NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Distributed Systems

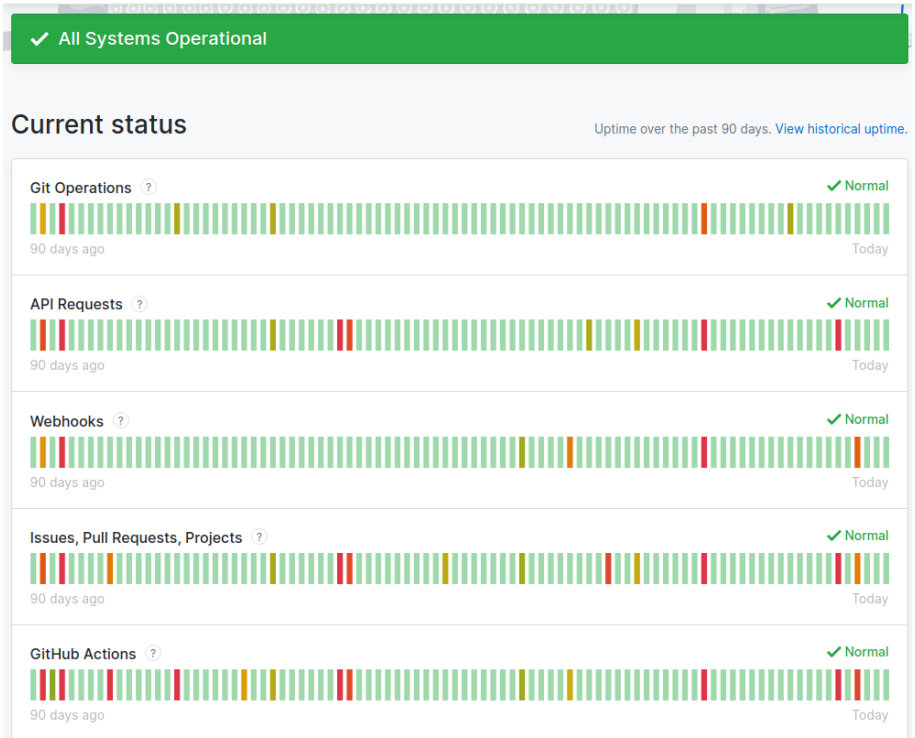
Principles of Consensus Protocols

Prof WEN Yonggang
Professor, SCSE, NTU

Fault Tolerance

Motivations, Concepts and Principles

Faults and Fault Tolerant Systems



Availability

GitHub Status

The probability that the system is operating correctly at any given moment and is available to perform its functions on behalf of its users.

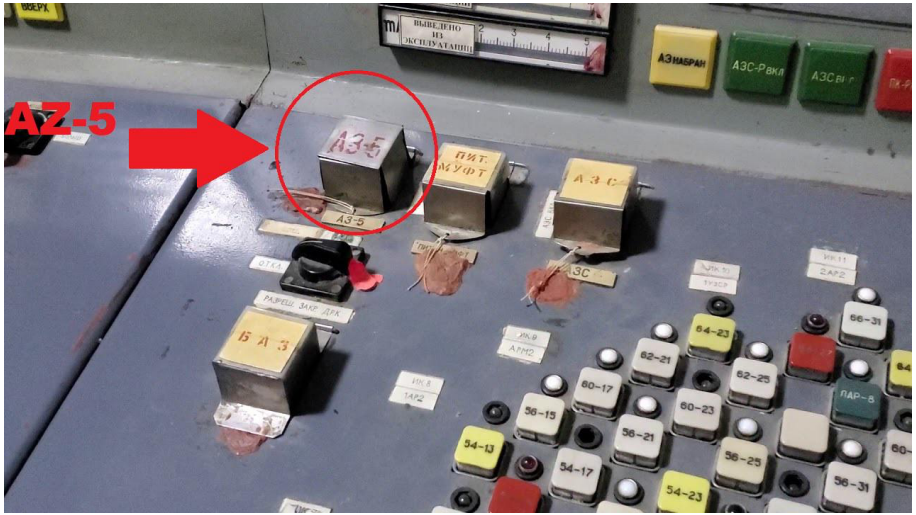
Faults and Fault Tolerant Systems

Reliability

Program Crash

The property that a system can run continuously without interruption or failure for relatively long time periods.

Faults and Fault Tolerant Systems



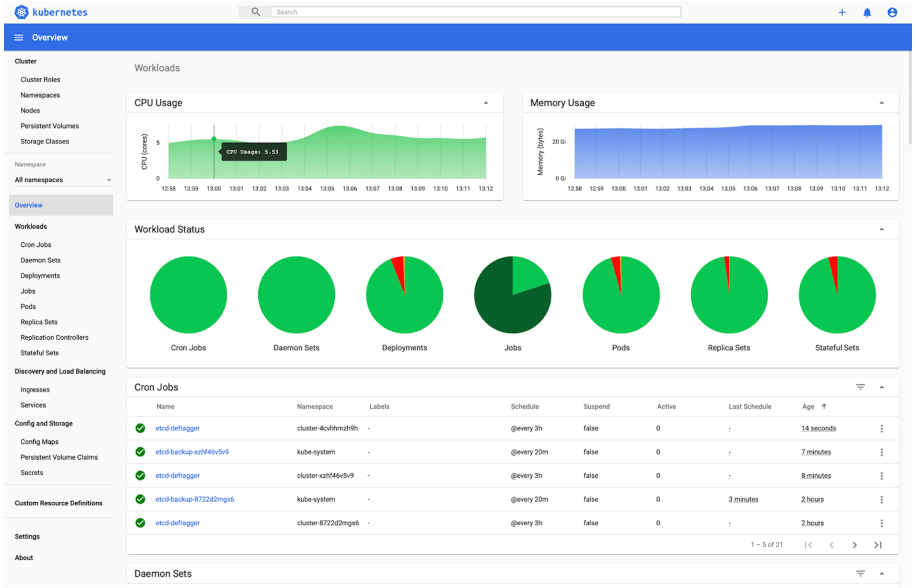
Example : In a nuclear power plant, there is a button designed to safely shut down the plant by inserting all the control rods and effectively terminating all reactions, avoiding possible catastrophic events.

Safety

Critical Controls

The condition that when a system temporarily fails to operate correctly, no catastrophic event happens.

Faults and Fault Tolerant Systems



Example : Kubernetes used for container orchestration in enterprise systems detects failure of instances/containers and auto heals the system by deploying a new fresh instance/container to make up.

Maintainability

Self-healing Systems

The property that how easily a failed system can be repaired, automatically, if possible.

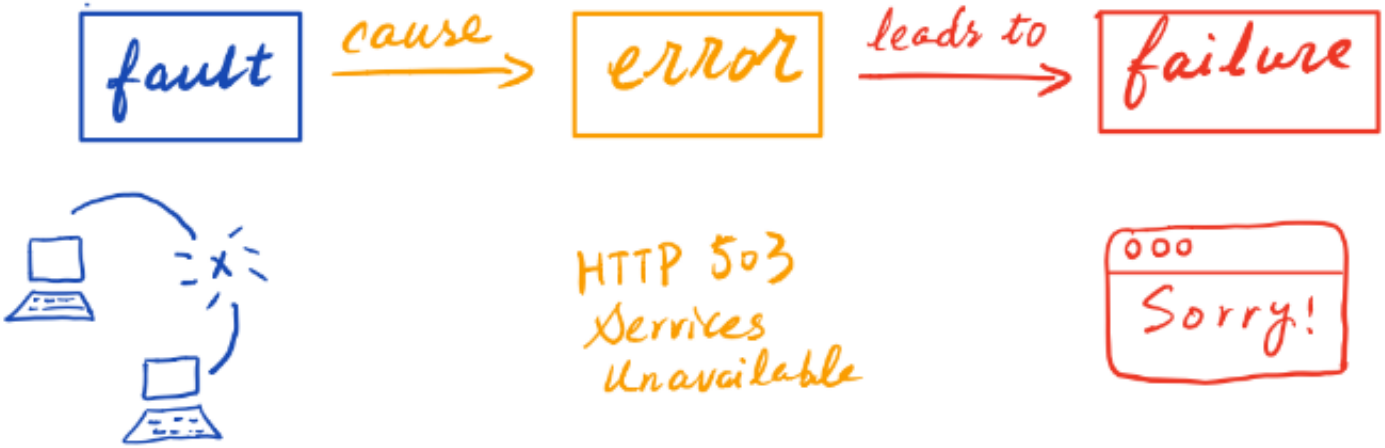
Motivation for Fault Tolerance

Faults are everywhere, some of which inevitable

- program bugs,
- unreliable network,
- faulty process,
- hardware failures,
- even malicious actors

Goal of fault tolerant distributed system: When failures occur, continue to operate in an acceptable way while repairs are being made -- Robustness

Fault, Error and Failure



Fault, Error and Failure

Faults can be

- *Transient* – occur once then disappear
  (e.g. transmission timeout, bitflip due to cosmic ray)

- *Intermittent* – occur at intervals, regularly
  (e.g. loose contact on a connector)

- *Permanent* – exists until the faulty components are replaced
  (e.g. hard-disk crash, software bugs)

Fault, Error and Failure

Type of Failure	Server Behavior
Crash failure	Halts (was working correctly until it halts)
Omission failure	Fails to respond to incoming messages (either fail to receive or fail to send)
Timing failure	Fail to respond within a specified time interval
Response failure	Response is incorrect
Arbitrary failure (a.k.a Byzantine failure)	May produce arbitrary responses at arbitrary times

Quick question : If a process **P** no longer perceives any action from another process **Q**, can process **P** always assume that process **Q** has halted/failed?

Short answer : It depends on the **synchrony assumption** of the network.

Fault Tolerance

Synchrony Assumptions

Communication Models

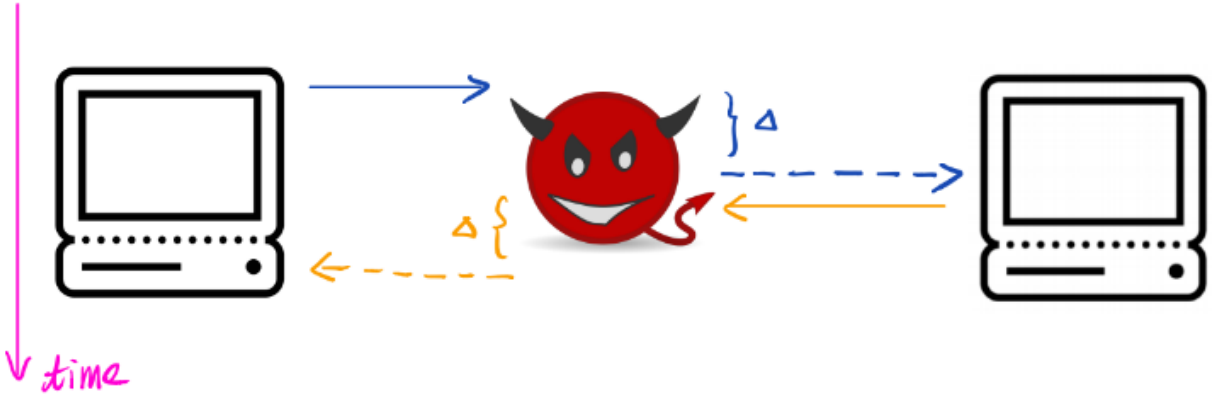
Why do we need different communication models
(aka. synchrony assumption) in case of distributed systems?

- Encapsulates various notions of network reliability and performance in case of different applications (e.g. in case of a distributed file system within an intranet vs. a globally dispersed database with open access by anyone).
- Protocols based on different communication models and synchrony assumptions usually have different complexity of making disparate tradeoff.
- If the assumption is violated (in reality), then the properties guaranteed (in theory) by the distributed system protocols might not hold anymore, leading to unexpected or undesirable outcomes.

Synchrony, Asynchrony and Partial Synchrony [1]

How to describe communication models?

- Communication uncertainty is captured by an **adversary** that controls the message delay in the network under the assumed communication model.
- Communication model defines the **limit of the power** of such an adversary.

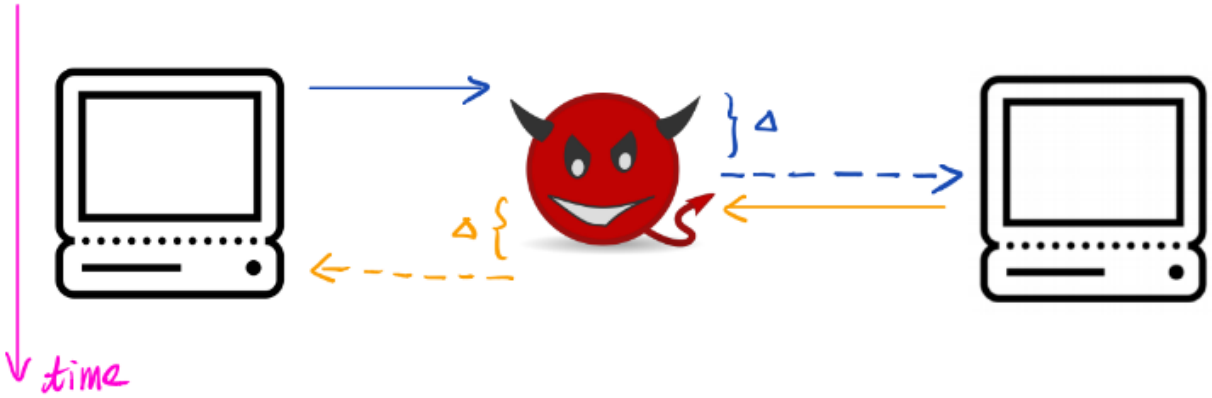


[1] reading: https://decentralizedthoughts.github.io/2019-06-01-2019-5-31-models/

Synchrony, Asynchrony and Partial Synchrony

Synchrony model – known finite time bound Δ

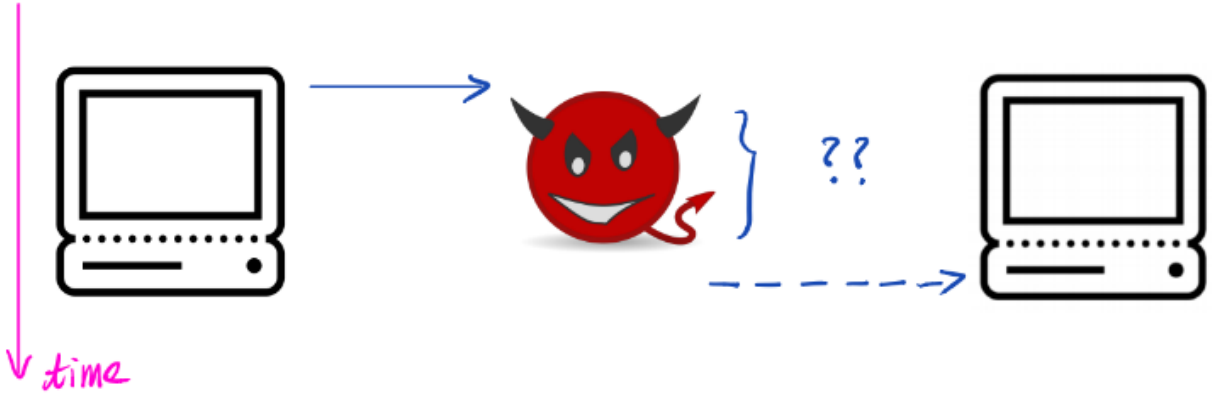
The adversary can delay the message by at most time Δ



Synchrony, **Asynchrony** and Partial Synchrony

Asynchrony model – no time bound, only eventually delivered

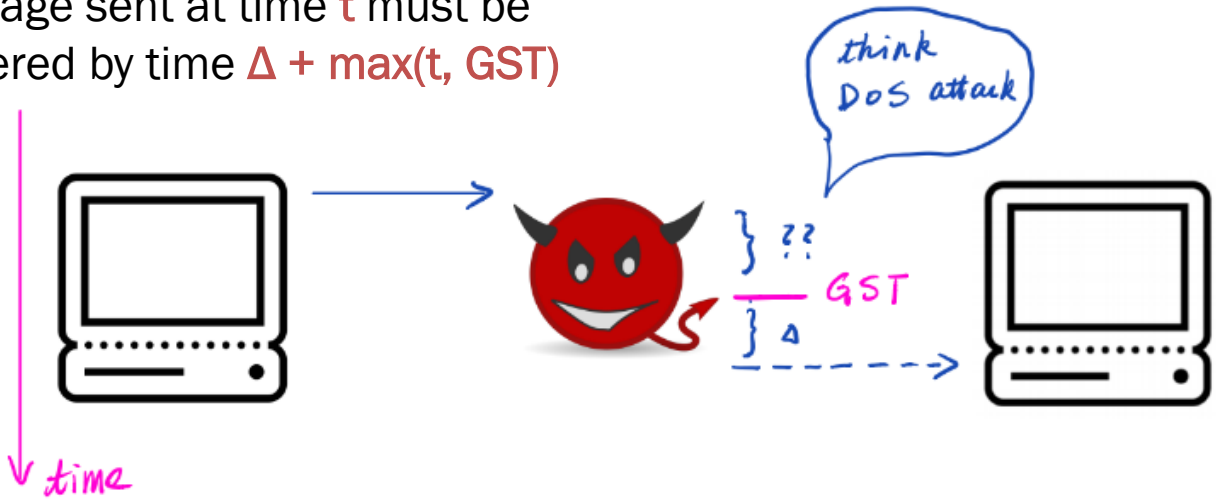
The adversary can delay the message by any finite (arbitrarily long) time



Synchrony, Asynchrony and Partial Synchrony

Partial synchrony model [DLS88]– asynchronous until Global Stabilization Time (GST), then eventually synchronous with known time bound Δ

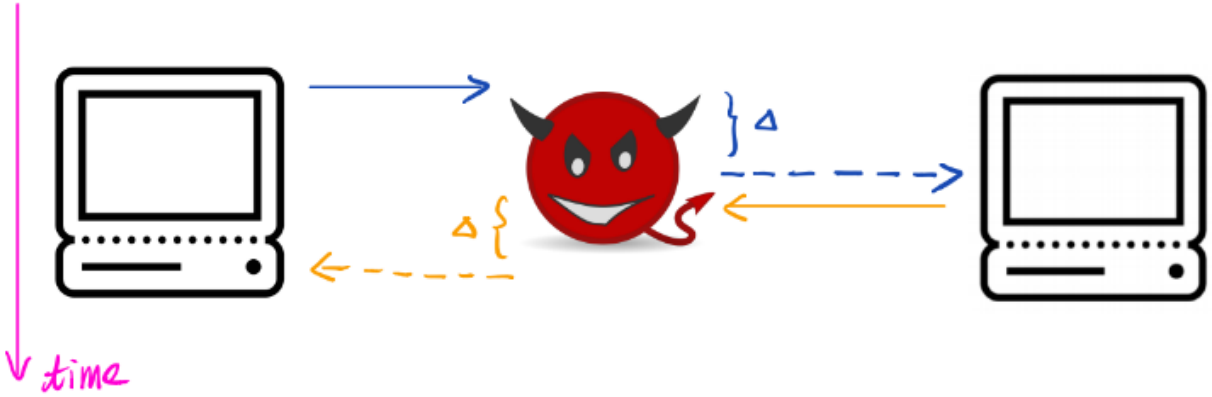
- adversary must cause the GST event to happen after some unknown finite time
- message sent at time t must be delivered by time Δ + max(t, GST)



Synchrony, Asynchrony and Partial Synchrony

Why not using only synchrony model, and set a conservative (large) bound Δ?

- Many protocols run in "rounds/epoch", but a large Δ (e.g. 1 hour) would result in a long timeout for each round of communication, thus degrading performance.
- An aggressively small Δ may not faithfully model the reality, and thus, protocols whose safety relies on the realistically set bound might suffer safety violations.



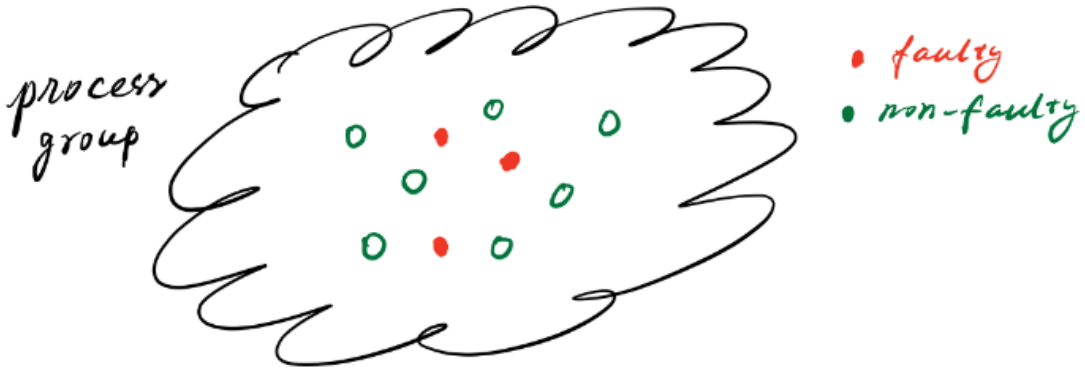
Achieving Fault Tolerance

1. Process resilience (protection against process failure) via **replicating processes into groups**
    - Having a group of identical processes to mask one or more faulty processes
    - Group management required in dynamic (member can leave and join) group
    - Two types of replications – *primary-based* replication (a hierarchical group with a leader) and *replicated-write* replication (a flat group structure)
2. **Consensus** among the group members
    - Ensure that the whole group behaves as "a single, highly robust process”
    - Each non-faulty process executes the same user-defined commands or incoming user requests *in the same order* to maintain homogeneity!

K-Fault Tolerant

A distributed system is k-fault tolerant if it can survive faults in k components and still meet its specifications, requirements and deliverables to the users.

Example : A 2-fault tolerant system would fail in the case with 3 faulty components, whereas a 5-fault tolerant system should work fine under the same circumstances.



Fault Tolerance

Consensus Protocols

What is Consensus? [2]

"Consensus" is different parties coming to an agreement

What counts as an agreement?

- If all parties trivially output 1, does that count as a meaningful agreement?
- Can parties output anything outside of the input values?
- Can a party equivocate and change his/her mind at a later stage?
- What happens if the agreement process does not finish/halt?

What if there are faulty parties?

- Should we allow the faulty parties to crash arbitrarily?
- Should we allow the faulty parties to act arbitrarily and drop messages however he/she likes, and/or send wrong messages at will?
- Should we allow the faulty parties to collude and trick others into a wrong value?

[2] reading: https://decentralizedthoughts.github.io/2019-06-27-defining-consensus/

Consensus vs. Agreement

The Agreement Problem

Assume a set of n nodes, where each node has input v_i from some known set of input values $v_i \in V$, and eventually the protocol *decides* a value $v'_i \in V$.

In case V only contains 0 and 1, this is called a *Binary Agreement Problem*.

Conditions of consensus:

- **Agreement**: No two honest nodes *decide* on different values at the end.
- **Validity**: If all honest nodes have input v, then v must be the decision value.
- **Termination**: Honest nodes must eventually decide on a value in V and halt.

Consensus vs. Agreement

Agreement: no two honest nodes decide on different values at the end

This is the formal definition of our colloquial intuition of "an agreement"

- We don't need to bother whether outputs of the faulty nodes agree with each other, they are FAULTY after all! -- They can burn, crash, die, reply correctly if they like, output gibberish, and/or behave arbitrarily.
- As long as all the honest nodes agree and we have enough honest nodes, then the protocol overall can finally decide on a value.
- If any two honest nodes fail to decide on the same value, it means faulty processes have "successfully tricked" the honest nodes into a bad place, thus indicating a bad protocol design or that it is not fault tolerant enough.

Validity of Consensus

Validity: if all honest nodes have input *v*, then *v* must be the decision value

This is a so-called non-trivial property
preventing a protocol that gives trivial, meaningless output

> Example of a **trivial** consensus algorithm that satisfy "agreement" but not "validity" : Regardless of the input, every honest nodes outputs 1.

Consensus must Halt

Termination: Honest nodes must eventually decide on a value in *V* and halt

We want the protocol to eventually halt and decide or agree on a value. This requires all honest nodes to halt (faulty nodes can fail, crash, output rubbish).

Example : If an honest node receive 5 Yes and 5 No, what should it output?

- If the node waits for more votes, yet everyone has already voted, then the node will be stuck forever, violating the crucial "termination" property.
- If the node just randomly chooses Yes or No, it is quite likely to be different from at least one of other honest nodes, violating the core "agreement" property.
- This means the protocol is terrible, or it cannot tolerate that many faults anyway.

Safety and Liveness !!

Safety **property**: nothing bad will happen

Liveness **property**: something good will *eventually* happen

Note: These two are not concrete properties,
rather two categories/classes of properties.

> If you see *"the consensus protocol is safe in asynchrony, live in synchrony"*, it means safety properties are guaranteed even if the network is asynchronous, but liveness is only upheld in synchronous condition, where the exact meanings of "safety" and "liveness" are to be understood in the context of that protocol.

Safety and Liveness !!

Safety property: nothing bad will happen

- "agreement" will not be disturbed in Agreement Problem
- "no transaction executed twice" in an accounting system
- "consistency" in context of CAP theorem (coming up soon)

Liveness property: something good will *eventually* happen

- "termination" will be guaranteed in Agreement Problem
- "new blocks will always be mined" in a blockchain system
- "document on the cloud is available for access and edit" indicates the liveness being held among the cloud servers running their consensus
- "availability" in context of CAP theorem (coming up soon)

Crash Failure Tolerant Consensus

Paxos (the most popular textbook consensus for crash failure)

- Proposed by the legendary Leslie Lamport in 1998 in a paper called “The Part-time Parliament” capturing the “join and leave” of parliament members analogous to the crash failures encountered by some nodes in the group.
- Reading: https://lamport.azurewebsites.net/pubs/paxos-simple.pdf

Raft (the most widely used real-life consensus for crash failure)

- Aims at much better understandability than the arcane Paxos. Wildly popular and deployed in real systems (e.g. Hyperledger Fabric has option to use Raft).
- This too is a leader-based consensus algorithm with clean “leader election”.
- Reading: https://raft.github.io

Crash Failure vs. Byzantine Failure

In case of public blockchain like Bitcoin or Ethereum …
(actually, in most permissionless blockchain systems)

- anyone (honest or malicious) can join and participate in the consensus process
- it is NOT enough to just use a crash failure tolerant consensus, since malicious nodes (attacker) can act arbitrarily, trying to “bias the outcome” to his/her favor

Let us start with the quintessential problem in this domain …

Byzantine Generals Problem [Shostack, Lamport, 1978]

Byzantine Generals Problem

Context: A group of generals call a meeting to decide/agree on a strategy – whether to attack a city or retreat. Some generals prefer to attack, while others prefer to retreat.

Desired Outcome: Every general must finally agree on a common decision, for a halfhearted attack by a few generals would eventually be worse than either a coordinated attack or a coordinated retreat. Hence the need for consensus.

Constraints: There exist **treacherous generals** in the group, who may cast a vote for a suboptimal strategy/outcome, and they may do so selectively.

Assuming there are **f dishonest/treacherous generals** within the group, how many total generals are required in the group to reach consensus?

Fault Tolerance

Byzantine Generals Problem

Byzantine Generals Problem

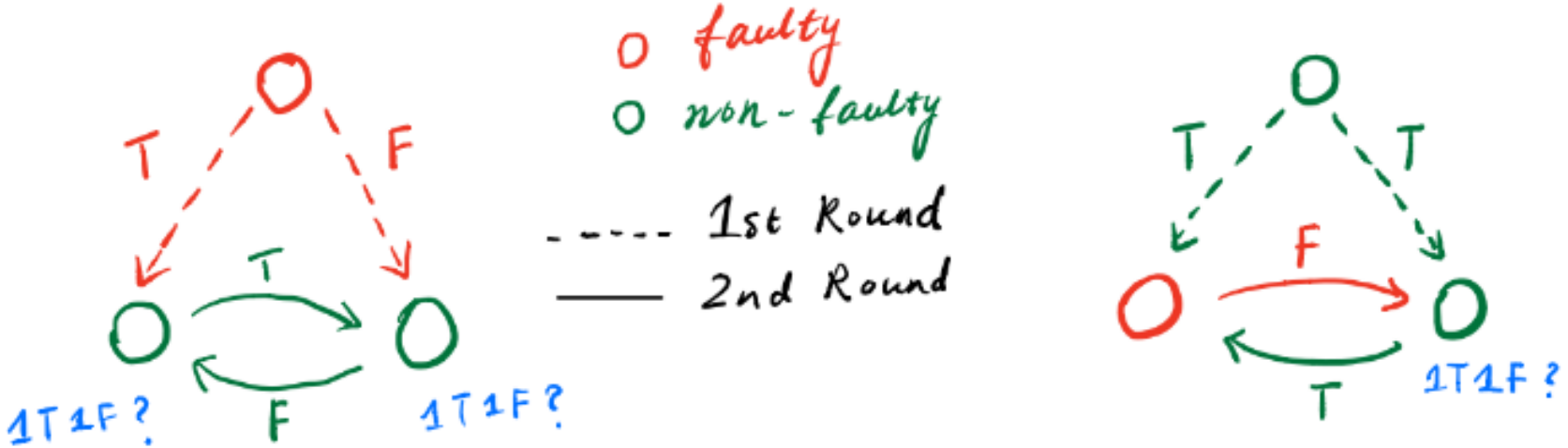
Context	Group of generals in a meeting to decide/agree on a strategy
Outcome	Whether to attack a city or retreat
Individuals	Some generals prefer to attack, while others prefer to retreat
Goals	Every general must finally agree on a common decision
Constraints	There exist treacherous generals in the group
Treachery	Cast selective votes for a suboptimal strategy

Assuming there are **f dishonest/treacherous generals** within the group, how many total generals are required in the group to reach consensus?

Answer : We need at least 3f + 1 generals in the group for consensus.

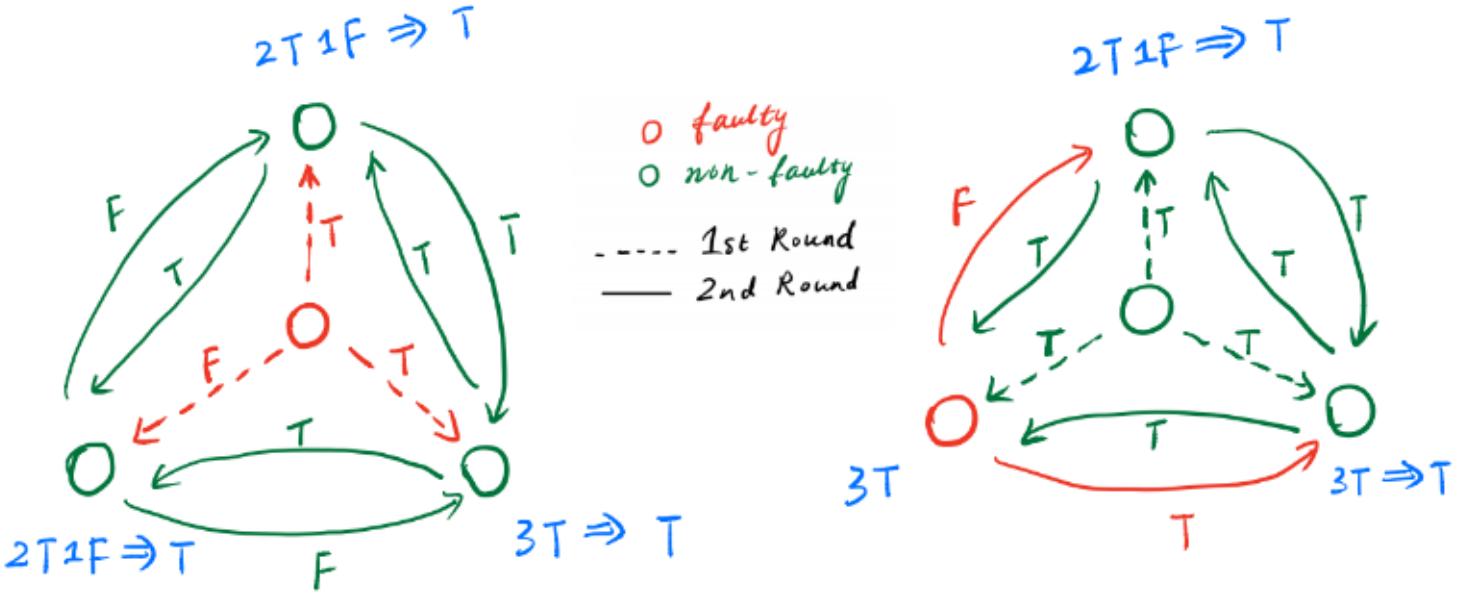
Byzantine Generals Consensus

Why 3f is NOT enough? Think of f = 1 dishonest general.



Byzantine Generals Consensus

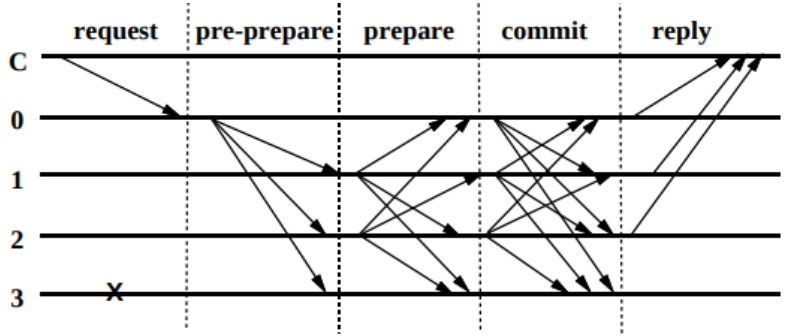
Why 3f + 1 is enough? Think of f = 1 dishonest general.



Practical Byzantine Fault Tolerant (PBFT)

Practical Byzantine Fault Tolerance

Miguel Castro and Barbara Liskov
Laboratory for Computer Science,
Massachusetts Institute of Technology,
545 Technology Square, Cambridge, MA 02139
{castro,liskov}@lcs.mit.edu



Introduced almost 20 years after Paxos (in 1999), the PBFT consensus protocol assumes a practical model of Asynchronous Networks and Byzantine Faults.

Performance is much better than standard protocols, and due to low overhead, it can run in real applications, resulting in an active adoption in the industry.

PBFT Consensus Bounds

PBFT runs among 3f+1 nodes and tolerates up to f byzantine fault

(> 3f+1 replications only degrade performance without improving resiliency)

PBFT is "safe in asynchrony and live in synchrony"

in practice, it works well in partial synchrony

- Safety: all non-faulty replicas agree on a total order for the execution of (concurrent) requests despite errors
- Replicas in PBFT must be deterministic with the same initial states

Fault Tolerance

Limitations and Lower Bounds

CAP Theorem [3]

"When designing distributed web services, there are three properties that are commonly desired: consistency, availability, and partition tolerance. It is impossible to achieve all three." – GL'02

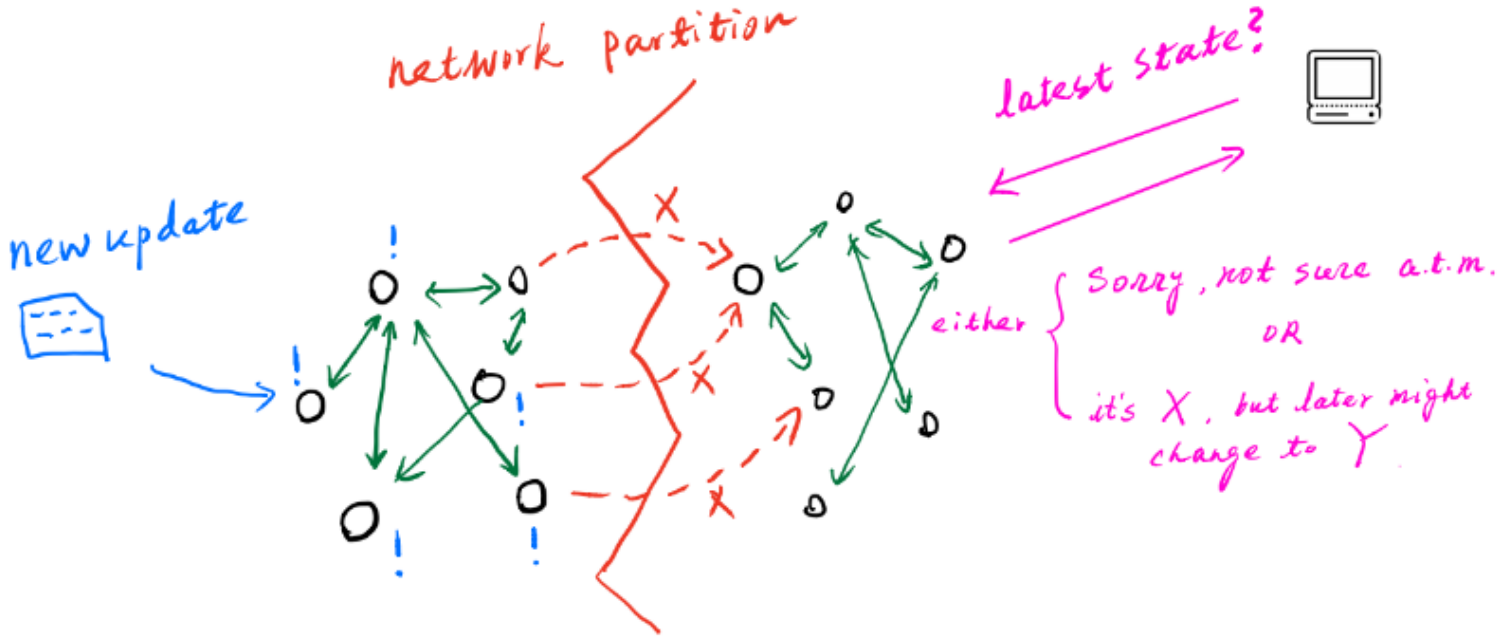
Any networked system providing shared data
can provide only two out of three properties:

1. Consistency: shared, replicated data appears as a single, up-to-date copy
2. Availability: updates will always be eventually executed on the system
3. Partition tolerance: the system will be tolerant to the partitioning of the process group due to failed network

[3] reference: https://dl.acm.org/doi/10.1145/564585.564601

CAP Theorem

Equivalent statement: “In the presence of a network partition, you can choose either consistency or availability (not both) in a distributed system”.



FLP Impossibility [4]

Any protocol **P** solving consensus in the *asynchronous* model, that is, resilient to even just one crash failure, must have an infinite execution time.

Bad news

Deterministic asynchronous consensus is *impossible*.

Good news

With randomization, asynchronous consensus is possible in constant expected time (e.g., in case of HoneyBadgerBFT, discussed later in the course).

[4] reading: https://decentralizedthoughts.github.io/2019-12-15-asynchrony-uncommitted-lower-bound/