

Highly Dependable Systems

Lecture 1

Course introduction

Lecturers: Miguel Matos and Paolo Romano

Instructors – Lectures & Labs

- Prof. Miguel Matos
 - Office hours: Tue: 17:00 @ Discord
- Prof. Paolo Romano
 - Office hours:
 - Tue: 14:00 -18:30 - Sala 514 INESC-ID
 - Fri: 14:00 – 16:00 - Sala 2N3.1 Tagus Park
- Prof. Christof Torres Ferreira
 - Office hours: Thursday: 12:00 - 14:00 - Sala 604 INESC-ID

Contacts

- For administrative / exam questions contact both:
 - miguel.marques.matos@tecnico.ulisboa.pt
 - romano@inesc-id.pt
- For lab / project questions use
 - Discord – To be announced shortly
 - Office Hours

Syllabus

- Dependability fundamentals
- Security and crypto refresher
- Blockchain fundamentals
 - Byzantine fault-tolerant primitives and consensus
 - Other consensus (proof of work, proof of stake, etc.)
 - Blockchain systems
 - Applications: cryptocurrencies, smart contracts, etc.
- Trusted computing

Bibliography

- First half of the course follows closely the following book:
 - Introduction to Reliable and Secure Distributed Programming, 2nd Edition. C. Cachin, R. Guerraoui, L. Rodrigues 2011. Springer - ISBN: 978-3-642-15259-7
 - We will post other materials online

Evaluation methods

- Exam (50%)
 - Minimum grade: 8
- Project (40%)
 - Group of 3 students, minimum grade: 8
- Paper presentation and reviewing (10%)
 - Same group as the project
- The first 2 components are mandatory
 - Failing any them → failing this course

Evaluation methods – project

- 2 stages
- Evaluation based on demo, report, code review and discussion
- Grading:
 - if grade of 2nd stage is higher than the grade of 1st stage then
 - project grade = grade of the 2nd stage
 - else
 - project grade = 50% 1st stage + 50% 2nd stage

Evaluation methods – paper presentation

- Recent papers from top conferences
- Each group presents one paper in one of the last two lectures
 - Technical content
 - Key ideas
 - Positive aspects
 - Negative aspects
 - Reviewing the paper – would you accept this paper if you were on the program committee? Why or why not?
- Questions from the audience
 - Not only clarifications – we expect lots of insightful criticism

Lab info

- Lab registration open through fénix
- If needed, ok for project groups to span different labs or different campi
 - Across campi, presentations and discussions must be in Tagus
- First lab class will be intro to crypto API in Java
- Subsequent classes will focus on project guidance

Questions?

What is dependability?

- A system is dependable if it is able to work (i.e., deliver its service) in a way that is justifiably trusted, namely avoiding service failures that are either more frequent or severe than acceptable.
- Concepts of dependability and trust are intertwined: system A trusts system B if it accepts placing a dependence on system B.

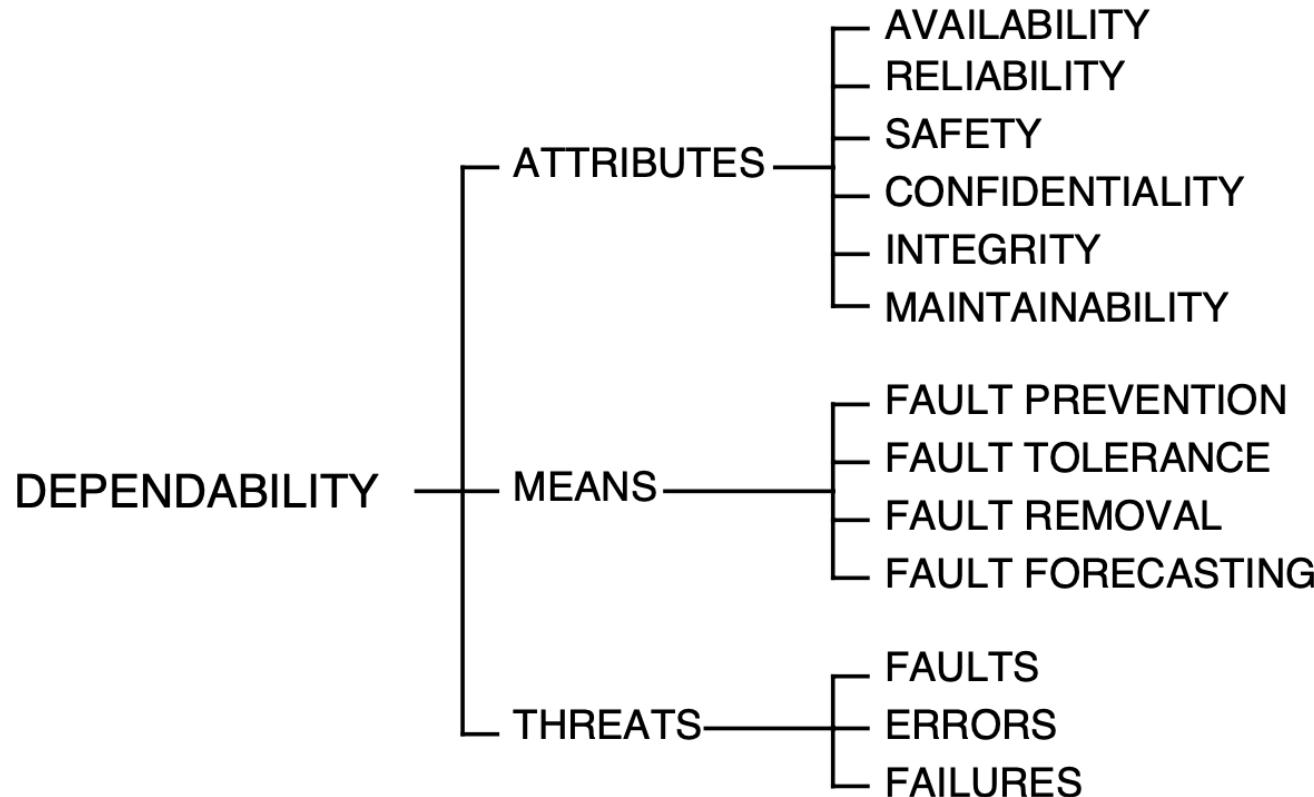
Early history

- First generation of electronic computers (late 1940s to mid 1950s) used rather unreliable components
- Hardware and software redundancy methods were developed (including foundational work from von Neumann, Moore, Shannon in the mid-50s)

A tale of two communities

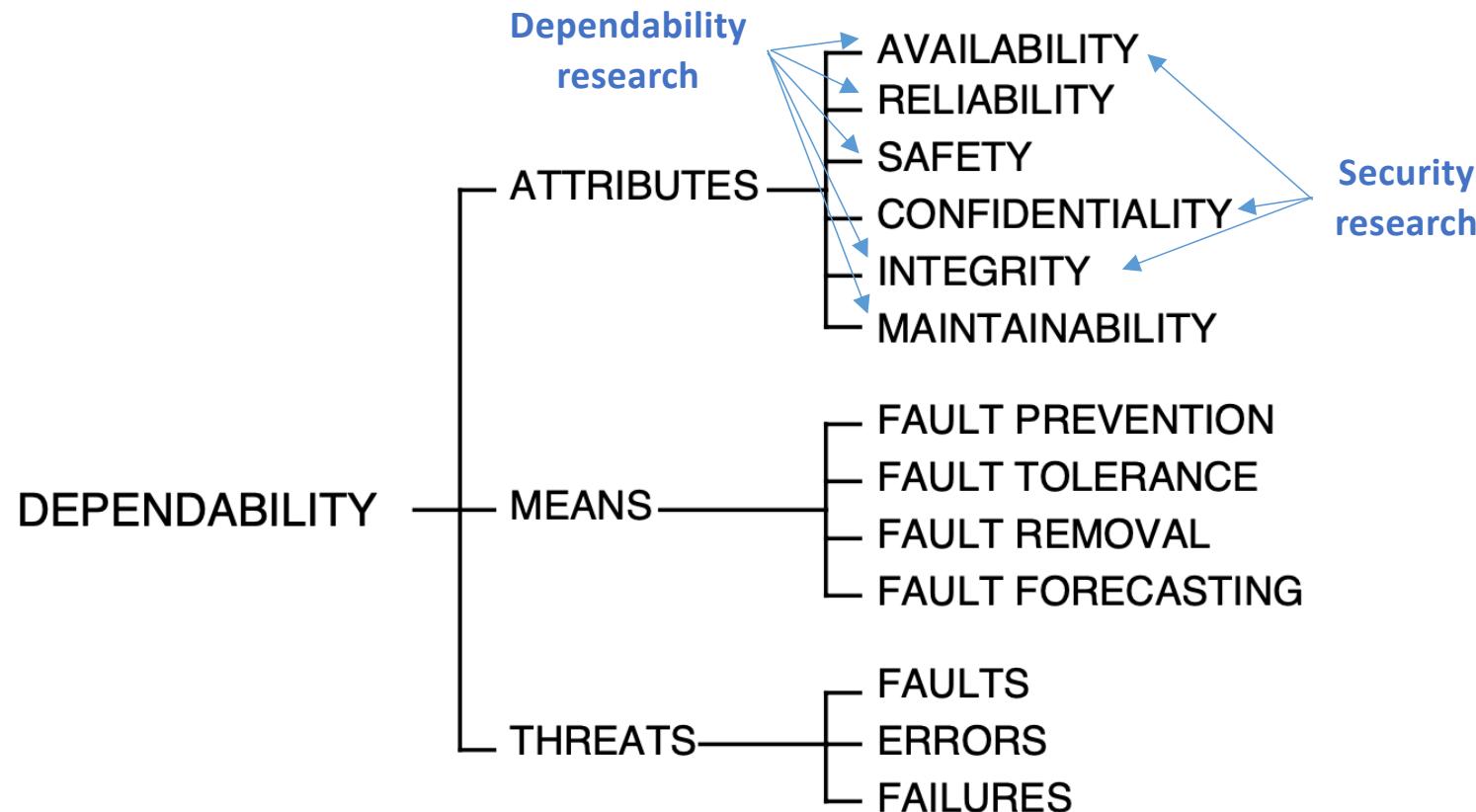
- Dependable systems community evolved from this work, initially focusing on non-malicious faults (machines or components suddenly halting or malfunctioning)
- In parallel, security research community looked at intrusions in computer systems
- Dependability realizes the convergence between the two:
 - Non-malicious faults are only one of the parts of building dependable systems
 - Security research focused on confidentiality, but needed to pay more attention to availability and integrity

Main dependability concepts



- For more details, read: Avizienis, Laprie, Randell. “Fundamental Concepts of Dependability” – available in course fénix page

Main dependability concepts



- For more details, read: Avizienis, Laprie, Randell. “Fundamental Concepts of Dependability” – available in course fénix page

Dependability and security attributes

- Availability
 - *Readiness* for correct service
 - Instantaneous
 - Expressed as a percentage representing the ratio of uptime to total time
- Reliability
 - *Continuity* of correct service
 - Expectation of the predictable remaining uptime, measured using metrics such as Mean Time Between Failures (MTBF), or Mean Time to Fail (MTTF)

Dependability and security attributes

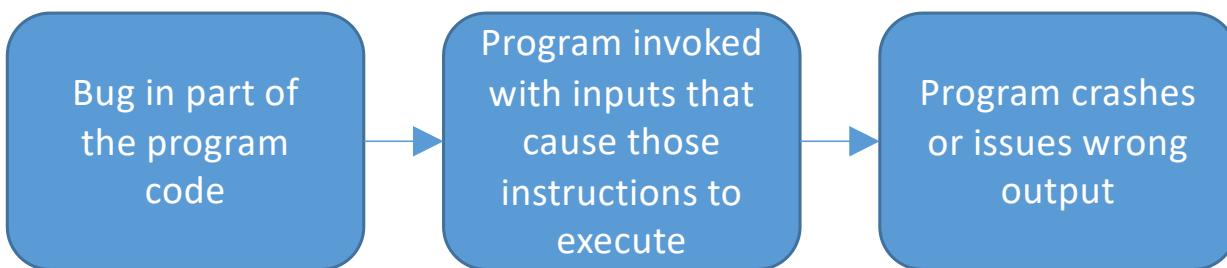
- Safety
 - Absence of catastrophic consequences on the user(s) and the environment
- Confidentiality
 - Absence of unauthorized disclosure of information
- Integrity
 - Absence of improper system alterations
- Maintainability
 - Ability to undergo modifications and repairs

Threats: fault, error, failure

- Faults
 - A fault is the hypothesized cause of an error
- Errors
 - Part of total state of the system that may lead to subsequent service failure
 - Active
 - Latent
- Failure
 - Event that occurs when the delivered service deviates from correct service
 - Time between failure and service restoration is called an outage



Fault, error, and failure by example



Means to attain dependability

- Fault prevention / avoidance
 - means to prevent the occurrence or introduction of faults
- Fault removal
 - means to reduce the number and severity of faults (testing, verification, fault injection)
- Fault forecasting / prediction
 - identify components that are likely to become a source of faults and when they fail
- Fault tolerance
 - means to avoid service failures in the presence of faults

Many concepts are widely used today:

The screenshot shows a Medium article page. At the top right are a search icon, a user profile icon, and a dropdown menu. Below the header, the author's profile picture and name 'Soufiane Bouchaara' are shown, along with a 'Follow' button, the date 'Mar 5, 2022', a '6 min read' duration, and a 'Listen' button. Below the author information is a social sharing bar with 'Save', Twitter, Facebook, LinkedIn, and a link icon. The main title of the article is 'SLA and SLO fundamentals and how to calculate SLA'.

SLA and SLO fundamentals and how to calculate SLA

SLA aka Service-Level Agreement is an agreement you make with your clients/users, which is a measured metric that can be time-based or aggregate-based.

SLA time-based

We can calculate the **tolerable duration of downtime** to reach a given number of nines of availability, using the following formula:

$$\text{availability} = \frac{\text{uptime}}{(\text{uptime} + \text{downtime})}$$

For example, a web application with an availability of 99.95% can be down for up to **4.38 hours max in a year**.

The following table explains the maximum duration of tolerated downtime per year/month/week/day/hour.

	per year	per month	per week	per day	per hour
90%	36.5 days	3 days	16.8 hours	2.4 hours	6 minutes
95%	18.25 days	1.5 days	8.4 hours	1.2 hours	3 minutes
99%	3.65 days	7.2 hours	1.68 hours	14.4 minutes	36 seconds
99.50%	1.83 days	3.6 hours	50.4 minutes	7.20 minutes	18 seconds
99.90%	8.76 hours	43.2 minutes	10.1 minutes	1.44 minutes	3.6 seconds
99.95%	4.38 hours	21.6 minutes	5.04 minutes	43.2 seconds	1.8 seconds
99.99%	52.6 minutes	4.32 minutes	60.5 seconds	8.64 seconds	0.36 seconds
99.999%	5.26 minutes	25.9 seconds	6.05 seconds	0.87 seconds	0.04 seconds

Important output of dependability research
(we will see why in a few slides)

The Byzantine Generals Problem

LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE
SRI International

Reliable computer systems must handle malfunctioning components that give conflicting information to different parts of the system. This situation can be expressed abstractly in terms of a group of generals of the Byzantine army camped with their troops around an enemy city. Communicating only by messenger, the generals must agree upon a common battle plan. However, one or more of them may be traitors who will try to confuse the others. The problem is to find an algorithm to ensure that the loyal generals will reach agreement. It is shown that, using only oral messages, this problem is solvable if and only if more than two-thirds of the generals are loyal; so a single traitor can confound two loyal generals. With unforgeable written messages, the problem is solvable for any number of generals and possible traitors. Applications of the solutions to reliable computer systems are then discussed.

Categories and Subject Descriptors: C.2.4. [Computer-Communication Networks]: Distributed

Byzantine consensus (or agreement) problem

- Defined by Lamport in late 1970s / early 80s
- Treats malicious behavior as arbitrary (a.k.a. Byzantine) faults
- N processes (machines), up to f may fail in an arbitrary fashion
- Each has an input value, all must agree on common output
- For the first two decades (80s, 90s) little attention was paid to Byzantine fault tolerance (BFT)

PBFT (1999)

(Can we get good performance from BFT?)

Appears in the *Proceedings of the Third Symposium on Operating Systems Design and Implementation, New Orleans, USA, February 1999*

Practical Byzantine Fault Tolerance

Miguel Castro and Barbara Liskov

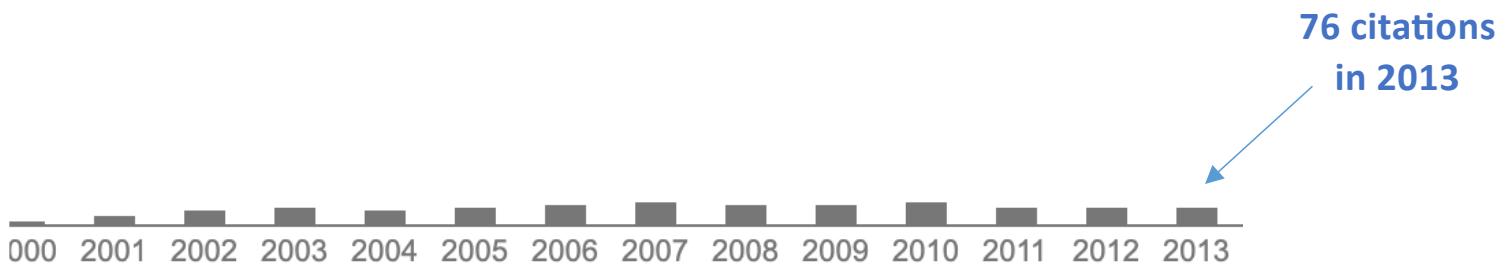
*Laboratory for Computer Science,
Massachusetts Institute of Technology,
545 Technology Square, Cambridge, MA 02139
{castro, liskov}@lcs.mit.edu*

Abstract

This paper describes a new replication algorithm that is able to tolerate Byzantine faults. We believe that Byzantine-fault-tolerant algorithms will be increasingly important in the future because malicious attacks and software errors are

and replication techniques that tolerate Byzantine faults (starting with [19]). However, most earlier work (e.g., [3, 24, 10]) either concerns techniques designed to demonstrate theoretical feasibility that are too inefficient to be used in practice, or assumes synchrony, i.e.,

Picked up some attention, but wore off



Scholar articles [Practical byzantine fault tolerance](#)
M Castro, B Liskov - OsDI, 1999
[Cited by 5017](#) [Related articles](#) [All 118 versions](#)

Until a decade later...

Bitcoin: A Peer-to-Peer Electronic Cash System

Satoshi Nakamoto
satoshin@gmx.com
www.bitcoin.org

The screenshot shows a news article on the Slashdot homepage. The top navigation bar includes links for Stories, Firehose, All, Popular, Polls, Software, and a search bar. Below the navigation is a "Topics" section with links for Devices, Build, Entertainment, Technology, Open Source, Science, and more. A sidebar on the left encourages users to become fans on Facebook. The main content area features a headline "Bitcoin Releases Version 0.3" with a timestamp of "Posted by kdawson on Sunday July 11, 2010 @04:09PM from the nobody-to-prosecute dept.". The article text discusses the peer-to-peer nature of Bitcoin and its proof-of-work concept. An "Ads by Google" banner is visible above the comments section.

Abstract. A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution. Digital signatures provide part of the solution, but the main benefits are lost if a trusted third party is still required to prevent double-spending. We propose a solution to the double-spending problem using a peer-to-peer network. The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power. As long as a majority of CPU power is controlled by nodes that are not cooperating to attack the network, they'll generate the longest chain and outpace attackers. The network itself requires minimal structure. Messages are broadcast on a best effort basis, and nodes can leave and rejoin the network at will, accepting the longest proof-of-work chain as proof of what happened while they were gone.

Teppy writes

"How's this for a disruptive technology? [Bitcoin](#) is a peer-to-peer, network-based digital currency with no central bank, and no transaction fees. Using a proof-of-work concept, nodes burn CPU cycles searching for bundles of coins, broadcasting their findings to the network. Analysis of energy usage indicates that the market value of Bitcoins is already above the value of the energy needed to generate them, indicating healthy demand. The community is hopeful the currency will remain outside the reach of any government."

Here are [the FAQ](#), a paper describing Bitcoin in [more technical detail](#) (PDF), and the [Wikipedia article](#). Note: a commercial service called BitCoin Ltd., in pre-alpha at [bitcoin.com](#), bears no relation to the open source digital currency.

--
[Related: Crypto Tools](#)



Last Updated: 19th February 2016
Who is Satoshi Nakamoto?

Craig Wright is not Satoshi Nakamoto

Craig Wright is not Satoshi Nakamoto. He wasn't Satoshi Nakamoto before or after [Wired](#) and [Gizmodo](#) suspected him to be last year, and he still isn't Satoshi Nakamoto after trying to reveal himself to be on [his own blog](#) and to [The BBC](#), [The Economist](#), [GQ](#), [Jon Matonis](#) and [Gavin Andresen](#).

There is a long and fraught history in Bitcoin of claims and counterclaims about who Satoshi is, and one would think that lessons had been learned and a high standard would be set for subsequent claims regarding Satoshi Nakamoto. The proof posted today by Wright and others does not meet any standard for identifying him as Nakamoto.

Bitcoin is a currency based on cryptography. The ownership of coins can be proven cryptographically and verified by any network participant in a process that is at the core of the system. The proof offered by Wright can not be independently verified.

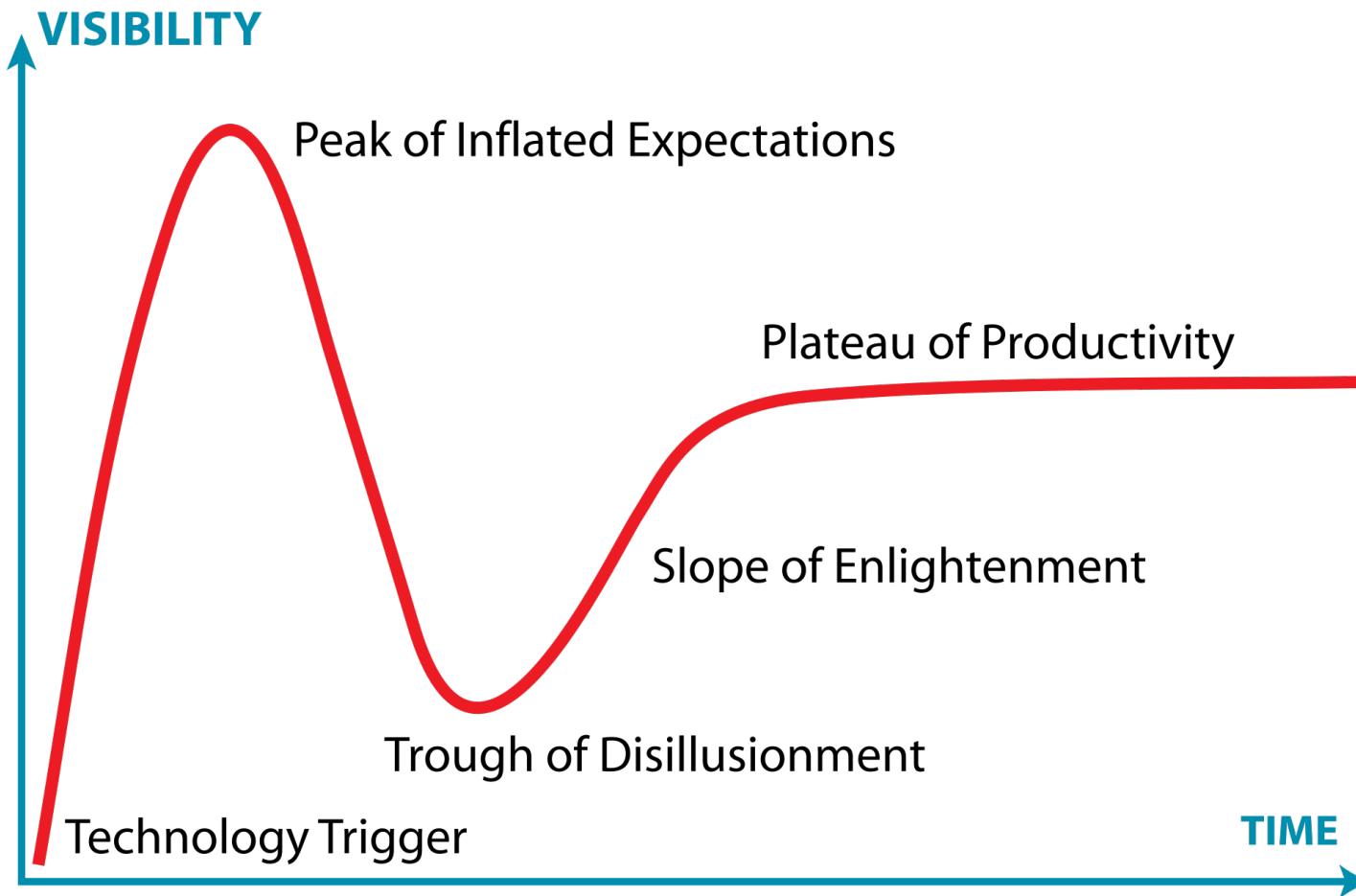


+20,282.86 (6,606.36%) ↑ all time

Feb 14, 19:12 UTC · [Disclaimer](#)



Bears some resemblance?



Source: Wikipedia
(CC BY-SA 3.0)

Are we living a Lehman moment or just growing pains?



BITCOIN NETWORK SHAKEN BY BLOCKCHAIN FORK

VITALIK BUTERIN • MAR 13, 2013

Stellar Switches To Centralized System After Node Issue Causes Accidental Fork

A failure in its consensus system has caused Stellar to fork, requiring a roll back. Stellar claims that the issue affects Ripple as well, something Ripple denies.

ANS

DEC 08, 2014



IAN DEMARTINO

use to the Charges

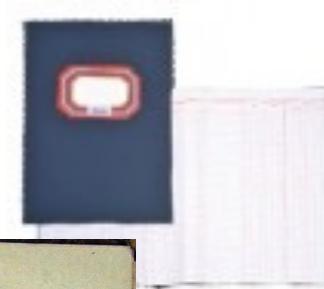
New Insight Into Final Days of FTX

This course covers the principles of blockchains (not just bitcoin)

- Blockchain is the technology behind bitcoin
- Crucial to understand and get the technology right
 - Thus we should treat the subject systematically, based on the principles of dependability

So what is a blockchain then?

- Abstraction of a ledger
 - Append-only sequence of events
 - Tamper-proof and agreed upon by different stakeholders

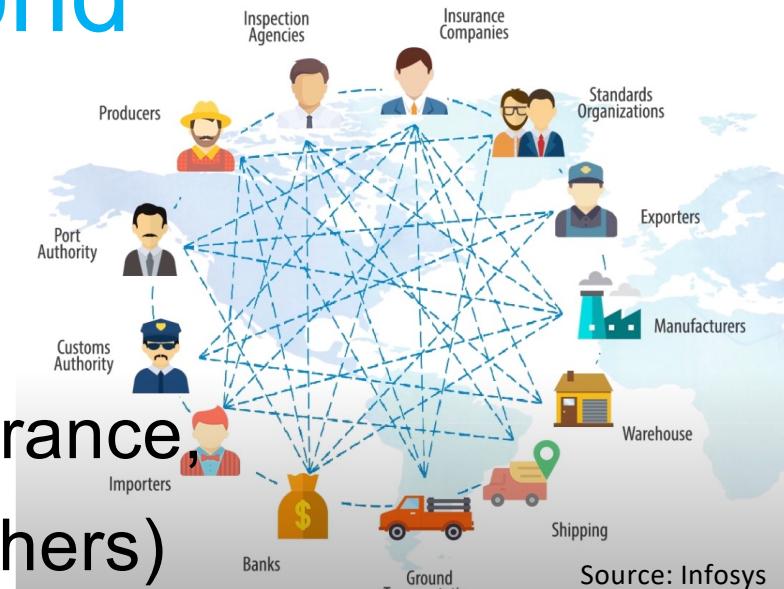


nº	Descrição	descripción	Débito										Crédito												
			Movimentação	Caixa	Salários	Caixa forte	Comissão	Produtos	Pagamento	Cobrados	Despesas	Receitas	Aluguel	Corridos	Produção	Pagamento	Caixa	Salários	Caixa forte	Comissão	Produtos	Pagamento	Cobrados	Despesas	Receitas
Marcos 31	Salário dos servidores da rede	9860										9860													
	Fornecedores da rede	120600										120600													
	Andamento	4800		3000							1800														
	Recebimento	4 667,75		4 667,75																					
	Acrescimos	1 404,00		1 404,00																					
	Descontos	3 323,70				591,50		133,00	2 592,20				3 323,70												
	Rebара de coligadas	271,00		271,00										271,00											
	"	1 404,00				1 404,00																			
		14 862,05		6 667,75	2 710,00	1 404,00		591,50	1 932,00	1 206,00	133,60	2 592,20													
Hort 30	Salários dos servidores	10846										10846													
	Aluguel da rede	170,00										170,00													
	Total	52,00		33,50							19,50														

- Implemented in a distributed way, without requiring centralized trust

Does it have any use beyond cryptocurrencies?

- Example: agriculture supply chain
- Stakeholders: producer, inspection, insurance, logistics, importer, consumer (among others)
- Difficult to trace as goods moves among these
- What if importer wants to know ingredients, place of origin, or ethical / sustainability guarantees of imported produce?
 - Nowadays, even more important due to food/drug safety concerns

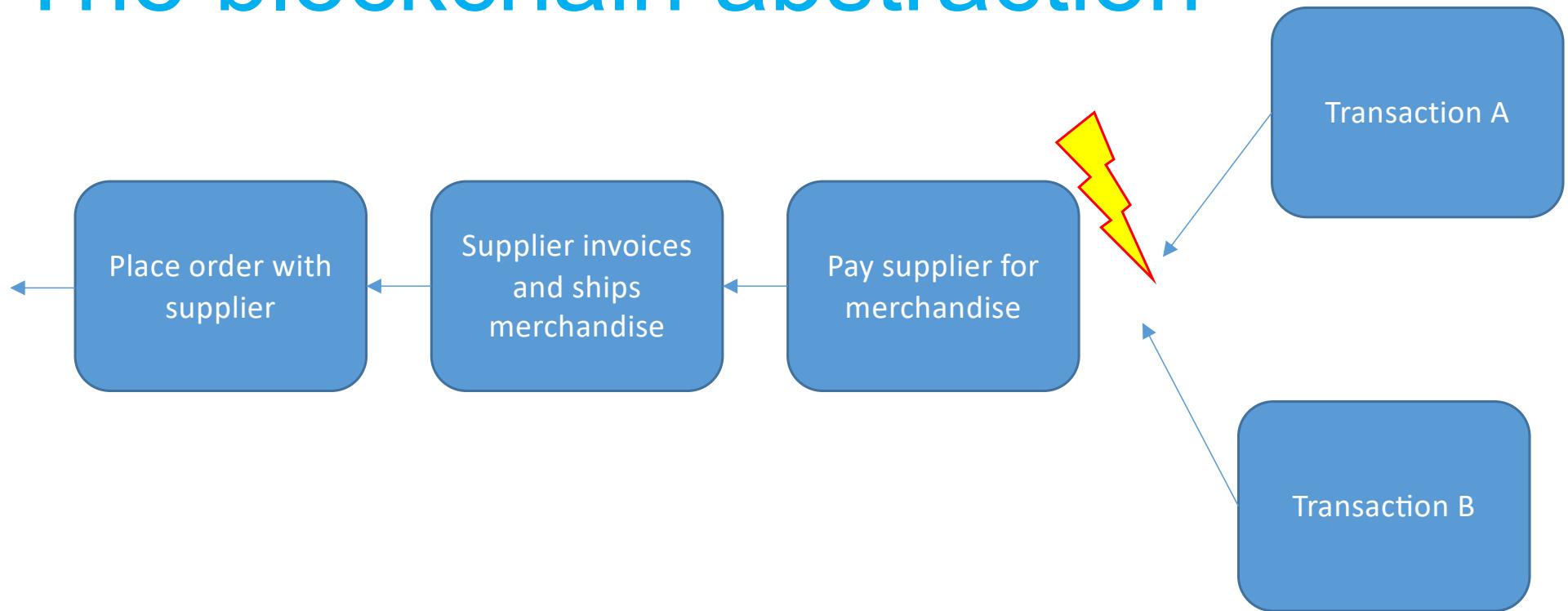


The blockchain abstraction



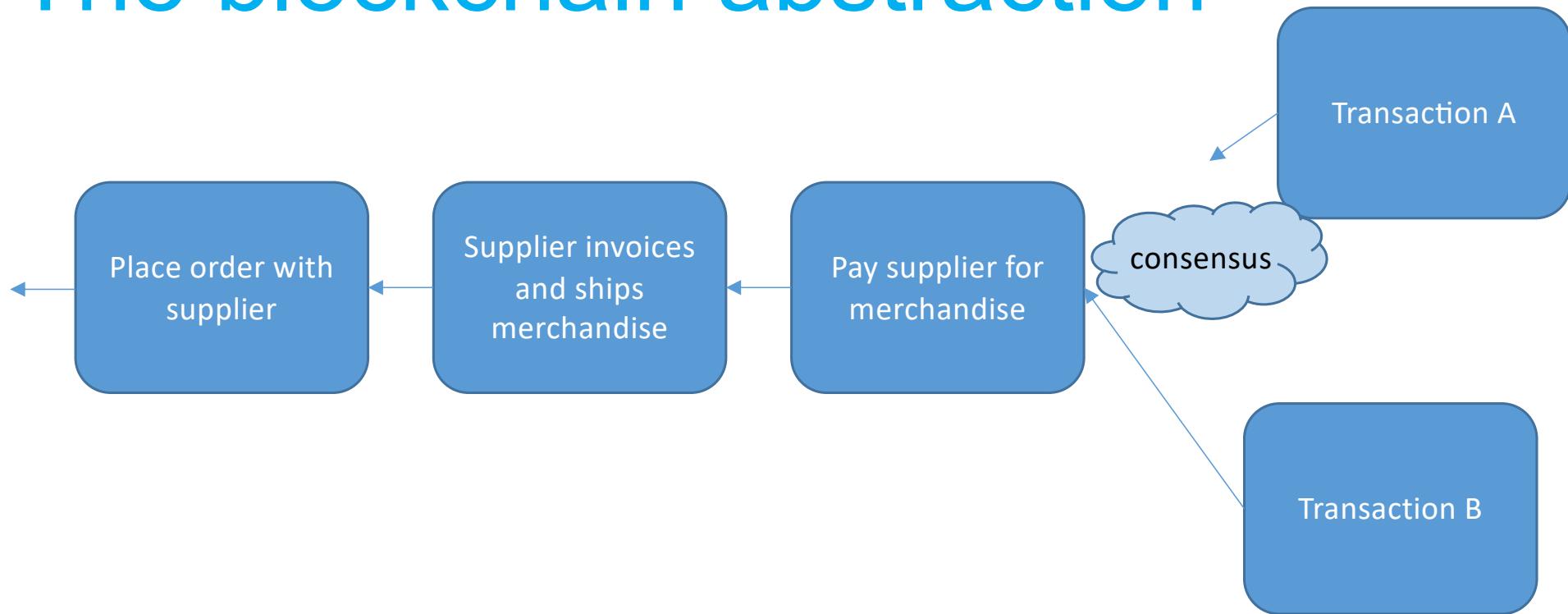
- Each newly inserted block points to its predecessor

The blockchain abstraction



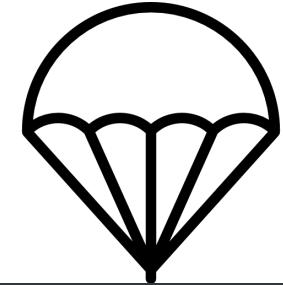
- How to resolve contention for appending a new operation?

The blockchain abstraction



- Run consensus on the next block.
- How to have multiple participants trust the output of consensus?

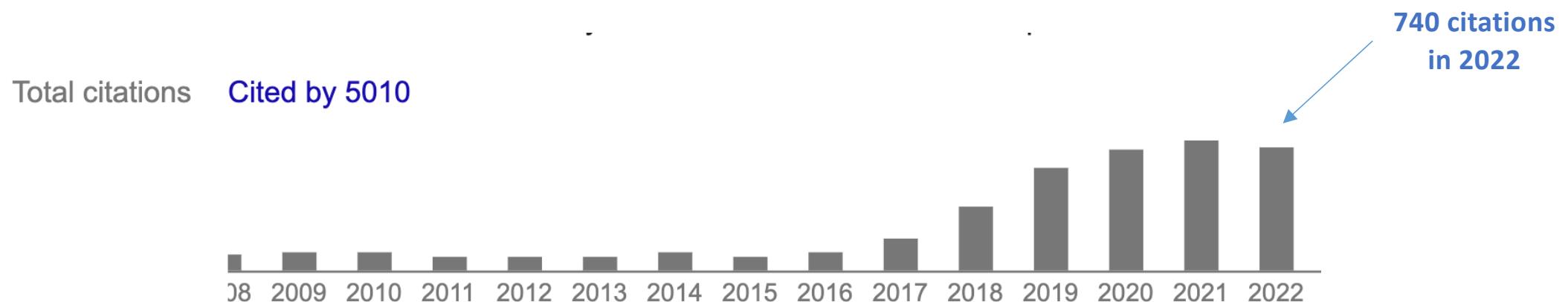
BFT consensus to the rescue



- Caveats:
 - set of machines running consensus are well-known (vetted, closed membership)
 - no more than a fraction (1/3) of the machines collude to break the system
- Next lecture, we will start studying BFT consensus, based on a closed membership
- Later in the course we will understand how open membership blockchains work



PBFT citation count revisited



Scholar articles [Practical byzantine fault tolerance](#)
M Castro, B Liskov - OsDI, 1999
Cited by 5010 Related articles All 118 versions

Rest of today's lecture

- Revisit the basic cryptographic primitives
- We will make available in Fénix slides with a more detailed treatment of the subject

Definitions

- Information
 - Ordered collection of (binary) symbols, with significance and value
- Entity
 - Party participating in a protocol (person, computer, smart device)
- Attacker (or enemy)
 - Attempts to break security properties of system
- Cryptographic key
 - Collection of numerical parameters, may be known only by one or more entities
 - In particular, we assume the attacker does not know some of the keys

Cryptographic primitives

- Symmetric encryption
- Block cipher modes & padding
- Hash functions and MACs
- Asymmetric encryption
- Digital signatures

Kerckhoffs's Principle

- A cryptosystem must be secure despite an attacker that knows all about the system, except the keys
- In other words, security through obscurity doesn't work
- Better to have scrutiny by open experts
 - “The enemy knows the system”
 - Claude Shannon

Symmetric encryption

- Use the same key for encryption and decryption
- Key (**k**) is a shared secret between communicating entities



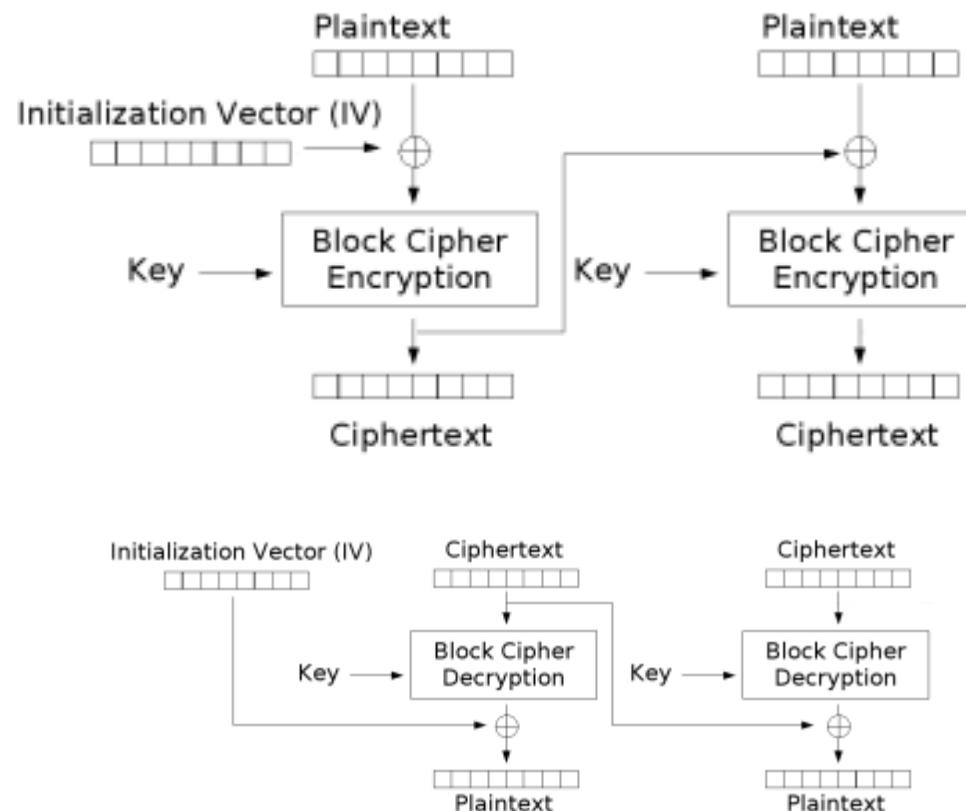
- Without access to k, cannot obtain Plaintext from Ciphertext

Block ciphers and padding

- Block ciphers encrypt fixed size blocks
 - E.g., AES encrypts 128-bit blocks
- How to encrypt larger blocks or streams of data?
- NIST defines 5 modes of operations:
 - Electronic codebook mode (ECB)
 - Cipher block chaining mode (CBC)
 - Output feedback mode (OFB)
 - Cipher feedback mode (CFB)
 - Counter mode (CTR)

High level idea (explained through CBC mode)

- + Repeated plaintext blocks result in different ciphertext blocks
- A corrupted bit in the ciphertext will affect all subsequent blocks
(addressed by other schemes)



Cryptographic Hash Functions

- One-way:
 - For every integer x , easy to compute $H(x)$
 - Given $H(x)$, hard to find any information about x
- Collision resistance:
 - “Impossible” to find pair (x, y) where $x \neq y$ and $H(x) = H(y)$
 - 2nd preimage resistance: given x , it is hard to find $y \neq x$ such that $H(y) = H(x)$.
 - (Strong) collision resistance: it is hard to find any x and $y \neq x$ such that $H(y) = H(x)$.

MACs - Message authentication Codes

- Proves authenticity of a message
 - Not modified in transit (integrity) + generated from the correct sender (authenticity)
 - based on a shared secret key between sender and receiver
- Only the entities who hold the key can:
 - Generate the MAC value
 - Check the MAC value
- Does not provide non-repudiation, nor transferrable authentication

Implementing MACs with hashes (HMAC)

- Strawman: Alice and Bob share K , Alice wants to authenticate m and send it to Bob
 - Alice computes $\text{HMAC}(m, K) = H(K||m)$
 - Alice sends $\langle m, \text{HMAC}(m, K) \rangle$
 - Bob retrieves m and computes $H(K||m)$, compares against received version
 - If they match, message came from Alice (caveat: replay attacks)
- Problem with strawman:
 - Hash functions (e.g., SHA-1) work as state machines, making it easy for an attacker to produce $H(K||m||m')$ from $H(K||m)$
 - Note: $||$ is the concatenation operator

RFC 2104 - HMAC: Keyed-Hashing for Message Authentication

- $\text{HMAC}(m, k) = \text{hash}(k \oplus \text{opad} \parallel \text{hash}(k \oplus \text{ipad} \parallel m))$
 - ipad = the byte 0x36 repeated B times
 - opad = the byte 0x5C repeated B times
 - B = block length in bytes

Asymmetric (or public key) encryption

- Each entity has a pair <public key, private key>
- Suppose that Bob wants to send encrypted message to Alice



- Without access to Alice's private key, cannot obtain Plaintext from Ciphertext

Digital signatures

- Provide integrity + authenticity of the author of a message, in a way can be verified by third parties to resolve disputes
 - Non-repudiation property: author cannot claim it did not sign a message (assuming private key was not leaked)

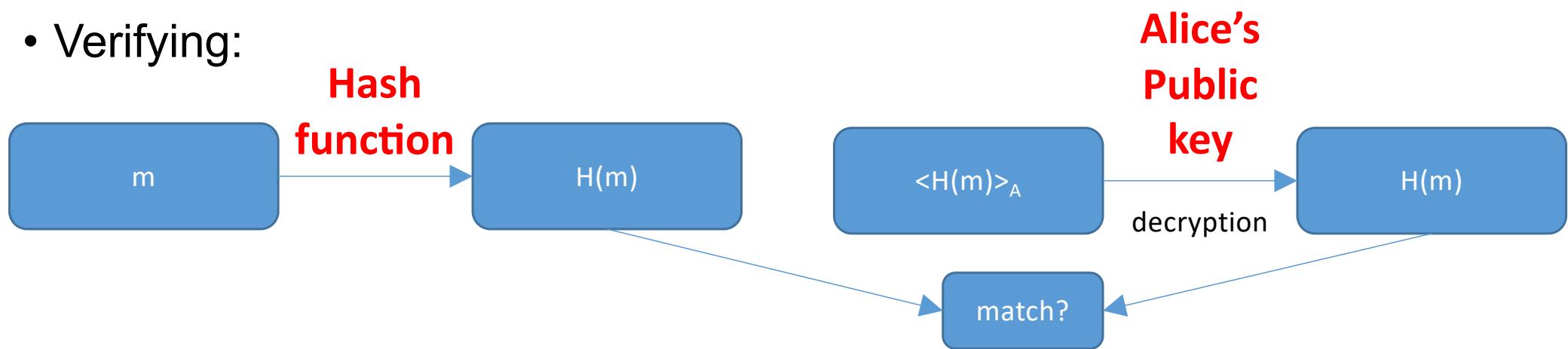
Digital signature scheme

- Signing:



- Alice transmits both m and the signature $\langle H(m) \rangle_A$

- Verifying:



Next lecture: BFT protocols (building BFT consensus)

- Acknowledgements:
 - Some crypto slides from Ricardo Chaves @ IST, AISS course
 - Some intro slides from Maurice Herlihy @ Brown Univ.

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

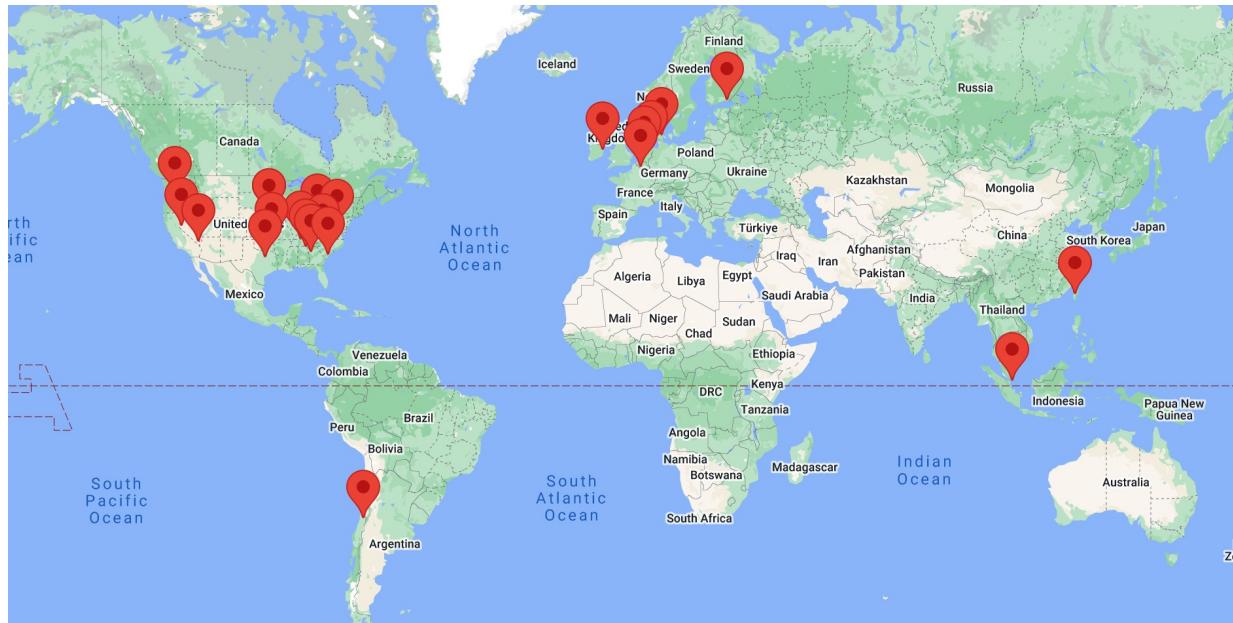
The pessimistic view

- Concurrency => different interleavings => incorrectness
- Partial failures => loss of state => 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)

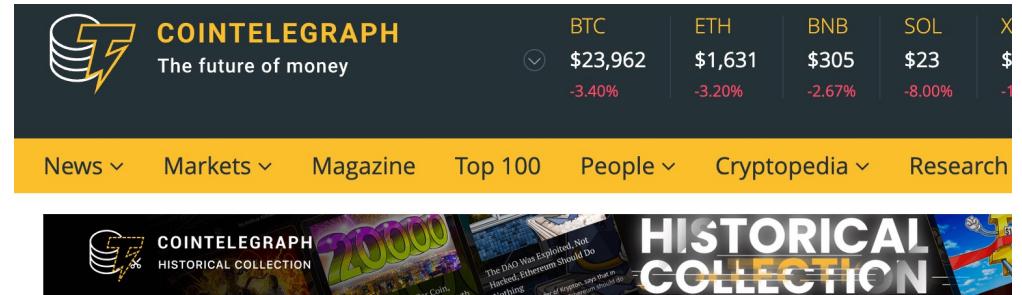


Failure is the norm, not the exception

- 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

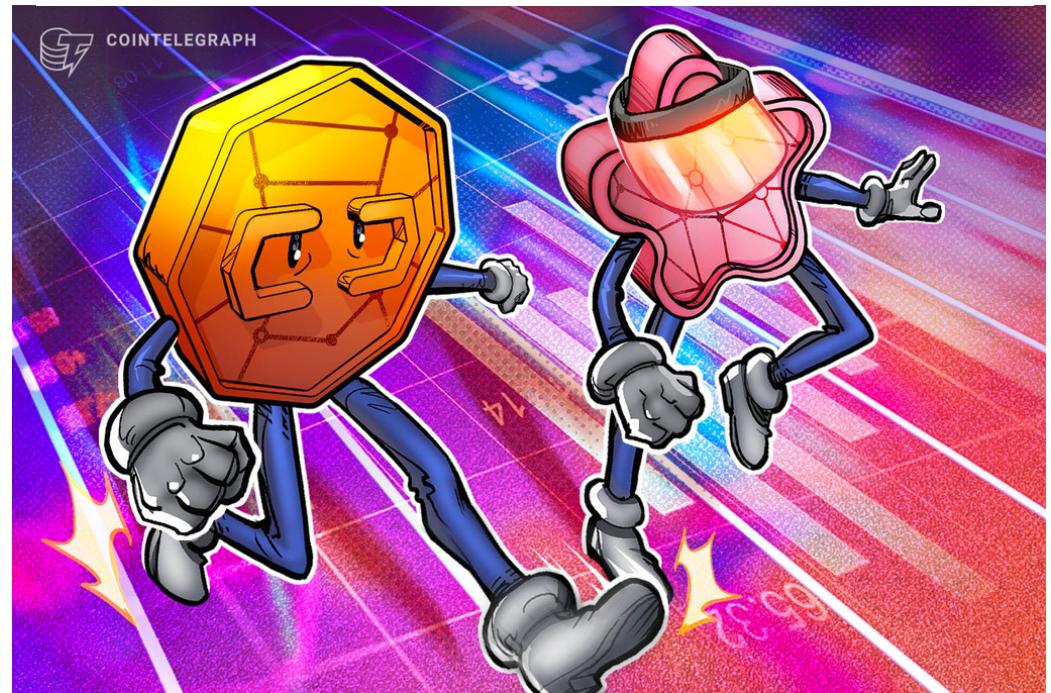
- Even in a fixed membership setting, machines fail or become unreachable
- Furthermore, need to reach consensus without having to trust a single entity or any individual participant
 - Adversarial setting - subset of machines may try to sway outcome (e.g., front running attacks try to reorder transactions)



What is front-running in crypto and NFT trading?

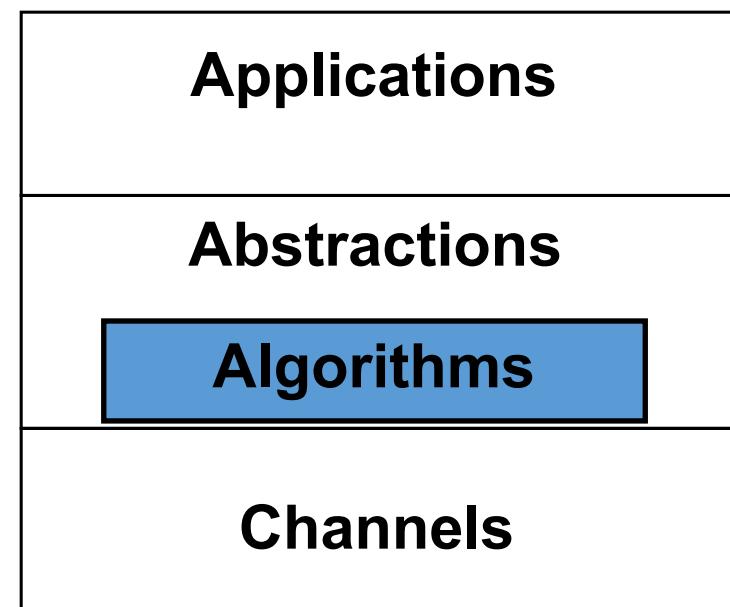
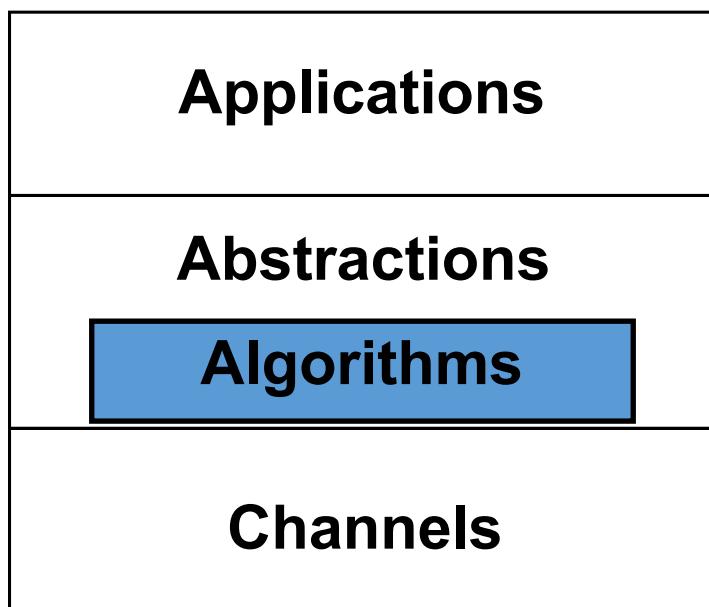
Onkar Singh

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 $p_1,..p_N$ 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

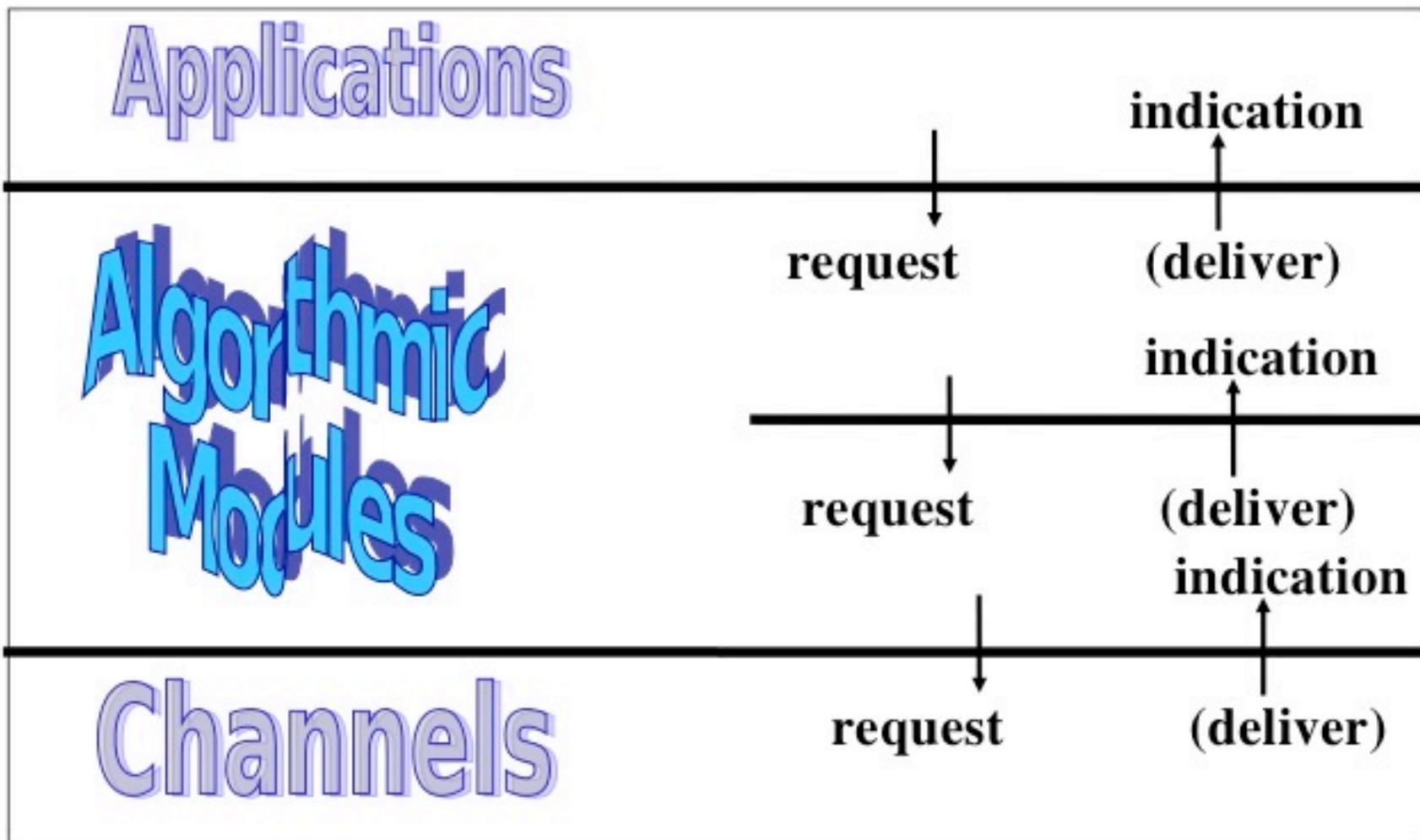
Processes

- 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 $m \notin$ 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:
 - MAC a $\leftarrow \text{authenticate}(\text{send_proc } p, \text{rec_proc } q, \text{message } m)$
 - 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 U { 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: <crash, p>
- 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: <Leader, p>
- 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 <Complain, p> 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

Towards a dependable consensus: Broadcast primitives

Highly dependable systems

Lecture 3

Lecturers: Miguel Matos and Paolo Romano

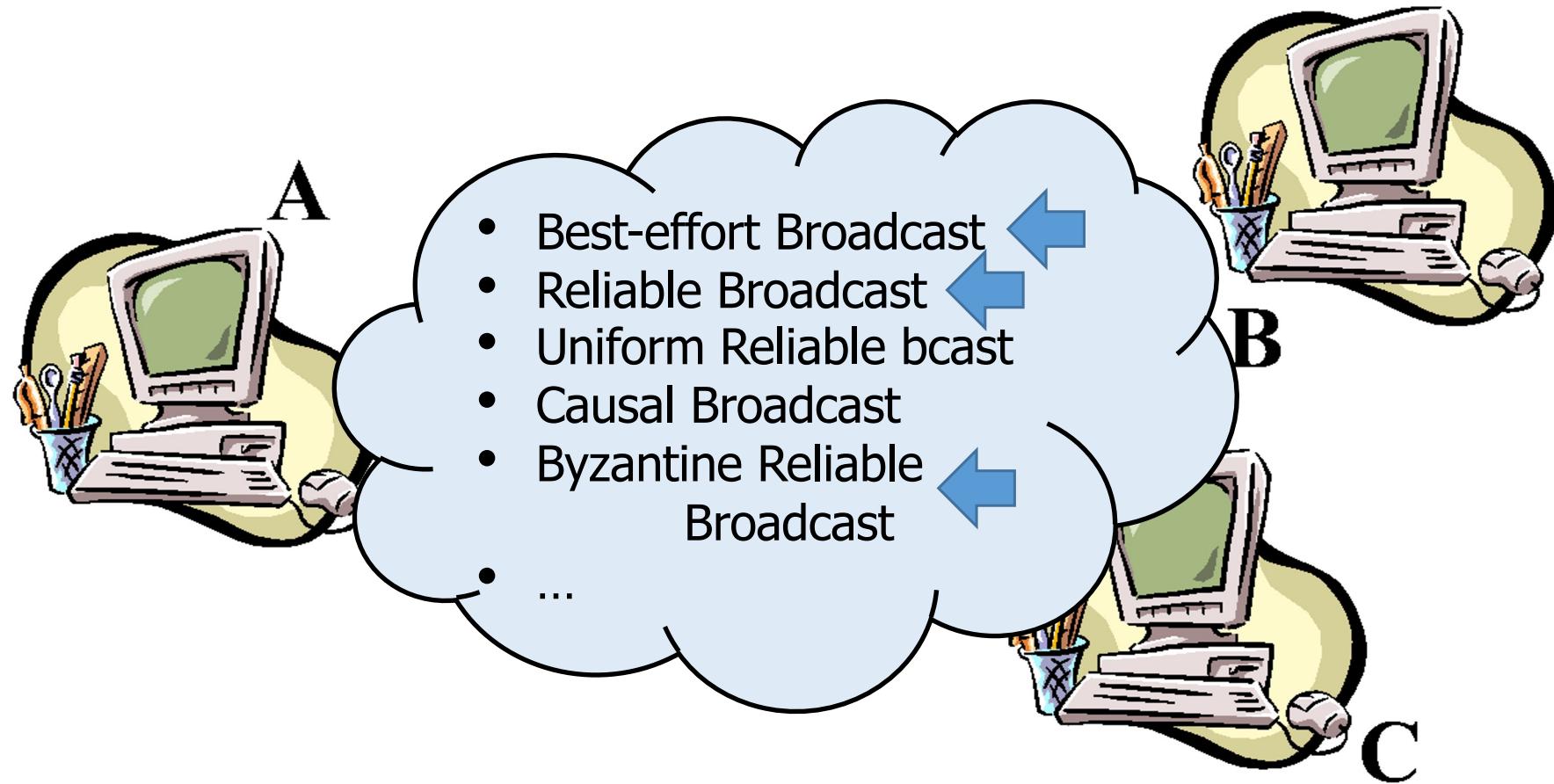
Last lecture: safety and liveness

- 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

Last lecture - Byzantine Leader Election

- 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.

Broadcast Abstractions



Intuition

- Broadcast is useful per se, for instance, in applications where some processes subscribe to events published by other processes (e.g., stocks)
- But also, a crucial building blocks in other protocols, namely consensus
- The receivers might require some **reliability** guarantees from the broadcast service (we say sometimes quality of service QoS) that the underlying network does not provide

Overview

We shall consider three forms of reliability for a broadcast primitive

- (1) Best-effort broadcast ←
 - (2) Reliable broadcast in the crash fault model
 - (3) Reliable broadcast in the arbitrary fault model
- We provide specifications and then algorithms

Best-effort Broadcast (beb)

- **Events**

- Request: <bebBroadcast, m>

- Indication: <bebDeliver, src, m>

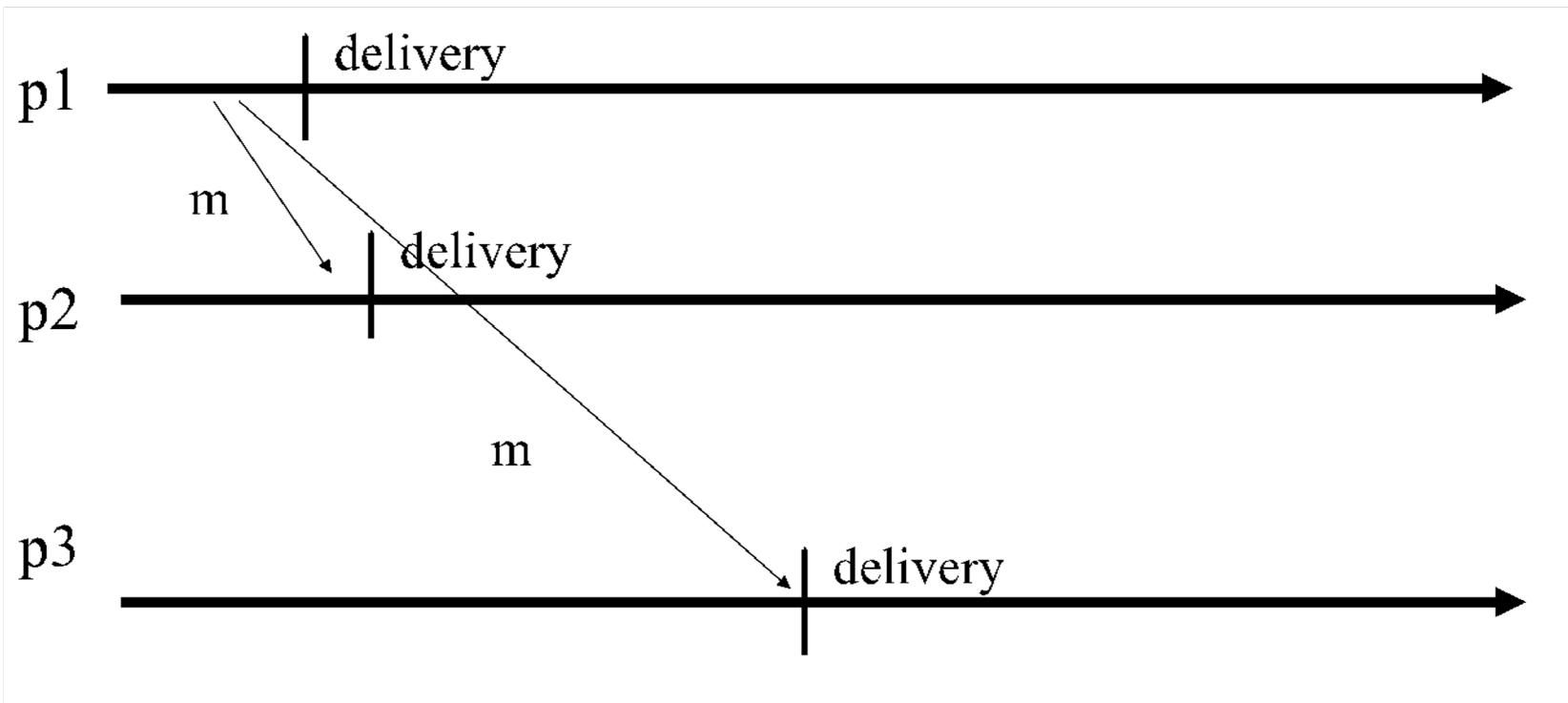
- **Properties: BEB1, BEB2, BEB3**

Best-effort broadcast (beb)

■ Properties

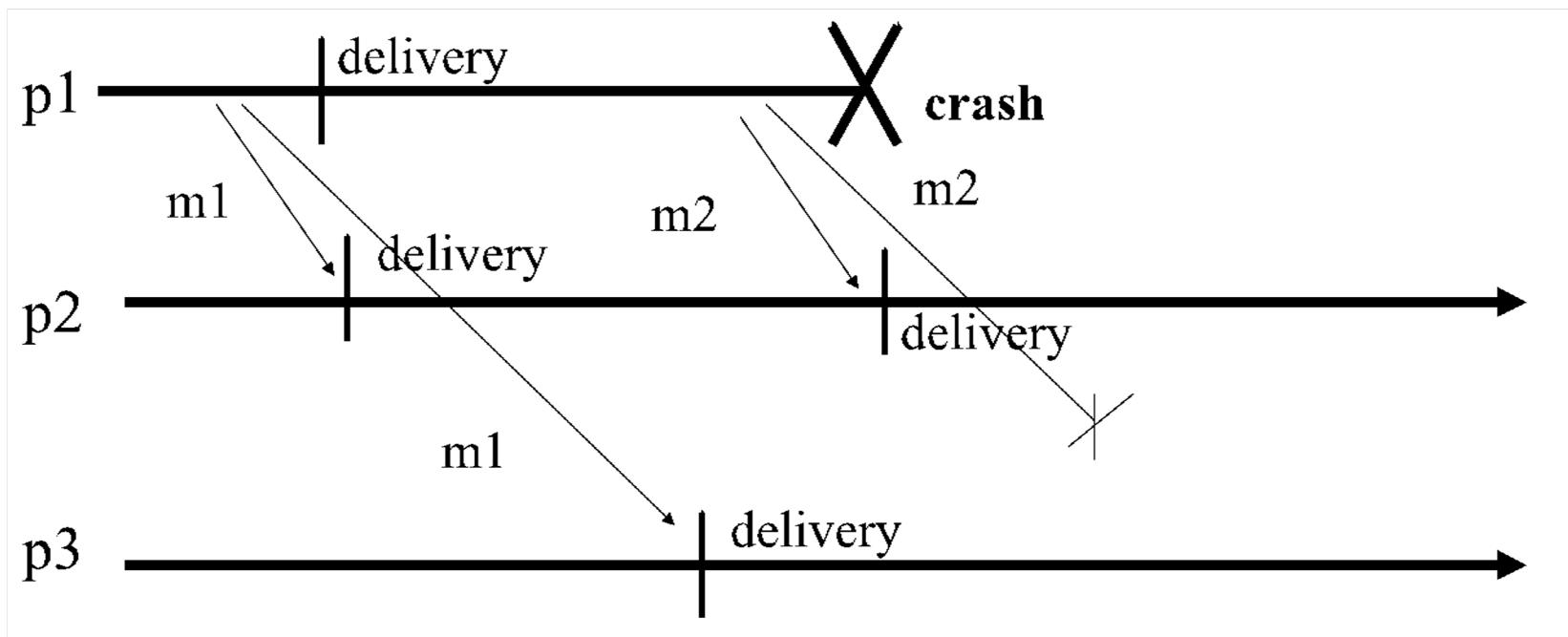
- **BEB1. Validity:** If p_i and p_j are correct, then every message broadcast by p_i is eventually delivered by p_j
- **BEB2. No duplication:** No message is delivered more than once
- **BEB3. No creation:** No message is delivered unless it was broadcast

Best-effort Broadcast



Meets the specification

Best-effort Broadcast



Meets the specification

Overview

We shall consider three forms of reliability for a broadcast primitive

- (1) Best-effort broadcast ←
- (2) Reliable broadcast in the crash fault model
- (3) Reliable broadcast in the arbitrary fault model
- We provide **specifications** and then **algorithms**

Algorithm (beb)

- **Implements:** BestEffortBroadcast (beb).
- **Uses:** PerfectLinks (pp2p).
- **upon event** < bebBroadcast, m> **do**
 for all $pi \in S$ **do**
 trigger < pp2pSend, pi, m>;
- **upon event** < pp2pDeliver, pi, m> **do**
 trigger < bebDeliver, pi, m>;

Algorithm (beb)

- **Proof (sketch)**
 - **BEB1. Validity:** By the validity property of perfect links and the very facts that (1) the sender sends the message to all and (2) every correct process that pp2pDelivers a message bebDelivers it
 - **BEB2. No duplication:** By the no duplication property of perfect links
 - **BEB3. No creation:** By the no creation property of perfect links

Overview

We shall consider three forms of reliability for a broadcast primitive

- (1) Best-effort broadcast
 - (2) Reliable broadcast in the crash fault model ←
 - (3) Reliable broadcast in the arbitrary fault model
-
- We provide **specifications** and then **algorithms**

Reliable Broadcast (rb)

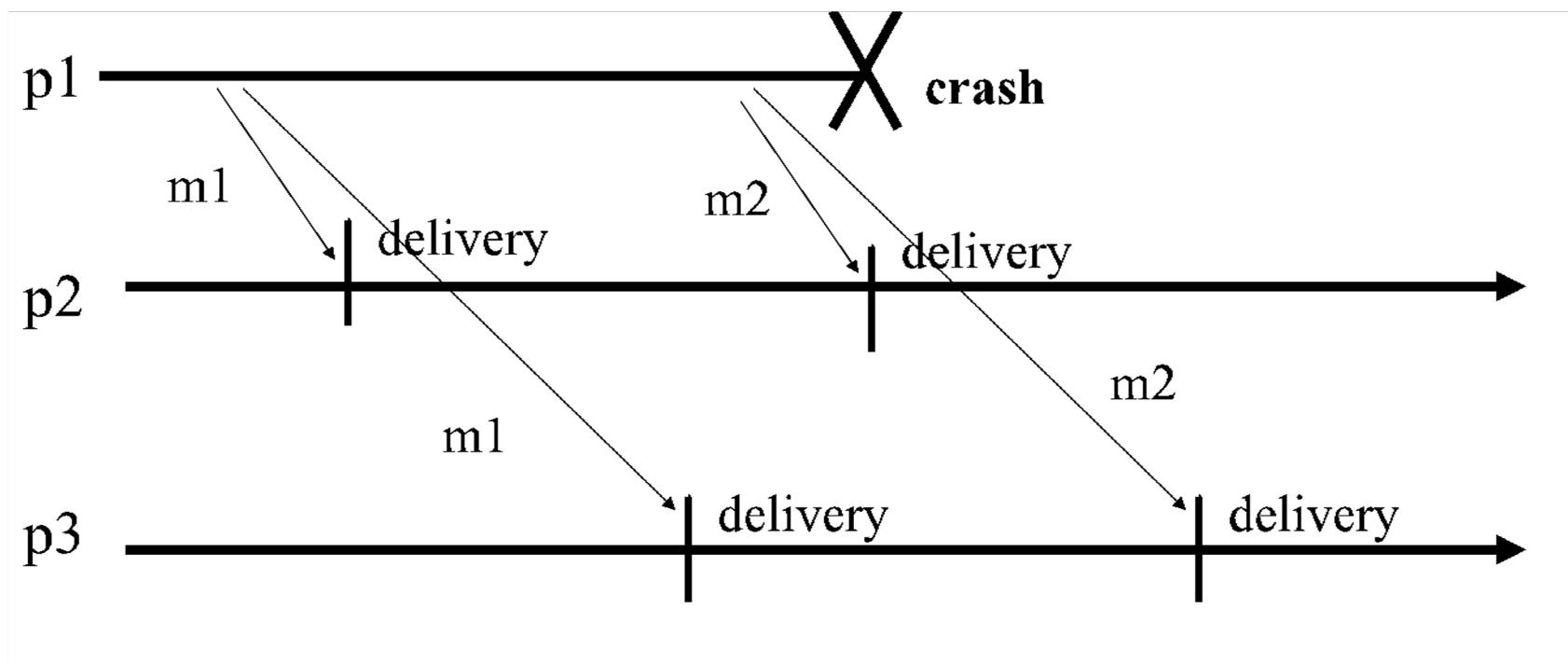
- **Events**
 - Request: <rbBroadcast, m>
 - Indication: <rbDeliver, src, m>
- **Properties: RB1, RB2, RB3, RB4**

Reliable broadcast

■ Properties

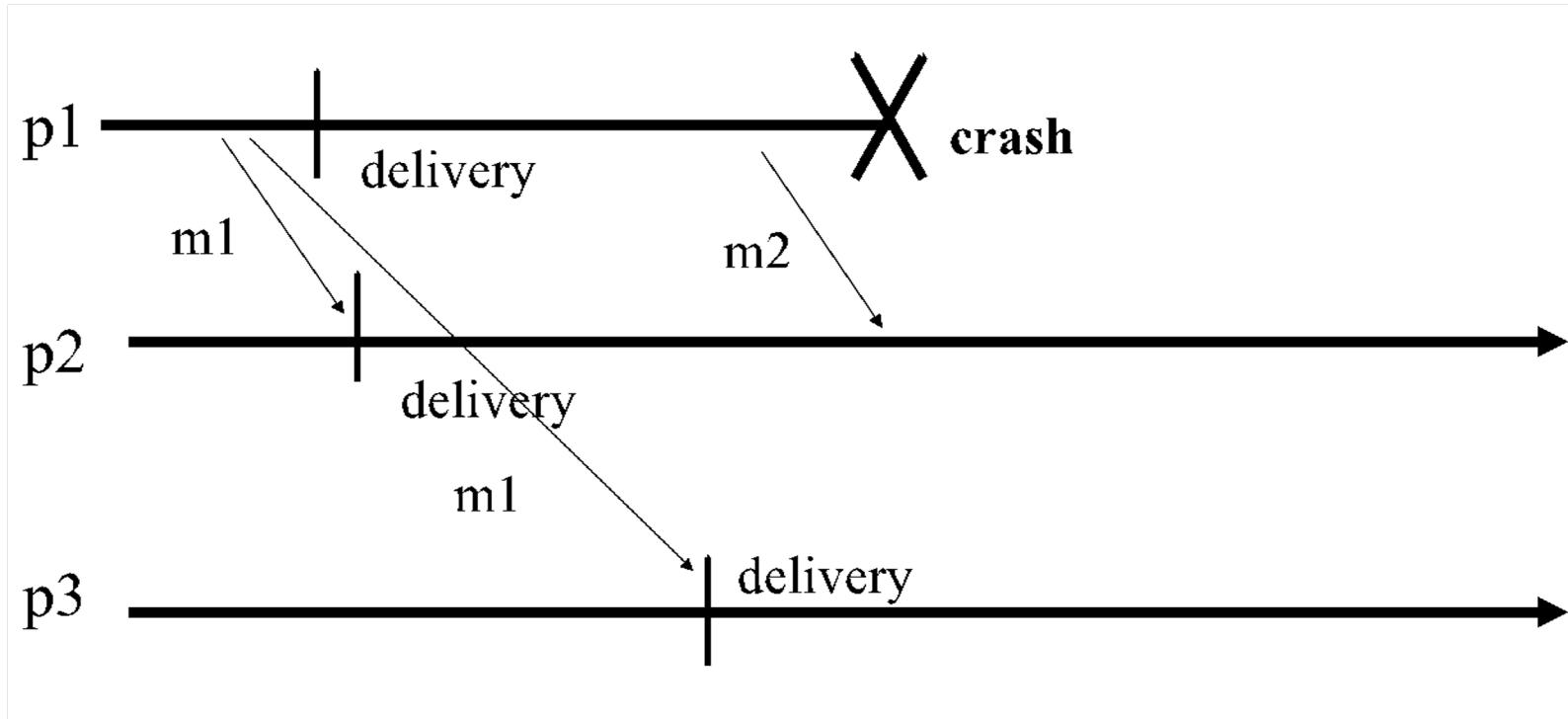
- **RB1 = BEB1.**
- **RB2 = BEB2.**
- **RB3 = BEB3.**
- **RB4. Agreement:** For any message m , if a correct process delivers m , then every correct process delivers m

Reliable Broadcast



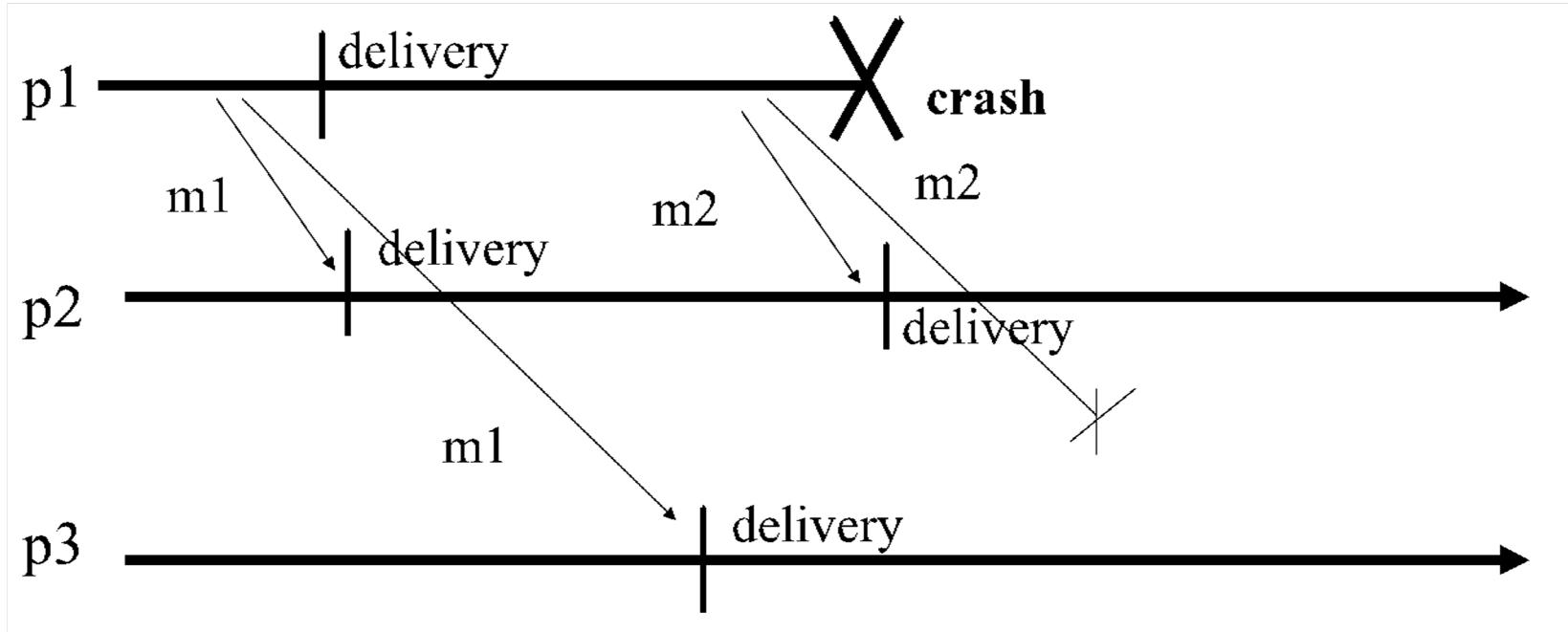
Meets the specification

Reliable Broadcast



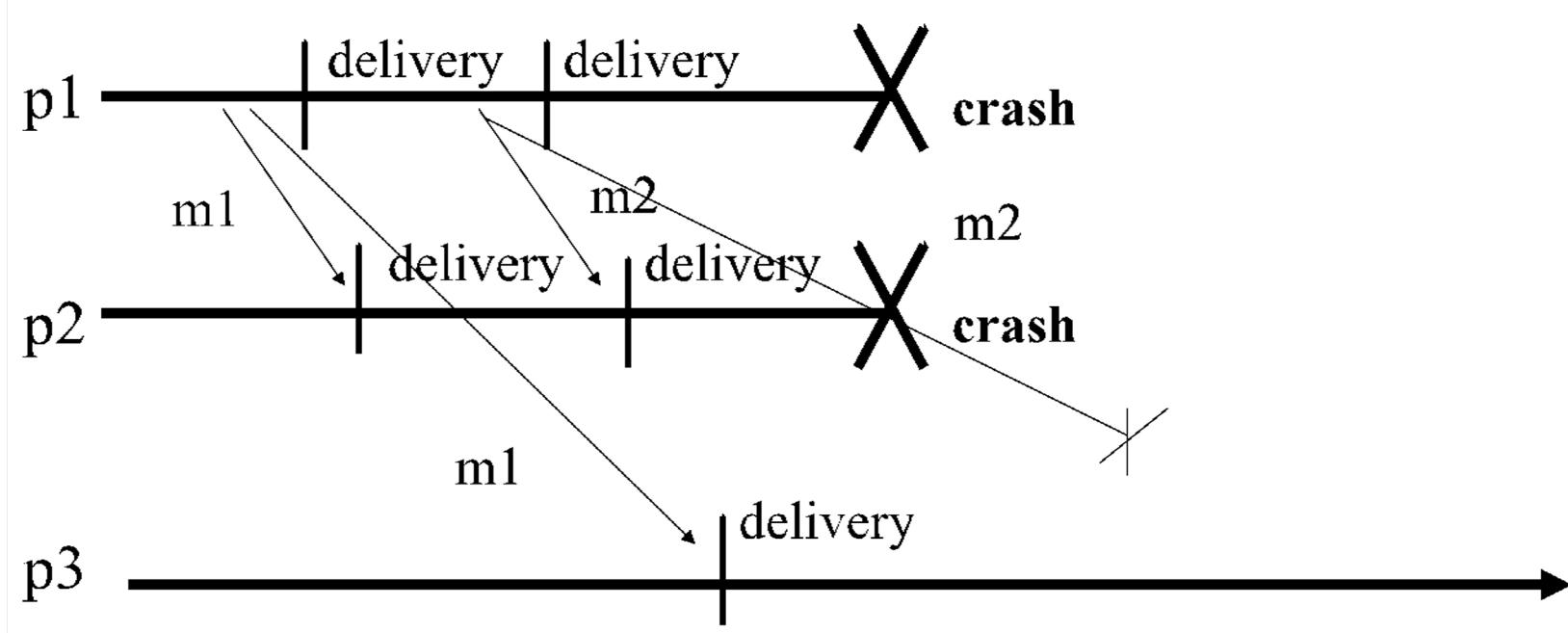
Meets the specification

Reliable Broadcast



Does not meet the specification

Reliable Broadcast



Meets the specification

Overview

We shall consider three forms of reliability for a broadcast primitive

- **(1) Best-effort broadcast**
 - **(2) Reliable broadcast in the crash fault model** ←
 - **(3) Reliable broadcast in the arbitrary fault model**
-
- We provide **specifications** and then **algorithms**

Eager Reliable Broadcast

Implements:

ReliableBroadcast, **instance** *rb*.

Uses:

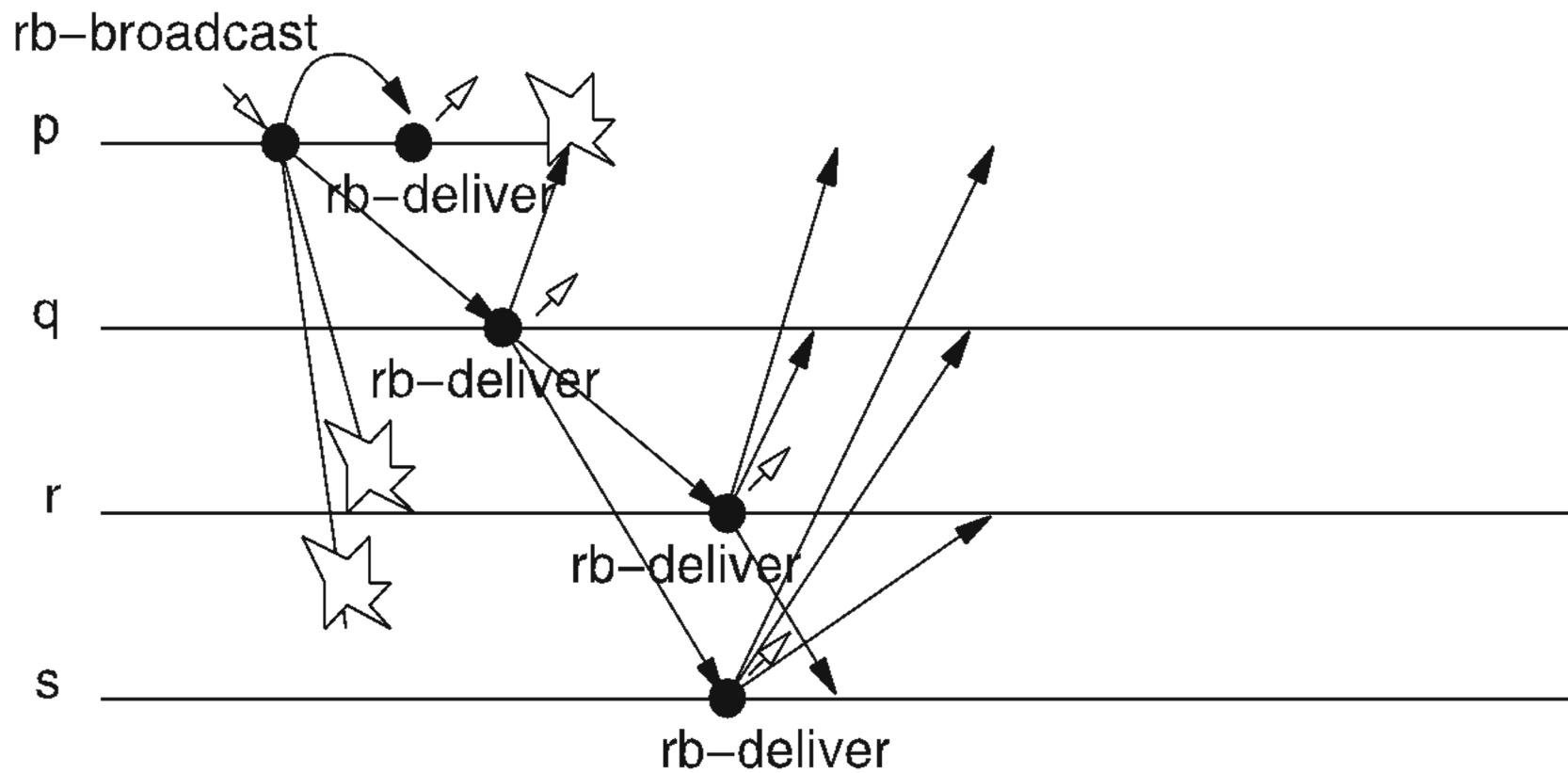
BestEffortBroadcast, **instance** *beb*.

upon event $\langle rb, \text{Init} \rangle$ **do**
delivered := \emptyset ;

upon event $\langle rb, \text{Broadcast} \mid m \rangle$ **do**
trigger $\langle beb, \text{Broadcast} \mid [\text{DATA}, \text{self}, m] \rangle$;

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

Eager Reliable Broadcast



Proof (sketch)

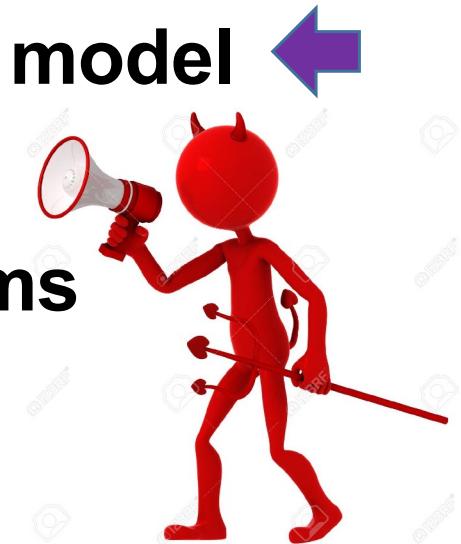
- **RB1. RB2. RB3:** as for the 1st algorithm
- **RB4. Agreement:**
 - every correct process immediately relays every message it *rb-delivers*
 - the *validity* property of the underlying best-effort broadcast primitive ensures that all other correct process will eventually deliver the message.

Note: Does not use a failure detector \Leftrightarrow
Does not make any synchrony assumptions

Overview

We shall consider three forms of reliability for a broadcast primitive

- (1) Best-effort broadcast
 - (2) Reliable broadcast in the crash fault model
 - (3) Reliable broadcast in the arbitrary fault model ←
- We provide **specifications** and then **algorithms**



Reliable broadcast in the byzantine model

- A byzantine process may, e.g.:
 - broadcast messages different from the ones sent by applications,
 - forge messages so that they look as originated by other processes
 - violation of the No creation property
 - send different messages to different processes
 - violation of the *agreement* property

Reliable broadcast in the byzantine model

- Authenticated Perfect Links are a useful tool:
 - disallows forging messages as originated from other processes
 - still does not solve the problems above
- Digital signatures can also be helpful:
 - allow any process to verify the authenticity of a message
 - but faulty processes may sign “illegal” msgs
 - e.g., sending two different messages, after having signed them

Byzantine Consistent and Reliable Broadcasts

- We consider two variants of reliable broadcasts in the byzantine model:
 - Byzantine Consistent Broadcast
 - Byzantine Reliable Broadcast

Byzantine Consistent and Reliable Broadcasts

- Byzantine Consistent Broadcast
 - If the sender s is correct then every correct process should later deliver m .
 - If s is faulty, then every correct process delivers the same message, if it delivers one at all.
 - i.e., correct processes are not guaranteed to deliver the same set of messages if sender is faulty
- Byzantine Reliable Broadcast
 - Ensures that if a correct process delivers m , then every correct process delivers m
 - independently of whether the sender is faulty

Byzantine Consistent Broadcast: specification

- **BCB1: Validity:** If a correct process p broadcasts a msg 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 .

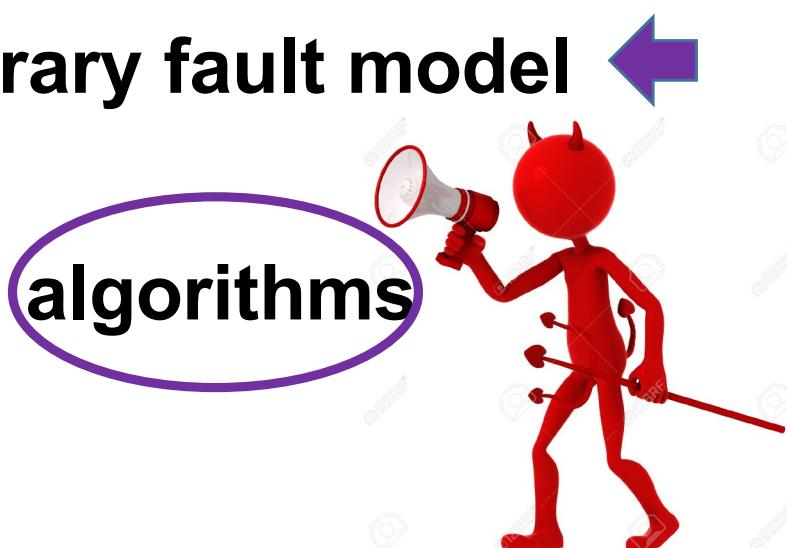
BCB4: Consistency: If some correct process delivers a message m and another correct process delivers a message m' , then $m = m'$.

Note: “Correct” now means non-Byzantine-faulty

Overview

We shall consider three forms of reliability for a broadcast primitive

- (1) Best-effort broadcast
- (2) Reliable broadcast in the crash fault model
- (3) Reliable broadcast in the arbitrary fault model ←
- We provide **specifications** and then **algorithms**

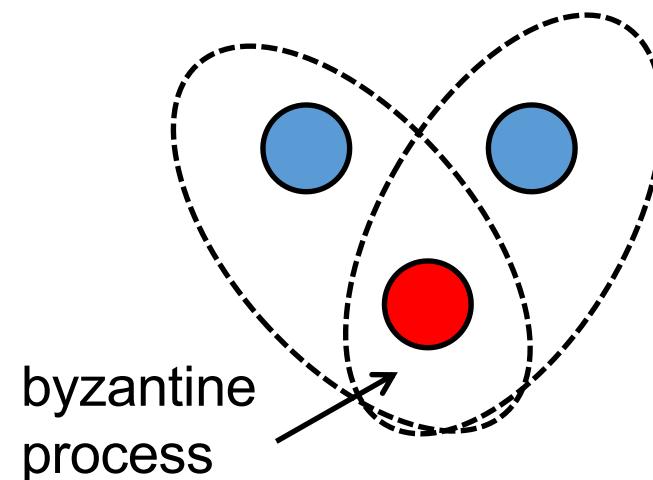


Byzantine Consistent Broadcast: Algorithms

- Byzantine Consistent Broadcast
 - If the sender s is correct then every correct process should later deliver m .
 - If s is faulty, then every correct process delivers the same message, if it delivers one at all.
 - i.e., correct processes are not guaranteed to deliver the same set of messages
- Two algorithms:
 - Authenticated Echo Broadcast
 - exchanges a quadratic number of messages
 - Signed Echo Broadcast
 - linear no. of messages but uses digital signatures (costly)
 - Leverage **Byzantine quorums**

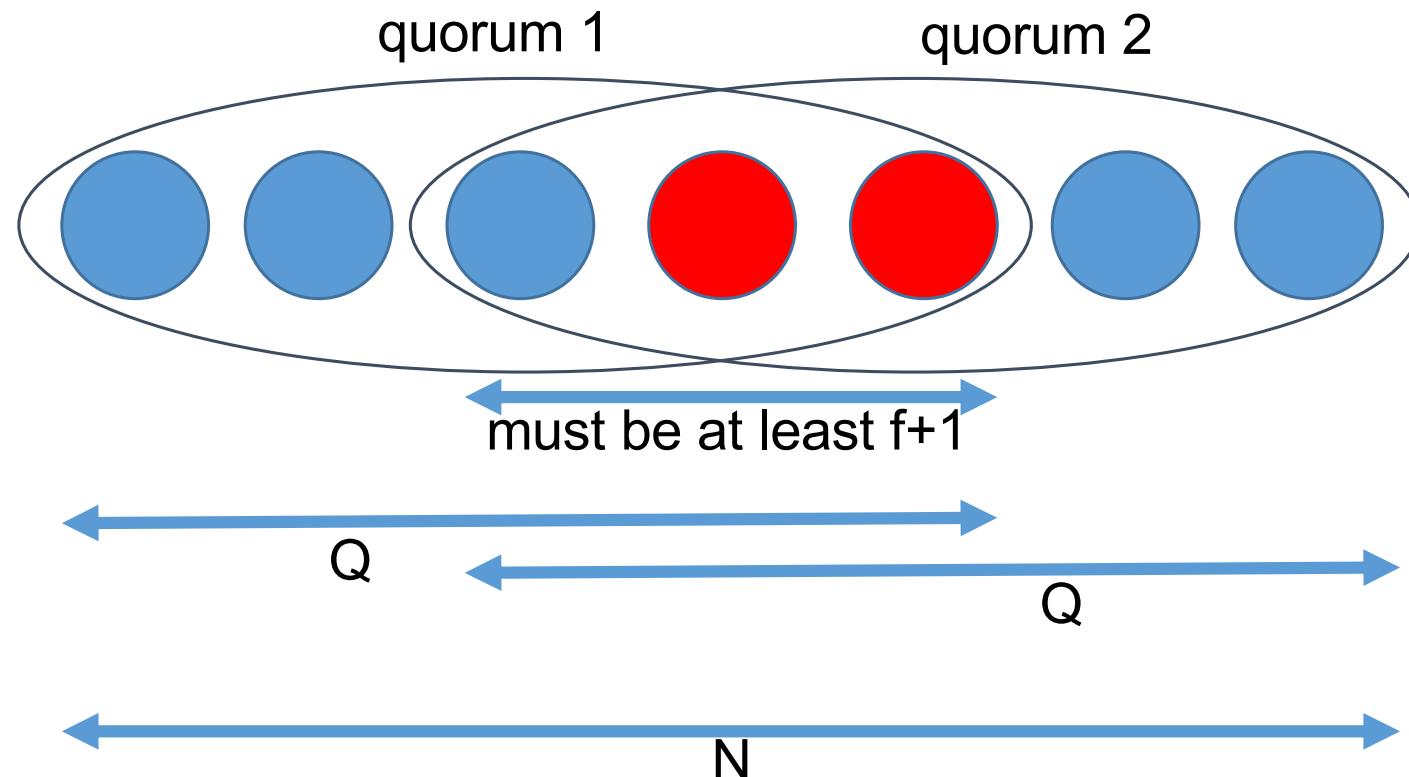
Quorums and Byzantine Quorums

- Quorums are sets of subsets with a specific intersection property:
 - they always overlap in at least one correct process
 - example:
 - majority of processes, assuming $f < N/2$ **crash failure model**
 - intersection of 2 majority quorums is necessarily a correct process with crash faults
 - no longer true with byzantine faults!



Derivation of Byzantine quorum sizes

- Safety: every pair of quorums must intersect in at least one correct process



Derivation of Byzantine quorum sizes

- $Q + Q - N \geq f+1$
- To simplify, we take the optimal size (no need to intersect more than strictly necessary)
- $2Q - N = f+1$

Derivation of Byzantine quorum sizes

- Liveness: a quorum always needs to be available
- Challenge: f processes may never reply (crashed or deliberately silent)
- Thus quorums can't be larger than $N - f$
- $N - f \geq Q$
- Again, we turn this into an equality (don't use more processes than strictly necessary)
- $N - f = Q$

Solving a system of two equations

- Note that f is a system parameter (constant)

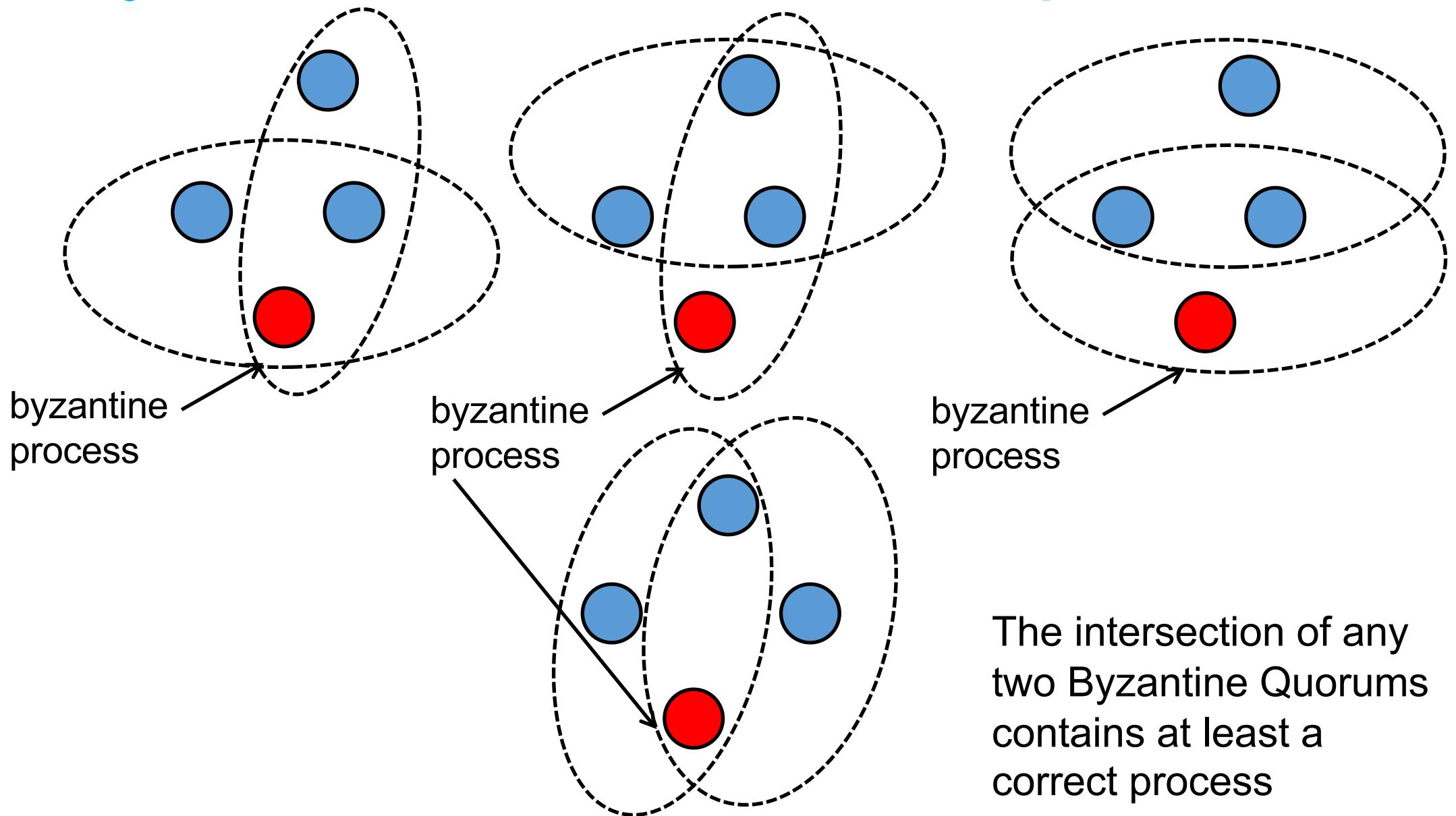
$$\begin{cases} 2Q - N = f+1 \\ N - f = Q \end{cases}$$

- Thus, solving these equations for N and Q yields:

$$N = 3f+1$$

$$Q = 2f+1$$

Byzantine Quorums: examples



Acknowledgements

- Rachid Guerraoui, EPFL

Towards a dependable consensus: Broadcast primitives (cont.) and read/write protocols

Highly dependable systems

Lecture 4

Lecturers: Miguel Matos and Paolo Romano

Byzantine Consistent Broadcast: specification

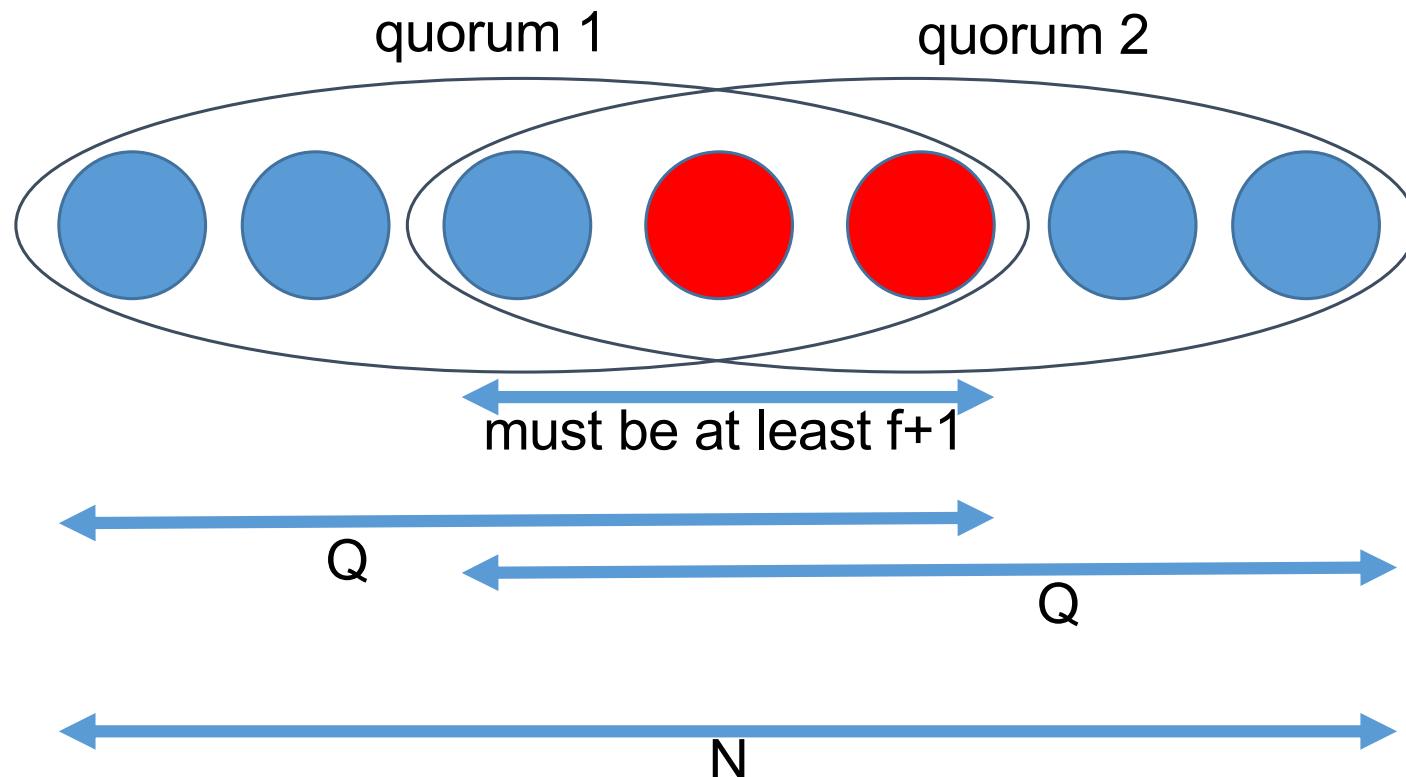
- **BCB1: Validity:** If a correct process p broadcasts a msg m , then every correct process eventually delivers m .
- **BCB2: No duplication:** Every correct process delivers at most one message.
- **BCB3: Integrity:** If some correct process delivers a message m with sender p and process p is correct, 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 $m = m'$.

Note: “Correct” now means non-Byzantine-faulty

Derivation of Byzantine quorum sizes

- Safety: every pair of quorums must intersect in at least one correct process



Solving a system of two equations

- Note that f is a system parameter (constant)

$$\begin{cases} 2Q - N = f+1 \\ N - f = Q \end{cases}$$

- Thus, solving these equations for N and Q yields:

$$N = 3f+1$$

$$Q = 2f+1$$

- Generically, for $N > 3f$, we have $Q = \lceil (N+f)/2 \rceil$

Authenticated Echo Broadcast

- exploits *Byzantine quorums* (uses N,Q derived before)
 - two rounds of message exchanges.
 - 1st round: sender disseminates msg to all processes.
 - 2nd round:
 - every process acts as a witness for m
 - resends m in an ECHO message to all others.
 - deliver only when received more than a quorum (Q) of ECHOs
 - Byzantine processes cannot cause a correct process to *bcb-deliver* a message $m' \neq m$.
- Relies on authenticated links

Authenticated Echo Broadcast

- **Implements:**
ByzantineConsistentBroadcast, with sender s .
- **Uses:**
AuthPerfectPointToPointLinks, **instance** al .
- **upon event** $\langle bcb, \text{Init} \rangle$ **do**
 $\text{sentecho} := \text{FALSE};$
 $\text{delivered} := \text{FALSE};$
 $\text{echos} := [\perp]^N;$

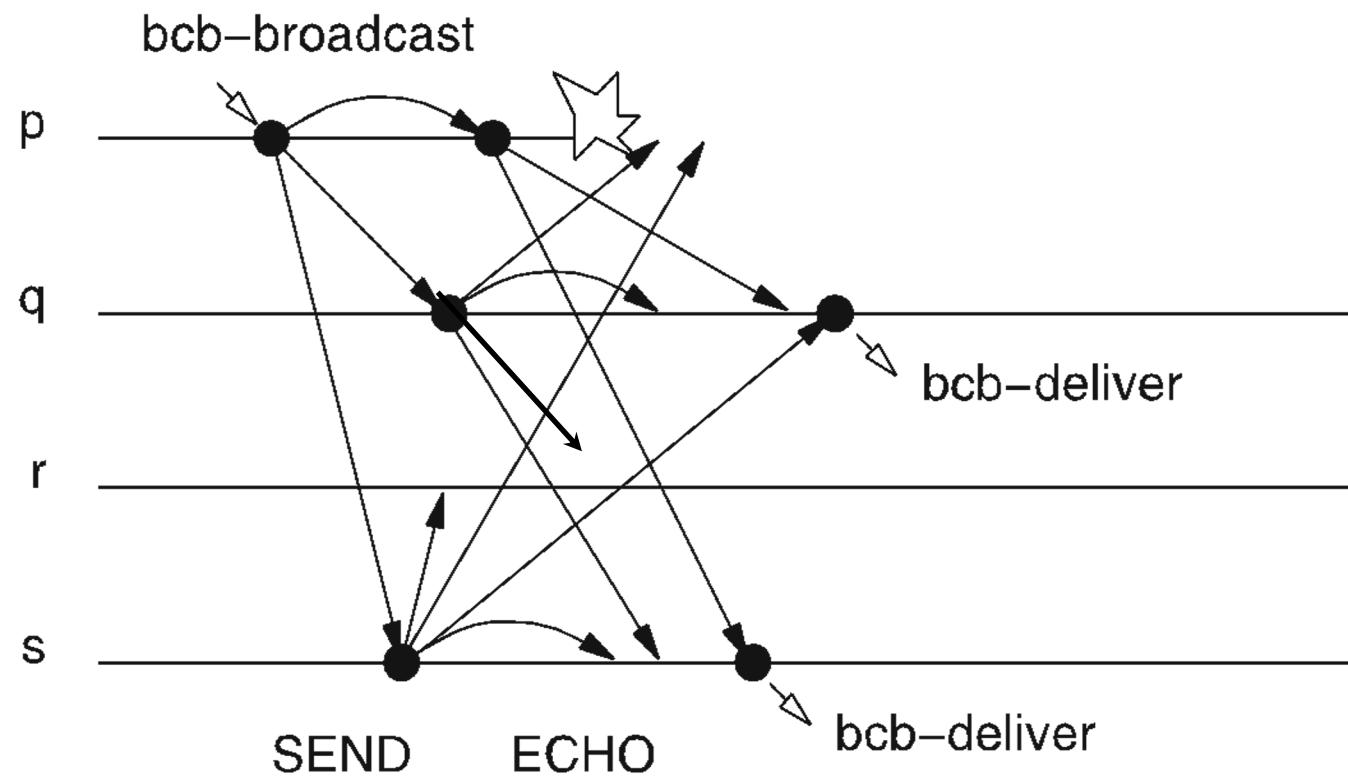
Authenticated Echo Broadcast

- **upon event** $\langle bcb, \text{Broadcast} \mid m \rangle$ **do** // only process s
forall $q \in \Pi$ **do**
 trigger $\langle al, \text{Send} \mid q, [\text{SEND}, m] \rangle$;
- **upon event** $\langle al, \text{Deliver} \mid p, [\text{SEND}, m] \rangle$
 such that $p = s$ **and** $\text{sentecho} = \text{FALSE}$ **do**
 $\text{sentecho} := \text{TRUE}$;
 forall $q \in \Pi$ **do**
 trigger $\langle al, \text{Send} \mid q, [\text{ECHO}, m] \rangle$;

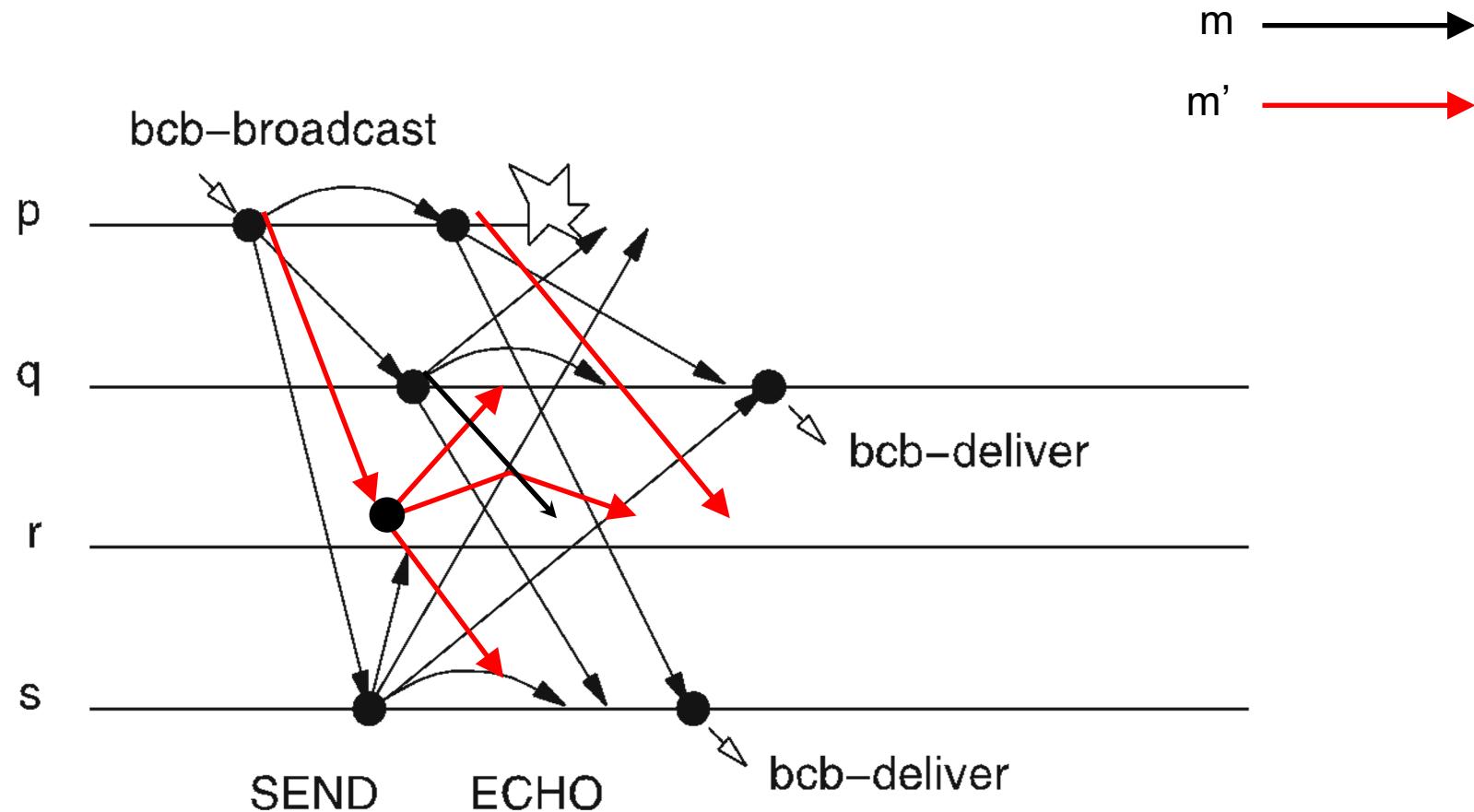
Authenticated Echo Broadcast

- **upon event** $\langle al, \text{Deliver} \mid p, [\text{ECHO}, m] \rangle$ **do**
if $\text{echos}[p] = \perp$ **then**
 $\text{echos}[p] := m;$
- **upon exists** $m \neq \perp$ **such that** :
 $\#\{p \in \Pi \mid \text{echos}[p] = m\} > (N+f)/2$ **and** $\text{delivered} =$
 FALSE do
 $\text{delivered} := \text{TRUE};$
 trigger $\langle bcb, \text{Deliver} \mid s, m \rangle;$

Authenticated Echo Broadcast



Authenticated Echo Broadcast



r can never deliver m' , if q and s deliver m

r should get 3 ECHOES, but can get at most 2 (including its own)

Correctness

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

Proof (sketch)

- if the sender is correct, then every correct process a/l -sends an ECHO message
- every correct process a/l -delivers at least $N - f$ of them.
- $N - f > (N + f)/2$ under the assumption that $N > 3f$
implies that every correct process also bcb -delivers the message m contained in the ECHO messages

Correctness

- **BCB2: No duplication:** Every correct process delivers at most one message.
- **Proof (sketch):** enforced by the *delivered* variable

Correctness

- **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 .
- **Proof (sketch):** by the Authenticated Perfect Link's properties

Correctness

- **BCB4: Consistency:** If some correct process delivers a message m and another correct process delivers a message m' , then $m = m'$.
- For a correct process p to bcb -deliver some m , it needs to receive ECHO messages for m from a Byzantine quorum.

Correctness

- Two Byzantine quorums overlap in at least one correct process.
- Assume, by contradiction, that a different correct process p' *bcb*-delivers some message $m' \neq m$.
 - p' has received a Byzantine quorum of ECHO messages for m'
 - the correct process in the intersection of the two Byzantine quorums sent different ECHO messages to p and to p'

Contradiction found!

Signed echo broadcast

- Exploits digital signatures:
 - more powerful than MACs
 - allow a third process to verify the authenticity of a message sent from a 1st process s to a 2nd process r
 - avoid quadratic number of message exchanges of prev. algorithm

Signed echo broadcast

- High level idea:
 - Witnesses authenticate a request not by sending an ECHO message to all processes
 - but by signing a statement which they return to the sender
 - Sender collects a Byzantine quorum of these signed statements and relays them in a third communication step to all processes.

Signed Echo Broadcast

- **Implements:**
ByzantineConsistentBroadcast, **instance** *bcb*.
- **Uses:**
AuthPerfectPointToPointLinks, **instance** *al*.
- **upon event** < *bcb*, Init > **do**
sentecho := FALSE;
sentfinal := FALSE;
delivered := FALSE;
echos := [\perp]^N ; Σ := [\perp]^N ;

Signed Echo Broadcast

- **upon event** $\langle bcb, \text{Broadcast} \mid m \rangle$ **do** // only process s
forall $q \in \Pi$ **do**
 trigger $\langle al, \text{Send} \mid q, [\text{SEND}, m] \rangle$;
- **upon event** $\langle al, \text{Deliver} \mid p, [\text{SEND}, m] \rangle$
 such that $p = s$ **and** $\text{sentecho} = \text{FALSE}$ **do**
 $\text{sentecho} := \text{TRUE};$
 $\sigma = \text{sign}(self, bcb \parallel self \parallel \text{ECHO} \parallel m);$
 trigger $\langle al, \text{Send} \mid p, [\text{ECHO}, m], \sigma \rangle$;

algorithm instance identifier bcb is included in the input argument to the digital signature (else, a Byzantine process might reuse a signature issued by a correct process in a different context)

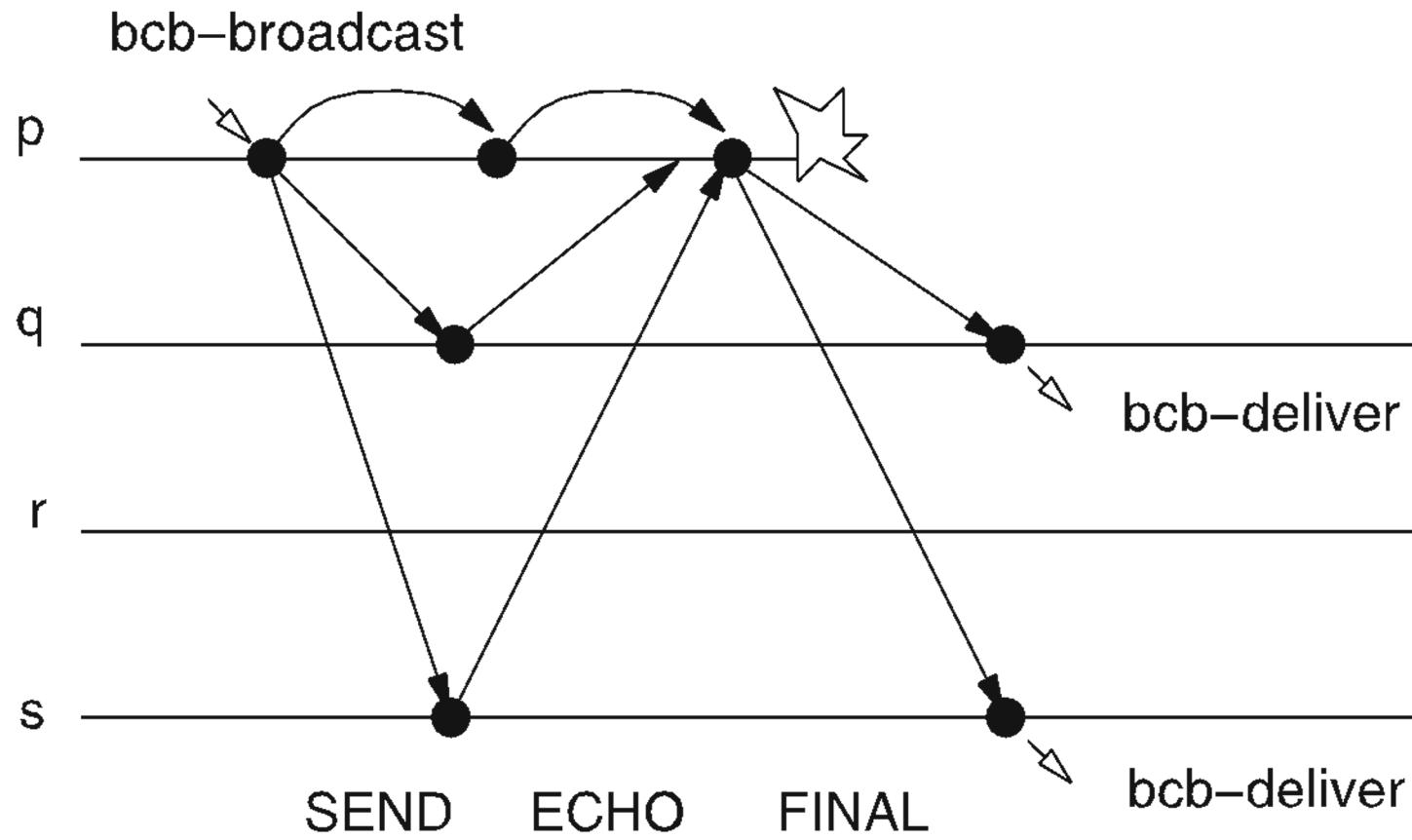
Signed Echo Broadcast

- *// only process s*
upon event $\langle al, \text{Deliver} \mid p, [\text{ECHO}, m, \sigma] \rangle$ **do**
 if $\text{echos}[p] = \perp \wedge \text{verifysig}(p, bcb \parallel p \parallel \text{ECHO} \parallel m, \sigma)$
 then $\text{echos}[p] := m; \Sigma[p] := \sigma;$
- **upon exists** $m \neq \perp$ **such that :**
 $\#\{p \in \Pi \mid \text{echos}[p] = m\} > (N+f)/2$ **and** $\text{sentfinal} =$
 FALSE **do**
 $\text{sentfinal} := \text{TRUE};$
 forall $q \in \Pi$ **do**
 trigger $\langle al, \text{Send} \mid q, [\text{FINAL}, m, \Sigma] \rangle;$

Signed Echo Broadcast

- **upon event** $\langle al, \text{Deliver} \mid p, [\text{FINAL}, m, \Sigma] \rangle$ **do**
if ($delivered = \text{FALSE}$) **and**
 ($\#\{p \in \Pi \mid \Sigma[p] \neq \perp \wedge$
 $\text{verifysig}(p, bcb \parallel p \parallel \text{ECHO} \parallel m, \Sigma[p])\} > (N+f)/2$)
 $delivered := \text{TRUE};$
trigger $\langle bcb, \text{Deliver} \mid s, m \rangle;$

Signed Echo Broadcast



Correctness

- **BCB1-BCB3:** as in prev. algorithm
- **BCB4: Consistency:** If some correct process delivers a message m and another correct process delivers a message m' , then $m = m'$.
 - The digitally signed echo messages convey the same information as the ECHO message that a process sends directly to all others in the prev. algorithm

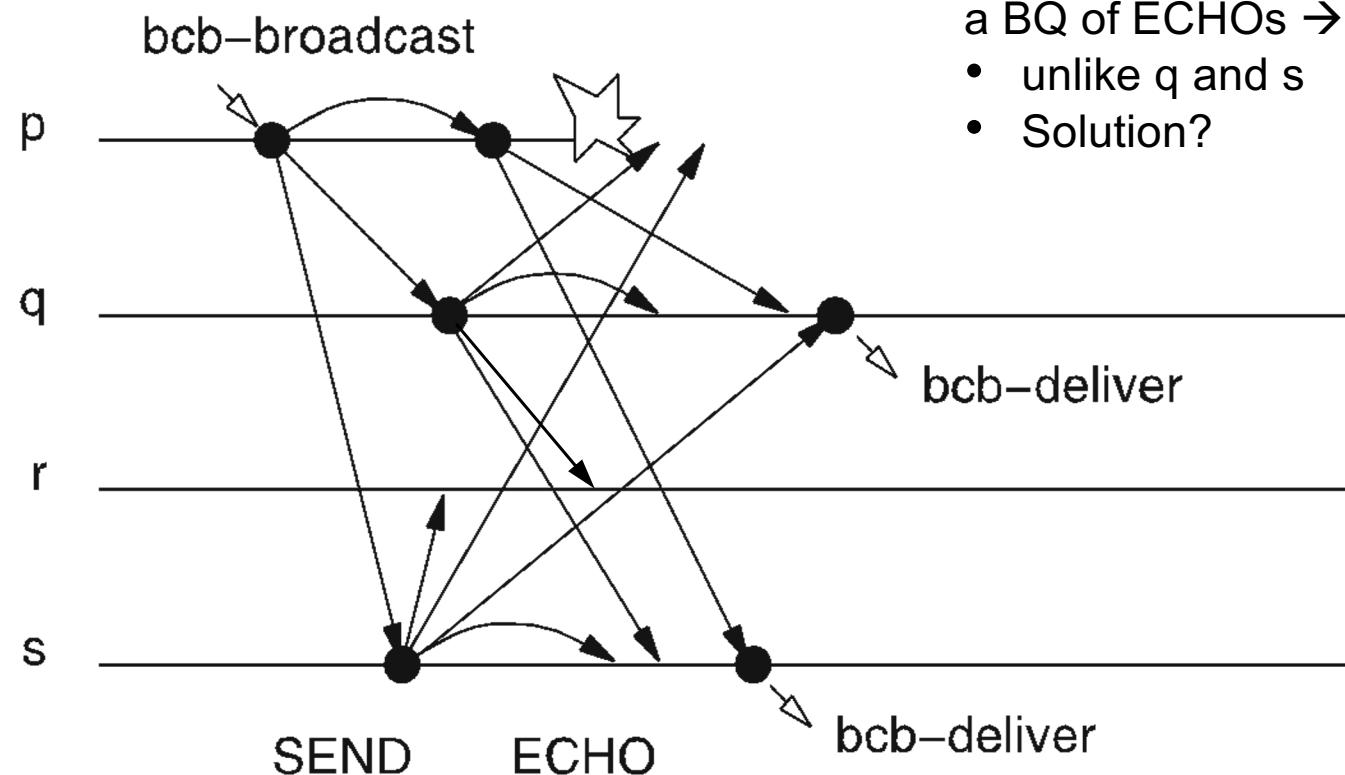
Byzantine Reliable Broadcast: specification

- **BRB1–BRB4** : same as in BCB1-BCB4
- **BRB5:** *Totality*: If some message is delivered by any correct process, every correct process eventually delivers a message.
- **Note**
- Both BCB and BRB are defined for a designated sender, a given message
 - not for a stream of messages, unlike in crash model
- *Totality + Consistency* are equivalent to *Agreement* of Reliable Broadcast in the crash model

Byzantine Reliable Broadcast

- Specification of Byzantine Reliable Broadcast
 - Ensures that if a correct process delivers m , then every correct process delivers m
 - **independently of whether the sender is faulty**
 - Stronger spec than Byzantine Consistent Broadcast
- Algorithm: Authenticated Double-Echo Broadcast
- Extends Authenticated Echo Broadcast. How?

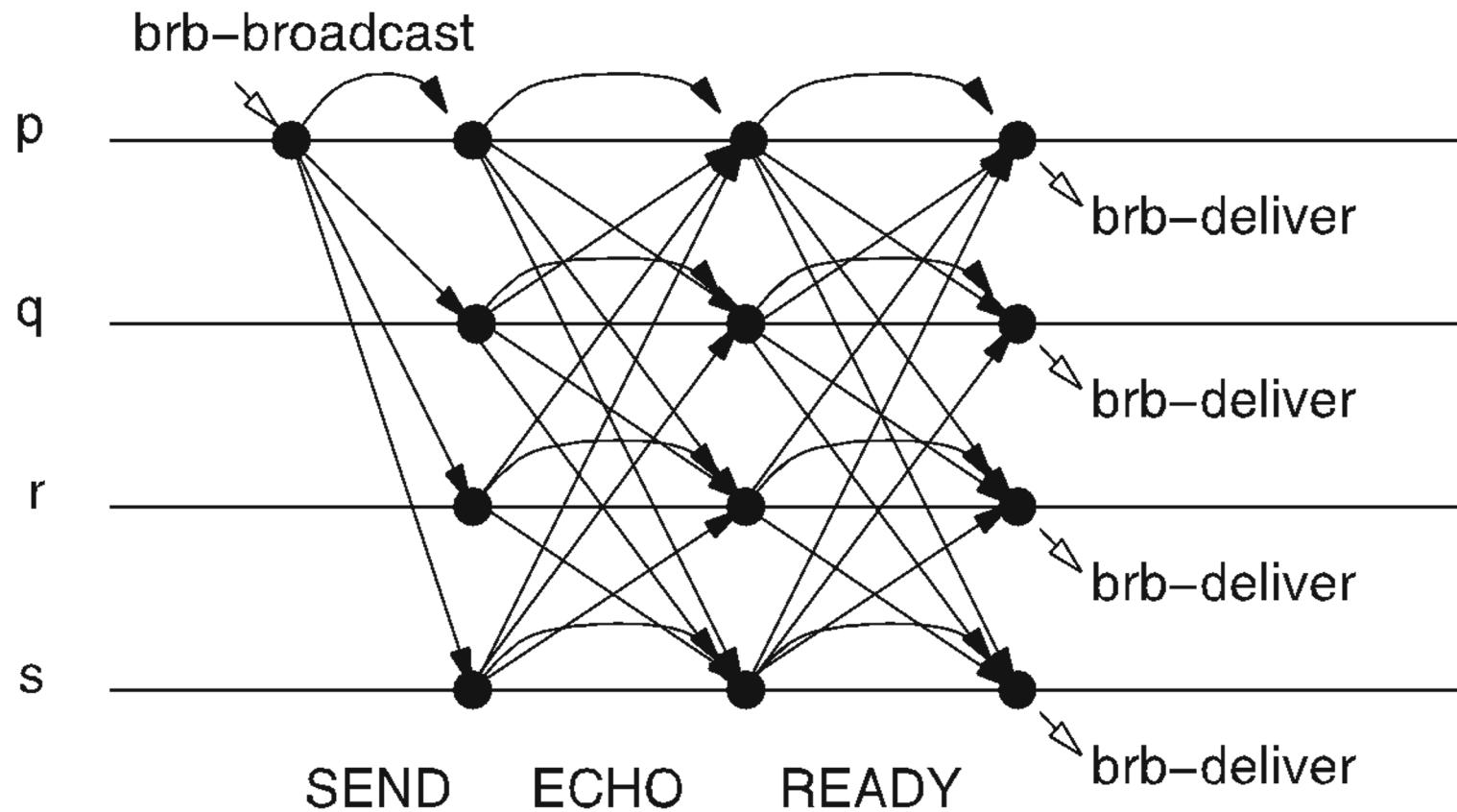
Authenticated Echo Broadcast



PROBLEM:

- process r never gets to receive a BQ of ECHOes → does not deliver m
- unlike q and s
- Solution?

Authenticated Double-Echo Broadcast



Authenticated Double Echo Broadcast

- **Implements:**

- ByzantineReliableBroadcast, with sender s .

- **Uses:**

- AuthPerfectPointToPointLinks, **instance** al .

- **upon event** $\langle brb, \text{Init} \rangle$ **do**

- $sentecho := \text{FALSE};$

- $sentready := \text{FALSE};$

- $delivered := \text{FALSE};$

- $echos := [\perp]^N;$

- $readys := [\perp]^N;$

Authenticated Double Echo Broadcast

- **upon event** $\langle brb, \text{Broadcast} \mid m \rangle$ **do** // only process s
forall $q \in \Pi$ **do**
 trigger $\langle al, \text{Send} \mid q, [\text{SEND}, m] \rangle$;
- **upon event** $\langle al, \text{Deliver} \mid p, [\text{SEND}, m] \rangle$
 such that $p = s$ and $\text{sentecho} = \text{FALSE}$ **do**
 $\text{sentecho} := \text{TRUE}$;
forall $q \in \Pi$ **do**
 trigger $\langle al, \text{Send} \mid q, [\text{ECHO}, m] \rangle$;

Authenticated Double Echo Broadcast

- **upon event** $\langle al, \text{Deliver} | p, [\text{ECHO}, m] \rangle$ **do**
 if $\text{echos}[p] = \perp$ **then** $\text{echos}[p] := m;$
- **upon exists** $m \neq \perp$ **such that :**
 $|\{p \in \Pi \mid \text{echos}[p] = m\}| > (N+f)/2$ **and** $\text{sentready}=\text{FALSE}$ **do**
 $\text{sentready}:= \text{TRUE}$
 forall $q \in \Pi$ **do**
 trigger $\langle al, \text{Send} | q, [\text{READY}, m] \rangle;$
- **upon event** $\langle al, \text{Deliver} | p, [\text{READY}, m] \rangle$ **do**
 if $\text{readys}[p] = \perp$ **then** $\text{readys}[p] := m;$

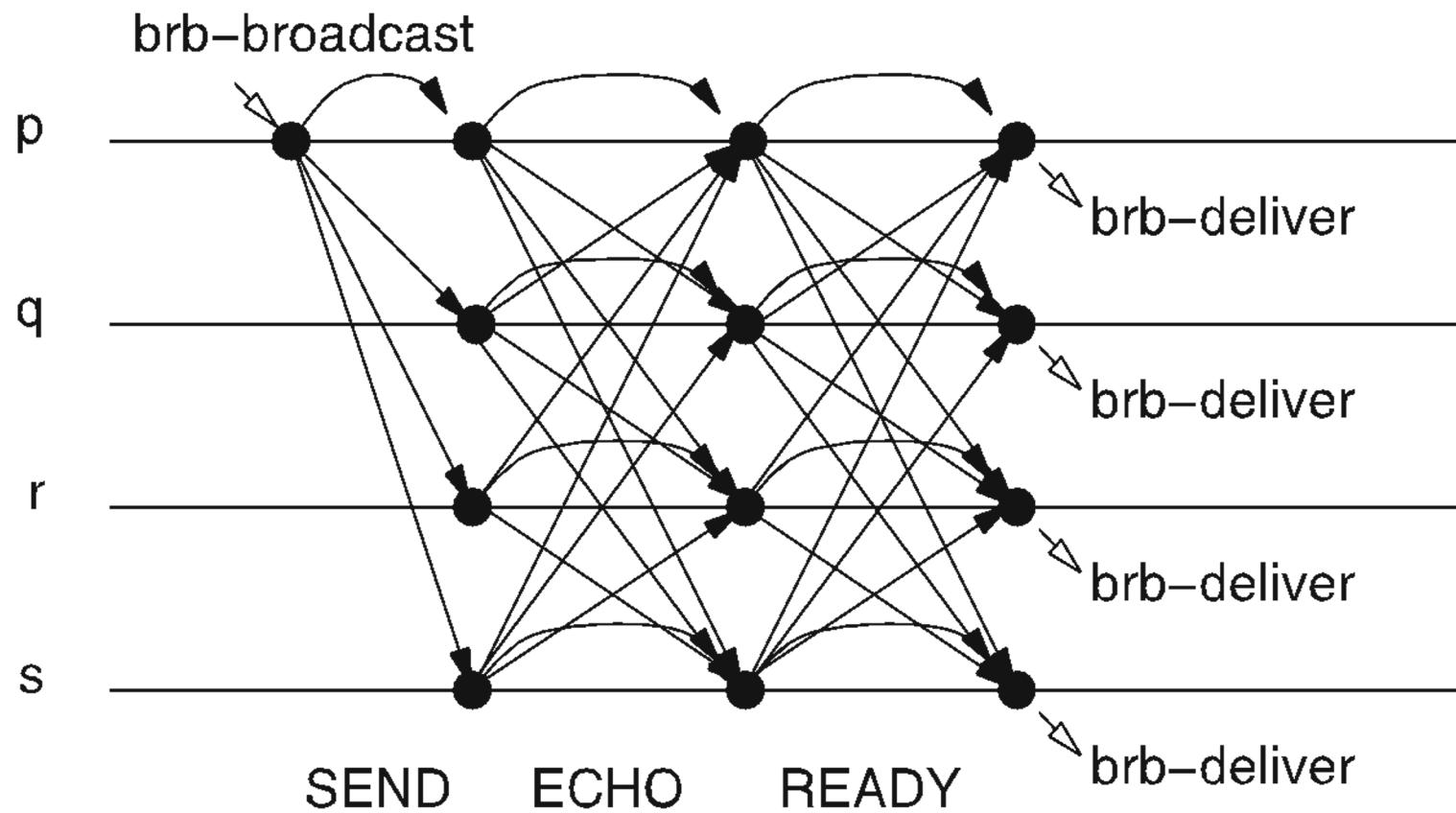
Retransmission step:
inform other correct
processes about
existence of byzantine
quorum for m

Authenticated Double Echo Broadcast

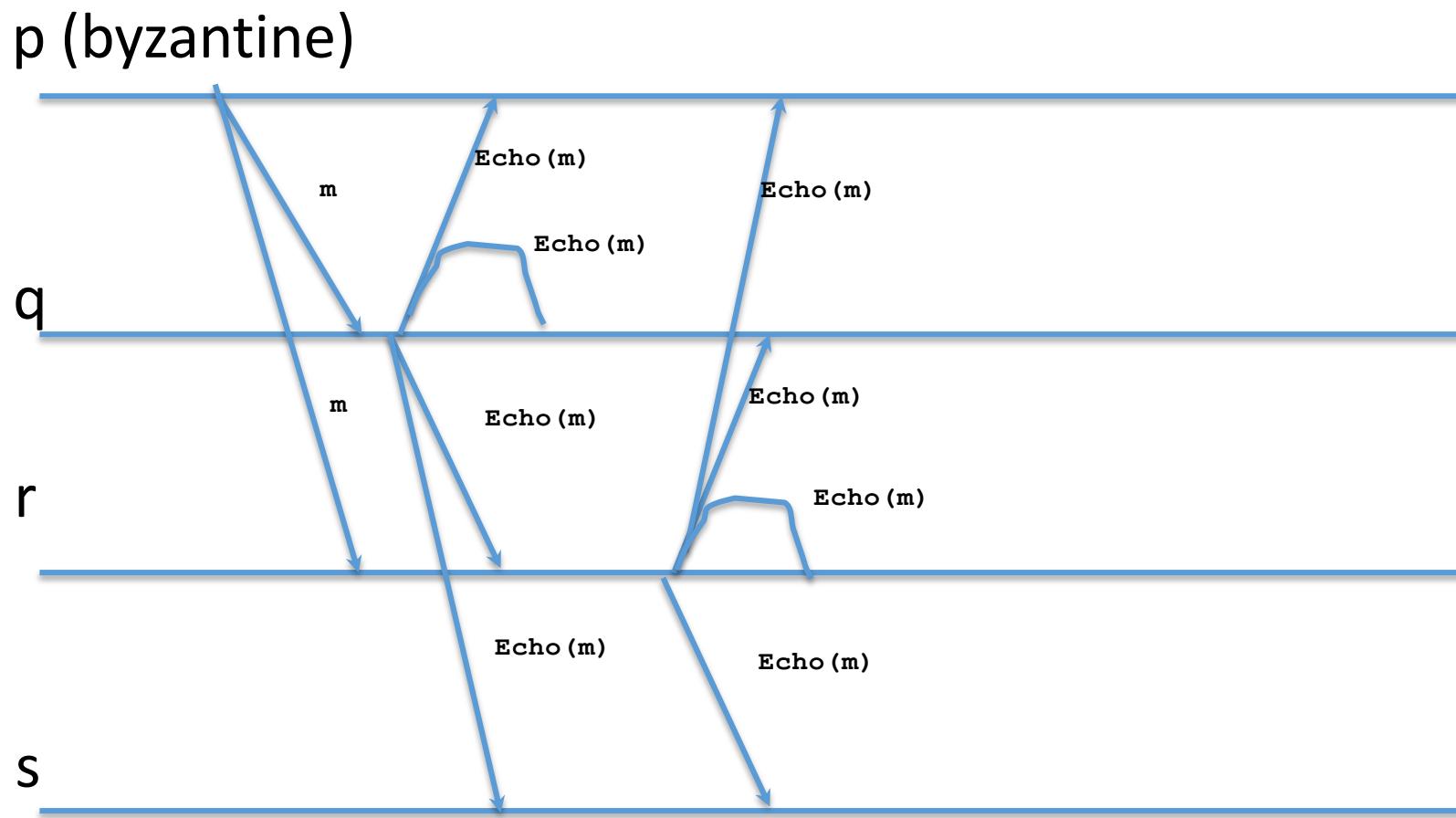
Amplification step:
Why is it needed?

- **upon exists** $m \neq \perp$ such that
 $|\{p \in \Pi | \text{readys}[p] = m\}| > f$ **and** $\text{sentready} = \text{FALSE}$ **do**
 $\text{sentready} := \text{TRUE};$
forall $q \in \Pi$ **do**
trigger $\langle al, \text{Send} | q, [\text{READY}, m] \rangle;$
- **upon exists** $m \neq \perp$ **such that**
 $|\{p \in \Pi | \text{readys}[p] = m\}| > 2f$ **and** $\text{delivered} = \text{FALSE}$ **do**
 $\text{delivered} := \text{TRUE};$
trigger $\langle brb, \text{Deliver} | s, m \rangle;$

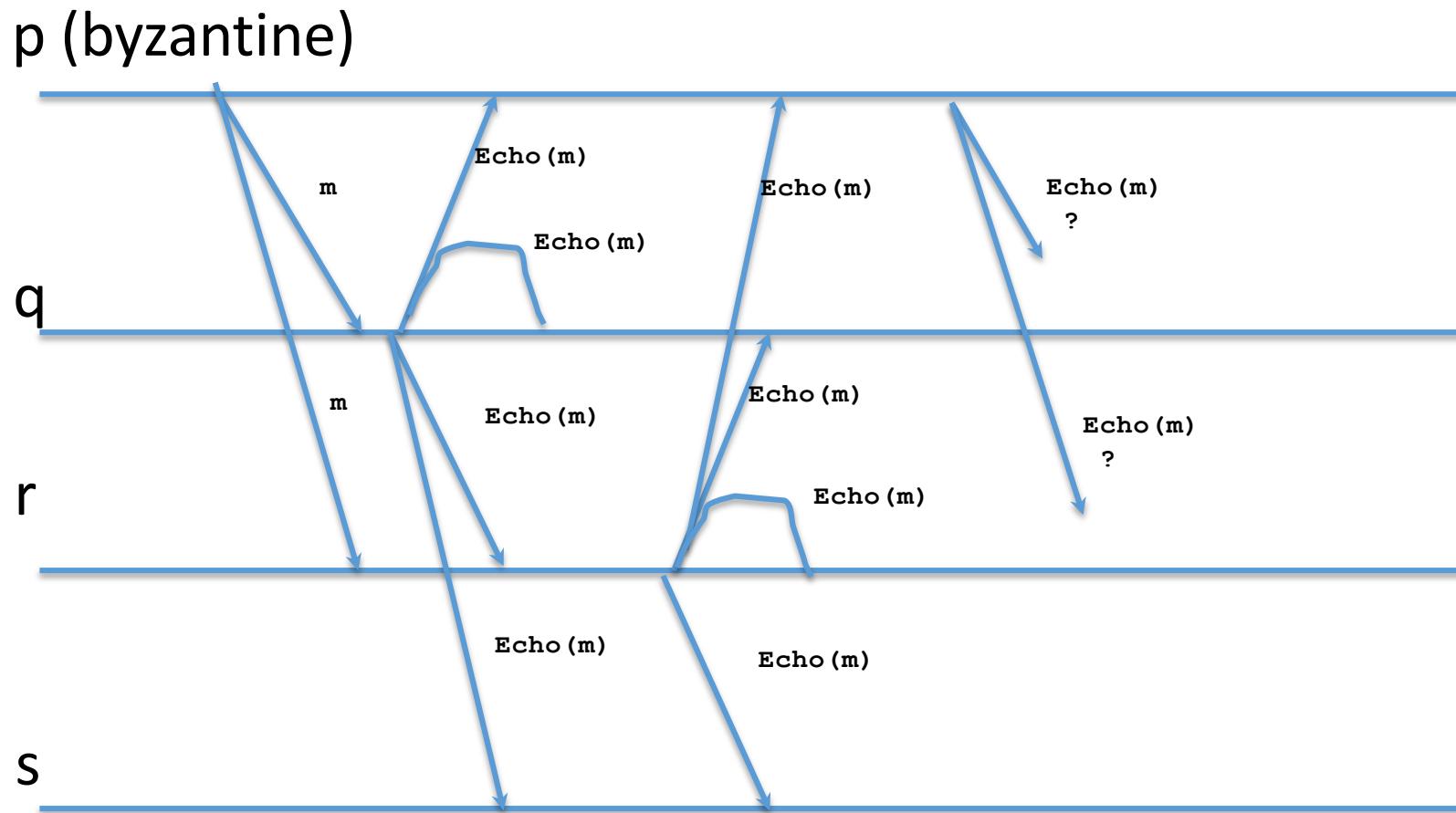
Authenticated Double-Echo Broadcast



Is it possible that only q or only r send a ready msg?

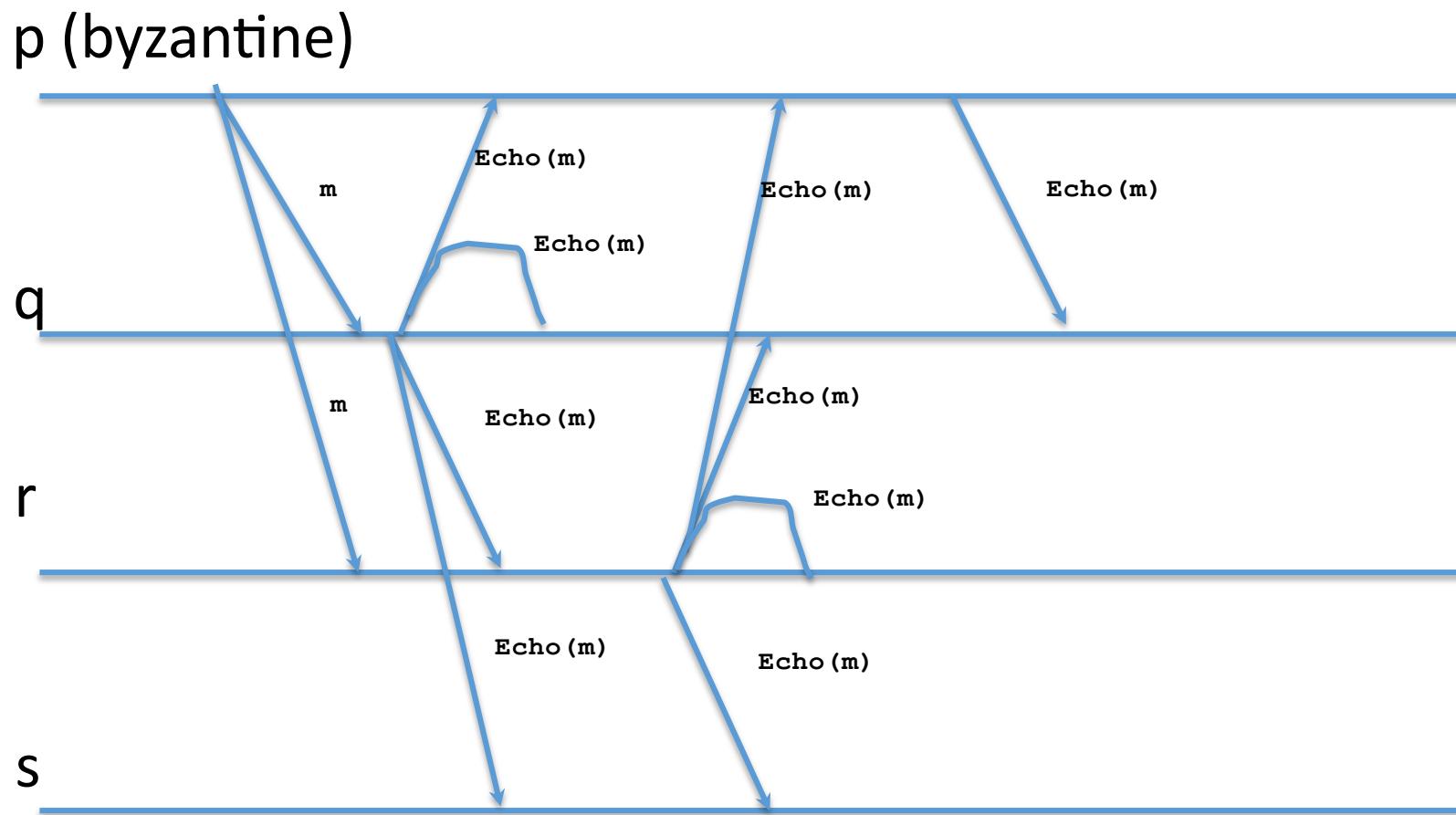


Is it possible that only q or r send a ready msg?



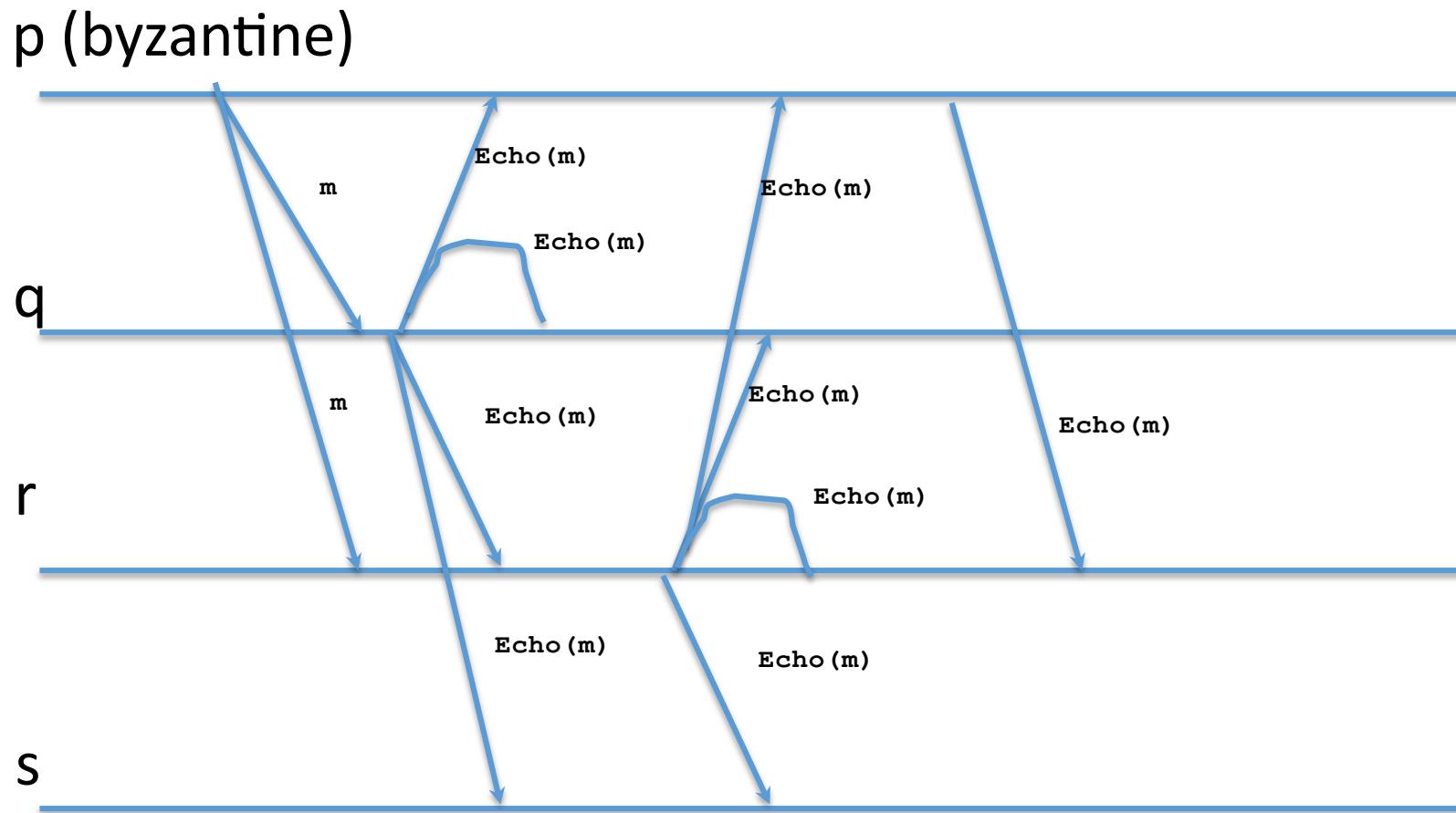
Yes, depending on whether p sends an echo to q and/or r

Is it possible that only q or r send a ready msg?



Yes, depending on whether p sends an echo to q and/or r

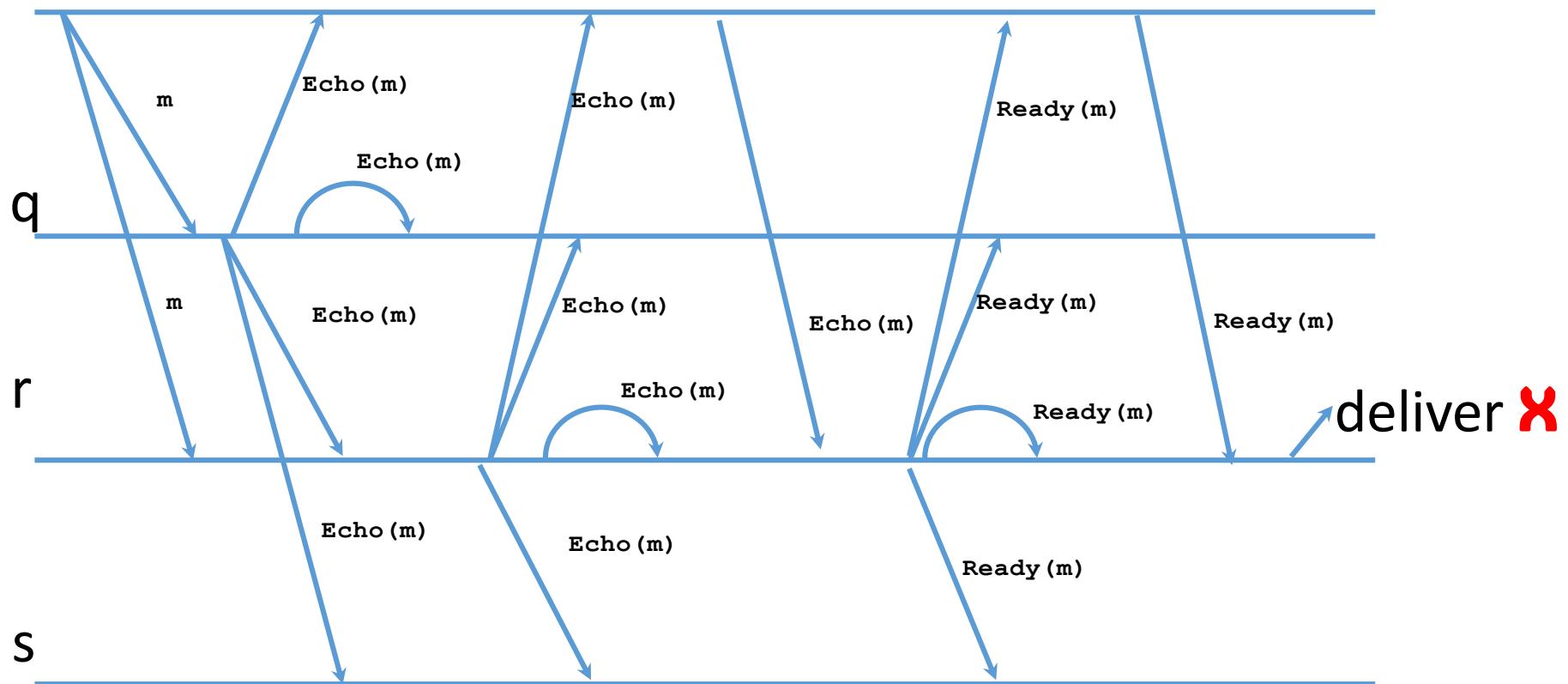
Is it possible that only q or r send a ready msg?



Yes, depending on whether p sends an echo to q and/or r

Is it ok to deliver after $f+1$ Ready msgs?

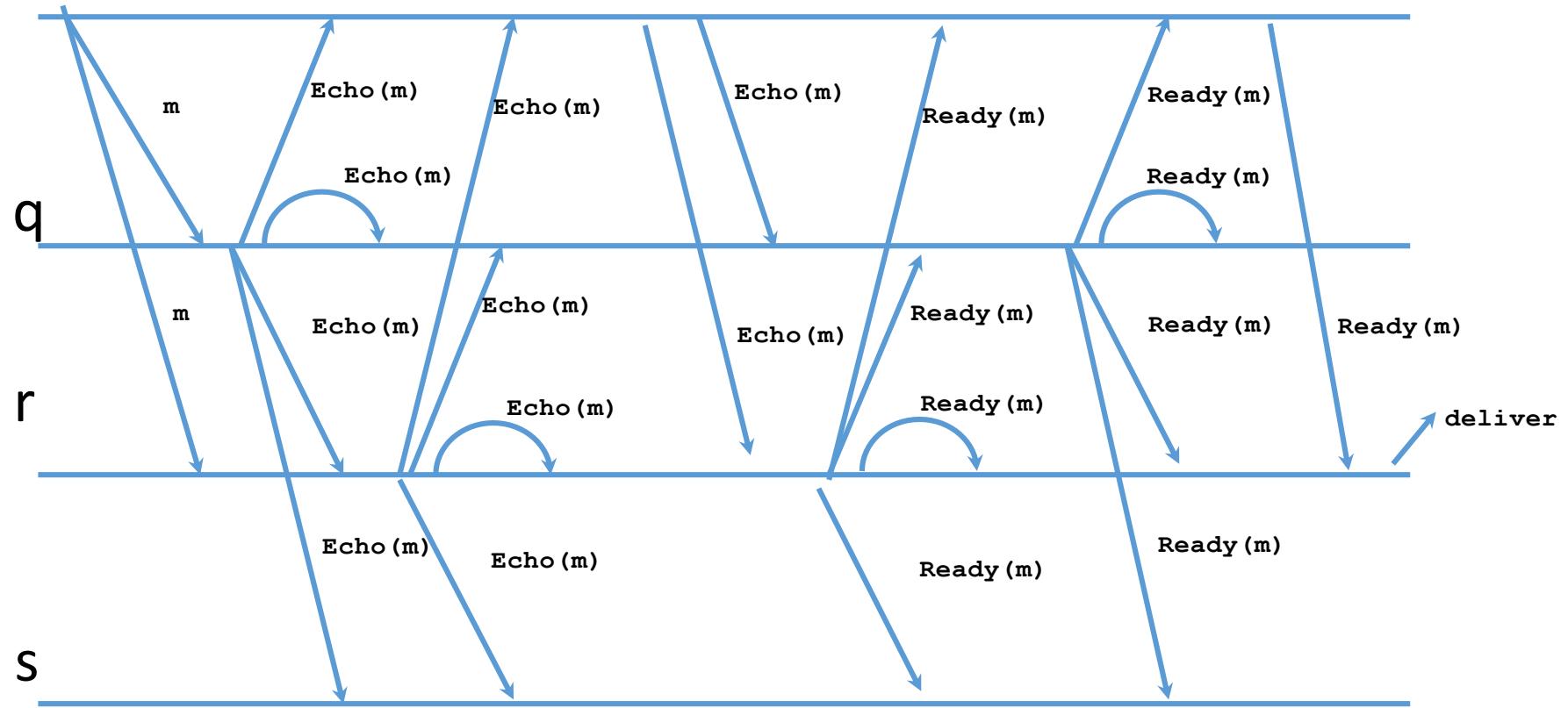
p (byzantine)



Need to wait for $2f+1$ Ready msgs to secure totality

Why is the amplification phase needed?

p (byzantine)



- r receives 3 Ready(m) msgs and delivers m ...
- but q does not, as p does not send Ready(m) to q

Correctness (1/3)

- Validity, No duplication, Integrity:
 - same as in Authenticated Echo Broadcast

Correctness (2/3)

- Consistency: If some correct process delivers a message m and another correct process delivers a message m' , then $m = m'$.
 - if some correct process broadcasts ready messages for m and another correct process broadcasts ready for m' , then $m=m'$
 - Proof by contradiction, same as in previous proof
 - Directly implies consistency because cannot have more than f ready messages for different messages

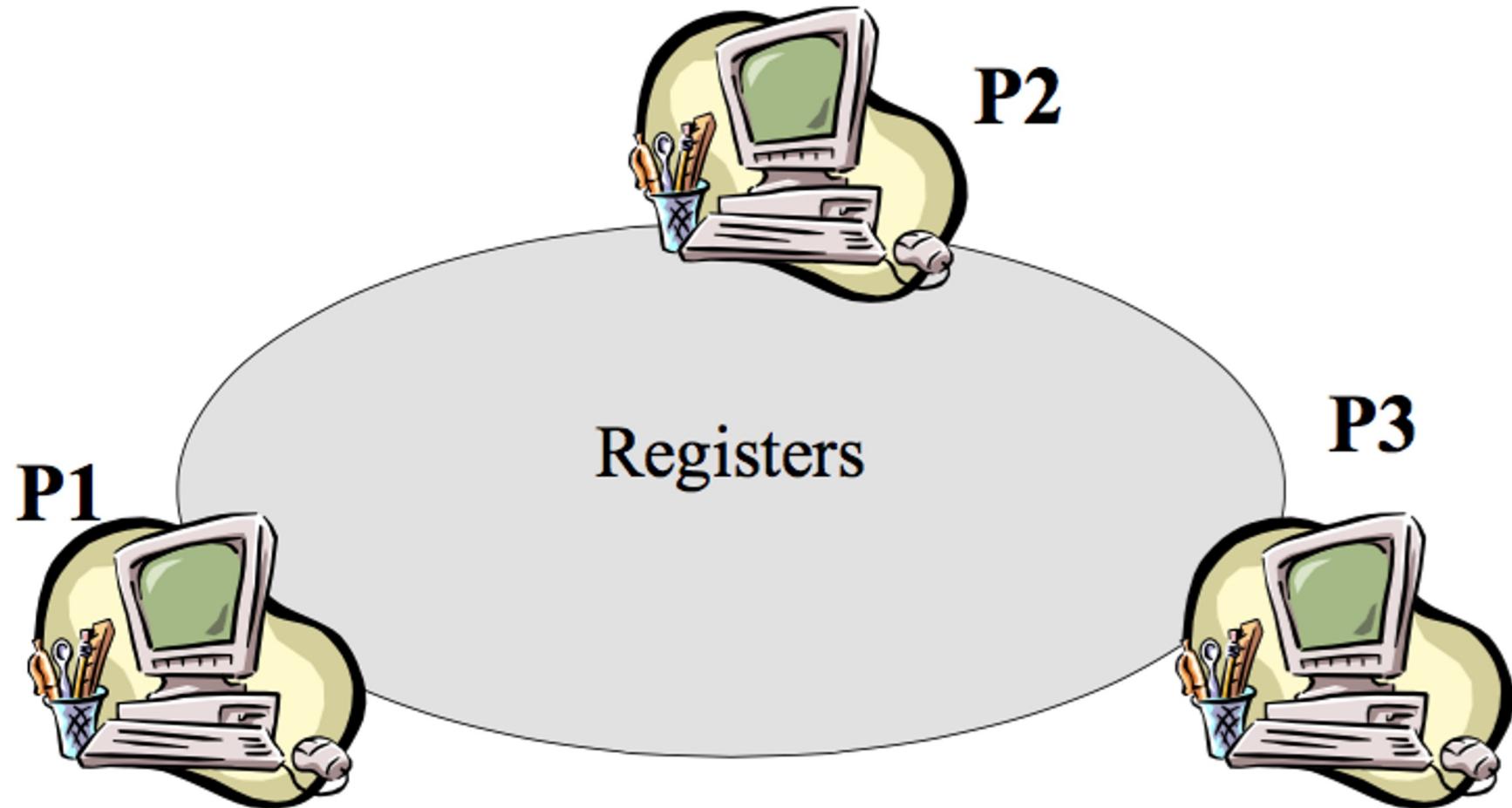
Correctness (3/3)

- Totality:
 - if a correct process delivers, then it received at least $2f+1$ ready messages, at least $f+1$ from correct process.
 - These $f+1$ will eventually be delivered to all correct processes, triggering amplification step, and consequently sufficient ready messages for totality

Byzantine broadcast channels

- Byzantine broadcast works only for a single message
- It is simple to implement a broadcast channel
 - Either consistent or reliable broadcast channels
- Idea: use multiple instances of Byzantine broadcast

New abstraction: read/write registers



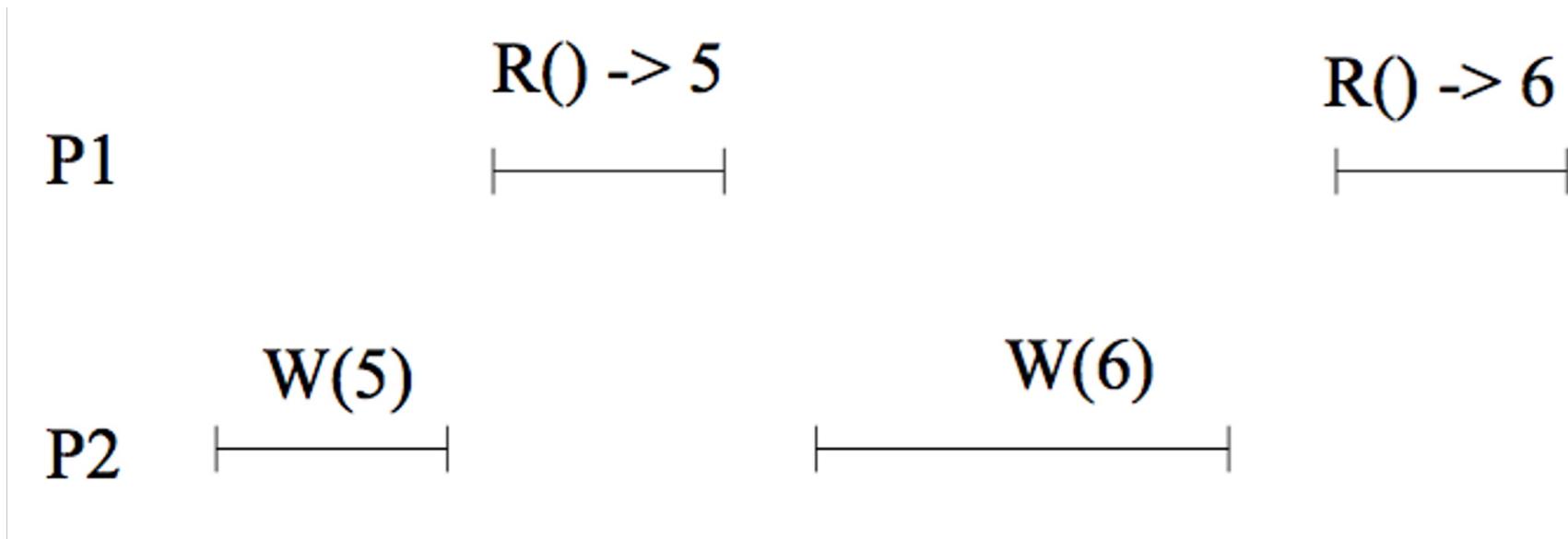
P1,P2,P3 are both clients and replicas (without loss of generality)

Assumptions

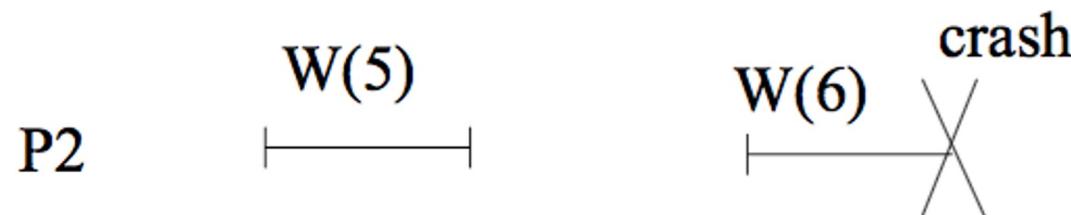
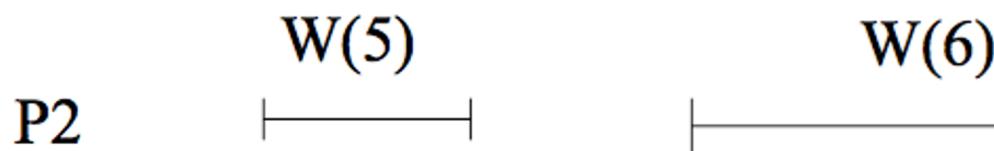
- Single register, shared by all processes
- Interface:
 - `Read()` → returns `value`
 - `Write(value)` → returns "ack"
- Values written are integers
- Initial value of register is zero
- Every written value is unique
 - this can be enforced by associating a process id and a timestamp with the value

Sequential specification

- **Read()** returns the most recent **value** that was written



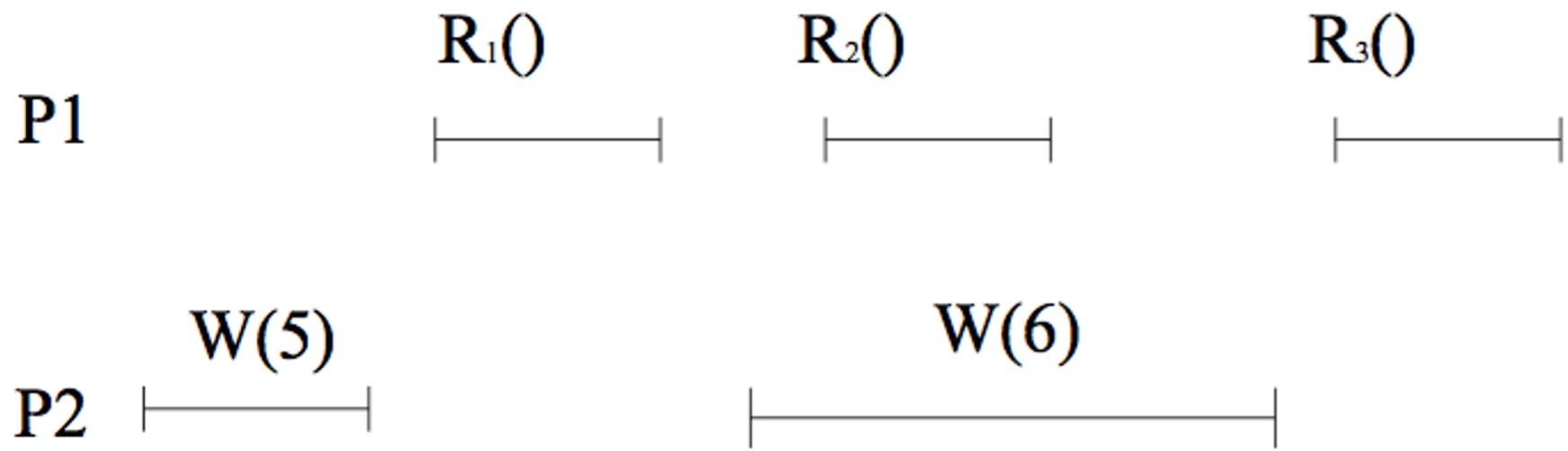
How to specify concurrent / faulty execution?



Specification #1: Regular register

- Initially, we will assume a single writer
- [Liveness] If a correct process invokes an operation, then the operation eventually completes.
- [Safety] Read must return:
 - the last value written if there is no concurrent write operation, otherwise
 - either the last value or any value concurrently written
- When a process crashes in the middle of reading/writing, the operation interval does not have an upper limit

Exercise: which values are admissible for R₁, R₂, R₃?



Exercise: which values are admissible?

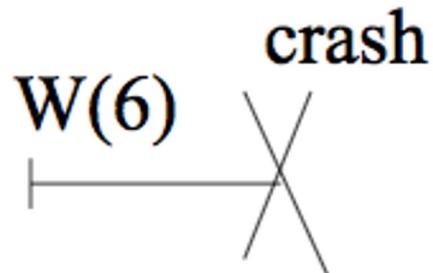
P1

$R(0) \rightarrow ?$



P2

$W(5)$



(On a side note, there exists a weaker specification: safe register)

- Similar to regular register, but:
- [Safety] Read must return:
 - the last value written if there is no concurrent write operation, otherwise
 - **any** value if there are concurrent writes
- In this course we will skip the design of safe registers

Implementation of a (1,N)-Regular register

- Uses BestEffortBroadcast + Perfect-p2p-links
- Each process maintains:

```
<ts,val>    /* Current value and associated timestamp, ts */
readlist      /* List of returned values, for reading */
rid          /* id of current read operation */
```
- Writer process maintains:

```
wts        /* Next timestamp to be written */
acks       /* How many writes have been acknowledged */
```
- Algorithm must tolerate up to f crash faults. How?

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 $\langle onrr, Write \mid v \rangle$ **do**

$wts := wts + 1$;
 $acks := 0$;
trigger $\langle beb, Broadcast \mid [WRITE, wts, v] \rangle$;

upon event $\langle beb, Deliver \mid p, [WRITE, ts', v'] \rangle$ **do**

if $ts' > ts$ **then**
 $(ts, val) := (ts', v')$;
trigger $\langle pl, Send \mid p, [ACK, ts'] \rangle$;

upon event $\langle pl, Deliver \mid q, [ACK, ts'] \rangle$ **such that** $ts' = wts$ **do**

$acks := acks + 1$;
if $acks > N/2$ **then**
 $acks := 0$;
trigger $\langle onrr, WriteReturn \rangle$;

```

upon event < onrr, Read > do
    rid := rid + 1;
    readlist := [⊥]N;
    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
        v := highestval(readlist);
        readlist := [⊥]N;
    trigger < onrr, ReadReturn | v >;

```

Is this execution valid?



Yes, reads might return the last value written of the value concurrently being written

Homework: come up with an execution that leads to this trace of outputs

Acknowledgements

- Rachid Guerraoui, EPFL

Towards a dependable consensus: Read/write protocols (cont.)

Highly dependable systems – 2024/25

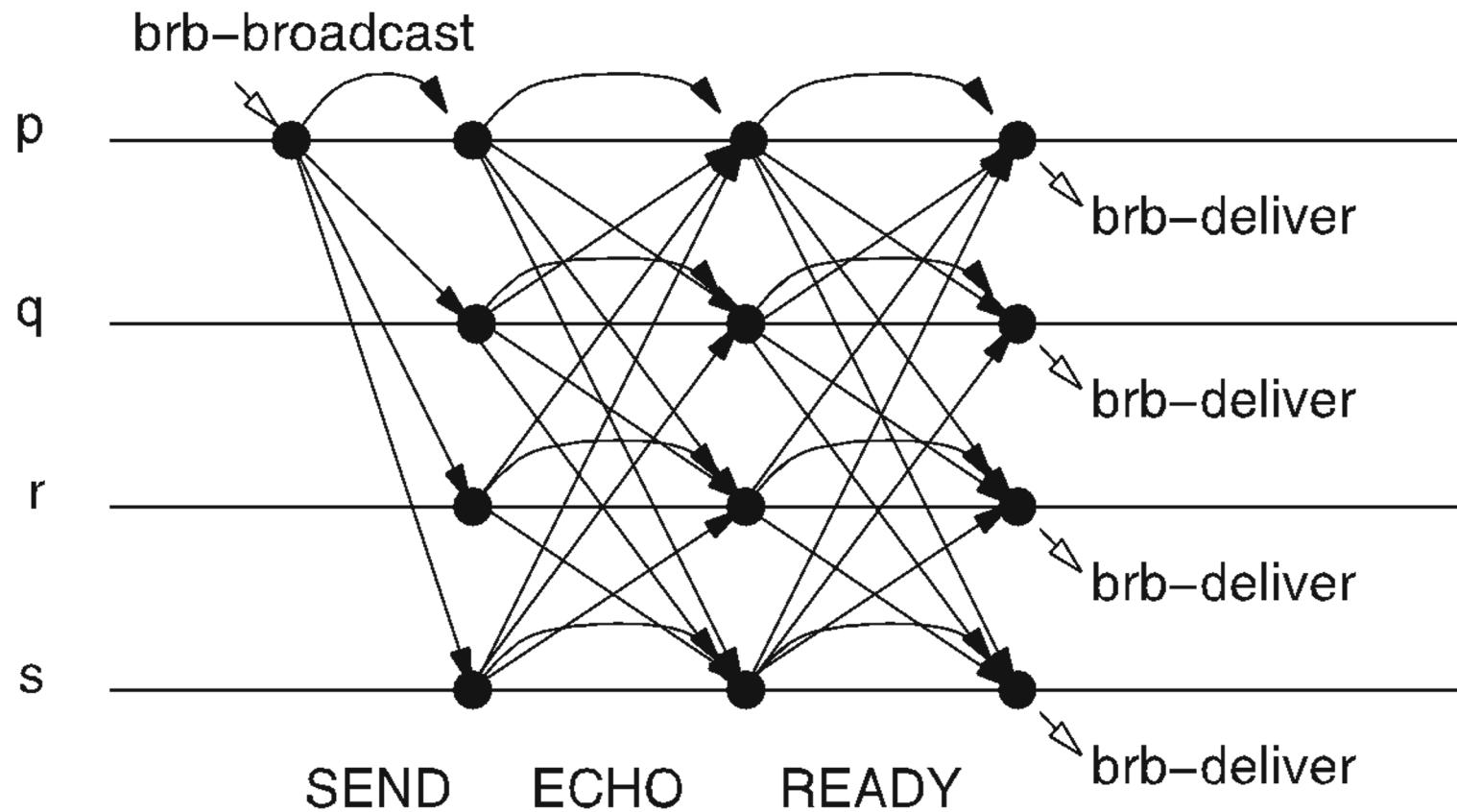
Lecture 5

Lecturers: Miguel Matos and Paolo Romano

Last lecture: Byzantine Reliable Broadcast

- BCB1: Validity: If a correct process p broadcasts a msg 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 $m = m'$.
- **BRB5: Totality: If some message is delivered by any correct process, every correct process eventually delivers a message.**

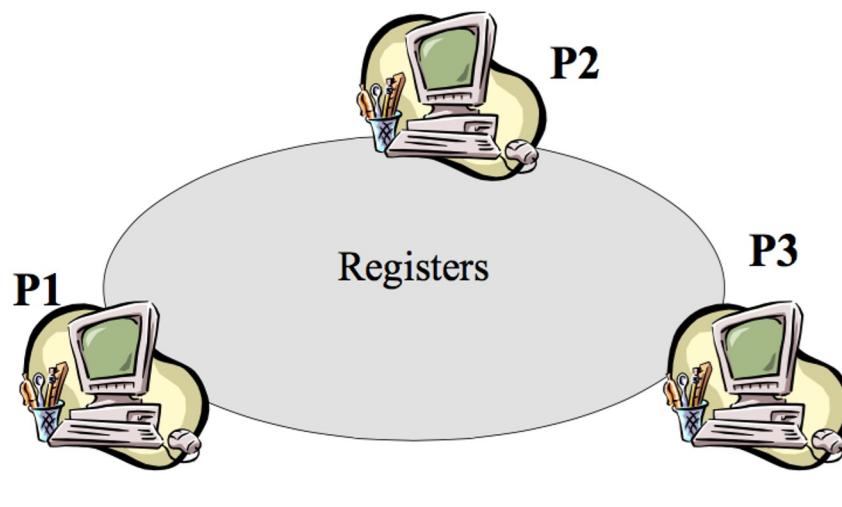
Authenticated Double-Echo Broadcast



Key proof elements

- **Consistency:** If some correct process delivers a message m and another correct process delivers a message m' , then $m = m'$.
 - if some correct process broadcasts ready messages for m and another correct process broadcasts ready for m' , then $m=m'$
 - Proof by contradiction, same as in previous proof
 - Directly implies consistency because cannot have more than f ready messages for different messages
- **Totality:**
 - if a correct process delivers, then it received at least $2f+1$ ready messages, at least $f+1$ from correct process.
 - These $f+1$ will eventually be delivered to all correct processes, triggering amplification step, and consequently sufficient ready messages for totality

New abstraction: read/write registers



- P1,P2,P3 are both clients and replicas (without loss of generality)
- **Read ()** → returns **value**
- **Write (value)** → returns "ack"

Specification #1: Regular register

- Initially, we will assume a single writer
- [Liveness] If a correct process invokes an operation, then the operation eventually completes.
- [Safety] Read must return:
 - the last value written if there is no concurrent write operation, otherwise
 - either the last value or any value concurrently written
- When a process crashes in the middle of reading/writing, the operation interval does not have an upper limit

Number of writers Number of readers

Implementation of a $(1,N)$ -Regular register

- Uses BestEffortBroadcast + Perfect-p2p-links
- Each process maintains:
 - `<ts,val>` /* Current value and associated timestamp, ts */
 - `readlist` /* List of returned values, for reading */
 - `rid` /* id of current read operation */
- Writer process maintains:
 - `wts` /* Next timestamp to be written */
 - `acks` /* How many writes have been acknowledged */
- Algorithm must tolerate up to f crash faults. How?

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 $\langle onrr, Write \mid v \rangle$ **do**

$wts := wts + 1$;
 $acks := 0$;
trigger $\langle beb, Broadcast \mid [WRITE, wts, v] \rangle$;

upon event $\langle beb, Deliver \mid p, [WRITE, ts', v'] \rangle$ **do**

if $ts' > ts$ **then**
 $(ts, val) := (ts', v')$;
trigger $\langle pl, Send \mid p, [ACK, ts'] \rangle$;

upon event $\langle pl, Deliver \mid q, [ACK, ts'] \rangle$ **such that** $ts' = wts$ **do**

$acks := acks + 1$;
if $acks > N/2$ **then**
 $acks := 0$;
trigger $\langle onrr, WriteReturn \rangle$;

```

upon event < onrr, Read > do
    rid := rid + 1;
    readlist := [⊥]N;
    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
        v := highestval(readlist);
        readlist := [⊥]N;
    trigger < onrr, ReadReturn | v >;

```

Correctness

- Liveness:
 - Any Read() or Write() eventually returns by the assumption of a majority (*quorum*) of correct processes and the liveness properties of the channels

Correctness

- Safety:
 - Split into two cases:
 1. In the absence of concurrent or failed operation, a `Read()` returns the last value written:
 - Assume a `Write(x)` terminates, and no other `Write()` is invoked. A quorum of processes have x in their local value, and this is associated with the highest timestamp in the system (because it is the last value written). Any subsequent `Read()` invocation by some process p_j returns x , as required by the specification
 2. If writes are issued concurrently, the reader will choose the highest timestamp in the quorum, which can be either the last value in case no one in the quorum had received any of the concurrent writes, or otherwise any value concurrently written

Is this execution valid?



Yes, reads might return the last value written of the value concurrently being written

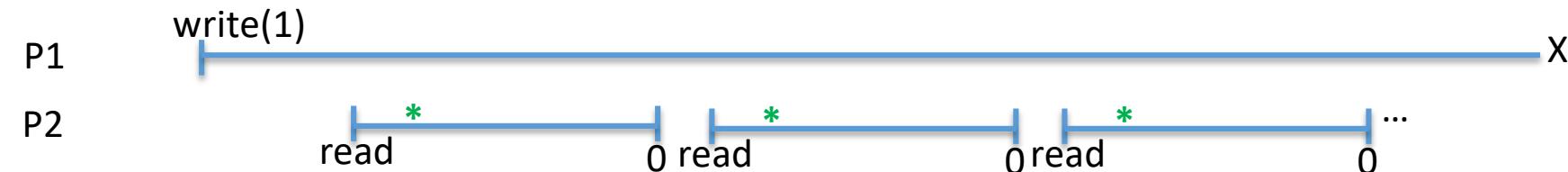
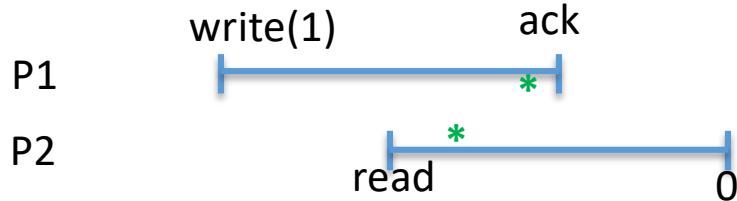
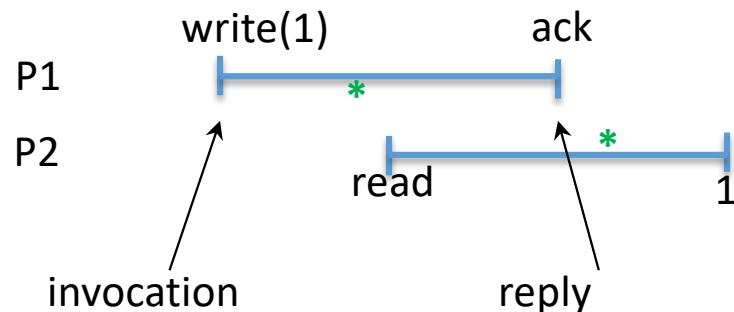
Specification #2: (1,N) Atomic register

- Additional new safety property:
- 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 :
 - $\text{write}(w)$ can either be concurrent with or follow v
- Intuition: behavior of the replicated register is the same as that of a non-replicated register
- Also known as linearizability

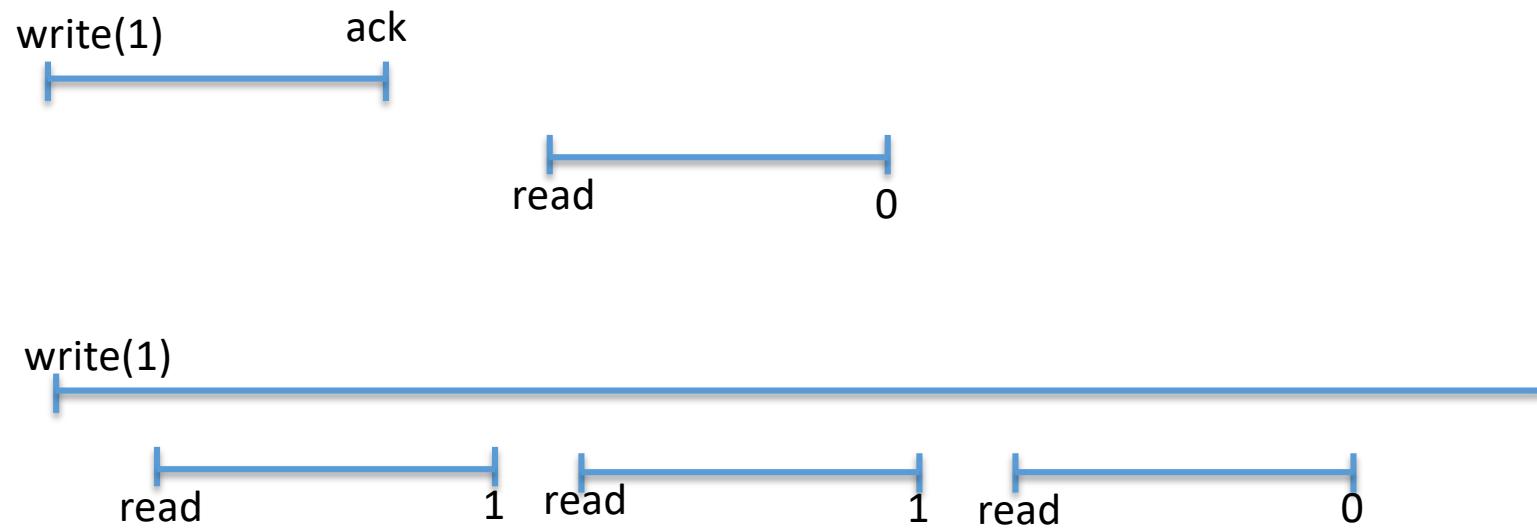
Alternative definition for atomicity (works for (N,N) -atomic registers as well)

- For any operation, there exists a **serialization point**, between the invocation and the reply, such that if we move the invocation and the reply to that point, the resulting execution obeys the sequential specification of a read/write register (operations appear to be executed at some instant between its invocation and reply time)
 - If the last operation does not return, the serialization point may or may not be included
 - (failed writes may or may not complete)

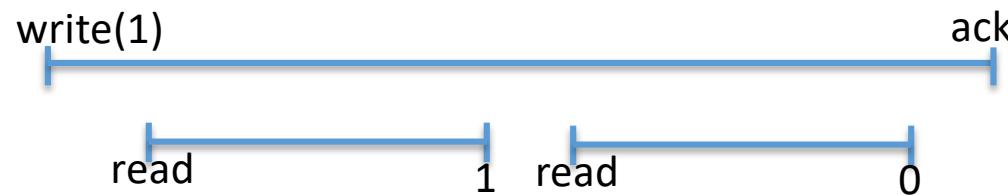
Examples of atomic executions



Non-atomic executions



How can the previous algorithm lead to a non-atomic execution?



- Exercise: draw the timing diagram of an execution that leads to these outputs
- How to fix this problem?

(1,N)-Atomic register implementation

- Write algorithm is the same as regular register
- Add write-back phase after reading
- Intuition: Reader "helps" the concurrent write by completing it in a quorum before returning

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
     $(ts, val) := (0, \perp)$ ;
     $wts := 0$ ;
     $acks := 0$ ;
     $rid := 0$ ;
     $readlist := [\perp]^N$ ;
     $readval := \perp$ ;
     $reading := \text{FALSE}$ ;

upon event <  $onar$ , Read > do
     $rid := rid + 1$ ;
     $acks := 0$ ;
     $readlist := [\perp]^N$ ;
     $reading := \text{TRUE}$ ;
    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
         $(maxts, readval) := \text{highest}(readlist)$ ;
         $readlist := [\perp]^N$ ;
        trigger <  $beb$ , Broadcast | [WRITE,  $rid$ ,  $maxts$ ,  $readval$ ] >;
```

Algorithm 4.7: Read-Impose Write-Majority (part 2, write and write-back)

upon event $\langle onar, \text{Write} \mid v \rangle$ **do**
 $rid := rid + 1;$
 $wts := wts + 1;$
 $acks := 0;$
 trigger $\langle beb, \text{Broadcast} \mid [\text{WRITE}, rid, wts, v] \rangle$;

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

upon event $\langle pl, \text{Deliver} \mid q, [\text{ACK}, r] \rangle$ **such that** $r = rid$ **do**
 $acks := acks + 1;$
 if $acks > N/2$ **then**
 $acks := 0;$
 if $reading = \text{TRUE}$ **then**
 $reading := \text{FALSE};$
 trigger $\langle onar, \text{ReadReturn} \mid readval \rangle$;
 else
 trigger $\langle onar, \text{WriteReturn} \rangle$;

From (1,N) to (N,N) atomic register

- With a single writer, writing process can simply increment counter to determine next timestamp
- How to do this with multiple writers?
- Must pick timestamp greater than most recent write
 - writers determine first the timestamp using a majority
- Must be able to break ties
 - timestamp is <seq_number,id> pair

Uses:

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

```
upon event < nmar, Init > do
    (ts, wr, val) := (0, 0, ⊥);
    acks := 0;
    writeval := ⊥;
    rid := 0;
    readlist := [⊥]N;
    readval := ⊥;
    reading := FALSE;

upon event < nmar, Read > do
    rid := rid + 1;
    acks := 0;
    readlist := [⊥]N;
    reading := TRUE;
    trigger < beb, Broadcast | [READ, rid] >;

upon event < beb, Deliver | p, [READ, r] > do
    trigger < pl, Send | p, [VALUE, r, ts, wr, val] >

upon event < pl, Deliver | q, [VALUE, r, ts', wr', v'] > such that r = rid do
    readlist[q] := (ts', wr', v');
    if #(readlist) > N/2 then
        (maxts, rr, readval) := highest(readlist);
        readlist := [⊥]N;
        if reading = TRUE then
            trigger < beb, Broadcast | [WRITE, rid, maxts, rr, readval] >;
        else
            trigger < beb, Broadcast | [WRITE, rid, maxts + 1, rank(self), writeval] >;
```

```

upon event < nnar, Write | v > do
    rid := rid + 1;
    writeval := v;
    acks := 0;
    readlist := [⊥]N;
    trigger < beb, Broadcast | [READ, rid] >

upon event < beb, Deliver | p, [WRITE, r, ts', wr', v'] > do
    if (ts', wr') is larger than (ts, wr) then
        (ts, wr, val) := (ts', wr', 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 < nnar, ReadReturn | readval >;
        else
            trigger < nnar, WriteReturn >;

```

Byzantine (1,N) read/write regular register

- To specify registers in the Byzantine model, we will simplify by considering that clients only suffer crash faults
- With this restriction and adapting the notion of "correct", specification is the same as in crash model
- How to implement?

Strawman attempt

- Start with (1,N) regular register algorithm
- Replace links with perfect authenticated links
- Replace majorities with Byzantine quorums of $2f+1$ out of $3f+1$
- Does this work?
- Problem: Byzantine process can return an arbitrary $\langle ts, value \rangle$

Illustration of problem with strawman

- Exercise: draw a timing diagram to illustrate why the strawman doesn't work
- Solution?
 - Make written values and timestamp unforgeable by byzantine processes... how?

Solution: clients sign written values

- Upon writing, clients piggyback signature of $\langle \text{seq_number}, \text{value} \rangle$
- Replicas store this signature and return it upon read requests
- When reading, clients discard incorrectly signed values
- Need to send identifier associated with read (rid), to avoid replay attacks

Implements:

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

Uses:

AuthPerfectPointToPointLinks, **instance** al .

upon event $\langle bonrr, Init \rangle$ **do**

$(ts, val, \sigma) := (0, \perp, \perp);$

$wts := 0;$

$acklist := [\perp]^N;$

$rid := 0;$

$readlist := [\perp]^N;$

upon event $\langle bonrr, Write \mid v \rangle$ **do**

// only process w

$wts := wts + 1;$

$acklist := [\perp]^N;$

$\sigma := sign(self, bonrr \parallel self \parallel \text{WRITE} \parallel wts \parallel v);$

forall $q \in \Pi$ **do**

trigger $\langle al, Send \mid q, [\text{WRITE}, wts, v, \sigma] \rangle;$

upon event $\langle al, Deliver \mid p, [\text{WRITE}, ts', v', \sigma'] \rangle$ **such that** $p = w$ **do**

if $ts' > ts$ **then**

$(ts, val, \sigma) := (ts', v', \sigma');$

trigger $\langle al, Send \mid p, [\text{ACK}, ts'] \rangle;$

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

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

upon event $\langle al, Deliver \mid p, [READ, r] \rangle$ **do**
 trigger $\langle al, Send \mid p, [VALUE, r, ts, val, \sigma] \rangle;$

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

Byzantine (1,N) read/write atomic register

- Also assume crash-faulty clients
- Also same specification as in the crash model
- Similar adaptation to the crash-fault-tolerant counterparts (from regular to atomic registers)
 - Add write back phase to read algorithm
 - Exercise: extend the algorithm from the previous two slides

Byzantine clients

- Now, assume that clients can be byzantine in digital signature-based BFT algorithm
- Read operations:
 - Do not modify the state of the processes → safe from everyone else's point of view
- Write operations:
 - Byzantine clients can pick wrong timestamps, send write requests for different values to different servers, or only write to a subset
 - Leave it as an exercise to think through consequences and countermeasures

Acknowledgements

- Rachid Guerraoui, EPFL

Consensus protocols

Highly dependable systems

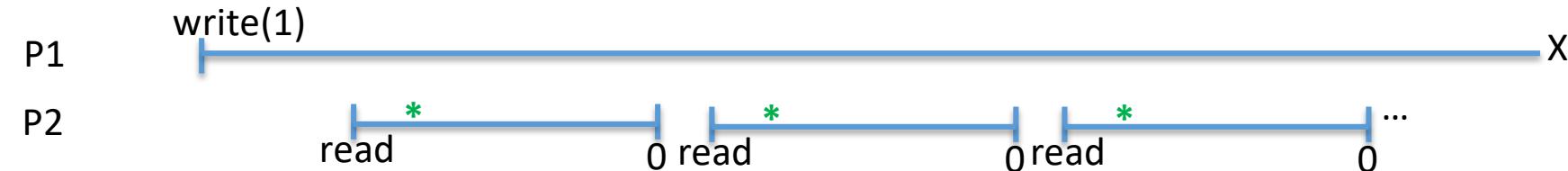
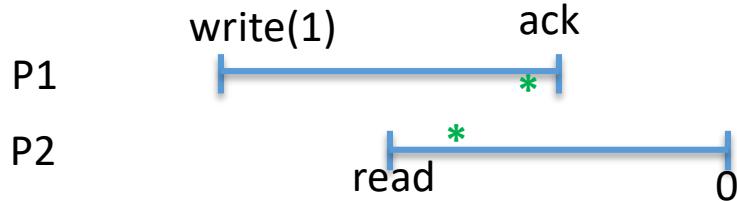
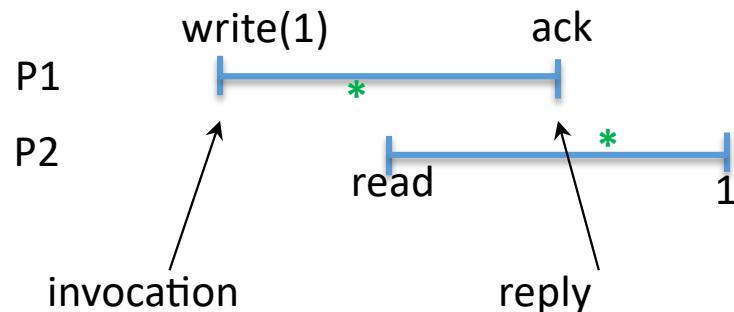
Lecture 6

Lecturers: Miguel Matos and Paolo Romano

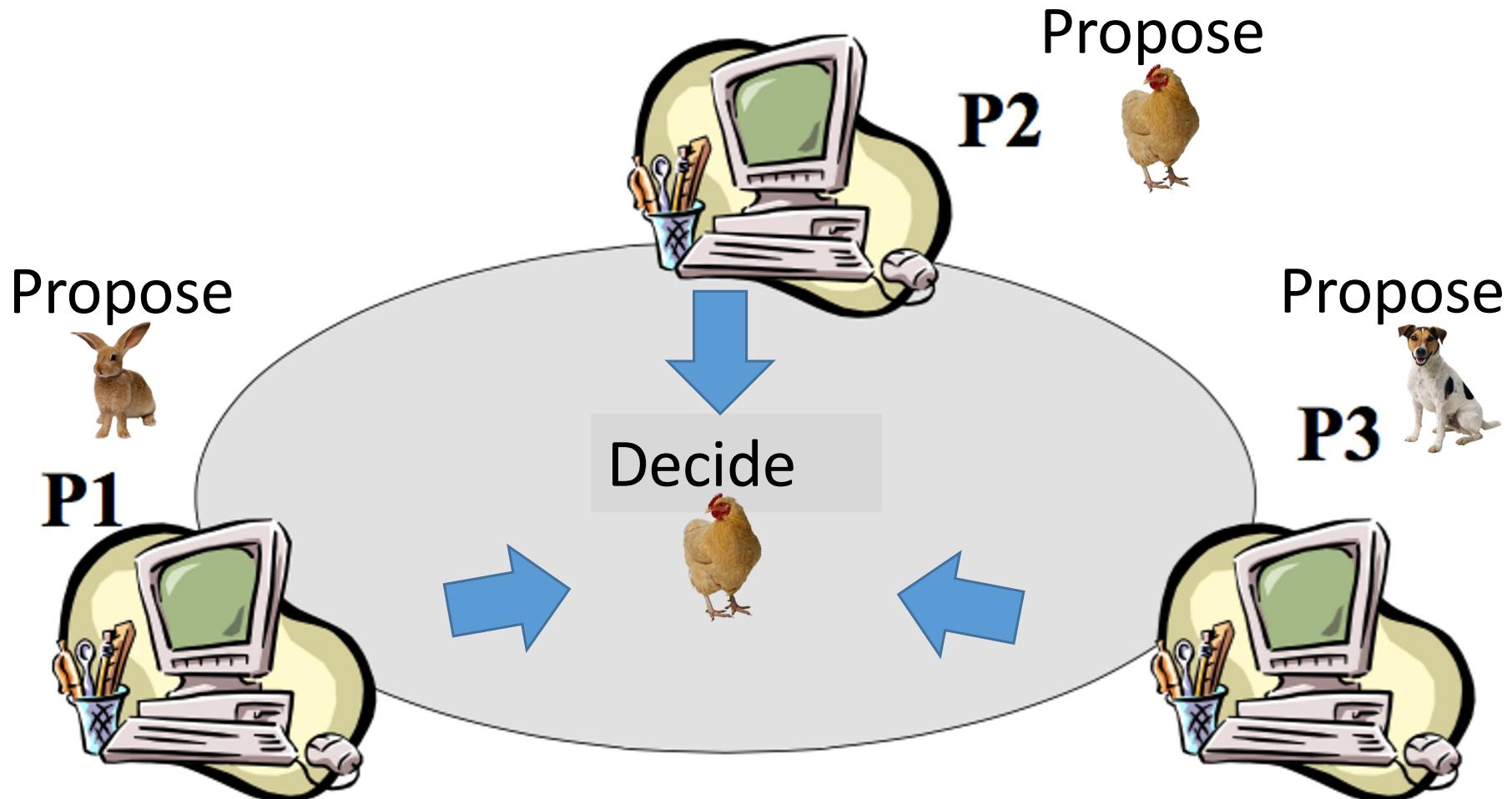
Last lecture: atomicity / linearizability (works for (N,N)-atomic registers as well)

- For any operation, there exists a **serialization point**, between the invocation and the reply, such that if we move the invocation and the reply to that point, the resulting execution obeys the sequential specification of a read/write register (operations appear to be executed at some instant between its invocation and reply time)
 - If the last operation does not return, the serialization point may or may not be included
 - (failed writes may or may not complete)

Examples of atomic executions



Consensus



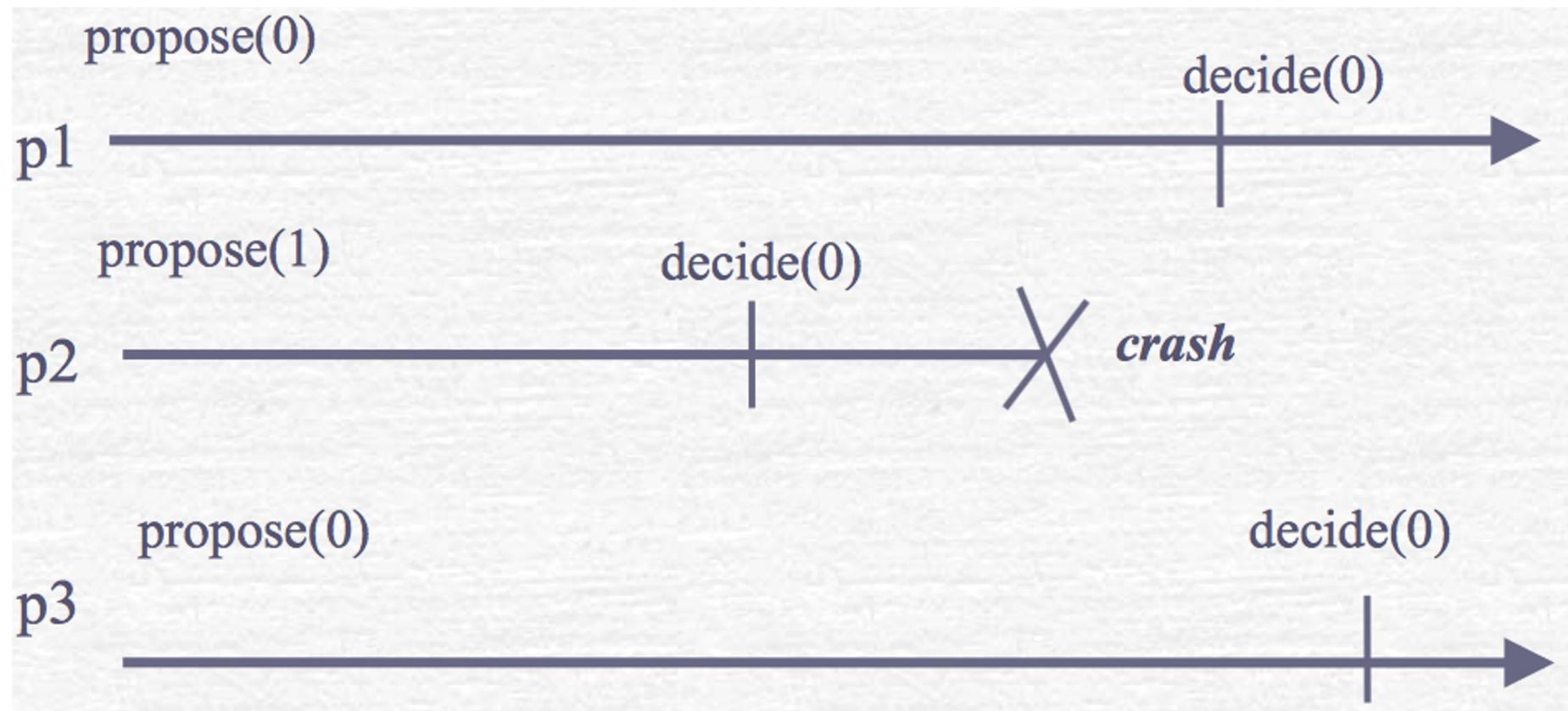
The consensus problem

- Basic idea: each process has an input proposal
- All processes must reach the same output decision
- Must be safe despite faults, asynchrony
- This is a key building block in many systems
 - generic state machine replication
 - coordination systems like Apache ZooKeeper (CFT)
 - permissioned blockchains or permissionless side chains (BFT)

Specification in the crash model: Uniform consensus

- Events:
 - Request: <Propose, v>
 - Indication: <Decide, v'>
- Properties:
 - C1. Validity: Any value decided is a value proposed
 - C2. [Uniform] Agreement: No two processes decide differently
 - C3. Termination: Every correct process eventually decides
 - C4. Integrity: No process decides twice

Example of a valid trace



Algorithm to solve consensus in the crash model: Paxos

- Submitted for publication in 1990
- Reviewers said it was mildly interesting, though not very important – and that the presentation was distracting
- Paper was rejected and shelved
- Eventually published after a decade
- Then adopted at Google (published in 2006)
- Now a standard building block used by many systems

Paxos in a nutshell

- This is covered in another course, plus our focus is not on the crash model
- Here, we give a brief outline

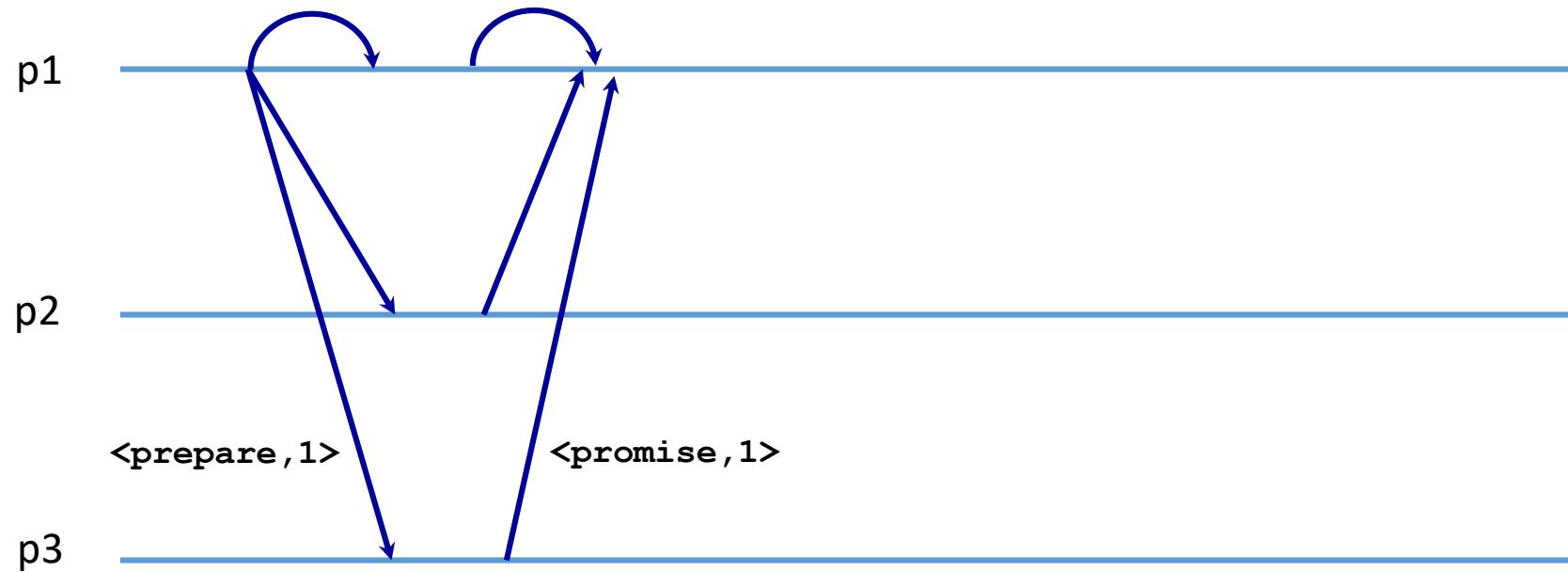
Overview

- Any process can propose v , first to reach a majority wins
- How do we select among multiple proposals?
- Associate timestamp $\langle \text{seqno}, \text{process id} \rangle$ with v
- Protocol has two phases:
 - First, processes read the state of others to form proposal
 - Second, try to convince others to accept their proposal

Protocol steps (first phase)

1. Process p chooses a proposal timestamp $n = [sn, p]$
2. All processes keep track of:
 - timestamp accepted and associated value $\langle n_a, v_a \rangle$, and
 - most recent promise not to accept lower timestamps, n_h
3. p sends prepare msg, asking all processes if they already accepted any proposals with $n_a < n$
4. if so, reply $\langle n_a, v_a \rangle$ else set $n_h = n$ (and return this promise not to accept anything below n)

First phase example run



Protocol steps (second phase)

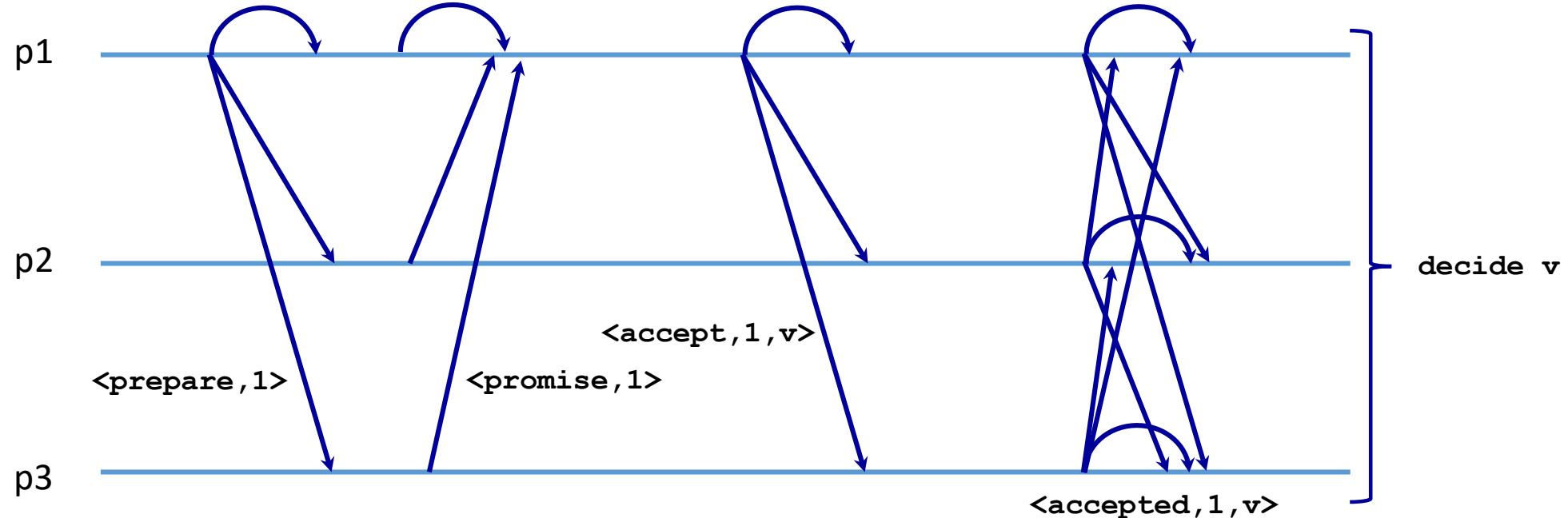
1. After p collects quorum of replies, send either a previously accepted value (if it was received) or its own proposal in an $\langle \text{accept}, \langle n, v \rangle \rangle$ message
2. Processes accept proposal if $n \geq n_h$ setting:

$$n_h = n_a = n$$

$$v_a = v$$

(Then convey decision to all processes through accepted message)

Second phase example run

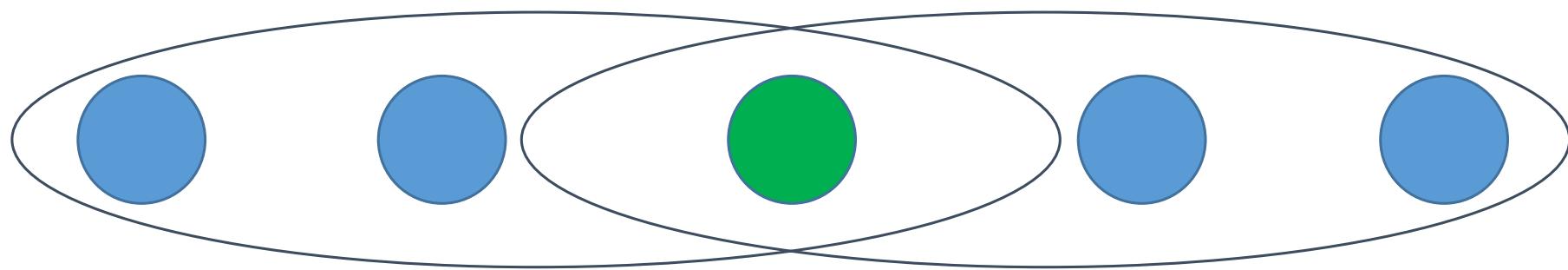


Paxos in practice – multi-Paxos

- Instead of running two phases for every “operation”:
 - use phase 1 to nominate a leader (run phase 1 for all possible operations / instances of consensus)
 - let the leader run phase 2 each time an operation is executed (thus concluding one of the consensus instances)
 - if leader is non-responsive, then goto first step
- Parallel to IBFT (phase 1 is a round change, phase 2 is the normal case operation)

Why is Paxos safe?

- Agreement is guaranteed by the fact that if a proposal with v is accepted (majority of accept s were issued), then any higher-numbered proposal must have value v



v is chosen →
Quorum accepted $\langle n, v \rangle$

An attempt to decide in
 $n+1$ will have to propose v

But is it live?

Impossibility of consensus (FLP)

- There is no **deterministic** protocol that solves consensus in an asynchronous system where even a single process may suffer a crash fault
 - Fisher, Lynch, and Paterson. Impossibility of distributed consensus with one faulty process. JACM, Vol. 32, no. 2, April 1985, pp. 374-382
- We will present a simple and elegant proof for consensus among two processes
 - The main result applies to an arbitrary number of processes

Proof of the impossibility of consensus

- By contradiction, let's consider that there exists an algorithm that solves consensus
- We consider three different executions of that algorithm, with varying network conditions
 - Note that any behavior from the network is possible in an asynchronous system
- The two processes executing consensus are called A and B

Execution #1

- Both processes propose 0 initially
- Process B crashes as soon as the execution starts
- By the validity condition of the specification, process A must decide 0
- And by the termination property it must eventually decide → let's say it decides at some instant t1

Execution #2

- Both processes propose 1 initially
- Process A crashes as soon as the execution starts
- By the validity condition of the specification, process B must decide 1
- And by the termination property it must eventually decide → let's say it decides at some instant t2

Execution #3

- Process A proposes 0 and process B proposes 1 initially
- Messages between A and B (in both directions) are delayed such that they are never delivered up until $\max(t_1, t_2)$
- Process A decides 0 by t_1 , since its execution is indistinguishable from execution #1
- Process B decides 1 by t_2 , since its execution is indistinguishable from execution #2
- We found a contradiction (which?)

Byzantine fault tolerant consensus

- Recall previous specification (crash model):
 - Termination: Every correct process eventually decides
 - Validity: Any value decided is a value proposed
 - Integrity: No process decides twice
 - Agreement: No two processes decide differently
- Which property needs to be revisited in the Byzantine model?

Weak Byzantine consensus

- Termination: Correct processes eventually decide.
- Weak validity: If all processes are correct and some process decides v , then v was proposed by some process.
 - If some processes are faulty, any value may be decided
- Integrity: No correct process decides twice.
- Agreement: No two correct processes decide differently.

Strong Byzantine consensus

- 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 \square

Weak vs Strong Byzantine consensus

- *Strong validity* does *not* imply weak validity
- Strong validity allows to decide \square
- *Weak validity* requires (only if all processes are correct) that the decided value was proposed by some (correct) process
- The two Byzantine consensus notions are not directly comparable
- For this class, we **focus on weak validity**

Implementing BFT consensus

- Strategy is similar to Paxos, i.e., modularize into:
- EpochChange
 - Choose a leader, and make sure any previously decided value carries over to the new epoch
- EpochConsensus
 - Try to reach decision within an epoch
 - May fail, in which case it aborts and returns state to initialize new EpochConsensus

Byzantine Epoch Change

- Leverage Byzantine leader election protocol from Lecture 3
- Recap: if the consensus algorithm is not making progress (timeout), process i broadcasts a NEWEPOCH message to all processes.
- If a process receives more than f NEWEPOCH messages, also broadcasts NEWEPOCH
 - Prevents unwanted epoch change. Why?
- If a process receives more than $2f$ NEWEPOCH messages it changes epoch.
 - Cannot wait for more. Why?

EpochConsensus: interface

- Tries to achieve consensus within an epoch, but may abort unless leader is correct and network behaves synchronously
- Interface (events):
 - **Request:** $\langle \text{bep}, \text{Propose} \mid v \rangle$: Proposes value v for epoch consensus. Executed only by the leader l .
 - **Request:** $\langle \text{bep}, \text{Abort} \rangle$: Aborts epoch consensus.
 - **Indication:** $\langle \text{bep}, \text{Decide} \mid v \rangle$: Outputs a decided value v of epoch consensus.
 - **Indication:** $\langle \text{bep}, \text{Aborted} \mid st \rangle$: Signals that epoch consensus has completed the abort and outputs internal state st .

EpochConsensus: specification (for epoch with timestamp ts)

- *Validity:*
If (all processes are correct and) a process ep-decides v , then v was ep-proposed by a leader of epoch consensus with timestamp $ts' \leq ts$.
- *Uniform agreement:*
No two correct processes ep-decide different values.
- *Lock-in:*
If a correct process ep-decided v in an epoch consensus with timestamp $ts' < ts$, processes cannot decide a value $v' \neq v$.
- *Termination:*
If the leader is correct, has ep-proposed a value, and no correct process aborts this epoch consensus, then every correct process eventually ep-decides

Byzantine Epoch Consensus (read phase)

- Leader sends READ to all processes
- Processes reply with STATE message containing its local state <valts, val, writeset>:
 1. **(valts, val)** - a timestamp/value pair with the value that the process received most recently in a **Byzantine quorum of WRITE messages**
 2. **writeset** - a set of timestamp/value pairs with one entry for every value that this process has ever written (where timestamp == most recent epoch where the value was written).

Outcome of the read phase

- Read phase obtains the states from a byz. quorum of processes to determine whether there exists a value that may have been epoch-decided (if so, it must be written, to ensure lock-in property)
- If so, send this value in the subsequent WRITE
- What are the required conditions to be able to affirm that a value may have been epoch-decided?

Outcome of the read phase

1. The value corresponds to the highest timestamp in a byzantine quorum of (timestamp,value) pairs reported in distinct STATE messages
 - This is the most recent value for which a process claims to have received a Byzantine quorum of WRITES
2. The value appears in the writeset of at least $f+1$ processes
 - This ensures value occurs in the writeset of a correct process
 - If no value meets these two conditions, then outcome is *unbound*

Read phase: coping with byzantine leaders

- Leader sends the STATEs collected in the read phase to all
 - processes send their states digitally signed, to prevent tampering
- All processes independently check, based on information in state messages, if some value may have already been ep-decided in a previous epoch (lock-in property)

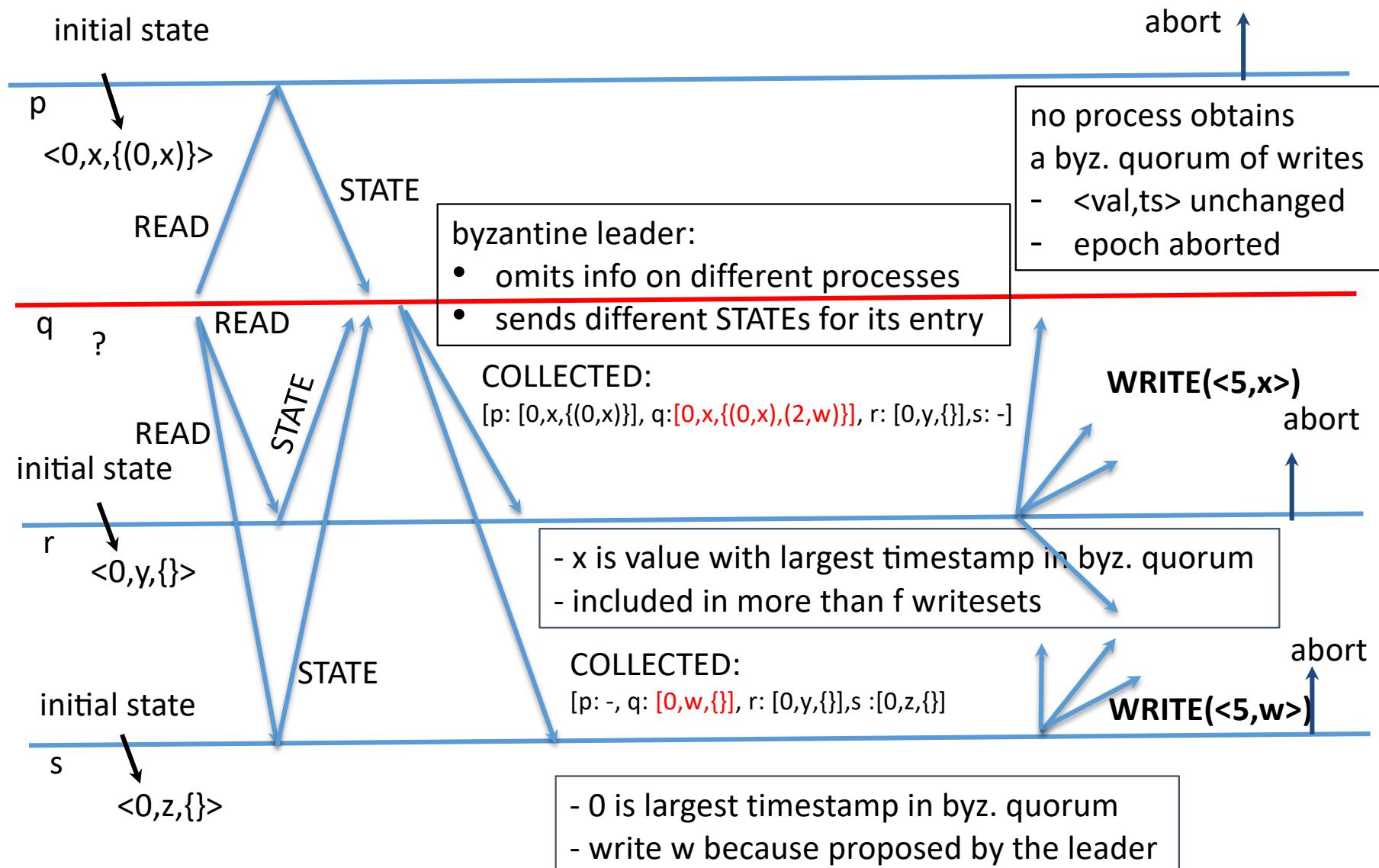
Read phase: coping with byzantine leaders

- A leader cannot forge STATE values of other processes, thanks to the use of digital signatures
 - but it can omit information from some process
 - or send different values regarding its state to different processes
- However, the conditions governing the outcome of the read phase prevent safety violations

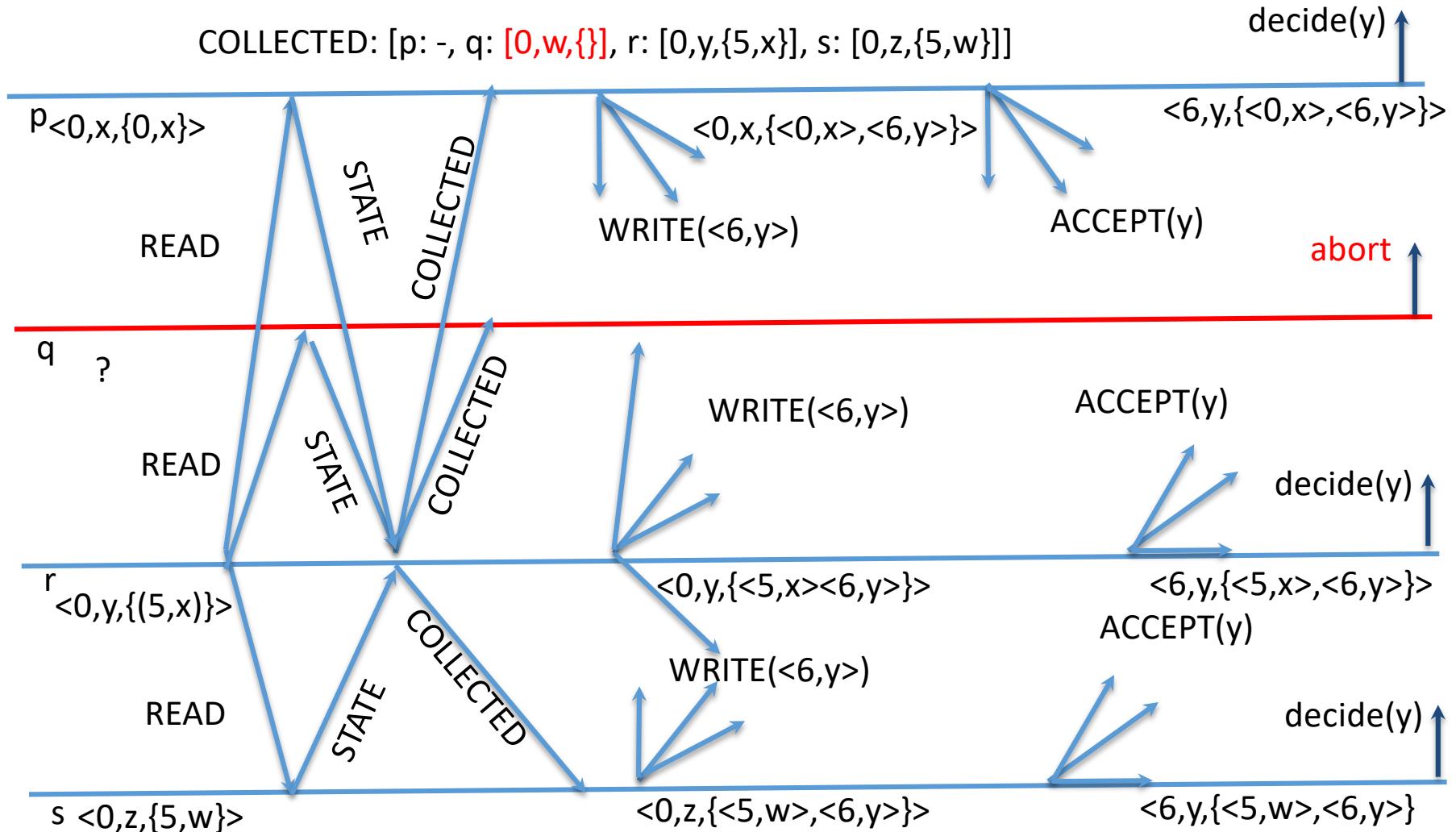
Write phase

- If a process receives a Byzantine quorum of WRITE messages from distinct processes containing the same value v , it sets its state to $(\text{current_epoch}, v)$ and broadcasts an ACCEPT message
- When a process receives a Byz. quorum of ACCEPT messages from distinct processes containing the same value v , it epoch-decides v

Example execution: byzantine leader q in epoch 5



Example execution: correct leader r in epoch 6



r is correct → all processes get the same COLLECTED unless they time out (asynchrony), they will all write the same value, and accept it

Correctness sketch

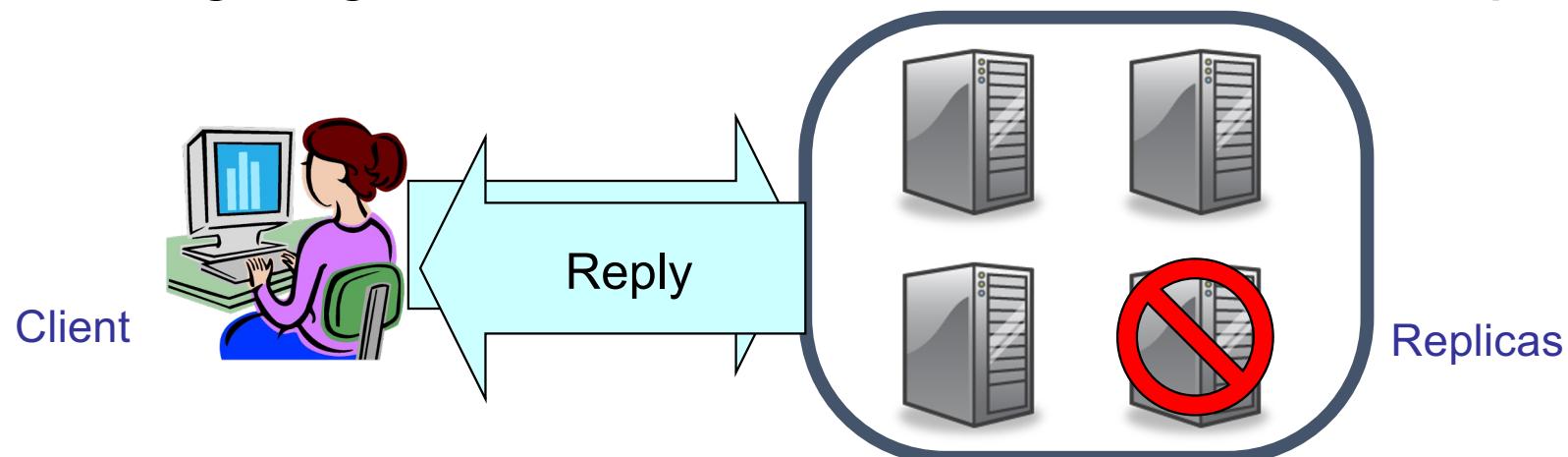
- Agreement property:
 - Usual contradiction proof based on collecting $2f+1$ ACCEPTs, and the fact that correct processes do not send conflicting ACCEPT messages
- Validity property:
 - Weak validity applies only to executions with only correct processes, simplifying the proof
- Termination and abort behavior property:
 - Follows from sequence of steps after correct leader starts the protocol

Correctness sketch (lock-in property)

- assume process p ep-decided v in consensus instance $ts' < ts$
- then, p collected $2f+1$ ACCEPTs for v, at least $f+1$ from correct processes, who set value and timestamp to $\langle v, ts' \rangle$
- those ACCEPTS follow from receiving $2f+1$ WRITES, at least $f+1$ from correct processes, who added (ts', v) to their writeset
- now let's consider the first subsequent instance ts^* where a correct process receives COLLECTED, we prove that the outcome of the read phase has to be v
 - Between ts' and ts^* no correct process received COLLECTED, thus did not send write, thus state variables valts, val, and writeset did not change
 - Thus the $f+1$ correct processes use (ts', v) as the starting value of ts^* and include it in writeset
 - By construction of the outcome of the read phase, its output must be bound to ts'
 - Therefore, all correct processes that write will write v, implies that correct processes that decide will decide v in ts^*
 - Recursively using the same argument until round ts establishes the property

State machine replication (SMR)

1. Take an arbitrary service, make it deterministic
Example: an append-only sequence of blocks of transactions
2. Replicate the server
3. Enforce that correct replicas execute request in the same order (follow the same sequence of state transitions)
4. Use voting to guarantee that client sees correct output



From consensus to state machine replication

- Consensus protocol is at the heart of solving point number 3
 - Clients issue several requests independently of each other
 - Each request is assigned a sequence number, thus defining order by which they are executed
 - Instantiate one consensus instance per sequence number, to determine which request gets executed at that point in the sequence
- Can optimize the EpochConsensus protocol for this setting:
- When instantiating new epoch, read phase of the protocol can be executed only once for requests in the interval $[current, +\infty)$

Acknowledgements

- Rachid Guerraoui, EPFL

Permissionless blockchains. Bitcoin

Highly dependable systems

Lecture 7

Lecturers: Miguel Matos and Paolo Romano

Last Lecture: Consensus

- Fundamental building block in distributed systems
- Through the State Machine Replication approach, it can be used to replicate any (deterministic) system, including:
 - databases, file-systems, video-games,...
 - as well as **distributed ledgers**

Distributed Ledgers

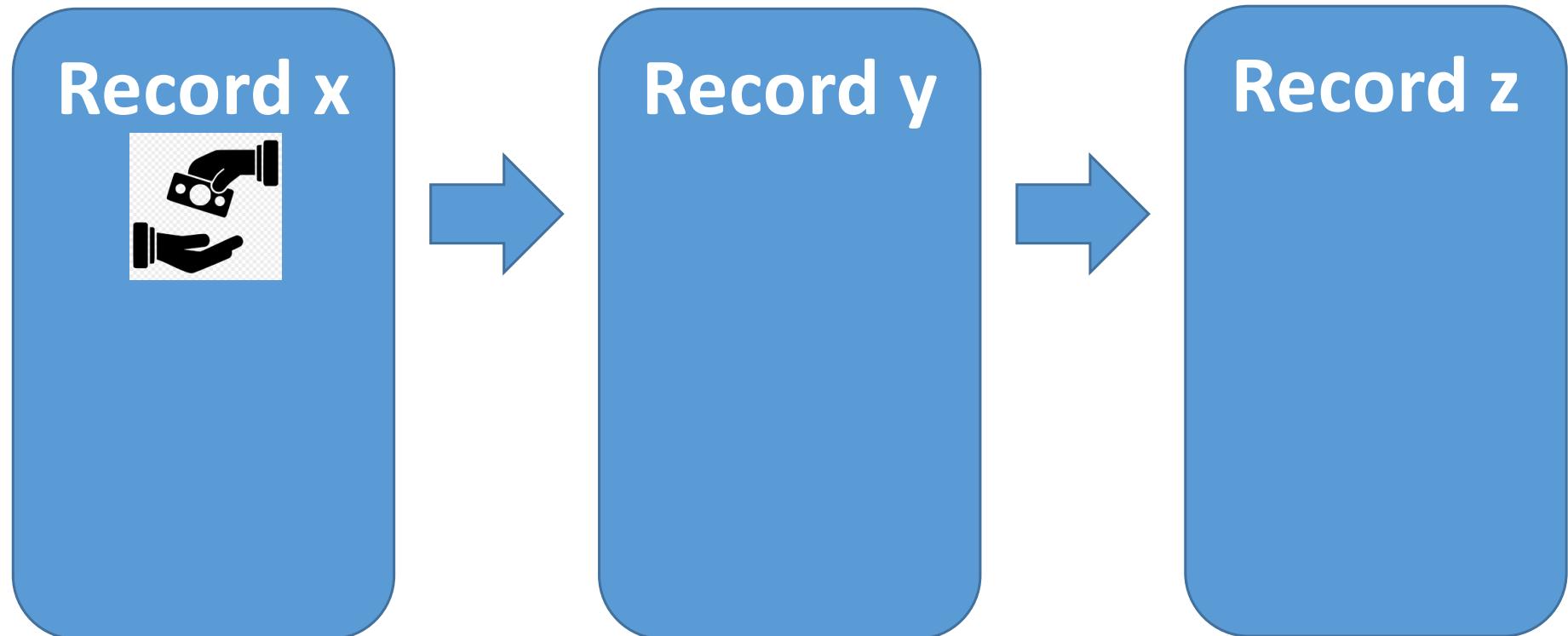
- Ledger
 - Record of transactions
 - Tamper proof
 - Ordered
- Distributed Ledger with a fixed set of participants (aka “permissioned”), without centralized authority or control
 - E.g., consortium of companies, special entity approves new members
 - Solution: replicate the ledger
 - Use consensus to decide which operation to append next to the ledger
- Challenge: in an open Internet environment, membership is open and dynamic (aka “permissionless”)

General Ledger Sheet			Sheet No:		
Account:	Description	Journal ref.	Transaction	Balance	
Date				Debit	Credit

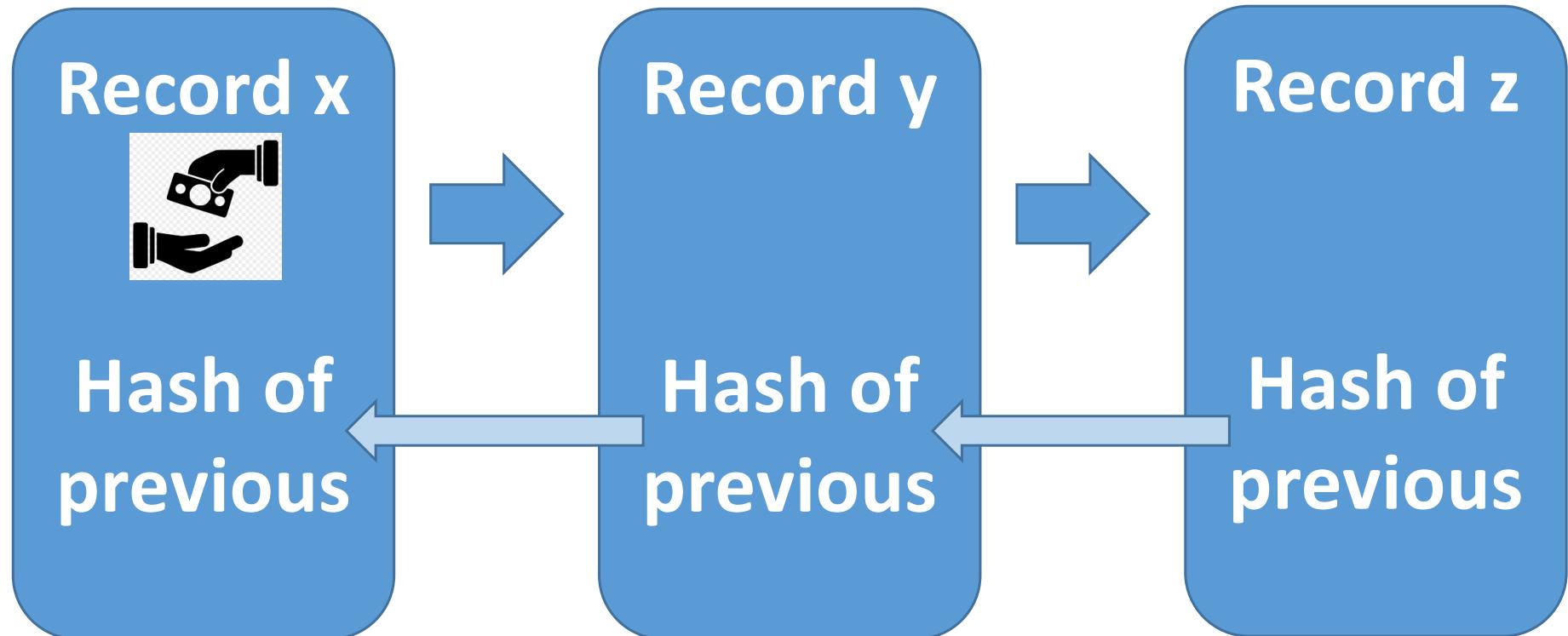
Débito				Crédito			
Saldo de contas corrente	9880	Caixa	Reservado	Saldo de contas corrente	9880	Cash	Reservado
Arrendamento	1.526,00			1206,00			1042,80
Aprendiz	- 480,00	3600		1800			480,00
Aluguel	- 4.667,75	4.667,75				2.100,00	2.100,00 / 1.737,75 / 1000,00 Renda fixa
Aluguel	- 1.620,00		1.620,00			40,00	1.620,00
Assinatura	- 33,25,30			57,50	137,00	3.313,30	50,00
Caixa de seguros	2.710,00	2.710,00		130,00	2.070,20	2.710,00	
Caixa de seguros	- 2.626,00		130,00			140,00	
	14.812,05	4.667,75	2.710,00	130,00	2.070,20	3.313,30	80,00
Abertura de contas	108,66				108,66		
Arrendamento	- 110,00			110,00			110,00
Aluguel	- 52,00	32,50		19,00			32,00

What is the blockchain?

Merely a list of events that were recorded



But also append-only and tamper-proof



Origins associated with Bitcoin, but...



Video TV News Tech Rec Room Food World News

MOTHERBOARD
TECH BY VICE

The World's Oldest Blockchain Has Been Hiding in the New York Times Since 1995

This really gives a new meaning to the "paper of record."

J. Cryptology (1991) 3: 99–111

Journal of Cryptology

© 1991 International Association for
Cryptologic Research

How To Time-Stamp a Digital Document¹

Stuart Haber and W. Scott Stornetta
Bellcore, 445 South Street,
Morristown, NJ 07960-1910, U.S.A.
stuart@bellcore.com stornetta@bellcore.com

Abstract. The prospect of a world in which all text, audio, picture, and video

**NOTICES &
LOST AND
FOUND**

(5100-5102)

Universal Registry Entries:

Zone 2-

dS8492cgVOFAoP9kyE1XzMOrQ
HgEwzkVbVafNylkUz99qvq8/ME
p5v9EFSG8XxzMBalGQQ==

Zone 3-

JnFCg+HCmvhj8GmmUP7VZna71
NgZup/RfuKUQNzCHWXMuqLK
durxHQV5pSHLqBGPRiy+mg==

These base64-encoded values represent the combined fingerprints of all digital records notarized by Surety between 2009-06-03Z 2009-06-09Z.
www.surety.com 571-748-5800

In whom do we trust?

- Publishing a weekly hash in the NYT was a way to avoid having centralized trust on Surety's internal records.
- In permissioned blockchains, this trust is replaced with voting
 - Provably correct if colluding attackers do not exceed 1/3 of the members
- Problem: in permissionless settings, anyone can vote. What is the meaning of “anyone”?

Depends on who (and when) you ask



- Voted in 1911 (using a loophole)

On the Internet, it's easy to get voting rights

- Entities != identities
- Easy to create multiple public/private key pairs or IP addresses
- Known as a Sybil attack
- Force any decision, e.g.:
 - Double-spend
 - Break agreement



Proof-of-Work

- Invented in the context of fighting e-mail spam [Hashcash 1997]
 - Every time an e-mail is about to be sent, sender must solve a puzzle that requires computational power
 - Puzzle depends on the contents of the e-mail
 - Hard to produce – hence requires time
 - Easy to verify – if the puzzle is not correct, reject message
 - Limits the number of e-mails that can be sent
- From the original bitcoin paper:

“The proof-of-work also solves the problem of determining representation in majority decision making. If the majority were based on one-IP-address-one-vote, it could be subverted by anyone able to allocate many IPs. Proof-of-work is essentially one-CPU-one-vote.”

“The system is secure as long as honest nodes collectively control more CPU power than any cooperating group of attacker nodes.”

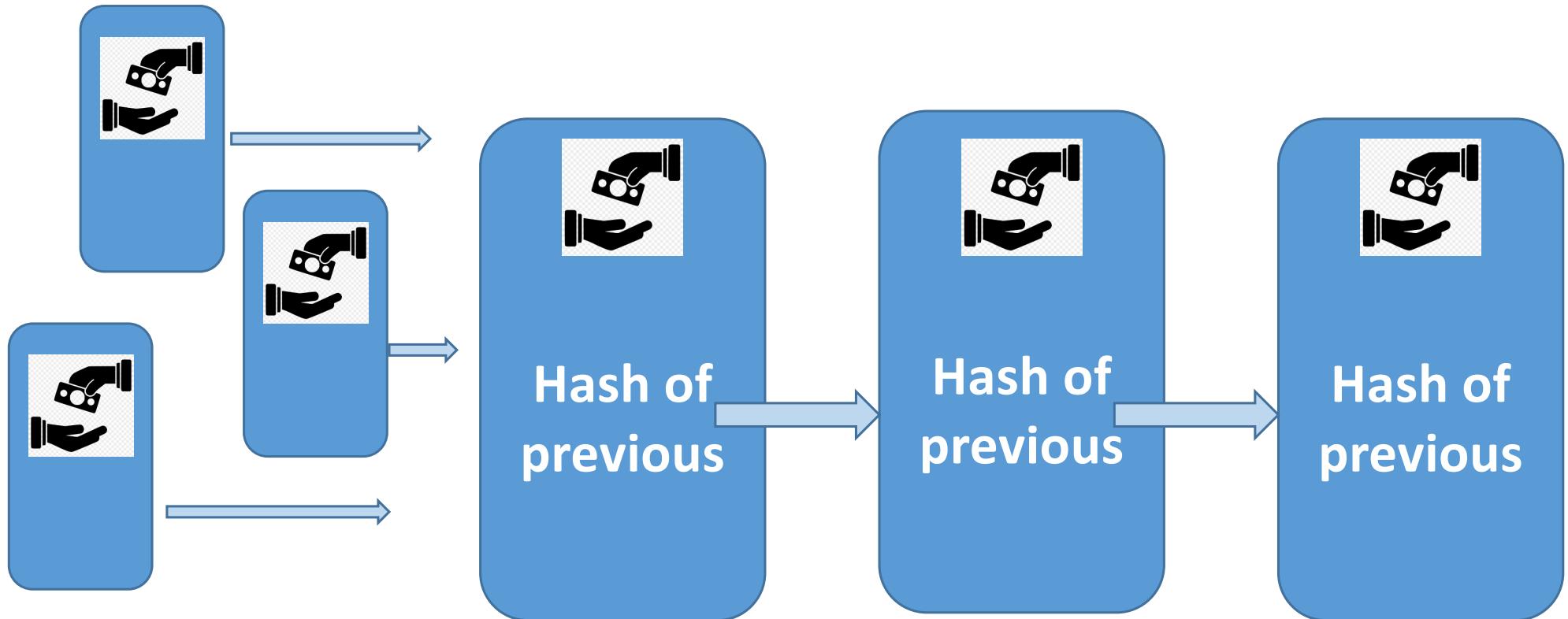
Crypto puzzles used in PoW

- Find **nonce** such that
 - Hash(e-mail || nonce)
- has **X** leading zeros
 - Hash("hello world" || 0000) = 0xfad....
 - Hash("hello world" || 0001) = 0xdeb....
 - ...
 - Hash("hello world" || 0999) = 0x**000fe**... Ok for X=3
- X is the difficulty of the puzzle

Proof-of-Work

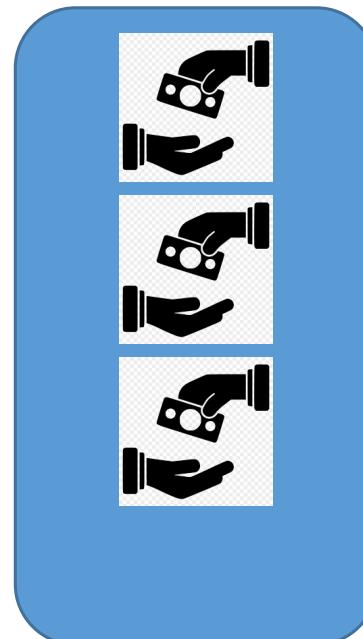
- A secure hash function is hard to invert
 - Best approach is to use brute force
 - Puzzle solver needs to try many combinations (which takes time) until finding the solution
 - Trying more combinations per unit of time implies more CPU power
- Puzzle verifier needs to execute the hash function only once to verify if the puzzle is correct
 - Verifying correctness of digest is very fast

How is PoW used by bitcoin?



- Everyone needs to agree on next record to be appended

Miners aggregate transactions in blocks (allows for better throughput)

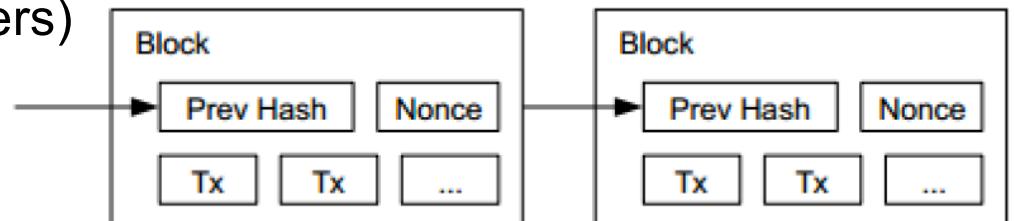


Miners compete to get block appended (winner gets reward, newly minted BTC)



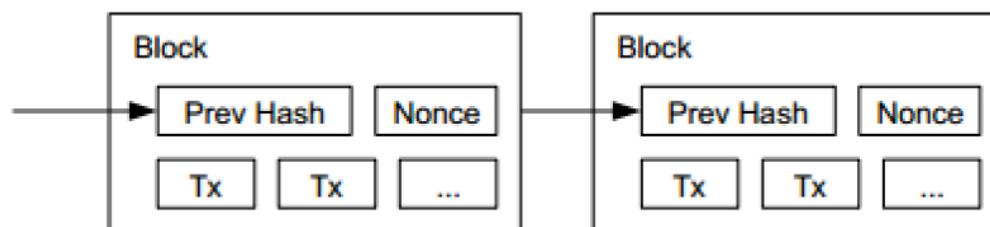
PoW in bitcoin

- Each miner has a transaction pool
 - Set of transactions received over the network
 - Or created by the node itself
- Miners continually try to create a valid block by solving a puzzle:
 $\text{SHA256}(h \parallel T \parallel K \parallel \text{nonce}) < D$
h – hash of last block; T – digest of txns; K – pub. key (owner gets reward); D – difficulty;
 - When solved, broadcast new block to all miners
- A block includes:
 - A subset of transactions, chosen by the miner, in the transaction pool
 - A pointer to the previous block
 - The identity of the miner (among others)
 - The solution to the puzzle

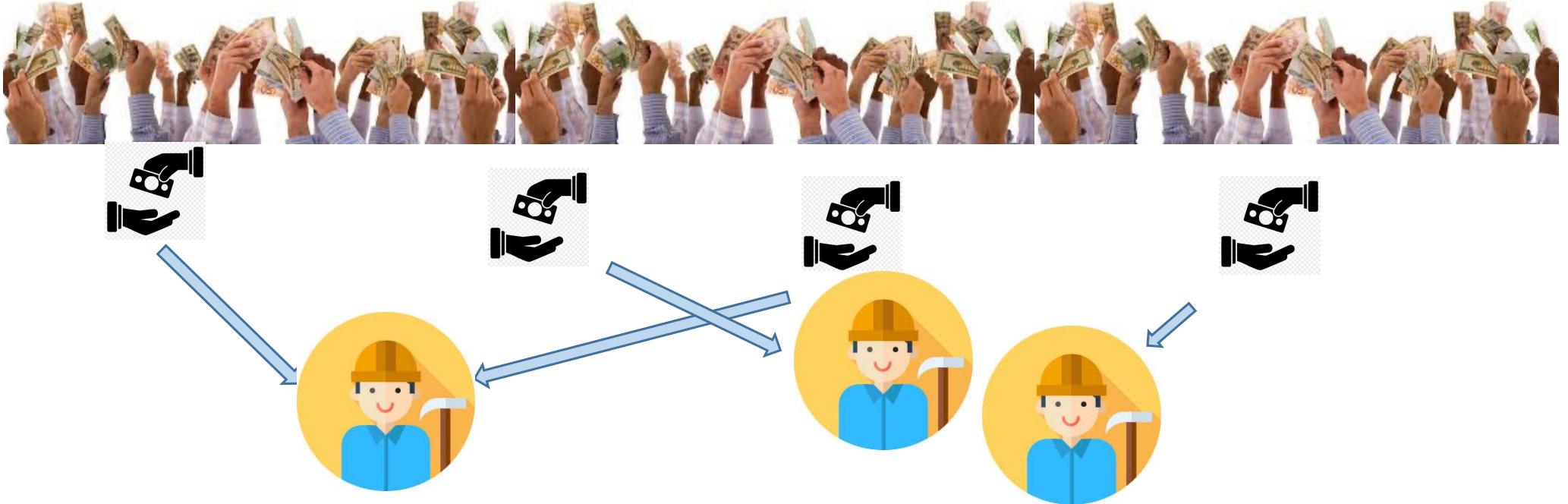


Block dissemination

- Every node keeps a copy of every block of transactions
 - Can be a scalability bottleneck
 - Led to distinction between full nodes and compact nodes
- Upon receiving a new block:
 - Miners validate the block (incl. money not being already spent)
 - If valid, they append it to the local blockchain
 - Start mining the next block with a pointer to the accepted block



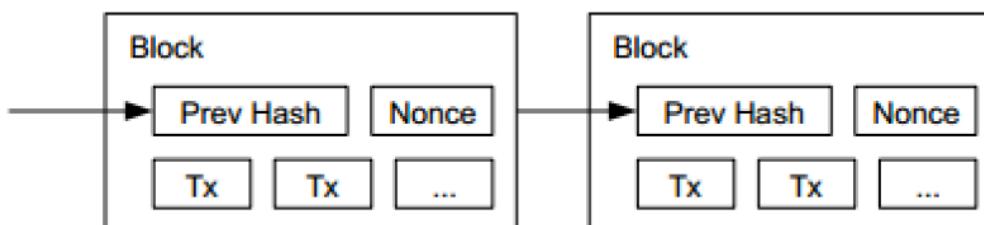
Clients != miners != compact nodes



- Clients submit transactions to miners (broadcast to all miners through p2p layer)
- Miners solve PoW
- Full nodes keep a complete copy of the blockchain
- Compact/lightweight nodes keeps the current state (more on this later) but only a small suffix of the blockchain

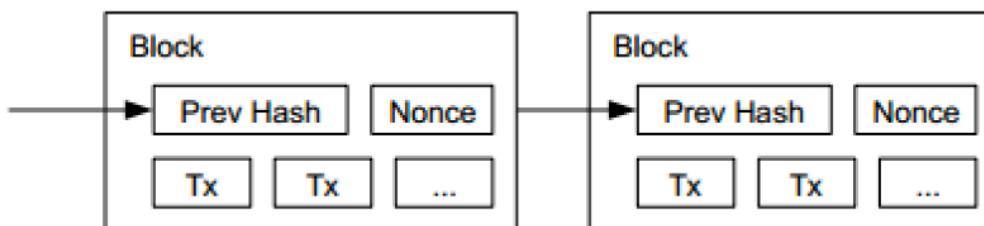
Properties

- Prevents Sybil Attacks
 - Solving the puzzle can effectively be seen as *vote*
 - Puzzle requires computational power – cannot be counterfeit without breaking the hash function
 - Controlling $x\%$ of the CPUs \sim controlling $x\%$ blocks
 - Still, some cryptocurrencies witnessed “**51% attacks**”



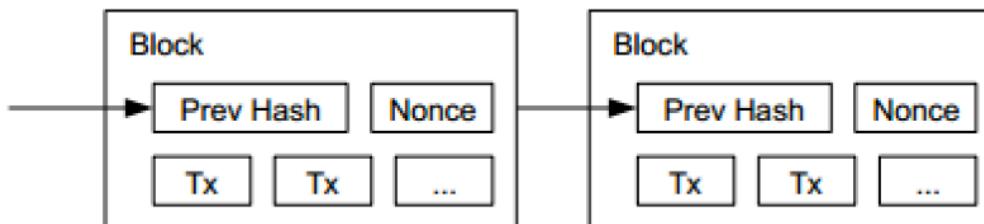
Properties

- Tamperproof
 - Pointer to the previous block
 - Changing a previous block would invalidate the subsequent chain
 - Rewriting a part of the chain requires solving all the associated puzzles
 - The longer the chain one wants to revert, the larger the needed computing power



Properties

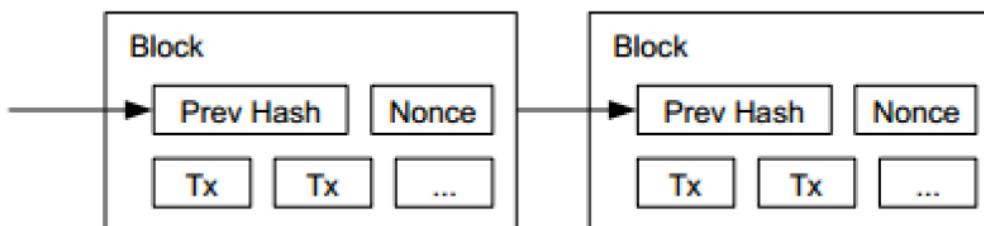
- Provides incentives for miners to participate
 - The miner of a block creates (mines) a new coin
 - Coin is given to the miner
 - Transactions in the block also pay a transaction fee to the miner



Properties

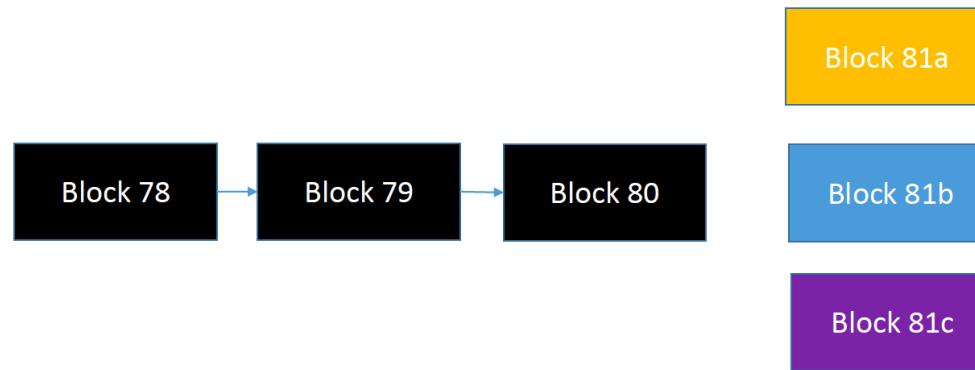
- Generating a block can be seen as a random Poisson process
 - Puzzle difficulty is automatically adjusted to meet a target rate of new blocks per unit of time
 - Bitcoin ~1 block every 10 minutes
 - Ethereum ~1 block every 20 seconds
 - Required to avoid frequent conflicts (multiple winners), but problematic as system becomes more widely used

Very limited throughput!



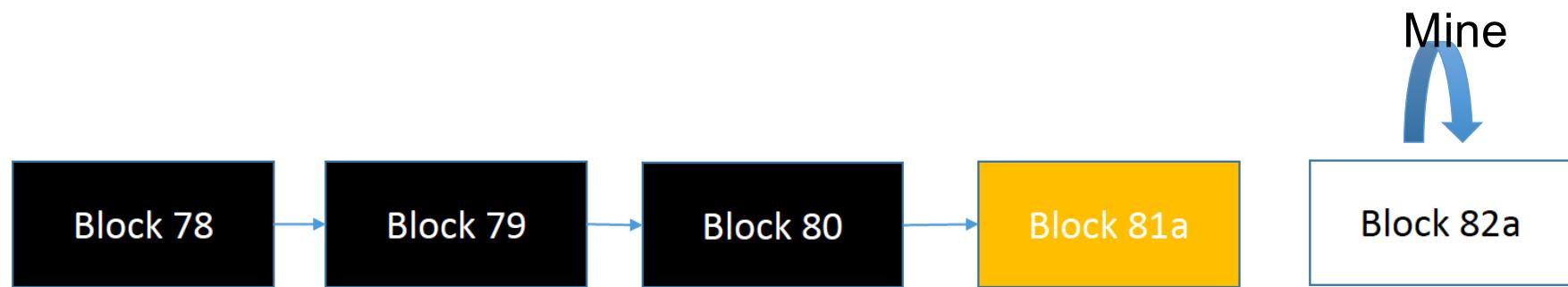
A fork in the road...

- What if a miner receives several valid new blocks to be appended
 - Which one to choose?
 - Miners can choose any block they want
- Resulting in miners trying to mine in parallel chains
 - Forks!



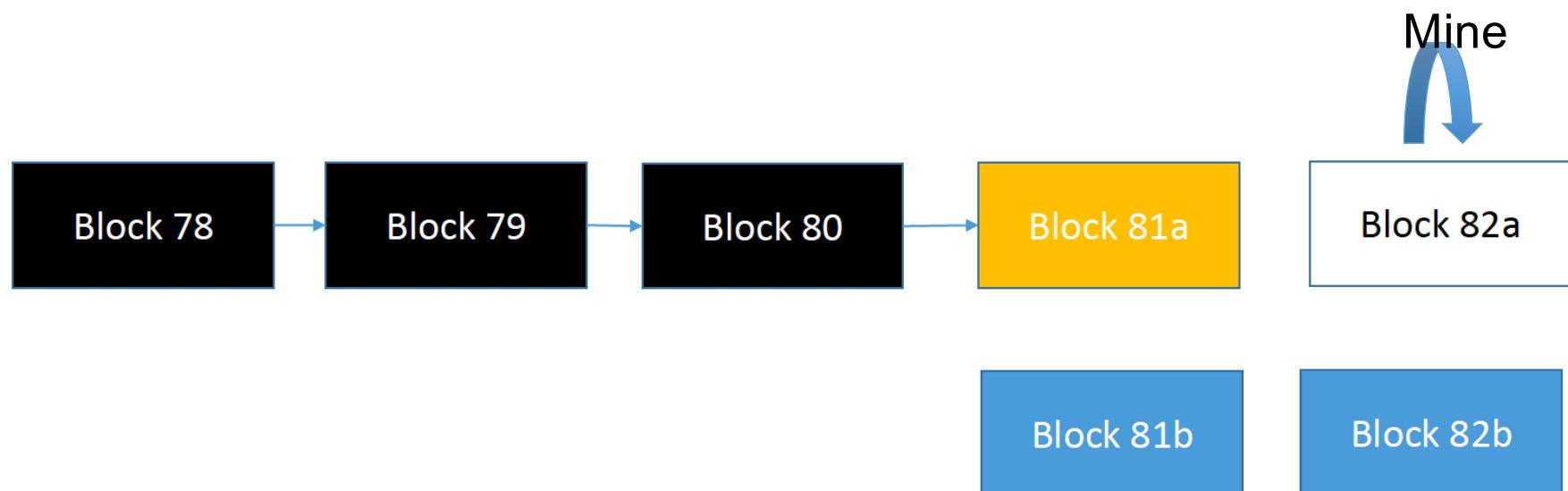
A fork in the road...

- Start mining on block 82a



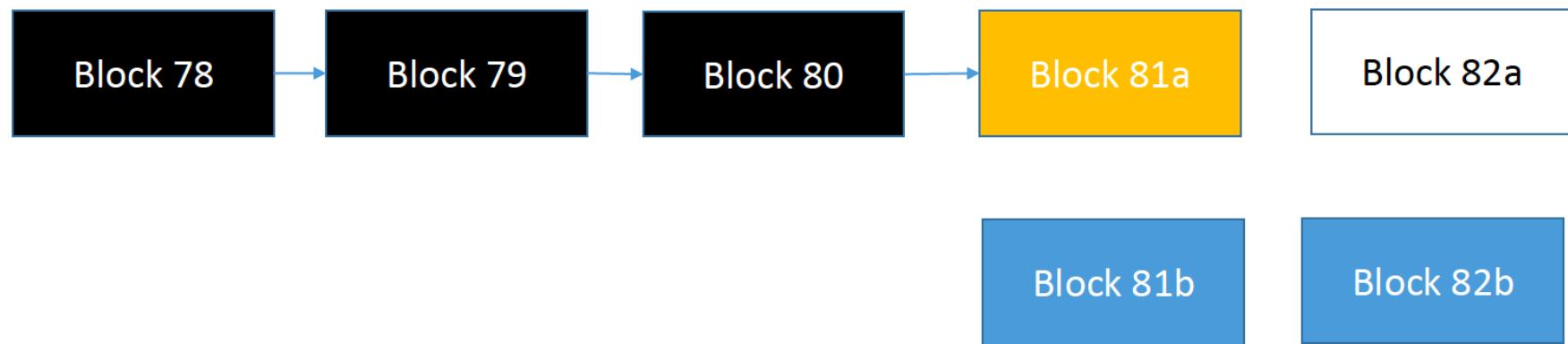
A fork in the road...

- What if blocks 81b and 82b appear?



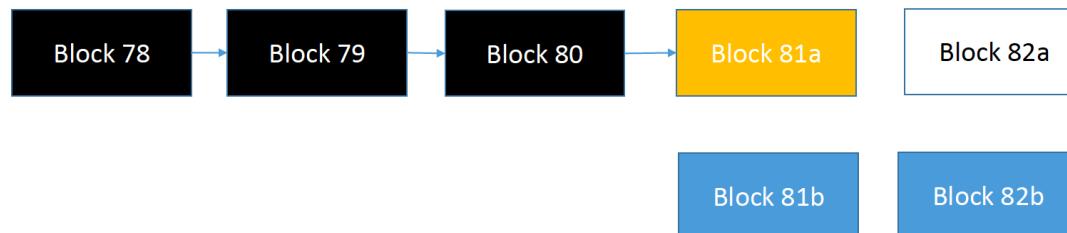
Longest chain rule

- Try to mine new Block 83
- Always build on the longest chain, i.e., the one with the most PoW
- Reflects the will of honest miners, forcing dishonest ones to out-compute all the honest miners



Problem: lack of “finality”

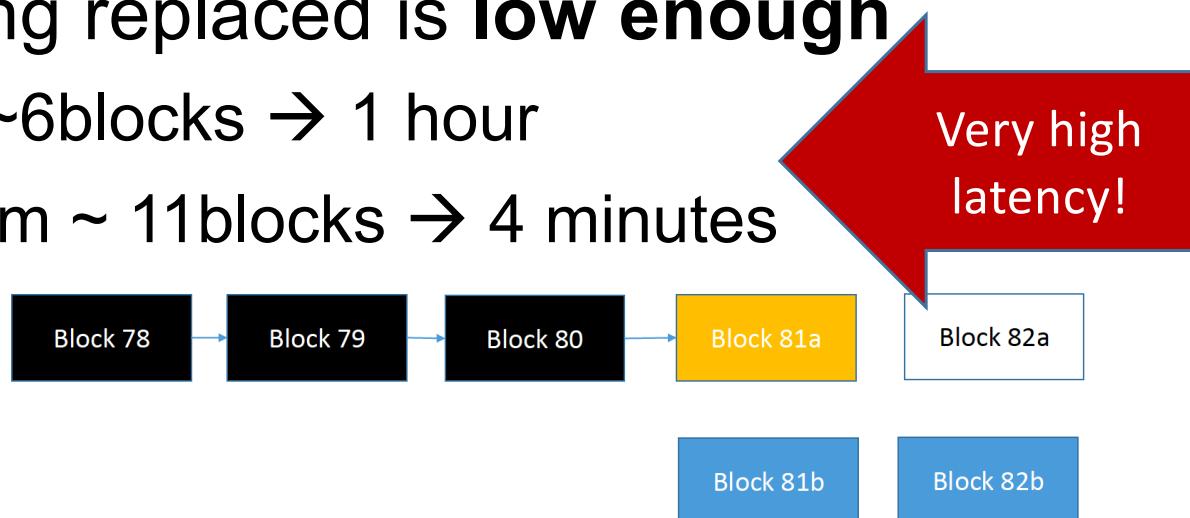
- Each miner picks the transactions to be included in a block
 - Thus, transactions in Blocks 81a and 81b differ
- Suppose you have an e-commerce site
 - You see a payment transaction Tx1 in Block 81a
 - Ship the goods
 - Tx1 no longer appears in either Block 81b or 82b
 - What to do?



Embrace the probabilistic nature

Probabilistic solution

- Need to wait for the tx's block to be deep *enough* in the chain
- *Enough* meaning the **probability** of a given chain suffix being replaced is **low enough**
 - Bitcoin ~6blocks → 1 hour
 - Ethereum ~ 11blocks → 4 minutes



Double-spending attacks

Can an attacker deliberately achieve the following:

1. Approve a transaction tx that moves coin to merchant
2. Merchant waits for tx's block to be deep enough and ships goods
3. Attacker releases a forked chain of blocks, including moving the same coin to another merchant
4. Forked chain of blocks is longer than first chain

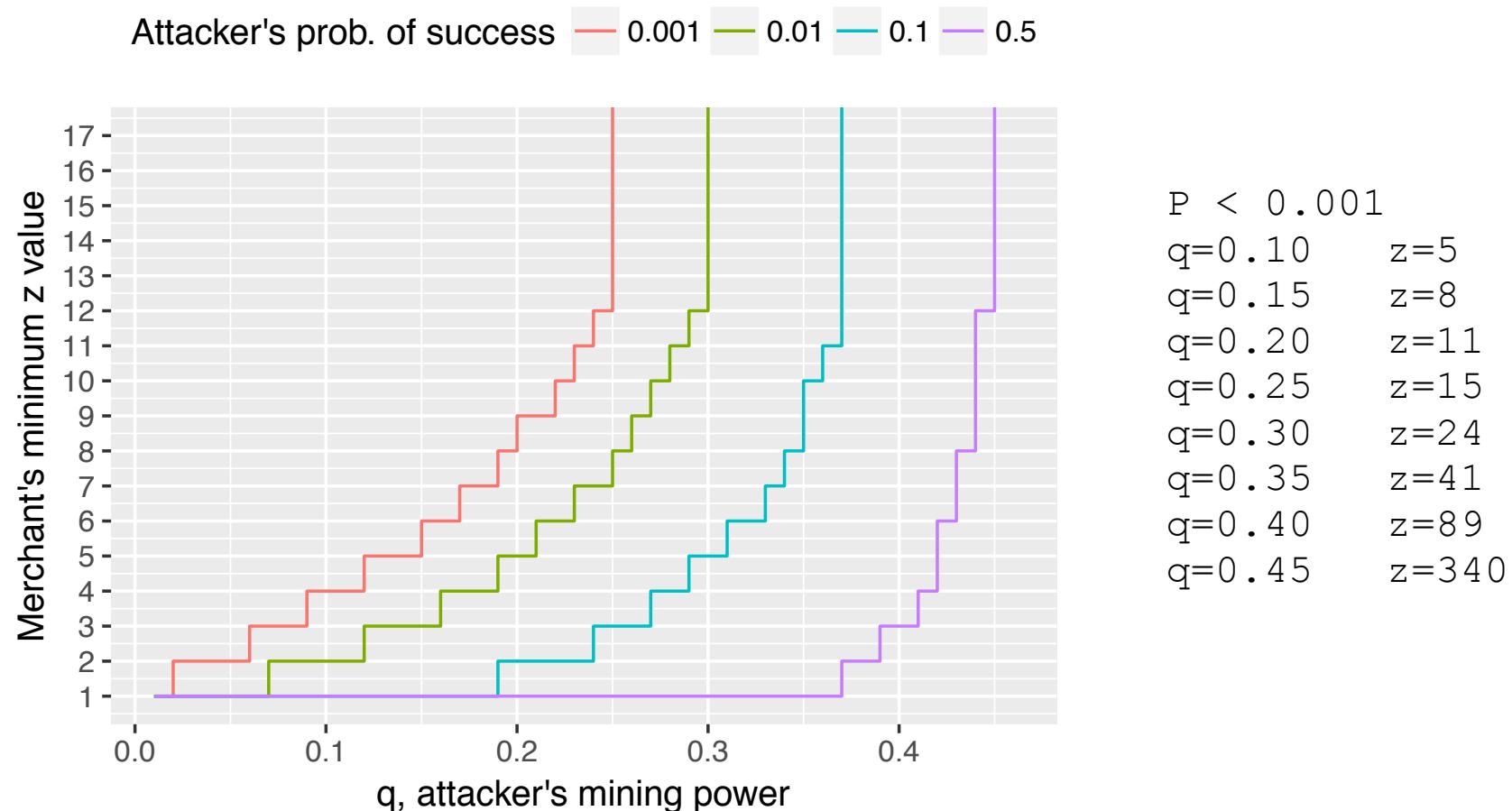
Reverting a chain of z blocks

- Assume an attacker tries to remove one of his txs that is stored in a block at depth z
- Equivalent to Binomial Random Walk (*):
 - p = prob. some honest node finds the next block
 - q = prob. the attacker node finds the next block
- Probability, $Q(z)$, that attacker will ever catch up from z blocks behind:
 - 1, if $p \leq q$
 - $(q/p)^z$, if $p > q$

Any attacker with $>50\%$ than the total computational resources can eventually catch up even if starting from an arbitrarily large gap (z)

(*) More details in: An Explanation of Nakamoto's Analysis of Double-spend Attacks, Ozisik and Levine

How safe is Blockchain?



Sources: Bitcoin: A Peer-to-Peer Electronic Cash System, Satoshi Nakamoto;
An Explanation of Nakamoto's Analysis of Double-spend Attacks, Ozisik and Levine

Rental attack

- Rent VMs to launch a 51% attack
- Overly expensive for Bitcoin and Ethereum
 - but turns out to be surprisingly cheap for other coins

More sophisticated form (Finney attack)

- Suppose merchant accepts unconfirmed transactions
- Attacker successfully pre-mines a block including a transaction that sends some of his coins back to self, without broadcasting it
- Attacker uses same coins for purchase at this merchant
- After the merchant accepts them, attacker broadcasts his block (overriding the payment)

Higher bar for mining → mining pools

- CPU → GPU → ASIC
 - Gets more effective and sophisticated as currency value increases
- As mining got more difficult, mining pools emerged
 - increase chances, hedge risk, split the reward among pool members
 - Strong incentive to participate
- Mining pools grew significantly (possibly reaching more than 51% of mining power)
 - Implies that bitcoin may not be that decentralized after all

Possible attack from mining pool: Feather-forking attack

- Suppose mining pool controls, e.g., only 20%, and wants to censor Alice
- Mining pool announces it will refuse to mine chains that include Alice in last N blocks
- Creates a disincentive for every miner to work on blocks that include Alice's txns, as these may fail to be included in the longest chain

Selfish mining

- Pool nodes withhold newly found blocks, continuing to work on their private chain
 - i.e., they create their own private fork
- When public chain catches up, publish private chain
- Creates an incentive to join the selfish mining pool (more profitable than following the protocol)

Let others do the mining?

- Malware can conduct mining without the owner of the machine noticing it
- But even without using malware, just by visiting a website, “drive-by mining” is possible (using JavaScript or WebAssembly)

MineSweeper: An In-depth Look into Drive-by Cryptocurrency Mining and Its Defense

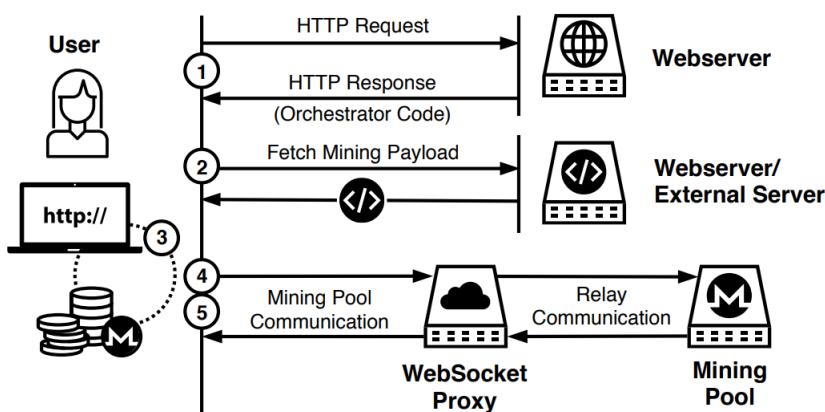


Table 1: Summary of our dataset and key findings.

Crawling period	March 12, 2018 – March 19, 2018
# of crawled websites	991,513
# of drive-by mining websites	1,735 (0.18%)
# of drive-by mining services	28
# of drive-by mining campaigns	20
# of websites in biggest campaign	139
Estimated overall profit	US\$ 188,878.84
Most profitable/biggest campaign	US\$ 31,060.80
Most profitable website	US\$ 17,166.97

Privacy aspects

- How was privacy of buyers and sellers handled in traditional commercial transactions?
- How is it handled in bitcoin?

Last but not least

The New York Times

Bitcoin Uses More Electricity Than Many Countries. How Is That Possible?

In 2009, you could mine one Bitcoin using a setup like this in your living room.

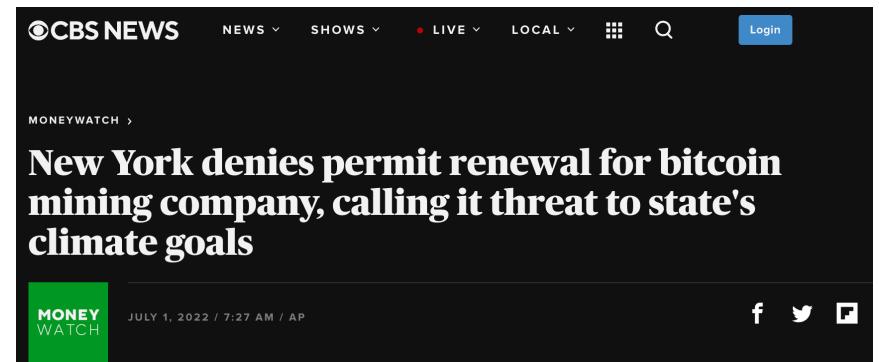
Today, you'd need a room full of specialized machines, each costing thousands of dollars.

The dark side of PoW

- PoW for cryptocurrencies consumes more energy than Argentina, Bitcoin is comparable to Greece
- ~2X Google, Meta, Microsoft, Amazon, Apple (combined)
 - While their value to society is incomparable
- Bitcoin emits 65 megatons of CO₂ annually
- Single bitcoin transaction emits almost 1 ton of CO₂ (6 orders of magnitude more than a credit card transaction)

Becoming less and less green

- Until recently, 75% of mining was in China, due to cheap electricity and hardware
 - Government banned it in 2021
- Nowadays 1/3 of mining is in the US, 2nd place is Kazakhstan
 - Shifted from hydropower to gas and coal
 - Reviving old, polluting power plants for mining
 - Regulation is now starting to kick in



Recall consensus specification

- **Termination:** Correct processes eventually decide.
- **Validity:** Any value decided is a value proposed
- **Integrity:** No correct process decides twice.
- **Agreement:** No two correct processes decide differently.

PoW blockchains vs consensus

- Which properties does PoW consensus satisfy?
- Consider:
 - **decide** as accepting a block at the blockchain head
 - **propose** as submitting a transaction to be eventually included in a blockchain block

PoW blockchains vs consensus

- **Termination:** Correct processes eventually decide.
 - Arguably, eventually a new block is appended to the chain
 - How fast depends on the solvability of the crypto-puzzle

PoW blockchains vs consensus

- **Integrity:** No correct process decides twice.
 - No. The accepted block at a given weight might change several times due to forks

PoW blockchains vs consensus

- **Agreement:** No two correct processes decide differently.
 - No. The accepted block at a given weight might change several times due to forks

PoW blockchains vs consensus

- **Validity:** any value decided is a value proposed
 - No, if Byzantine processes can generate transactions that were never proposed
- **Weak Validity:** if some processes are faulty, an arbitrary value may be decided
 - OK, although block structure cannot be completely arbitrary:
 - e.g., blocks containing illegal transactions are rejected
- **Strong Validity:** a correct process may only decide a value that was proposed by some correct process or the special value \square .
 - No, if Byz. processes can generate transactions that were never proposed

PoW blockchains vs. consensus

- Trade-off between liveness and safety
- Classical consensus emphasizes safety
 - Agreement is always respected but might not terminate
- PoW consensus emphasizes liveness
 - The system always makes progress (i.e., decides on *something*) but safety is at stake:
 - Guarantees are only of **probabilistic** nature
 - FLP is not contradicted... why?

Alternative Approaches

Proof-of-Stake

- Use stake (coins) in the system as the non-counterfeitable resource
- Users *vote* with their coins in the system rather than with CPU power
- Chance to select the next block proportional to the stake size
- Problems
 - Rich get richer
 - Proved to converge only in specific scenarios
- Ethereum moved to this model (more on this next lecture)

Alternative Approaches

Proof-of-Space

- Use disk space as the non-counterfeitable resource
- Users pre-generate a very large data structure in disk
- Subsequently answer queries for random subsets of the data
- Problems
 - Requires additional techniques – small PoW, PoET
- Examples: SpaceMint; Chia

Alternative Approaches

Proof-of-Elapsed-Time

- Relies on a Trusted Execution Environment to elect a leader that will choose the next block
 - Users “sleep” for a random period
 - User who wakes up first can select the next block to be include in the blockchain
 - limits the rate at which malicious nodes can generate blocks
- Problems
 - Need to rely on trusted hardware, e.g., Intel SGX, ARM Trust
 - Which has known vulnerabilities

Alternative Approaches

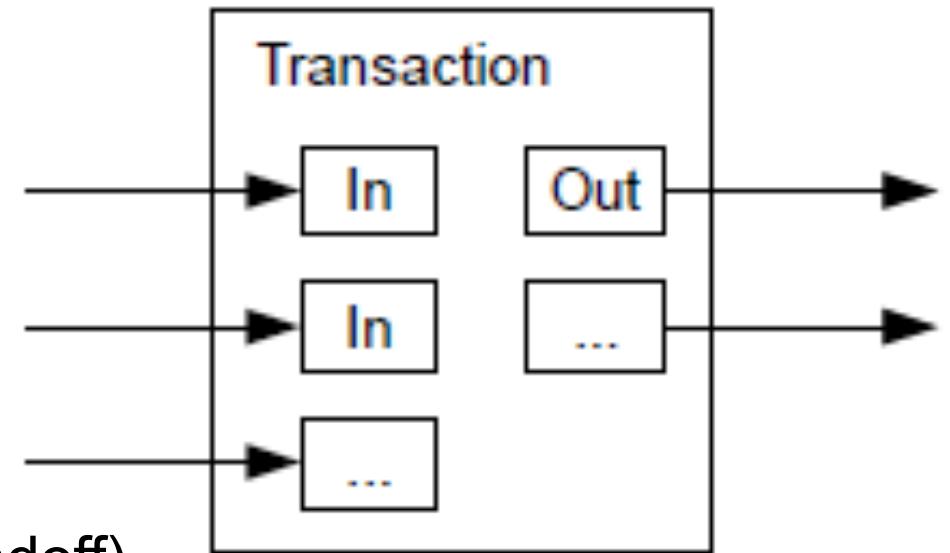
Committee based

- Elect a committee among the set of participants
 - Using for instance PoW
- Committee relies on classical consensus to select the next block
- Problems
 - Committee members can behave arbitrarily
 - Committee members prone to attacks
- Examples: Solida, Algorand

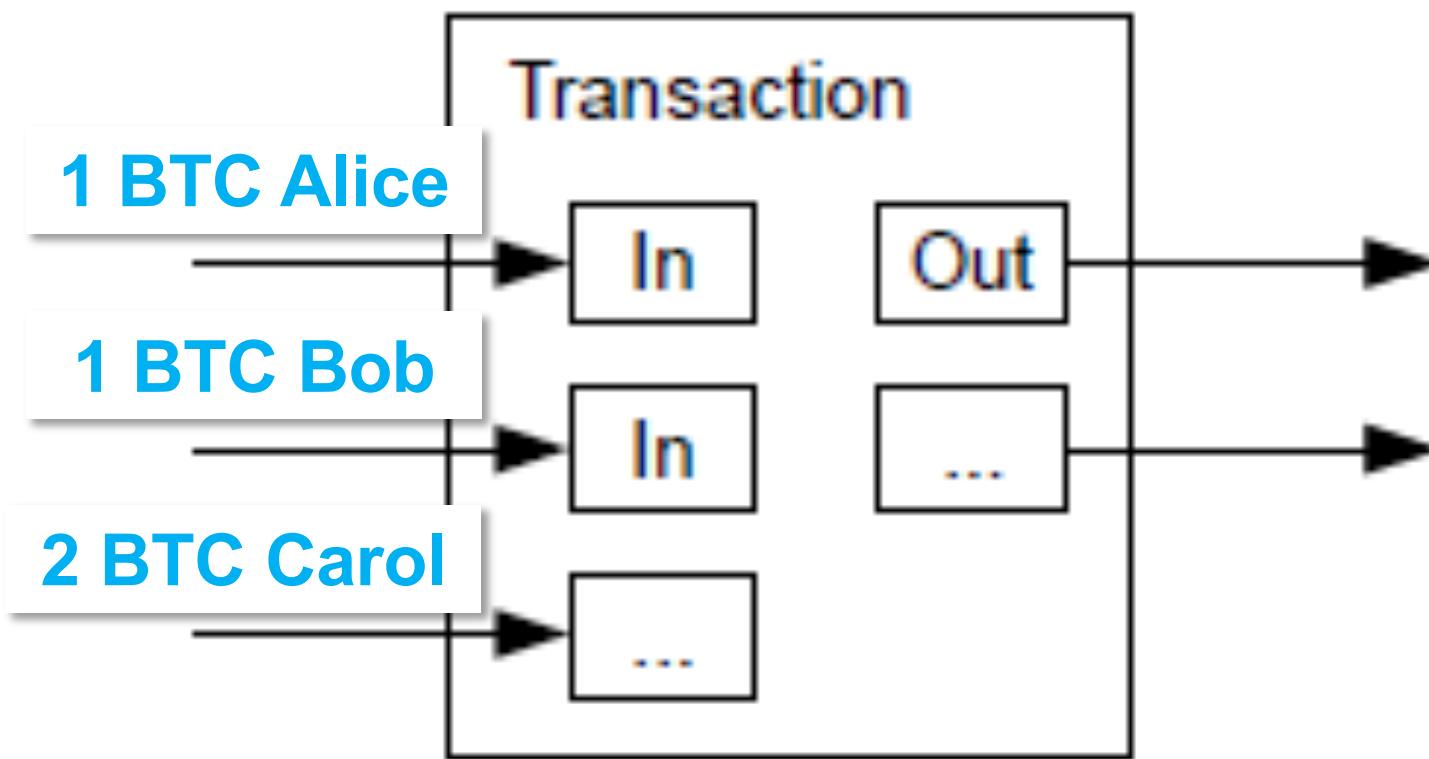
Bitcoin data structures

Each block in the blockchain is a set of transactions

- Transactions can have multiple inputs and ≤ 2 outputs
- Possible to specify a set of deterministic operations, with conditions on transfer (smart contracts, later this week)
- Each input describes debit
 - Sender → signs handoff of amount (signed with private key)
- Each output describes credit
 - Recipient → public key of entity (corresponding to private key that later can sign the subsequent handoff)



Bitcoin UTXO Model



UTXO model

- Blockchain stores all transaction since beginning of bitcoin (hundreds of GBs of information, increases as more txns occur)
- Additionally, keeps track of outputs that haven't been spent yet (about 4 GB of information, increases as more users join)
 - Called “unspent transaction output” (UTXO)
 - “Soft state” → can be reconstructed and verified from blockchain
 - Alternative would be to store account balances (has pros and cons)
- New transactions remove UTXOs from this set, and create new ones that are added to the set

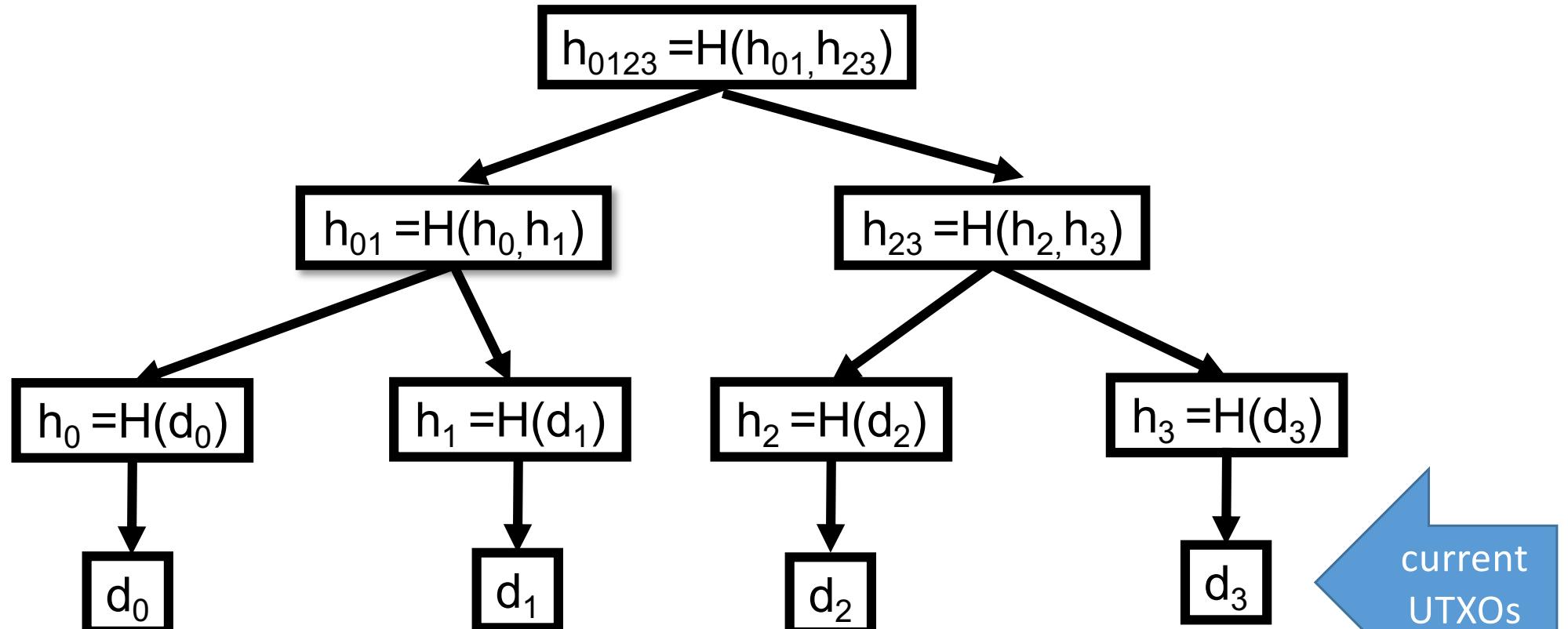
Bitcoin scalability and Utreexo

- Resources required for running a full node are a problem (barrier to entering the system)
- Initial block download can take 6 hours (SSD) to 1 week (HDD), but this is only done once per node
- Continuously maintaining a (growing) set of all UTXOs is more of a concern
- Light clients → do not store state nor validate transactions
 - At the cost of weaker security properties than full nodes
- Led to the proposal of Utreexo
 - Make it easier to run full nodes
 - New set of data structures, and new type of node (compact state node)

Compact state nodes using Utreexo

- Observation: most UTXOs aren't needed for years (from creation until being spent)
 - No need to store all the UTXOs
- Store only a summary (in crypto terms, an accumulator)
- Require transaction issuer to send additional information (proofs) to verify transactions
- Trades storage requirements for bandwidth (mostly) and CPU (not much)

How to compute a summary of UTXOs? (simplified)



- Data structure called a Merkle tree
- Compact nodes only store root of the tree

Upon receiving a UTXO, how to check it is part of the current set?

- Transaction issuer must send proof
- E.g., proof that d_2 is in the set of UTXOs?
- “ d_2, h_3, h_{01} ”
- From this info, receiver computes h_2, h_{23}, h_{0123}
- Then checks whether h_{0123} matches the currently held root of the tree
- (Tree maintenance after adds and deletes + supporting legacy transactions gets a bit more intricate – see paper for details)

Summary

- Classical consensus and permissionless blockchains address a different set of requirements
 - Safety vs Liveness tradeoff
- Many open challenges
 - Transaction latency and throughput
 - Application safety (correctness, censorship, etc)
 - Energy concerns
 - and many others

Further Reading

- S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System
- A. Ozisik, B. Levine. An Explanation of Nakamoto's Analysis of Double-spend Attacks. arXiv:1701.03977
- I. Eyal, E. G. Sirer. Majority is not Enough: Bitcoin Mining is Vulnerable. arXiv:1311.0243
- R Konoth, E Vineti, V Moonsamy, M Lindorfer, C Kruegel, H Bos, G Vigna. MineSweeper: An In-depth Look into Drive-by Cryptocurrency Mining and Its Defense. CCS 2018: 1714-1730
- Y Gilad, R Hemo, S Micali, G Vlachos, N Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. SOSP 2017.
- Thaddeus Dryja. Utreexo: A dynamic hash-based accumulator optimized for the Bitcoin UTXO set. Cryptology ePrint Archive, Paper 2019/611

Acknowledgements

- Rachid Guerraoui, EPFL
- Maurice Herlihy, Brown U.

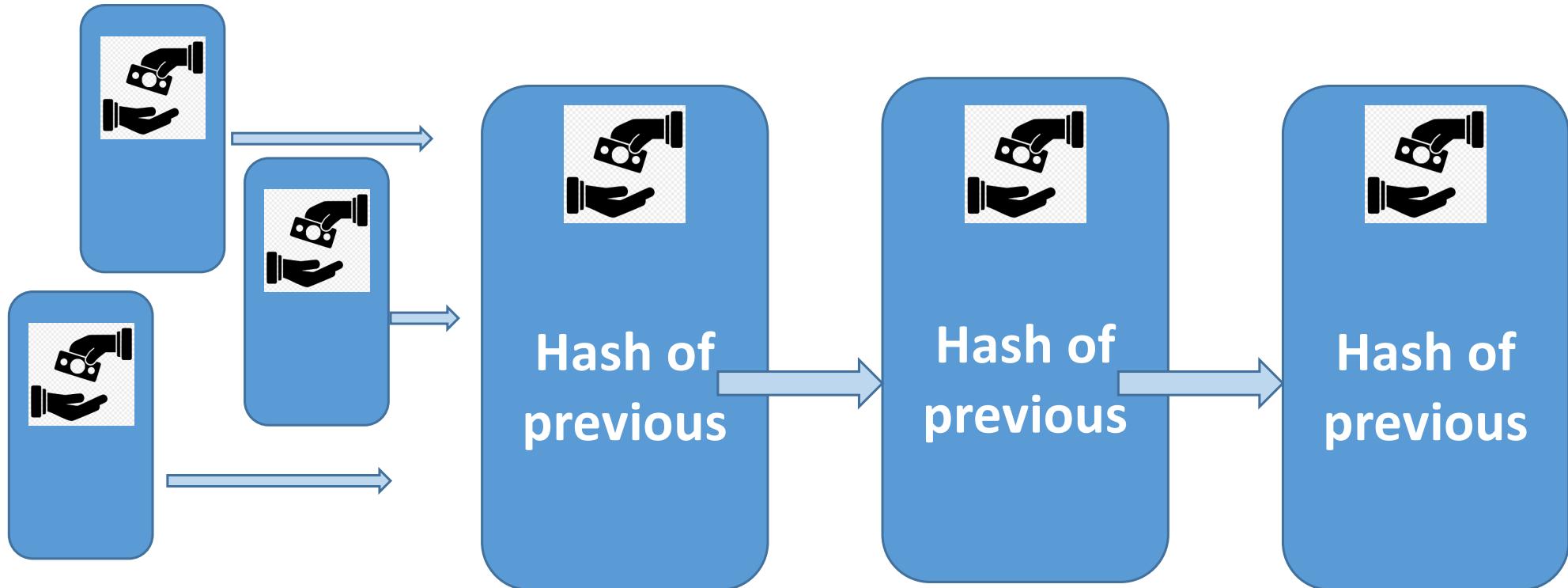
Ethereum Virtual Machine

Highly dependable systems – 2024/25

Lecture 8

Lecturers: Miguel Matos and Paolo Romano

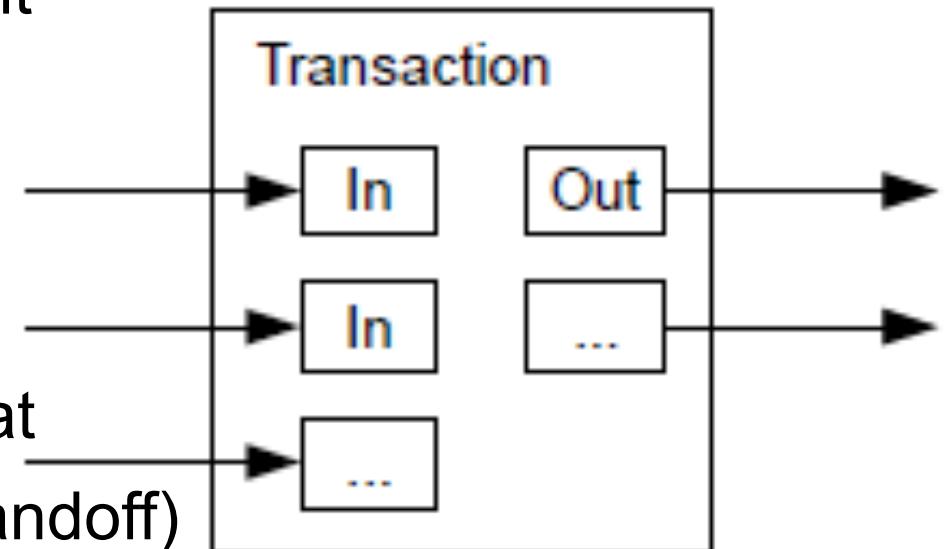
Last lecture: Bitcoin



- PoW → miners compete to append next record
- In some cases may lead to fork → longest chain rule
- Shortcomings: studied several attacks + energy waste

Each block in the blockchain is a set of transactions

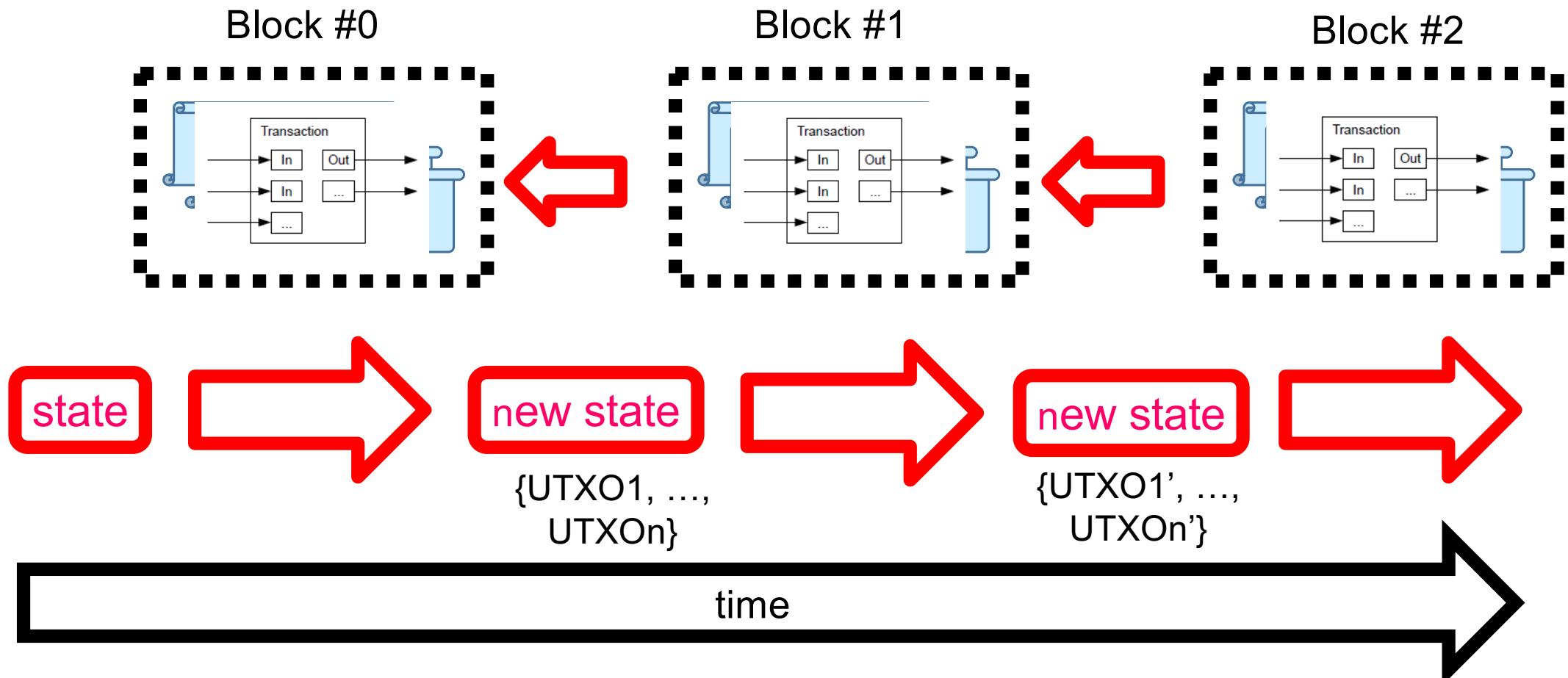
- Transactions can have multiple inputs and ≤ 2 outputs
- Each input describes debit
 - Sender → signs handoff of amount (signed with private key)
- Each output describes credit
 - Recipient → public key of entity (corresponding to private key that later can sign the subsequent handoff)



Bitcoin as a state machine

- State is current set of UTXOs
 - (when a node boots, this is derived from all transactions since inception)
- Transactions cause the following state transition:
 - Remove all UTXOs in “In” from set
 - Add all UTXOs in “Out” to the set

Bitcoin's state machine



Today: Ethereum

- Ethereum is the second largest cryptocurrency after Bitcoin
- Very generic platform, supporting myriad of distributed applications (dapps), with several currencies, NFTs, DAOs, etc.
- Introduced broad support for “smart contracts”
 - Programs that run in the blockchain
 - Ability to read and modify the blockchain state
 - Turing complete
- Adversarial environment without centralized trust
 - Guaranteeing safety is key

Ethereum's layered design

Distributed applications (dapps)

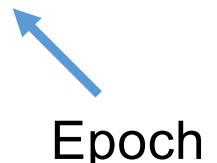
Compute layer (EVM – executes smart contracts)

Consensus layer (PoS based blockchain – “beacon chain”)

Ethereum's PoS blockchain

- Moved from PoW to PoS in December 2020 (“beacon chain”)
- One block every 12 seconds (100-200 txns/block)
- Participants (validators) must stake 32 ETH (~3500 ~1732 EUR today)
 - Large amount, some systems allow for pooling
- Block proposer is chosen at random* among validators
 - Broadcast new block; collects transaction fees
- Remaining validators check correctness and sign correct blocks
 - Block is finalized after a certain threshold of signatures
 - Validators that behave correctly → rewarded (every 32 blocks)
 - Behave incorrectly → part of stake is slashed

* probability is proportional to stake size

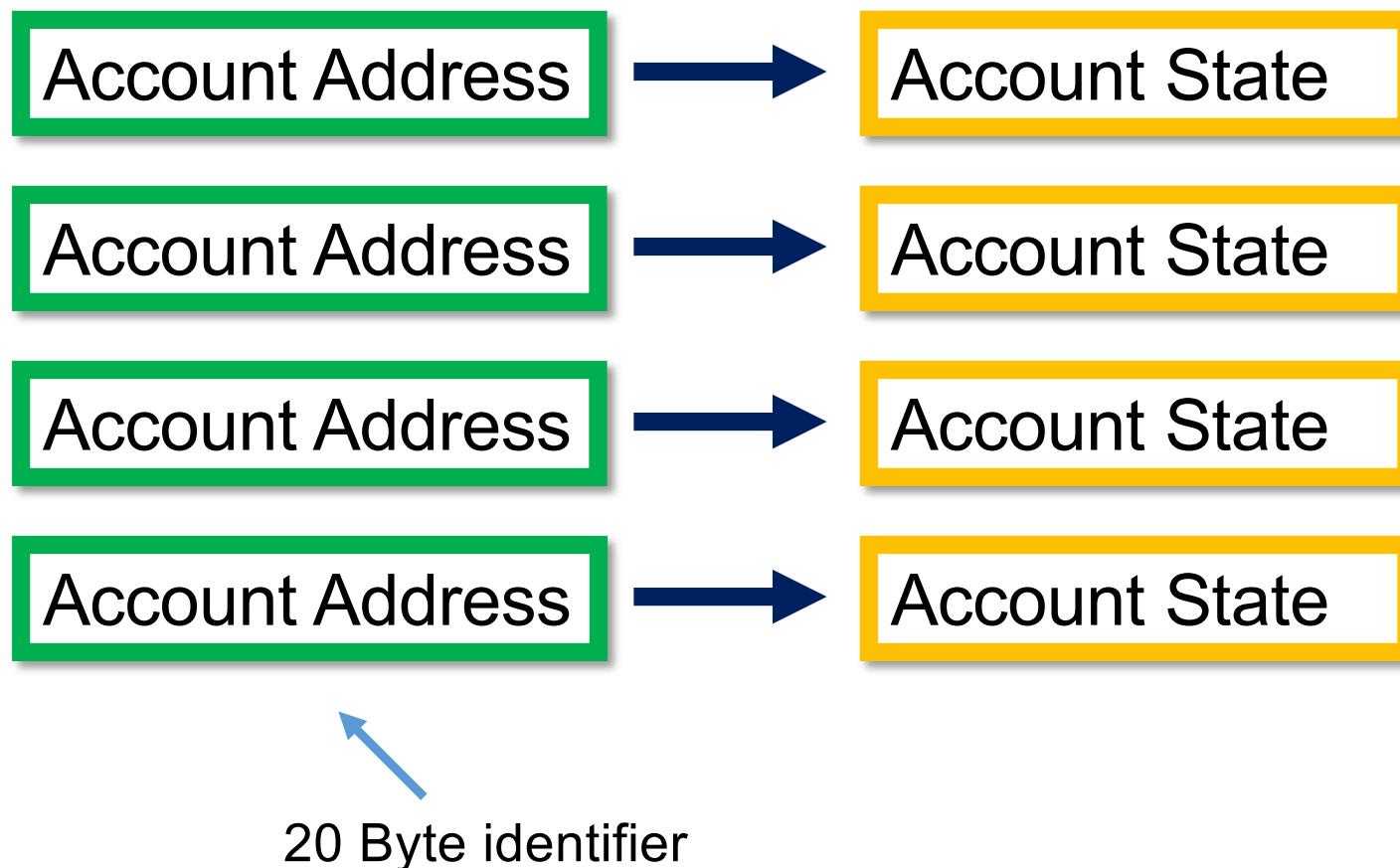


Epoch

Ethereum has an account model

- Instead of UTXOs, the state of Ethereum is a mapping from a set of accounts to their respective state
- At a basic level, the state is the account balance (but it gets more sophisticated)

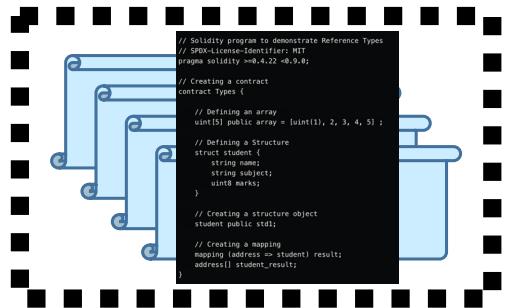
Ethereum State



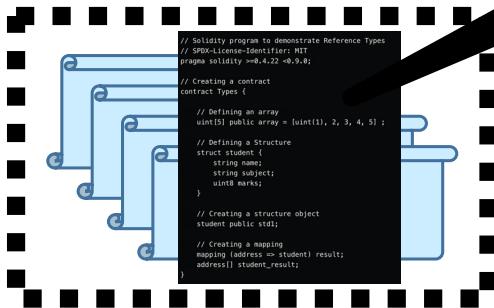
Ethereum's state machine

A state transition executes a whole program

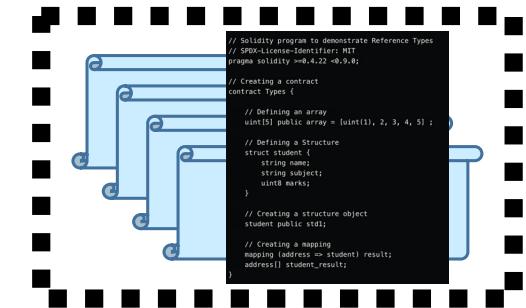
Block #0



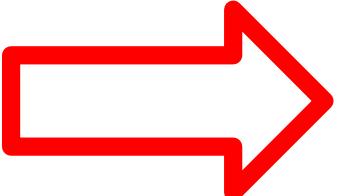
Block #1



Block #2



state



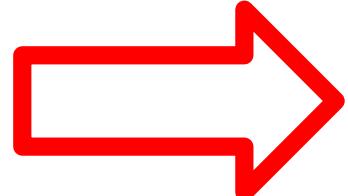
new state

{acct 1 → 20€,
acct 2 → 30€}

time

new state

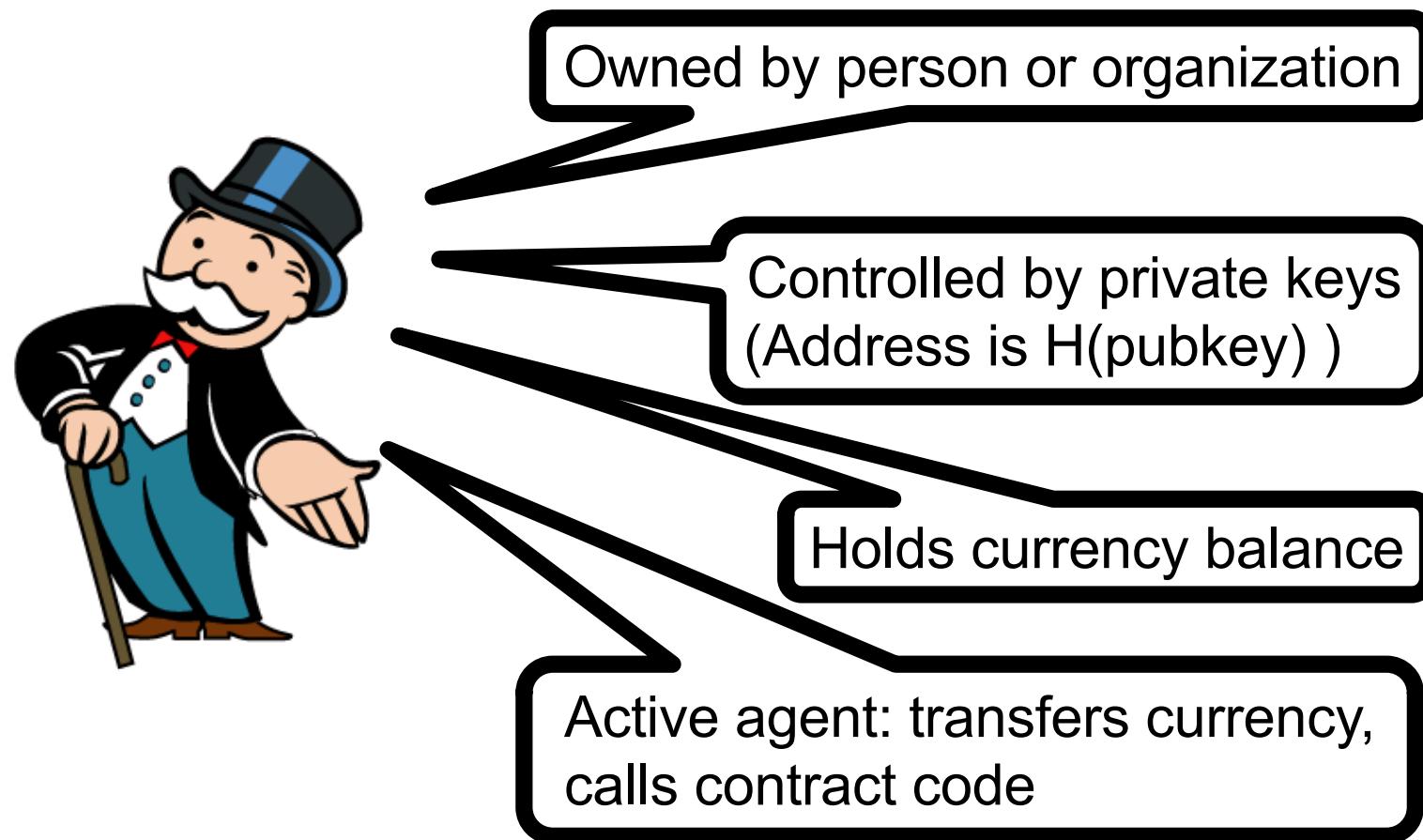
{acct 1 → 25€,
acct 2 → 25€}



Two types of accounts in Ethereum

- External (owned accounts)
- Contract

External Account (“owned”)



External Account (“owned”)

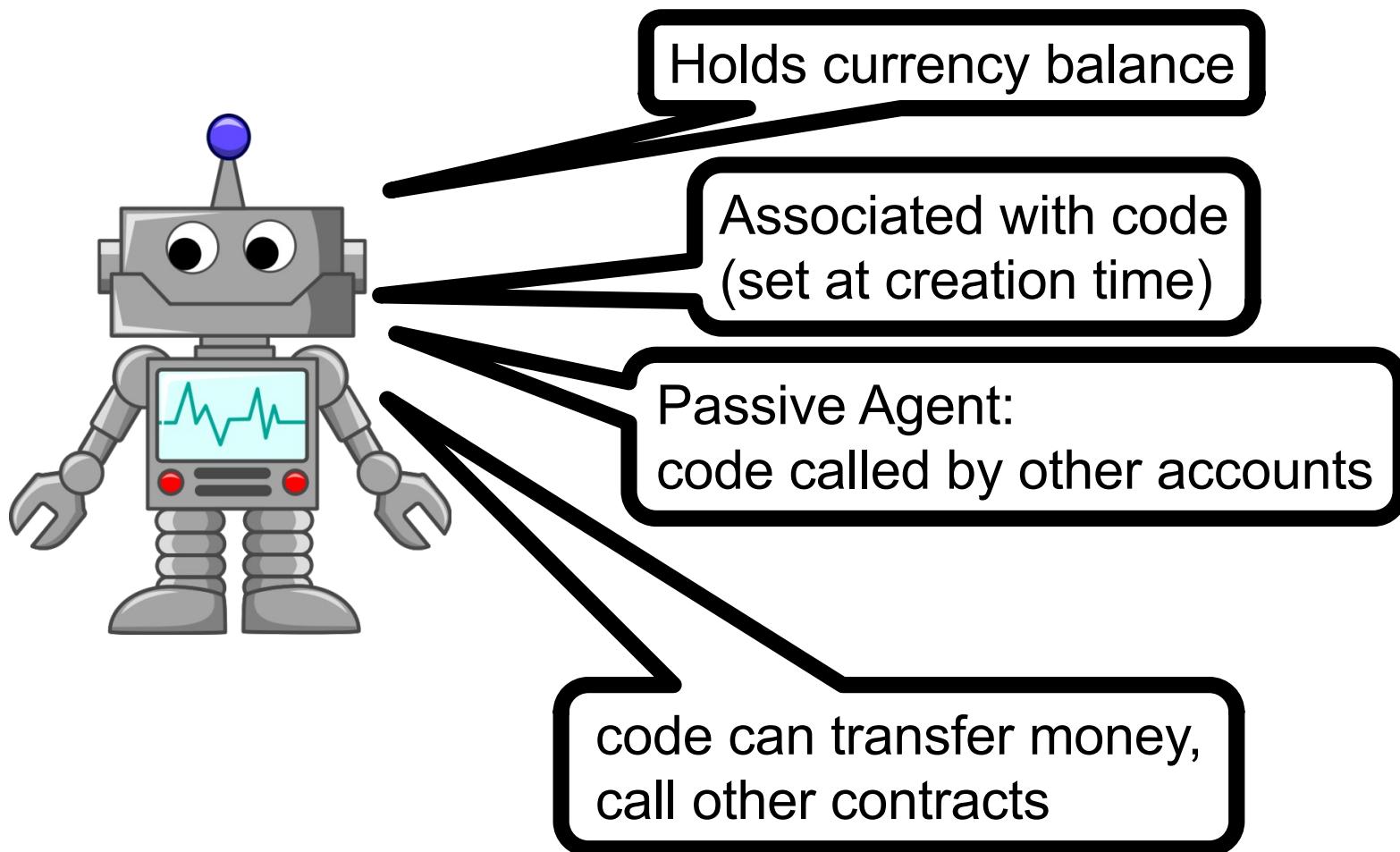


Address

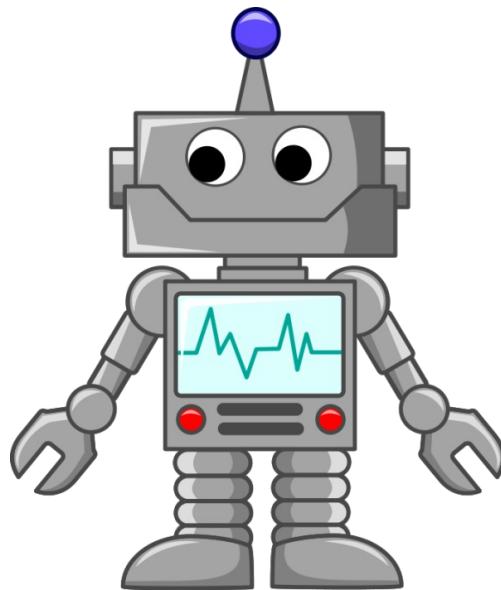


balance

Contract Account



Contract Account



Address



balance
code
storage

Array of
32-byte cells

Transaction Creation

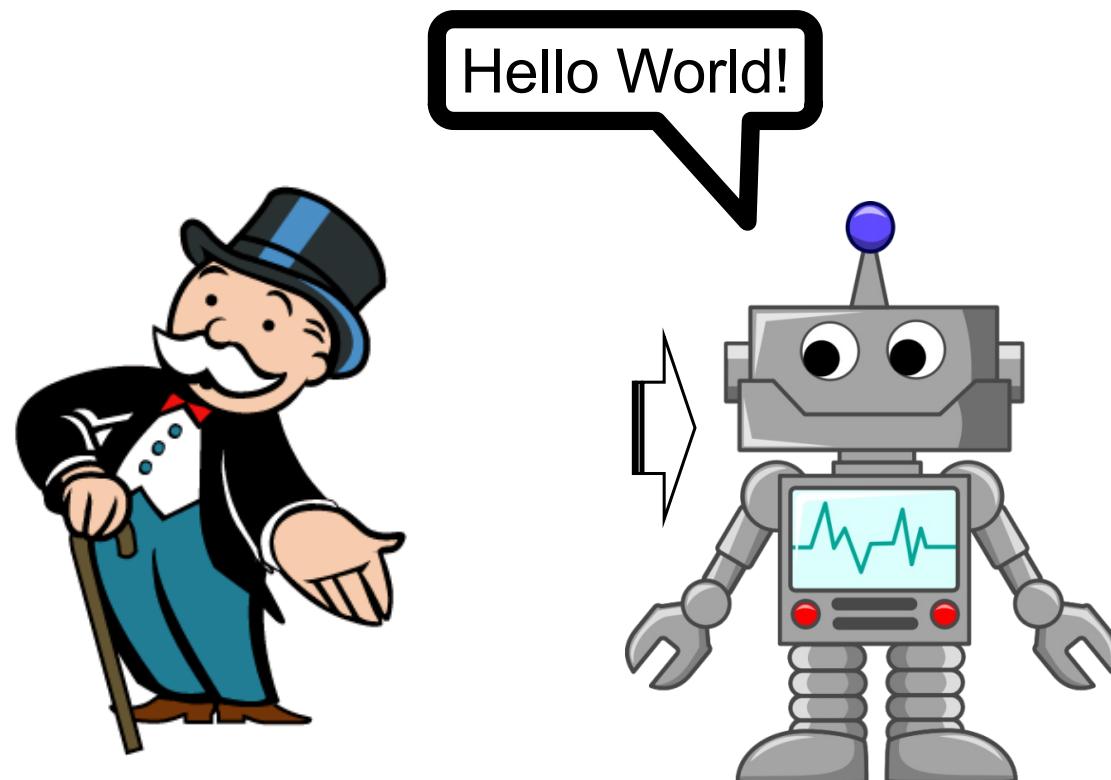


Submitted by external party

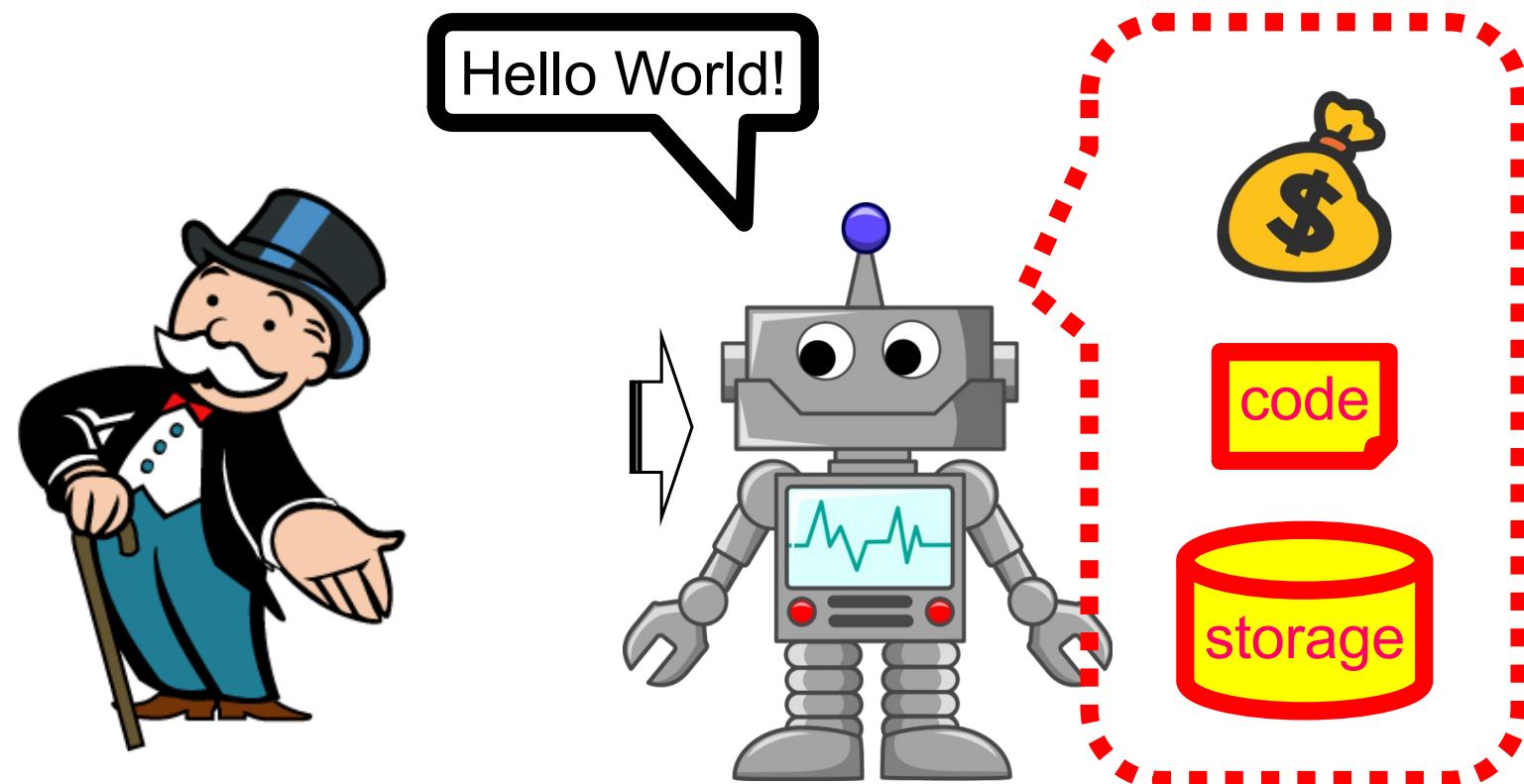
Contract Creation Transaction



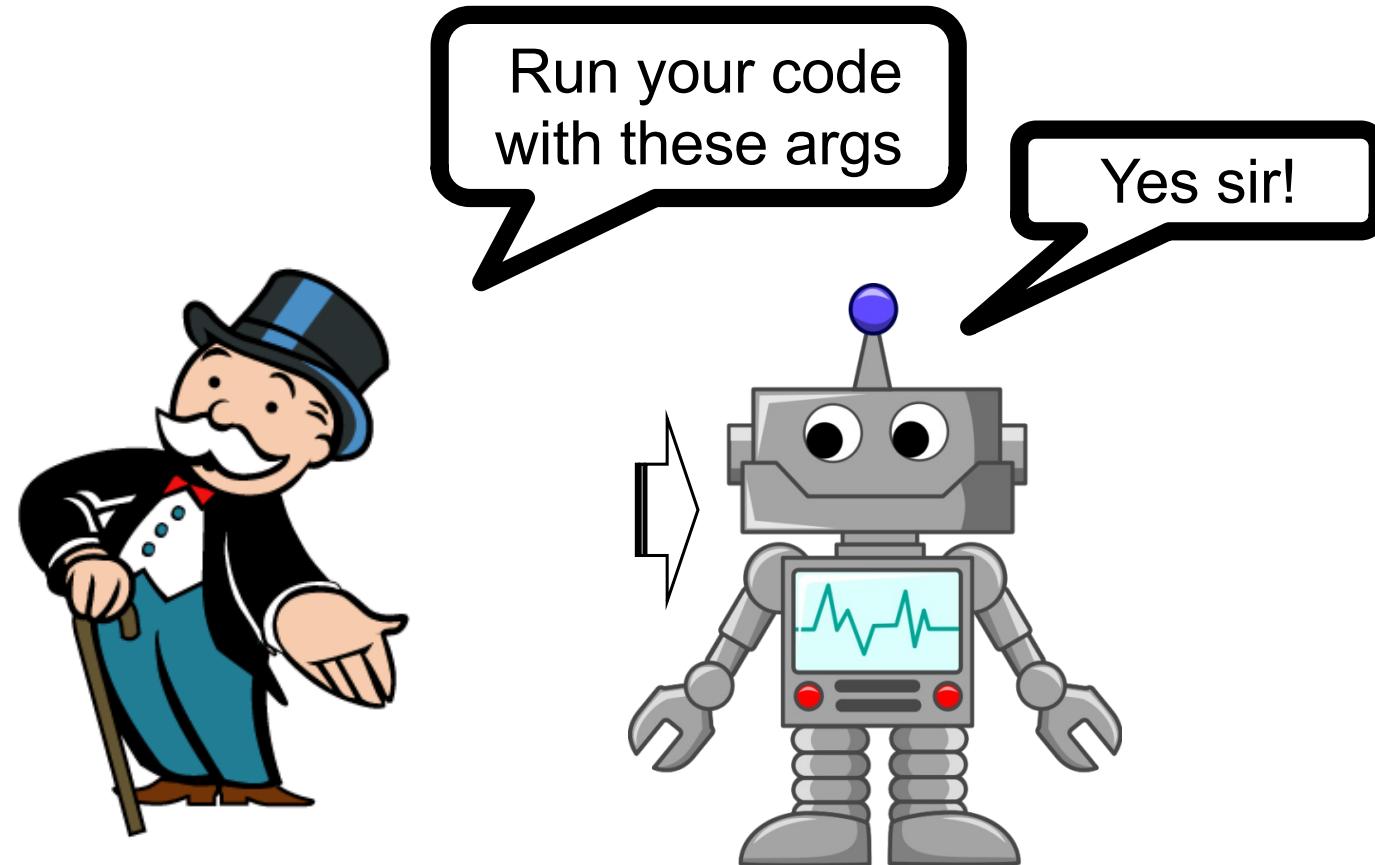
Contract Creation Transaction



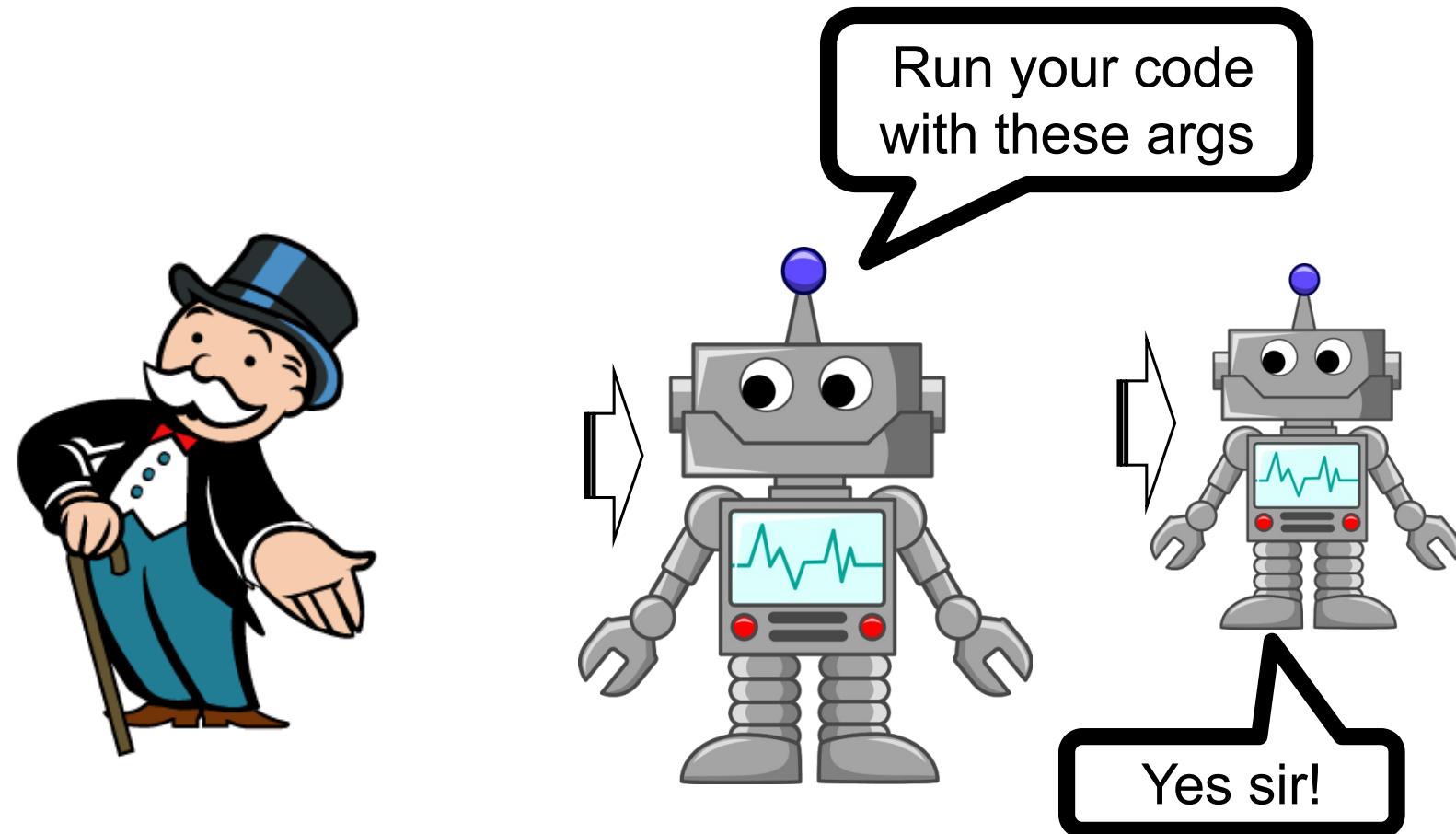
Contract Creation Transaction



Message Call Transaction

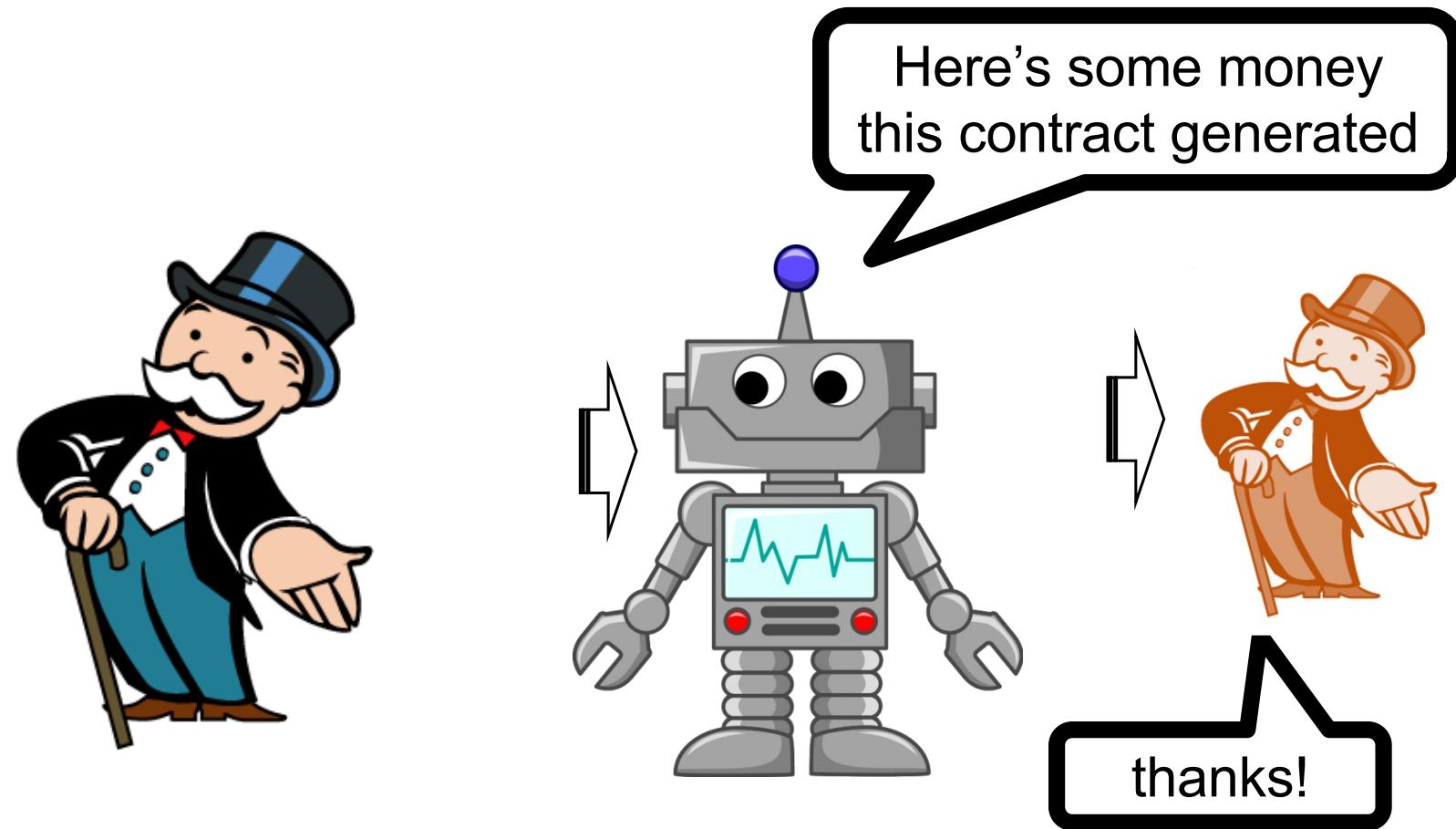


Message Call Transaction



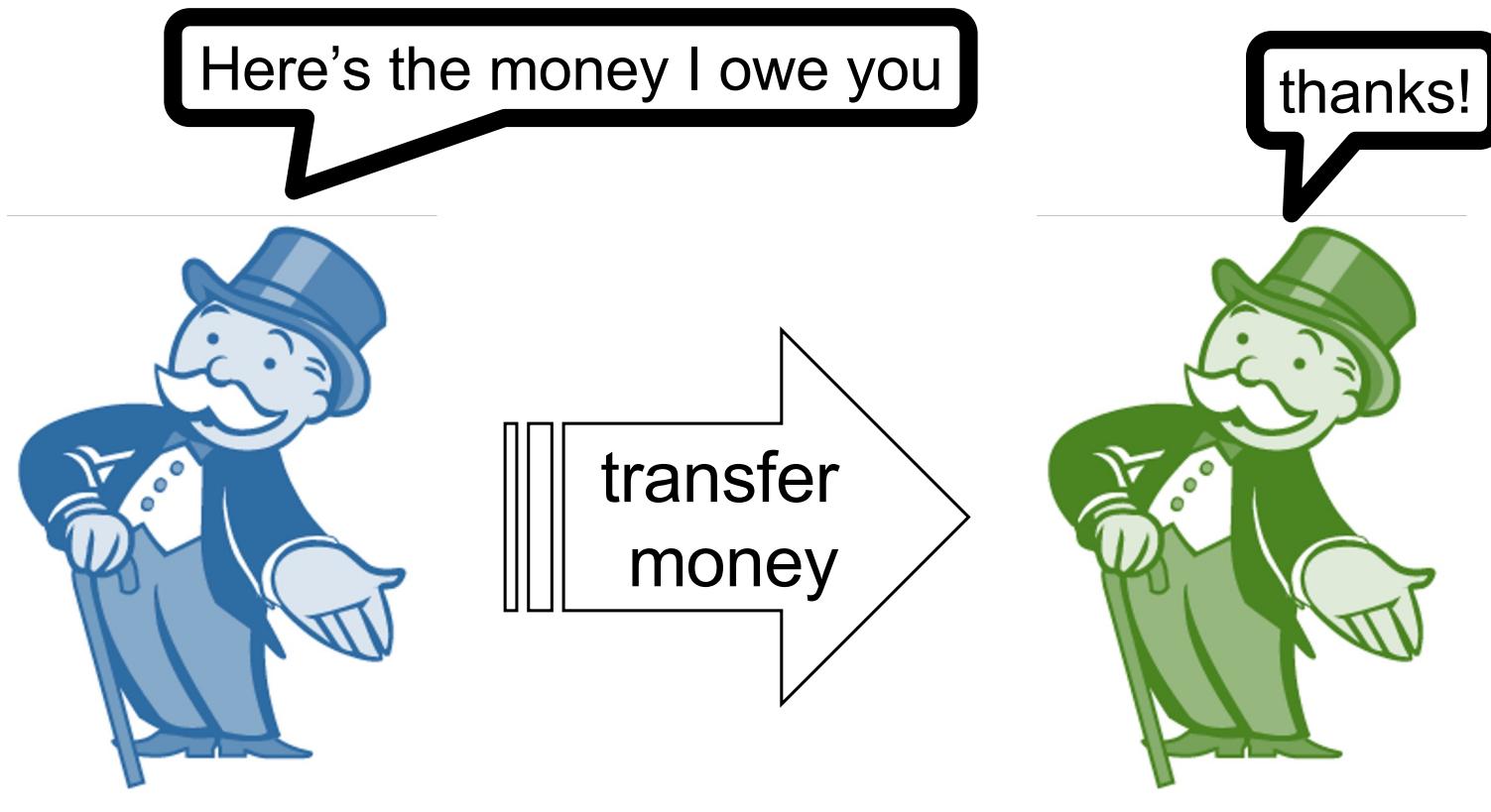
Contracts can call other contracts

Message Call Transaction

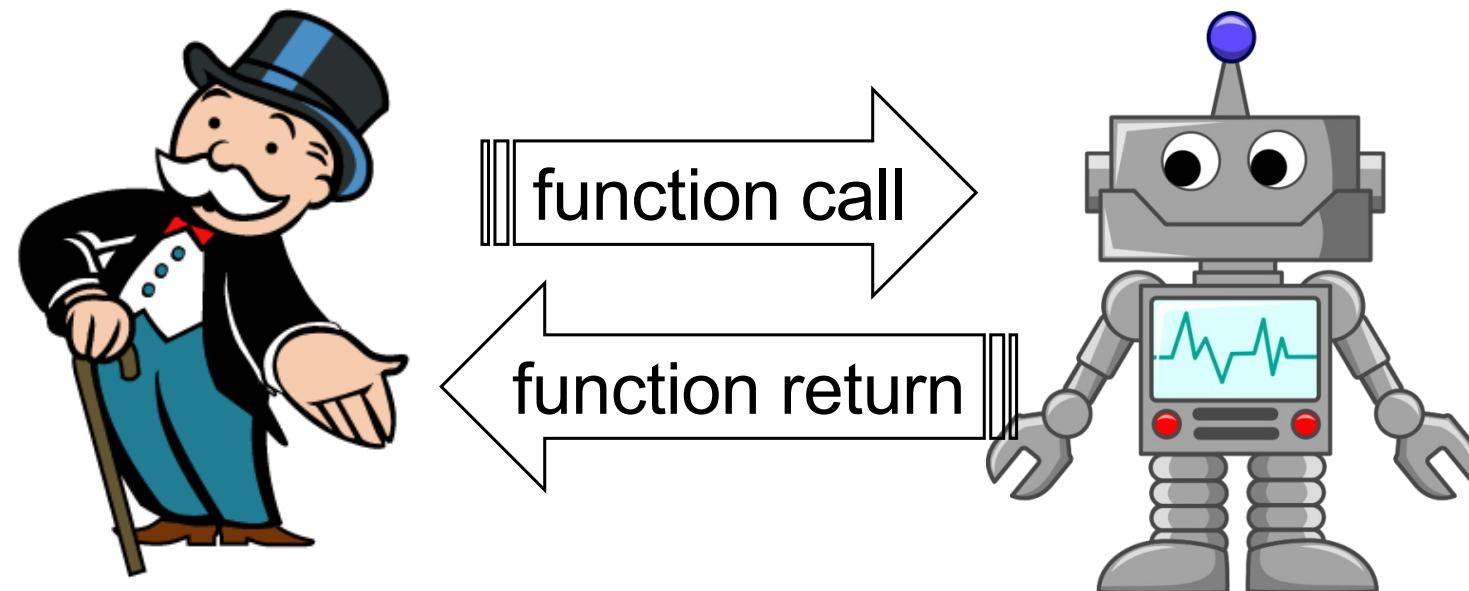


Contracts can also send funds to users

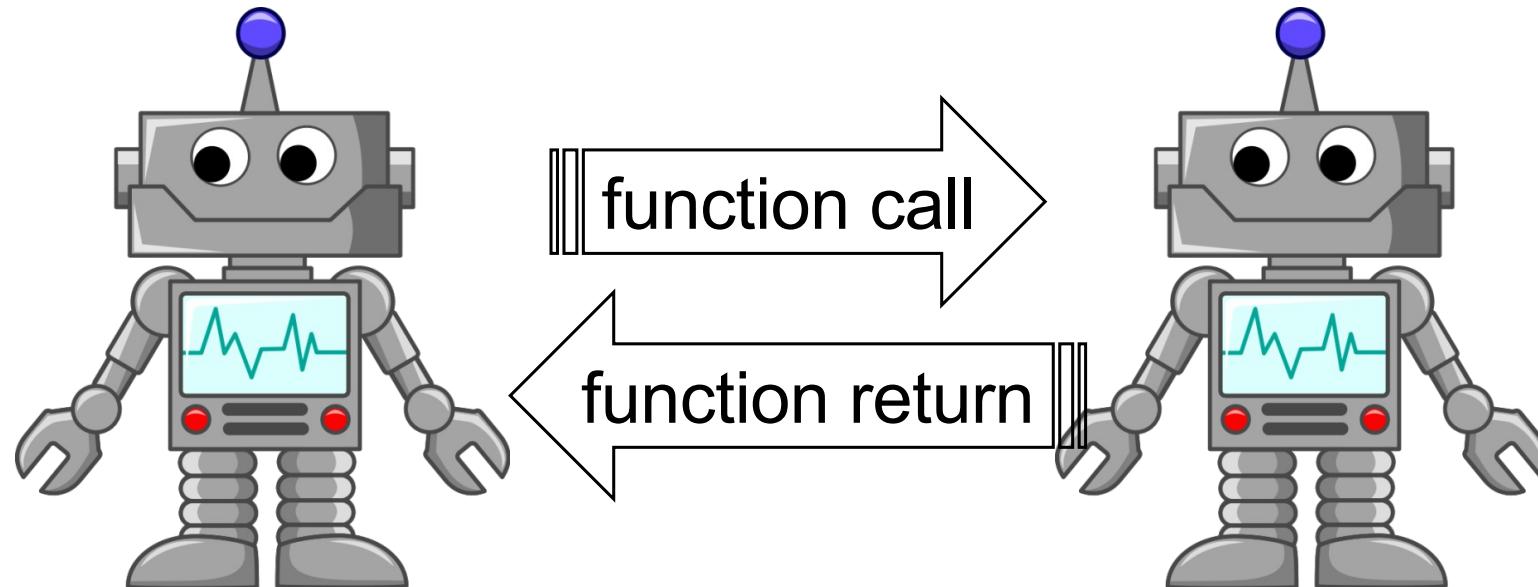
External to External Message



External to Contract & Vice-Versa



Contract to Contract Message

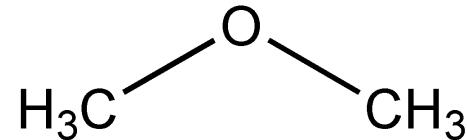


Money, Honey

Native currency called *ether*



not this



but this



Gas

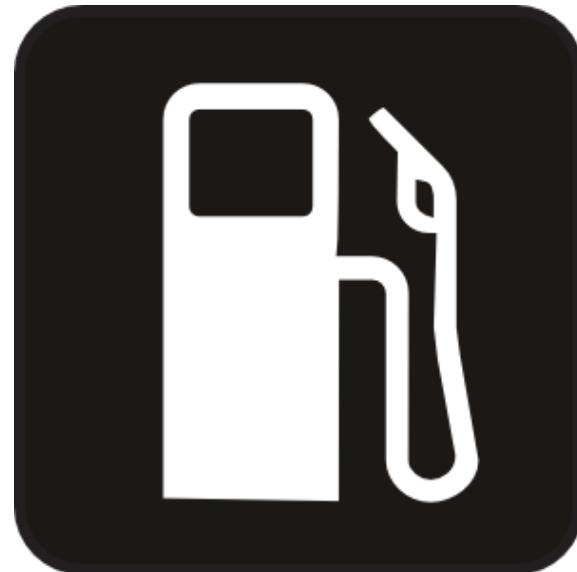
By calling a contract we are essentially running code, written on a Turing complete language, on thousands of nodes worldwide

Problems?

Gas

Caller pays fee for each transaction step

Denial of Service attacks expensive



Gas

Each step has fixed “gas” fee

But gas price in Ether up to caller!

Low price means low priority ...

And vice-versa



Gas

If a call runs out of gas ...

Effects discarded

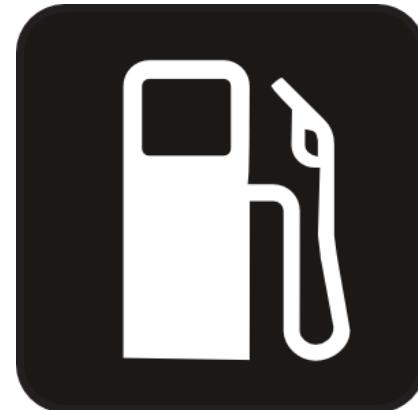
Gas not refunded

If a call has leftover gas ...

Unused gas refunded



Block Gas Limit



Bitcoin has limit on block size

Ethereum has limit on block gas

Block full when transactions' gas costs reach limit

We will see how this can be exploited later

Transaction Fields

Gas price

Value

Gas limit

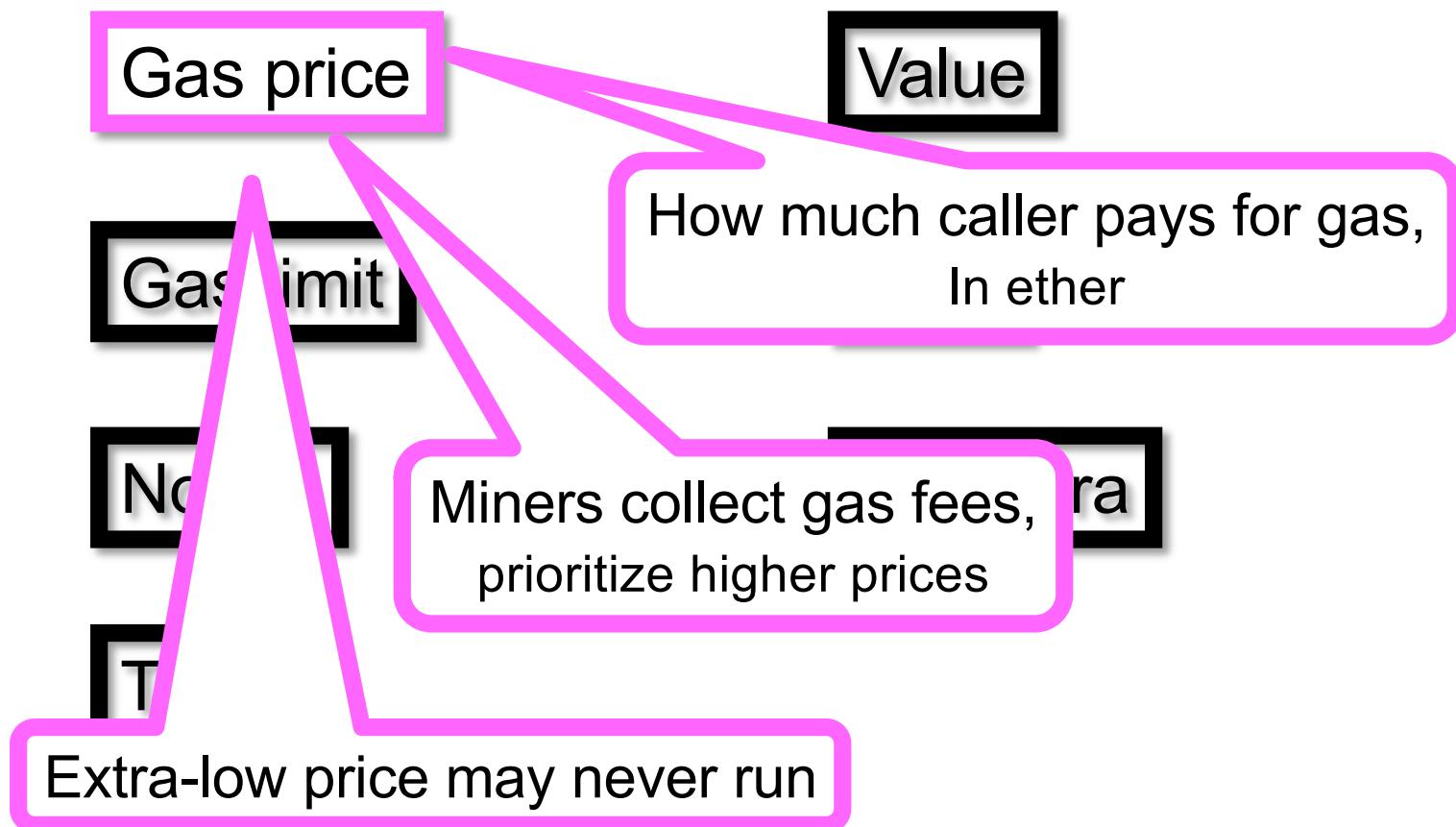
Data

Nonce

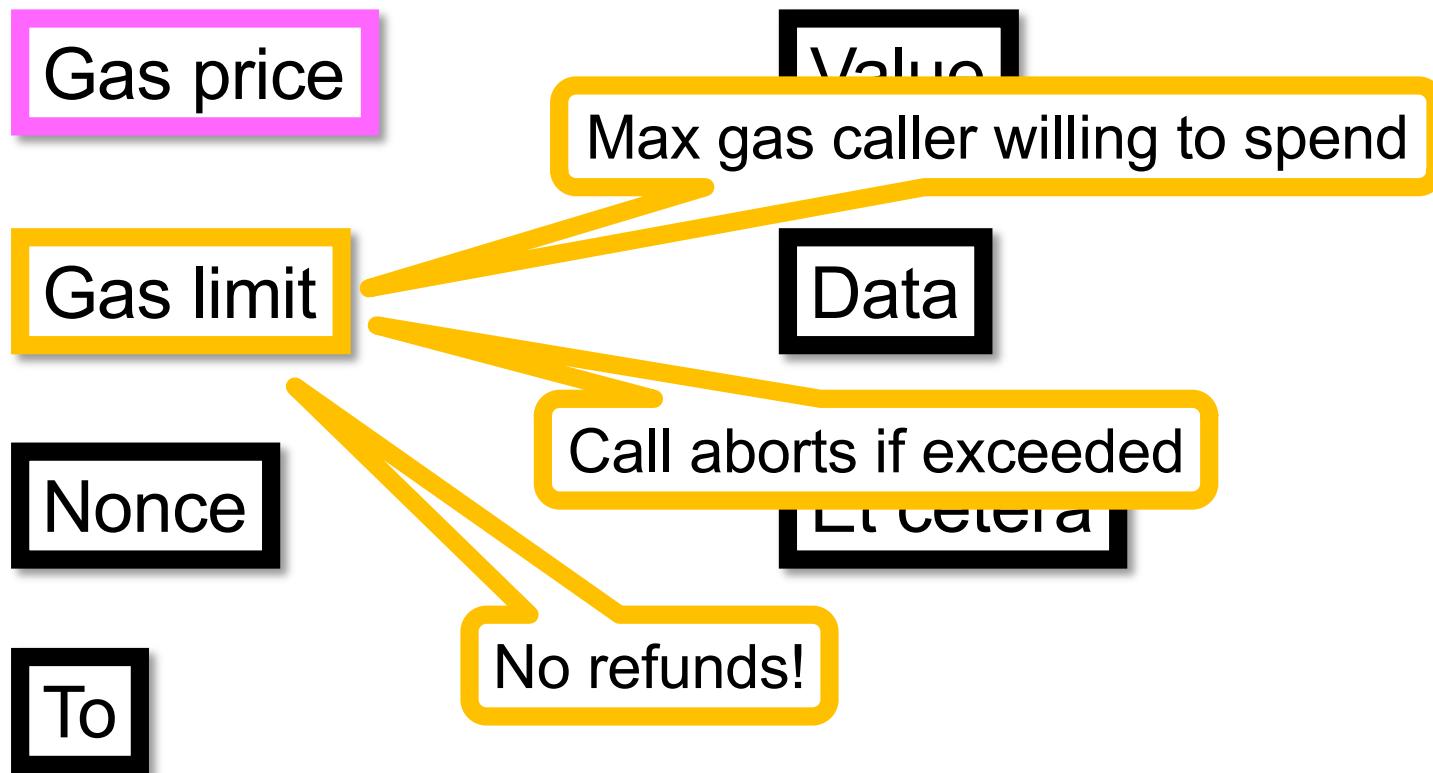
Et cetera

To

Transaction Fields

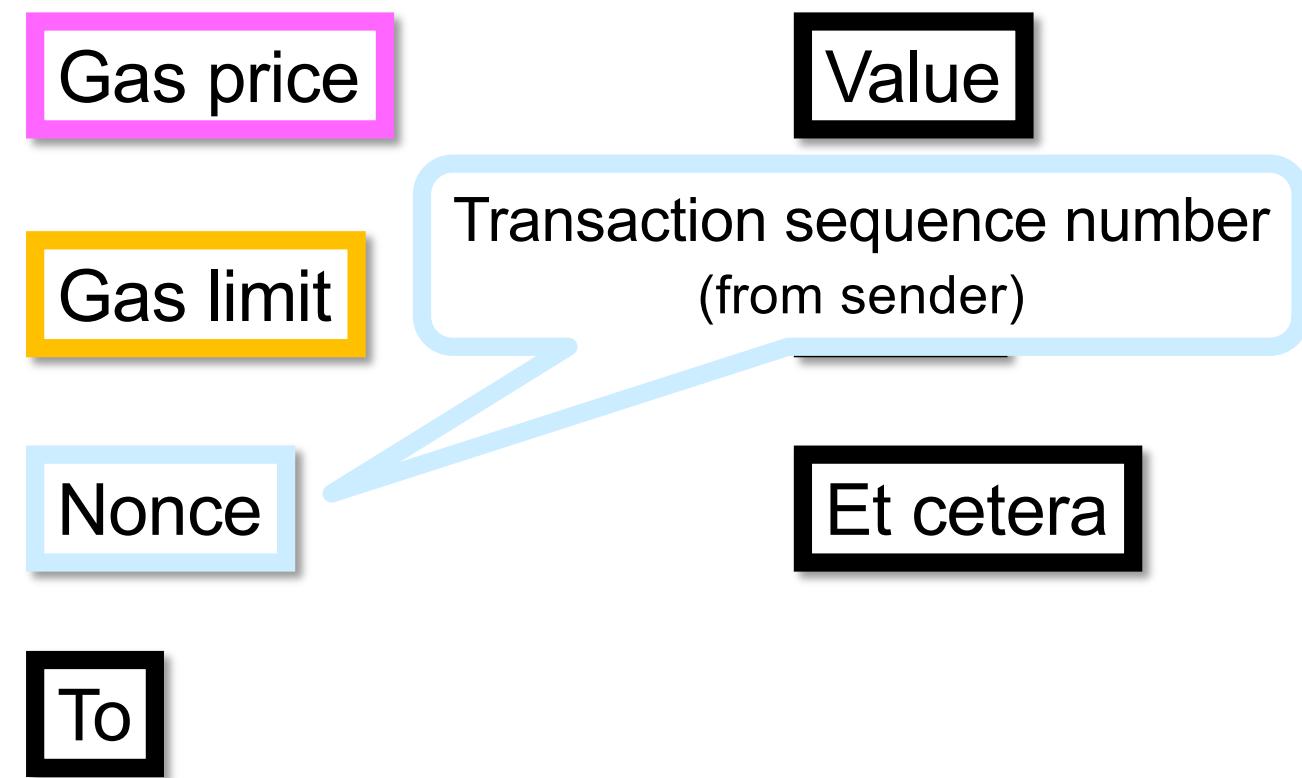


Transaction Fields



Can a caller be sure of the gas that is going to be spent?

Transaction Fields



Transaction Fields

Gas price

Value

Gas limit

Data

Nonce

Et cetera

To

destination address
(external or contract)

Transaction Fields

Gas price

Gas limit

Nonce

To

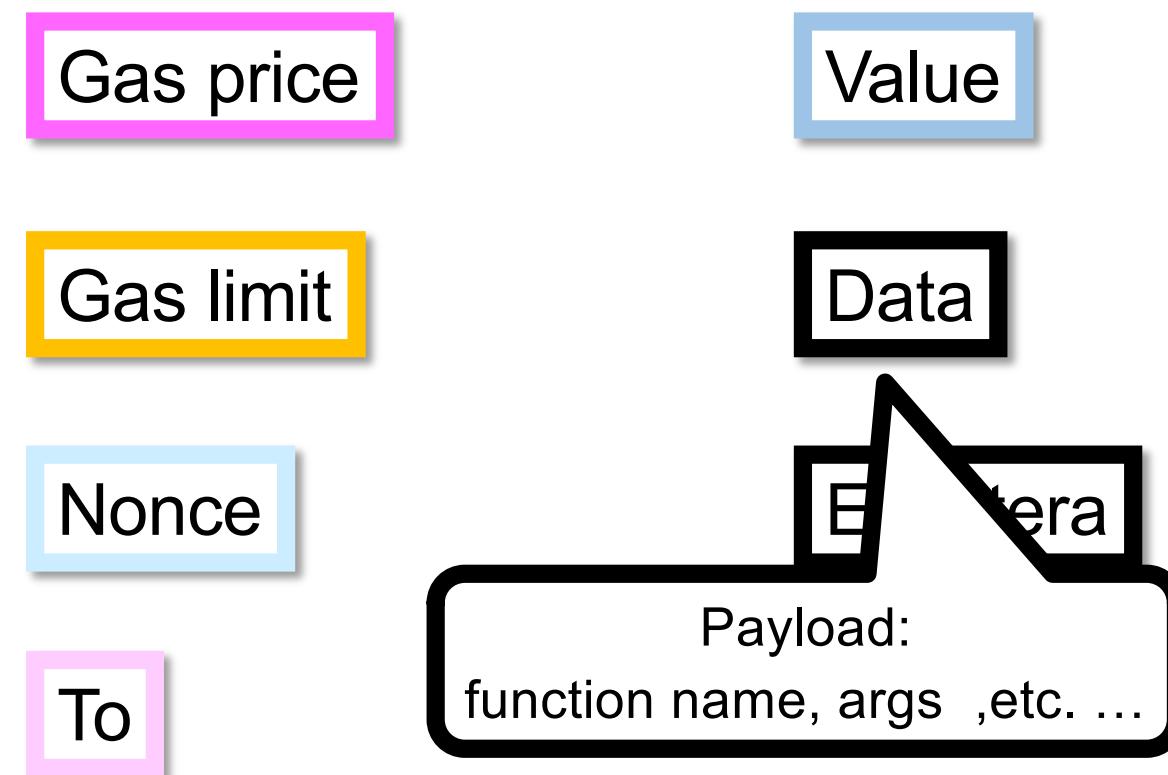
Value

Data

How much ether to transfer

Call criteria

Transaction Fields



Transaction Fields

Gas price

Value

Gas limit

Data

Nonce

Et cetera

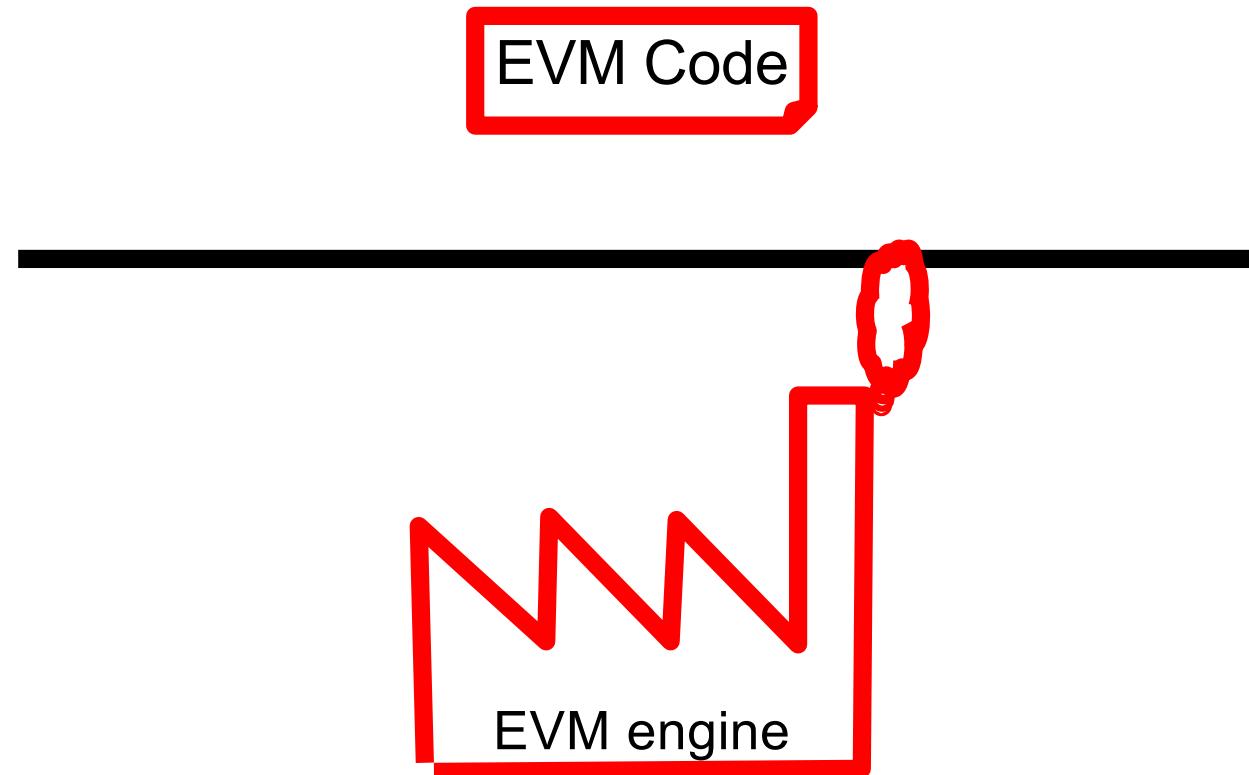
To

ECDSA signature args

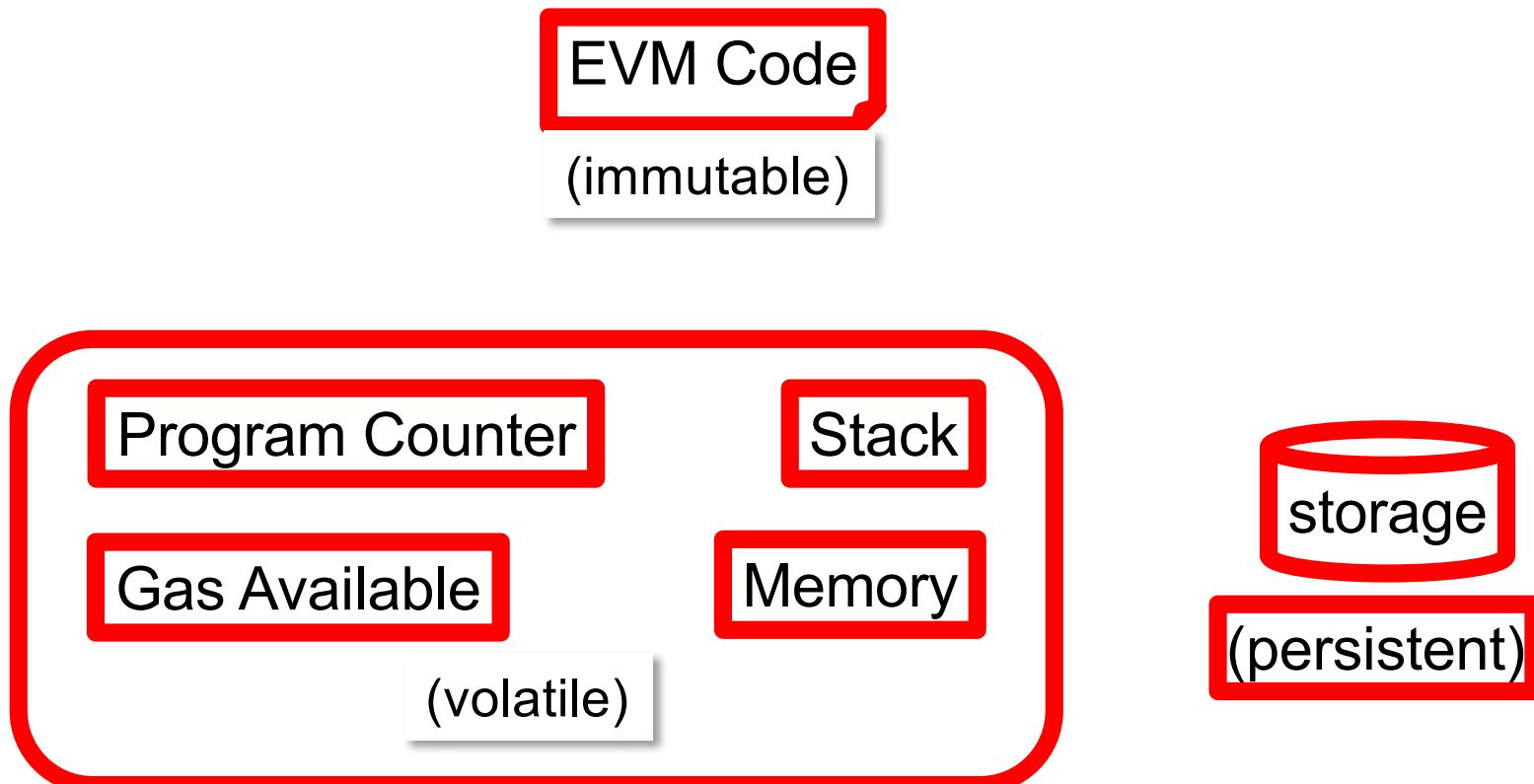
Recent changes in gas pricing

- EIP-1559 (part of “London hard fork” – Aug. 2021) defines new policies for gas pricing
- Idea: replace a first price auction with a base fee mechanism (though users can add a tip for their txns to be prioritized by validators)
- Base fee depends on how full recent blocks are:
 - Blocks do not fill up → base fee decreases
 - Blocks are completely full → base fee increases

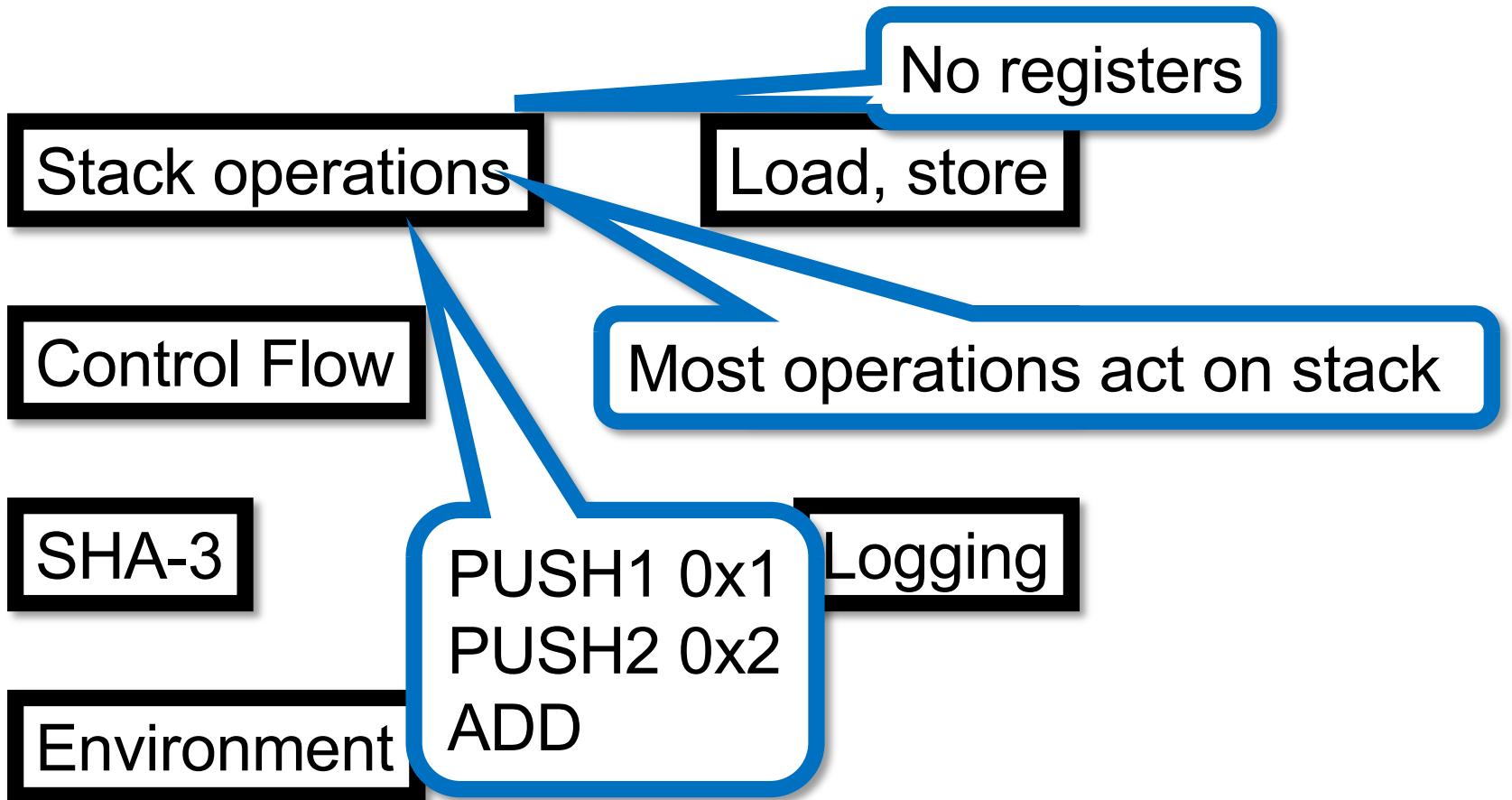
Ethereum Virtual Machine



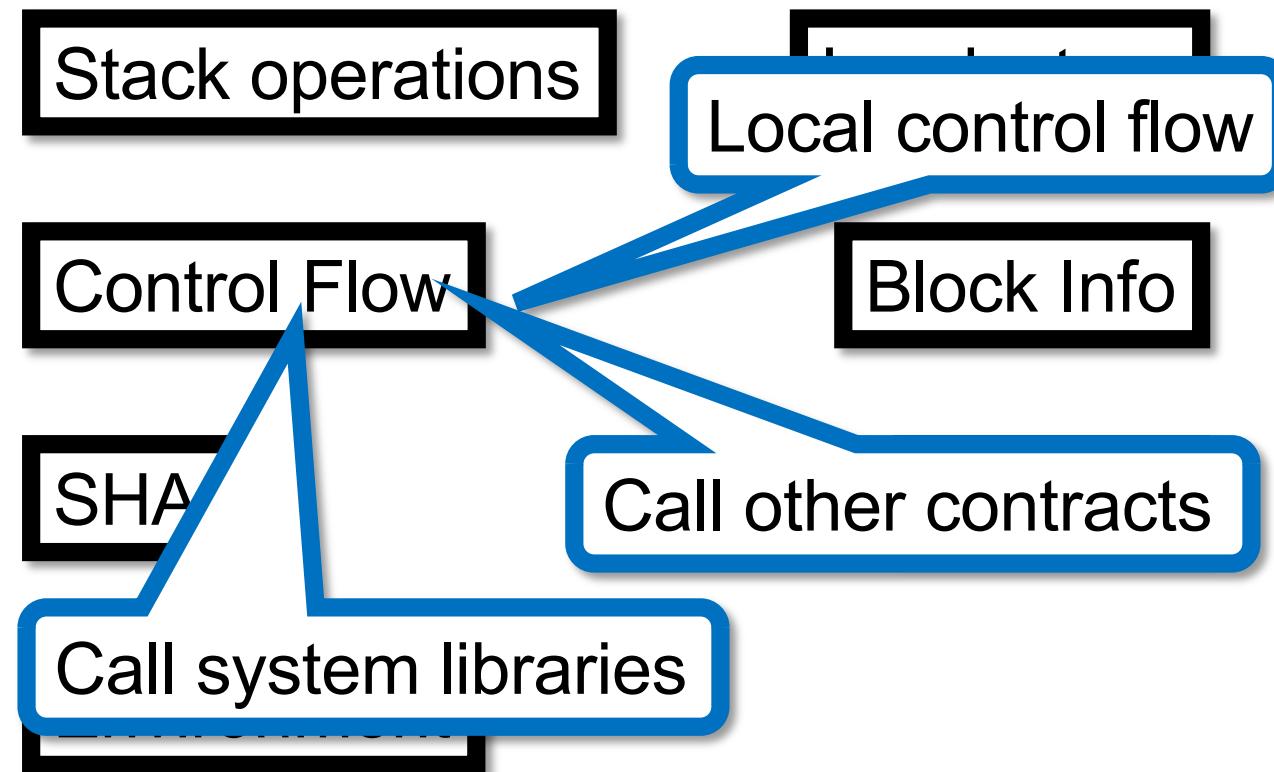
Ethereum Virtual Machine



Types of Instructions



Types of Instructions



Types of Instructions

Stack operations

Load, store

Control Flow

Various crypto hashes provided

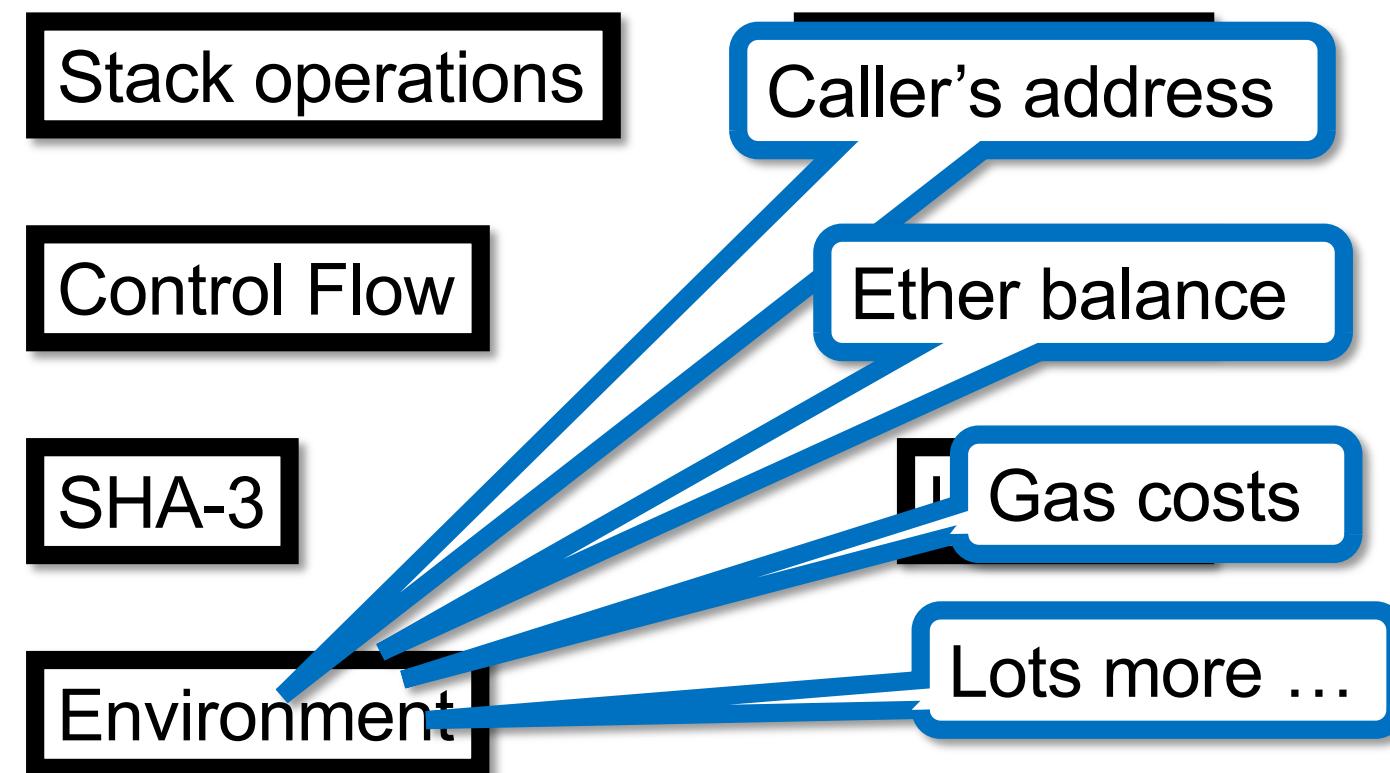
SHA-3

Logging

Environment

Gas costs too expensive to compute directly

Types of Instructions



Types of Instructions

Stack operations

Load, store

Control Flow

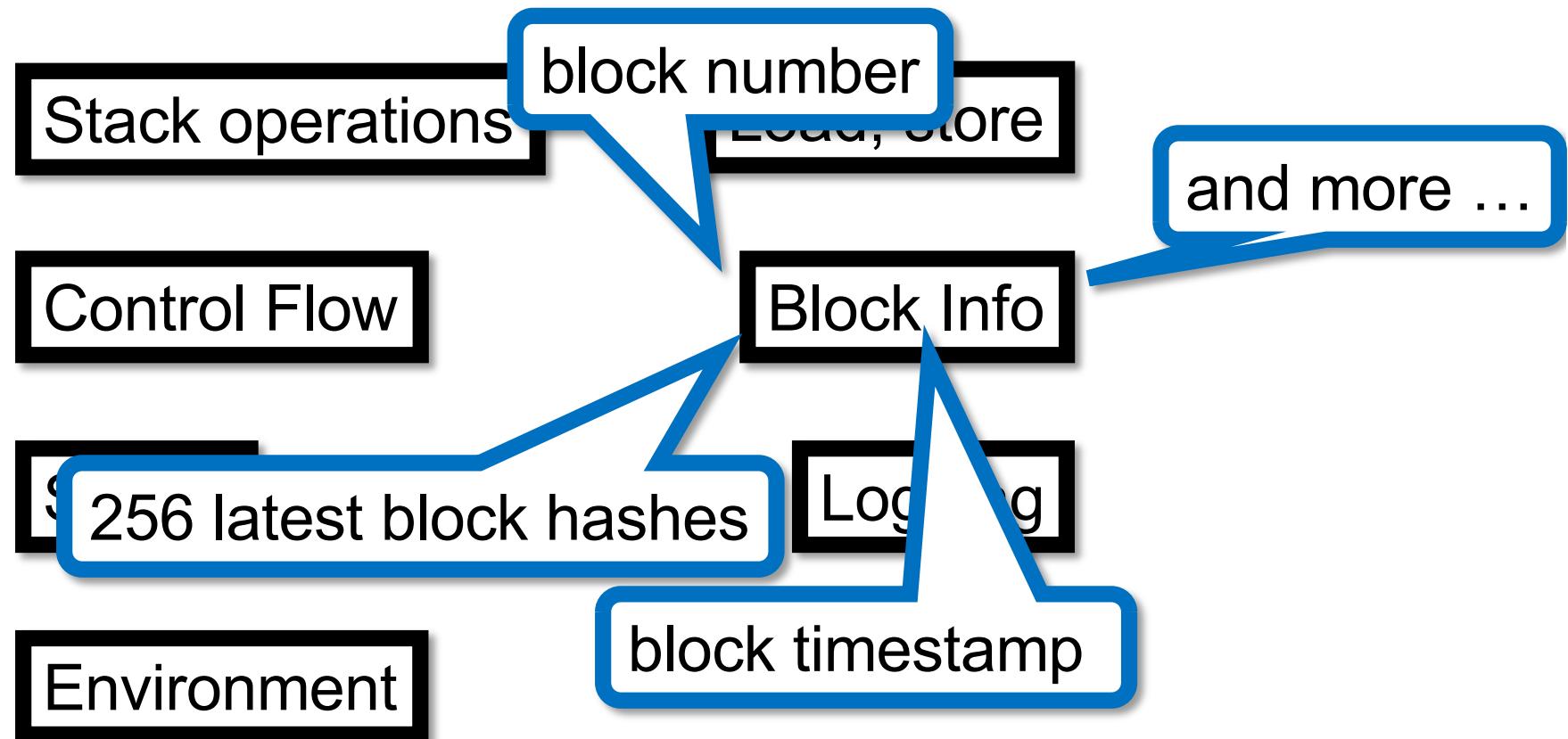
Block

SHA-3

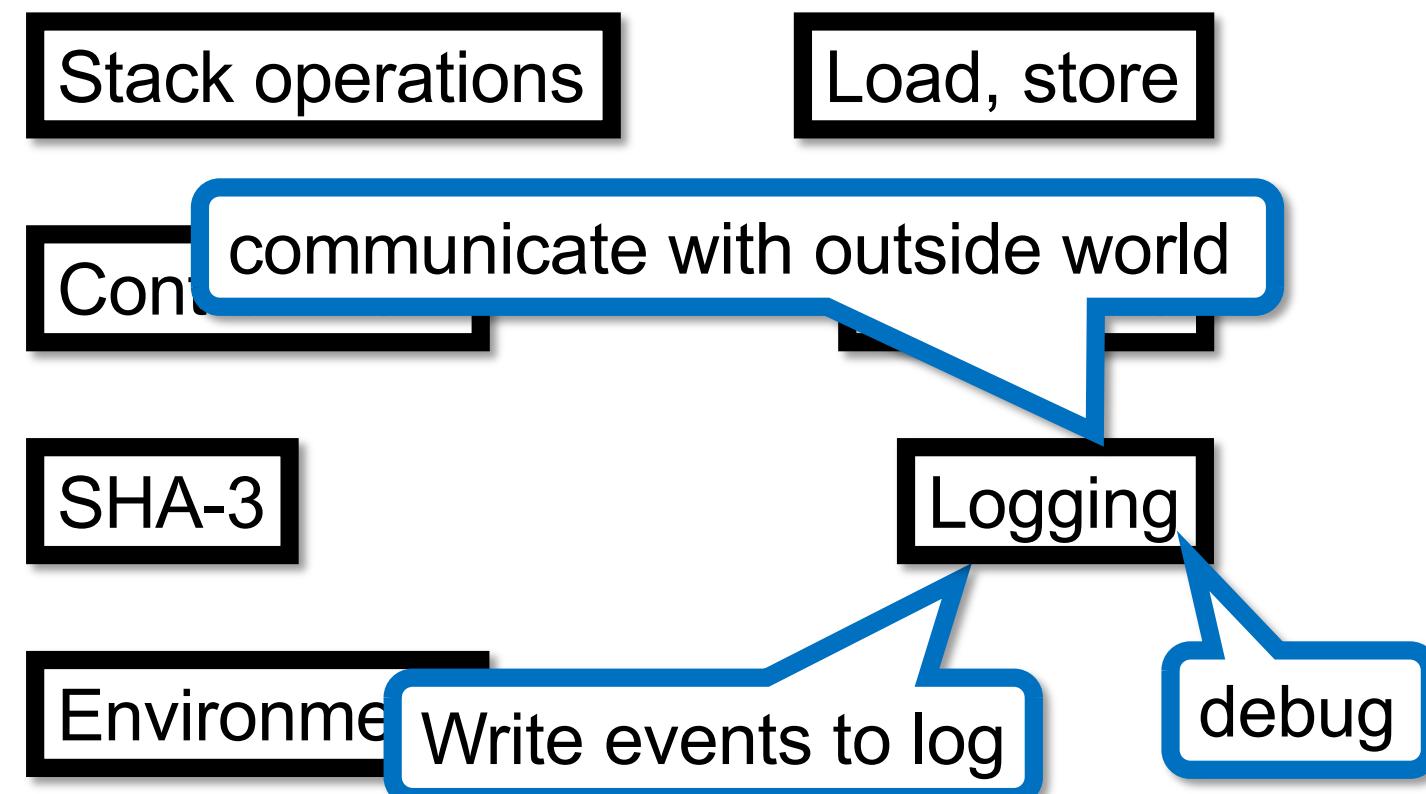
Environment

Load and store
from non-stack memory

Types of Instructions



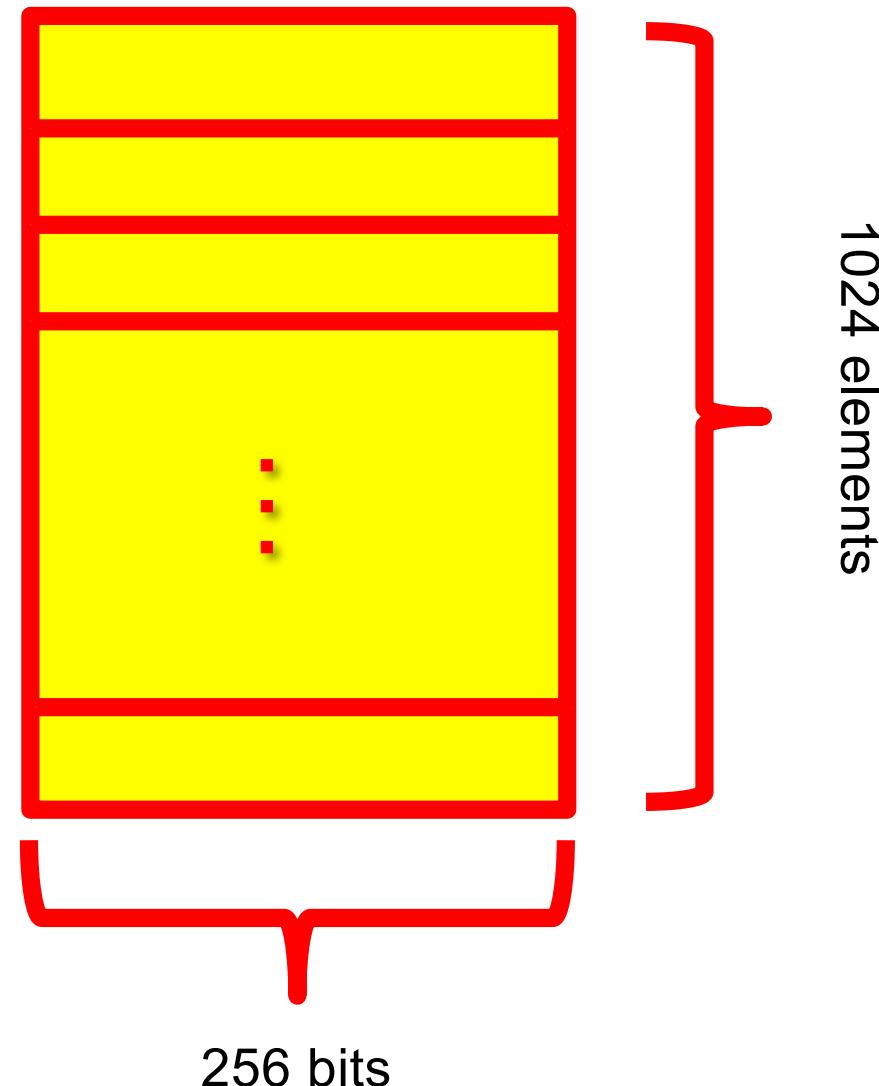
Types of Instructions



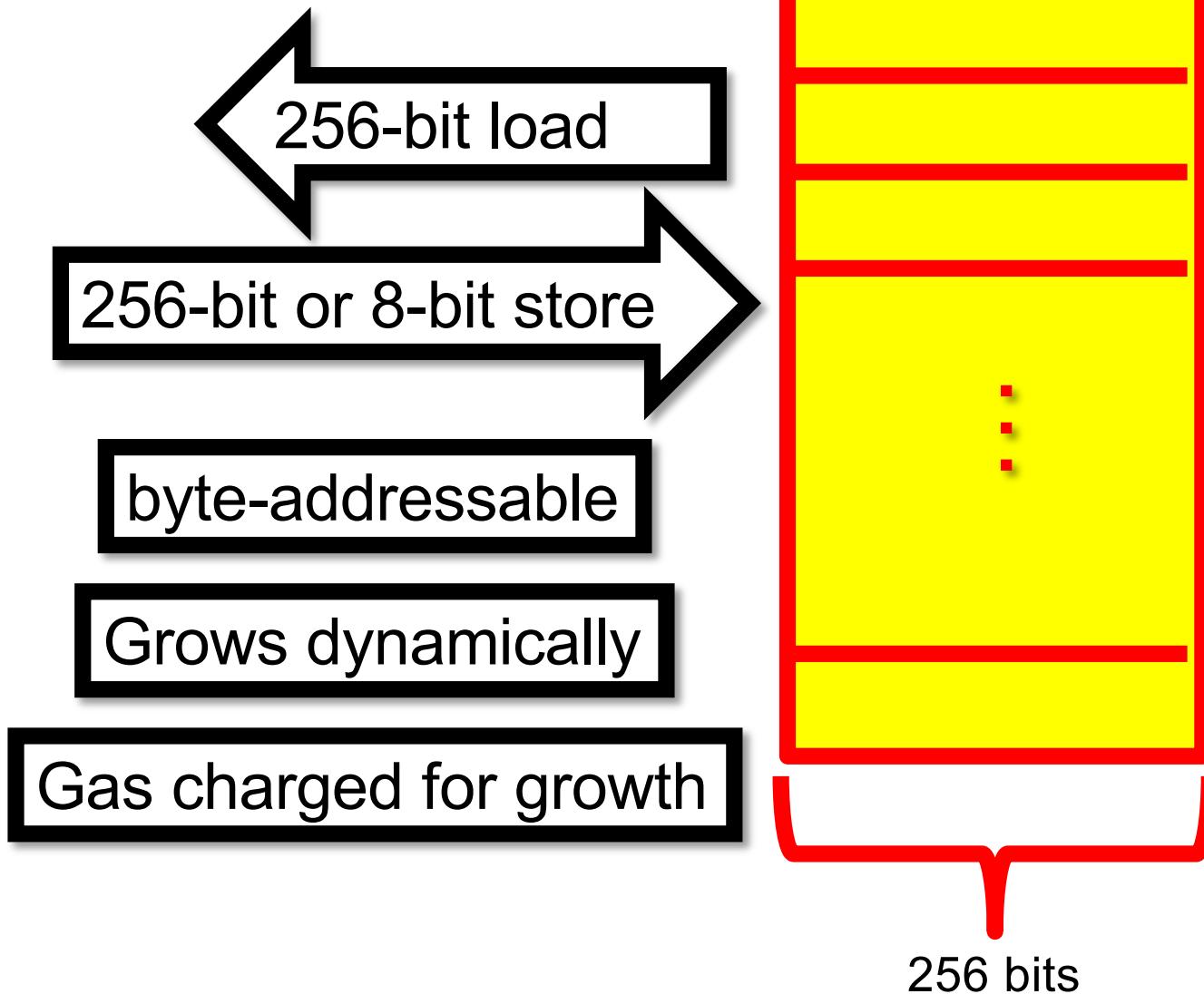
EVM Stack

All operations
act on stack
(e.g., arithmetic
Logic, etc.)

PUSH, POP,
SWAP ...



EVM Memory



EVM Storage

- API/abstraction is identical to memory, but values persist across transactions
- Storage is persisted in the blockchain itself



Storage is Expensive

Users need incentive to free up unneeded storage

Ethereum gives a gas refund for

Freeing storage (overwrite with 0s)

Destroying contracts

Gas Token

(Ethereum token that represents a storage of gas)

GasToken managed by a contract:

Leverage gas refund to “store” gas by creating empty contracts or storage, then tokenize

GasToken contract converts unused gas into GasToken tokens

Tokens can be transferred, sold, traded

Users can generate GasTokens when gas is cheap

GasToken is an Ethereum token that stores gas on the Ethereum network, storing gas when it is expensive. Using GasToken can save users to tokenize gas everything from arbitraging decentralized exchanges to the first contract on the Ethereum network that "hanks" of gas that

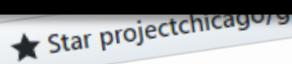
Gas Token Speculation

When gas prices are low

Buy GasTokens

Opposite strategy when prices climb

Speculators aim to profit from price fluctuations



GasToken is a new, cutting-edge Ethereum contract that allows users to tokenize gas on the Ethereum network, storing gas when it is cheap and using / deploying this gas when it is expensive. Using GasToken can subsidize high gas prices on everything from arbitraging decentralized exchanges to buying first contract on the Ethereum network that "banking" of gas that

Is That Ethical?

Negative effects:

Uses memory only for speculation

GasToken might drive up prices

Unethical behavior can arise

Is That Ethical?

Positive Effects

Gas-banking service for users

Hedge against increases

Predictability in face of volatility

Further Reading and Acknowledgements

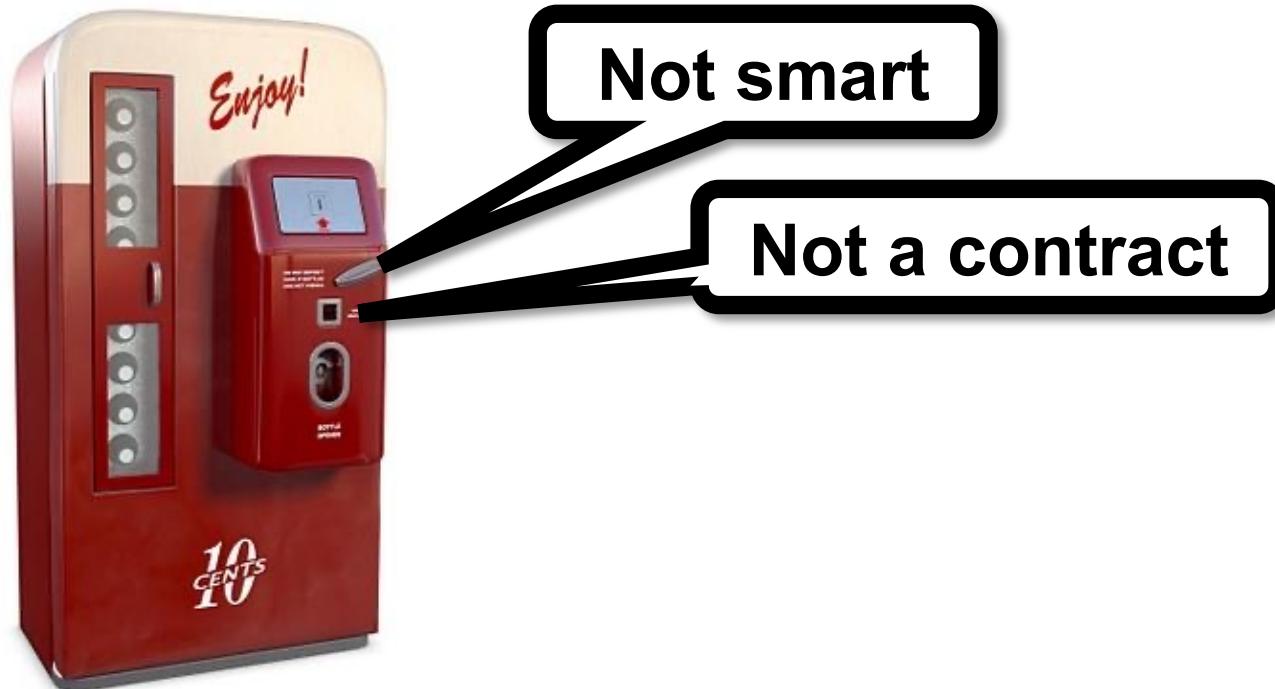
- Ethereum: A Secure Decentralised Generalized Transaction Ledger
- Slides adapted from Maurice Herlihy, Brown University, available under CC BY-NC 4.0 (<https://creativecommons.org/licenses/by-nc/4.0/>)

Solidity

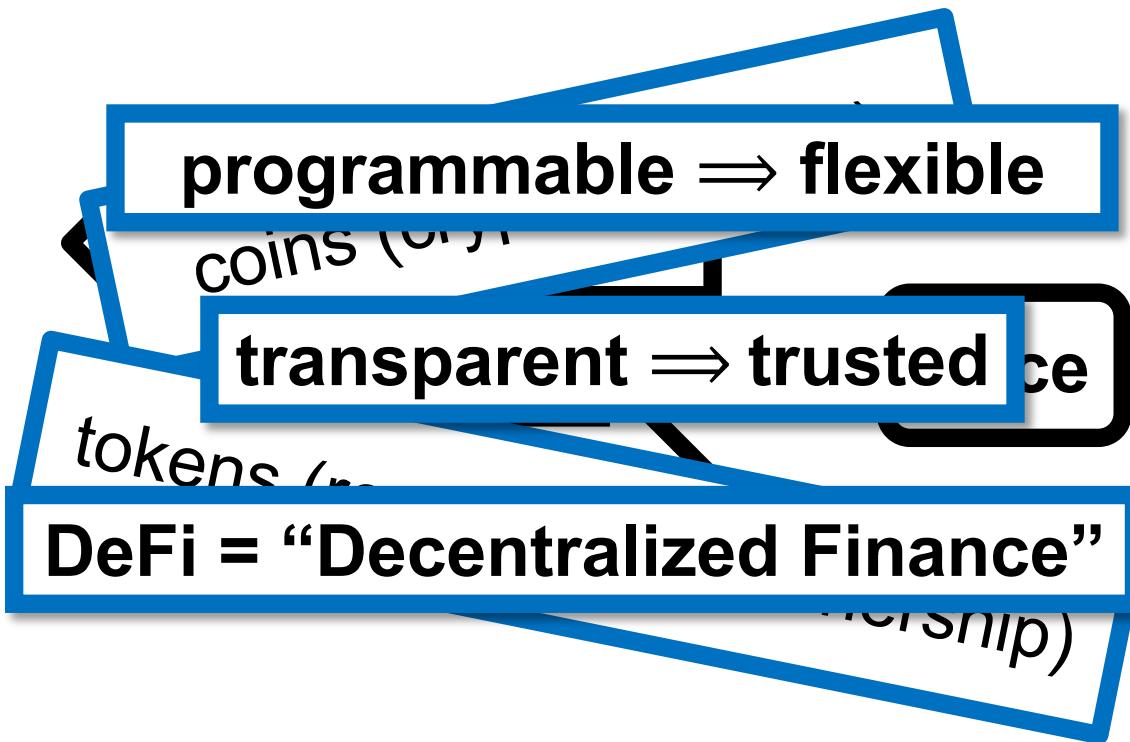
Highly dependable systems – 2024/25

Lecture 9

Smart Contracts



Smart Contracts



Solidity

- Solidity is a high-level language for the EVM
- Based on Javascript
- In the lecture we won't focus on how to program in Solidity
- But rather on the design objectives and challenges
 - Recall: Smart Contracts are executed in a Byz. environment



Coin example

```
pragma solidity ^0.5.2;
contract Coin {
    address minter;
    mapping (address => uint) balances;
    ...
}
```

Coin example

```
pragma solidity ^0.5.2;
```

```
contract Coin {
```

This program needs compiler version 0.5.2 or later

```
    mapping (address => uint) balances;
```

```
    ...
```

```
}
```

Coin example

```
pragma solidity ^0.5.2;  
contract Coin {  
    address minter;  
    ...  
    }  
    A contract is like a class.  
    This contract manages a simple coin  
};
```

Coin example

```
pragma solidity ^0.5.2;
contract Coin {
    address minter;
    mapping (address => uint) balances;
    ...
}
```

long-lived state in EVM storage

Coin example

```
pragma solidity ^0.5.2;
contract Coin {
    address minter;
    mapping (address => uint) balances;
```

Name of an account (person or contract).

Coin example

```
pragma solidity ^0.5.2;  
contract Coin {  
    address minter;  
    mapping (address => uint) balances;
```

Name of an account (person or contract).

Here, only one allowed to mint new coins.

Coin example

```
pragma solidity ^0.5.2;
contract Coin {
    address minter;
    mapping (address => uint) balances;
    ...
}
```

Like a hash table

Coin example

```
pragma solidity ^0.5.2;
contract Coin {
    address minter;
    mapping (address => uint) balances;
    ...
}
```

Like a hash table

Initially, every address maps to zero

Coin example

```
pragma solidity ^0.5.2;
contract Coin {
    address minter;
    mapping (address => uint) balances;
    ...
}
```

Like a hash table

Initially, every address maps to zero

Tracks client balances

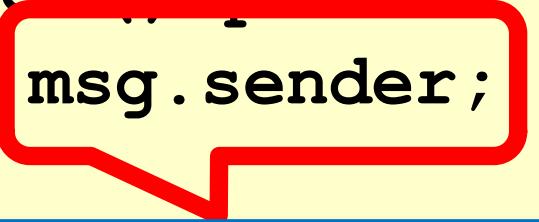
Coin example

```
pragma solidity ^0.5.2;
contract Coin {
    address minter;
    mapping (address => uint) balances;
    constructor() public {
        minter = msg.sender;
    }
}
```

Initializes a contract

Coin example

```
pragma solidity ^0.5.2;
contract Coin {
    address minter;
    mapping (address => uint) balances;
    constructor() public {
        minter = msg.sender;
    }
}
```



The code defines a contract named Coin. It includes variables for the minter (the address that creates the contract) and balances (a mapping of addresses to uint values). The constructor sets the minter to the address of the contract creator (msg.sender). The entire code block is highlighted with a yellow background.

address of contract making the call

Coin example

```
pragma solidity ^0.5.2;
contract Coin {
    address minter;
    mapping (address => uint) balances;
    constructor() public {
        minter = msg.sender;
    }
}
```

Remember the creator's address.

Coin example

```
pragma solidity ^0.5.2;
contract Coin {
    address minter;
    mapping (address => uint) balances;
    ..
    function mint(address owner, uint amount)
        public {
        if (msg.sender != minter) return;
        balances[owner] += amount;
    }
}
```

A function is like a method.

```
pragma solidity ^0.5.2;
contract Coin {
    address minter;
    mapping (address => uint) balances;
    ...
    function mint(address owner, uint amount)
        public {
        if (msg.sender != minter) return;
        balances[owner] += amount;
    }
    ...
}
```

If the caller is not authorized, just return.

```
pragma solidity ^0.5.2;
contract Coin {
    address minter;
    mapping (address => uint) balances;
    ...
    function mint(address owner, uint amount) public {
```

Otherwise credit the amount to the owner's balance

```
        if (msg.sender != minter) return;
        balances[owner] += amount;
    }
}
```

Coin example

```
pragma solidity ^0.5.2;
```

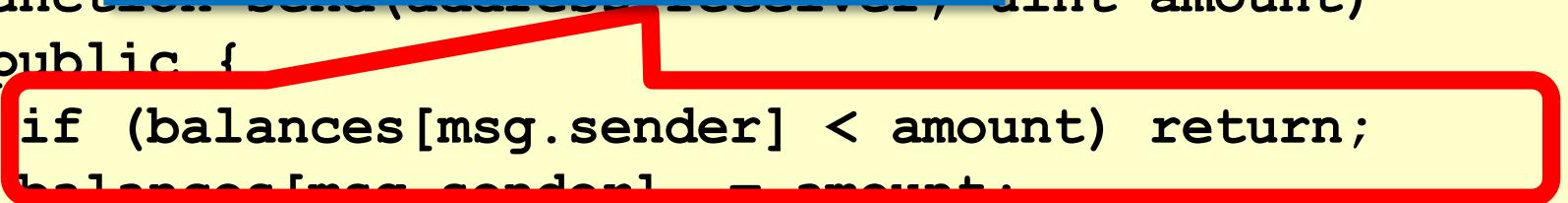
One party transfers coins to counterparty

```
mapping (address => uint) balances;  
...  
function send(address receiver, uint amount)  
public {  
    if (balances[msg.sender] < amount) return;  
    balances[msg.sender] -= amount;  
    balances[receiver] += amount;  
}  
...  
}
```

Coin example

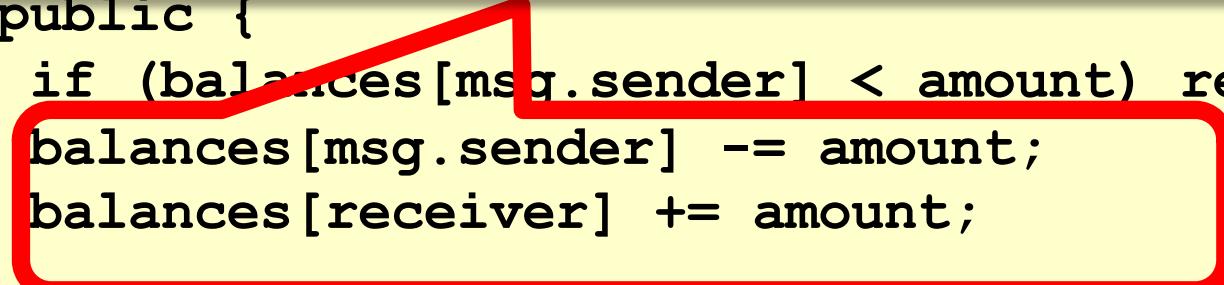
```
pragma solidity ^0.5.2;
contract Coin {
    address minter;
    mapping (address => uint) balances;
    ...
    function send(address receiver, uint amount)
        public {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
    }
    ...
}
```

Quit if insufficient funds



Coin example

```
pragma solidity ^0.5.2;
contract Coin {
    address minter;
    mapping (address => uint) balances;
    ...
    : transfer amount from one account to the other
    public {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
    }
    ...
}
```



Coin Flipping

Alice and Bob want to flip a fair coin

Each one is willing to cheat

**Obvious sources of randomness:
computer time, block hashes, etc. don't work**

Let's try something else

Coin Flipping

```
pragma solidity ^0.5.1;
contract CoinFlipMaybe {
    address[2] player;
    mapping (address => uint) vote;
    mapping (address => bool) played;
    bool result;
    bool done;
    ...
}
```

Coin Flipping

```
pragma solidity ^0.5.1;
contract CoinFlipMaybe {
    address[2] player;
    mapping (address => uint) vote;
    mapping (address => bool) played;
    bool result;
    bool done;
```

declare contract and compiler version

Coin Flipping

```
pragma solidity ^0.5.1;
contract CoinFlipMaybe {
    address[2] player;
    mapping (address => uint) vote;
    mapping (address => bool) played;
    bool result;
    bool done;
```

will want to restrict interactions to the two players

Coin Flipping

```
pragma solidity ^0.5.1;
contract CoinFlipMaybe {
    address[2] player,
    mapping (address => uint) vote;
    mapping (address => bool) played,
    bool result;
    bool done;
    ...
}
```

each player contributes to the result

Coin Flipping

```
pragma solidity ^0.5.1;
contract CoinFlipMaybe {
    address[2] player;
    mapping (address => uint) vote;
    mapping (address => bool) played;
    bool result;
    bool done;
    ...
}
```

each player gets one move only

Coin Flipping

```
pragma solidity ^0.5.1;
contract CoinFlipMaybe {
    address[2] player;
    mapping (address => uint) vote;
    mapping (address => bool) played;
    bool result;
    bool done;
    ...
}
```

remember outcome

Coin Flipping

```
pragma solidity ^0.5.1;
contract CoinFlipMaybe {
...
    constructor (address alice, address bob)
public {
        player[0] = alice;
        player[1] = bob;
    }
...
}
```

Coin Flipping

```
pragma solidity ^0.5.1;
contract CoinFlipMaybe {
...
    constructor (address alice, address bob)
    public {
        player[0] = alice;
        player[1] = bob;
    }
...
}
```

constructor just remembers players

Coin Flipping

Coin Flipping

```
function play(uint _vote) public {
    require(msg.sender == player[0] ||
           msg.sender == player[1]);
    require(!played[msg.sender]);
```

require(condition)

revert computation if condition not satisfied.

Like throwing an exception

```
result = ((vote[player[0]] ^
           vote[player[1]]) % 2) == 1;
}
```

Coin Flipping

```
function play(uint _vote) public {
    require(msg.sender == player[0] ||
           msg.sender == player[1]);
    require(!played[msg.sender]);
    vote[msg.sender] = _vote;
    played[msg.sender] = true;
    if (played[player[0]] && played[player[1]]) {
        done = true;
        result = ((vote[player[0]] +
                   vote[player[1]]) % 2) == 1;
    }
}
```

Only Alice and Bob allowed to play

Coin Flipping

```
function play(uint _vote) public {
    require(msg.sender == player[0] ||
            msg.sender == player[1]);
    require(!played[msg.sender]);
    vote[msg.sender] = _vote;
    played[msg.sender] = true;
    if (played[player[0]] && played[player[1]]) {
        done = true;
        result = ((vote[player[0]] ^
                    vote[player[1]]) % 2) == 1;
    }
}
```

Each player can play only once

Coin Flipping

```
function play(uint _vote) public {
    require(msg.sender == player[0] ||
            msg.sender == player[1]);
    require(!played[msg.sender]);
    vote[msg.sender] = _vote;
    played[msg.sender] = true;
    if (played[player[0]] && played[player[1]]) {
        done = true;
        result = ((vote[player[0]] ^
                    vote[player[1]]) % 2) == 1;
    }
}
```

record player's vote

Coin Flipping

```
function play(uint _vote) public {
    require(msg.sender == player[0] ||
           msg.sender == player[1]);
    require(!played[msg.sender]);
    vote[msg.sender] = _vote;
    played[msg.sender] = true;
    if (played[player[0]] && played[player[1]]) {
        done = true;
        result = ((vote[player[0]] ^
                   vote[player[1]]) % 2) == 1;
    }
}
```

record that player voted

Coin Flipping

```
function play(uint _vote) public {
    require(msg.sender == player[0] ||
            msg.sender == player[1]);
    require(!played[msg.sender]);
    vote[msg.sender] = _vote;
    played[msg.sender] = true;
    if (played[player[0]] && played[player[1]]) {
        done = true;
        result = ((vote[player[0]] ^  
                  vote[player[1]]) % 2) == 1;
    }
}
```

we are done if both players voted

Coin Flipping

```
function play(uint _vote) public {
    require(msg.sender == player[0] ||
           msg.sender == player[1]);
    require(!played[msg.sender]);
    vote[msg.sender] = _vote;
    played[msg.sender] = true;
    if (played[player[0]] && played[player[1]]) {
        done = true;
        result = ((vote[player[0]] ^
                   vote[player[1]]) % 2) == 1;
    }
}
```

remember we are done

Coin Flipping

```
function play(uint _vote) public {
    require(msg.sender == player[0] ||
           msg.sender == player[1]);
    require(!played[msg.sender]);
    vote[msg.sender] = _vote;
    played[msg.sender] = true;
    if (played[player[0]] && played[player[1]]) {
        done = true;
        result = ((vote[player[0]] ^
                   vote[player[1]]) % 2) == 1;
    }
}
```

take XOR of votes

Coin Flipping

```
function play(uint _vote) public {
    require(msg.sender == player[0] ||
           msg.sender == player[1]);
    require(!played[msg.sender]);
    vote[msg.sender] = _vote;
    played[msg.sender] = true;
    if (played[player[0]] && played[player[1]]) {
        done = true;
        result = ((vote[player[0]] ^
                   vote[player[1]]) % 2) == 1;
    }
}
```

flip value is parity of XOR of votes

Coin Flipping

```
function getResult() public view returns (bool) {  
    require (done);  
    return result;  
}
```

Coin Flipping

```
function getResult() public view returns (bool) {  
    require (done);  
    return result;  
}
```

anyone can request outcome

Coin Flipping

```
function getResult() public view returns (bool) {  
    require (done);  
    return result;  
}
```

make sure outcome is ready

Coin Flipping

This contract is insecure!

Why?

Coin Flipping

Suppose Alice plays first

Bob then observes contract state on Etherscan.io

Picks his vote after seeing hers

Bob controls coin flip outcome

Commit-Reveal Pattern

Step One: commit to a value without revealing it

you cannot change your value

Step Two: Wait for others to commit

Step Three: reveal your value

you can check validity

Commit-Reveal Pattern

Alice generates a random value r

Alice commits to r by sending $\text{hash}(r)$ to contract

Alice waits for Bob to commit

Alice reveals r to contract

Compute outcome when Bob reveals

Commit-Reveal Pattern

```
contract CoinFlipCR {  
    address[2] player;  
    mapping (address => uint) commitment;  
    mapping (address => uint) revelation;  
    mapping (address => bool) committed;  
    mapping (address => bool) revealed;  
    bool result;  
    bool done;
```

Commit-Reveal Pattern

```
contract CoinFlipCR {  
    address[2] player;  
    mapping (address => uint) commitment;  
    mapping (address => uint) revelation;  
    mapping (address => bool) committed;  
    mapping (address => bool) revealed;  
    bool result;  
    bool done;
```

track both commit and reveal

Commit-Reveal Pattern

```
function commit(uint _commit) public {
    require(msg.sender == player[0] ||
            msg.sender == player[1]);
    require(!committed[msg.sender]);
    commitment[msg.sender] = _commit;
    committed[msg.sender] = true;
}
```

Commit-Reveal Pattern

```
function commit(uint _commit) public {
    require(msg.sender == player[0] ||
           msg.sender == player[1]);
    require(!committed[msg.sender]);
    commitment[msg.sender] = _commit;
    committed[msg.sender] = true;
}
```

members only

Commit-Reveal Pattern

```
function commit(uint _commit) public {
    require(msg.sender == player[0] ||
            msg.sender == player[1]);
    require(!committed[msg.sender]);
    commitment[msg.sender] = _commit;
    committed[msg.sender] = true;
}
```

You only commit once

Commit-Reveal Pattern

```
function commit(uint _commit) public {
    require(msg.sender == player[0] ||
            msg.sender == player[1]);
    require(!committed[msg.sender]);
    commitment[msg.sender] = _commit;
    committed[msg.sender] = true;
}
```

close the deal

```
function reveal(uint _reveal) public {
    require(msg.sender == player[0] ||
           msg.sender == player[1]);
    require(!revealed[msg.sender]);
    require(
        commitment[msg.sender]==hash(_reveal));
    revelation[msg.sender] = _reveal;
    revealed[msg.sender] = true;
    if (revealed[player[0]] &&
        revealed[player[1]]) {
        done = true;
        result = ((revelation[player[0]] ^
                   revelation[player[1]]) %
                  2) == 1;
    }
}
```

```
function reveal(uint _reveal) public {
    require(msg.sender == player[0] ||
           msg.sender == player[1]);
    require(!revealed[msg.sender]);
    require(
        commitment[msg.sender] == hash(_reveal));
    revelation[_reveal] = true;
    revealed[msg.sender] = true;
    if (revealed[player[0]] &&
        revealed[player[1]]) {
        done = true;
        result = ((revelation[player[0]] ^
                  revelation[player[1]]) %
                  2) == 1;
    }
}
```

Usual preconditions

```
function reveal(uint _reveal) public {
    require(msg.sender == player[0] ||
           msg.sender == player[1]);
    require(!revealed[msg.sender]);
    require(
        commitment[msg.sender]==hash(_reveal));
    revelation[msg.sender] = _reveal;
    revealed[msg.sender] = true;
    if (revealed[player[0]] & revealed[player[1]])
        revealed[player[0]] make sure reveal is sincere
    done = true;
    result = ((revelation[player[0]] ^
               revelation[player[1]]) %
              2) == 1;
}
}
```

```
function reveal(uint _reveal) public {
    require(msg.sender == player[0] ||
           msg.sender == player[1]);
    require(!revealed[msg.sender]);
    require(
        commitment[msg.sender]==hash(_reveal));
    revelation[msg.sender] = _reveal;
    revealed[msg.sender] = true;
    if (!revealed[player[0]] &&
        revealed[player[1]]) {
        done = true;
        result = ((revelation[player[0]] ^
                  revelation[player[1]]) %
                  2) == 1;
    }
}
```

record revelation

```
function reveal(uint _reveal) public {
    require(msg.sender == player[0] ||
        msg.sender == player[1]);
    require(!_revealed[msg.sender]);
    require(
        commitment[msg.sender]==hash(_reveal));
    revelation[msg.sender] = _reveal;
    revealed[msg.sender] = true;
    if (revealed[player[0]] &&
        revealed[player[1]]) {
        done = true;
        result = ((revelation[player[0]] ^
            revelation[player[1]]) %
            2) == 1;
    }
}
```

If done, compute outcome as before

Smart Contract Programming

- Adversarial environment
- Modularity and information hiding often not enough
 - Attacker does not respect abstraction boundaries
- Smart contracts need to withstand attacks from motivated attackers
- Potential large gains for successful exploits

Javascript

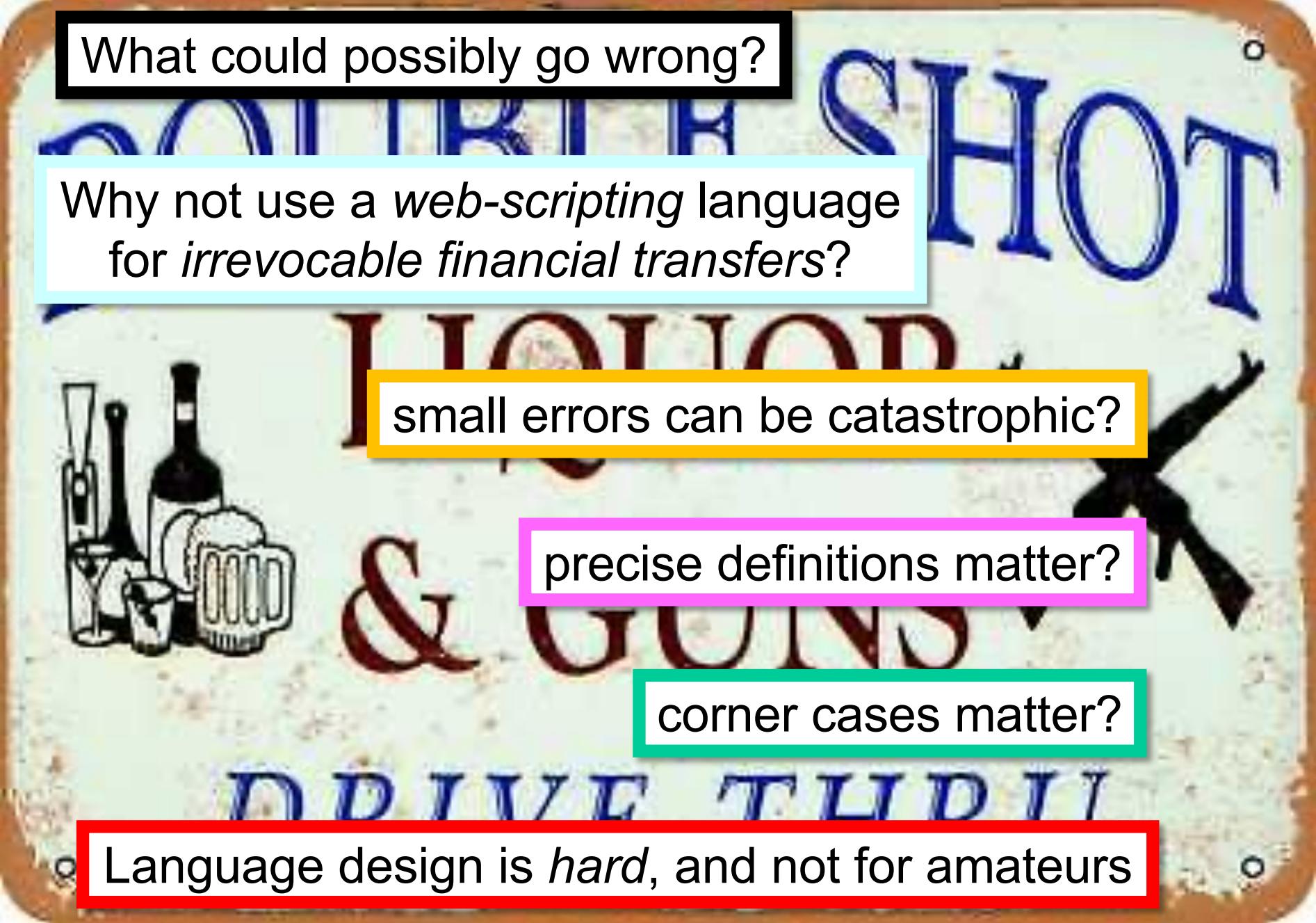
Intended for web page presentation

Odd, complex behavior considered normal

No precise notion of correctness

Must be “easy” for non-experts

No actual adversary



What could possibly go wrong?

Why not use a *web-scripting* language
for *irrevocable financial transfers*?

small errors can be catastrophic?

precise definitions matter?

corner cases matter?

Language design is *hard*, and not for amateurs

Rest of this lecture

Review several vulnerabilities

Some common to all blockchains

Some are Solidity idiosyncrasies ...

"nonsense is nonsense, but the history of nonsense is scholarship"

Re-Entrancy Attack



An Application



DAO = Decentralized Autonomous Organization

Invests in other businesses: about \$50 Million capital

In *Ether* cryptocurrency

No managers or board of directors

Controlled by smart contracts and investor voting

The DAO: Or How A Leaderless Ethereum Project Raised \$50 Million

FEATURE

Michael Castillo (@DelRayMan) | Published on May 12, 2016 at 21:19 BST

DAO =

Invests in other

“code is law”

organization

lion capital

In *Ether* cryptocurrency

No managers or board of directors

Controlled by smart contacts and investor voting

The DAO: a radical experiment that could be the future of decentralised governance

May 10, 2016 4

- FDT

The DAO

The DAO is a New Dow

Nolan Bauerle | Published on May 22, 2016 at 17:22 BST

OPINION

Why the DAO Ethereum is Revolutionary

By Adam Hayes, CFA

/ May 16, 2016 – 2:02 PM EDT

f 235

g+ 17

in 240

22



On April 30, 2016, a brand new organizational structure was born. It's called a decentralized autonomous organization, or DAO. This organization, built on the Ethereum blockchain has already raised over \$100 million for its first project to date. What is the DAO?

What Is the DAO?

The blockchain is a shared ledger that records transactions across many computers in such a way that there is no central authority or need for an intermediary in processing the transaction. It is a permanent record that is timestamped and cannot be altered by anyone without changing all subsequent blocks. Most well-known blockchains are used for cryptocurrencies like Bitcoin and Ethereum, but they can also be used for other purposes such as supply chain management, voting systems, and more.

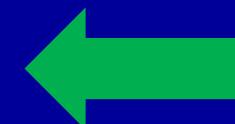
the future of business a company of shareholders, managers, or a

Much hyperventilation about effect on future of finance

```
[ ] noname01.pas 1-[1]
function withdraw(uint amount) {
    client = msg.sender;
    if (balance[client] >= amount) {
        if (client.call.sendMoney(amount)) {
            balance[client] -= amount;
        }}}
```

Client wants to withdraw own money

```
[ ] noname01.pas 1-[1]-  
function withdraw(uint amount) {  
    client = msg.sender;  
    if (balance[client] >= amount} {  
        if (client.call.sendMoney(amount)) {  
            balance[client] -= amount;  
        }}}}
```



Which client?

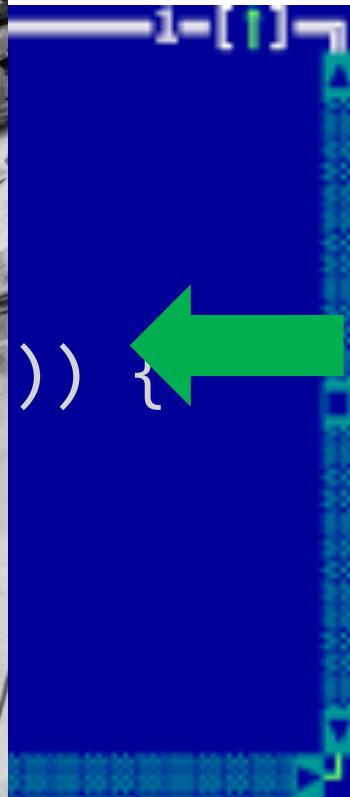
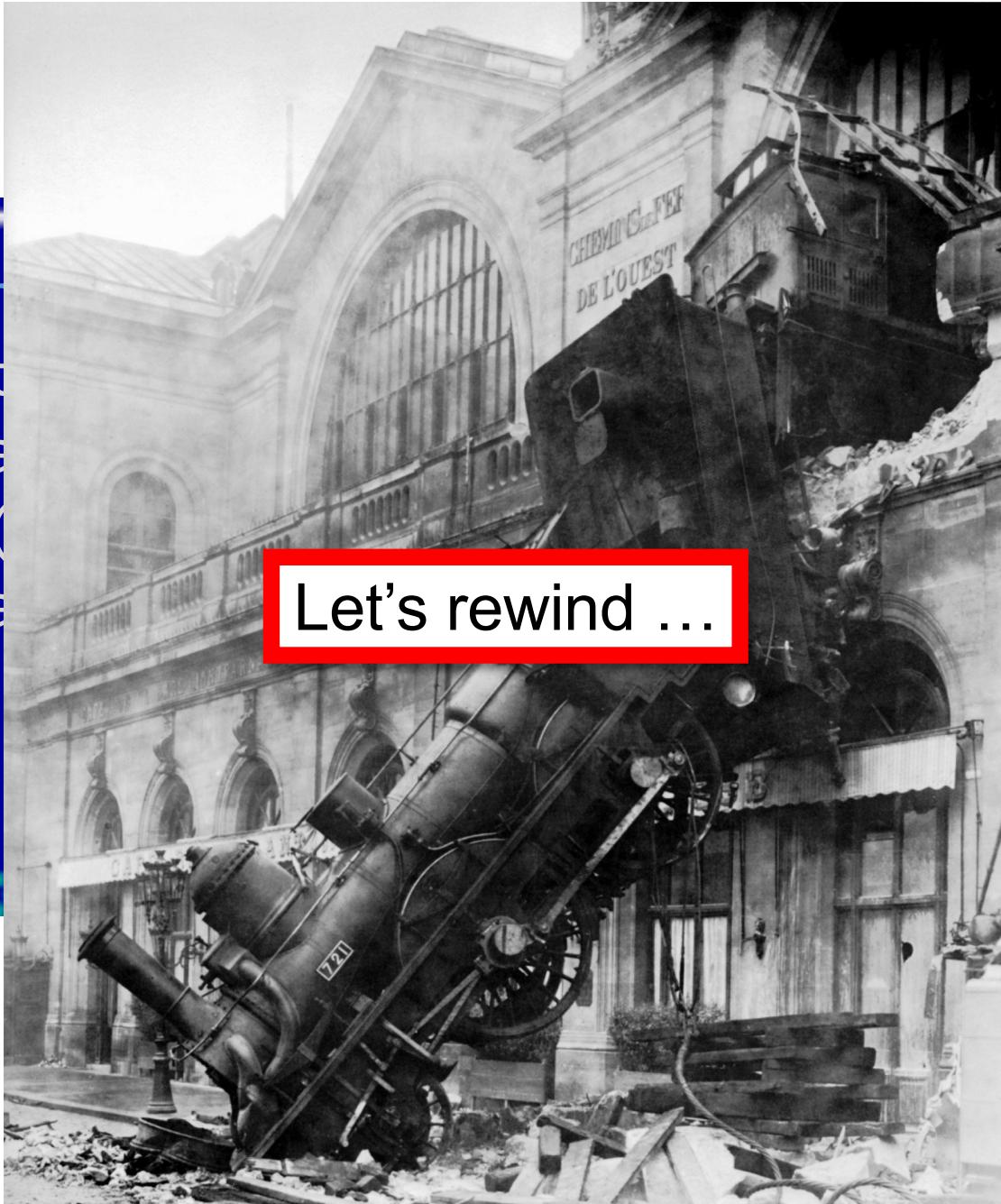
```
[ ] noname01.pas 1-[1]-  
function withdraw(uint amount) {  
    client = msg.sender;  
    if (balance[client] >= amount} {  
        if (client.call.sendMoney(amount)) {  
            balance[client] -= amount;  
        }}}  
  
Does client have enough money?  
56:1
```

```
[ ] noname01.pas [ ]  
function withdraw(uint amount) {  
    client = msg.sender;  
    if (balance[client] >= amount) {  
        if (client.call.sendMoney(amount)) {  
            balance[client] -= amount;  
        }  
    }  
}
```

Transfer the money by calling a function in another contract ...

```
[ ] function client  
  if (ba  
    if (ba  
      ba  
    } })
```

75



75

```
function withdraw(uint amount) {  
    client = msg.sender;  
    if (balance[client] >= amount) {  
        if (client.call.sendMoney(amount)) ←  
            balance[client] -= amount;  
    } } }
```

Transfer the money by calling another contract ...

```
function sendMoney(uint amount) {  
    balance += amount  
    msg.sender.call.withdraw(amount)  
    ...  
}
```

```
function withdraw(uint amount) {  
    client = msg.sender;  
    if (balance[client] >= amount) {  
        if (client.call.sendMoney(amount)) ←  
            balance[client] -= amount;  
    } } }
```

Credit account

```
function sendMoney(uint amount) {  
    balance += amount  
    msg.sender.call.withdraw(amount)  
    ...  
}
```

```
function withdraw(uint amount) {  
    client = msg.sender;  
    if (balance[client] >= amount) {  
        if (client.call.sendMoney(amount)) ←  
            balance[client] -= amount;  
    } } }
```

Wait, what?

Client makes “re-entrant” withdraw request!

```
function sendMoney(uint amount) {  
    balance += amount  
    msg.sender.call.withdraw(amount) ←  
    ...  
}
```

```
noname01.pas
```

```
function withdraw(uint amount) {
    client = msg.sender;
    if (balance[client] >= amount) {
        if (client.call.sendMoney(amount)) ←
            balance[client] -= amount;
    }}}
```

```
noname01.pas
```

```
function sendMoney(uint amount) {
    balance += amount
    msg.sender.call.withdraw(amount)
    ...
}
```

```
function withdraw(uint amount) {  
    client = msg.sender;  
    if (balance[client] >= amount) {  
        if (client.call.sendMoney(amount)) {  
            balance[client] -= amount;  
        }  
    }  
}
```

Second time around, balance still looks OK ...

```
function sendMoney(uint amount) {  
    balance += amount  
    msg.sender.call.withdraw(amount)  
    ...  
}
```

```
function withdraw(uint amount) {  
    client = msg.sender;  
    if (balance[client] >= amount) {  
        if (client.call.sendMoney(amount)) ←  
            balance[client] -= amount;  
    } } }
```

Send money again ...

and again and again ...

```
function sendMoney(uint amount) {  
    balance += amount  
    msg.sender.call.withdraw(amount)  
    ...  
}
```

The DAO Attacked: Code Issue Leads to \$60 Million Ether Theft

Michael del Castillo (@DelRayMan) | Published on June 17, 2016



This happened

≡ BUSINESS
INNOVATION

NEWS

TECH

“The attack is a *recursive calling vulnerability*, where an attacker called the “split” function, and then calls the split function recursively ...”

The actual fix?

sitting in a separate grouping dubbed Ether markets plunged or Poloniex. With ether cur cryptocurrency at mor News of the hack fi prompting Ether well as those fo group. TheDAO w/ The price per unit dropped to \$15 from record h



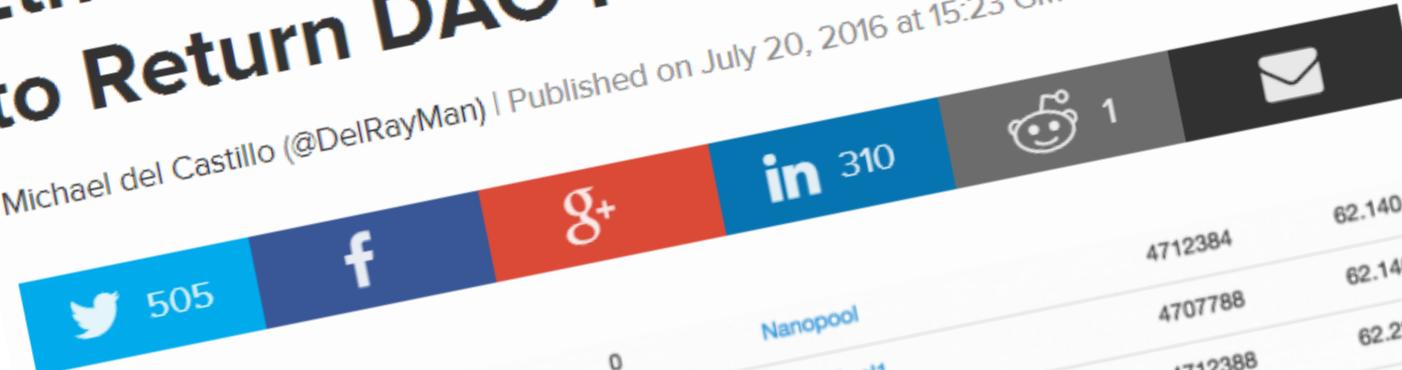
The Fix?

ETHEREUM • TECHNOLOGY

Ethereum Executes Blockchain Hard Fork to Return DAO Funds

Michael del Castillo (@DelRayMan) | Published on July 20, 2016 at 15:23 GMT

NEWS



			Nanopool			
1920004	47 mins ago	6	0	DwarfPool1	4712384	62.140 TH
1920003	48 mins ago	1	0	ethpool	4707788	62.140 TH
1920002	49 mins ago	39	0	bw.com	4712388	62.231 TH
1920001	49 mins ago	57	0	bw.com	4712388	62.322 TH
1920000	50 mins ago	4	0	DwarfPool1	4712384	62.413 TH
1920000	50 mins ago	83	0	bw.com	4707788	62.383 TH
1919999	50 mins ago	0	0	bw.com	4712388	62.352 TH
		20	0			

Blockchain has been implemented, giving those potential for stability after weeks of

The Fix?

ETHEREUM • TECHNOLOGY

Ethereum Executes Blockchain Hard Fork to Return DAO Funds

Castillo (@DelRayMan) | Published on July 20, 2016 at 15:23 GMT

NEWS

Hard-Fork Ethereum and roll back ...

LOL, just kidding about that “code is law” thing ...

Because language design is *hard*

Blockchain has been implemented, giving those potential for stability after weeks of

Rationale for Hard Fork

Client did something stupid?

Client's fault, no refund.

Bug in Ethereum run-time system?

Ethereum's fault, refund.

No warning about reentrancy vulnerability?

Ethereum's fault, refund.



WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information

Article

Talk

Monitor (synchronization)

From Wikipedia, the free encyclopedia

This article contains **instructions, advice, or how-to content**. The purpose of Wikipedia is to help improve this article either by rewriting the how-to content or by moving it to a more appropriate article. (January 2014)

Classical “monitor lock” pitfall

Calling another contract =
releasing monitor lock

most one thread can access the object at a certain condition (thus using the object/class/module".

Monitors were invented by Per Brinch Hansen[1] and C. A. R. Hoare,[2] and were implemented in Brinch Hansen's

Don't violate invariants before a call

Prevention

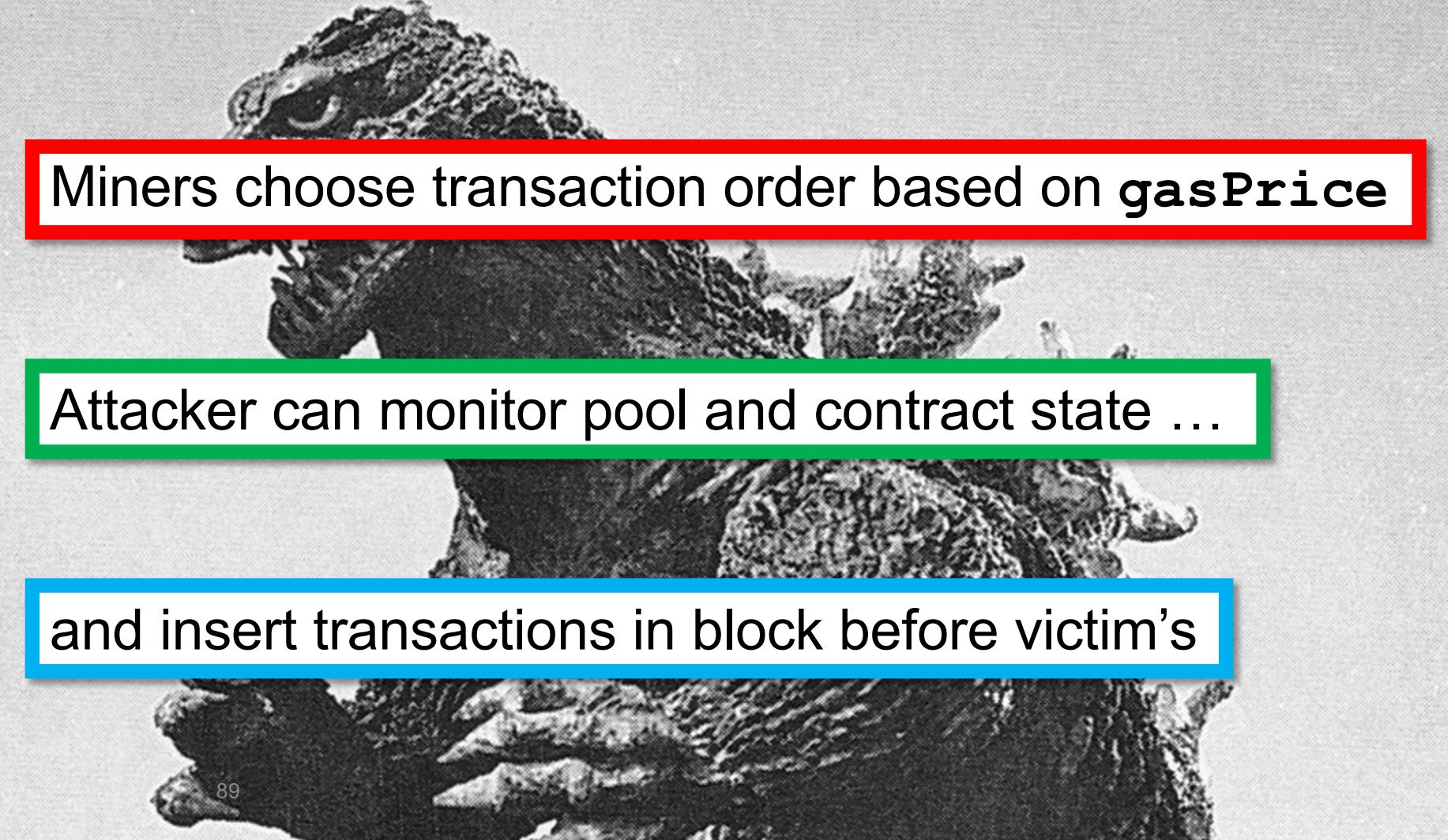
Change state *before* any external calls
("Checks-Effects-Interactions" pattern)

Use a mutex

Make the function non-reentrant

```
[ ] noname01.pas [ ]  
Contract Dao {  
    bool internal locked;  
  
    modifier noReentrancy() {  
        require(!locked, "No reentrancy");  
        locked = true;  
        _;  
        locked = false;  
    }  
  
    // . . .  
    function withdraw() public noReentrancy  
    { // withdraw logic goes here... }  
56:1 =
```

Race Conditions



Miners choose transaction order based on **gasPrice**

Attacker can monitor pool and contract state ...

and insert transactions in block before victim's

“Find This Hash” Game

```
contract FindThisHash {  
    bytes32 constant public hash =  
0xb5b5b97fafd9855eec9b41f74dfb6c38f59511  
41f9a3ecd7f44d5479b630ee0a;  
    constructor() public payable {}  
    function solve(string solution)  
        public {  
        require(hash == keccak256(solution));  
        msg.sender.transfer(1000 ether);  
    }  
}
```

Load with ether

“Find This Hash” Game

```
contract FindThisHash {  
    bytes32 constant public hash =  
0xb5b5b951f9a3eca7144a5479b850eeea,  
    constructor() public payable {}  
    function solve(string solution)  
        public {  
            require(hash == sha3(solution));  
            msg.sender.transfer(1000 ether);  
        }  
}
```

Alice realizes solution is “Ethereum!”



She calls
`FTHContract.solve("Ethereum!")`

Bob sees her transaction, validates answer,
and resubmits with much higher `gasPrice`

Most likely, miners will order Bob's
transaction before Alice's

Prevention

Contracts can put upper bound on `gasPrice`

Does not protect against abuse by miners

Prevention

Parties use commit-reveal scheme

Commit to bid sending **hash(bid)**

After contract accepts commitment,
reveal actual bid

Prevents both miners and users
from front-running

Does not hide transaction value

This Happened

The screenshot shows a Wikipedia-style page for the "ERC20 Token Standard". The page title is "ERC20 Token Standard". Below the title, there's a sub-section titled "Ethereum Based Tokens and ERC20 Wallet Support". A large red box highlights the text "Standard for tradeable tokens". Another green box highlights "Widely used for ICOs". A blue box highlights "Market cap about \$40 Billion". The page includes a sidebar with navigation links like "in page", "Discussion", "Contents [hide]", and "What links here". It also features a sidebar with code snippets related to the standard.

ERC20 Token Standard

Ethereum Based Tokens and ERC20 Wallet Support

Standard for tradeable tokens

Widely used for ICOs

Market cap about \$40 Billion

Re

Page Discussion

Contents [hide]

1 The ERC20 Token Standard Interface

2 Token Contract Work?

2.3 Approve And Transfer

3 Sample Fixed Supply Token Contract

Further Information On Ethereum Tokens

Following is an interface contract for an ERC20 token:

```
function totalSupply() constant returns (uint totalSupply);  
function balanceOf(address owner) constant returns (uint balance);  
function transfer(address to, uint value) returns (bool success);
```

I am willing to
allow this party ...

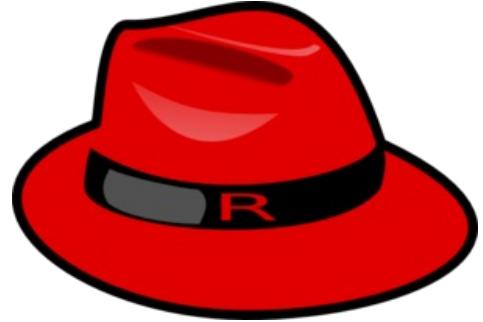
... to withdraw this
amount ...

```
}
```

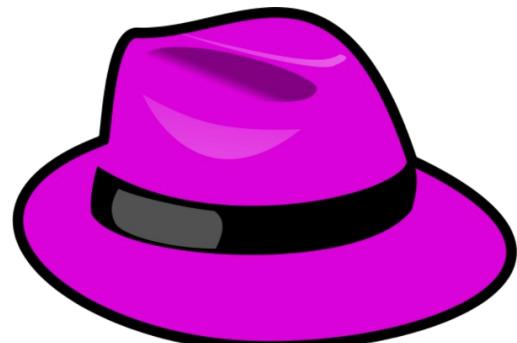
```
function approve(address _spender, uint256 _value) returns (bool success) {  
    allowed[msg.sender][_spender] = _value;  
    Approval(msg.sender, _spender, _value);  
    return true;  
}
```

```
function allowance(address _owner, address _spender) constant returns (uint256 remaining) {
```

... from my account.



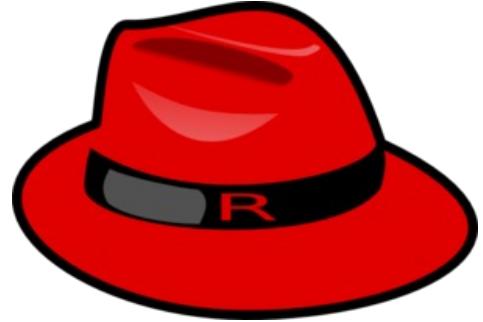
Alice



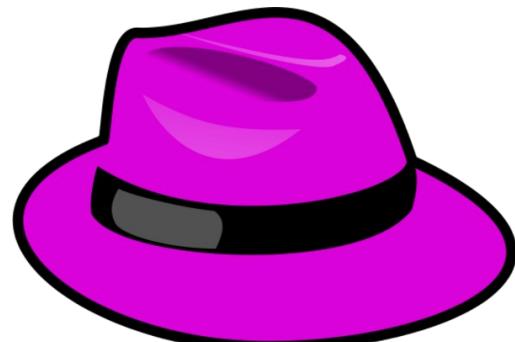
Bob



miner



Alice



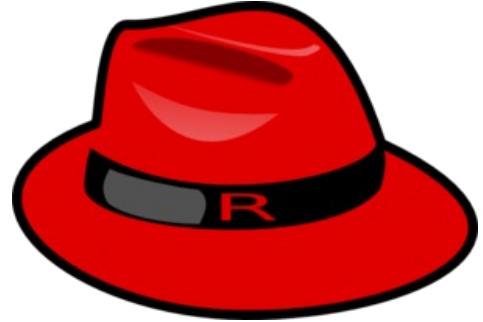
Bob



I am Bob's secret friend

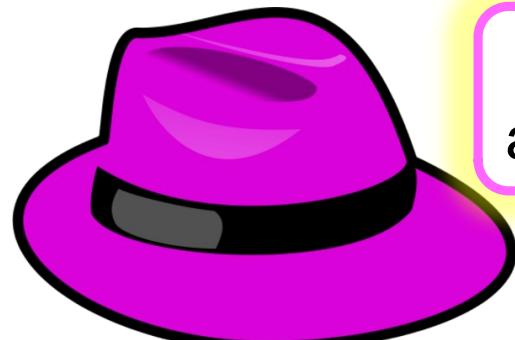


miner



Alice

Alice
authorizes
Bob
to withdraw
\$100

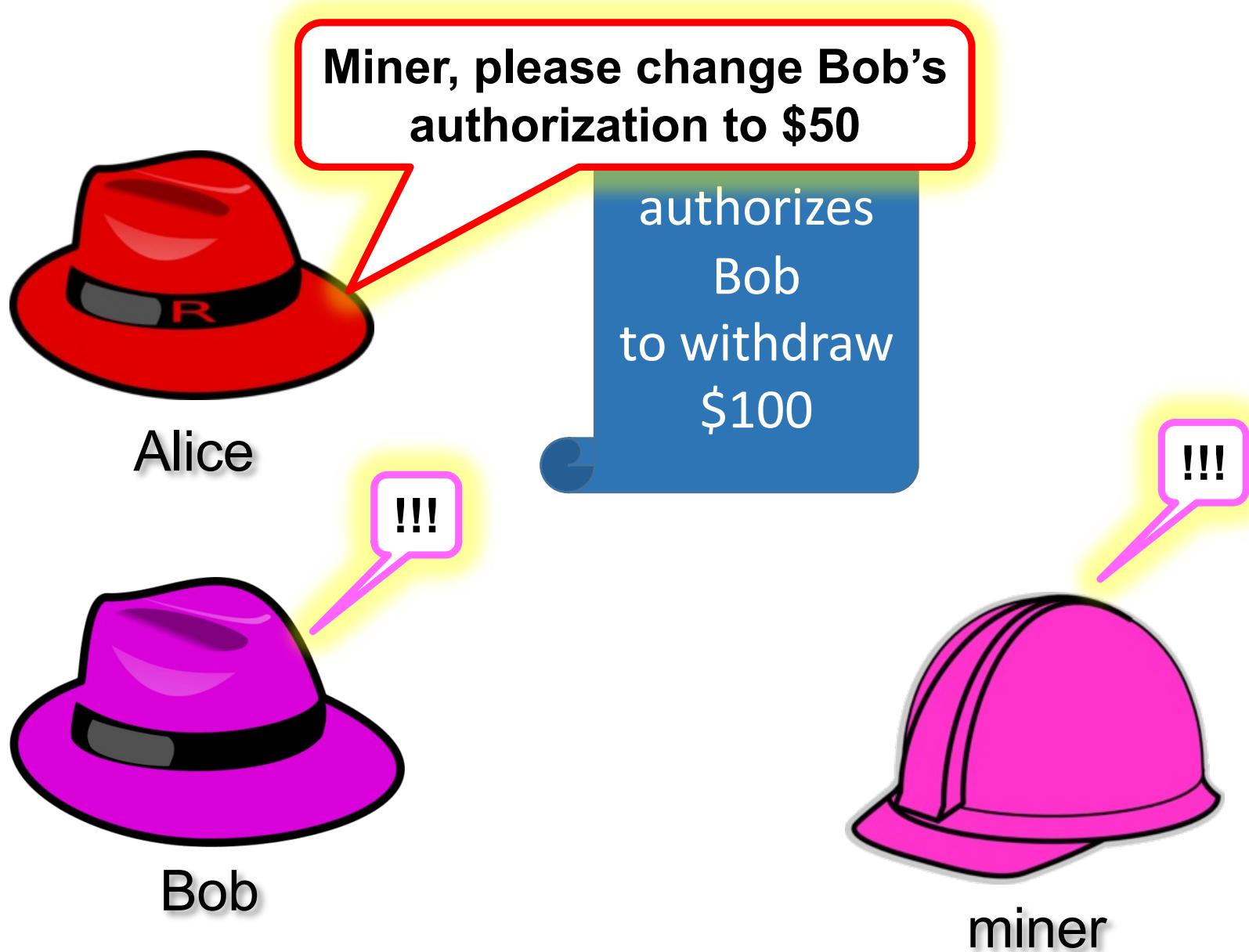


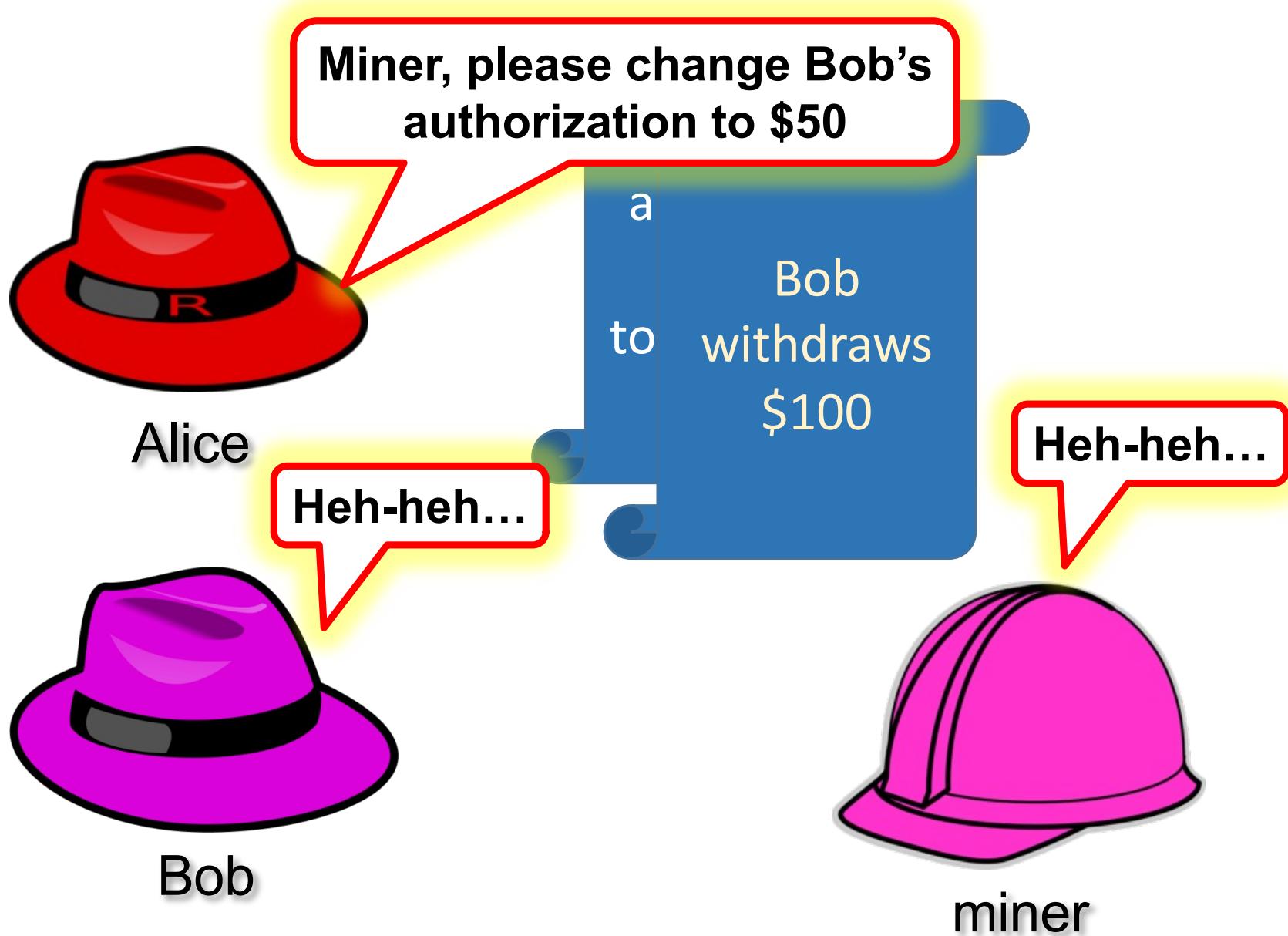
Bob

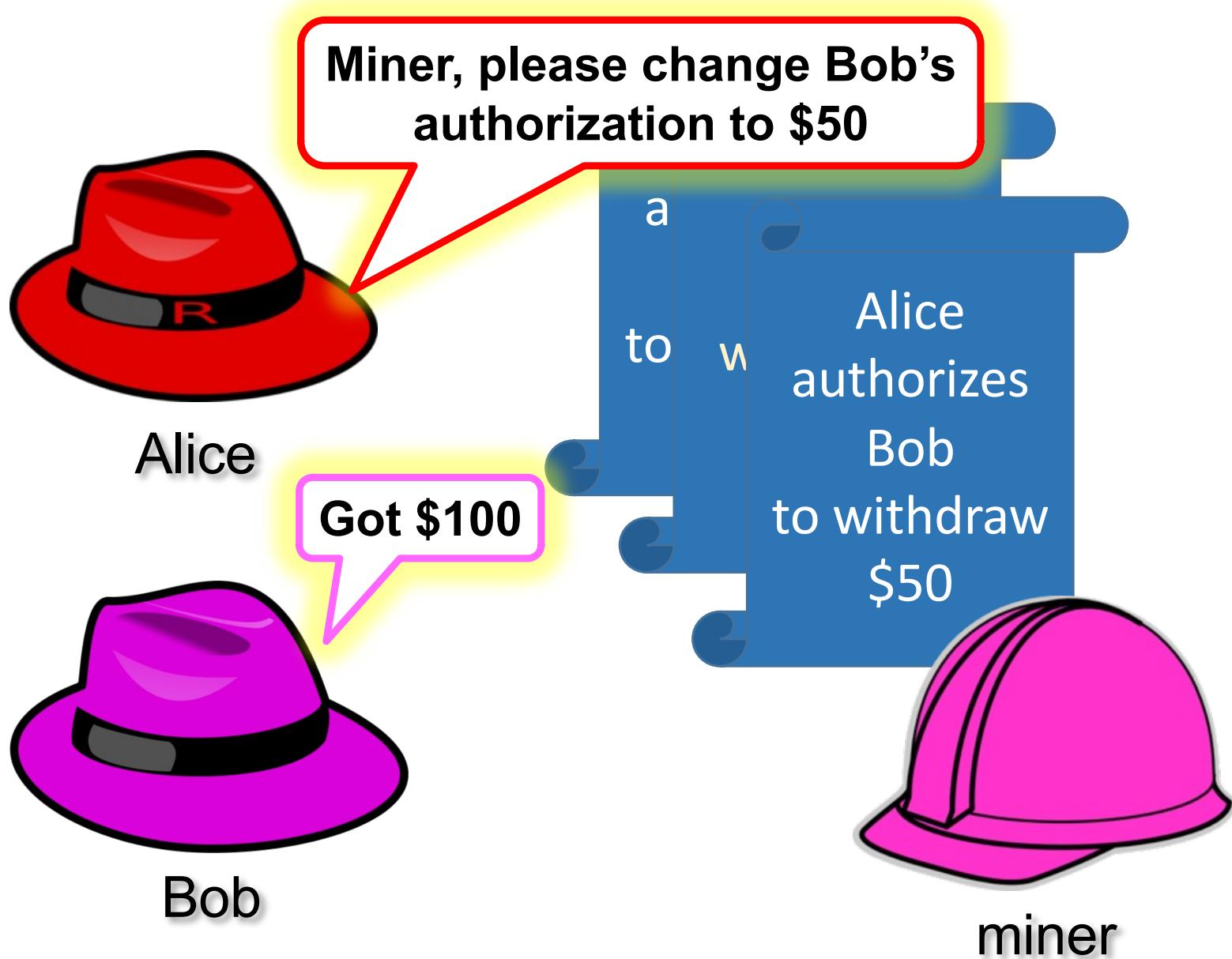
Think of me as an
adversarial scheduler

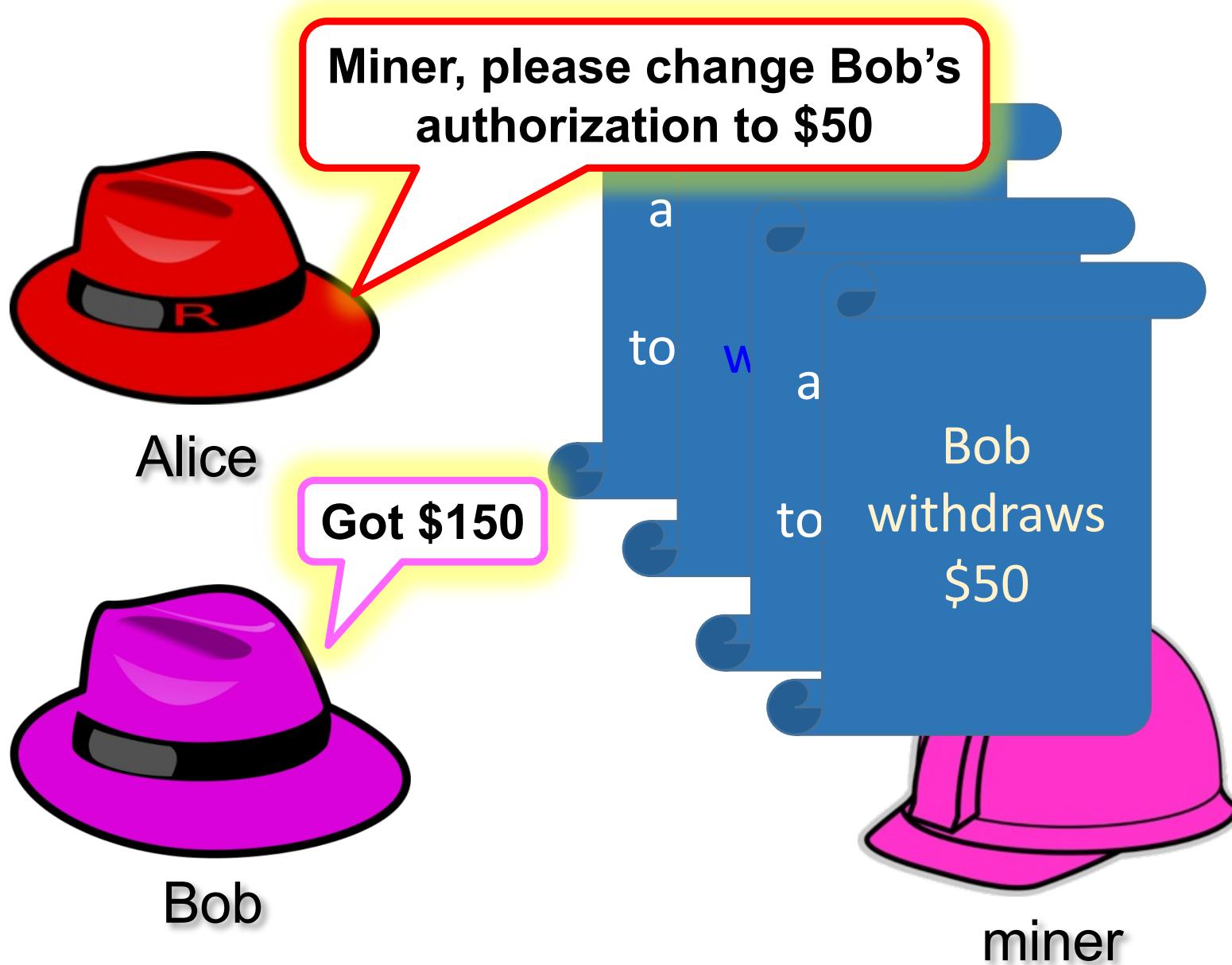


miner









The problem

```
}

function approve(address _spender, uint256 _value) returns (bool success) {
    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
    return true;
}
```

```
function allowance(address _owner, address _spender) constant returns (uint256 remaining) {
```

Problem is overwriting
(instead of an atomic
Read-modify-write)

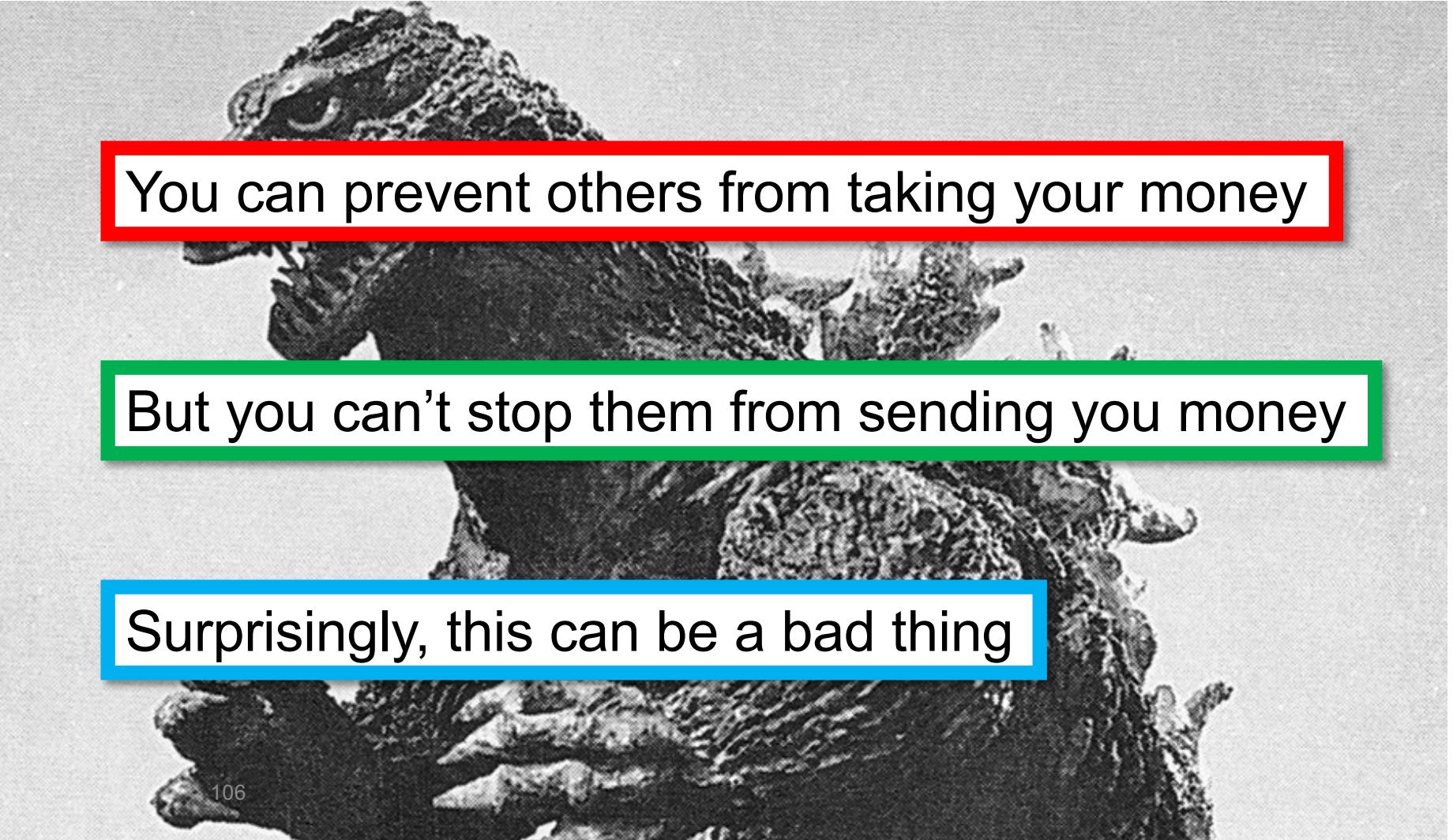
The fix

```
}

function approve(address _spender, uint256 _value) returns (bool success) {
    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value),
    return true;
}
```

- (1) Could require approval to be: 100 → 0 → 50.
(2) Standard implementation of ERC20 created
increaseAllowance() and decreaseAllowance()
to prevent this

Unexpected Ether Attack



You can prevent others from taking your money

But you can't stop them from sending you money

Surprisingly, this can be a bad thing

Sending Ether

```
destination.foo.value(1000)(arg)
```

send 1000 wei via
payable named function

```
contract destination {  
    function foo(uint arg) public payable {  
        ...  
    }  
}
```

Sending Ether

```
destination.transfer(1000);
```

send 1000 wei via
payable fallback function

```
contract destination {  
    fallback() external payable {  
        // Code to execute when Ether is sent to the contract without a specific function call  
    }  
}
```

Controlling Ether Receipt

Ether receipt triggers ...

Function, named or fallback

Possible to refuse ether by

reverting (throwing)

not being payable

Ether You Cannot Refuse

```
selfdestruct(destination);  
  
contract destination {  
    ...  
}  
send balance without  
calling destination code  
can forcibly send Ethers to even the  
contract that does not implement the  
receive or fallback function
```

Example

Simple gambling game

Sequence of milestones (amounts)

Players send small ether amounts ...

First player to reach each milestone wins!

```
[ ] noname01.pas [ ]  
contract EtherGame {  
    uint public targetAmount = 5 ether;  
    address public winner;  
    function play() public payable {  
        require(msg.value == 1 ether, "1 Ether only");  
        uint balance = address(this).balance;  
        require(balance <= targetAmount, "Game is over");  
        if (balance == targetAmount) {  
            winner = msg.sender;  
        }  
    } ...  
* 56:1 =
```

```
[ ] noname01.pas [ ]  
contract EtherGame {  
    uint public targetAmount = 5 ether;  
    address public winner;  
    function play() public payable {  
        require(msg.value == 1 ether, "1 Ether only");  
        uint balance = address(this).balance;  
        require(balance <= targetAmount, "Game is over");  
        if (balance == targetAmount) {  
            winner = address(this);  
        }  
    } ...  
}* 56:1 =
```

```
[ ] noname01.pas [ ]  
contract EtherGame {  
    uint public targetAmount = 5 ether;  
    address public winner;  
    function play() public payable {  
        require(msg.value == 1 ether, "1 Ether only");  
        uint balance = address(this).balance;  
        require(balance >= targetAmount, "Game is over");  
        if (balance == targetAmount) {  
            winner = msg.sender;  
        } ...  
    } ...  
}  
* 56:1 =
```

```
[ ] noname01.pas [ ]  
contract EtherGame {  
    uint public targetAmount = 5 ether;  
    address public winner;  
    function play() public payable {  
        require(msg.value == 1 ether, "1 Ether only");  
        uint balance = address(this).balance;  
        require(balance <= targetAmount, "Game is over");  
        if (balance == targetAmount) {  
            winner = msg.sender;  
        }  
    } ...  
56:1
```

Player wins if it hits target amount

```
[ ] ━━━━━━ noname01.pas ━━━━━━ [ ]  
contract EtherGame {  
    uint public targetAmount = 5 ether;  
    address public winner;  
    function play() public payable {  
        require(msg.value == 1 ether, "1 Ether only");  
        uint balance = address(this).balance;  
        require(balance <= targetAmount, "Game is over");  
        if (balance == targetAmount) {  
            winner = msg.sender;  
        }  
    }  
}
```

If adversary transfers 0.0001 ether via
selfdestruct(), this.balance
becomes slightly off, & comparison fails!

The attack

1. Deploy the Attack contract specifying the EtherGame contract address as the contract deployment argument
2. Supply some Ethers for attacking and invoke the Attack.attack() function

```
contract Attack {  
    address immutable eGame;  
  
    constructor(address _eGame) {  
        eGame = _eGame;  
    }  
  
    function attack() external payable {  
        require(msg.value != 0, "Need some Ether to attack");  
  
        address payable target = payable(eToken);  
        selfdestruct(target);  
    }  
}
```

Prevention

Never rely on exact value of `this.balance`

Replace `==` with `>=` or range `[5,6[`

Recommended reading

<https://docs.soliditylang.org/en/v0.8.19/introduction-to-smart-contracts.html>

<https://docs.soliditylang.org/en/v0.8.19/solidity-by-example.html>

<https://blog.chain.link/reentrancy-attacks-and-the-dao-hack/>

<https://medium.com/valixconsulting/solidity-smart-contract-security-by-example-08-unexpected-ether-with-forcibly-sending-ether-e13be2c6b985>

<https://www.adrianhetman.com/unboxing-erc20-approve-issues/>

Further reading (comprehensive list of attacks)

<https://blog.sigmaprime.io/solidity-security.html>

Acknowledgements

Slides adapted from Maurice Herlihy, Brown University, available under CC BY-NC 4.0 (<https://creativecommons.org/licenses/by-nc/4.0/>)

Food Traceability with Blockchain

Miguel L. Pardal, INESC-ID / IST

Friday, March 28th, 2025

Invited lecture for
Highly Dependable Systems/
Sistemas de Elevada Confiabilidade
course



Prof. Dr. Miguel L. Pardal

- PhD at IST in 2014
 - "Scalable and Secure RFID Data Discovery"
- Visiting Student at MIT in 2009
- Lecturer at Técnico since 2002
- Visiting Scholar at TU Munich in 2017 and 2024
- Teaching/Research Interests:
 - Cybersecurity applied to Internet of Things and Cloud Computing
 - Enterprise Integration and Blockchain



<http://web.tecnico.ulisboa.pt/miguel.pardal>

Agenda

- 1 Motivation and Overview
- 2 Traceability
- 3 Hyperledger Fabric
- 4 FoodSteps
- 5 Research in Progress
- 6 Conclusion

1.

Motivation and Overview

Food



- Fundamental for survival and quality of life
 - Proper nutrition, food security, and environmental sustainability
 - Essential for the well-being of current and future generations
- Access to high-quality information about food products empowers consumers to make informed purchase decisions
 - Transparency in food production fosters trust

Image credits: <https://wisecrop.com/>

Challenges in Agri-Food

- Producers and suppliers need to balance cost efficiency with quality and sustainability
 - Quality and price at the same time
 - “Bom e barato!”
- Food safety requires recalls
 - Contamination
 - Fraud
- Recalls require traceability

Track & Trace queries



- *Where is the object now?*
 - Track
 - Identify the current location of an object
- *Where has the object been at?*
 - Trace
 - Retrieve the complete history of an object's movements and transformations

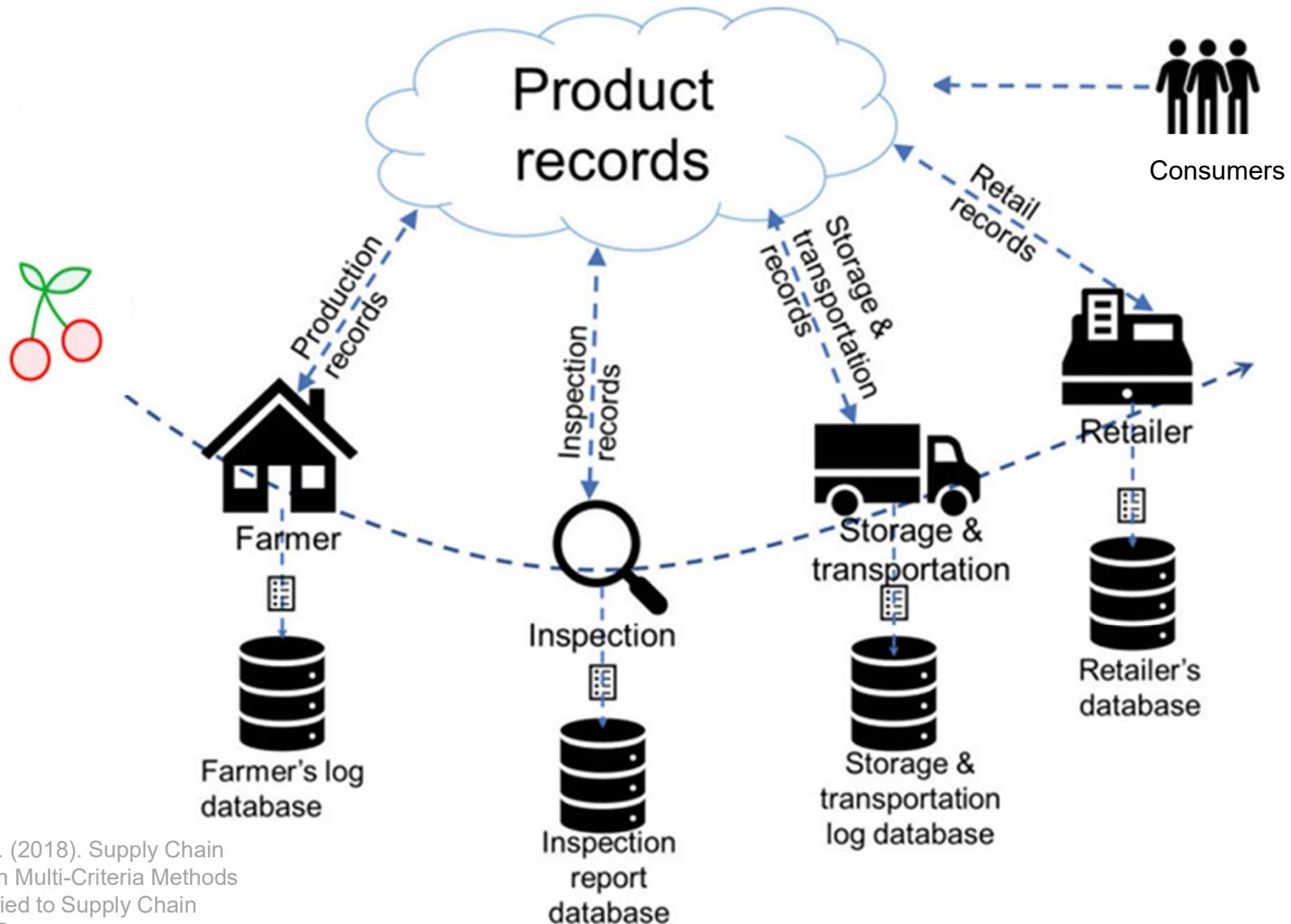


Image credits: Cui, Y. (2018). Supply Chain Innovation with IoT. In Multi-Criteria Methods and Techniques Applied to Supply Chain Management. IntechOpen.

Characteristics of Food Supply Chain

- Flows of physical goods between trading partners
 - Food products
 - From producer to consumer
 - Supported by Enterprise systems
- Scattered locations
 - Distributed system
- Open system
 - No single authority
 - No dominant partner



Who should see the data?

Eurich et al. survey

- 16 interviews with organizations from Europe and USA
- Data sharing risks and rewards
 - Companies fear giving competitors data that they currently do not possess
- To promote use of traceability, we need:
 - Fine-grained access control policy enforcement
 - Share useful results without revealing all data
 - Rely on a data sharing trusted third party



What about
Blockchain?

Eurich, M., N. Oertel, and R. Boutellier (2010, December). The impact of perceived privacy risks on organizations' willingness to share item-level event data across the supply chain. Journal of Electronic Commerce Research 10(3-4), 423–440. ISSN: 1572-9362.

Blockchain for Supply Chains?

- The use of Blockchain technology could ensure the integrity of collected data
 - provide a reliable source of information
- Blockchains are public ledgers, by default
 - Examples: Bitcoin, Ethereum
 - Open to any participant
- Downside: expose data to anyone
 - or, at least, metadata
- Data providers lose ownership of data
 - so, they do not participate!

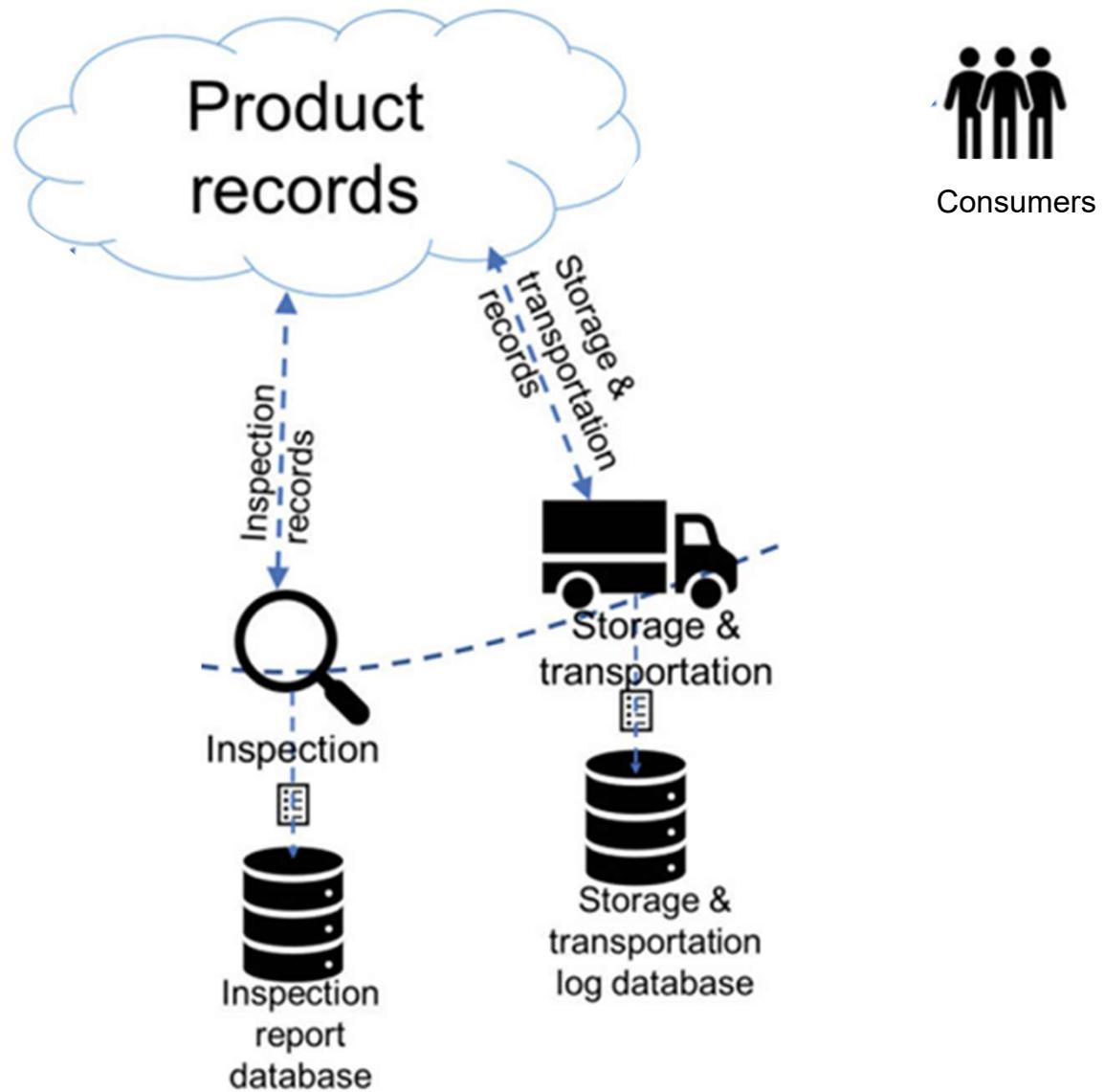
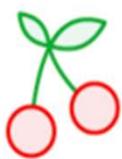


Image credits: Cui, Y. (2018). Supply Chain Innovation with IoT. In Multi-Criteria Methods and Techniques Applied to Supply Chain Management. IntechOpen.

Blockchain Permissions

- Permissionless (Public Blockchain)
 - Open to anyone
 - no central authority
 - Examples: Bitcoin, Ethereum
- Permissioned (Private/Consortium Blockchain)
 - Access is restricted to approved participants
 - controlled governance
 - Example: Hyperledger Fabric

Blockchain for Supply Chain Consortium

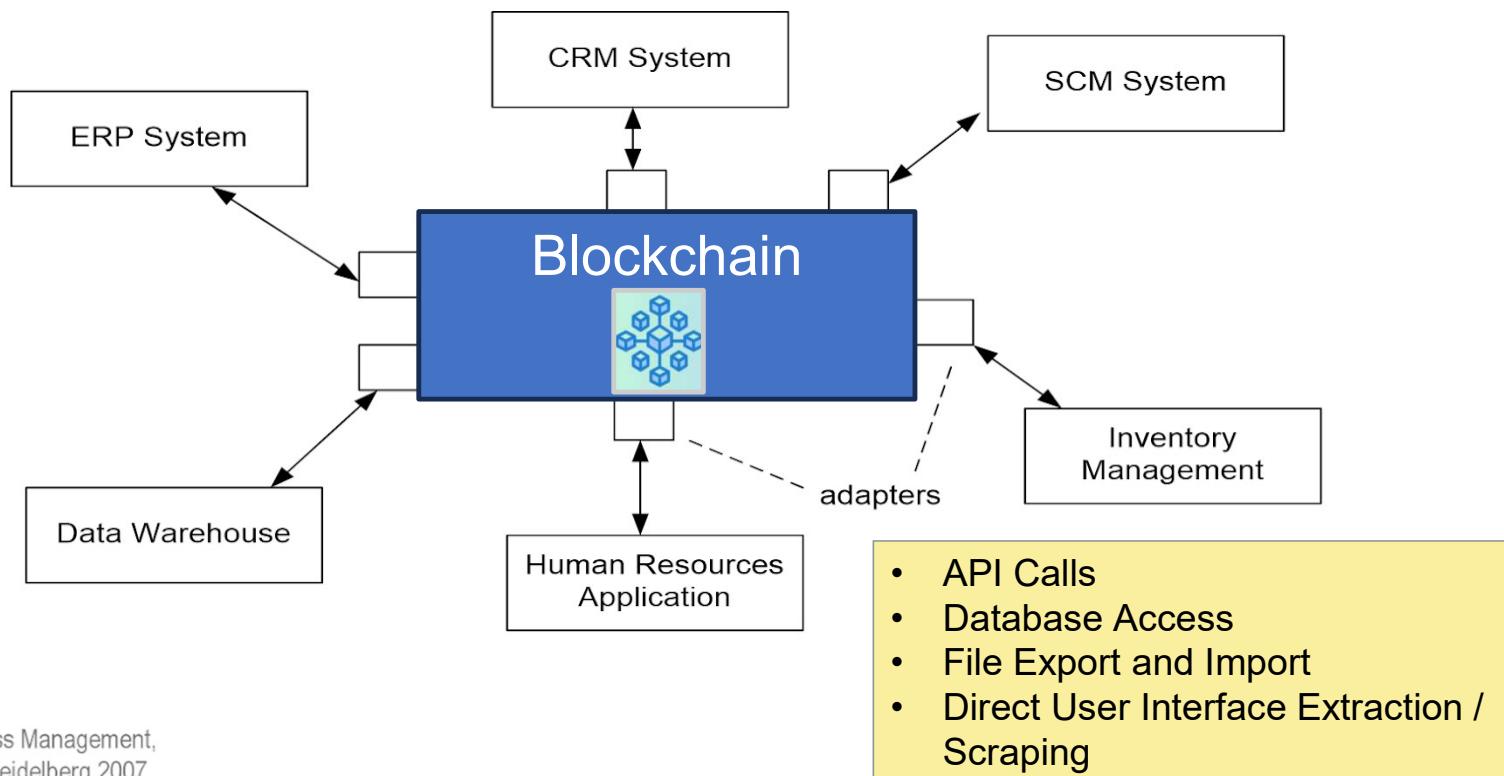
- Consortium
 - Trading partners
 - Include many producers, distributors, retailers
- System is controlled by a **consortium of partners**
 - or technology providers appointed to represent them
 - not just by a single entity
- Blockchain is used both as...
 - Notary
 - Integration Hub

Blockchain as a Notary

- The blockchain can be used to “notarize” data that reflects the state of the supply chain at a given moment
 - without relying on a single partner
- A notary is a service or system that verifies and certifies the authenticity of documents, transactions, or records
- Digital notaries ensure the integrity, immutability, and timestamping of electronic records
 - using cryptographic techniques
 - to provide proof of existence and authenticity at a given time



Blockchain as Integration Hub



Potential benefits of Blockchain

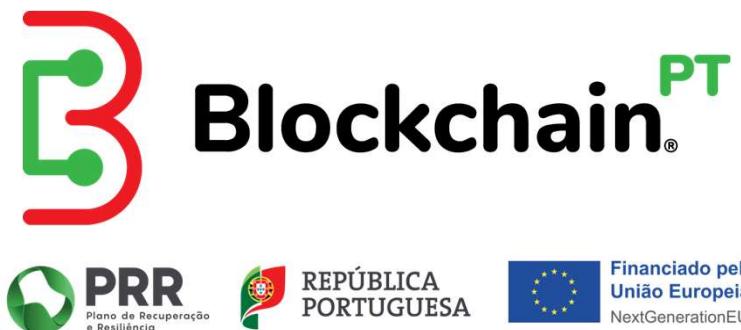
- Decentralized
 - Blockchain is governed by a consortium of trading partners
- Secure
 - New information is only added to the Blockchain after being validated and accepted by consortium partners
- Transparent
 - All partners can view information on the Blockchain according to the permissions agreed upon by the consortium
- Efficient
 - May reduce the need for duplicate verifications

Potential pitfalls of Blockchain

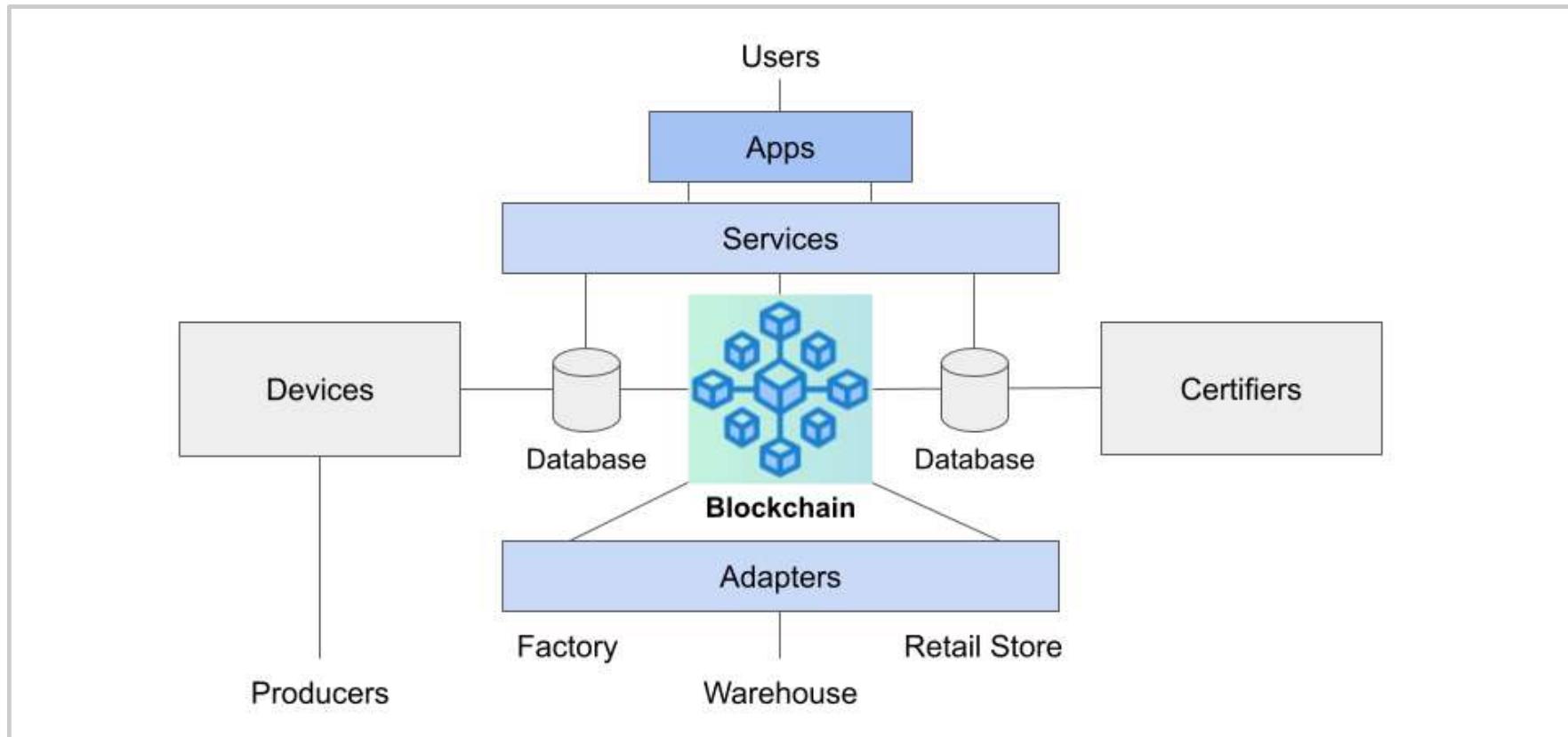
- Partner adoption
 - Producers, suppliers, and retailers need to adopt new tools
 - Resistance to change and lack of technical expertise can hinder adoption
 - Integrating blockchain with legacy systems can be complex and costly
- Performance and scalability
- Confidentiality
 - Partners may be reluctant to share business information on a blockchain
 - even in a permissioned network
- Accuracy
 - While blockchain ensures data immutability once recorded, the accuracy of the input data can be questioned
 - e.g. reliable sensors?
 - c.f. the “blockchain oracle” problem

FoodSteps: Blockchain-based Food Traceability System

- FoodSteps is a solution under development for the reliable registration and sharing of information about food products
 - Designed to provide trustworthy data about food traceability, ensuring transparency and integrity in the food supply chain
- Part of ongoing Portuguese Project with European Funds
 - Agenda "Descentralizar Portugal com Blockchain"



FoodSteps overview



Flagship use case: Cherries from Fundão



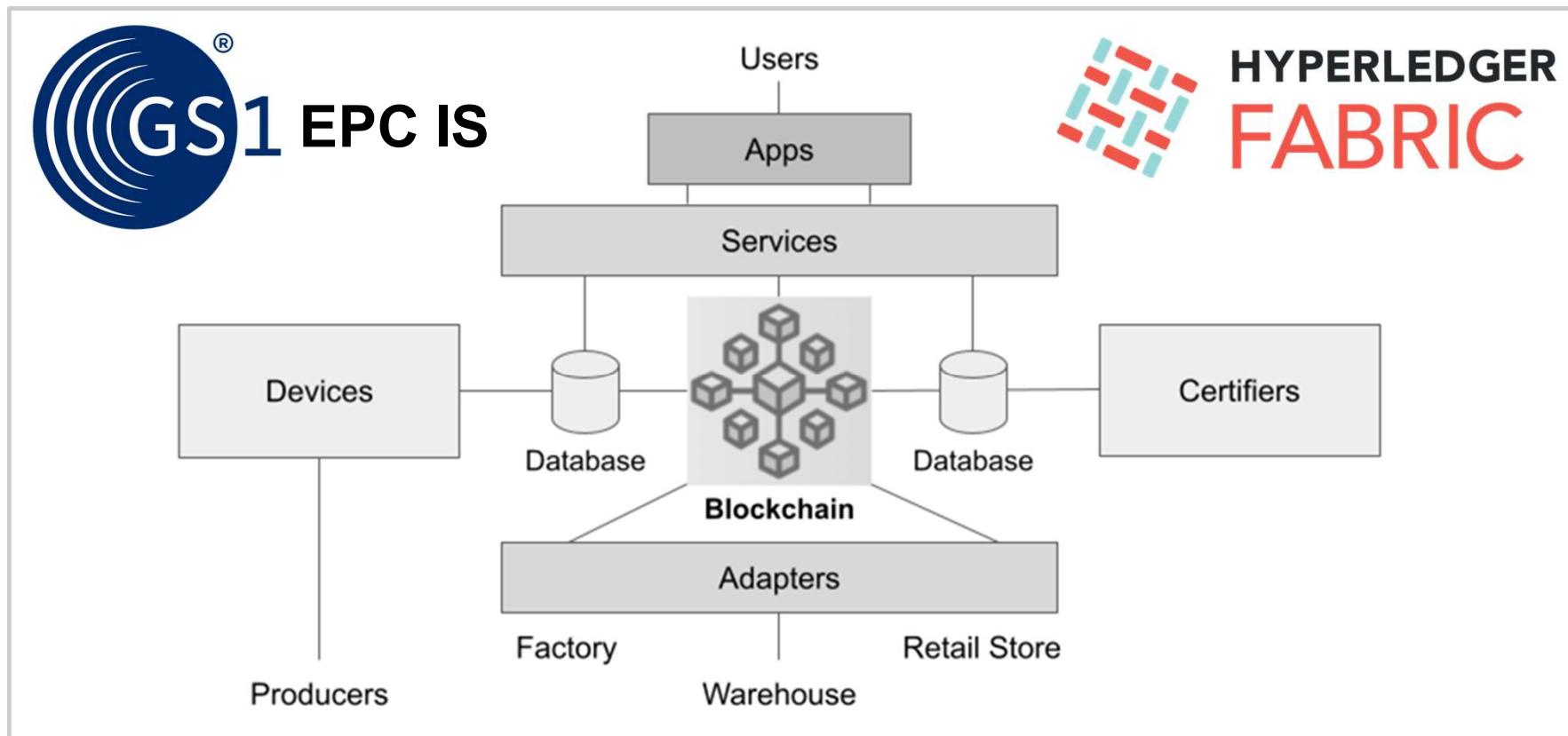
Cerfundão



SENSE
FINITY



But first, we need some background



2.

Traceability

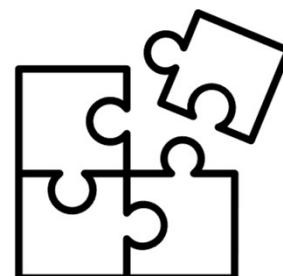


What is traceability?

- The ability to track an object's history, location, and status throughout the supply chain
- Goal: ensure transparency, authenticity, and quality control
- The physical world is in constant change...
 - The world representation in the information system is (always) outdated
- The sense-control loop needs to be tightened
 - to achieve an accurate mapping between physical and virtual worlds

Traceability Meta-Model

- Segments and Junctions
- We can have segments of observations data
 - of the same object
 - using the same identifier
 - when the identifier changes, the segment ends
- We need junctions to connect segments
 - to keep the trace



Identify, Capture, Share

- Objects need unique identifiers
 - Text
 - Barcode
 - RFID (Radio-Frequency Identification) Tag
- Identification requires automated or assisted reading points
 - each observation point must be linked to a specific location
- Data sharing requires a storage system
 - with query capabilities





GS1 standards for identification

- GS1 is an international non-profit organization responsible for global standards for business
 - since 1974
 - including barcode and RFID standards
 - provides a common framework to uniquely identify products, locations, and assets in the supply chain
- Examples:
 - GTIN (Global Trade Item Number)
 - SGTIN (Serialized Global Trade Item Number) – adds serial number to GTIN
 - SSCC (Serial Shipping Container Code)
 - GLN (Global Location Number)



What is EPC IS?

- Electronic Product Code Information Services (EPC IS) is a standard for recording and sharing traceability data about physical objects
 - Designed for the use of RFID or barcode technology
- Developed within the GS1 EPCglobal Framework
- EPC IS enables secure data capture and retrieval across supply chains

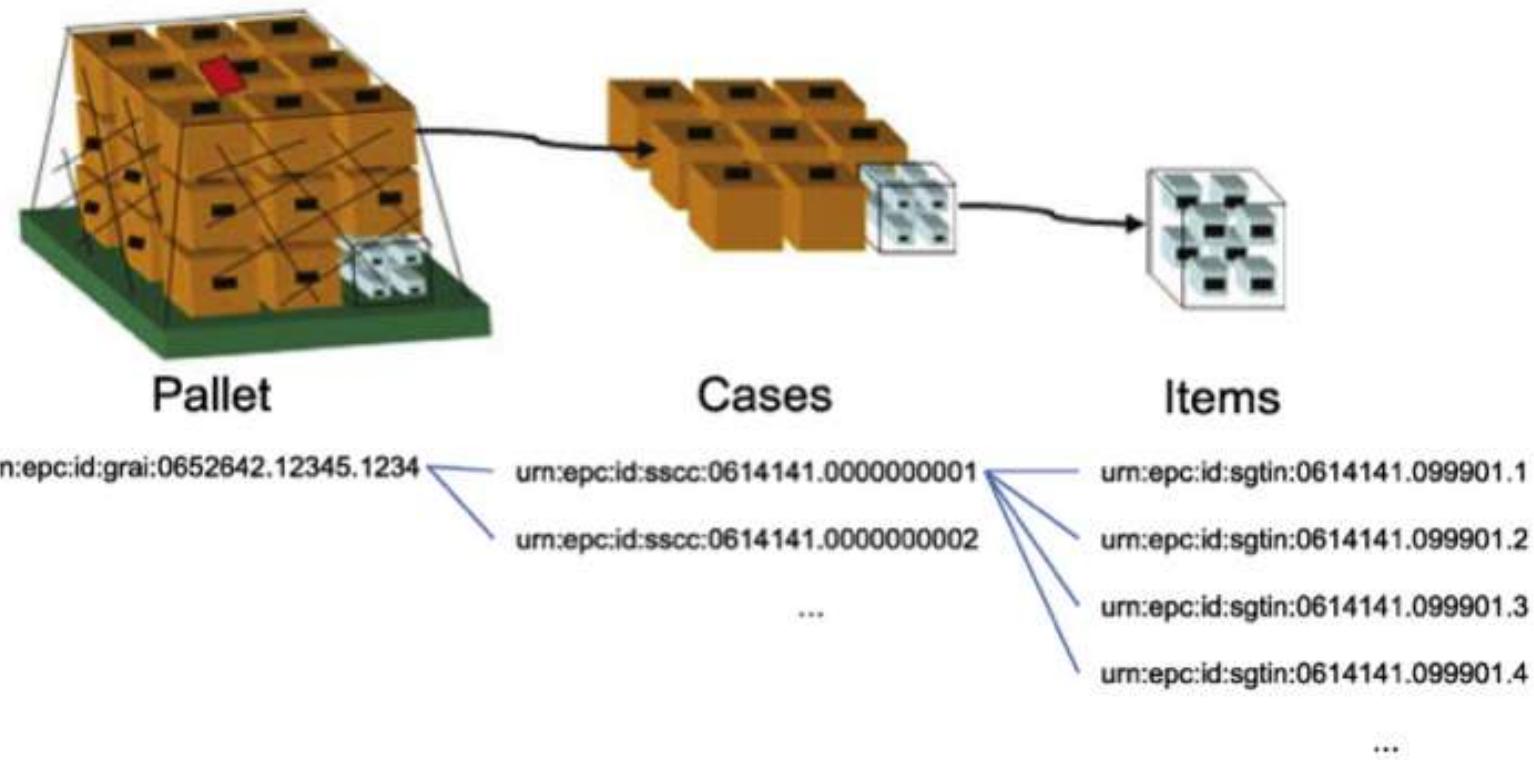
Object Event example

```
{  
  "@context": "https://gs1.github.io/EPCIS/epcis-context.jsonld",  
  "@type": "ObjectEvent",  
  "eventTime": "2024-04-26T10:00:00Z",  
  "eventTimeZoneOffset": "+01:00",  
  "epcList": ["urn:epc:id:sgtin:4012345.012345.6789"],  
  "action": "OBSERVE",  
  "bizStep": "urn:epcglobal:cbv:bizstep:receiving",  
  "disposition": "urn:epcglobal:cbv:disp:in_process",  
  "readPoint": {  
    "id": "urn:epc:id:sgln:4012345.00001.0"  
  },  
  "bizLocation": {  
    "id": "urn:epc:id:sgln:4012345.00001.0"  
  }  
}
```

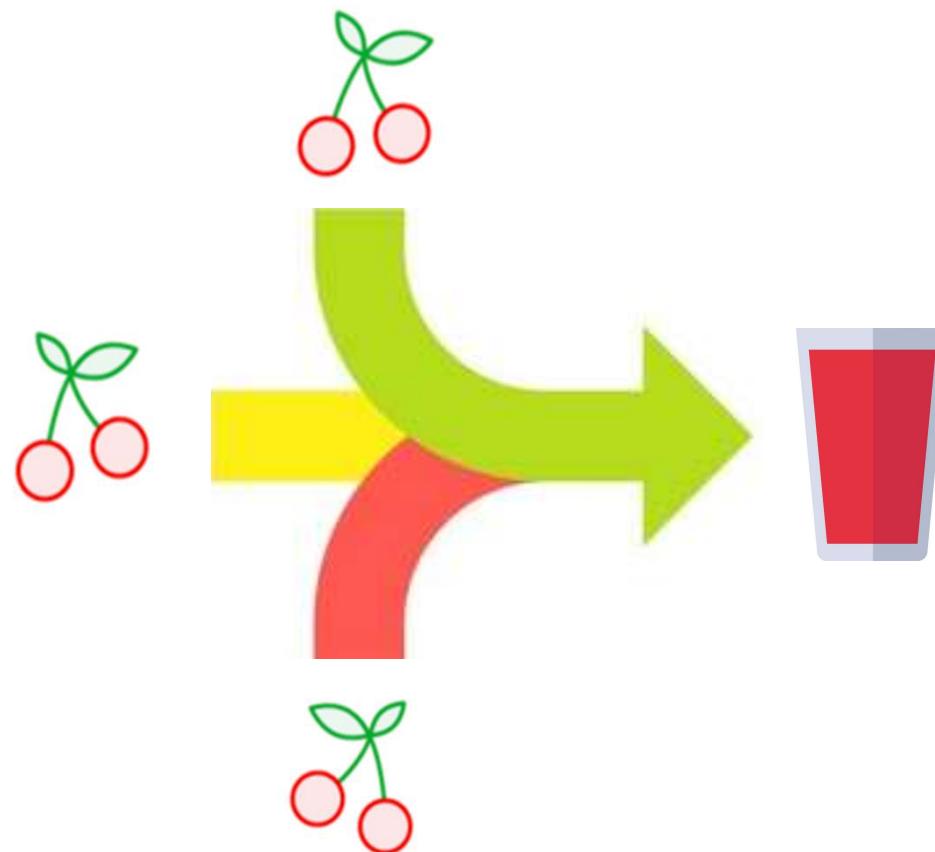
Types of EPC IS events

- Object event
 - Captures an observation of one or more EPCs
 - Example: a pallet of goods scanned at a warehouse dock
- Aggregation event
 - Records grouping/ungrouping of objects
 - Example: a shipment container with multiple boxes
- Transaction event
 - Associates object identifiers with a business transaction
 - Example: linking a product scan to a sales invoice
- Quantity event
 - Captures bulk item transactions
 - Example: recording 500 units moved between locations

Aggregation can have multiple levels



Transformation is irreversible



Chapter 3.

Hyperledger Fabric



Hyperledger Fabric (HLF)



- Permissioned blockchain platform
 - Designed for enterprise applications
 - Focus on privacy / confidentiality
- Modular architecture
 - organized into independent, interchangeable components
 - each performs a specific task and can be modified, replaced, or upgraded without affecting the entire system
- Scalable and efficient consensus mechanisms
 - By default, Crash-Fault Tolerant (CFT)
 - Not Byzantine Fault Tolerant (BFT)

Permissioned Blockchain

- Restricted Access
 - Only verified partners with permissions can interact with the blockchain
- Contrast with Public / Permissionless Blockchain
 - where any entity can access data
- Permissioned creates an authentication step
 - Exclusive Participation
 - Ensures only authorized entities contribute and retrieve information

A brief history of HLF



- 2017
 - HLF v1.0 - Permissioned blockchain, modular architecture, smart contracts
 - Ordering with Kafka
- 2019
 - HLF v1.1 _ v1.4 – add Private Data Collections (PDCs)
 - Chaincode lifecycle management
- 2020
 - HLF v2.0 – add Decentralized Chaincode Governance (approval process for upgrades)
 - Ordering with Raft
 - Enhanced private data sharing between peers
 - External chaincode launcher
- 2025
 - HLF v2.5 remains the long-term support (LTS) release for production users
 - v3.0 adds Byzantine Fault Tolerant (BFT) Ordering Service with the SmartBFT protocol
 - Should enable the network to remain operational even if some ordering nodes act maliciously or unpredictably

Key Concepts in HLF



- Data Assets
- Chaincode
- Ledger
- World State
- Channels
- Peers
- Consensus and Ordering

Data Assets in HLF

- Represent real-world items such as:
 - Physical assets
 - Goods
 - Digital assets
 - Contracts
 - Intellectual property
- Stored as key-value pairs in the world state database
- Modified through chaincode transactions

Chaincode

- also known as Smart Contracts
- Chaincode is a program that modifies assets
 - Written in Go, Java, or Node.js
- Executes transactions on the blockchain
- Can be invoked by clients using HLF libraries
- Code actually runs “off chain”
 - by each organization
 - but chaincode outcomes must agree for transaction to be valid

Ledger

- Sequential record of transactions
 - Append-only
- Only orderer nodes can add transactions to ledger
 - after reaching consensus on the next block
- May contain invalid transactions that were submitted
 - but these are ignored by peer nodes

World State

- Auxiliary key-value database, exists in peer nodes to enable efficient chaincode execution
 - Provide fast access to current asset values
- Storage
 - LevelDB
 - default key-value store
 - CouchDB
 - also a key-value store
 - supports views, indexes for complex queries on JSON data
- Contains the result of applying all valid transactions currently stored in a peer's local instance of the ledger
 - Kept consistent with ledger

World State and Ledger synchronization

- Atomic Updates
 - When a transaction is committed, both the Ledger and the World State DB are updated atomically, to ensure consistency
- Transaction Validation and Endorsement
 - Transactions are first endorsed and validated before being committed to the ledger
 - Only valid transactions update the World State
- Block Commitment Process
 - When an ordered block is received by a committing peer, it writes the block to the immutable blockchain and, next, updates the World State with the latest asset values
- Deterministic Execution
 - The state updates are derived deterministically from transactions, ensuring all peers compute the same World State
- Crash Recovery
 - If a peer crashes or loses synchronization, it can rebuild the World State from the immutable Ledger, ensuring consistency

Channels (for confidential transactions)

- Enable confidential/private communication between a subset of participants
- Each channel has its own ledger and policies
- Peers participate in specific channels
- Competing businesses can share a blockchain infrastructure (or parts of it) but keep transactions confidential

Peer Nodes (Infrastructure)

- The Peers maintain independent, up-to-date copies of the ledger, execute chaincode transactions and support endorsement
- Organizations may participate in many channels
 - No central authority – only channel participants need to know it exists

Types of Peer Nodes

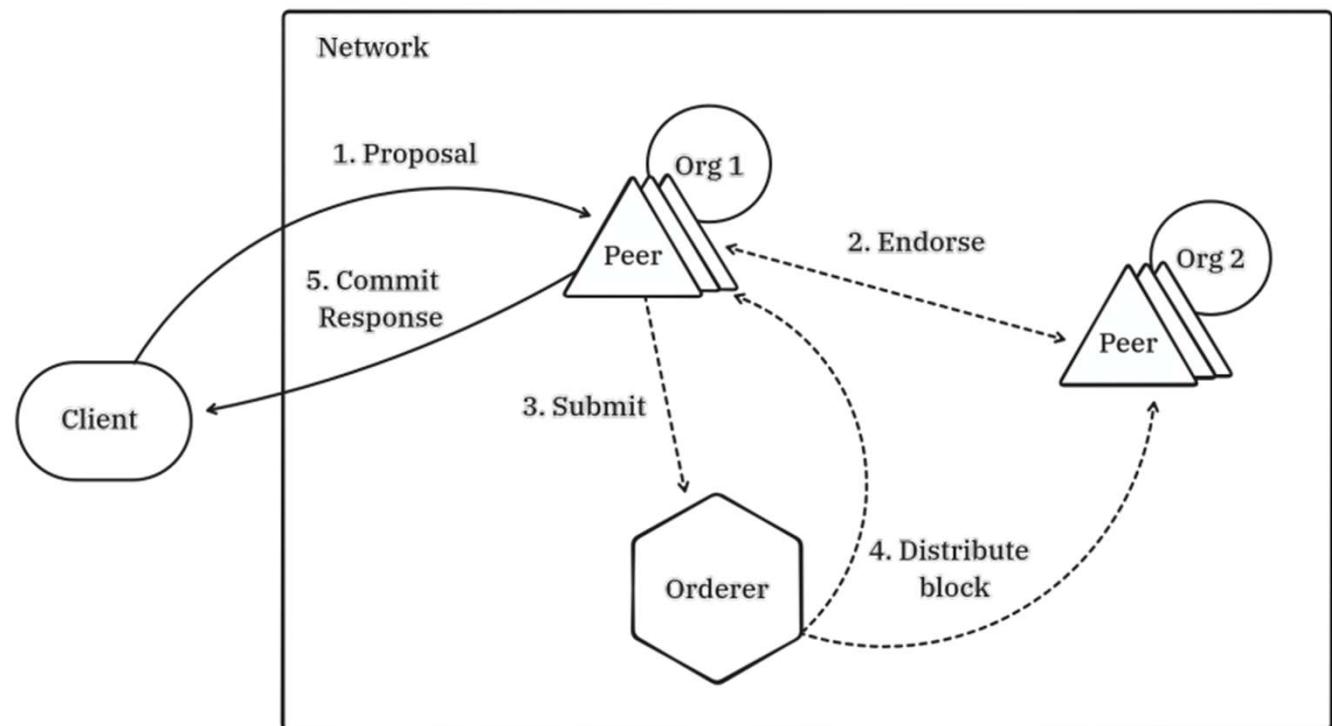
- Endorsing Peers
 - Simulate and validate transactions
- Committing Peers
 - Store transactions in the ledger
- Anchor Peers
 - Peer with well-known network address
 - Used to bootstrap gossip data dissemination between peers
 - Facilitate cross-organization communication
- Gateway Peers
 - Provide service that allows clients to delegate part of transaction process to a peer
 - Offloads some tasks that were previously done by clients

Consensus and Ordering

- HLF uses an ordering service rather than *mining* or *staking* to ensure transaction consistency
 - Proof-of-Authority
 - Pre-approved validators process transactions
 - Their identity is verified with strong authentication
 - The mechanism follows three phases:
 - Endorsement
 - Ordering
 - Validation and Commit

3 phases: execute-order-validate

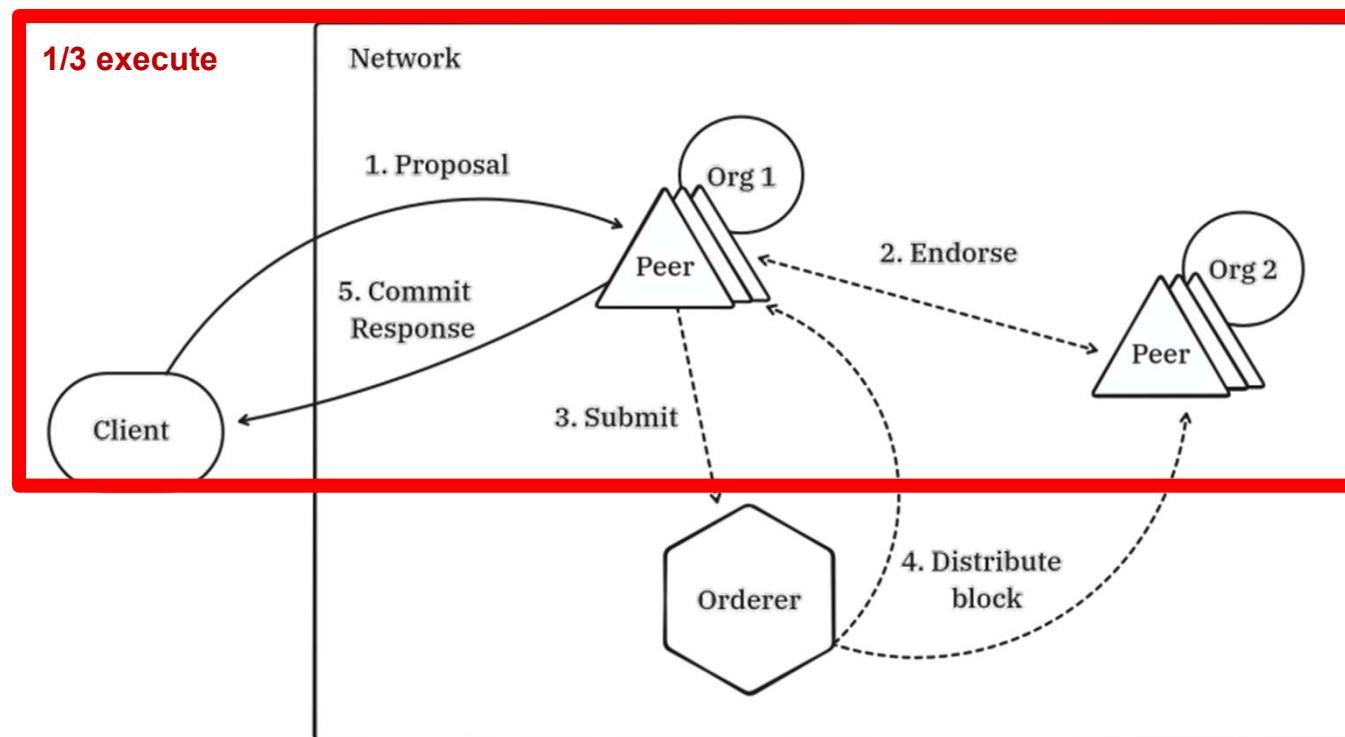
- Execute
- Order
- Validate



Phase 1/3: execute

- Execute
 - Client creates and signs a transaction proposal containing the transaction parameters
 - At a client's request, chaincode is executed in one or more peer nodes
 - Result of executions are transaction endorsements
 - Each is signed by the peer and includes transaction parameters, read assets, written assets, and return value
 - Client collects endorsements to create a transaction which it signs and submits to the ordering service
- Order
- Validate

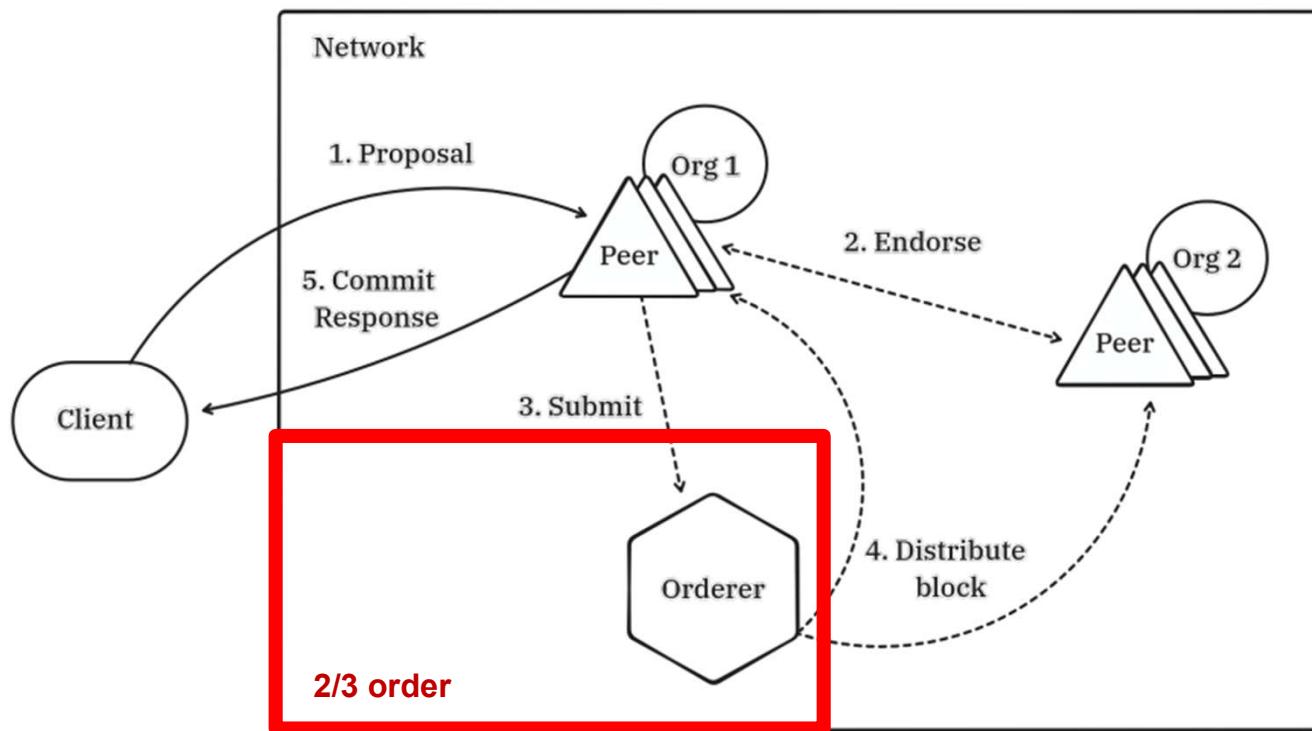
HLF transaction processing



Phase 2/3: order

- Execute
- Order
 - The ordering service receives submitted transactions and groups them into blocks, running a consensus algorithm between the ordering nodes to decide what the next block will be
 - Blocks are final once created and give transactions a total order within the channel
- Validate

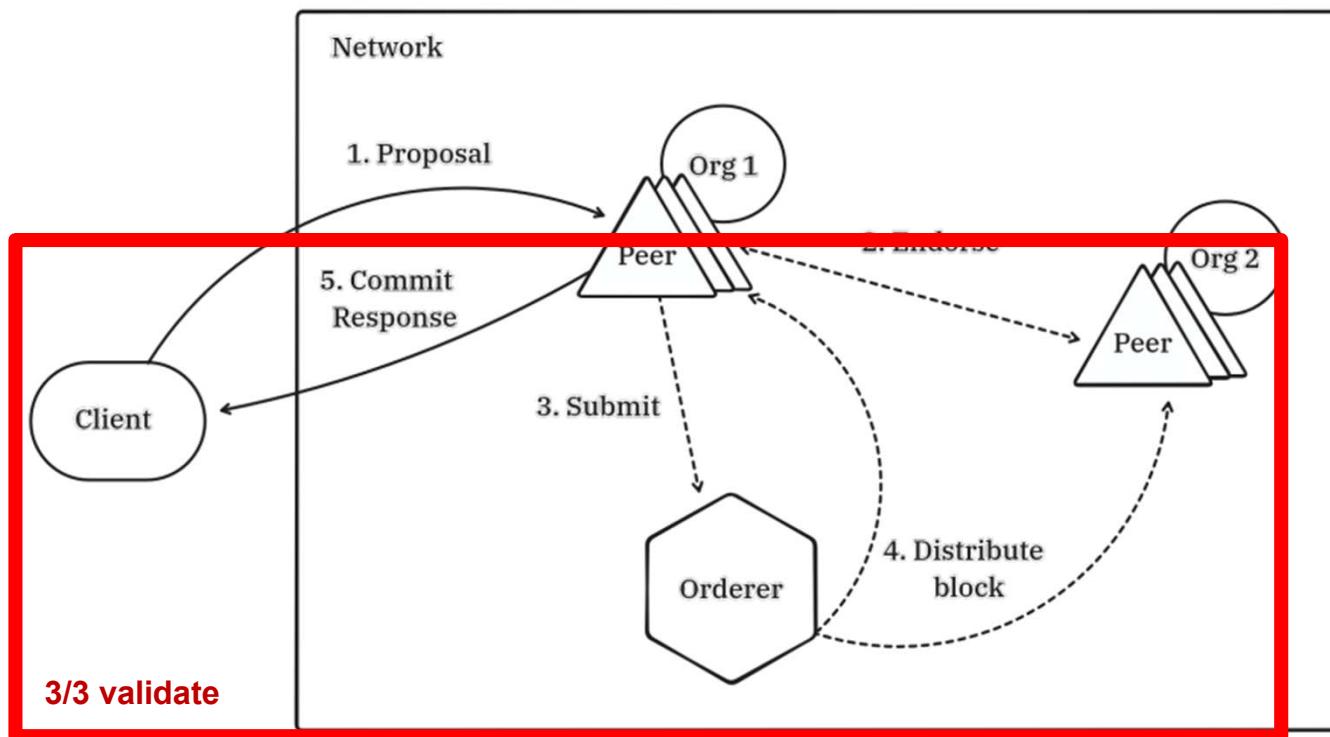
HLF transaction processing



Phase 3/3: validate

- Execute
- Order
- Validate
 - Peers are continuously listening for new blocks created by the ordering service
 - For each new block, the peer will go through each transaction in order and check its validity
 - The assets written by a transaction are applied to the World State if and only if the transaction is valid
 - The validation phase is a deterministic process performed independently by each peer node

HLF transaction processing



Summary of Key Concepts in HLF



- Data Assets
 - Digital representations of tangible and intangible assets
- Chaincode
 - Programs that enforce business rules
- Ledger
 - Immutable record of all transactions
- World State
 - Queriable database with current snapshot of ledger data
- Channels
 - Private sub-networks/ledgers within Fabric for data segmentation
- Peers
 - Nodes that maintain the ledger and execute transactions
- Consensus and Ordering
 - Mechanism for agreeing on transaction order

4.

FoodSteps

Disclaimer:
FoodSteps is a work-in-progress



Image credits: drogatnev, iStock

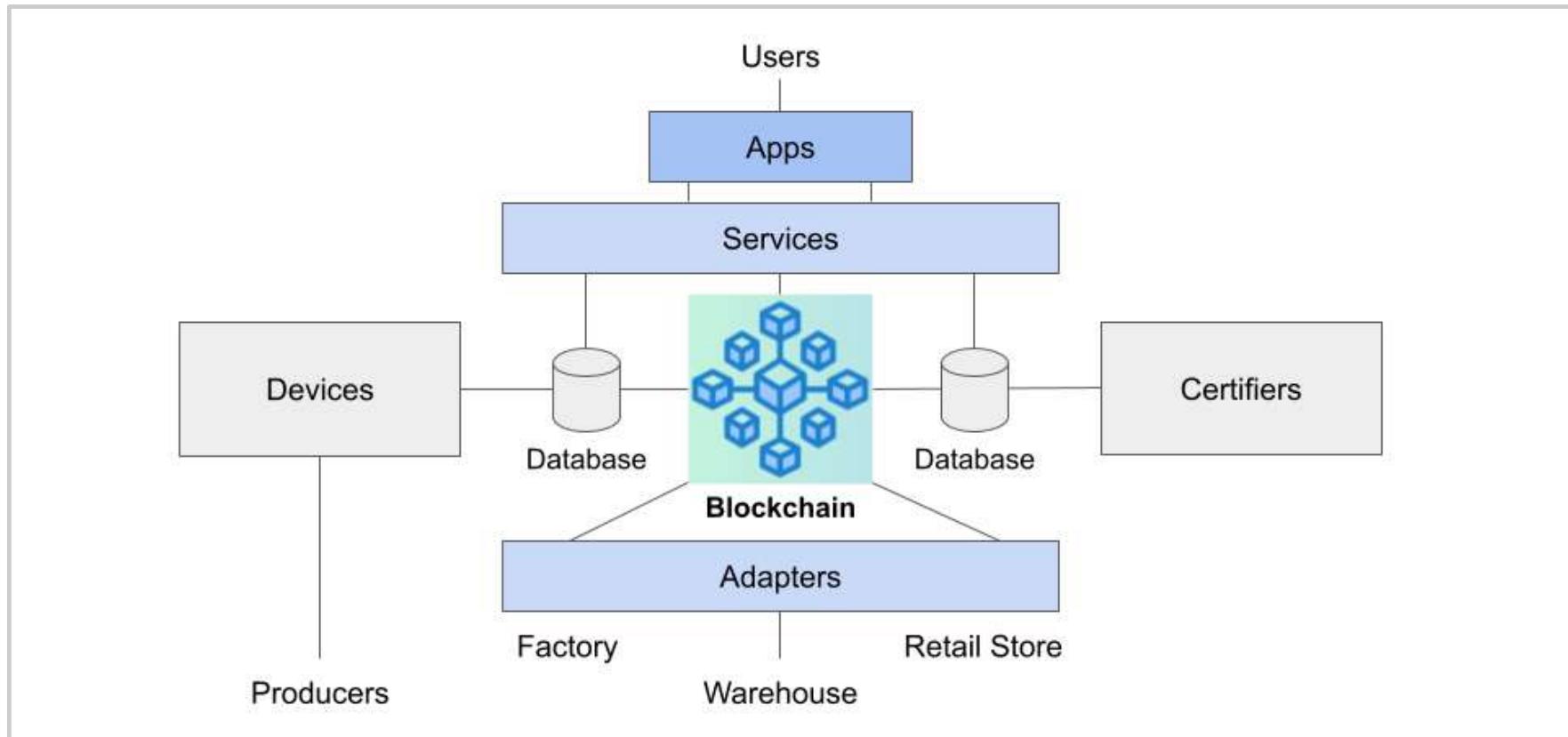
Blockchain Service Providers

- Organizations that represent themselves and others
 - Delegation
- Very important for Small and Medium Enterprise participation
 - Analogy to ISP – Internet Service Providers
- Currently:
 - SF - Sensefinity
 - IID - INESC-ID - Instituto de Engenharia de Sistemas e Computadores: Investigação e Desenvolvimento em Lisboa
 - IPL - Instituto Politécnico de Leiria

Sensor Providers

- Organizations that operate sensors devices
 - “Fleet” management
- Currently:
 - SF - Sensefinity
 - Proprietary
 - Optimized
 - IID - AmBox
 - Open
 - Experimental

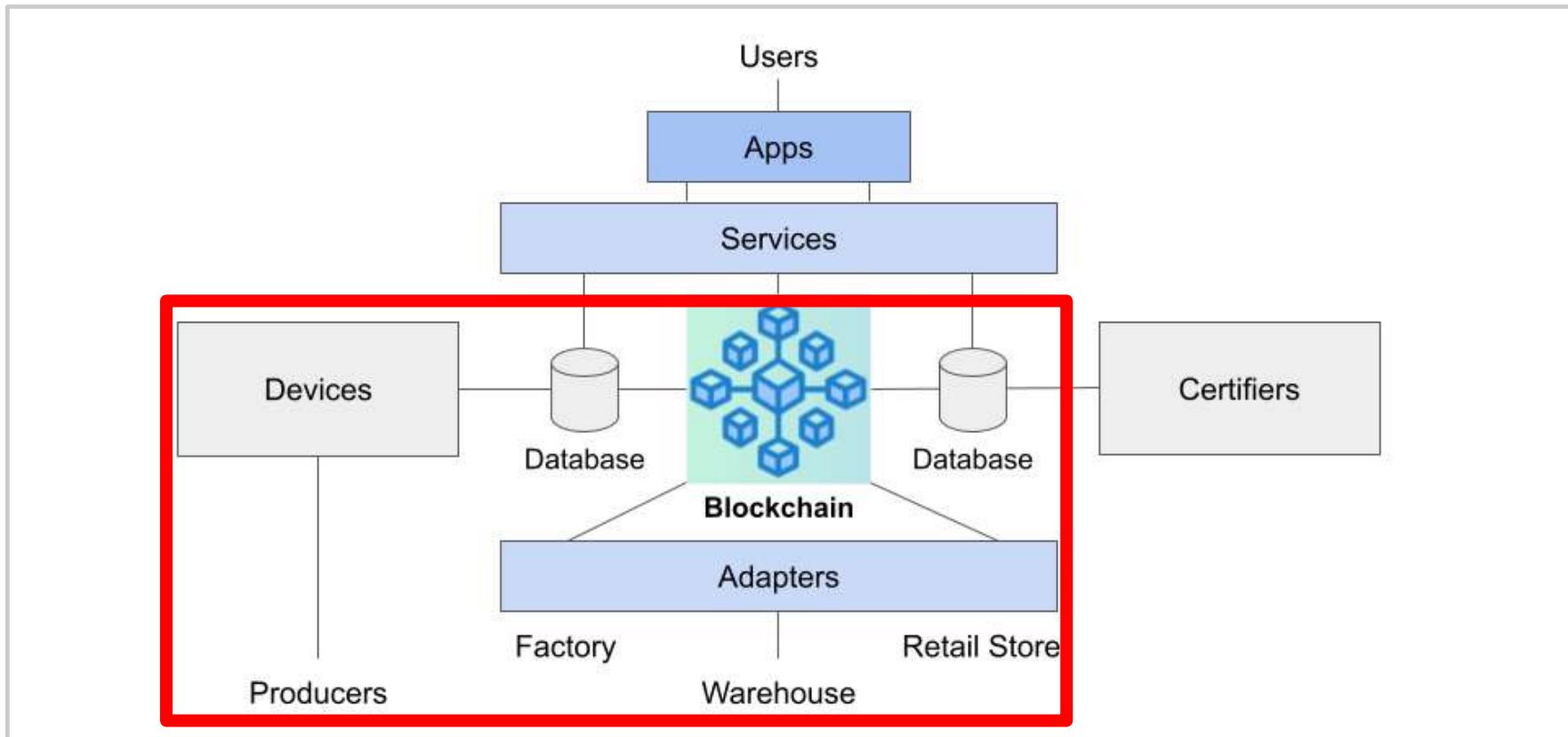
FoodSteps overview



FoodSteps captures data from two “parallel” tracks

- Information Systems
 - What does each trading partner record in its own systems?
 - Which subset of this data is shared?
 - When is it shared?
 - ASAP or only later
 - e.g. after sale
- Sensor Systems
 - What do the devices record?
 - What kinds of sensors exist?
 - Does data have continuity?
 - Does it fit well with existing data or are there discrepancies?

FoodSteps: Adapters and Devices



From farm to fork



Field Sensors

- Automated and reliable capture of sensor data
 - Location
 - Temperature
 - Humidity
 - e.g. of cherries
- Developed by Sensefinity
 - Early prototypes adapted from shipping containers
 - State-of-the-art hardware solutions being field-tested

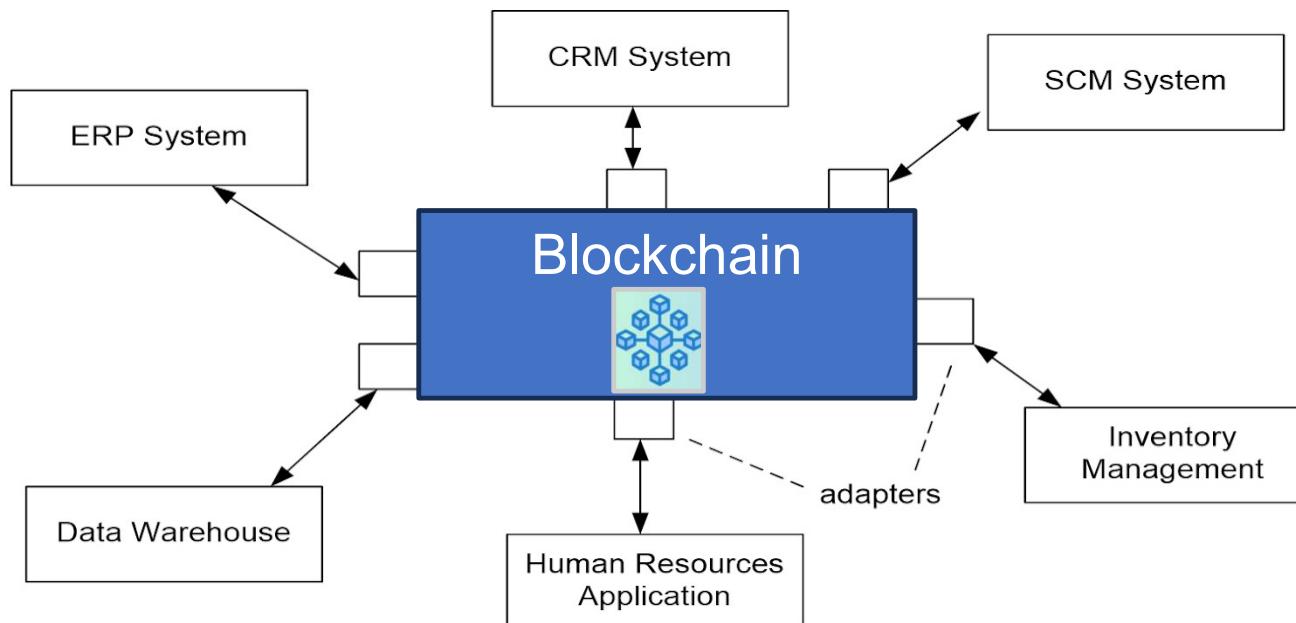


Open alternative: AMBOX

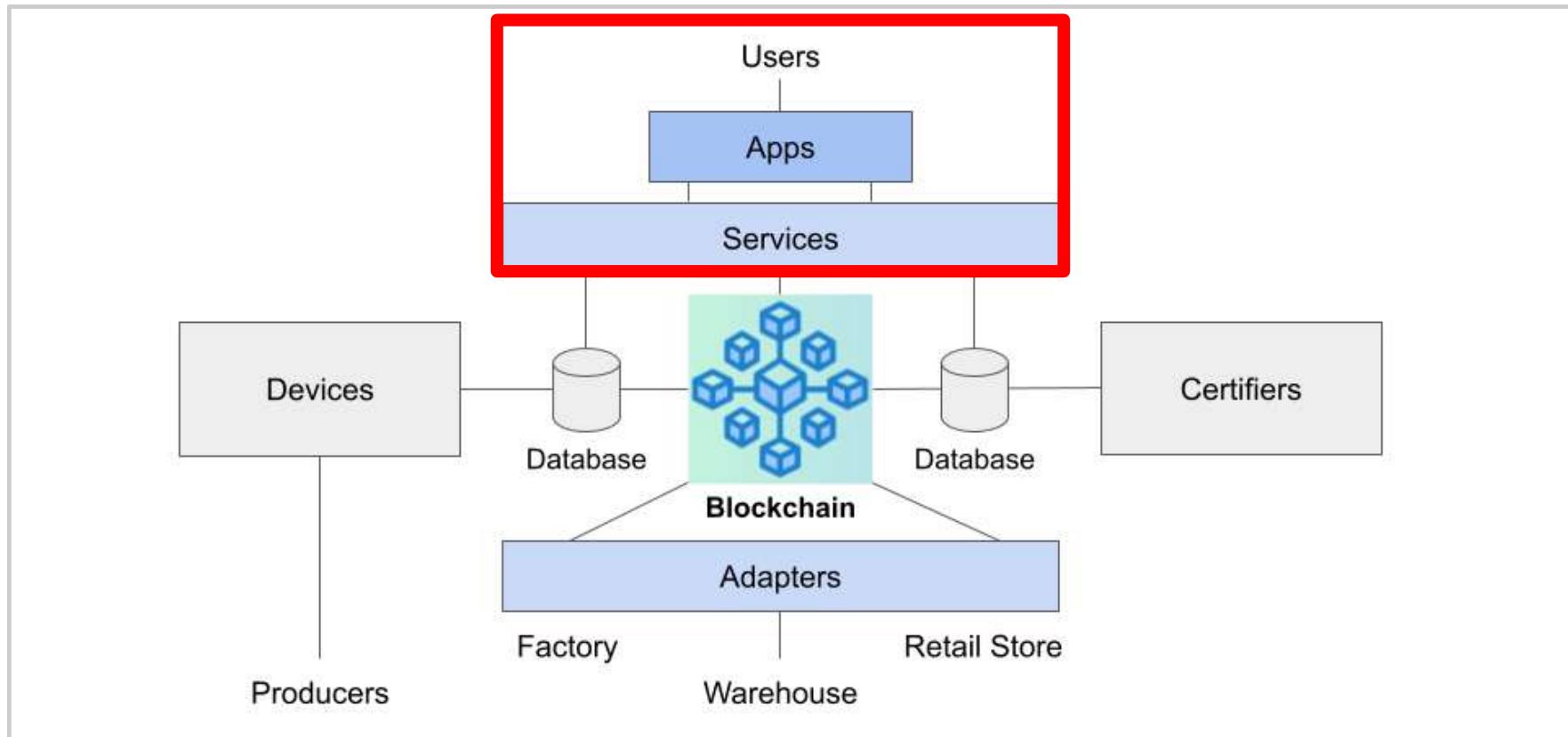
- AMBient Monitoring bOX
- Also integrates IoT and Blockchain for traceability
- Low-cost IoT hardware
for ambient monitoring
- Open for academic experimentation
 - uses off-the-shelf hardware
- Designed to write to a blockchain peer
 - As soon as possible
 - With as few intermediaries as possible



Blockchain as Integration Hub



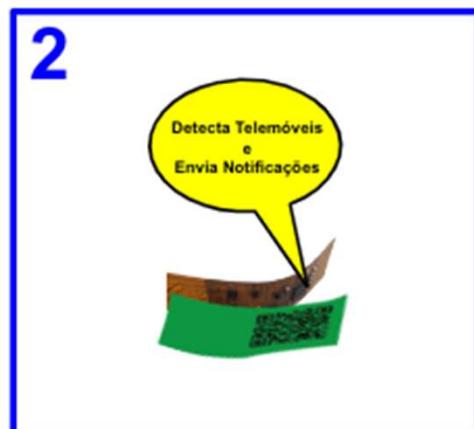
FoodSteps: Apps and Services



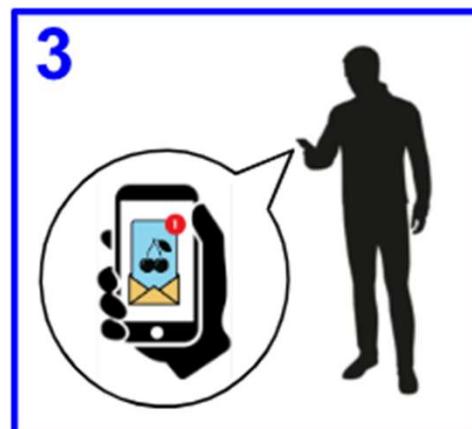
User Experience Design



Package with QR code



BLE beacon announcement



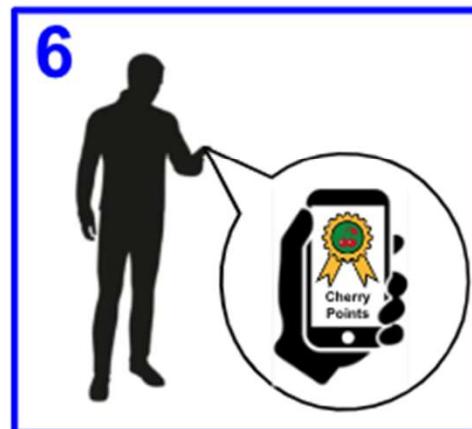
Mobile notification



QR code to access data

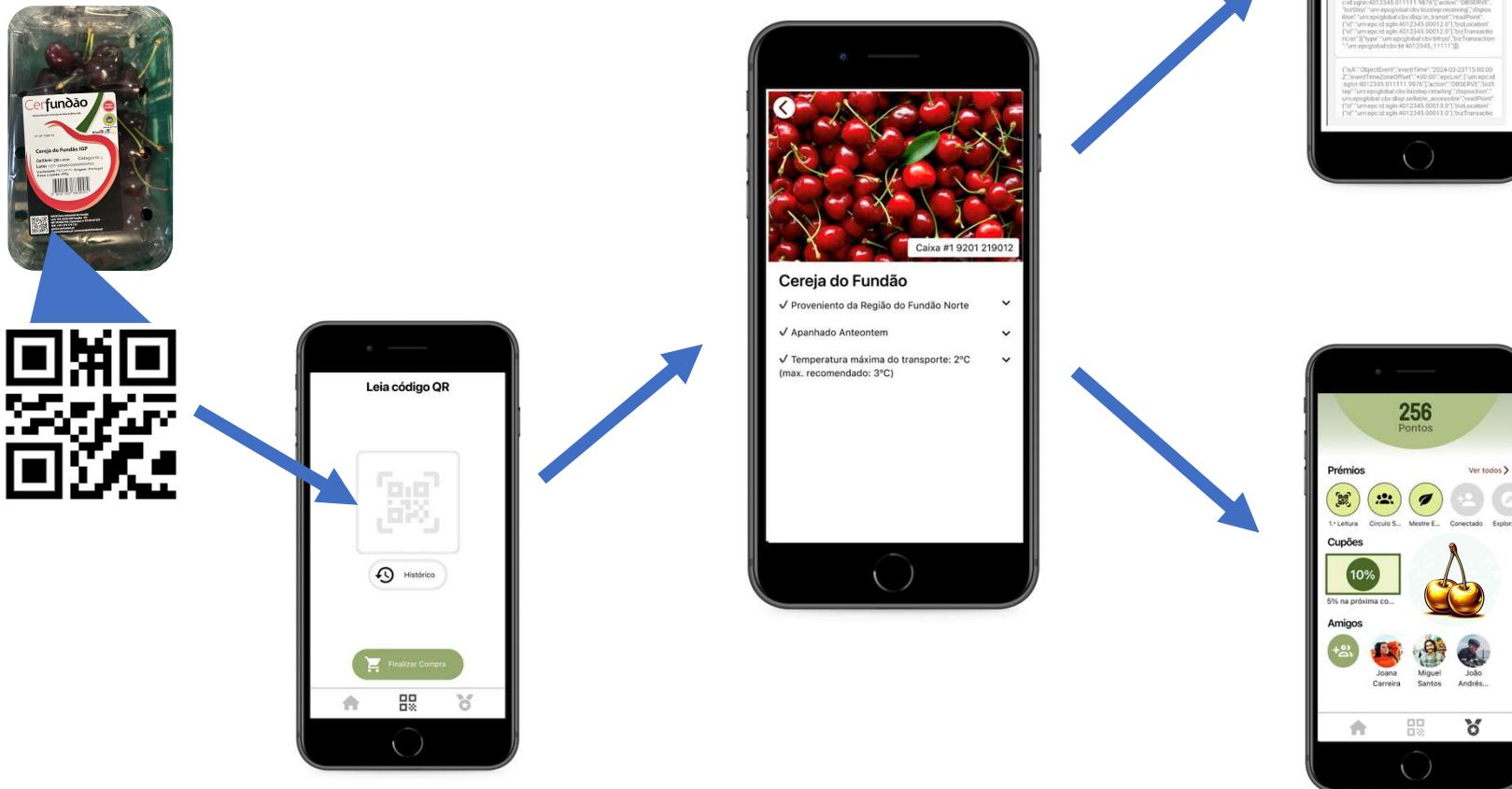


Information available in App



Consumer gets “gamified” rewards

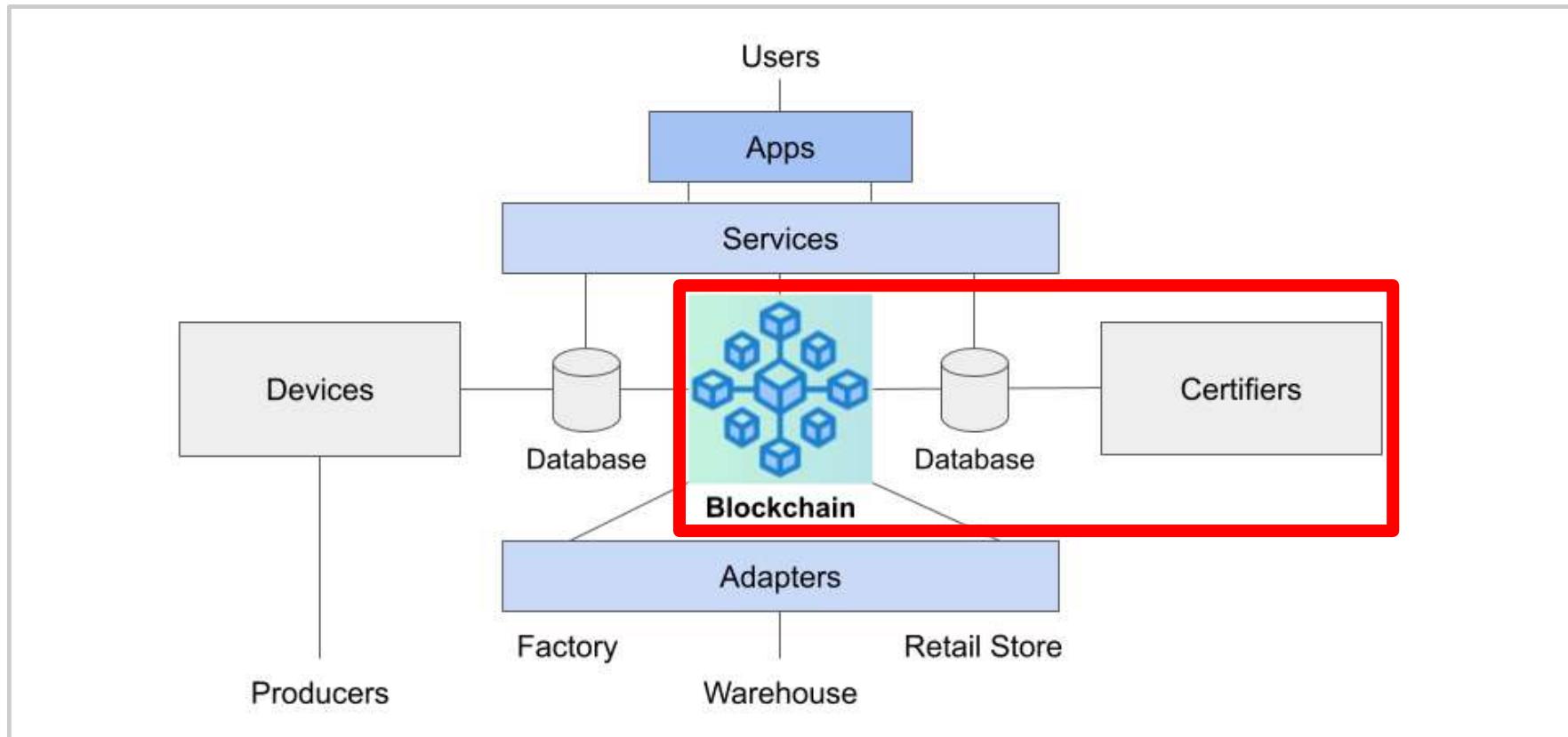
FoodSteps App navigation



Application Programming Interface

- Each service provides a REST API
 - Representational State Transfer
 - service that enables communication between systems using HTTP methods (GET, POST, PUT, DELETE) and JSON/XML data
- Goal: ensure seamless integration and interoperability between different applications
 - Facilitates access across multiple platforms
 - Can be reused by many mobile and web applications

FoodSteps: Blockchain and Certifiers

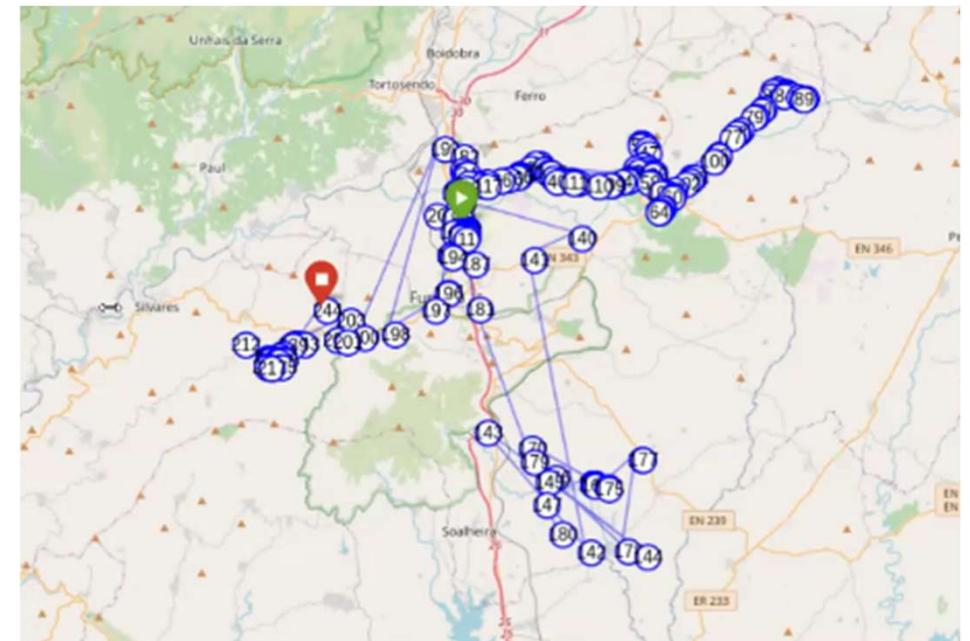


Why put data in the blockchain?

- Even if not restructured
- Working as a “notarization” mechanism
- Create a record
 - Replicated
 - Timestamped
 - Verified by set of endorsers

Decentralized Verification with ChainGuard

- Ensure data integrity in supply chains by using chaincode
- Leverage multiple endorsers
- Captures data from sensors
 - organizes them into “Points” and “Routes”
 - applies verification rules
 - Average Speed Rule
 - Geofencing Rule
 - Outlier Rule



5.

Research in Progress



Ongoing work: ChainGuards

- Verification of more types of sensed data
 - e.g. temperature, humidity, motion sensors
 - Rules still executed in chaincode – for auditability
- How to trust sensor inputs?
 - Single source of data
 - Hand-off between sources of data
 - Multiple source of data
 - Redundant
 - Diverse source of data
 - Different owners/operators

Ongoing work: Gypsum

- Meaning: mineral used in construction as *plaster* and *drywall*
- Analogy:
 - Can be changed while fresh
 - Cannot be changed after drying
- Goal: allow controlled-mutability blockchain transactions
 - Fix mistakes – OK
 - Frauds – KO

Ongoing work: FoodFights

- Adversarial cybersecurity evaluation of a mobile application and its support systems
- Identify attack surfaces
 - Apply tools
 - Build exploit catalog
 - Design a competitive game around the found exploits
 - Create hackathon-like environment for further exploitation
 - “Attackathon”

Upcoming work: ChainFights

- Adversarial cybersecurity evaluation of blockchain infrastructure
 - Challenge decentralization assumptions
 - Challenge security assumptions
- Continue to develop “Attackathon” methodology

Upcoming work: AmBox Twin

- Extend Ambient Box to provide Digital Twin of devices
- Goals:
 - Sensor data collection
 - More data visibility
 - Reliable device fleet management

Upcoming work: FastFood (Steps)

- Food Traceability Blockchain Performance Assessment
 - and Optimization
- Joint work with TU Munich (Network Architecture and Services Chair)
- Use case of the METHODA methodology and framework to configure and automate the deployment of FoodSteps blockchain
 - Use instrumentation and measurement capabilities to measure performance
 - Response time assessment
 - Dimension the system for the pilot deployment



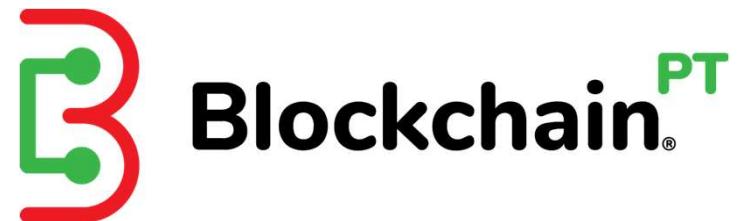
6.

Conclusion

Conclusion

- Food transparency empowers informed choices
- Challenges in food supply chains
 - Balancing cost, safety, and sustainability
 - Data sharing, security, and integration remain key obstacles
- Role of traceability
 - Enables tracking, authentication, and compliance
 - EPCIS structures event-based data
- Blockchain as a novel solution
 - Integration hub, but supported in a decentralized way
 - Hyperledger Fabric as enterprise blockchain platform
- We presented FoodSteps
 - Blockchain-based Food Traceability System

Obrigado!



Special Thanks to all
the FoodSteps team



Miguel.Pardal@tecnico.ulisboa.pt

This work was financially supported by Project Blockchain.PT – Decentralize Portugal with Blockchain Agenda, (Project no 51), WP 1: Agriculture and Agri-Food, Call no 02/C05-i01.01/2022, funded by the Portuguese Recovery and Resilience Program (PRR), The Portuguese Republic and The European Union (EU) under the framework of Next Generation EU Program.



PRR
Plano de Recuperação
e Resiliência



**REPÚBLICA
PORTUGUESA**



**Financiado pela
União Europeia**
NextGenerationEU

Highly Dependable Systems

Lecture 10

Trusted Hardware

Lecturers: Miguel Matos and Paolo Romano

Trusted Hardware

- So far we covered approaches that make no assumptions about the trustworthiness of the underlying hardware
 - We only trust the algorithms
 - And the cryptographic primitives

Trusted Hardware

- In this lecture
 - What can we additionally achieve if we have hardware support for building secure systems?

Trusted computing base (TCB)

- Set of hardware+software components on whose correctness a given security policy relies
- Larger TCB → wider attack surface
 - E.g., OS and hypervisor are prone to bugs, malware
 - Xen 150K LoC, 40+ vulnerabilities per year
 - Linux, 17M LoC, 100+ vulnerabilities per year

Agenda

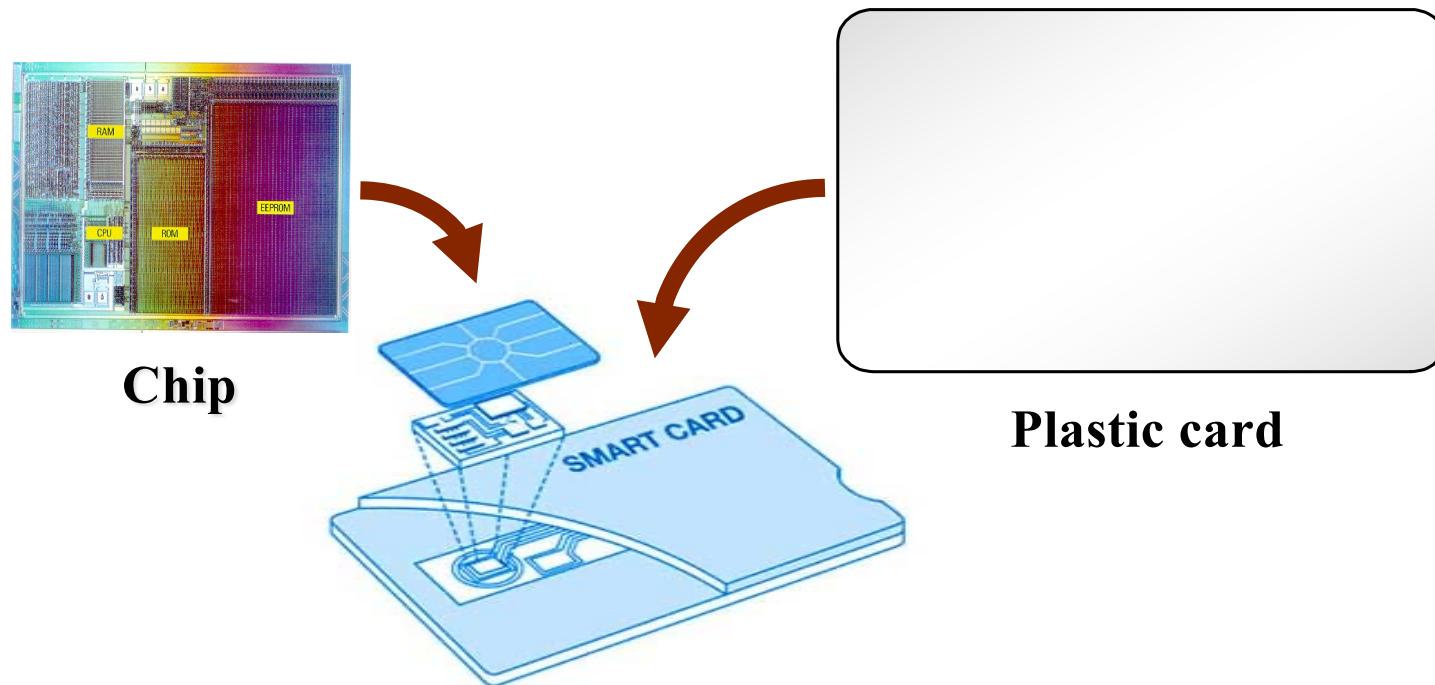
- Smart Cards
- Trusted Platform Module
- Trusted Execution Environment
- Trusted Domain Extensions

Smart Cards

Smartcard principles

- A secure way of storing a small amount of sensitive data, and perform some computations
 - Notably cryptographic operations without exposing private keys
- Small size, cheap, low power, small but sufficient amount of memory + CPU horsepower
- Successful strategy of separating:
 - expensive consumer electronic device (bulk production)
 - cheap authentication hardware (personalized)

Smart Card



Smart Card applications

- Retail
 - Sale of goods using Electronic Purses, Credit / Debit
 - Vending machines
 - Loyalty programs
 - Tags & smart labels
- Government
 - identification
 - Passport
- Et cetera
- Healthcare
 - Insurance data
 - Personal data
 - Personal file
- Communication
 - GSM
 - Payphones
 - Transportation
 - Car Protection
- Pay TV operators

Well-known Smart Card Application

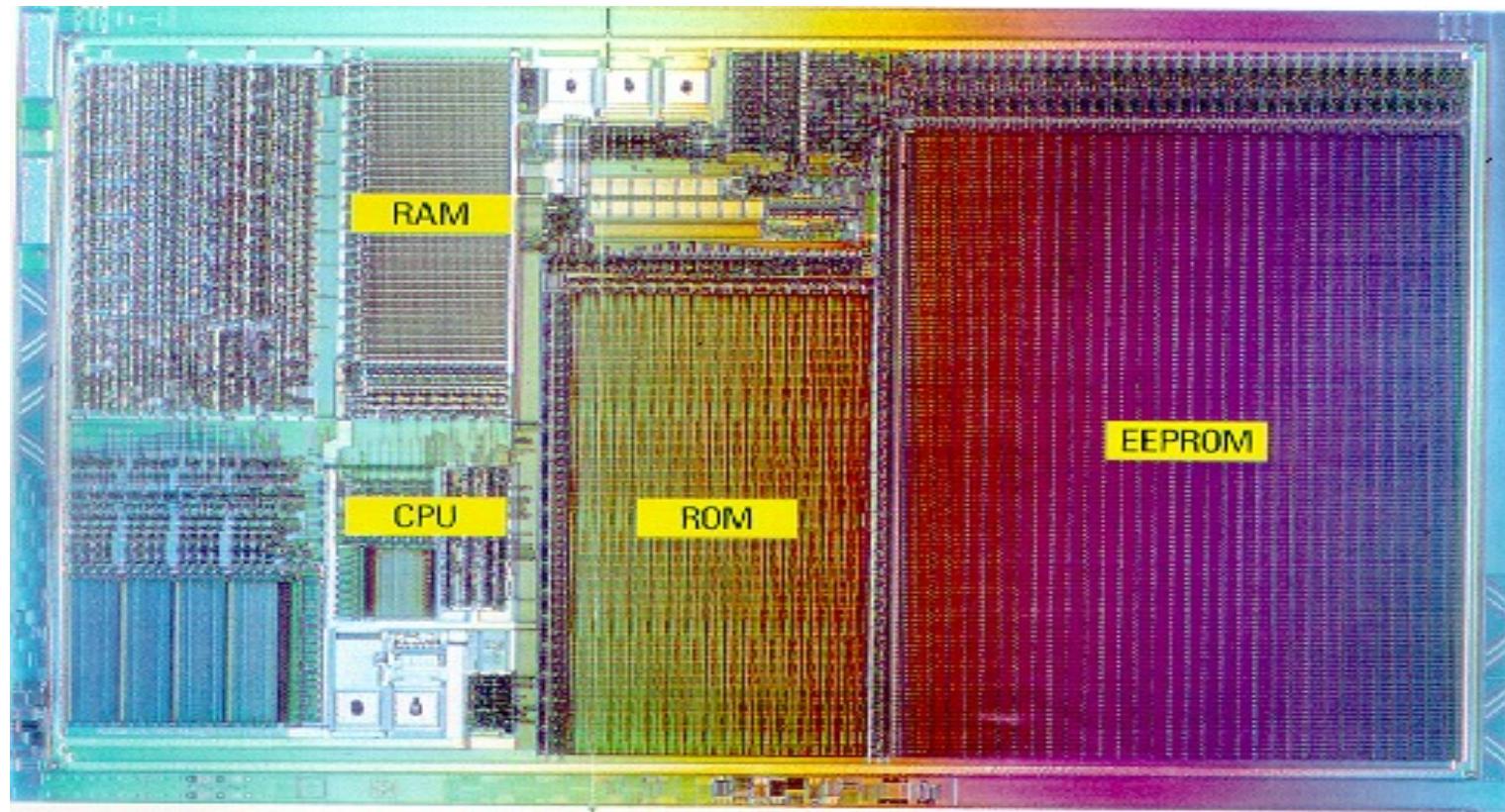
- Store sensitive data (private keys)
- Store these secrets with strong integrity and confidentiality
- Perform computation (e.g., signing) without exposing keys



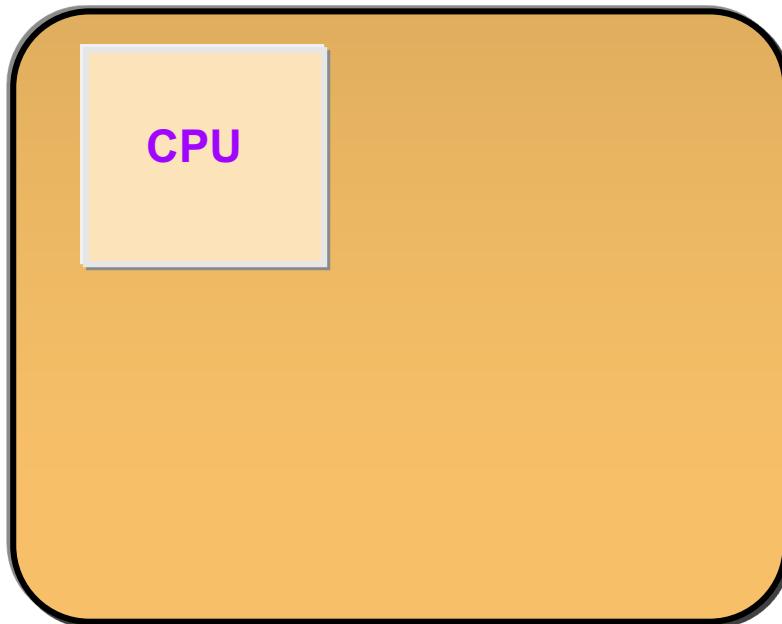
Smart Card characteristics

- Simple processors
 - 8-32 bits @ a few tens of MHz (up to 32 MHz)
- Small memory
 - Up to 128kB of ROM
 - Up to 128 kbit of EEPROM (electrically erasable programmable ROM)
 - Up to 4kB of RAM
- Crypto co-processors (incl. randomness generator)
- Low manufacturing price (< 10€)
- No internal power source
 - Source of potential attacks
- Low power consumption (tens of mW)

Inside the smart card



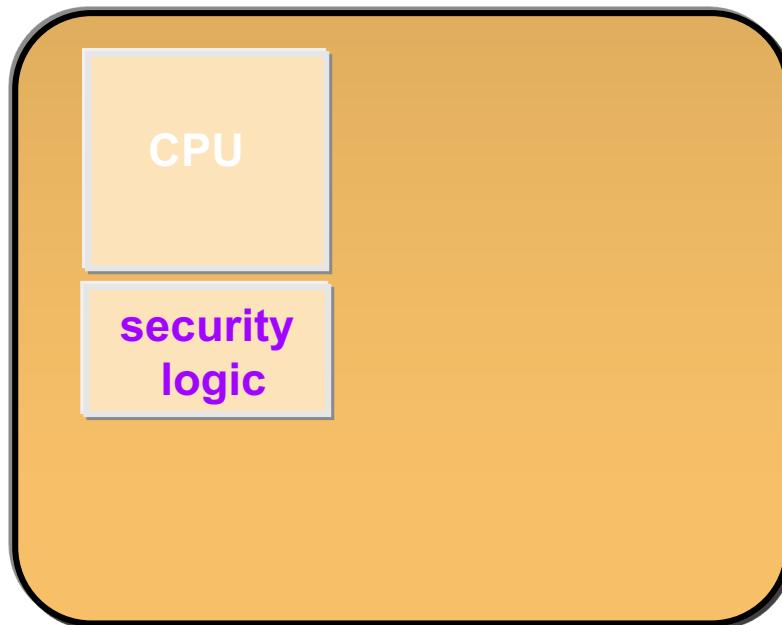
Inside the smart card



Central
Processing
Unit:

- heart of the chip

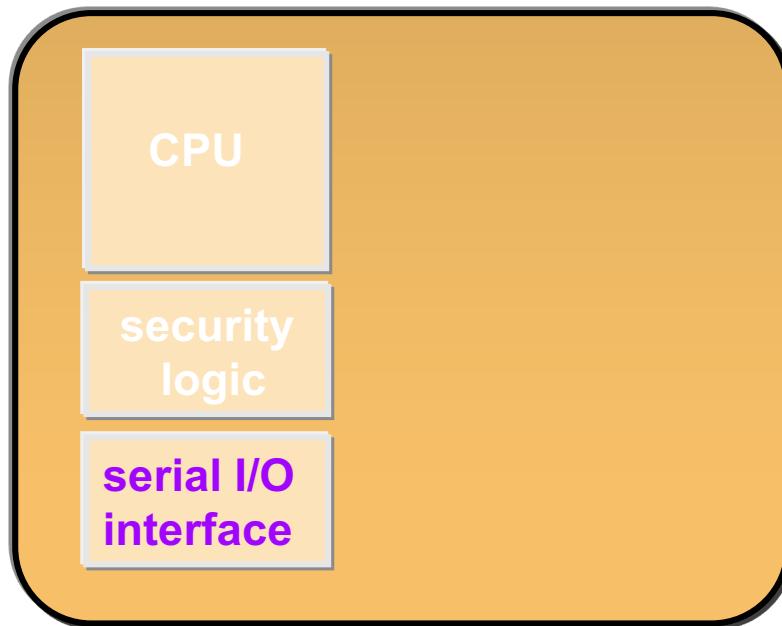
Inside the smart card



security logic:

- detecting abnormal conditions,
- e.g. low voltage

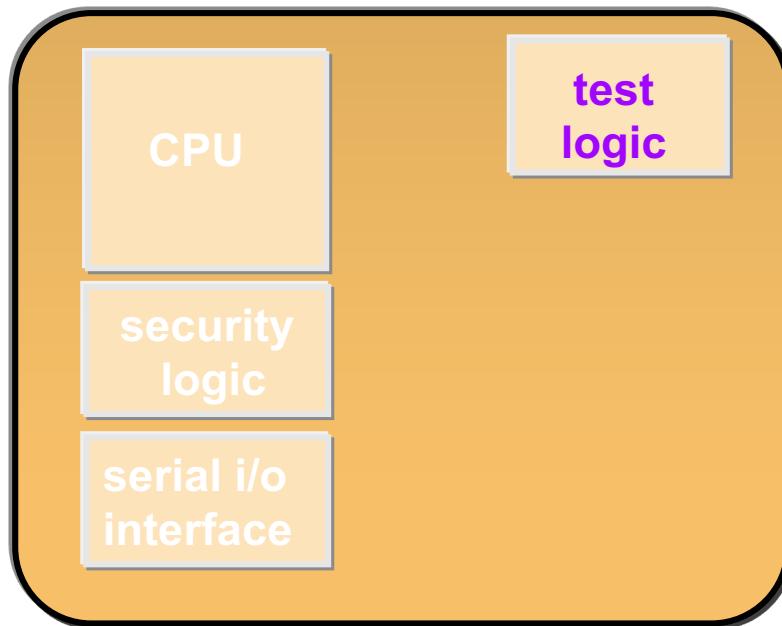
Inside the smart card



serial I/O interface:

- contact to the outside world

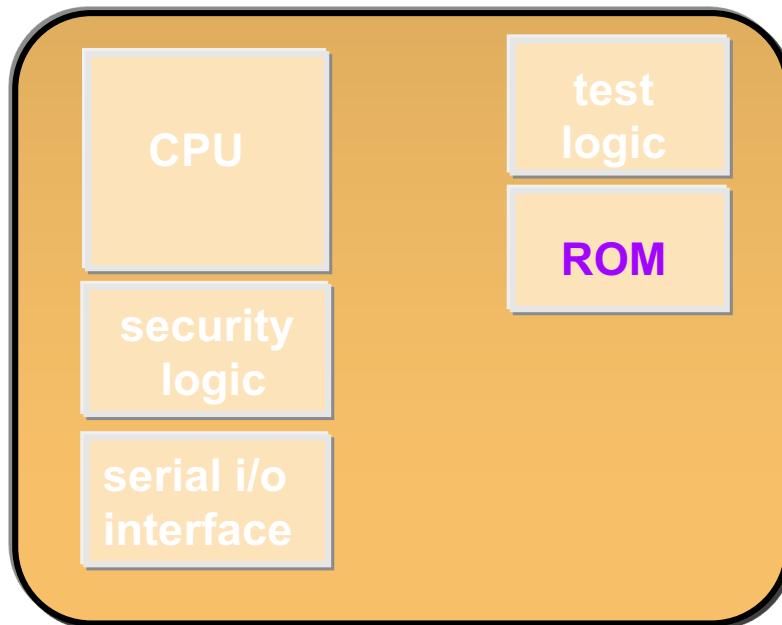
Inside the smart card



test logic:

- self-test procedures

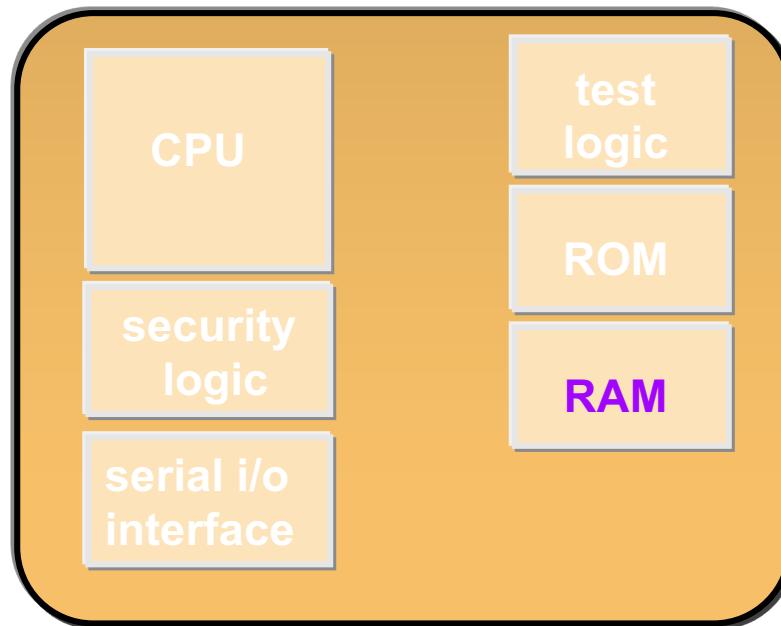
Inside the smart card



ROM:

- card operating system
- self-test procedures

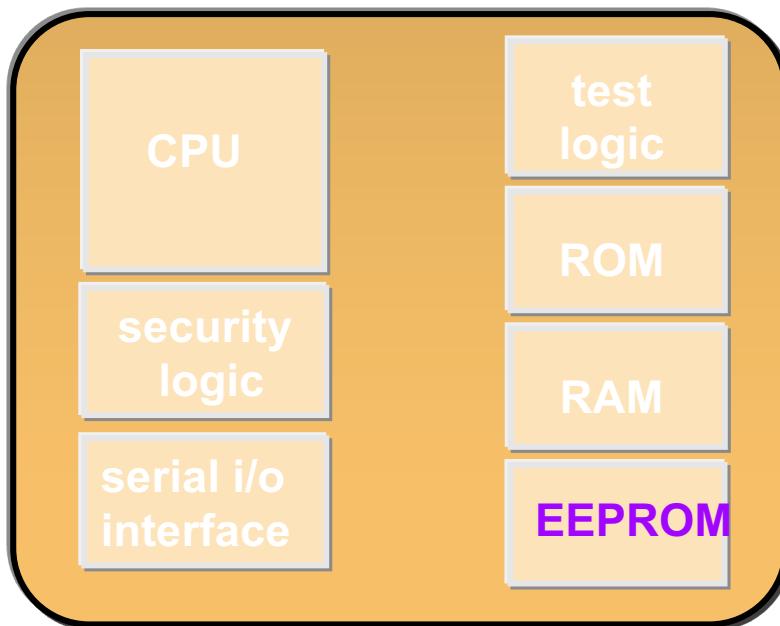
Inside the smart card



RAM:

- ‘scratch pad’ of the processor

Inside the smart card

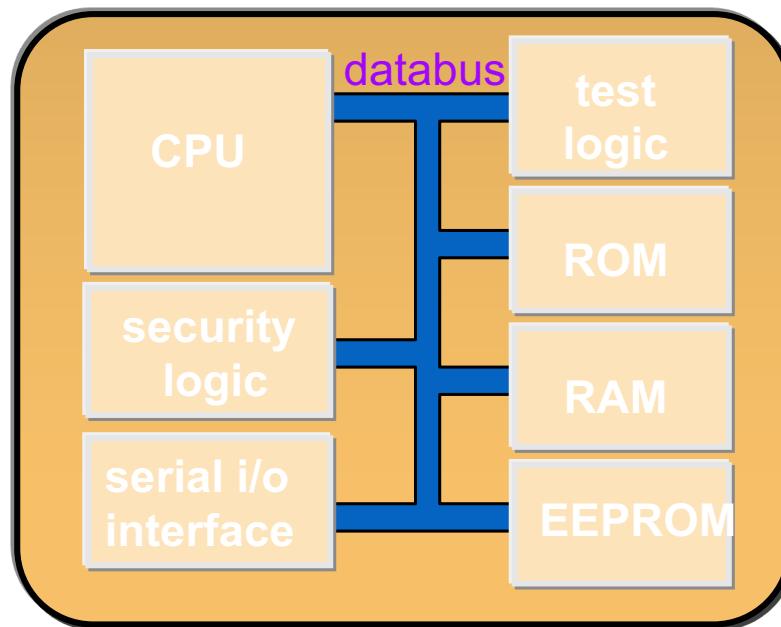


EEPROM:

- cryptographic keys
- PIN code
- biometric template
- balance
- application code

Recall: private key
never leaves card!

Inside the smart card

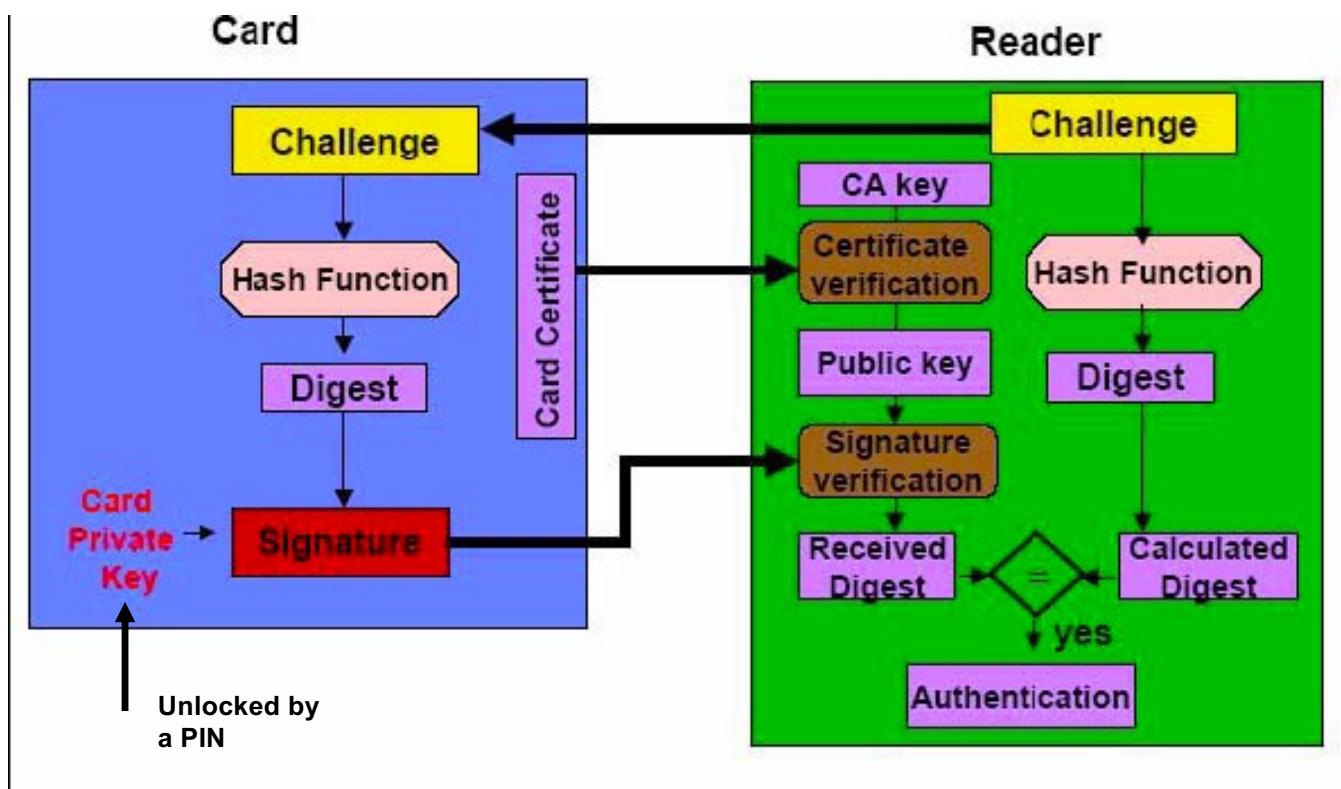


databus:

- connection between elements of the chip
- 8, 16, or 32 bits wide

Smartcard security

Authentication with Smartcards



Authentication with Smartcards

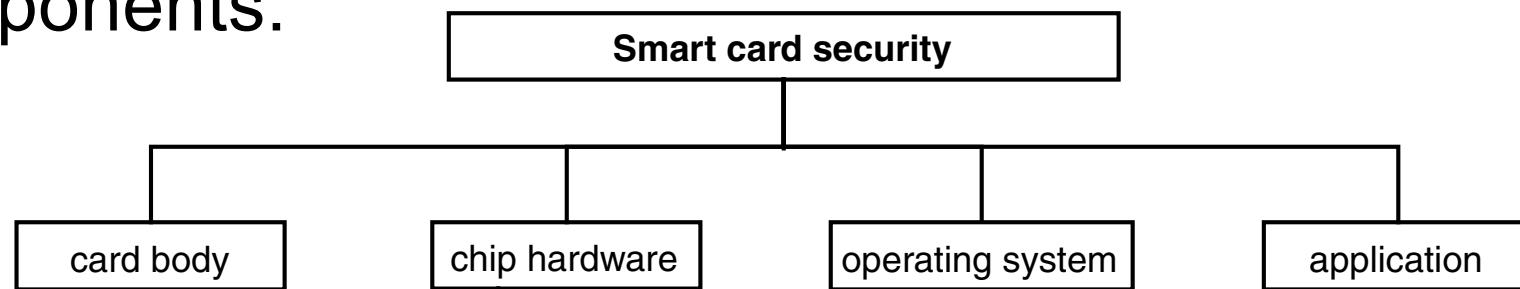
- Multi-factor authentication - combination of:
 - What you know
 - e.g.: passwords, PINs
 - What you have
 - e.g.: OTP tokens, smartcards
 - What you are (biometrics)
 - e.g.: fingerprints, iris scans, face recognition
- Typically two-factor authentication is used
 - e.g.:
 - PIN + Card (e.g. ATMs)
 - Password + One-time-password (OTP) token
 - Fingerprint / biometrics + Smartcard

Smartcard security

- Security properties of smartcards are very attractive:
 - Enables authentication and digital signatures without exposing private keys
 - Relies on a small Trusted Computing Base (TCB)
 - Using smartcards for authentication functions of a more expensive device enables:
 - bulk production of the device
 - easy replacement of the authentication if needed
 - Low price for manufacturing smartcards (a few euros)
- However, not full proof
- Main challenge: low price makes it easy for attackers to acquire many devices and try to tamper with them

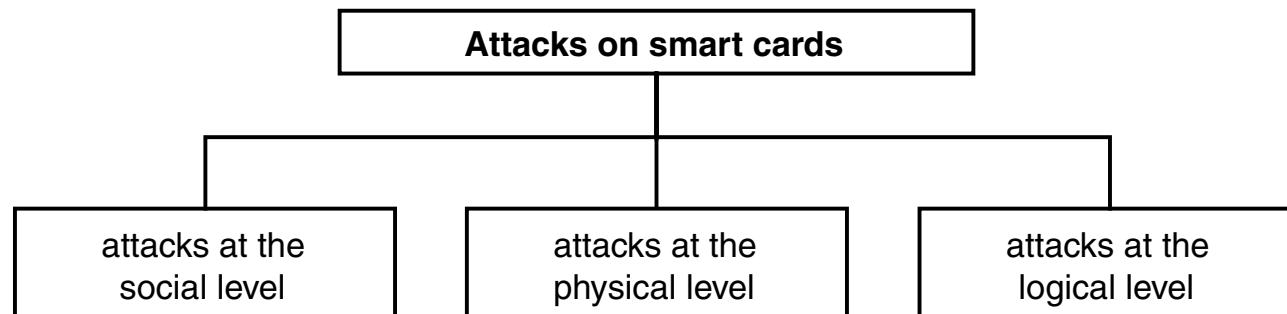
Smartcard security

- Smartcard security is ensured by 4 main components:



Attacks on smartcards

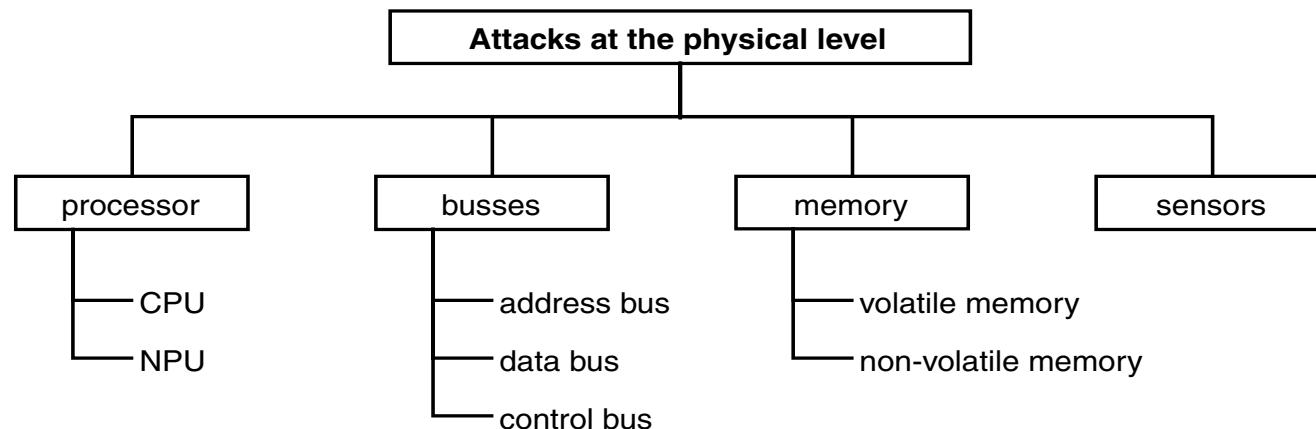
- 3 main types of attacks



Physical attack types

- As with the cryptanalysis of cryptographic protocols, physical attacks can be divided into:
 - Active attacks:
 - E.g., the attacker manipulates the data transmission process or the microcontroller.
 - Passive attacks:
 - E.g., attacker may make measurements on the semiconductor device.

Points of physical attack on a smart card



Smartcard security history

- Until mid-90s mostly "security through obscurity", combined with small size of devices
- Satellite TV was a game changer:
 - Lots of people trying to get access to pay TV channels
 - Anecdotally, when Star Trek was made available only within the UK, lots of young people elsewhere worked hard to break smart cards

Example smartcard attacks

- Initial pay TV systems would cancel subscription by sending message to card
 - Defeat for the strategy: interposed device would drop those messages

Example smartcard attacks: Prevent EEPROM from being overwritten

- Early smartcards used separate connector for special voltage (V_{pp}) to write to EEPROM
- EEPROM stores persistent and mutable state
 - E.g., value counters for money stored in wallet or number of failed PIN attempts
- Attack: freeze EEPROM contents by blocking V_{pp} connector
- Fix: generate V_{pp} from supply voltage (V_{cc})
- Defeat for the strategy: attacker can break the voltage multiplier circuit
- Can be exploited to try unlimited PINs

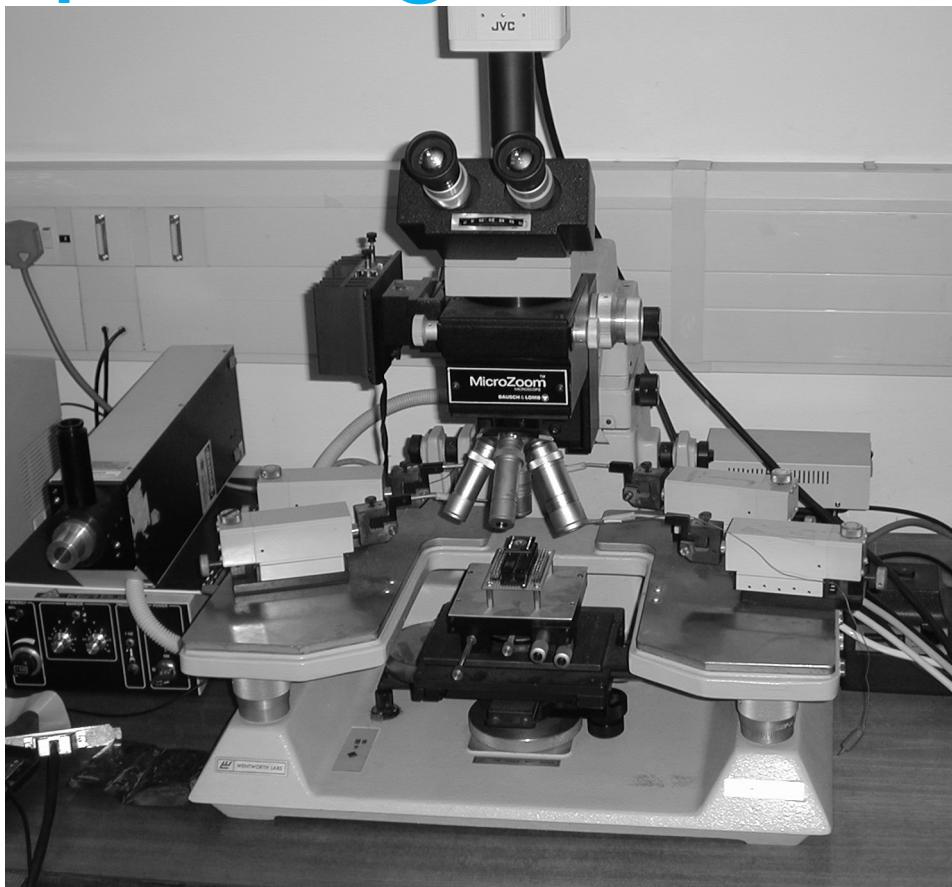
Example smartcard attacks: Reading memory with microscopes

- Use an electron microscope to read RAM contents
- However, "cheap" microscopes require low frequency clock → supply a slow clock through the clock input
- Fix: circuit detects low clock frequency and resets card
- Problem: trade-off between false positives and false negatives
 - low cost terminals suffer from ample frequency oscillations
 - security mechanism often disabled to maximize interoperability

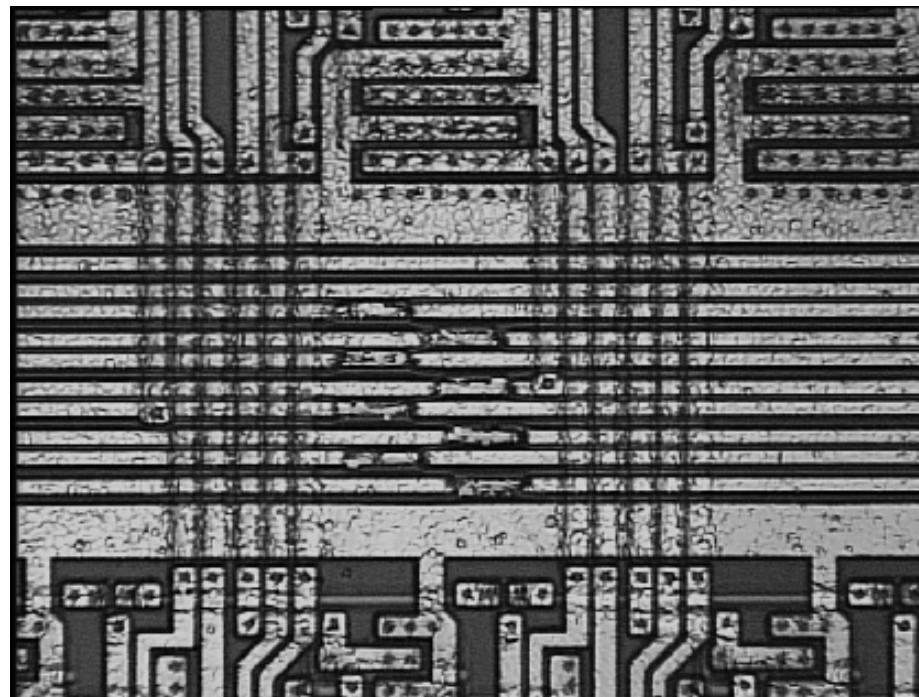
Example physical attacks

- Probing stations: microscopes + micromanipulators
 - Allow for inserting fine probes on chip surface
- Can be used to snoop the smartcard bus
- Trace from bus during encryption operation is sufficient to derive the key
- Some smartcards compute a checksum on memory immediately after reset:
 - give the attacker a complete listing of the card memory contents.

Low cost probing station



Data bus prepared for probing



Excavation of the passivation layer (insulation) with laser shots

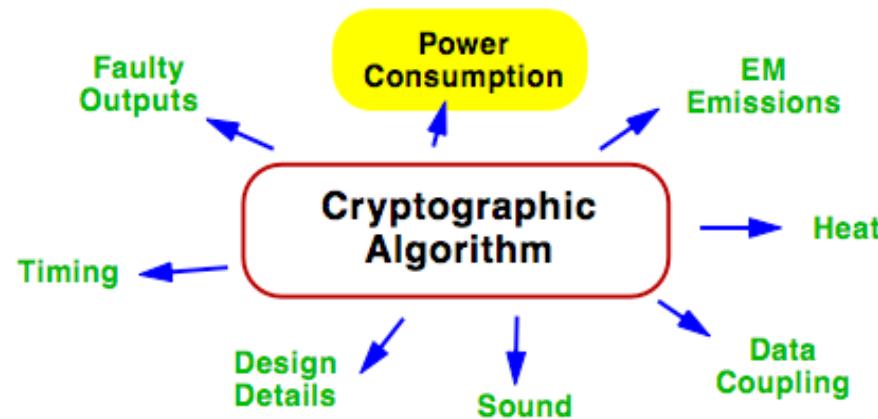
Side Channel Analysis

Two inroads for Attacks

- Traditional Mathematical Attacks
 - Algorithm modeled as ideal mathematical object
 - Attacks mostly theoretical rather than operational
- Implementation Attacks
 - Actual code/hardware is much more complex
 - Example: Side Channel Attacks

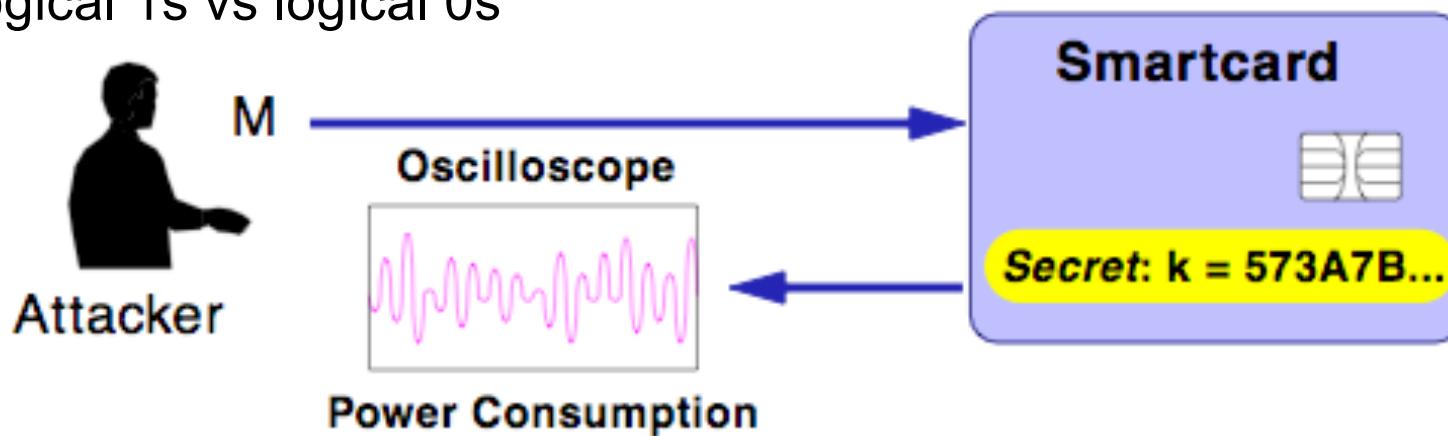
Side Channel Analysis

- Explores the information leaked from the chip
- Not exclusive to smartcards, but important in that context



Power Analysis Attacks

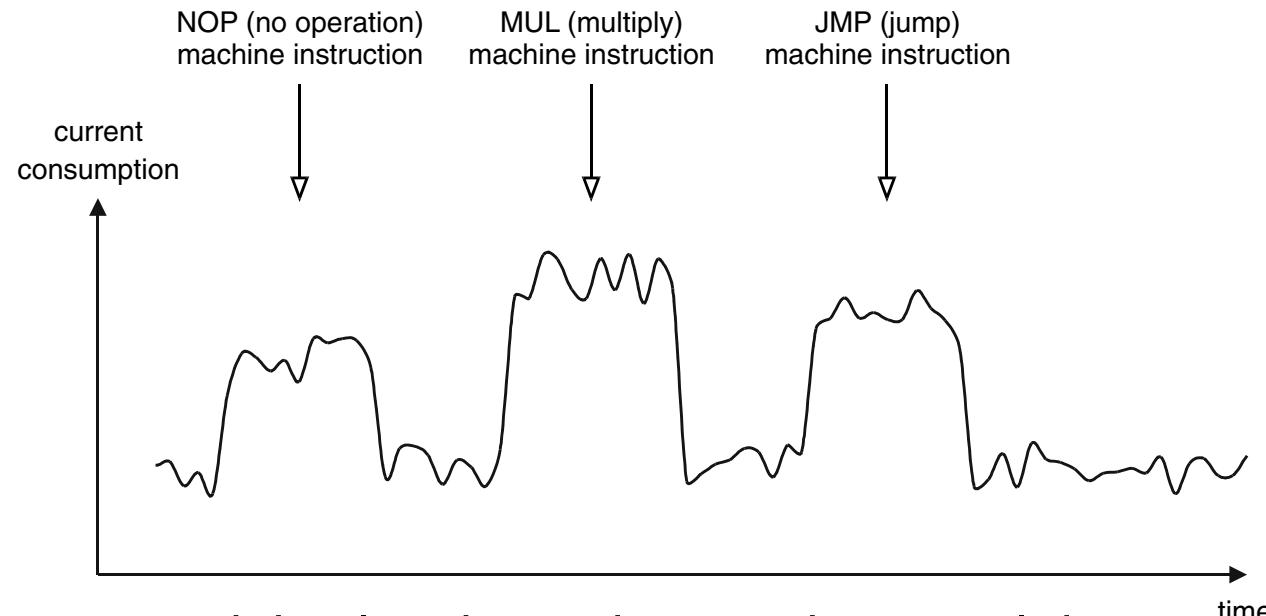
- Measure instantaneous power consumption of a device while it runs a cryptographic algorithm
- Different power consumption depending on code path, and even when operating on logical 1s vs logical 0s



- Non-invasive, passive attacks
 - Can be performed by modified terminals on unsuspecting customers
 - Compromise many cards at the cost of building a single Trojan terminal
 - But in practice more effective when attacker has the time to do several attempts

Simple Power Analysis

- Attacker knows encryption code and extract a power trace
- Power consumption is dependent on the machine instruction being processed



- But in many cases, it is also dependent on the actual data...

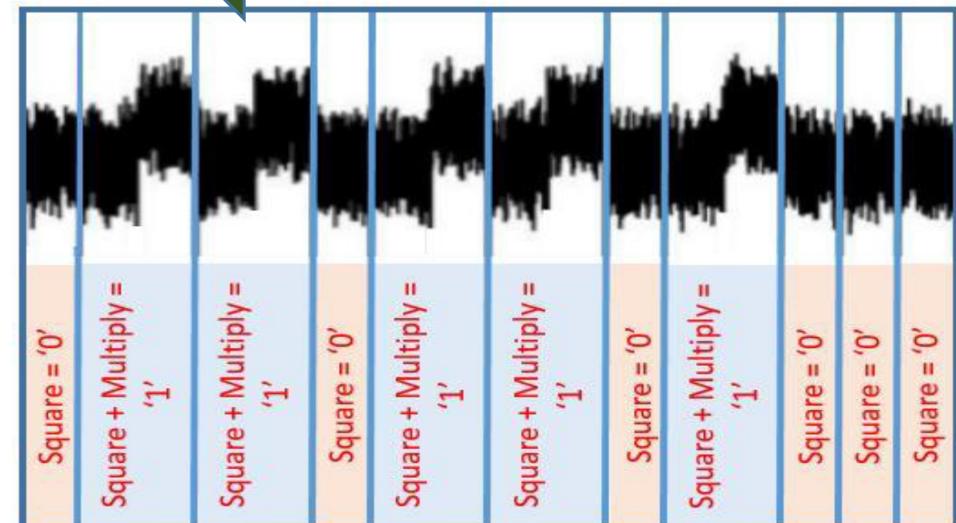
Simple power analysis – RSA

```
function modular_pow(base, exponent, modulus) is
// . .
result := 1
base := base mod modulus
while exponent > 0 do
    if (exponent mod 2 == 1) then
        result := (result * base) mod modulus
exponent := exponent >> 1
base := (base * base) mod modulus
return result
```

Fundamental operation in RSA:

- RSA decryption: $M = C^d \text{ mod } N$
- Ciphertext C is raised to the private exponent d modulo N

Multiplication
only done
when bit==1



Side-channel countermeasures

- Energy consumption must be independent of instructions and data!
- Hardware:
 - Artificial noise injection – add noise to power supplies, clock signals, or data paths
- Software:
 - Modify the code: research area of “oblivious data structures”
 - adversary cannot infer data from memory accesses

Threat model

- Who is the adversary in smartcard usage?
 1. (Traditional view) In a banking transaction, need to consider attackers that may listen or tamper with messages between card, terminal, and bank
 2. But the owner of the card is also untrusted:
 - May try to extract information from the card
 - Important that keys do not leave card to owner's computer (or to the vendor's terminal)

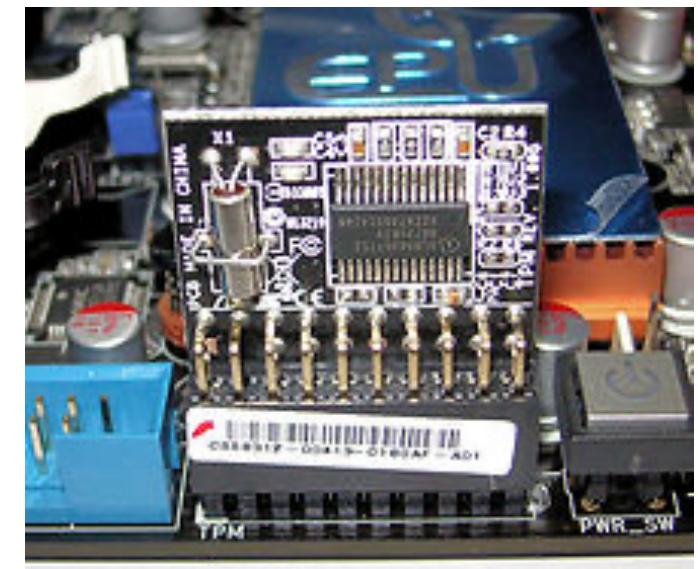
Defending against privileged software attacks

- When privileged software is compromised, then “all bets are off”
 - Can inspect+modify code
 - Can inspect+modify data
 - Thus having access to private keys and forging signatures
 - Also having access to sensitive data
 - And tampering with the state of the computations

Trusted Platform Module

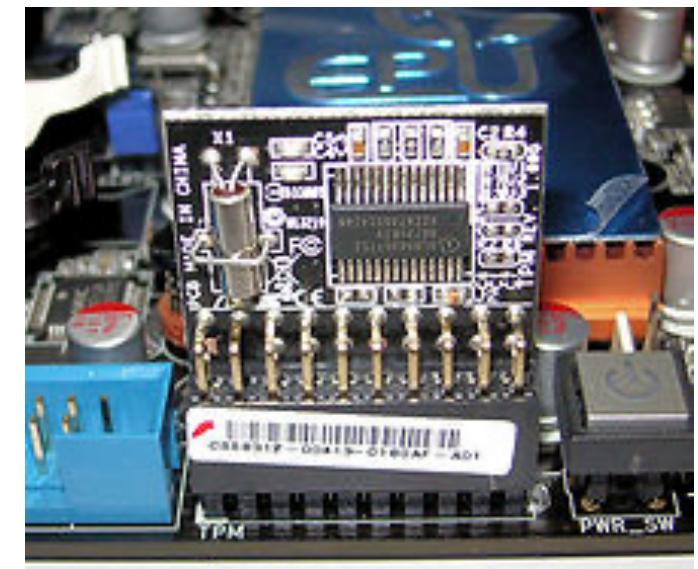
Trusted Platform Module (TPM)

- Standard for **secure cryptoprocessor**, running alongside main processor
- Widely available on most personal computers



Trusted Platform Module (TPM)

- Goal
 - Ensure/verify that machine is in a “trusted state”
- Based on the idea of **secure boot**
- Additionally used for authentication
 - stores a unique RSA key, burned into the chip

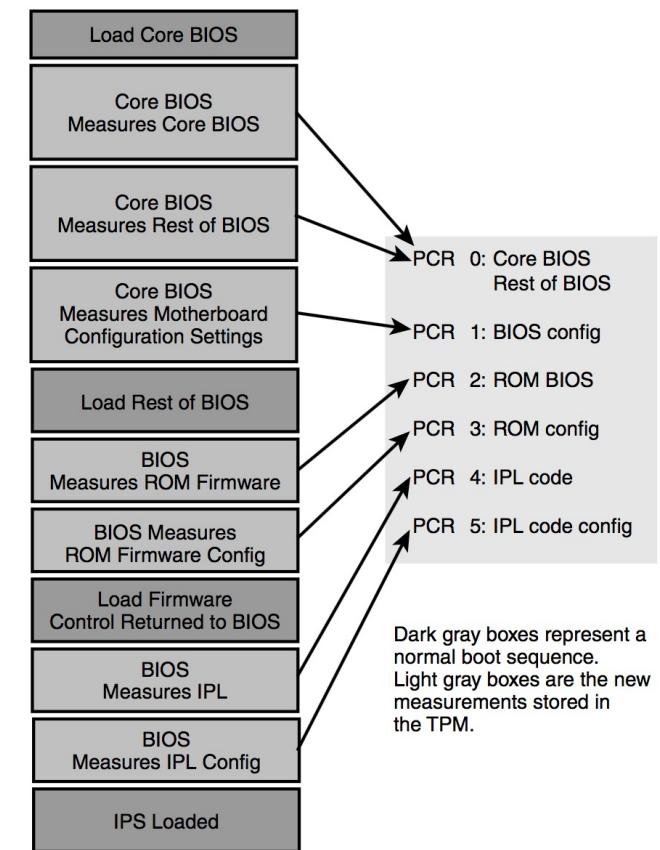


Secure boot – PCR registers

- Array of special registers called PCRs
- Records digest of kernel that booted + entire boot sequence
- Always extended (never overwritten)
 - $\text{PCR}_{\text{new}} = \text{SHA-1}(\text{PCR}_{\text{old}} \parallel \text{new value})$
- **Remote attestation** consists of requesting PCR value
 - TPM signs response (alongside nonce to prevent replays)

Secure boot steps (simplified)

- Hardware computes hash of BIOS and stores in PCR, passes control to BIOS
- BIOS measures master boot record (MBR) and extends PCR, passes control to MBR
- MBR measures OS, passes control
- If OS designers want, can recursively measure applications
- What is the TCB in this case?



TPM criticism

- Highly polarized debate – criticized for owner losing control over own machine
- Claims that it was meant for digital rights management (DRM)
 - Selling music and movies while preventing piracy
 - Making sure that software is licensed

TPM criticism

(more details at <https://www.cl.cam.ac.uk/~rja14/tcpa-faq.html>)

- Court can order defamatory paper to be censored
 - Software company that wrote word processor has to comply
 - Or, more broadly, censorship by governments or corporations can be made easier
- Promotes vendor lock-in (e.g., Word files might be encrypted with keys known only to Microsoft)
- Locking competitors out of application file formats is bad for innovation
- Even though you can turn TPM off, strong incentive not to due to apps not working well
 - Like moving from Windows/Macs to Linux: more freedom but less choice
- Concentrates power in the owners of the trusted computing infrastructure
 - Single corporation with a OS monopoly may decide to shut down an entire country

Arguments and counterarguments

- Against TPMs: TPMs constrain what users do with their own PCs, so that software and content vendors can make more money. TPMs promotes censorship
- Pro TPMs: there are some situations where constraining user control is useful, e.g., modifying odometer before selling car

Actual TPM applications

- Protects laptop contents in case of theft
- Full disk encryption
- Decryption keys are only provided upon correct boot sequence
- ChromeOS also uses TPMs, namely to prevent firmware rollback

The screenshot shows a Microsoft Learn page for the article "BitLocker". The top navigation bar includes links for Microsoft 365, Solutions and architecture, Apps and services, Training, Resources, Documentation (which is underlined), Learn, Training, Certifications, Q&A, Code Samples, Assessments, Shows, and Events. Below the navigation is a search bar labeled "Filter by title". The main content area has a breadcrumb trail: Learn / Windows / Security / BitLocker. The article title is "BitLocker" with a "Feedback" link. It was published on 02/17/2023, takes 6 minutes to read, and has 23 contributors. The "Applies to:" section lists Windows 10, Windows 11, and Windows Server 2016 and above. A summary states: "This article provides a high-level overview of BitLocker, including a list of system requirements, practical applications, and deprecated features." The "BitLocker overview" section describes BitLocker Drive Encryption as a data protection feature that integrates with the operating system and addresses threats of data theft or exposure from lost, stolen, or inappropriate decommissioned computers. It notes that BitLocker provides maximum protection with TPM version 1.2 or later. The page footer includes links for "Download PDF" and other Microsoft resources.

Trusted Execution Environment

Trusted execution environments

- TPMs turned out to be fairly inflexible
 - Secure boot attests to a very large software stack

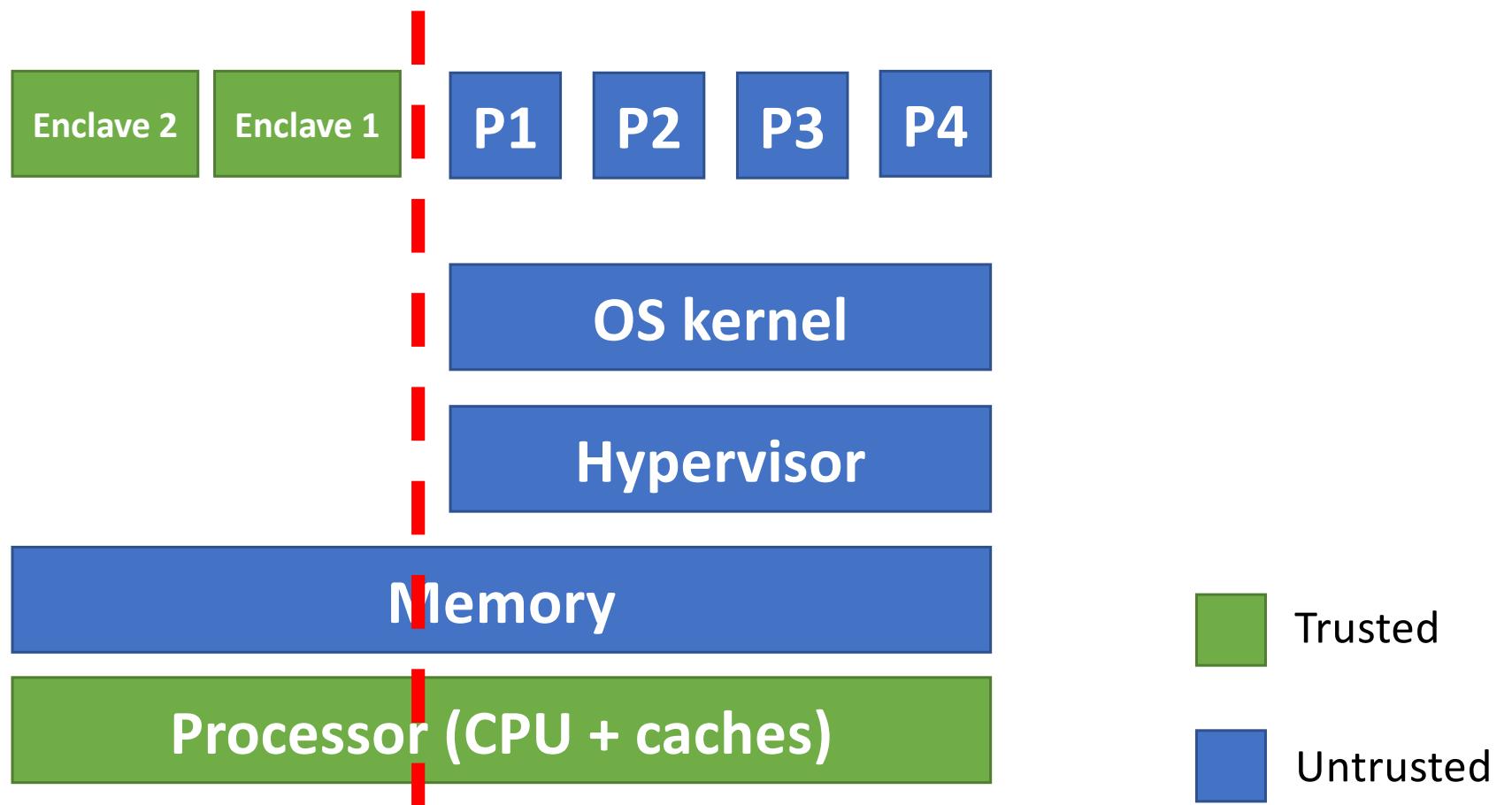
Trusted Execution Environments

- Allow for strong confidentiality and integrity of a generic piece of code, deployed in a commodity machine
 - Focus on confidentiality and integrity, not availability
- Same strong threat model
 - Do not trust the owner / user / operator of the machine
 - Handles a compromised OS kernel (e.g., with malware)
- Supported by several processors
 - Intel SGX → code runs in an **SGX enclave**
 - ARM TrustZone
 - AMD SEV

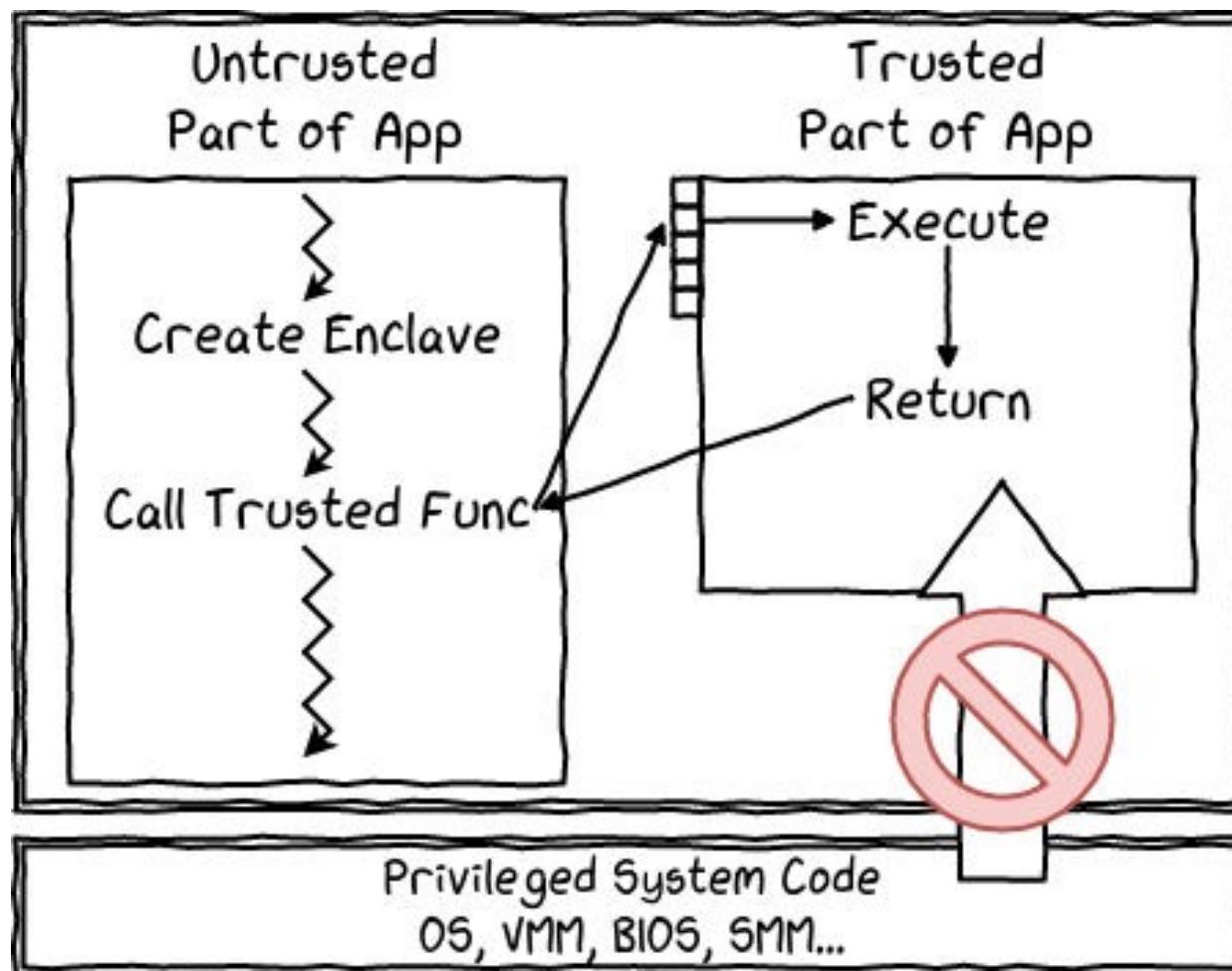
Outline

- What are SGX enclaves?
- What are they good for?
- How are they supported?
- Vulnerabilities
- Alternatives + Next generation: Intel TDX

Enclave abstraction



10,000 feet view of enclave invocation



Source: <https://blog.quarkslab.com/overview-of-intel-sgx-part-1-sgx-internals.html>

Security mechanisms of enclaves

- **Isolation** – its own data and code cannot be seen (confidentiality) or modified (integrity) by any other software on the same machine, including hypervisor, OS, or other enclaves
- **Attestation** – means to prove to a third party the code that is running
- **Sealing** – cryptographically protect data before it leaves enclave
- (While providing a fairly standard programming model, including multithreading and access to application's memory.)

Remote attestation

- Means by which a process, running on a remote machine, determines the TCB of the enclave:
 - Enclave produces a digest of its code, initial data
 - Also reports its CPU firmware and hardware
 - Simple protocol allows for simultaneously sending this digest, signed by the private key of the machine where enclave runs and establishing an authenticated channel between enclave and remote attesting party
 - This private key of the machine is, in turn, vouched by Intel (details are a bit more complex, involve contacting Intel's attestation service)
 - Also supported by TPM

Data sealing

- What if the enclave wants to have state persisted across calls?
- Sealing – enclave data can be encrypted before being stored persistently...
- ... in such a way that it can only be decrypted by the same enclave code running on the same platform
- however, it is vulnerable to a replay attack

Lecture outline

- What are SGX enclaves?
- **What are they good for?**
- How are they supported?
- Vulnerabilities
- Alternatives + Next generation: intel TDX



Azure

Explore ▾ Products ▾ Solutions ▾ Pricing ▾ Partners ▾ Resources ▾

Search

Learn Support Contact Sales

Free account

Sign in

Home / Google Cloud Overview Solutions Products Pricing Resources Docs Support Language Console

Security and identity products

IBM Cloud Products Solutions Pricing Docs Support Explore more Contact us Log in Create IBM Cloud account

IBM Cloud Hyper Protect Crypto Services

IBM Cloud Hyper Protect Crypto Services

Contact Us Support English My Account Sign In Create an AWS Account

Products Solutions Pricing Documentation Learn Partner Network AWS Marketplace Customer Enablement Events Explore More

AWS Nitro Enclaves Features FAQs

AWS Nitro Enclaves

Create additional isolation to further protect highly sensitive data within EC2 instances

Get started with AWS Nitro Enclaves



Technology preview: Private contact discovery for Signal

moxie0 on 26 Sep 2017

At Signal, we've been thinking about [the difficulty of private contact discovery](#) for a long time. We've been working on strategies to improve our current design, and today we've [published a new private contact discovery service](#).

Using this service, Signal clients will be able to efficiently and scalably determine whether the contacts in their address book are Signal users **without revealing the contacts in their address book to the Signal service.**

SGX contact discovery

Private contact discovery using SGX is fairly simple at a high level:

1. Run a contact discovery service in a secure SGX enclave.
2. Clients that wish to perform contact discovery negotiate a secure connection over the network all the way through the remote OS to the enclave.
3. Clients perform remote attestation to ensure that the code which is running in the enclave is the same as the expected published open source code.
4. Clients transmit the encrypted identifiers from their address book to the enclave.
5. The enclave looks up a client's contacts in the set of all registered users and encrypts the results back to the client.

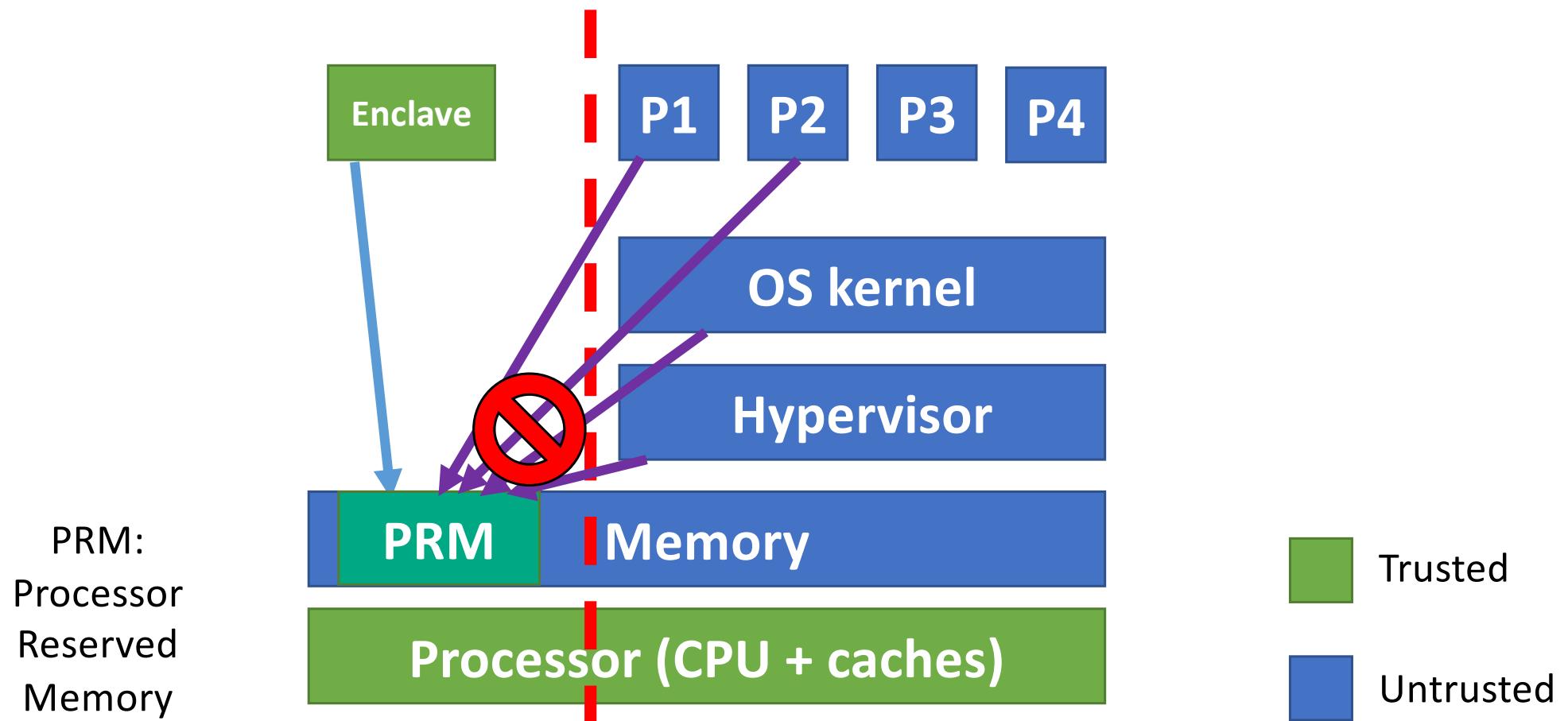
Lecture outline

- What are SGX enclaves?
- What are they good for?
- **How are they supported?**
- Vulnerabilities
- Alternatives + Next generation: intel TDX

Implementation of enclaves in SGX

- Goal: isolate enclave code and data from:
 - OS
 - Hypervisor
 - Regular processes
 - Devices attached to system bus
- TCB: enclave code + everything inside CPU chip
- Does not trust DRAM memory

Goal: isolation of enclave code+data



Lecture outline

- What are SGX enclaves?
- What are they good for?
- How are they supported?
- **Vulnerabilities**
- Alternatives + Next generation: intel TDX

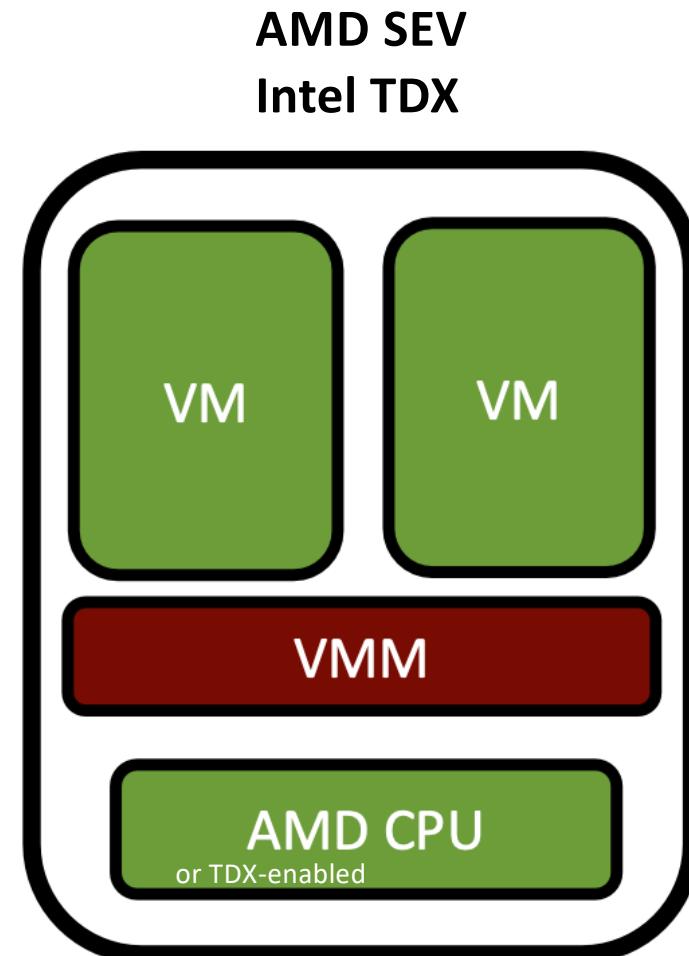
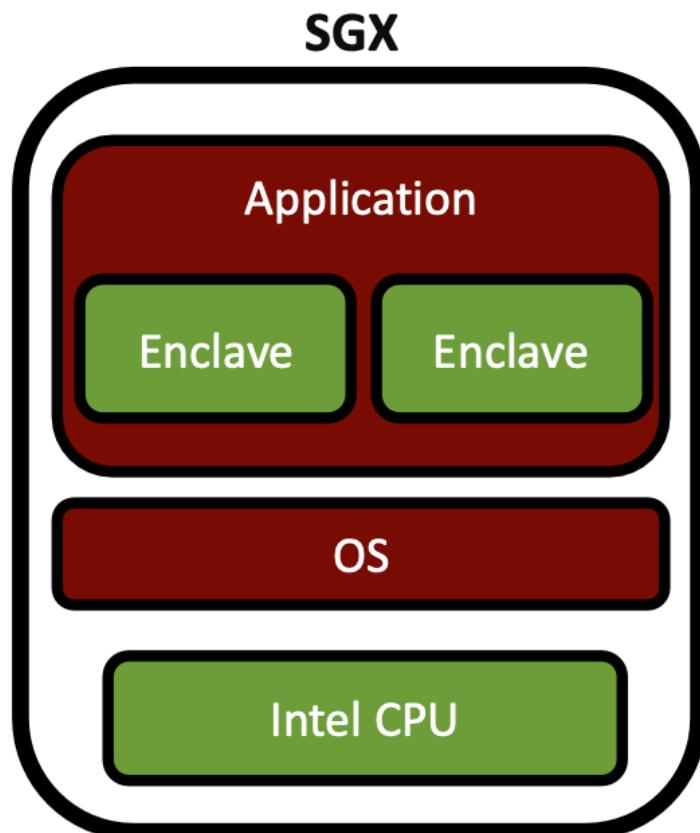
Attacks on Intel SGX

- Most of them are side-channel attacks
- Recently, lots of attention in the security community to Spectre and Meltdown → initially in “normal” code, but researchers found these to be applicable to SGX enclaves (with some adaptations)
- Exploit speculative execution
 - Processors execute a few instructions ahead, and rollback if needed
 - However, rollback does not erase subtle side effects such as cache modifications

Lecture outline

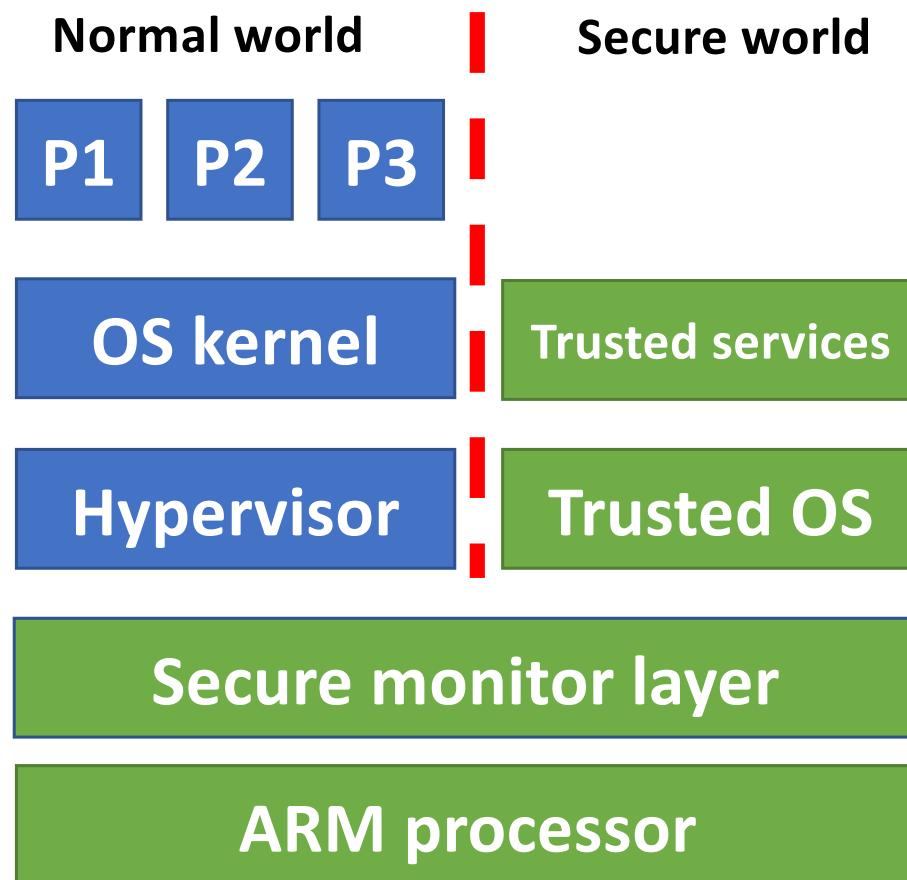
- What are SGX enclaves?
- What are they good for?
- How are they supported?
- Vulnerabilities
- Alternatives + Next generation: intel TDX

TDX + AMD SEV



Adapted from: M. Yan.
MIT 6.888 course slides

ARM TrustZone



Concluding remarks

- Recall definition from first lecture:
- A system is dependable if it is able to work (i.e., deliver its service) in a way that is justifiably trusted, namely avoiding service failures that are either more frequent or severe than acceptable.
- We saw how the Byzantine model allows for building trust even with a subset of malicious participants
 - Key for distributed trust in blockchain systems
- Hardware can increase trust in some of the system components
 - Namely smartcards and TEEs
- Throughout these topics, you learned several important concepts
- And hopefully had some fun in the process!

Bibliography

[1] Ross Anderson. Security Engineering. 2nd edition. Section 16.6

Available online at

<http://www.cl.cam.ac.uk/~rja14/Papers/SEv2-c16.pdf>

[2] Ross Anderson. Security Engineering. 3rd edition. Section 19.4

Available online at

<https://www.cl.cam.ac.uk/~rja14/Papers/SEv3-ch19-7sep.pdf>

Acknowledgments

- Original slides from Prof. Ricardo Chaves @ IST, AISS course, 2013.
- Adapted for SEC by Miguel Matos, Paolo Romano, Rodrigo Rodrigues.
- Previous acks from AISS: Jean Rodrigues @ STEVENS Institute of Technology
- Berk USTUNDAG @ Istanbul Technical University
- Marc Witteman @Riscure.com
- Dr. Hakim Fourar-Laidi @ Prince Sultan University
- Joshua Lawrence @ Florida State University
- Erik Poll @ University of Nijmegen
- Robert Sloan @ University of Illinois at Chicago