

Implements:
$(1, N)$-RegularRegister, **instance** *onrr*.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*.

upon event $\langle$ *onrr*, *Init* $\rangle$ **do**
$(ts, val) := (0, \perp)$;
$wts := 0$;
$acks := 0$;
$rid := 0$;
$readlist := [\perp]^N$;

upon event $\langle$ *onrr*, *Write* | v $\rangle$ **do**
$wts := wts + 1$;
$acks := 0$;
trigger $\langle$ *beb*, *Broadcast* | [WRITE, wts, v] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [WRITE, ts', v'] $\rangle$ **do**
if $ts' > ts$ **then**
$(ts, val) := (ts', v')$;
trigger $\langle$ *pl*, *Send* | p, [ACK, ts'] $\rangle$;

upon event $\langle$ *pl*, *Deliver* | q, [ACK, ts'] $\rangle$ **such that** $ts' = wts$ **do**
$acks := acks + 1$;
if $acks > N/2$ **then**
$acks := 0$;
trigger $\langle$ *onrr*, *WriteReturn* $\rangle$;

upon event $\langle$ *onrr*, *Read* $\rangle$ **do**
$rid := rid + 1$;
$readlist := [\perp]^N$;
trigger $\langle$ *beb*, *Broadcast* | [READ, rid] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [READ, r] $\rangle$ **do**
trigger $\langle$ *pl*, *Send* | p, [VALUE, r, ts, val] $\rangle$;

upon event $\langle$ *pl*, *Deliver* | q, [VALUE, r, ts', v'] $\rangle$ **such that** $r = rid$ **do**
$readlist[q] := (ts', v')$;
if $\#(readlist) > N/2$ **then**
$v :=$ highestval($readlist$);
$readlist := [\perp]^N$;
trigger $\langle$ *onrr*, *ReadReturn* | v $\rangle$;

ATOMIC REGISTER

IT'S A REGULAR REGISTER BUT GUARANTEES TOTAL ORDER BETWEEN READ() AND WRITE() OP. A SUBSEQUENT READ() CANNOT RETURN A VALUE WRITTEN BEFORE A VALUE ALREADY READ.

Module 4.2: Interface and properties of a $(1, N)$ atomic register

Module:

Name: $(1, N)$-AtomicRegister, **instance** *onar*.

Events:

Request: $\langle$ *onar*, *Read* $\rangle$: Invokes a read operation on the register.

Request: $\langle$ *onar*, *Write* | v $\rangle$: Invokes a write operation with value v on the register.

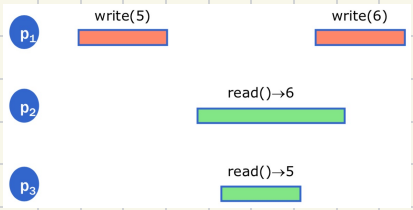
Indication: $\langle$ *onar*, *ReadReturn* | v $\rangle$: Completes a read operation on the register with return value v.

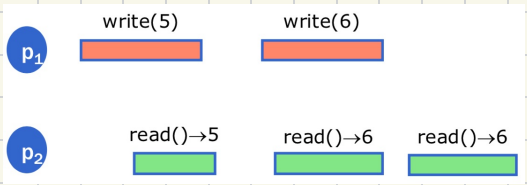
Indication: $\langle$ *onar*, *WriteReturn* $\rangle$: Completes a write operation on the register.

Properties:

ONAR1–ONAR2: Same as properties ONRR1–ONRR2 of a $(1, N)$ regular register (Module 4.1).

ONAR3: *Ordering:* If a read returns a value v and a subsequent read returns a value w, then the write of w does not precede the write of v.





THE ALGORITHM CONSISTS OF TWO PHASES:

PHASE 1: WE USE A (1,N) REGULAR REGISTER TO BUILD A (1,1) ATOMIC REGISTER. IT ENSURES CONSISTENCY BETWEEN A SINGLE WRITER AND READER.

Algorithm 4.3: From $(1, N)$ Regular to $(1, 1)$ Atomic Registers

Implements:
$(1, 1)$-AtomicRegister, **instance** *ooar*.

Uses:
$(1, N)$-RegularRegister, **instance** *onrr*.

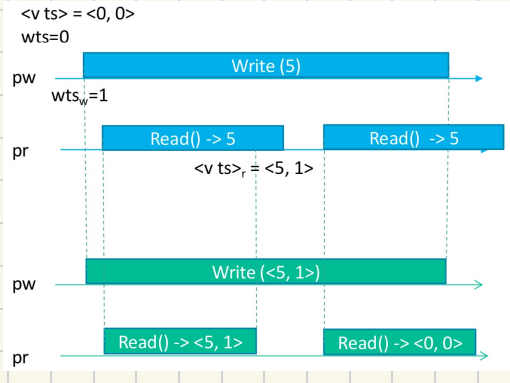
upon event $\langle$ *ooar*, *Init* $\rangle$ **do**
$(ts, val) := (0, \perp)$;
$wts := 0$;

upon event $\langle$ *ooar*, *Write* | v $\rangle$ **do**
$wts := wts + 1$;
trigger $\langle$ *onrr*, *Write* | (wts, v) $\rangle$;

upon event $\langle$ *onrr*, *WriteReturn* $\rangle$ **do**
trigger $\langle$ *ooar*, *WriteReturn* $\rangle$;

upon event $\langle$ *ooar*, *Read* $\rangle$ **do**
trigger $\langle$ *onrr*, *Read* $\rangle$;

upon event $\langle$ *onrr*, *ReadReturn* | (ts', v') $\rangle$ **do**
if $ts' > ts$ **then**
$(ts, val) := (ts', v')$;
trigger $\langle$ *ooar*, *ReadReturn* | val $\rangle$;



PHASE 2: WE USE A SET OF (1,1) ATOMIC REGISTER TO BUILD A (1,N) ATOMIC REGISTER. IT EXPANDS CONSISTENCY TO MULTIPLE PROCESSES.

Algorithm 4.4: From $(1, 1)$ Atomic to $(1, N)$ Atomic Registers

Implements:
$(1, N)$-AtomicRegister, **instance** *onar*.

Uses:
$(1, 1)$-AtomicRegister (multiple instances).

upon event ⟨ *onar*, *Init* ⟩ **do**
  $ts := 0$;
  $acks := 0$;
  writing := FALSE;
  readval := $\perp$;
  readlist := $[\perp]^N$;
  forall $q \in \Pi, r \in \Pi$ **do**
    Initialize a new instance *ooar.q.r* of $(1, 1)$-AtomicRegister with writer r and reader q;

upon event ⟨ *onar*, *Write* | v ⟩ **do**
  $ts := ts + 1$;
  writing := TRUE;
  forall $q \in \Pi$ **do**
    trigger ⟨ *ooar.q.self*, *Write* | (ts, v) ⟩;

upon event ⟨ *ooar.q.self*, *WriteReturn* ⟩ **do**
  $acks := acks + 1$;
  if $acks = N$ **then**
    $acks := 0$;
    if *writing* = TRUE **then**
      trigger ⟨ *onar*, *WriteReturn* ⟩;
      writing := FALSE;
    else
      trigger ⟨ *onar*, *ReadReturn* | *readval* ⟩;

upon event ⟨ *onar*, *Read* ⟩ **do**
  forall $r \in \Pi$ **do**
    trigger ⟨ *ooar.self.r*, *Read* ⟩;

upon event ⟨ *ooar.self.r*, *ReadReturn* | (ts', v') ⟩ **do**
  readlist[r] := (ts', v');
  if $\#(readlist) = N$ **then**
    $(maxts, readval) := highest(readlist)$;
    readlist := $[\perp]^N$;
    forall $q \in \Pi$ **do**
      trigger ⟨ *ooar.q.self*, *Write* | $(maxts, readval)$ ⟩;

FAIL-STOP ALGORITHM: IN THE **READ-IMPOSE-WRITE-ALL** IMPLEMENTATION A READ OPERATION IMPOSES TO ALL CORRECT PROCESSES TO UPDATE THEIR LOCAL COPY OF THE REGISTER WITH THE VALUE READ, UNLESS THEY STORE A MORE RECENT VALUE.

Algorithm 4.5: Read-Impose Write-All

Implements:
$(1, N)$-AtomicRegister, **instance** *onar*.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *onar*, *Init* ⟩ **do**
  $(ts, val) := (0, \perp)$;
  correct := Π;
  writeset := $\emptyset$;
  readval := $\perp$;
  reading := FALSE;

upon event ⟨ $\mathcal{P}$, *Crash* | p ⟩ **do**
  correct := *correct* $\setminus \{p\}$;

upon event ⟨ *onar*, *Read* ⟩ **do**
  reading := TRUE;
  readval := *val*;
  trigger ⟨ *beb*, *Broadcast* | [WRITE, ts, *val*] ⟩;

upon event ⟨ *onar*, *Write* | v ⟩ **do**
  trigger ⟨ *beb*, *Broadcast* | [WRITE, $ts + 1$, v] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [WRITE, ts', v'] ⟩ **do**
  if $ts' > ts$ **then**
    $(ts, val) := (ts', v')$;
  trigger ⟨ *pl*, *Send* | p, [ACK] ⟩;

upon event ⟨ *pl*, *Deliver* | p, [ACK] ⟩ **then**
  writeset := *writeset* $\cup \{p\}$;

upon *correct* $\subseteq$ *writeset* **do**
  writeset := $\emptyset$;
  if *reading* = TRUE **then**
    reading := FALSE;
    trigger ⟨ *onar*, *ReadReturn* | *readval* ⟩;
  else
    trigger ⟨ *onar*, *WriteReturn* ⟩;

READ

WRITE

FAIL-SILENT ALGORITHM: **READ-IMPOSE-WRITE-MAJORITY** IS SIMILIAR TO THE PREVIOUS ALGORITHM, BUT THE OPERATIONS INVOLVE ONLY A QUORUM.

Algorithm 4.6: Read-Impose Write-Majority (part 1, read)

Implements:
$(1, N)$-AtomicRegister, **instance** *onar*.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*.

upon event ⟨ *onar*, *Init* ⟩ **do**
  $(ts, val) := (0, \perp)$;
  $wts := 0$;
  $acks := 0$;
  $rid := 0$;
  readlist := $[\perp]^N$;
  readval := $\perp$;
  reading := FALSE;

upon event ⟨ *onar*, *Read* ⟩ **do**
  $rid := rid + 1$;
  $acks := 0$;
  readlist := $[\perp]^N$;
  reading := TRUE;
  trigger ⟨ *beb*, *Broadcast* | [READ, *rid*] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [READ, r] ⟩ **do**
  trigger ⟨ *pl*, *Send* | p, [VALUE, r, ts, *val*] ⟩;

upon event ⟨ *pl*, *Deliver* | q, [VALUE, r, ts', v'] ⟩ **such that** $r = rid$ **do**
  readlist[q] := (ts', v');
  if $\#(readlist) > N/2$ **then**
    $(maxts, readval) := highest(readlist)$;
    readlist := $[\perp]^N$;
    trigger ⟨ *beb*, *Broadcast* | [WRITE, *rid*, *maxts*, *readval*] ⟩;

upon event ⟨ *onar*, *Write* | v ⟩ **do**
  $rid := rid + 1$;
  $wts := wts + 1$;
  $acks := 0$;
  trigger ⟨ *beb*, *Broadcast* | [WRITE, *rid*, wts, v] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [WRITE, r, ts', v'] ⟩ **do**
  if $ts' > ts$ **then**
    $(ts, val) := (ts', v')$;
  trigger ⟨ *pl*, *Send* | p, [ACK, r] ⟩;

upon event ⟨ *pl*, *Deliver* | q, [ACK, r] ⟩ **such that** $r = rid$ **do**
  $acks := acks + 1$;
  if $acks > N/2$ **then**
    $acks := 0$;
    if *reading* = TRUE **then**
      reading := FALSE;
      trigger ⟨ *onar*, *ReadReturn* | *readval* ⟩;
    else
      trigger ⟨ *onar*, *WriteReturn* ⟩;

SOFTWARE REPLICATION

LESSON 17

REPLICATION IS ESSENTIAL TO GUARANTEE FAULT TOLERANCE AND MAINTAIN THE AVAILABILITY OF A SERVICE EVEN IN THE PRESENCE OF FAILURES.

AN OBJECT O WITH FAILURE PROBABILITY p HAS AN AVAILABILITY EQUAL TO $1-p$. IF O IS REPLICATED ACROSS n INDEPENDENT NODES (EACH WITH FAILURE PROB p), AVAILABILITY INCREASES TO $1-p^n$.

CLIENT PROCESSES (SET OF PROCESSES) INTERACT WITH DISTRIBUTED OBJECTS $X = \{x_1, x_2, \dots, x_n\}$ MANAGED BY PROCESSES AT VARIOUS SITES. PROCESSES ARE CONNECTED THROUGH PP2PL AND THEY MAY FAIL BY CRASH.

REQUIREMENTS

TO TOLERATE FAULTS, EVERY LOCAL OBJECT X MUST HAVE MULTIPLE PHYSICAL REPLICAS ($x_1, x_2, \dots, x_n$).
CLIENTS MUST INTERACT WITH THE OBJECT AS IF IT WERE UNIQUE, WITHOUT KNOWING ABOUT REPLICAS.

CONSISTENCY CRITERIA

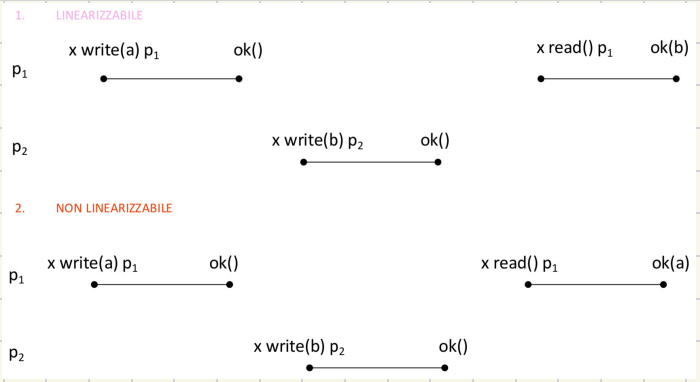
A CONSISTENCY CRITERION DEFINES THE RESULT RETURNED BY AN OPERATION.

LINEARIZABILITY: EACH OPERATION MUST APPEAR AS IF IT WERE PERFORMED ISTANTANEOUSLY AT A UNIQUE POINT IN TIME.

STRONG

AN EXECUTION E IS LINEARIZABLE IF:

1. RESPECT THE ORDER OF PRECEDENCE OF OPERATIONS $O_1 < O_2$ DURING EXECUTION.
2. THE SEQUENCE MUST BE LEGAL WITH RESPECT TO THE SEQUENTIAL SPECIFICATION OF THE OBJECT



Sequential Consistency ⇒ Causal Consistency
Causal Consistency ⇏ Sequential Consistency

SEQUENTIAL CONSISTENCY: THE OPERATIONS APPEAR TO BE PERFORMED IN A ORDER CONSISTENT WITH THE LOCAL SEQUENCES OF PROCESSES, BUT NOT NECESSARILY LINEARIZABLE.

STRONG

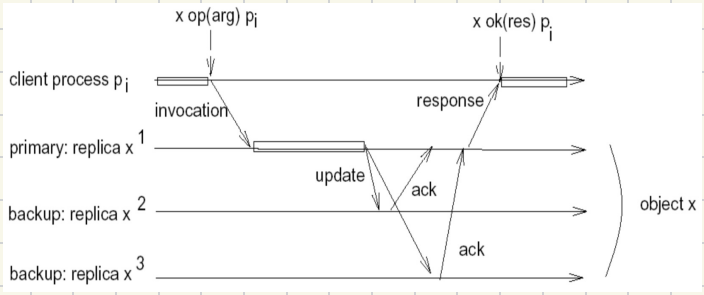
CASUAL CONSISTENCY: IF AN OPERATION O_1 CAUSE O_2, THE ORDER $O_1 < O_2$ MUST BE RESPECTED.

WEAK

PASSIVE REPLICATION: PRIMARY BACKUP

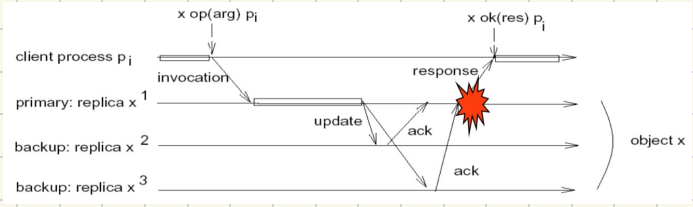
PRIMARY: RECEIVES INVOCATIONS FROM CLIENTS AND SENDS BACK THE ANSWERS.
BACKUP: UPDATES ITS STATUS BASED ON COMMANDS RECEIVED FROM THE PRIMARY.

a. THE PRIMARY RECEIVES THE OPERATION, UPDATES THE BACKUPS AND WAITS FOR THE ACKS
b. AFTER RECEIVING THE ACKS, THE PRIMARY SENDS THE RESULT TO THE CLIENT.

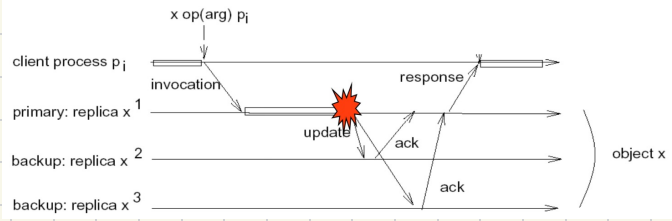


IF THE PRIMARY FAILS, A NEW PRIMARY IS ELECTED AMONG THE BACKUPS.
WE HAVE THREE SCENARIOS:

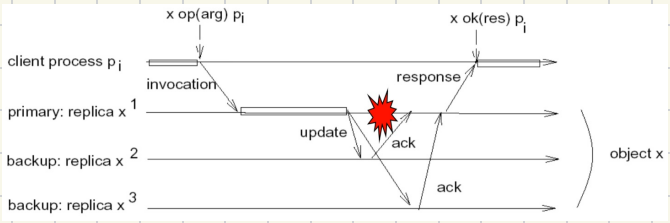
CRASH AFTER RESPONSE: CLIENT CAN RESEND REQUEST. THE NEW PRIMARY RECOGNIZES THAT THE RESPONSE HAS ALREADY BEEN PROCESSED AND RESENDS IT IF NECESSARY.



CRASH BEFORE UPDATES: CLIENT DOESN'T GET AN ANSWER AND RESENDS THE REQUEST. THE NEW PRIMARY TREATS THE REQUEST AS NEW.

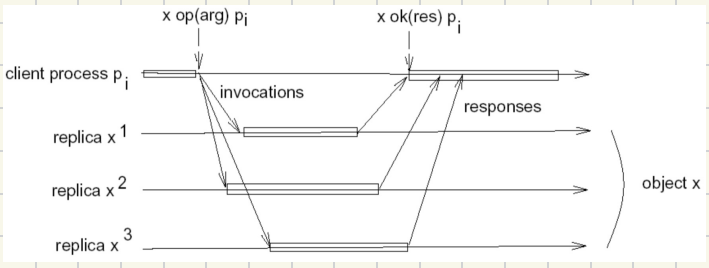


CRASH DURING UPDATES: THE NEW PRIMARY CHECKS THE CONSISTENCY STATE.



ACTIVE REPLICATION

THE REPLICAS START FROM THE SAME STATE, PERFORM THE SAME OPERATIONS IN THE SAME ORDER VIA TOTAL ORDER BROADCAST, AND PRODUCE THE SAME OUTPUT INDEPENDENTLY.



NO RECOVERY ACTION IS NECESSARY IN THE EVENT OF A REPLICA FAILURE

CAP THEOREM AND PARTITION MANAGEMENT FOR DISTRIBUTED SHARED MEMORIES

LESSON 18,19

CAP THEOREM

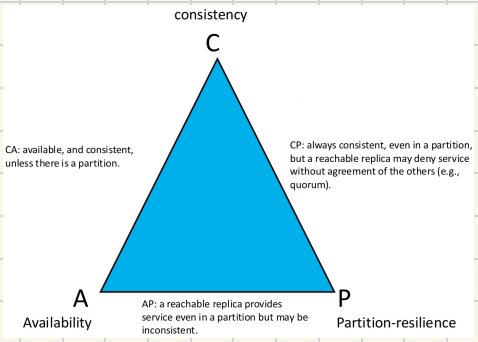
STATES THAT IN DISTRIBUTED SYSTEMS WITH SHARED DATA, IT IS POSSIBLE TO GUARANTEE AT MOST TWO OF THESE THREE PROPERTIES SIMULTANEOUSLY:

CONSISTENCY (C): ALL NODES IN THE SYSTEM SEE THE SAME DATA AT THE SAME TIME

AVAILABILITY (A): EVERY REQUEST RECEIVES A RESPONSE, EVEN IN THE EVENT OF FAILURES.

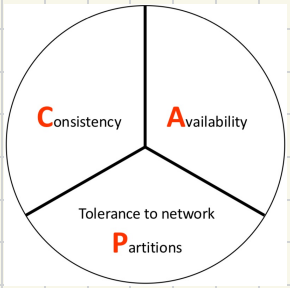
PARTITION TOLERANCE (P): THE SYSTEM CONTINUES TO FUNCTION CORRECTLY DESPITE PARTITIONS IN THE NETWORK.

RARE EVENT



C + A:
SYSTEMS WITHOUT NETWORK PARTITIONS, SUCH AS CENTRALIZED DATABASES.

A + P:
SYSTEMS LIKE DNS OR DISTRIBUTED CACHING.

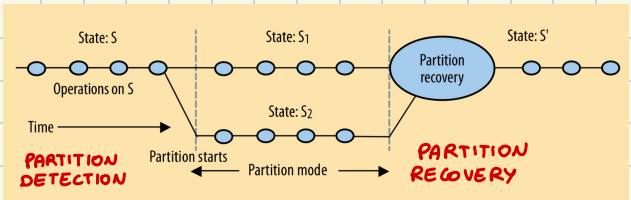


C + P:
SYSTEMS THAT SACRIFICE AVAILABILITY FOR PARTITIONS, SUCH AS SOME DISTRIBUTED DATABASES WITH QUORUMS.

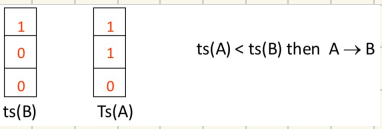
CONSISTENCY REQUIRES COMMUNICATION BETWEEN NODES, INCREASING LATENCY
SYSTEMS MUST BALANCE LATENCY, AVAILABILITY AND CONSISTENCY BASED ON APPLICATION REQUIREMENTS.

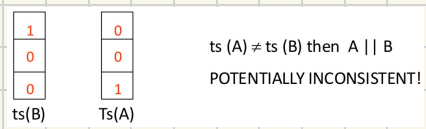
PARTITION MANAGEMENT

PARTITION DETECTION: TIMEOUT BASED TO IDENTIFY IF A PARTITION IS IN PROGRESS.



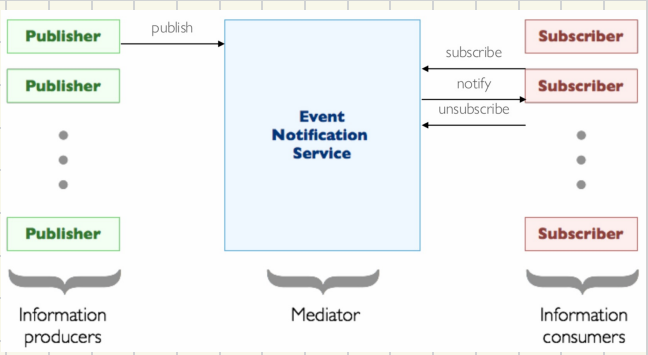
PARTITION RECOVERY: ROLL-BACK AND EXECUTE AGAIN OPERATIONS IN THE PROPER ORDER USING VERSION VECTORS.





DISABLE A SUBSET OF OPERATIONS USING COMMUTATIVE REPLICATED DATA TYPE (CRDT).

PUBLISH/SUBSCRIBE PARADIGM



PUBLISHER: PRODUCE DATA IN THE FORM OF EVENTS.
SUBSCRIBERS: DECLARE INTERESTS ON PUBLISHED DATA WITH SUBSCRIPTIONS.
EVENT NOTIFICATION SERVICE (ENS): NOTIFIES TO EACH SUBSCRIBER EVERY PUBLISHED EVENT THAT MATCHES AT LEAST ONE OF ITS SUBSCRIPTIONS.

THE PUB/SUB MODEL IS CHARACTERIZED BY:

MANY-TO-MANY COMMUNICATION. MANY PRODUCERS AND CONSUMERS CAN INTERACT AT THE SAME TIME. EVENTS CAN BE DISTRIBUTED TO MULTIPLE CONSUMERS AND RECEIVED BY MULTIPLE PRODUCERS.
SPACE DECOUPLING: INTERACTING PARTIES DO NOT NEED TO KNOW EACH OTHER.
TIME DECOUPLING: INTERACTING PARTIES DO NOT NEED TO BE ACTIVELY PARTICIPATING IN THE INTERACTION AT THE SAME TIME.
SYNCHRONIZATION DECOUPLING: SYNCHRONIZATION AMONG INTERACTING PARTIES IS NOT NEEDED.
PUSH/PULL: EVENT CAN BE SENT TO SUBSCRIBERS OR REQUESTED BY THEM.

EVENT SCHEMA

THE EVENTS ARE STRUCTURED ACCORDING TO A FIXED SCHEME KNOW TO ALL PARTICIPANTS, WHICH DEFINES A SET OF FIELDS OR ATTRIBUTES.

name	type	allowed values
blog_name	string	ANY
address	URL	ANY
genre	enumeration	[hardware, software, peripherals, development]
author	string	ANY
abstract	string	ANY
rating	integer	[1-5]
update_date	date	>1-1-1970 00:00

Event Schema

Event

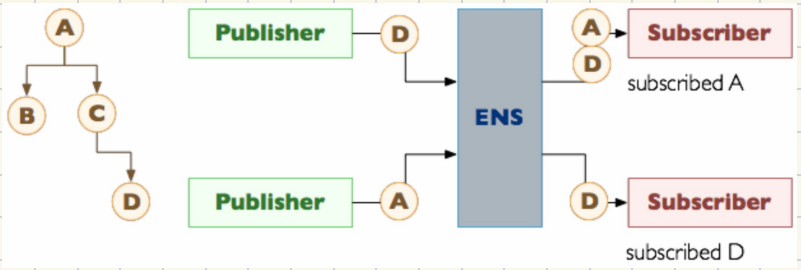
name	value
blog_name	Prad.de
address	http://www.prad.de/en/index.html
genre	peripherals
author	Mark Hansen
abstract	"The review of the new TFT panel..."
rating	4
update_date	26-4-2006 17:58

SUBSCRIPTION MODELS

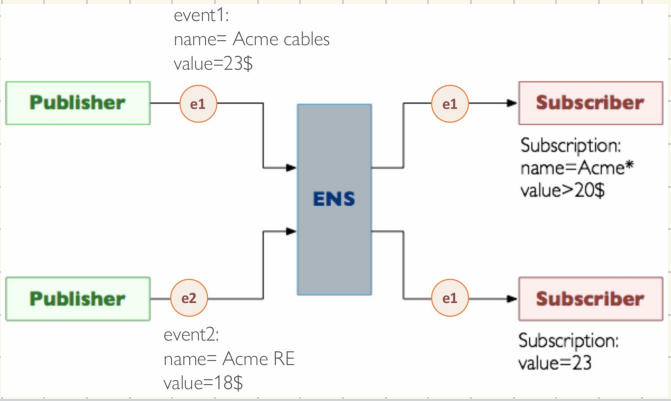
SUBSCRIBERS EXPRESS THEIR INTERESTS BY DEFING CONSTRAINTS ON EVENT SCHEMA ATTRIBUTES. DEPENDING ON THE SUBSCRIPTION MODEL USED WE DISTINGUISH VARIOUS FLAVORS OF PUB/SUB:

TOPIC-BASED: EVENTS ARE ASSOCIATED WITH A TOPIC. USERS SUBSCRIBE TO TOPICS THAT INTEREST THEM.

HIERARCHY-BASED: TOPICS ARE ORGANIZED HIERARCHICALLY, WITH NOTIONS OF CONTAINMENT



CONTENT-BASED: EVENTS ARE STRUCTURED, AND SUBSCRIPTIONS DEFINE CONSTRAINTS ON ATTRIBUTES.



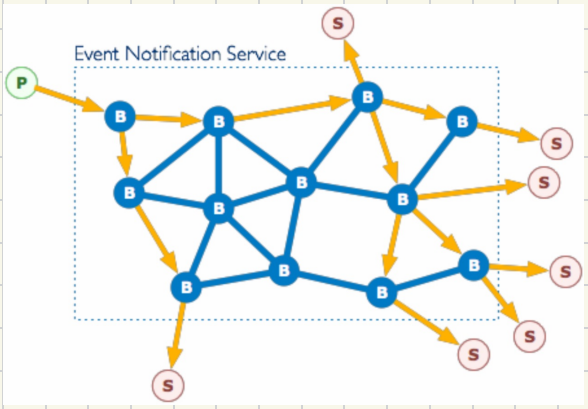
TYPE-BASED, XML-BASED, CONCEPT-BASED: MORE SPECIFIC MODELS.

ENS ARCHITECTURE

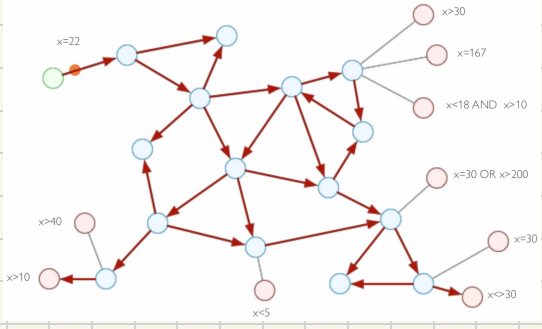
CENTRALIZED SERVICE: IMPLEMENTED ON A SINGLE SERVICE. LIMITED SCALABILITY.
DISTRIBUTED SERVICE: THE ENS IS COSTITUTED BY A SET OF NODES, EVENT BROKERS, WHICH COOPERATE TO IMPLEMENT THE SERVICE. USUALLY PREFERRED FOR LARGE SETTINGS.

EVENT ROUTING

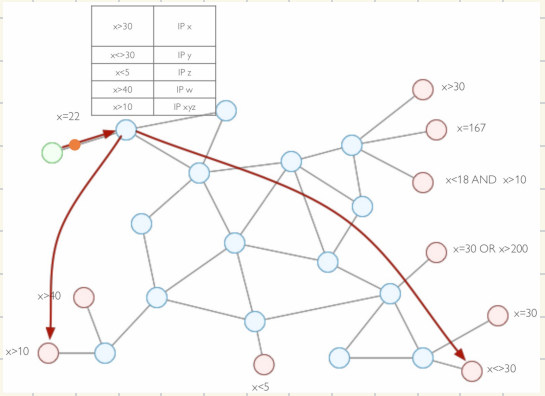
EACH CLIENT (P/S) ACCESS THE SERVICE THROUGH A BROKER THAT MASKS THE SYSTEM COMPLEXITY.
AN EVENT ROUTING MECHANISM ROUTES EACH EVENT INSIDE THE ENS FROM THE BROKER WHERE IT IS PUBLISHED TO THE BROKERS WHERE IT MUST BE NOTIFIED.



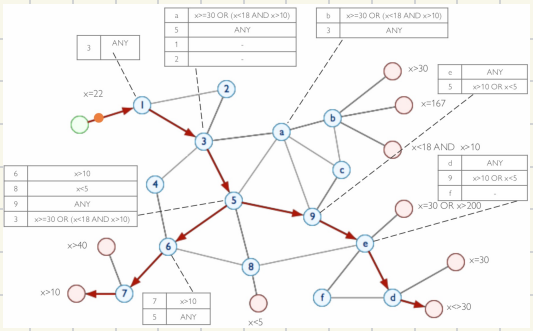
EVENT FLOODING: EACH EVENT IS BROADCAST FROM THE PUBLISHER IN THE WHOLE SYSTEM.
STRAIGHTFORWARD BUT VERY EXPENSIVE.



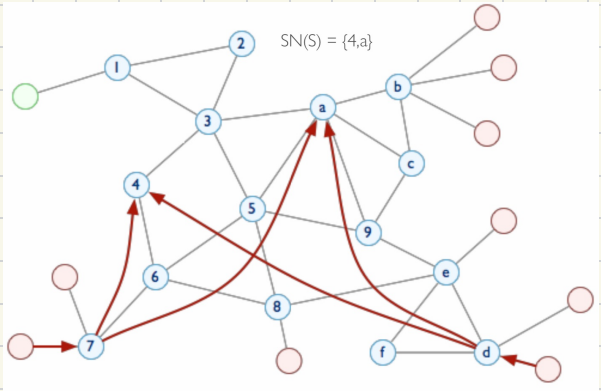
SUBSCRIPTION FLOODING: EACH SUBSCRIPTION IS COPIED ON EVERY BROKER, IN ORDER TO BUILD LOCALLY COMPLETE SUBSCRIPTION TABLES. THIS TABLES ARE THEN USED TO LOCALLY MATCH EVENTS AND DIRECTLY NOTIFY INTERESTED SUBSCRIBERS. SUFFERS FROM A LARGE MEMORY OVERHEAD.

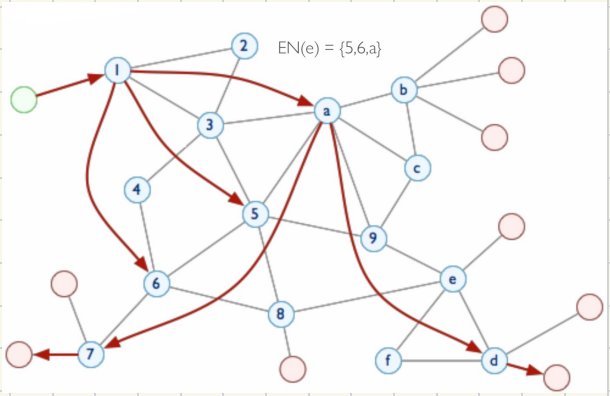


FILTER-BASED ROUTING: SUBSCRIPTIONS ARE PARTIALLY DIFFUSED TO BUILD ROUTING TABLES, CREATING MULTICAST TREES TO CONNECT PUBLISHERS TO INTERESTED SUBSCRIBERS.



RENDEZ-VOUS ROUTING: EVENTS AND SUBSCRIPTIONS ARE MAPPED TO A SUBSET OF NODES. THESE NODES VERIFY CORRESPONDENCES BETWEEN EVENTS AND SUBSCRIPTIONS.





BROADCAST WITH BYZANTINE PROCESSES

LESSON 22

A BYZANTINE PROCESS CAN ARBITRARILY DEVIATE FROM INSTRUCTIONS ASSIGNED BY THE ALGORITHM, CREATING FALSE MESSAGES, DELETING THEM OR ALTERING THEIR CONTENTS.
PROTOCOLS THAT ARE RESILIENT TO BYZANTINE PROCESSES MUST BE DESIGNED TO ENSURE THE CORRECT FUNCTIONING OF THE DISTRIBUTED SYSTEM

AUTHENTICATED PERFECT LINK

APL ARE AN IMPROVED VERSION OF PP2PL TO COUNTER BYZANTINE PROCESSES.
THEY USE CRYPTOGRAPHIC MECHANISMS TO AUTHENTICATE MESSAGES AND PREVENT MANIPULATION.

Module 2.5: Interface and properties of authenticated perfect point-to-point links

Module:

Name: AuthPerfectPointToPointLinks, **instance** *al*.

Events:

Request: ⟨ *al*, *Send* | *q*, *m* ⟩: Requests to send message m to process q.

Indication: ⟨ *al*, *Deliver* | *p*, *m* ⟩: Delivers message m sent by process p.

Properties:

AL1: *Reliable delivery:* If a correct process sends a message m to a correct process q, then q eventually delivers m.

AL2: *No duplication:* No message is delivered by a correct process more than once.

AL3: *Authenticity:* If some correct process q delivers a message m with sender p and process p is correct, then m was previously sent to q by p.

BYZANTINE CONSISTENT BROADCAST

Module 3.11: Interface and properties of Byzantine consistent broadcast

Module:

Name: ByzantineConsistentBroadcast, **instance** *bcb*, with sender s.

Events:

Request: ⟨ *bcb*, *Broadcast* | *m* ⟩: Broadcasts a message m to all processes. Executed only by process s.

Indication: ⟨ *bcb*, *Deliver* | *p*, *m* ⟩: Delivers a message m broadcast by process p.

Properties:

BCB1: *Validity:* If a correct process p broadcasts a message m, then every correct process eventually delivers m.

BCB2: *No duplication:* Every correct process delivers at most one message.

BCB3: *Integrity:* If some correct process delivers a message m with sender p and process p is correct, then m was previously broadcast by p.

BCB4: *Consistency:* If some correct process delivers a message m and another correct process delivers a message m', then $m = m'$.

Algorithm 3.16: Authenticated Echo Broadcast

Implements:
ByzantineConsistentBroadcast, **instance** *bcb*, with sender s.

Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event ⟨ *bcb*, *Init* ⟩ **do**
sentecho := FALSE;
delivered := FALSE;
echos := $[\bot]^N$;

upon event ⟨ *bcb*, *Broadcast* | *m* ⟩ **do** // only process s
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | *q*, [SEND, *m*] ⟩;

upon event ⟨ *al*, *Deliver* | *p*, [SEND, *m*] ⟩ **such that** $p = s$ **and** *sentecho* = FALSE **do**
sentecho := TRUE;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | *q*, [ECHO, *m*] ⟩;

upon event ⟨ *al*, *Deliver* | *p*, [ECHO, *m*] ⟩ **do**
if *echos*[*p*] = $\bot$ **then**
echos[*p*] := *m*;

upon exists $m \neq \bot$ such that $\#(\{p \in \Pi \mid echos[p] = m\}) > \frac{N+f}{2}$
and *delivered* = FALSE **do**
delivered := TRUE;
trigger ⟨ *bcb*, *Deliver* | *s*, *m* ⟩;

Correctness is ensured if N>3f

BYZANTINE RELIABLE BROADCAST

Module 3.12: Interface and properties of Byzantine reliable broadcast

Module:

Name: ByzantineReliableBroadcast, **instance** *brb*, with sender s.

Events:

Request: ⟨ *brb*, *Broadcast* | *m* ⟩: Broadcasts a message m to all processes. Executed only by process s.

Indication: ⟨ *brb*, *Deliver* | *p*, *m* ⟩: Delivers a message m broadcast by process p.

Properties:

BRB1–BRB4: Same as properties BCB1–BCB4 in Byzantine consistent broadcast (Module 3.11).

BRB5: *Totality:* If some message is delivered by any correct process, every correct process eventually delivers a message.

Algorithm 3.18: Authenticated Double-Echo Broadcast

Implements:
ByzantineReliableBroadcast, **instance** *brb*, with sender s.

Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event ⟨ *brb*, *Init* ⟩ **do**
sentecho := FALSE;
sentready := FALSE;
delivered := FALSE;
echos := $[\bot]^N$;
readys := $[\bot]^N$;

upon event ⟨ *brb*, *Broadcast* | *m* ⟩ **do**
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | *q*, [SEND, *m*] ⟩;

upon event ⟨ *al*, *Deliver* | *p*, [SEND, *m*] ⟩ **such that** $p = s$ **and** *sentecho* = FALSE **do**
sentecho := TRUE;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | *q*, [ECHO, *m*] ⟩;

upon event ⟨ *al*, *Deliver* | *p*, [ECHO, *m*] ⟩ **do**
if *echos*[*p*] = $\bot$ **then**
echos[*p*] := *m*;

upon exists $m \neq \bot$ such that $\#(\{p \in \Pi \mid echos[p] = m\}) > \frac{N+f}{2}$
and *sentready* = FALSE **do**
sentready := TRUE;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | *q*, [READY, *m*] ⟩;

upon event ⟨ *al*, *Deliver* | *p*, [READY, *m*] ⟩ **do**
if *readys*[*p*] = $\bot$ **then**
readys[*p*] := *m*;

upon exists $m \neq \bot$ such that $\#(\{p \in \Pi \mid readys[p] = m\}) > f$
and *sentready* = FALSE **do**
sentready := TRUE;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | *q*, [READY, *m*] ⟩;

upon exists $m \neq \bot$ such that $\#(\{p \in \Pi \mid readys[p] = m\}) > 2f$
and *delivered* = FALSE **do**
delivered := TRUE;
trigger ⟨ *brb*, *Deliver* | *s*, *m* ⟩;

REGISTERS WITH BYZANTINE PROCESSES

LESSON 23

IT MUST BE ENSURED THAT AFTER A WRITE OPERATION IS COMPLETED BY A WRITER, EACH SUBSEQUENT READ RETURNS THE MOST RECENTLY WRITTEN VALUE

THE WRITER SENDS THE TIMESTAMPED VALUE (v, wts) TO THE SERVERS AND WAITS FOR ACK MESSAGES. THE QUORUM FOR ACKS MUST BE SUFFICIENT TO ENSURE THAT THE CORRECT SERVERS HAVE REGISTERED (v, wts).

THE READER SENDS A READ REQUEST TO THE SERVERS AND WAITS FOR SUFFICIENT RESPONSES TO DETERMINE THE LATEST VALUE.

FOR $N > 3f$ A QUORUM OF $(N+f)/2$ MAY BE SUFFICIENT, BUT TO IMPLEMENT SAFE REGISTERS WE USE MASKING QUORUMS → $N > 4f$, QUORUM: $(N+2f)/2$

Module 4.5: Interface and properties of a $(1, N)$ Byzantine safe register

Module:

Name: $(1, N)$-ByzantineSafeRegister, **instance** *bonsr*, with writer w.

Events:

Request: ⟨ *bonsr*, *Read* ⟩: Invokes a read operation on the register.

Request: ⟨ *bonsr*, *Write* | v ⟩: Invokes a write operation with value v on the register. Executed only by process w.

Indication: ⟨ *bonsr*, *ReadReturn* | v ⟩: Completes a read operation on the register with return value v.

Indication: ⟨ *bonsr*, *WriteReturn* ⟩: Completes a write operation on the register. Occurs only at process w.

Properties:

BONSR1: *Termination:* If a correct process invokes an operation, then the operation eventually completes.

BONSR2: *Validity:* A read that is not concurrent with a write returns the last value written.

Algorithm 4.14: Byzantine Masking Quorum

Implements:
$(1, N)$-ByzantineSafeRegister, **instance** *bonsr*, with writer w.

Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event ⟨ *bonsr*, *Init* ⟩ **do**
$(ts, val) := (0, \perp)$;
$wts := 0$;
$acklist := [\perp]^N$;
$rid := 0$;
$readlist := [\perp]^N$;

upon event ⟨ *bonsr*, *Write* | v ⟩ **do**
$wts := wts + 1$;
$acklist := [\perp]^N$;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [WRITE, wts, v] ⟩;

upon event ⟨ *al*, *Deliver* | p, [WRITE, ts', v'] ⟩ **such that** $p = w$ **do**
if $ts' > ts$ **then**
$(ts, val) := (ts', v')$;
trigger ⟨ *al*, *Send* | p, [ACK, ts'] ⟩;

upon event ⟨ *al*, *Deliver* | q, [ACK, ts'] ⟩ **such that** $ts' = wts$ **do**
1 *acklist*[q] := ACK;
if $\#(acklist) > (N + 2f)/2$ **then**
$acklist := [\perp]^N$;
trigger ⟨ *bonsr*, *WriteReturn* ⟩;

upon event ⟨ *bonsr*, *Read* ⟩ **do**
$rid := rid + 1$;
$readlist := [\perp]^N$;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [READ, rid] ⟩;

upon event ⟨ *al*, *Deliver* | p, [READ, r] ⟩ **do**
trigger ⟨ *al*, *Send* | p, [VALUE, r, ts, val] ⟩;

upon event ⟨ *al*, *Deliver* | q, [VALUE, r, ts', v'] ⟩ **such that** $r = rid$ **do**
readlist[q] := (ts', v');
2 **if** $\#(readlist) > \frac{N+2f}{2}$ **then**
v := byzhighestval(*readlist*);
$readlist := [\perp]^N$;
trigger ⟨ *bonsr*, *ReadReturn* | v ⟩;

REGULAR REGISTERS

THE SPECIFICATION DOESN'T CHANGE, BUT IMPLEMENTATIONS DIFFER BASED ON WHETHER ENCRYPTION IS PRESENT OR NOT.

WITH CRYPTOGRAPHY: THE WRITER SIGNS THE VALUE WITH TIMESTAMP (v, wts) AND SENDS IT TO ALL SERVERS THAT STORE THE SIGNED VALUE. THE READER VERIFIES THE SIGNATURE ON EACH VALUE RECEIVED AND IGNORES THOSE WITH INVALID SIGNATURES BYZANTINE PROCESSES CANNOT ARBITRARILY ALTER THE VALUE OR TIMESTAMP.

Algorithm 4.15: Authenticated-Data Byzantine Quorum

Implements:
$(1, N)$-ByzantineRegularRegister, **instance** *bonrr*, with writer w.

Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event ⟨ *bonrr*, *Init* ⟩ **do**
$(ts, val, \sigma) := (0, \perp, \perp)$;
$wts := 0$;
$acklist := [\perp]^N$;
$rid := 0$;
$readlist := [\perp]^N$;

upon event ⟨ *bonrr*, *Write* | v ⟩ **do** // only process w
$wts := wts + 1$;
$acklist := [\perp]^N$;
$\sigma := sign(self, bonrr \| self \| \text{WRITE} \| wts \| v)$;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [WRITE, wts, v, σ] ⟩;

upon event ⟨ *al*, *Deliver* | p, [WRITE, ts', v', σ'] ⟩ **such that** $p = w$ **do**
if $ts' > ts$ **then**
$(ts, val, \sigma) := (ts', v', \sigma')$;
trigger ⟨ *al*, *Send* | p, [ACK, ts'] ⟩;

upon event ⟨ *al*, *Deliver* | q, [ACK, ts'] ⟩ **such that** $ts' = wts$ **do**
acklist[q] := ACK;
if $\#(acklist) > (N + f)/2$ **then**
$acklist := [\perp]^N$;
trigger ⟨ *bonrr*, *WriteReturn* ⟩;

i.e. > 2f

Assumption
N>3f

upon event ⟨ *bonrr*, *Read* ⟩ **do**
$rid := rid + 1$;
$readlist := [\perp]^N$;
forall $q \in \Pi$ **do**
trigger ⟨ *al*, *Send* | q, [READ, rid] ⟩;

upon event ⟨ *al*, *Deliver* | p, [READ, r] ⟩ **do**
trigger ⟨ *al*, *Send* | p, [VALUE, r, ts, val, σ] ⟩;

upon event ⟨ *al*, *Deliver* | q, [VALUE, r, ts', v', σ'] ⟩ **such that** $r = rid$ **do**
if *verifysig*($q, bonrr \| w \| \text{WRITE} \| ts' \| v', \sigma'$) **then**
readlist[q] := (ts', v');
if $\#(readlist) > \frac{N+f}{2}$ **then**
v := highestval(*readlist*);
$readlist := [\perp]^N$;
trigger ⟨ *bonrr*, *ReadReturn* | v ⟩;

i.e. > 2f

WITHOUT CRYPTOGRAPHY: WE HAVE TWO PHASES TO WRITE A VALUE:

PRE-WRITE: THE WRITER SENDS PREWRITE MESSAGES (v, wts) AND WAITS FOR RESPONSES FROM N-f PROCESSES.

WRITE: THE WRITER SENDS WRITE MESSAGES (v, wts) AND WAITS FOR RESPONSES FROM N-f PROCESSES.

THE READER COLLECTS THE VALUES AND SELECTS THE VALUE WITH THE HIGHEST TIMESTAMP AMONG THOSE PRESENT IN AT LEAST f+1 ANSWERS

Algorithm 4.16: Double-Write Byzantine Quorum (part 1, write)

Implements:
(1, N)-ByzantineRegularRegister, **instance** *bonrr*, with writer w.
Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event ⟨ *bonrr*, *Init* ⟩ **do**
  $(pts, pval) := (0, \perp)$;
  $(ts, val) := (0, \perp)$;
  $(wts, wval) := (0, \perp)$;
  preacklist := $[\perp]^N$;
  acklist := $[\perp]^N$;
  rid := 0;
  readlist := $[\perp]^N$;

upon event ⟨ *bonrr*, *Write* | v ⟩ **do** // only process w
  $(wts, wval) := (wts + 1, v)$;
  preacklist := $[\perp]^N$;
  acklist := $[\perp]^N$;
  forall $q \in \Pi$ **do**
    trigger ⟨ *al*, *Send* | q, [PREWRITE, $wts, wval$] ⟩;

upon event ⟨ *al*, *Deliver* | p, [PREWRITE, $pts', pval'$] ⟩
    such that $p = w \wedge pts' = pts + 1$ **do**
  $(pts, pval) := (pts', pval')$;
  trigger ⟨ *al*, *Send* | p, [PREACK, pts] ⟩;

upon event ⟨ *al*, *Deliver* | q, [PREACK, pts'] ⟩ **such that** $pts' = wts$ **do**
  preacklist[q] := PREACK;
  if #(*preacklist*) $\geq N - f$ **then**
    preacklist := $[\perp]^N$;
    forall $q \in \Pi$ **do**
      trigger ⟨ *al*, *Send* | q, [WRITE, $wts, wval$] ⟩;

upon event ⟨ *al*, *Deliver* | p, [WRITE, ts', val'] ⟩
    such that $p = w \wedge ts' = pts \wedge ts' > ts$ **do**
  $(ts, val) := (ts', val')$;
  trigger ⟨ *al*, *Send* | p, [ACK, ts] ⟩;

upon event ⟨ *al*, *Deliver* | q, [ACK, ts'] ⟩ **such that** $ts' = wts$ **do**
  acklist[q] := ACK;
  if #(*acklist*) $\geq N - f$ **then**
    acklist := $[\perp]^N$;
    trigger ⟨ *bonrr*, *WriteReturn* ⟩;

Algorithm 4.17: Double-Write Byzantine Quorum (part 2, read)

upon event ⟨ *bonrr*, *Read* ⟩ **do**
  rid := *rid* + 1;
  readlist := $[\perp]^N$;
  forall $q \in \Pi$ **do**
    trigger ⟨ *al*, *Send* | q, [READ, *rid*] ⟩;

upon event ⟨ *al*, *Deliver* | p, [READ, r] ⟩ **do**
  trigger ⟨ *al*, *Send* | p, [VALUE, $r, pts, pval, ts, val$] ⟩;

upon event ⟨ *al*, *Deliver* | q, [VALUE, $r, pts', pval', ts', val'$] ⟩ **such that** $r = rid$ **do**
  if $pts' = ts' + 1 \vee (pts', pval') = (ts', val')$ **then**
    readlist[q] := $(pts', pval', ts', val')$;
  if exists (ts, v) in an entry of *readlist* **such that** *authentic*$(ts, v, readlist)$ = TRUE
    and exists $Q \subseteq readlist$ **such that**
      $\#(Q) > \frac{N+f}{2} \wedge selectedmax(ts, v, Q) =$ TRUE **then**
    readlist := $[\perp]^N$;
    trigger ⟨ *bonrr*, *ReadReturn* | v ⟩;
  else
    trigger ⟨ *al*, *Send* | q, [READ, r] ⟩;

CONSENSUS WITH BYZANTINE PROCESSES

IDEALLY, WE WOULD LIKE TO OBTAIN THESE PROPERTIES:

- TERMINATION
- VALIDITY
- INTEGRITY
- AGREEMENT

HOWEVER, THE VALIDITY PROPERTY MUST BE ADAPTED AS BYZANTINE PROCESSES MAY INVENT VALUES OR CLAIM TO HAVE PROPOSED DIFFERENT VALUES. THUS WE DEFINE TWO DIFFERENT VERSIONS OF VALIDITY (WEAK AND STRONG).

Module 5.10: Interface and properties of weak Byzantine consensus

NOTE: Weak Validity allows to decide an arbitrary value if some process is Byzantine.

Properties:

WBC1: *Termination:* Every correct process eventually decides some value.

WBC2: *Weak validity:* If all processes are correct and propose the same value v, then no correct process decides a value different from v; furthermore, if all processes are correct and some process decides v, then v was proposed by some process.

WBC3: *Integrity:* No correct process decides twice.

WBC4: *Agreement:* No two correct processes decide differently.

Module 5.11: Interface and properties of (strong) Byzantine consensus

Module:

Name: ByzantineConsensus, **instance** *bc*.

Events:

Request: ⟨ *bc*, *Propose* | v ⟩: Proposes value v for consensus.

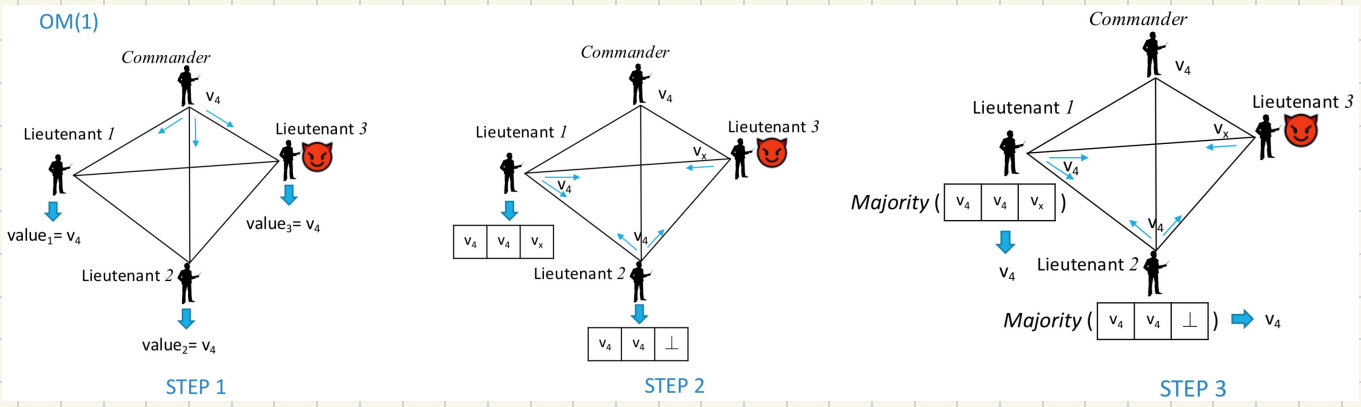
Indication: ⟨ *bc*, *Decide* | v ⟩: Outputs a decided value v of consensus.

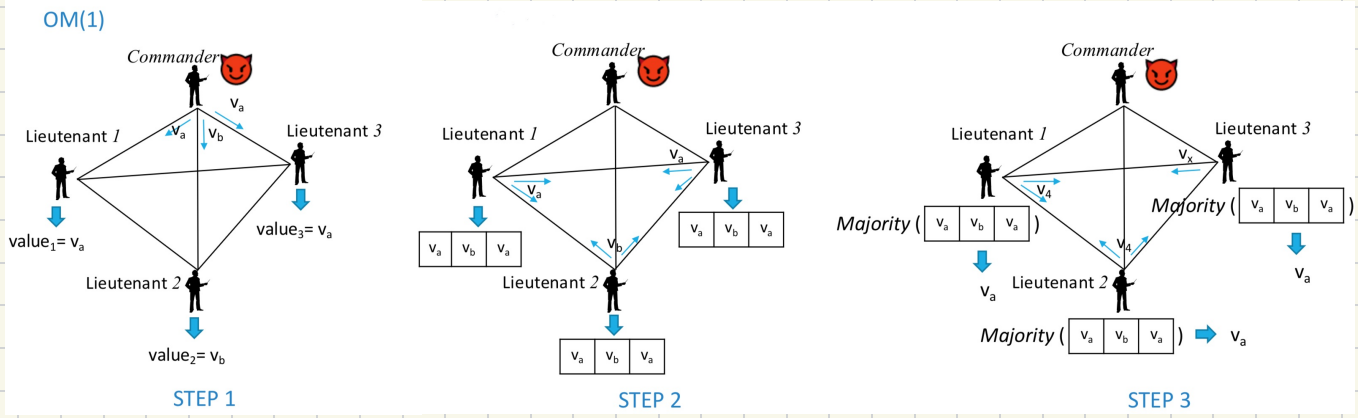
Properties:

BC1 and **BC3–BC4:** Same as properties WBC1 and WBC3–WBC4 in weak Byzantine consensus (Module 5.10).

BC2: *Strong validity:* If all correct processes propose the same value v, then no correct process decides a value different from v; otherwise, a correct process may only decide a value that was proposed by some correct process or the special value □.

OM(f) PROTOCOLS





SIGNATURE PROTOCOL

THE COMMANDER SIGNS THE MESSAGE AND SENDS IT TO THE LIEUTENANTS. EACH LIEUTENANT FIRST FORWARDS THE SIGNED MESSAGE TO THE OTHERS, ADDING HIS OWN SIGNATURE, AND THEN VALIDATES THE RECEIVED SIGNATURES TO DECIDE THE FINAL VALUE.

AVOIDS EXPONENTIAL GROWTH OF THE NUMBER OF MESSAGES IN CLASSICAL OM(f) PROTOCOLS, AND GUARANTEES AUTHENTICITY OF MESSAGES.

DLT AND BLOCKCHAIN

LESSON 25,26

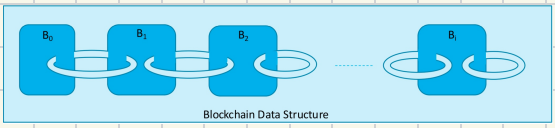
DISTRIBUTED LEDGER TECHNOLOGY

IT IS A DATABASE REPLICATED ACROSS MULTIPLE NODES OR PARTICIPANTS USED TO STORE RECORD OF TRANSACTIONS. KEY PROPERTIES ARE:

- CONSISTENCY
- INTEGRITY
- AVAILABILITY

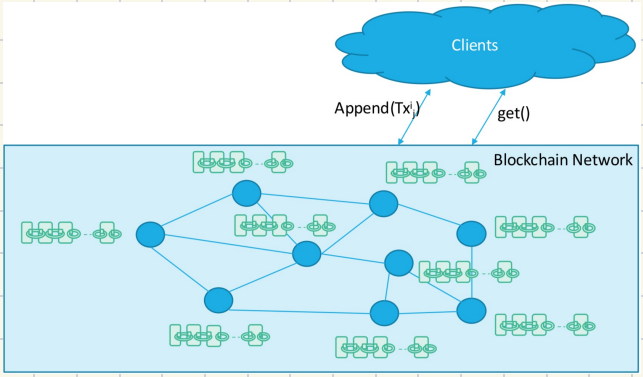
BLOCKCHAIN

IT IS A DECENTRALIZED, DISTRIBUTED DATA STRUCTURE USED TO RECORD TRANSACTIONS IN IMMUTABLY CHAINED BLOCKS.



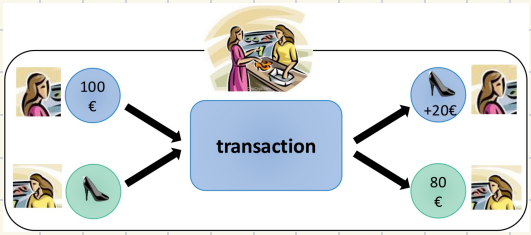
EACH BLOCK CONTAINS:

- A SET OF TRANSACTIONS.
- THE POINTER (HASH) TO THE PREVIOUS BLOCK.



THE CONCATENATION OF BLOCKS CREATES A LINEAR AND IMMUTABLE SEQUENCE.

EVERY BLOCKCHAIN REPRESENTS A DLT BUT THE VICE VERSA IS NOT TRUE.

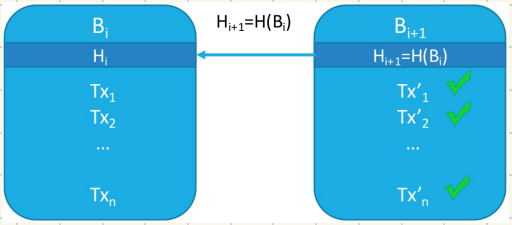


A TRANSACTION IS AN ISTANCE OF BUYING OR SELLING SOMETHING, OR AN EXCHANGE OR INTERACTION BETWEEN PEOPLE.

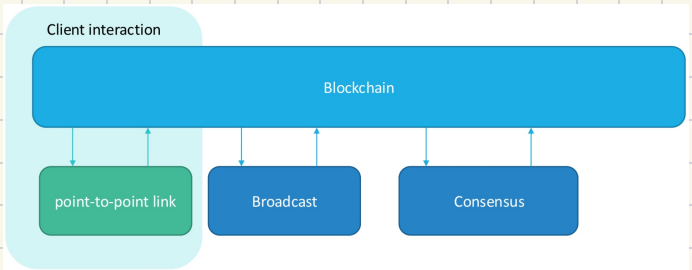
CREATING AND CONNECTING BLOCKS

PROCESSES COLLECT TRANSACTIONS FROM CLIENTS, WHICH MUST BE VALID WITH RESPECT TO THE LEDGER SPECIFICATION.

AFTER VALIDATION, THE BLOCK IS ADDED TO THE BLOCKCHAIN BY CALCULATING ITS HASH AND CONCATENATING IT WITH THE PREVIOUS BLOCK.



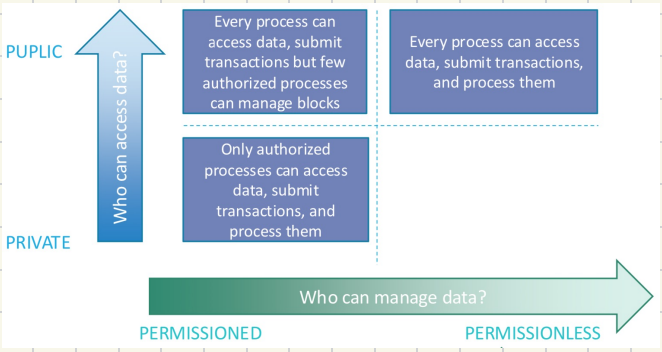
MULTIPLE NODES CAN ATTEMPT TO CREATE BLOCKS SIMULTANEOUSLY, CAUSING FORKS, SO A CONSENSUS PROTOCOL IS NEEDED.



BLOCKCHAIN CLASSIFICATION

PUBLIC: THERE ARE NO RESTRICTIONS ON READING BLOCKCHAIN DATA AND SUBMITTING TRANSACTIONS.

PRIVATE: DIRECT ACCESS TO BLOCKCHAIN DATA AND SUBMITTING TRANSACTIONS IS LIMITED.

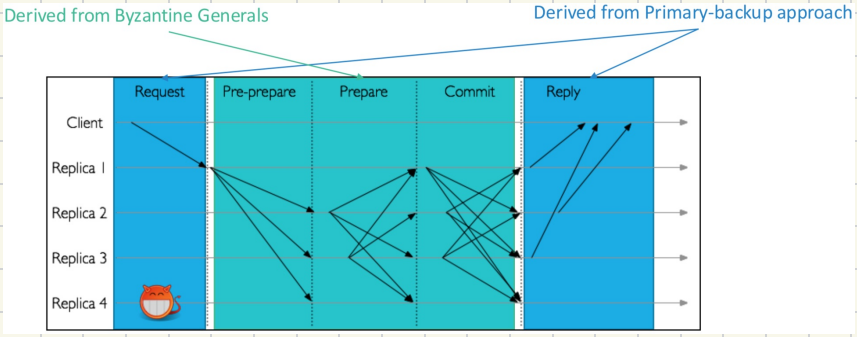


PERMISSIONED: RESTRICTIONS ON IDENTITIES OF TRANSACTION PROCESSORS.

PERMISSIONLESS: NO RESTRICTIONS ON IDENTITIES OF TRANSACTION PROCESSORS.

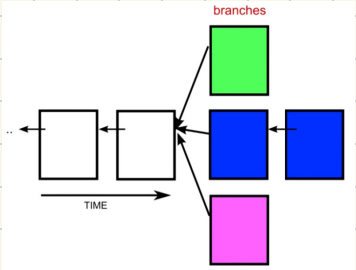
PRACTICAL BYZANTINE FAULT TOLERANCE (PBFT): THE IDEA IS THAT PROCESSES MAINTAIN A COPY OF THE BLOCKCHAIN LOCALLY AND RUN A BFT CONSENSUS PROTOCOL TO AGREE ON THE NEXT BLOCK.

THE PROTOCOL IS DIVIDED IN 5 PHASES:
- REQUEST, PRE-PREPARE, PREPARE, COMMIT AND REPLY.



A CLIENT SENDS A REQUEST TO INVOKE AN OPERATION TO THE PRIMARY. THE PRIMARY MULTICASTS THE REQUEST TO THE BACKUPS. REPLICAS EXECUTE THE REQUEST AND SEND A REPLY TO THE CLIENT. THE CLIENT WAITS FOR $f+1$ REPLIES FROM DIFFERENT REPLICAS WITH THE SAME RESULT, WHICH IS THE RESULT OF OPERATION.

PROOF OF WORK (POW): NODES COMPETE TO SOLVE A COMPUTATIONAL PROBLEM, AND THE WINNER ADDS A NEW BLOCK TO THE CHAIN.



OCCASIONALLY TWO OR MORE BLOCKS MAY ARRIVE TOGETHER. IN CASE OF BRANCHES THE NETWORK HAS TO CONVERGE TO THE LONGEST BRANCH.

POW REQUIRES A HUGE AMOUNT OF COMPUTATIONAL RESOURCES.

PROOF OF STAKE (POS): NODES WITH A LARGER AMOUNT OF STAKES (e.g. COINS) ARE MORE LIKELY TO ADD BLOCKS IT REDUCES ENERGY USE COMPARED TO PoW.

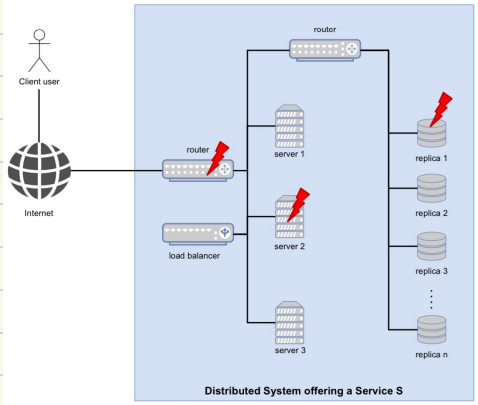
SERVICE AVAILABILITY

LESSON 28

AVAILABILITY IS THE FRACTION OF TIME THE SYSTEM IS OPERATIONAL.

A DISTRIBUTED SYSTEM OFFERING A SERVICE S IS COMPOSED OF:

- CLIENT: USERS WHO REQUEST THE SERVICE.
- ROUTER/LOAD BALANCER: ELEMENTS THAT DISTRIBUTE THE WORKLOAD.
- SERVER: REPLICAS OF THE SERVICE.



THE OVERALL AVAILABILITY OF THE SYSTEM DEPENDS ON THE AVAILABILITY OF EACH COMPONENT.

COMPONENT LEVEL AVAILABILITY

A COMPONENT CAN BE IN TWO STATES:
- UP: WORKING
- DOWN: NOT WORKING

KEY METRICS:

MEAN TIME TO FAILURE (MTTF): MEAN TIME BEFORE FAILURE.

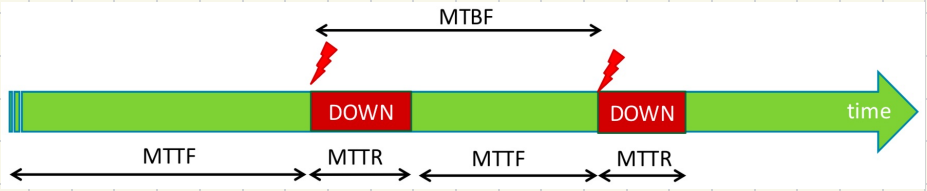
$$\lambda = \frac{1}{MTTF}$$

MEAN TIME TO REPAIR (MTTR): AVERAGE TIME NEEDED TO REPAIR A FAULT.

$$\mu = \frac{1}{MTTR}$$

MEAN TIME BETWEEN FAILURES (MTBF): AVERAGE TIME BETWEEN TWO CONSECUTIVE FAILURES.

$$MTBF = MTTF + MTTR$$



THE AVAILABILITY A IS THE PROBABILITY OF BEING IN STATE UP:

$$A = P_{UP} = \frac{\mu}{\mu + \lambda} = \frac{1/MTTR}{1/MTTR + 1/MTTF} = \frac{MTTF}{MTTF + MTTR} = \frac{MTTF}{MTBF}$$

TO INCREASE A WE NEED TO:

- REDUCE THE FREQUENCY OF FAILURES (INCREASE MTTF).
- REDUCE REPAIR TIME (MTTR).

ES.

Let us assume that the machine M reported during test a MTBF of 5 minutes and maintenance takes 30 seconds to correctly reboot the machine M and restarts A.

MTBF: 5 min , MTTR: 30 s = 0.5 min

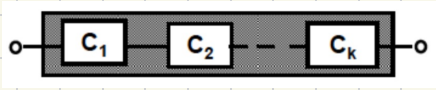
MTBF: MTTF + MTTR → 5: 0.5 + MTTF → MTTF: 4.5 min

$$A = \frac{MTTF}{MTBF} = \frac{4.5}{5} = 0.9 = 90\%$$

AVAILABILITY OF COMPLEX SYSTEMS

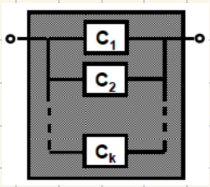
THEY CAN BE COMPOSED OF COMPONENTS CONNECTED IN THREE WAYS:

SERIAL: ALL COMPONENTS MUST BE AVAILABLE FOR THE SYSTEM TO WORK



$$A_{SERIAL} = \Pi A_i$$

PARALLEL: THE SYSTEM WORKS IF AT LEAST ONE OF THE COMPONENTS IS AVAILABLE.



$$A_{PARALLEL} = 1 - \Pi(1 - A_i)$$

HYBRID M OUT OF N: THE SYSTEM WORKS IF AT LEAST M COMPONENTS OUT OF N ARE AVAILABLE

$$A = \sum_{i=0}^{K} \binom{N}{i} A_c^{N-i} (1 - A_c)^i$$

PERFORMANCE MODEL

LESSON 29

CAPACITY PLANNING

IT CONSISTS OF PREDICTING AND ESTIMATING THE RESOURCES NEEDED (HARDWARE, SW, STORAGE...) TO MAINTAIN SERVICE LEVELS OVER A GIVEN PERIOD. IT IS NECESSARY TO DETERMINE THE MOST CONVENIENT WAY TO AVOID SATURATION OF THE SYSTEM.

UNDERSTANDING THE ENVIRONMENT: THE GOAL IS TO LEARN WHAT KIND OF HARDWARE, SW. NETWORK, SLA ARE PRESENT IN THE ENVIRONMENT.

WORKLOAD MODEL: IT'S THE SET OF ALL INPUTS THAT THE SYSTEM RECEIVES FROM ITS ENVIRONMENT IN A GIVEN PERIOD OF TIME.

PERFORMANCE MODEL: PREDICTS PERFORMANCE METRICS BASED ON SYSTEM DESCRIPTION AND WORKLOAD PARAMETERS.

COST MODEL: WE TALK ABOUT HARDWARE, SW, TELECOMUNICATIONS COST ETC...

COST/PERFORMANCE ANALYSIS: PERFORMANCE METRICS, COSTS AND ROI ARE ESTIMATED FOR ALL POSSIBLE SCENARIOS.

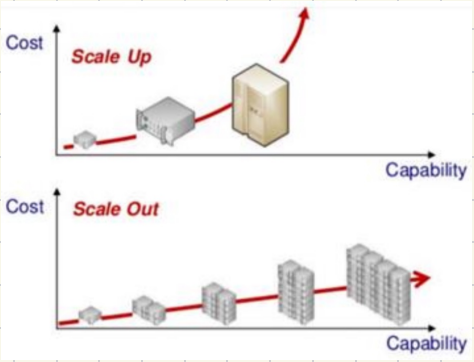
ELASTICITY AND SCALABILITY

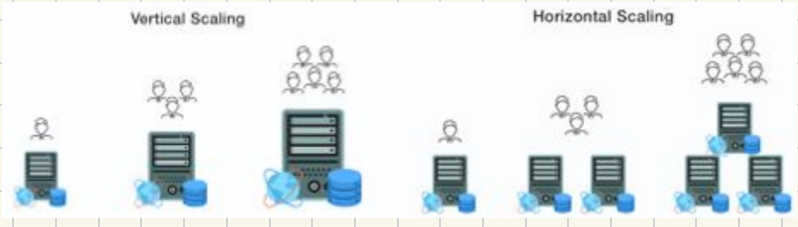
ELASTICITY IS THE ABILITY OF THE SYSTEM TO DYNAMICALLY ADAPT TO LOAD CHANGES BY ADDING OR REMOVING RESOURCES.

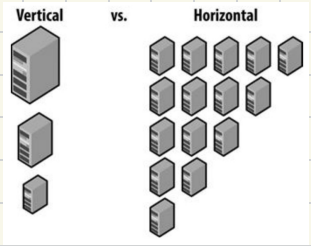
SCALABILITY IS THE PROPERTY OF A SYSTEM TO HANDLE A GROWING AMOUNT OF WORK.

VERTICAL (SCALE UP): MORE POWERFUL RESOURCES TO A SINGLE NODE. LIMITED BY PHYSICAL CONSTRAINTS.

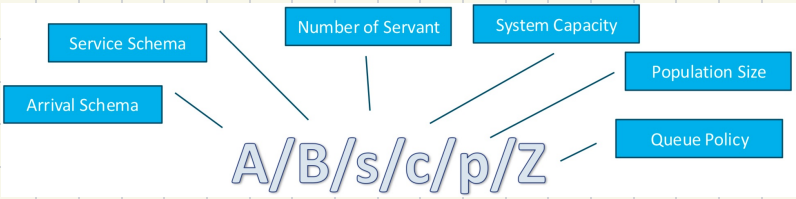
HORIZONTAL (SCALE OUT): MORE RESOURCES WITH SAME CAPACITY. OFFERS GREATER FAULT TOLERANCE AND RESILIENCE.







KENDALL NOTATION FOR QUEUING SYSTEMS



ARRIVAL SCHEMA: IT REPRESENT THE WAY IN WHICH USER RAISES REQUEST TO THE SYSTEM.

SERVICE SCHEMA: IT REPRESENT THE WAY IN WHICH SERVANTS PROVIDE THE SERVICE.

 # OF SERVANT: COMPLEX SYSTEMS MAY IMPLEMENT SERVICES BY ORCHESTRATING DIFFERENT COMPONENTS, MANAGED WITH MULTIPLE SERVANTS.

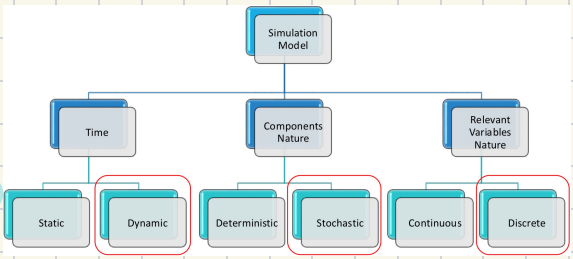
SYSTEM CAPACITY: DEFINE THE MAXIMUM SIZE THAT THE SYSTEM MAY CONSIDER.

POPULATION: ALL POTENTIAL USERS OF THE SYSTEM.

INTRO SIMULATION

LESSON 30

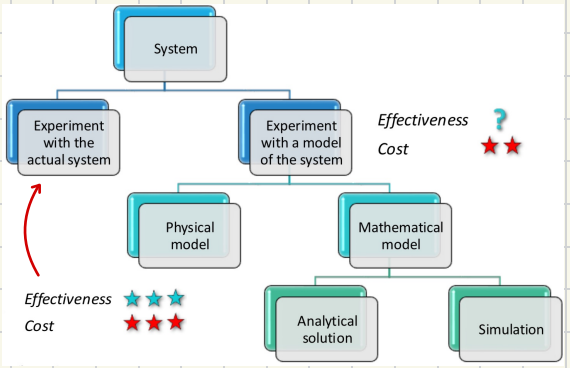
ANALYTICAL MODELS ARE NOT ALWAYS SUITABLE FOR ANALYZING REAL SYSTEMS DUE TO COMPLEXITY AND REPRESENTATIVENESS OF THE MODELS OR SIZE OF THE SYSTEM. SIMULTATION THEREFORE REPRESENTS A VALID ALTERNATIVE FOR STUDYING COMPLEX SYSTEMS IN A MORE REALISTIC WAY.



A SIMULATION MODEL CAN BE CLASSIFIED BASED ON TIME, COMPONENTS AND VARIABLES.

SYSTEMS

A SYSTEM IS A SET OF ENTITIES THAT INTERACT TO ACHIEVE A LOGICAL PURPOSE. ITS STATE IS DEFINED AS THE SET OF VARIABLES NECESSARY TO DESCRIBE IT AT A GIVEN MOMENT.



DISCRETE SYSTEM: STATE VARIABLES CHANGE INSTANTANEOUSLY AT SEPARATE POINTS IN TIME.

CONTINUOUS SYSTEM: STATE VARIABLES CHANGE CONTINUOUSLY OVER TIME.

HYBRID SYSTEM. SOME VARIABLES ARE DISCRETE, OTHERS CONTINUOUS.

DISCRETE EVENT SIMULATION MODEL

MODEL THE SYSTEM EVOLVING OVER TIME, WHERE STATES CHANGE AT DISCRETE EVENTS.

System state	The collection of state variables necessary to describe the system at a particular time
Simulation clock	A variable giving the current value of simulated time
Event list	A list containing the next time when each type of event will occur
Statistical counters	Variables used for storing statistical information about system performance
Initialization routine	A subprogram to initialize the simulation model at time 0
Timing routine	A subprogram that determines the next event from the event list and then advances the simulation clock to the time when that event is to occur

Event routine	A subprogram that updates the system state when a particular type of event occurs (there is one event routine for each event type)
Library routines	A set of subprograms used to generate random observations from probability distributions that were determined as part of the simulation model
Report generator	A subprogram that computes estimates (from the statistical counters) of the desired measures of performance and produces a report when the simulation ends
Main program	A subprogram that invokes the timing routine to determine the next event and then transfers control to the corresponding event routine to update the system state appropriately. The main program may also check for termination and invoke the report generator when the simulation is over.

MAIN STEPS IN A GOOD SIMULATION STUDY