

Towards a dependable consensus: Basic abstractions

Highly dependable systems

Lecture 2

Lecturers: Miguel Matos and Paolo Romano

Agenda

- Motivation: Dependable distributed systems
- Basic abstractions
 - processes
 - channels
 - timing assumptions
 - alternative timing abstractions:
 - failure detectors & leader election
 - Distributed system models

Dependable distributed systems

- Our society is critically dependent on a number of **inherently distributed** services, e.g.:
 - Traffic control
 - Finances:
 - e-transactions, e-banking, stock-exchange
 - Reservation systems
 - Pretty much everything on the cloud
- L. Lamport: “a distributed system is one that stops your application because a machine you have never heard from crashed” ~70’s

The optimistic view

- Concurrency => speed (load balancing)
- Partial failures => high availability

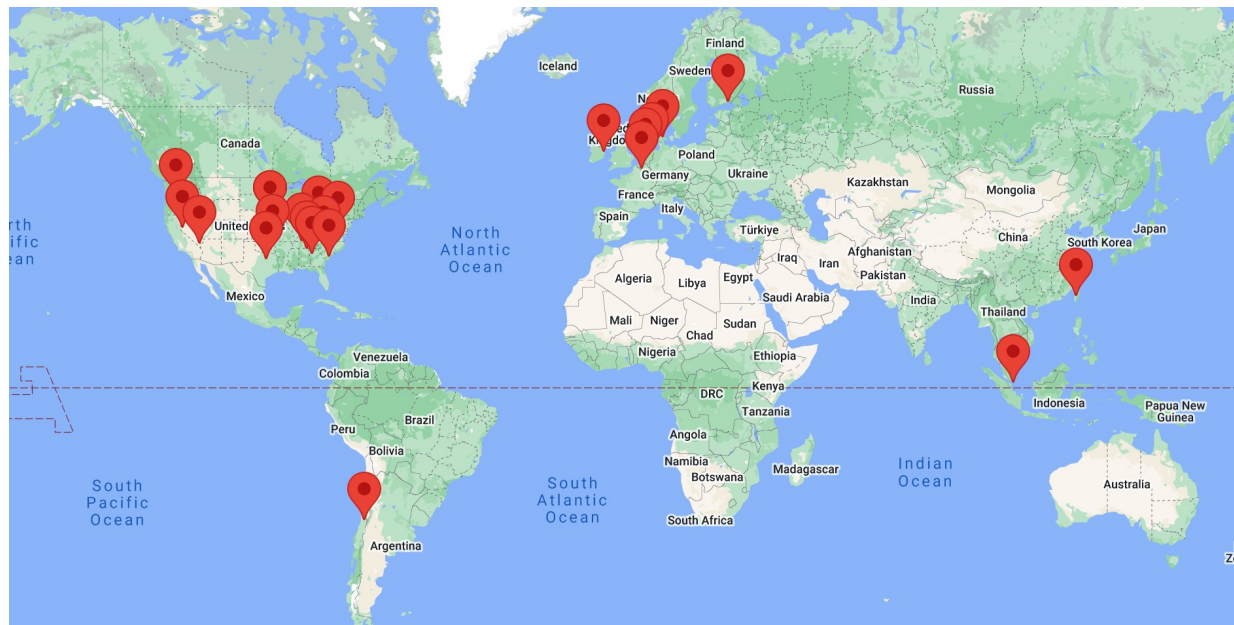
The pessimistic view

- Concurrency \Rightarrow different interleavings \Rightarrow incorrectness
- Partial failures \Rightarrow loss of state \Rightarrow incorrectness

Difficult to get right!

Distributed systems (Today: Google)

- Tens to hundreds of thousands of machines connected in each data center, 23 data centers in total
- A typical Google job can involve thousands of machines
 - GPT-3 took 405 GPU-years to train (on V100 GPUs)

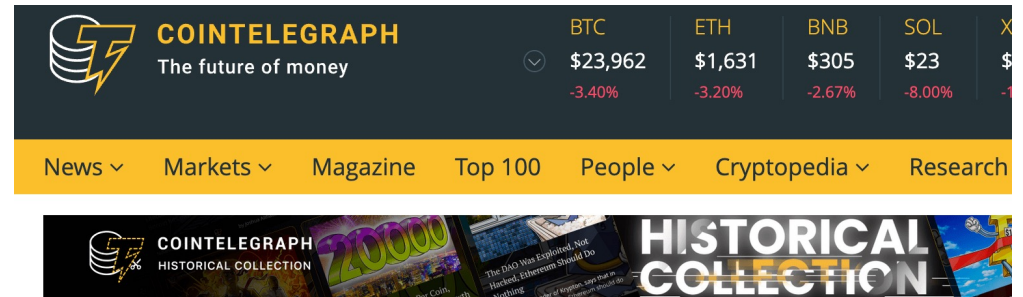


Failure is the norm, not the exception

- Tens of machines go down per day per data center
 - Due to power supply, hard drives, memory, etc.
- Need efficient algorithms to ensure **dependability** of distributed systems...
- ...and abstractions to **mask complexity** from programmers!

Blockchains introduce additional challenges

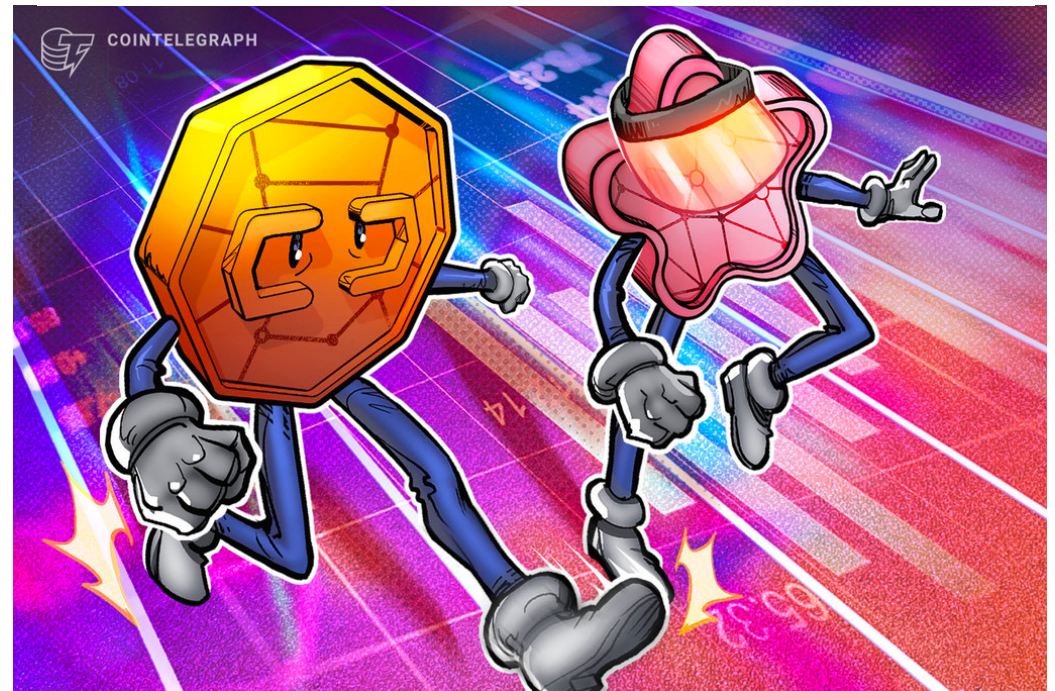
- Even in a fixed membership setting, machines fail or become unreachable
- Furthermore, need to reach consensus without having to trust a single entity or any individual participant
 - Adversarial setting - subset of machines may try to sway outcome (e.g., front running attacks try to reorder transactions)



What is front-running in crypto and NFT trading?

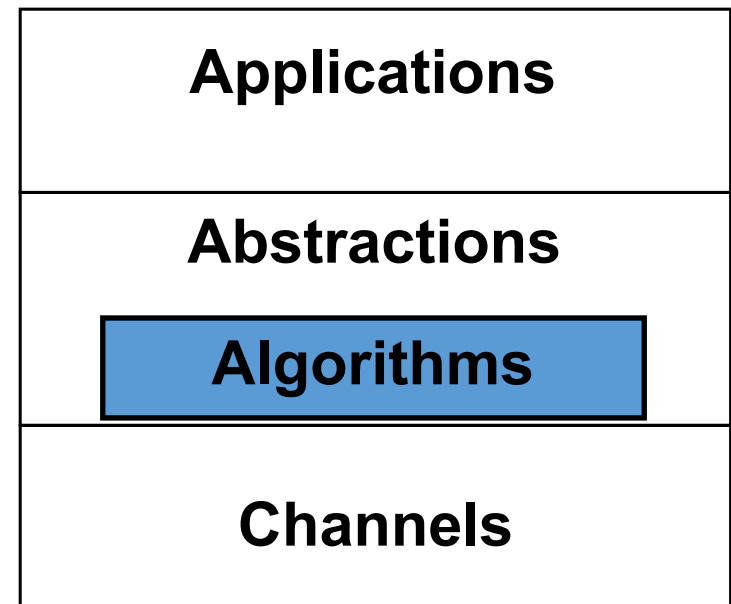
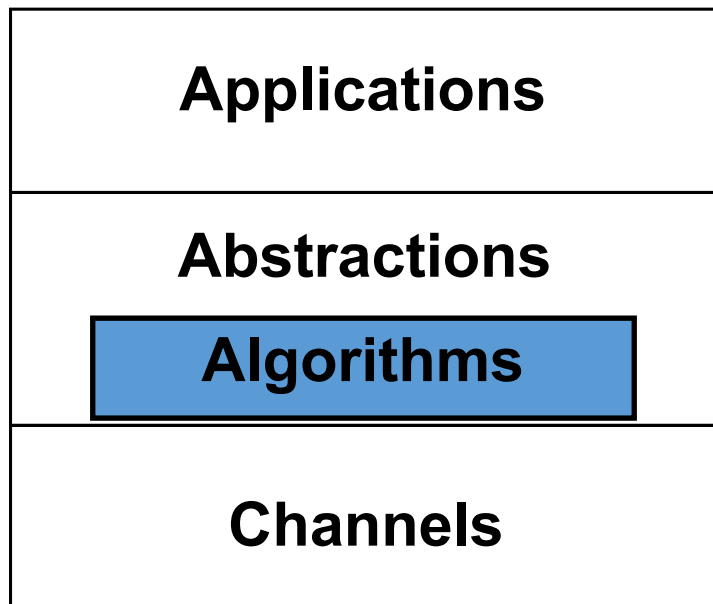
Onkar Singh

MAR 26, 2022



The power of abstraction

- Complexity is masked via **abstractions** for building dependable services
- The network is too low level, even with additional help from protocols:
- Reliability guarantees (e.g., TCP) are only offered for communication among pairs of processes, i.e., *one-to-one* communication (*client-server*)



Abstractions for dependable computing

Reliable broadcast
Causal order broadcast
Shared memory
Consensus
Total order broadcast
Atomic commit
Leader election
Terminating reliable broadcast
.....

- Several of you studied these abstractions (SD,DAD) in the benign (crash-stop) model
- In this course we will study (some of) them in the **Byzantine** fault model

Basic abstractions

- (1): *processes*
(abstracting computers)
- (2): *channels*
(abstracting networks)
- (3): *failure detectors/leader election*
(abstracting time)
- ... + *cryptographic primitives*
(hashes, MACs, digital signatures)

Processes

- The distributed system is made of a finite set of N processes: each process models a sequential program
- Processes are denoted by p_1, \dots, p_N or p, q, r
- Processes have unique identities and know each other
- Every pair of processes is connected by a link through which the processes exchange messages

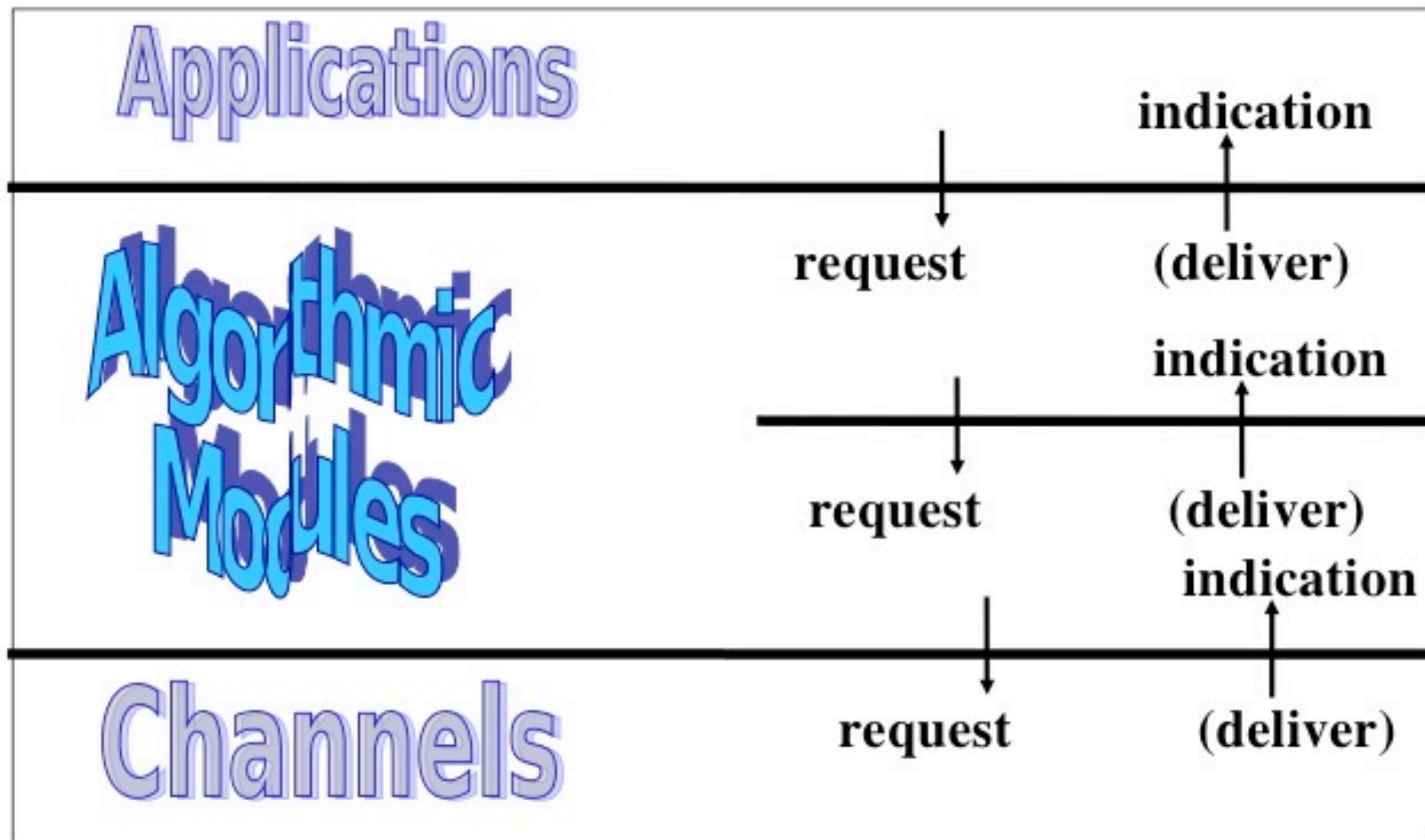
Processes

- A process executes a step at every tick of its local clock
- A step consists of:
 - Receiving a message
 - A local computation (local event), and
 - Sending a message
- One message is delivered from/sent to a process per step

Processes

- The program of a process is made of a finite set of modules (or components) organized as a stack
- Modules within the same process interact by exchanging events
- **upon** event <Event1, att1, att2,..> do
 // . . .
 trigger <Event2, att1, att2,..>

Layering of process modules



Defining a distributed service

- *Specification*: What is the service supposed to do?
i.e., correctness conditions over the set of outputs
(expressed in terms of liveness + safety)
- *Model (assumptions)*: What is the system model,
i.e., the power of the adversary and the capabilities
of the network?
- *Algorithms*: How do we implement the service?
Why does the algorithm work (proof)? What cost?

Liveness and safety

- *Safety* is a property which states that nothing bad should happen
- *Liveness* is a property which states that something good should happen (eventually)

Liveness and safety

- Example: Tell the truth
 - Having to say something is liveness
 - Not lying is safety

Recognizing safety and liveness

- Given a trace (sequence of outputs) of a distributed system
 - A safety property obeys:
 - If a finite trace does not obey the property, no extension of that trace obeys that property
 - A liveness property obeys:
 - A finite trace that does not obey the property can be extended so that the liveness property is upheld
- Any specification can be expressed in terms of liveness and safety properties

Specifications

- **Example 1: *reliable broadcast***

- Ensure that a message sent to a group of processes is received by all nonfaulty processes or none

- **Example 2: *consensus***

- Ensure that all nonfaulty processes reach a common decision among of set of proposed input values

Fault assumptions (model)

- A process either executes the algorithm assigned to it (steps) or fails
- A process is **correct** if it does not fail throughout its execution
- Two kinds of failures are mainly considered:
 - *Omissions*: the process omits to send messages it is supposed to send (distracted)
 - *Arbitrary*: the process sends messages it is not supposed to send (malicious or Byzantine)
- Other models exist in between

Crash-stop fault model

- Crash-stop: a more specific case of omissions
 - A process that omits a message to a process, omits all subsequent messages to all processes (permanent distraction): it crashes
- Also called “crash fault tolerance” (CFT)

Byzantine (arbitrary) faults

- A Byzantine faulty process may deviate in any conceivable way from the algorithm assigned to it:
 - due to both benign/unintentional faults, e.g., bugs
 - ...as well as malicious faults: attacks, collusions
 - ...and also omissions (i.e., encompasses the crash and similar models)
- To simplify reasoning:
 - unique adversary that coordinates actions of all faulty processes
- More expensive to tolerate than crashes:
 - Requires more replicas
 - cryptographic services
- Assume a maximum threshold (f out of N) of Byzantine processes

Abstracting Communication

- Processes communicate by message passing through communication channels
- Messages are uniquely identified (e.g., through <sender id, sequence number>)

Fair loss links (FLLs)

- *FLL1. Fair-loss*: If a message is sent infinitely often from correct process p_i to correct process p_j , then m is delivered infinitely often by p_j
- *FLL2. Finite duplication*: If a message m is sent a finite number of times from a correct process p_i to p_j , m is delivered a finite number of times by p_j
- *FLL3. No creation*: No message is delivered unless it was sent

Fair loss links abstraction

- This abstraction encodes a baseline guarantee from the network:
 - assume denial-of-service attacks do not last forever
 - with permanent network faults or DoS attacks of any possible duration it would be impossible to guarantee fair loss links

Stubborn links (SLs)

- *SL1. Stubborn delivery:* if a correct process p_i sends a message m to a correct process p_j , then p_j delivers m an infinite number of times
- *SL2. No creation:* No message is delivered unless it was sent

Implementing Stubborn Links using FLL (in the crash fault model)

Implements: StubbornLinks (sp2p).

Uses: FairLossLinks (flp2p).

```
upon event <sp2pSend, dest, m> do  
    while (true) do  
        trigger <flp2pSend, dest, m>;
```

```
upon event <flp2pDeliver, src, m> do  
    trigger <sp2pDeliver, src, m>;
```

Perfect (or Reliable) links

- *PL1. Validity:* If p_i and p_j are correct, then every message sent by p_i to p_j is eventually delivered by p_j
- *PL2. No duplication:* No message is delivered (to a process) more than once
- *PL3. No creation:* No message is delivered unless it was sent

Implementing Perfect Links using SLs (in the crash fault model)

Implements: PerfectLinks (pp2p).

Uses: StubbornLinks (sp2p).

upon event <init> **do**

 delivered = { };

upon event <pp2pSend, dest, m> **do**

trigger <sp2pSend, dest, m>;

upon event <sp2pDeliver, src, m> **do**

 if $m \notin \text{delivered}$ then

trigger <pp2pDeliver, src, m>;

 delivered = delivered \cup { m };

Authenticated perfect links

- Now we move to the Byzantine model – main challenge:
 - Byzantine processes may forge messages:
 - pretend that they were sent from any other process
 - no creation property of perfect/stubborn links can be endangered
 - Need to adapt specification and add machinery to the implementation
- Solution:
 - Authenticated Perfect Links

Authenticated perfect links

- *APL1. Reliable delivery:* If p_i and p_j are correct, then every message sent by p_i to p_j is eventually delivered by p_j
- *APL2. No duplication:* No message is delivered (to a correct process) more than once
- *APL3. Authenticity:* if correct process p_j delivers message m from correct process p_i , then m was previously sent from p_i to p_j

Authenticated perfect links

- Rely on Message Authentication Codes (MACs) to eliminate forgery of messages
- Primitives for MACs:
 - $\text{MAC } a \leftarrow \text{authenticate}(\text{send_proc } p, \text{rec_proc } q, \text{message } m)$
 - $\text{bool} \leftarrow \text{verifyauth}(\text{send_proc } p, \text{rec_proc } q, \text{message } m, \text{MAC } a)$
- Properties:
 - $\text{verifyauth}(p, q, m, a)$ returns true **if and only if** p had previously invoked $\text{authenticate}(p, q, m)$ and obtained a
 - only p can invoke $\text{authenticate}(p, \dots)$
 - only q can invoke $\text{verifyauth}(\dots, q, \dots)$

Authenticated perfect links using SLs

Implements: AuthenticatedPerfectLinks (alp2p).

Uses: StubbornLinks (sp2p).

upon event <init> **do**

delivered = { };

upon event <alp2pSend, dest, m> **do**

a = authenticate(*self*, dest, m);

trigger <sp2pSend, dest, [m,a]>;

upon event <sp2pDeliver, src, m, a> **do**

if verifyauth(src, self, m, a) && m \notin delivered then

trigger <alp2pDeliver, src, m>;

delivered = delivered \cup { m };

Reliable/authenticated links

- We shall assume reliable/perfect links when dealing with crash-stop failure models, and
- Authenticated reliable/perfect links when considering byzantine processes
- These abstractions are often found in standard transport level protocols (e.g., TCP, TLS/SSL)
 - allows to simplify reasoning on complex services
 - for efficiency, it may be useful to assume weaker channels (e.g., fair-loss) and avoid redundant use of similar information at different layers
 - e.g., sequence numbers may be needed also by higher level layers

Timing assumptions

- *Synchronous*:
 - *Processing*: the time it takes for a process to execute a step is bounded and known
 - *Delays*: there is a known upper bound limit on the time it takes for a message to be received
 - *Clocks*: the drift between a local clock and the global real time clock is bounded and known
- *Eventually Synchronous*: the timing assumptions hold eventually
- *Asynchronous*: no timing assumptions

Abstracting time

- Many distributed algorithms rely on timing assumptions only to detect faulty processes:
 - e.g., timeouts are used to suspect crashed processes
- Alternative approaches to formulate timing assumptions:
 - assume an asynchronous model for what regards:
 - process speed, communication latency, clock skew
 - augment the system with oracles that encapsulate the timing assumptions
- **Failure detectors:**
 - oracles that provide information on which processes are faulty
- **Leader election:**
 - identify one process (leader) that is not faulty

Failure detection

- A failure detector module is defined by events and properties
- *Events*
 - Indication: $\langle \text{crash}, p \rangle$
- Properties
 - Completeness: are faulty processes detected correctly?
 - Accuracy: can correct processes be suspected?

Failure detection

- *Perfect:*
 - *Strong Completeness:* Eventually, every process that crashes is permanently suspected by every correct process
 - *Strong Accuracy:* No process is suspected before it crashes
- *Eventually Perfect:*
 - *Strong Completeness*
 - *Eventual Strong Accuracy:* Eventually, no correct process is ever suspected

Eventually perfect failure detector

Implementation:

1. Processes periodically send heartbeat messages
 2. A process sets a timeout based on worst case round trip of a message exchange
 3. A process suspects another process if it times out on that process
 4. A process that delivers a message from a suspected process revises its suspicion and doubles its timeout
- Can be implemented in an eventually synchronous system
 - Fully encapsulates synchrony/timing assumptions
 - Algorithms (like reliable broadcast and consensus) can be designed assuming asynchronous channels/processes

Failure detection for crash and arbitrary faults

- Failure detection for crash failures can be effectively implemented using timeouts:
- Detecting arbitrary failures is a more complex problem:
 - malicious processes may selectively behave according to the protocol
 - yet, badly violate its algorithm's specification
 - detection of arbitrary failures is a difficult research problem
- Failure detectors are commonly employed only for detecting crash failures

Leader Election

- Identifies a correct process
- *Events*
 - Indication: $\langle \text{Leader}, p \rangle$
- Properties
 - **Eventual detection:** Either there is no correct process, or some correct process is eventually elected as the leader
 - **Accuracy:** If a process is leader, then all previously elected leaders have crashed
 - ensures stability of the leader: only crash of current leader triggers change
 - indirectly precludes two processes to be leader at the same time

Leader Election using a Perfect f.d.

Implements: LeaderElection

Uses: PerfectFailureDetector P

upon event <init> **do**

 suspected := empty set;

 curr_leader = null;

upon event <P, Crash, q> **do**

 add q to suspected

upon curr_leader != maxrank({processes not in suspected}) **do**

 curr_leader = maxrank({processes not in suspected})

trigger <Leader, curr_leader>;

Eventual leader election

- Leader election is impossible to implement using an eventually perfect failure detector. Why?
 - if current leader is falsely suspected, a new leader **must** be elected to guarantee eventual detection
 -violating accuracy.
- Eventual leader election:
 - Eventual accuracy: there is a time after which every correct process trusts some correct process
 - Eventual agreement: there is a time after which no two correct processes trust different correct processes
- Useful abstraction in consensus algorithms (e.g., Paxos)

Eventual Leader Election using an Eventually Perfect f.d.

Implements: LeaderElection

Uses: EventualPerfectFailureDetector $\diamond P$

upon event <init> **do**

 suspected:= empty set;

 curr_leader = null;

upon event < $\diamond P$, Suspect, q> **do**

 add q to suspected

upon event < $\diamond P$, Restore, q> **do**

 remove q from suspected

upon curr_leader != maxrank({processes not in suspected}) **do**

 curr_leader := maxrank({processes not in suspected})

trigger <Leader, curr_leader>;

Byzantine Leader Election

- Timeliness of heartbeats messages is insufficient to detect arbitrary faults
- “Trust but verify” approach:
 - allow other processes to monitor actions of current leader
 - should the leader not achieve the desired goal after some time, it should be replaced by a new leader
 - definitions of “goal achievement” and “some time” are application dependent
 - applications can trigger a $\langle \text{Complain}, p \rangle$ event
 - in eventually synchronous systems, correct processes should successively increase the time between issuing complaints:
 - eventually give correct leaders enough time to achieve its goal

Byzantine Leader Election - specification

- Properties
 - **Eventual succession:** if more than f correct processes that trust some process p complain about p , then every correct process eventually trusts a different process than p
 - **Putsch resistance:** A correct process does not trust a new leader unless at least one correct process has complained against the previous leader
 - **Eventual agreement:** there is a time after which no two correct processes trust different processes
- Eventually every correct process trusts some process that appears to perform its task in the higher-level algorithm.
 - cannot require that every correct process eventually trusts a **correct** process... Why?

Rotating Byzantine Leader Election

- Uses authenticated perfect links
- Assumes number of processes, N , larger than 3 times the number, f , of (byzantine) faulty processes ($N > 3f$)
- Algorithm advances in rounds, r
- Leader at round r is process having rank: $r \bmod N$
 - Locally generated Complaints for current leader are broadcast to all processes
 - When a process receives more than f Complaint messages it moves to the next round
 - At least one Complaint comes from a correct process

Rotating Byzantine Leader Election

- **Previous algorithm is incorrect. Why?**
- Problem:
 - only one complaint may come from a correct process, say p
 - the other f complaints may come from byzantine processes, which have sent the complaint only to p and not to the other processes
- In this case p changes leader, but the others do not!
 - Eventual agreement can be compromised

Rotating Byzantine Leader Election

- Leader at round r is process having rank: $r \bmod N$
 - Locally generated Complaints for current leader are broadcast
 - New round is triggered when a process receives more than $2f$ Complaint messages
 - prevents f byzantine processes to collude for a putsch
 - When a process receives more than f Complaint messages (and has not sent its Complaint), it also broadcasts
 - ensures that more than $2f$ Complaints are broadcast, when more than f Complaints are locally generated

Rotating Byzantine Leader Election

- Eventual succession.
 - if $>f$ processes complain, all correct processes chime in
 - every correct process receives $N-f > 2f$ Complaint messages

Rotating Byzantine Leader Election

- Putsch resistance.
 - even colluding the f byzantine processes cannot overthrow the leader:
 - need at least $f+1$ Complaint messages
 - one Complaint message must come from correct process

Rotating Byzantine Leader Election

- Eventual agreement.
 - all correct processes must eventually stop complaining about a correct leader
 - assumption → they increase (say double) the time between any two complaints
 - when all Complaint messages have been received, every correct process trusts the same process

Distributed system model

- Combination of:
 - process abstraction
 - link abstraction
 - failure-detector/leader-election abstraction

Distributed system models

- Some relevant combinations:
 - **Fail-stop:**
 - process abstraction: crash failure model
 - link abstraction: perfect
 - failure-detector/leader-election: perfect failure detector
 - **Fail-noisy:**
 - same as above but equipped with eventually perfect failure detection
 - **Fail-silent:**
 - as above but no failure-detection/leader-election oracle
 - **Fail-silent arbitrary:**
 - process abstraction: arbitrary failure model
 - link abstraction: authenticated perfect
 - failure-detector/leader-election: no
 - **Fail-noisy arbitrary**
 - as above but equipped with a byzantine eventual leader-detector

Acknowledgements

- Rachid Guerraoui, EPFL