Distributed Systems 2023-2024: Blockchain case study: Ethereum

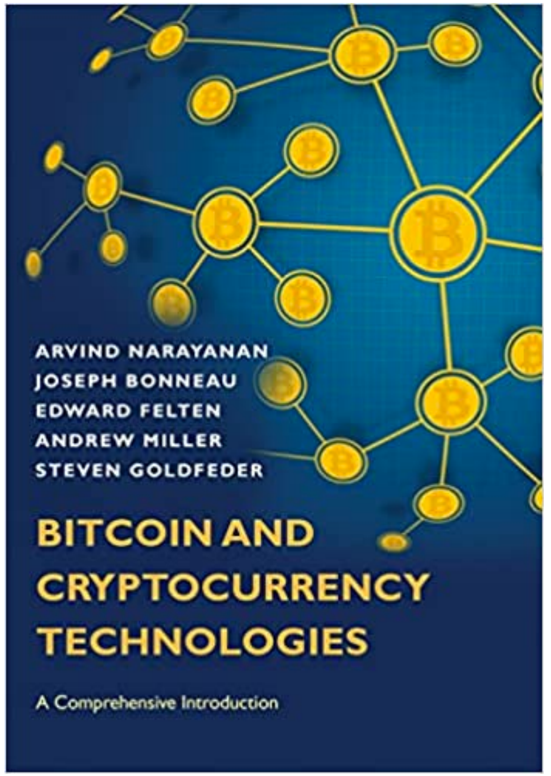
Wouter Joosen & Tom Van Cutsem
DistriNet KU Leuven
December 2023

Learning resources

- Recommended background reading:

- Narayanan *et al.* “Bitcoin and Cryptocurrency Technologies” Princeton University Press - available for free online at: https://bitcoinbook.cs.princeton.edu/

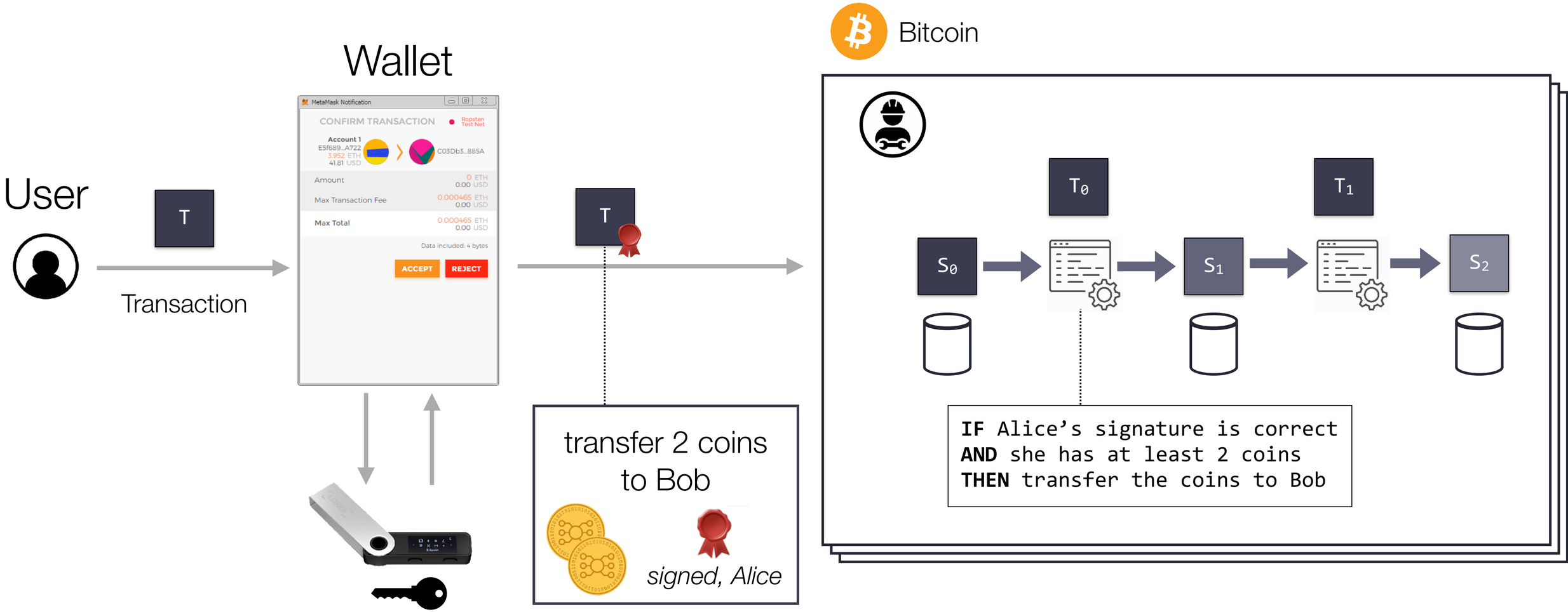
    - Chapter 10, section 10.7 “Ethereum and smart contracts”



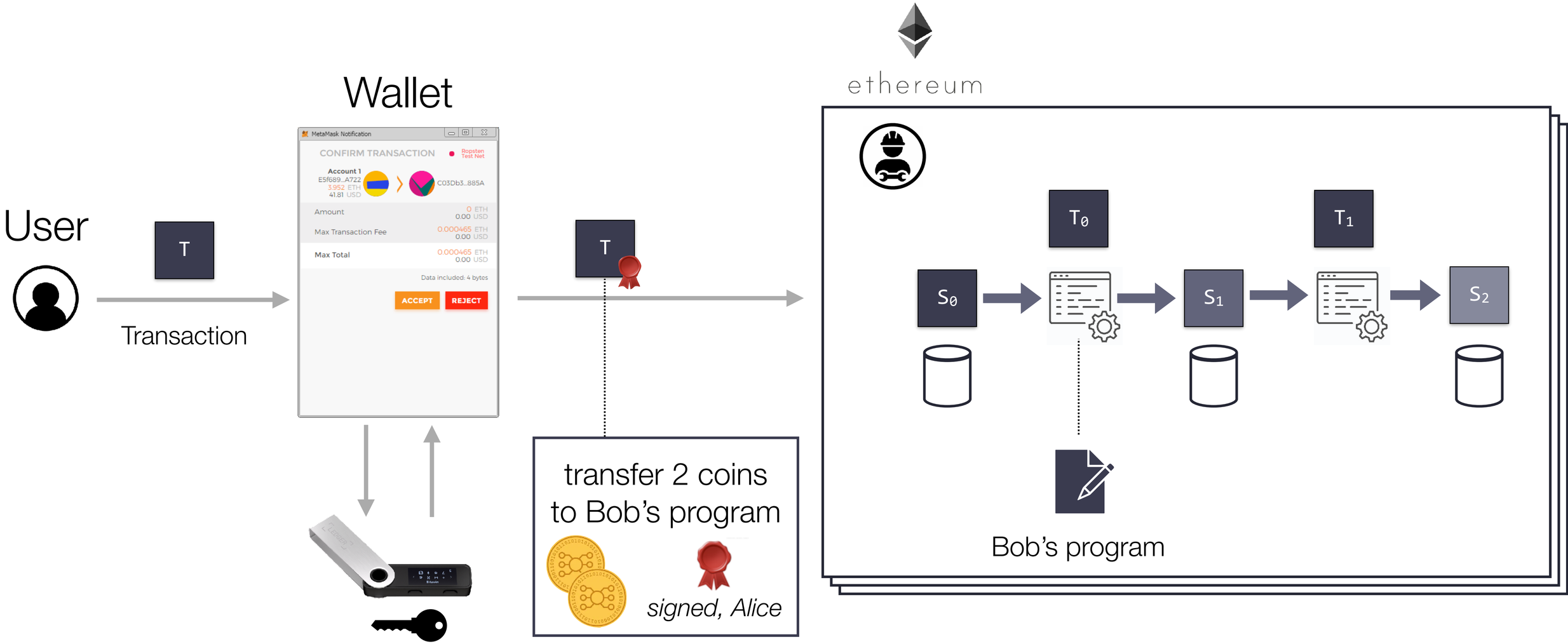
Outline

- A Blockchain case study: **Ethereum**, a “programmable” blockchain
- What is a smart contract? How does it relate to a blockchain network?
- Decentralized Applications (Dapps)
- Writing smart contracts using the Solidity programming language
- Ethereum’s blockchain: accounts, transactions, blocks
- Proof-of-Stake Consensus in Ethereum

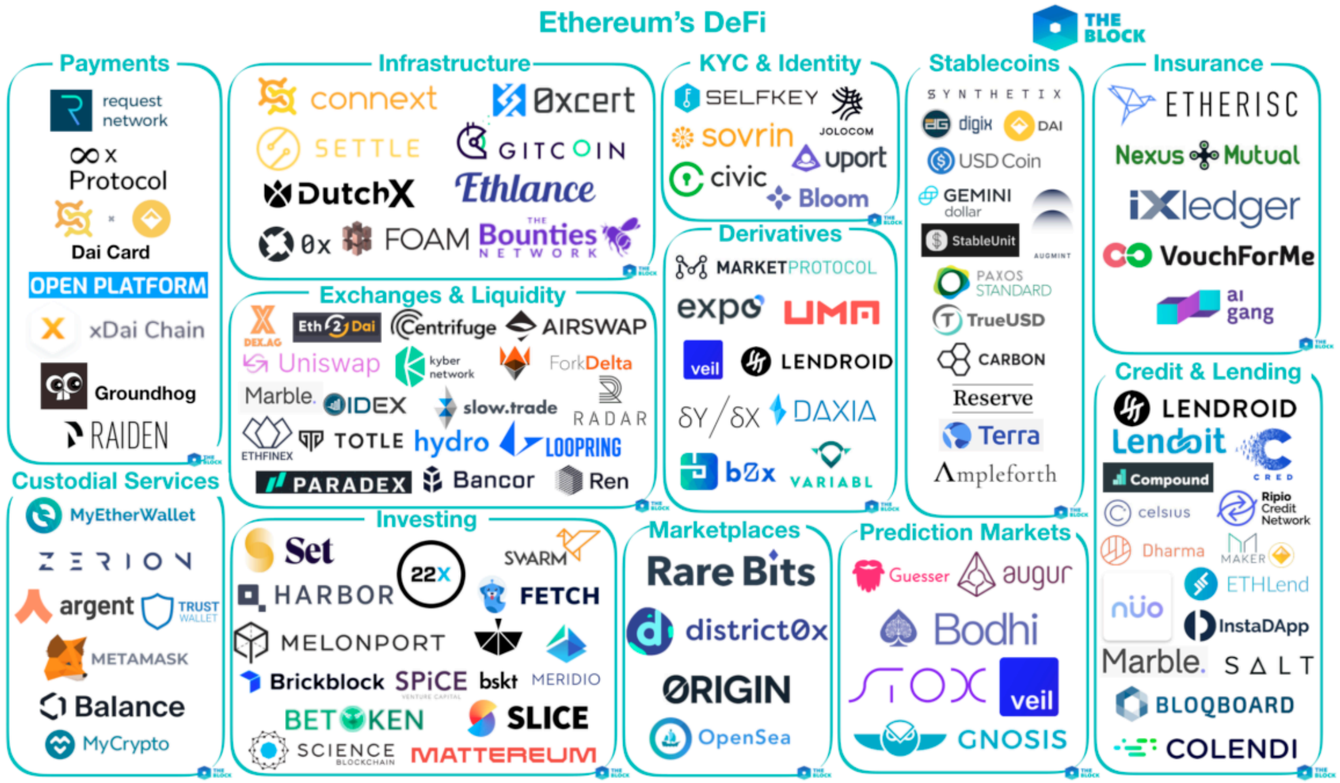
Recall: blockchain networks are replicated state machines



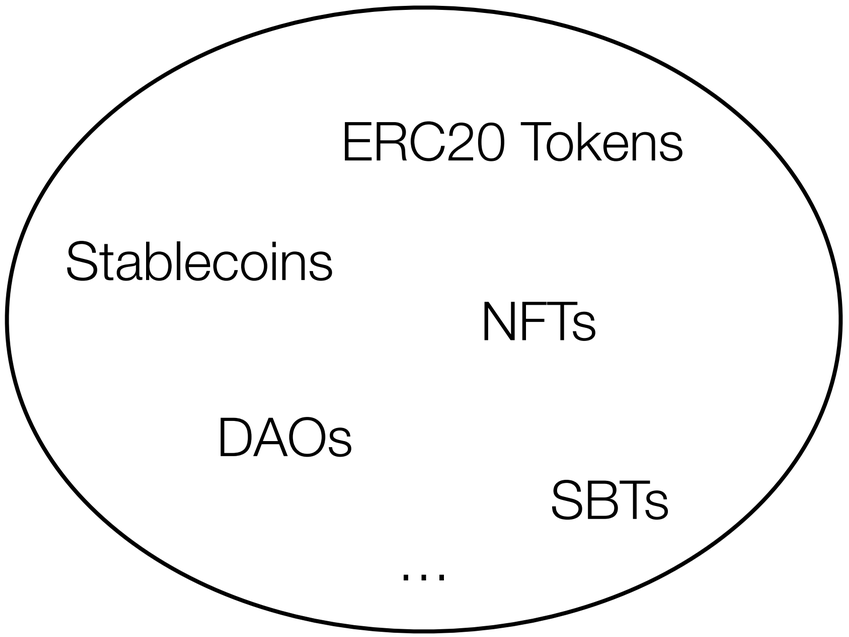
Ethereum: a *programmable* replicated state machine



Ethereum's "decentralized finance" (DeFi) ecosystem



(image credit: theblockcrypto.com)



New kinds of **digital property** collectively worth over **$9.3 trillion**

(source: coincodex.com, retrieved December 2023)

Smart contracts

What is a smart contract?

A software program that automatically moves digital assets according to arbitrary pre-specified rules

Vitalik Buterin, co-inventor of Ethereum
Author of the Ethereum whitepaper (2014)

(Vitalik Buterin, Ethereum White Paper, 2014)

What is a smart contract?

A software program that can receive, store & send “money”

Essentially, a program with its own “bank account”

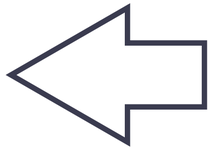
Smart contracts: origins

- The term “smart contract” was first proposed by cryptographer Nick Szabo in 1995.
- **Goal**: digitally automate multi-party business agreements using computer protocols and cryptography to **reduce counterparty risk** (the risk of the other party not executing on what they promised after they agreed to the contract)
- The key idea:
  - Express the **terms & conditions** of a trade agreement **as** executable **code**.
  - Parties agree to the contract by transferring control of their (digital) assets to the contract thus cryptographically “locking up” their assets.
  - The **contract keeps the assets in escrow**. Assets can only be transferred out of the contract according to the logic written in the code.
- A note on **terminology**: smart contracts are neither “smart” as in “using AI”, nor legally binding “contracts”.

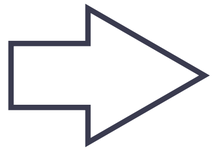
Cryptographer Nick Szabo,
Inventor of the term “smart contract”

Smart contracts: basic principle

- A vending machine is an **automaton** that can trade **physical** assets



1. insert coins



2. dispense drink

Smart contracts: basic principle

- A smart contract is an **automaton** that can trade **digital** assets

code + state

1. insert digital coins (tokens)

2. dispense other digital assets or electronic rights

But who should we trust to faithfully execute the automaton's code?

- A smart contract is an **automaton** that can trade **digital** assets

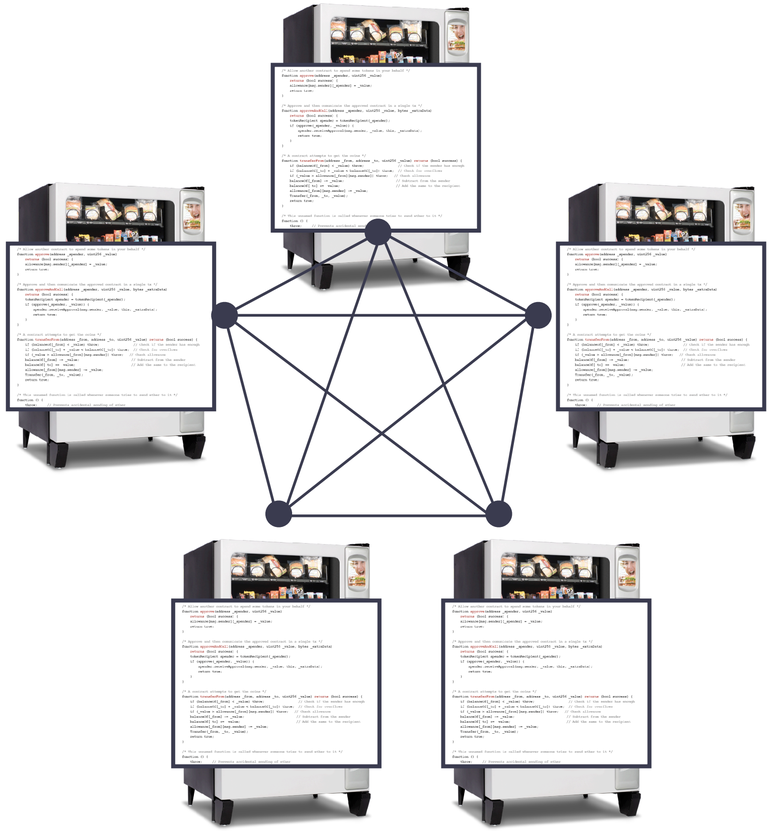
code + state

1. insert digital coins (tokens)

2. dispense other digital assets or electronic rights

Delegate trust to a decentralised network

- A smart contract is a **replicated automaton** that can trade **digital** assets



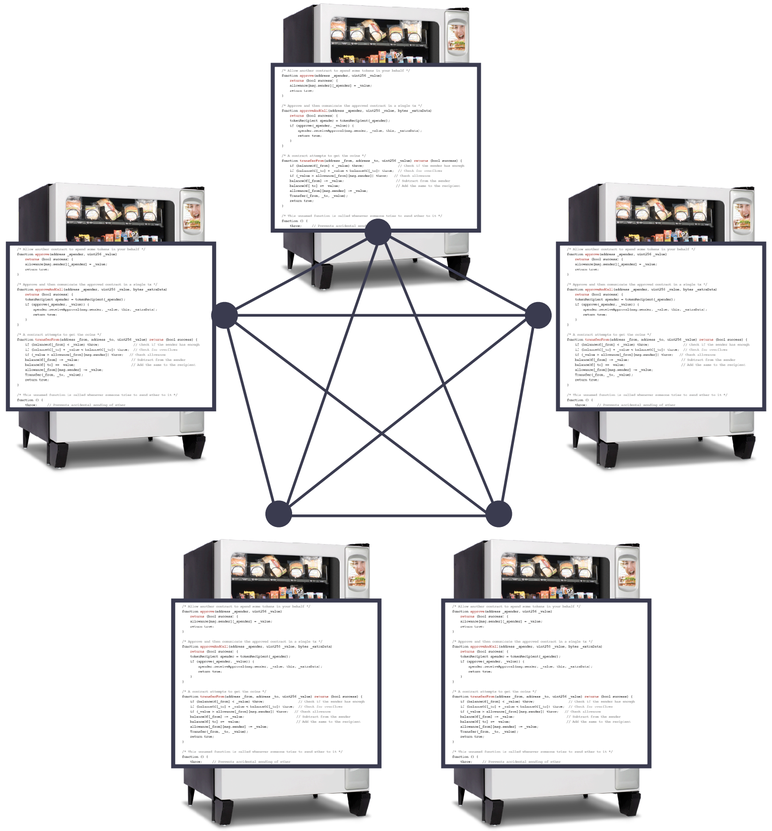
replicated code + state

1. insert digital coins (tokens)

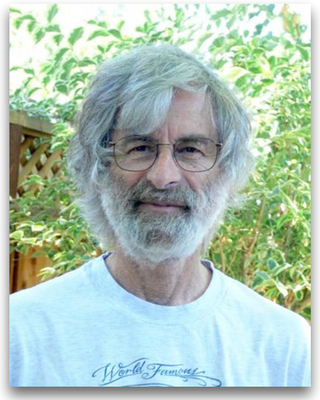
2. dispense other digital assets or electronic rights

Delegate trust to a decentralised network

- A smart contract is a **replicated state machine** that can trade **digital** assets

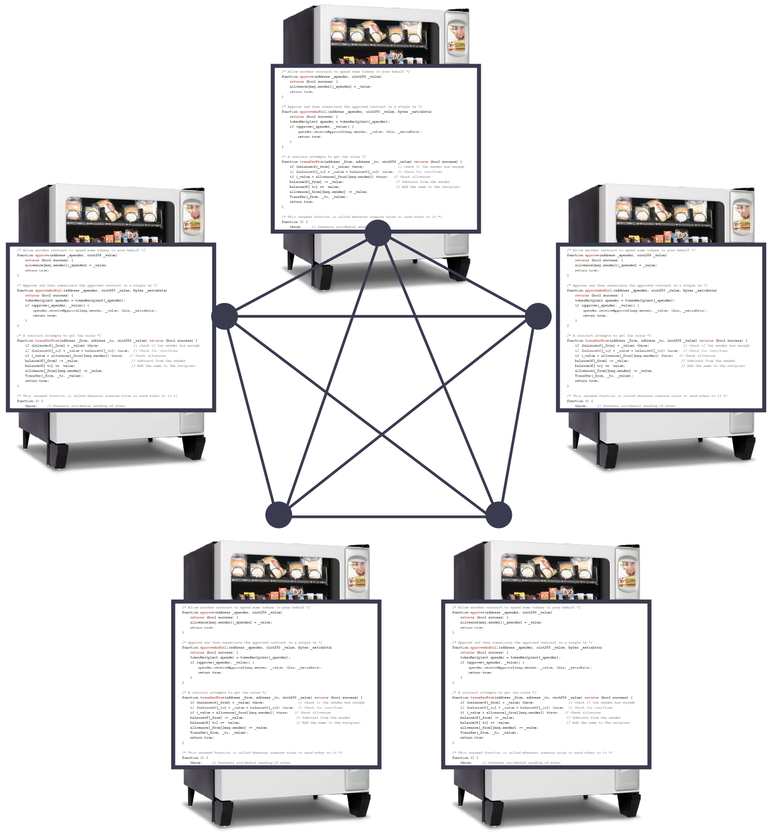


replicated code + state



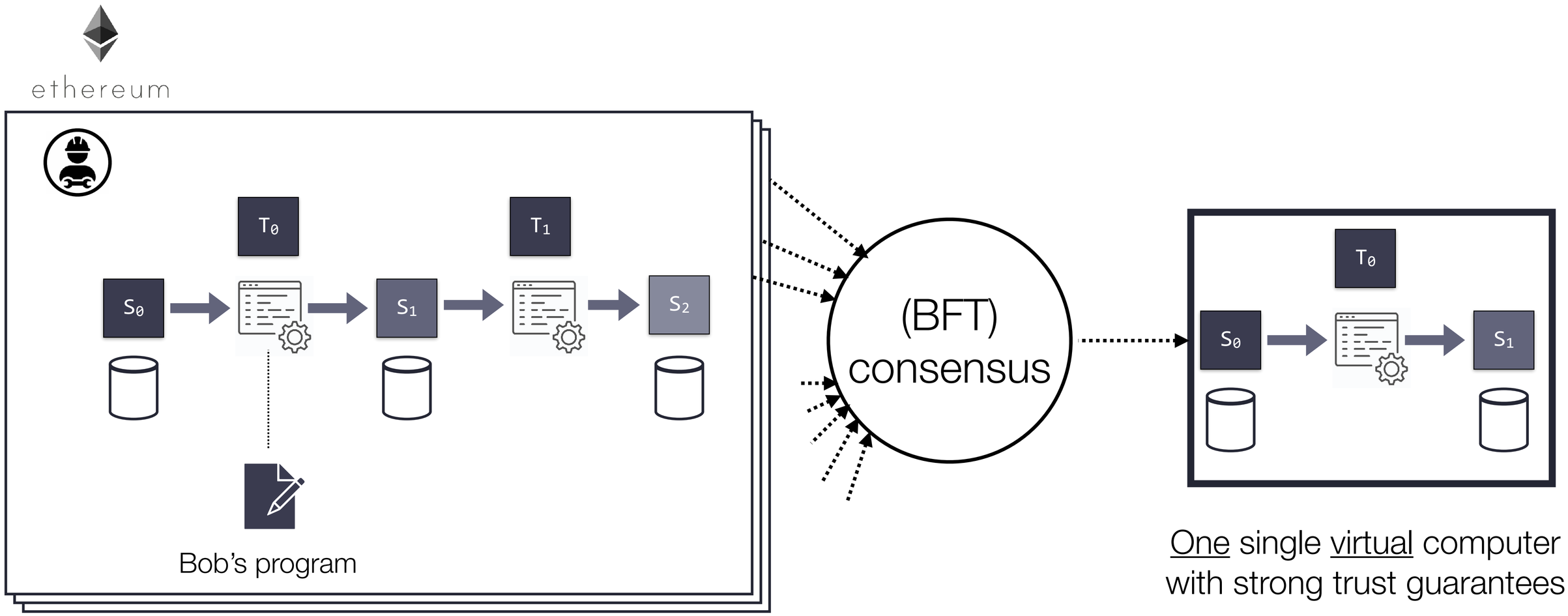
Leslie Lamport
Replicated State Machines (1978)
Byzantine consensus (1982)

Ethereum: the basic principle



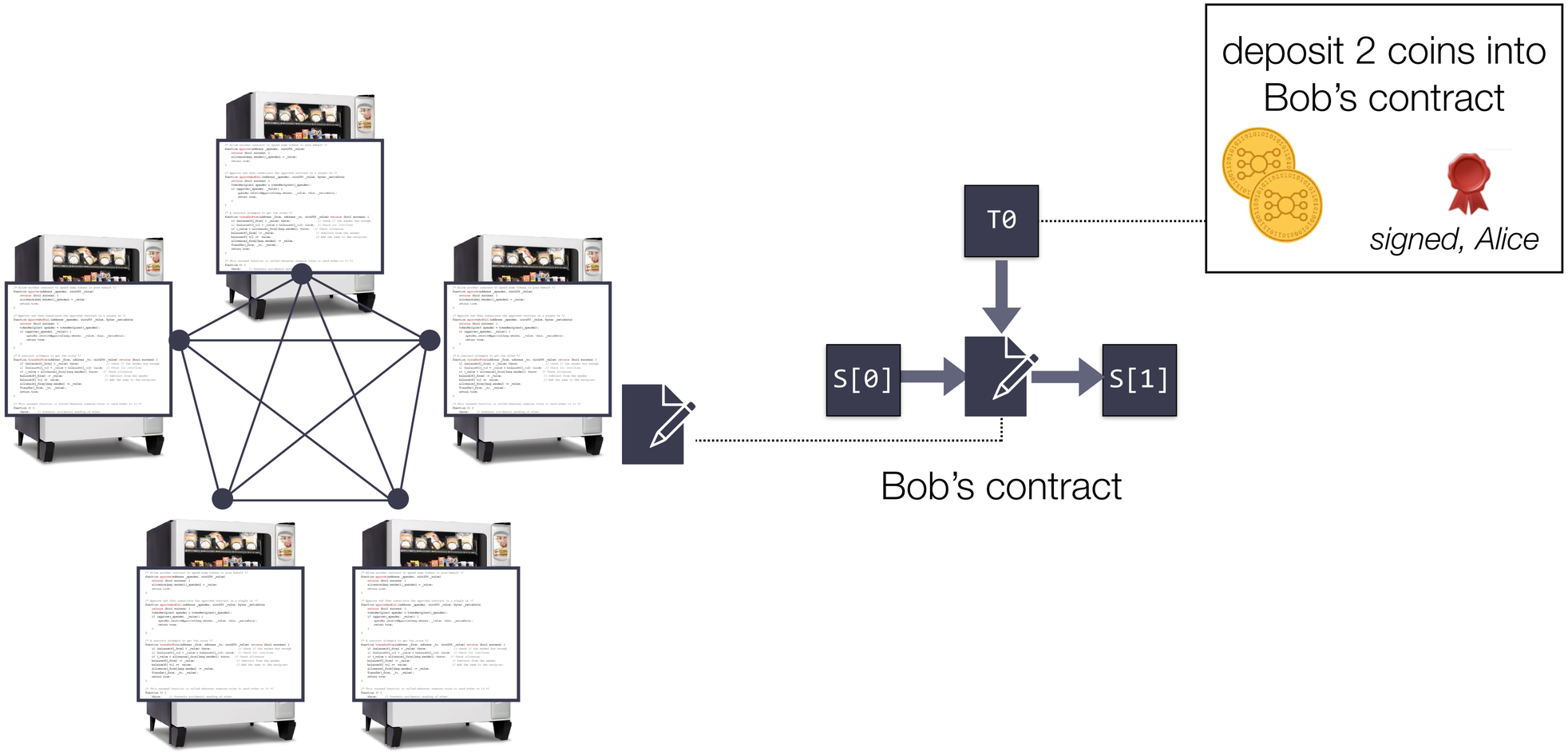
- Ethereum is sometimes described as a “**world computer**”
- It emulates a single *virtual* computer through the collective action of many *physical* computers
- Each individual physical computer is fallible (may crash or be compromised) and untrusted
- As long as the majority of network resources is controlled by honest parties, the single virtual computer is highly available and trustworthy - it is guaranteed to execute the code as described
- For network statistics, see ethernodes.org

A blockchain network can be seen as a trustworthy virtual computer



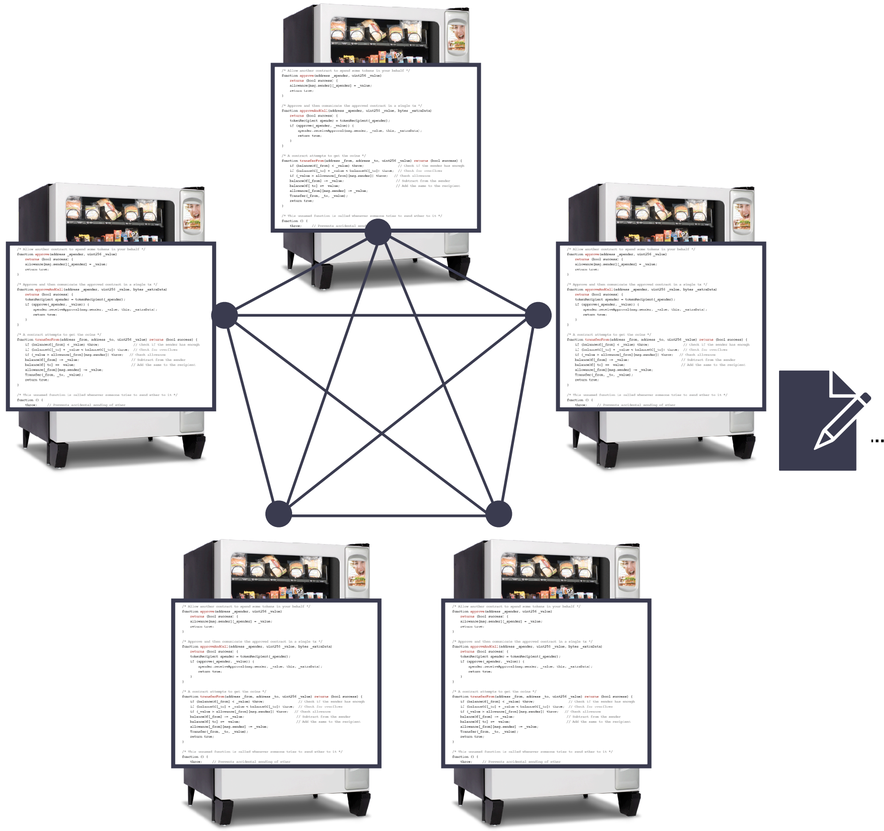
Many (O(1000s)) untrustworthy physical computers

Each transaction updates the virtual computer's replicated state

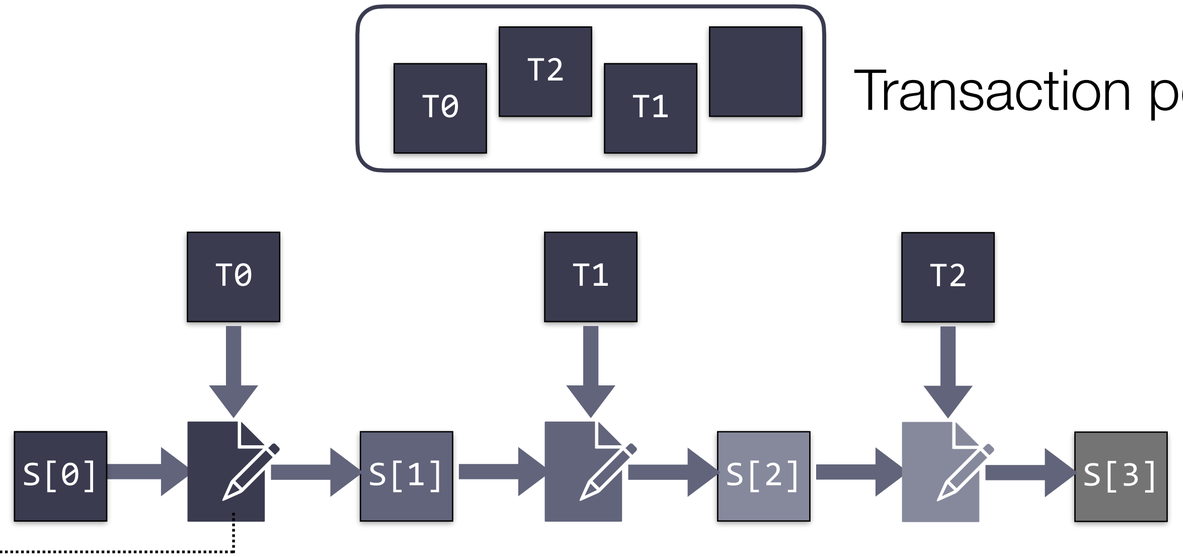


network of validator nodes

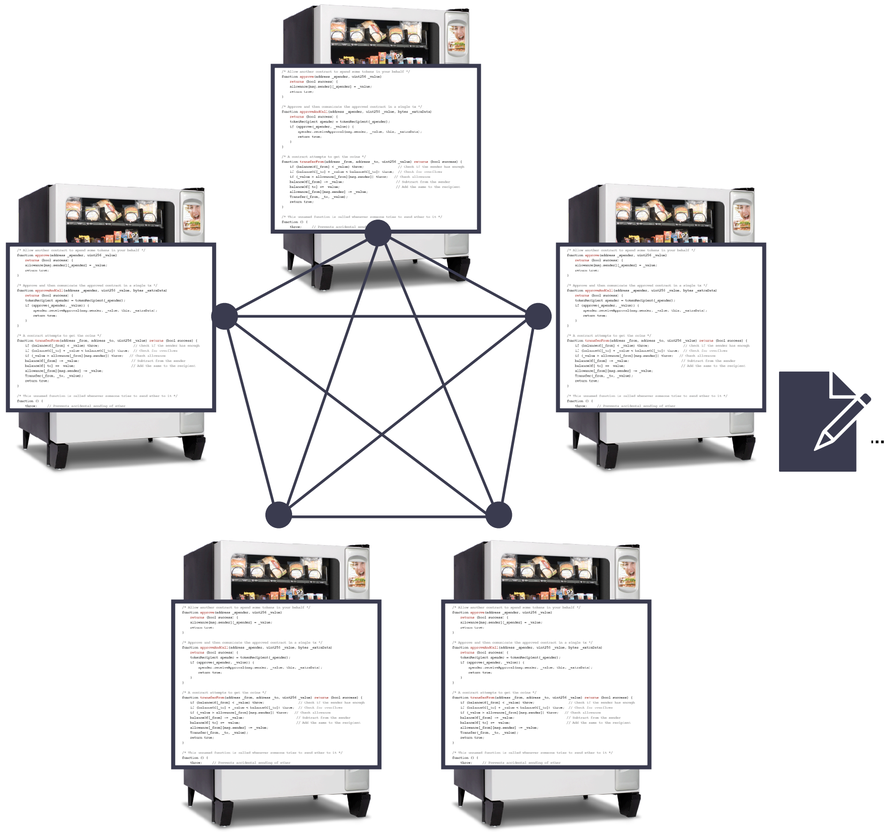
Incoming transactions are sequenced into blocks



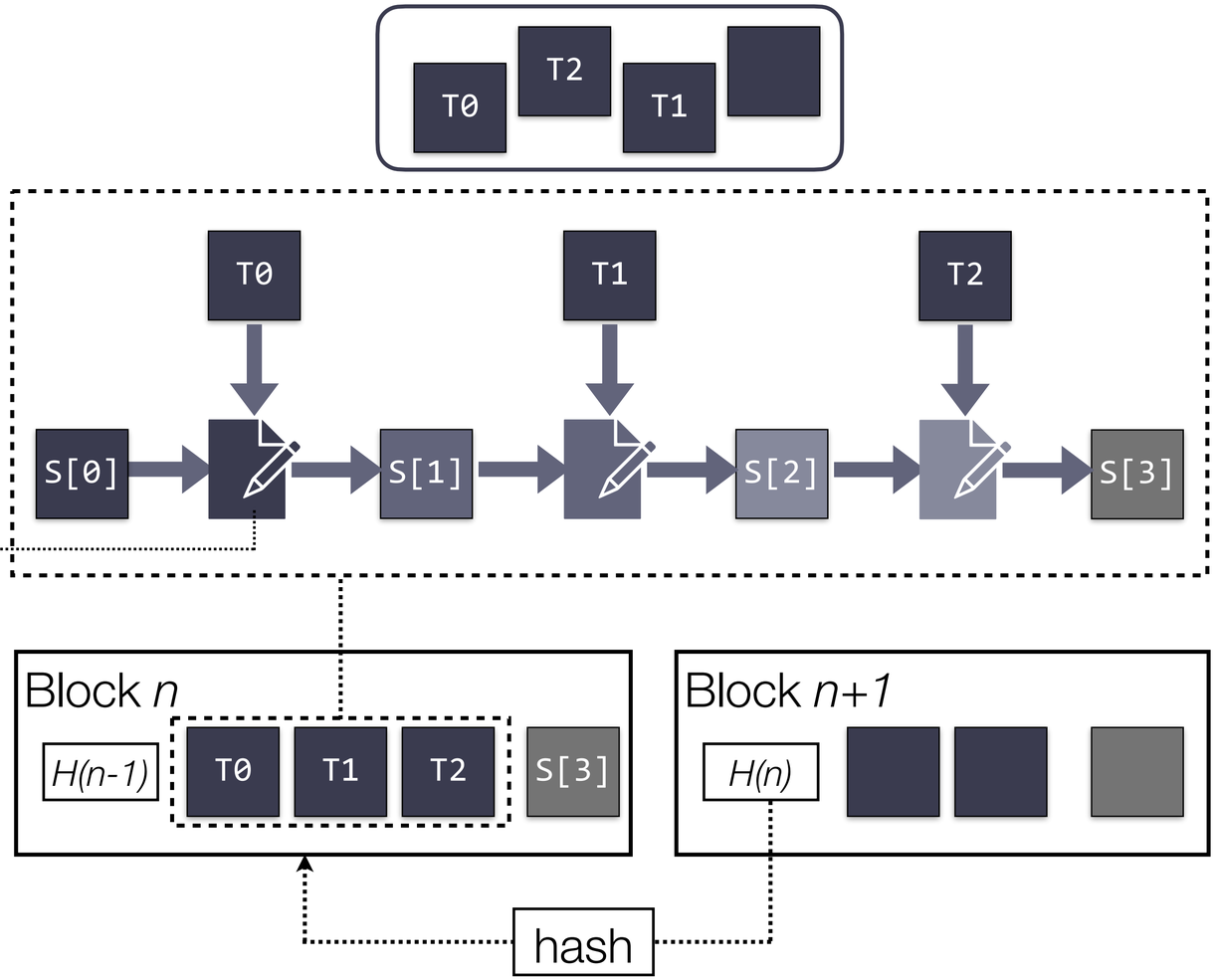
network of validator nodes



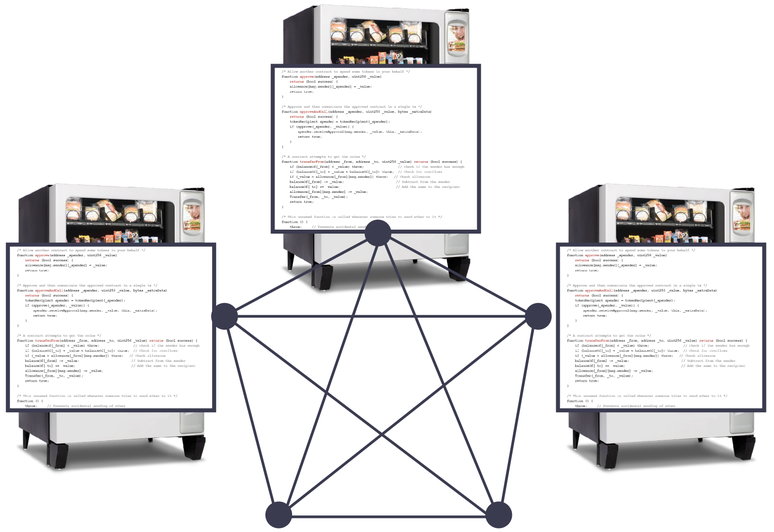
A blockchain ensures the network agrees on a single global order



network of validator nodes



Contracts are written in a high-level language but stored as bytecode



network of validator nodes

Bob's contract

.sol

solidity

solidity source code

Solidity compiler

EVM bytecode

Ethereum Virtual Machine

Decentralized Applications (Dapps)

Decentralized applications: what and why?

- **Decentralized applications (dapps)** are applications where parts of the software architecture are architecturally and politically decentralized.
  - To improve **reliability** (avoid a single point of failure)
  - To achieve **transparency** (publish the application logic on a blockchain, immutable and verifiable by anyone)
  - To resist **censorship** (avoid a single point of control)

Decentralized applications: examples

Decentralized autonomous organizations (DAOs)

Decentralized lending and borrowing protocol

Decentralized exchanges
Atomic token swaps

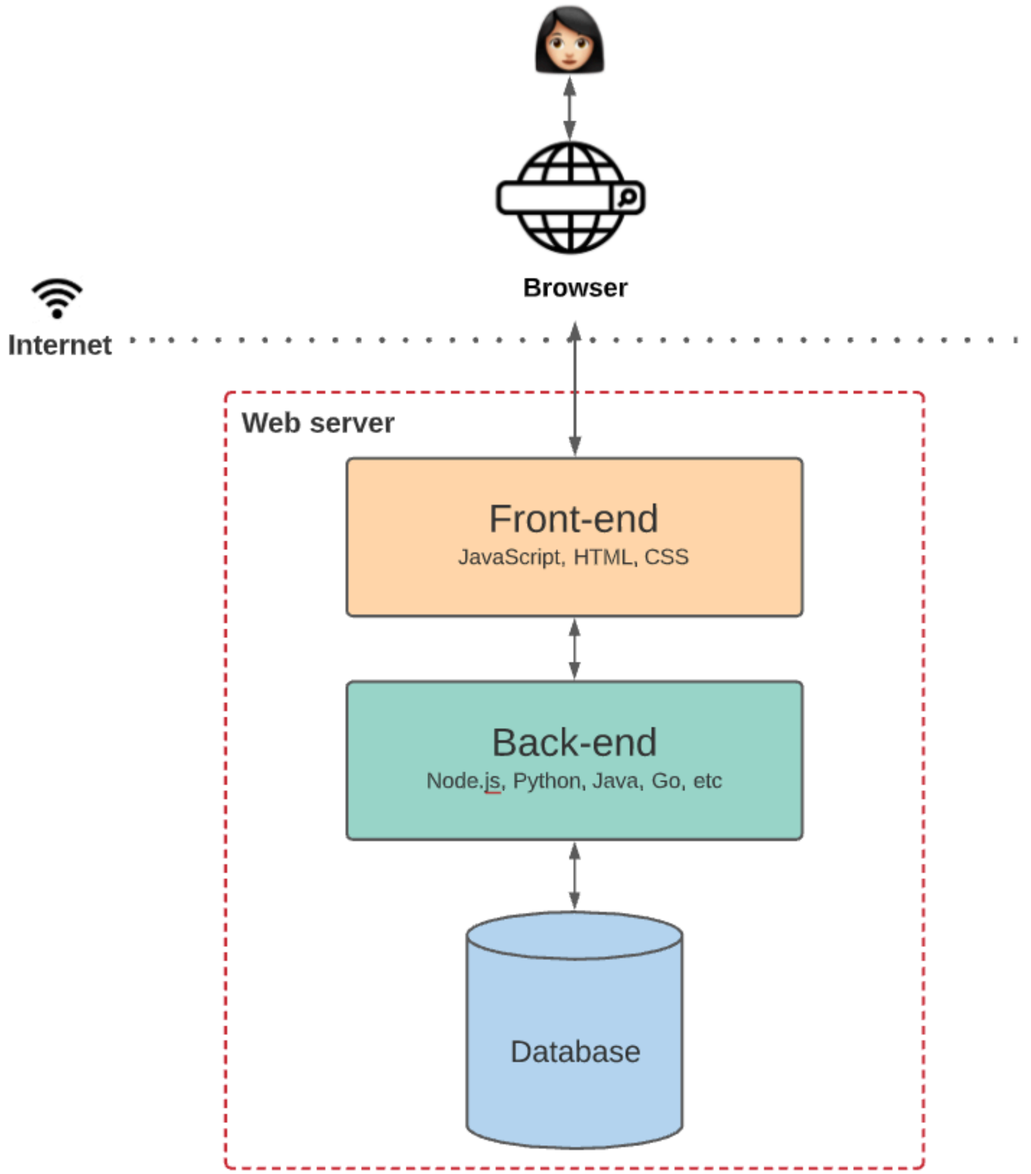
Decentralized prediction markets & betting platforms

“Play-to-earn” games

Decentralized crowd-funding

Traditional Web application architecture

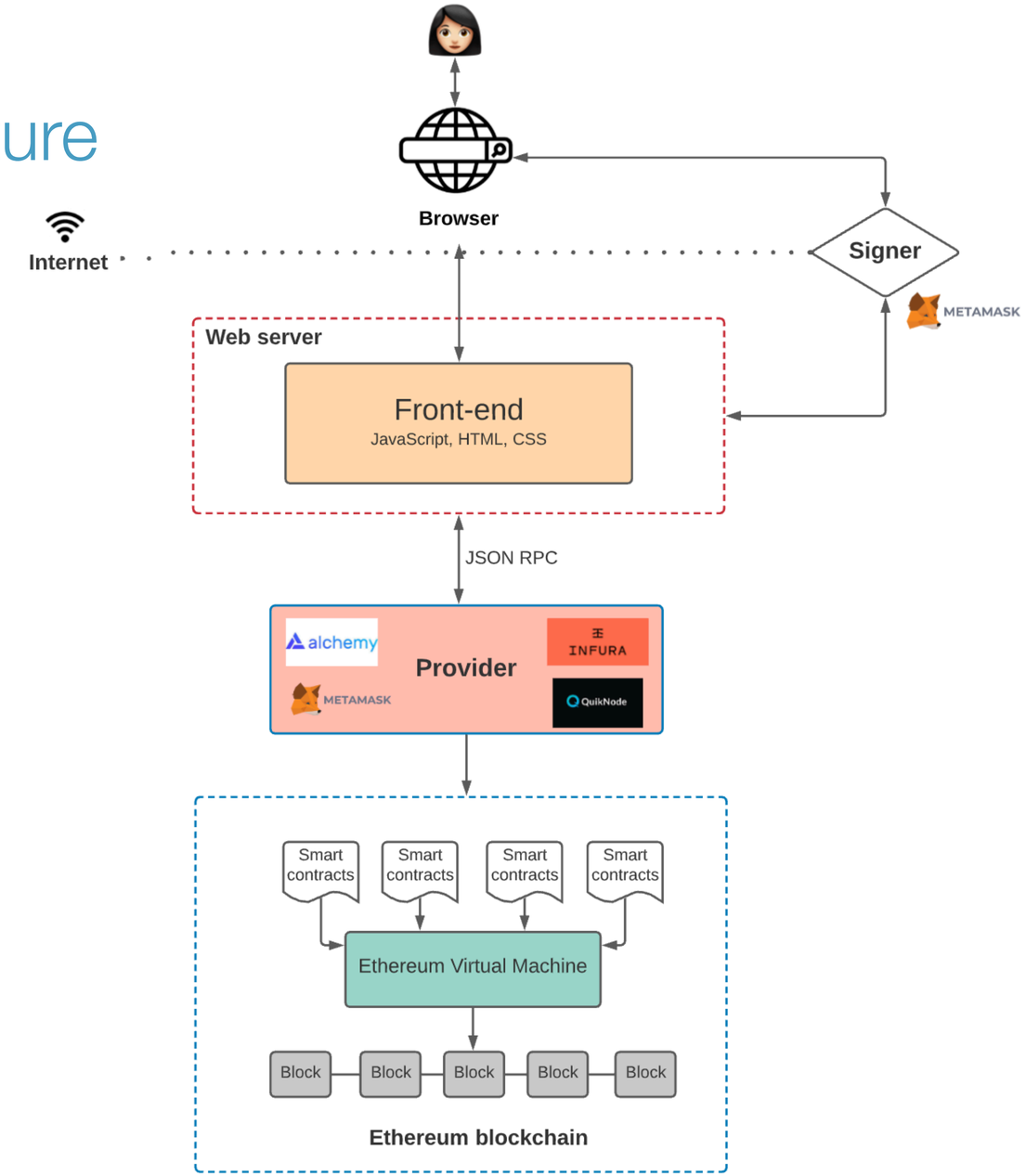
- Following a standard “3-tier” architecture:
- **Front-end**: code that runs in the browser (or on a mobile app), mostly UI logic
- **Back-end**: code that runs on a web server, focus on business logic
- **Database**: persists the application state
- It is common for the application to define the user’s identity and to store username and password in the database. The user **does not control** their identity.



(Source: P. Kasireddy, “The Architecture of a Web 3.0 application”, Medium.com: https://www.preethikasireddy.com/post/the-architecture-of-a-web-3-0-application)

Decentralized Web application architecture

- **Front-end**: largely unchanged (mostly UI logic)
- **Back-end**: (part of) the application logic is implemented as a smart contract and published on the blockchain
- **Database?** The state of the smart contract is persisted on the blockchain (replicated across all validator nodes)
- **Provider**: browsers or mobile apps cannot easily participate directly as a peer in the blockchain network, so usually send their requests through a web server that **relays** the request to a peer in the blockchain network.
- **Signer**: for any user action that results in an update to the smart contract, a **signature** is needed from the user. This task typically delegated to a wallet that securely stores the user's keys. The **user retains control** over their keys (they are *not* stored or controlled by the application).



Ethereum: addresses and accounts

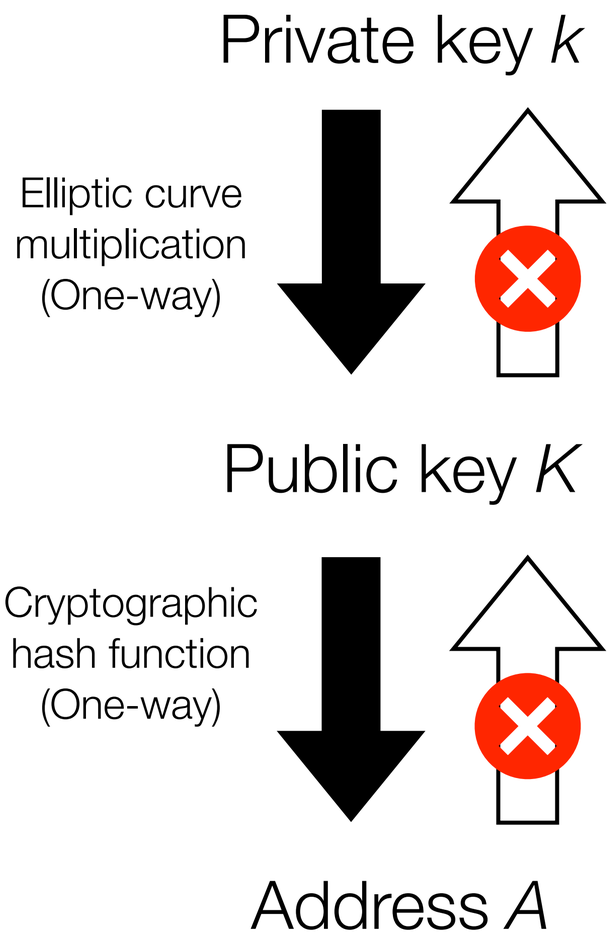
Addresses

- In reality, blockchains do not use “account names” like Alice and Bob
- Instead, users are **pseudonymous** and identified by their **address** instead
- In Ethereum, addresses are 20 bytes (160 bits)
- Typically formatted as a 40-digit hex string prefixed with “0x”
- Example Ethereum address:

0x001d3f1ef827552ae1114027bd3ecf1f086ba0f9

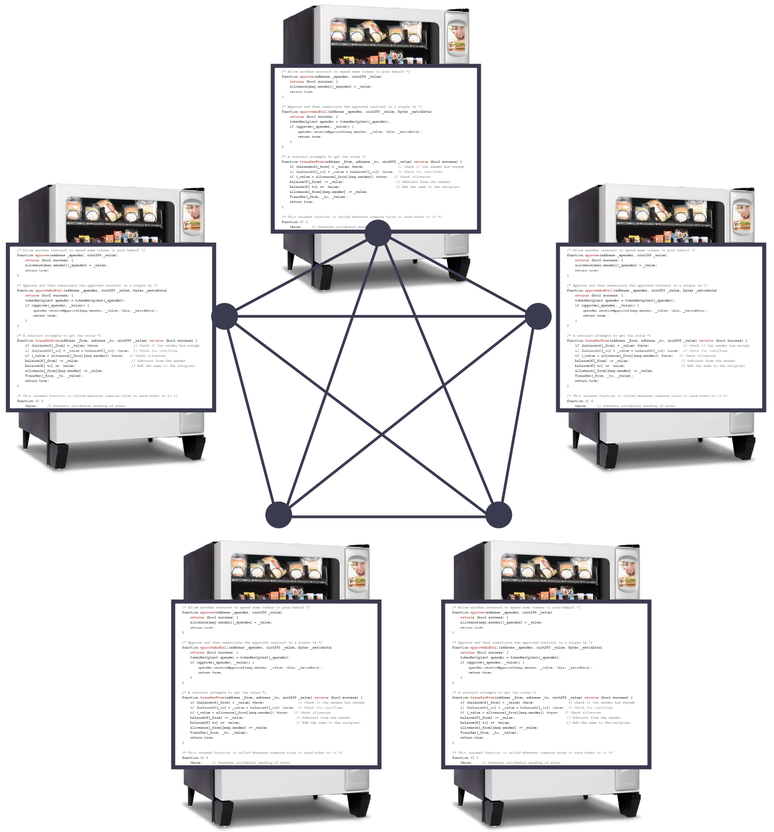
Accounts

- To create an account, generate a new public/private key pair:
  - The **private key** is generated (this involves a secure random number generator)
  - The **public key** can be derived from the private key, but not the other way around!
  - The account **address** is created by hashing the public key and selecting the last 20 bytes of the hash
- Access to **public key** or address allows one to **query** the account balance (all account balances are public anyway)
- Access to **private key** allows one to **spend** the account balance (by digitally **signing** transactions that transfer coins to a new address)



Ethereum: Smart contracts

Solidity: a programming language to write smart contracts



network of validator nodes

Bob's contract

.sol

solidity

solidity source code

Solidity compiler

EVM bytecode

Ethereum Virtual Machine

Smart contracts on Ethereum: a basic example


```
contract NameRegistry {

    mapping (string => address) public registry;

    constructor() {}

    function claimName(string name) public payable {
        require(msg.value >= 1 ether);
        if (registry[name] == address(0)) {
            registry[name] = msg.sender;
        }
    }

    function ownerOf(string name) public view {
        return registry[name];
    }
}
```

(Code example based on Narayanan *et al.* handbook, section 10.7)

Smart contracts on Ethereum: a basic example

Define a new contract.

```
contract NameRegistry {

    mapping (string => address) public registry;

    constructor() {}

    function claimName(string name) public payable {
        require(msg.value >= 1 ether);
        if (registry[name] == address(0)) {
            registry[name] = msg.sender;
        }
    }

    function ownerOf(string name) public view {
        return registry[name];
    }
}
```

Smart contracts on Ethereum: a basic example

```
contract NameRegistry {

    mapping (string => address) public registry;

    constructor() {}

    function claimName(string name) public payable {
        require(msg.value >= 1 ether);
        if (registry[name] == address(0)) {
            registry[name] = msg.sender;
        }
    }

    function ownerOf(string name) public view {
        return registry[name];
    }
}
```

Define the contract state.

All state is replicated and publicly persisted on the blockchain.

Smart contracts on Ethereum: a basic example

```
contract NameRegistry {

    mapping (string => address) public registry;

    constructor() {}

    function claimName(string name) public payable {
        require(msg.value >= 1 ether);
        if (registry[name] == address(0)) {
            registry[name] = msg.sender;
        }
    }

    function ownerOf(string name) public view {
        return registry[name];
    }
}
```

Define a constructor.

The constructor is run once during creation of the contract and cannot be called afterwards.

We don’t need to do any initialisation in this simple contract. The mapping by default maps every string to the 0 address

Smart contracts on Ethereum: a basic example

```
contract NameRegistry {

    mapping (string => address) public registry;

    constructor() {}

    function claimName(string name) public payable {
        require(msg.value >= 1 ether);
        if (registry[name] == address(0)) {
            registry[name] = msg.sender;
        }
    }

    function ownerOf(string name) public view {
        return registry[name];
    }
}
```

Define functions.

Can be called by users or by other contracts.

Can update the contract's state.

Functions can be "called" by sending a transaction to the Ethereum network.

Smart contracts on Ethereum: a basic example

```
contract NameRegistry {

    mapping (string => address) public registry;

    constructor() {}

    function claimName(string name) public payable {
        require(msg.value >= 1 ether);
        if (registry[name] == address(0)) {
            registry[name] = msg.sender;
        }
    }

    function ownerOf(string name) public view {
        return registry[name];
    }
}
```

A table that keeps track of the owner address of each registered name

string	address
“Alice”	0xde0b295669a9fd93d5f…
“Bob’s program”	0x2212D359CF1c5454Ae9…
“a message”	0x721E221531b7bC98DB2…
“ethereum.org”	0xC55EdDadEeB47fcDE0B…

Smart contracts on Ethereum: a basic example

```
contract NameRegistry {

    mapping (string => address) public registry;

    constructor() {}

    function claimName(string name) public payable {
        require(msg.value >= 1 ether);
        if (registry[name] == address(0)) {
            registry[name] = msg.sender;
        }
    }

    function ownerOf(string name) public view {
        return registry[name];
    }
}
```

Functions are “called” by sending a transaction.

Each transaction is cryptographically signed by the sender and contains the sender’s address (`msg.sender`) and may optionally contain any amount of tokens (ether) sent along with it (`msg.value`).

Smart contracts on Ethereum: a basic example

```
contract NameRegistry {

    mapping (string => address) public registry;

    constructor() {}

    function claimName(string name) public payable {
        require(msg.value >= 1 ether);
        if (registry[name] == address(0)) {
            registry[name] = msg.sender;
        }
    }

    function ownerOf(string name) public view {
        return registry[name];
    }
}
```

Bob can register the name “Bob” by creating a transaction containing at least 1 ether and calling the `claimName()` function

“Alice”	0xde0b295669a9fd93d5f…
“Bob’s program”	0x2212D359CF1c5454Ae9…
“a message”	0x721E221531b7bC98DB2…
“ethereum.org”	0xC55EdDadEeB47fcDE0B…
“bob”	0x931D387731bBbC988B3…

Smart contracts on Ethereum: a basic example

```
contract NameRegistry {

    mapping (string => address) public registry;

    constructor() {}

    function claimName(string name) public payable {
        require(msg.value >= 1 ether);
        if (registry[name] == address(0)) {
            registry[name] = msg.sender;
        }
    }

    function ownerOf(string name) public view {
        return registry[name];
    }
}
```

If the function completes without errors, any updates to the state variables are **stored** into the contract's persistent memory and later **committed** on the blockchain (if the transaction is eventually included in a block).

Smart contracts on Ethereum: a basic example

```
contract NameRegistry {

    mapping (string => address) public registry;

    constructor() {}

    function claimName(string name) public payable {
        require(msg.value >= 1 ether);
        if (registry[name] == address(0)) {
            registry[name] = msg.sender;
        }
    }

    function ownerOf(string name) public view {
        return registry[name];
    }
}
```

If a `require()` condition is not met, the transaction **reverts** and any updates to the contract state are rolled back (not persisted)

Here, if Bob does not transfer enough ether along with the transaction he cannot claim the name.

Smart contracts on Ethereum: a basic example

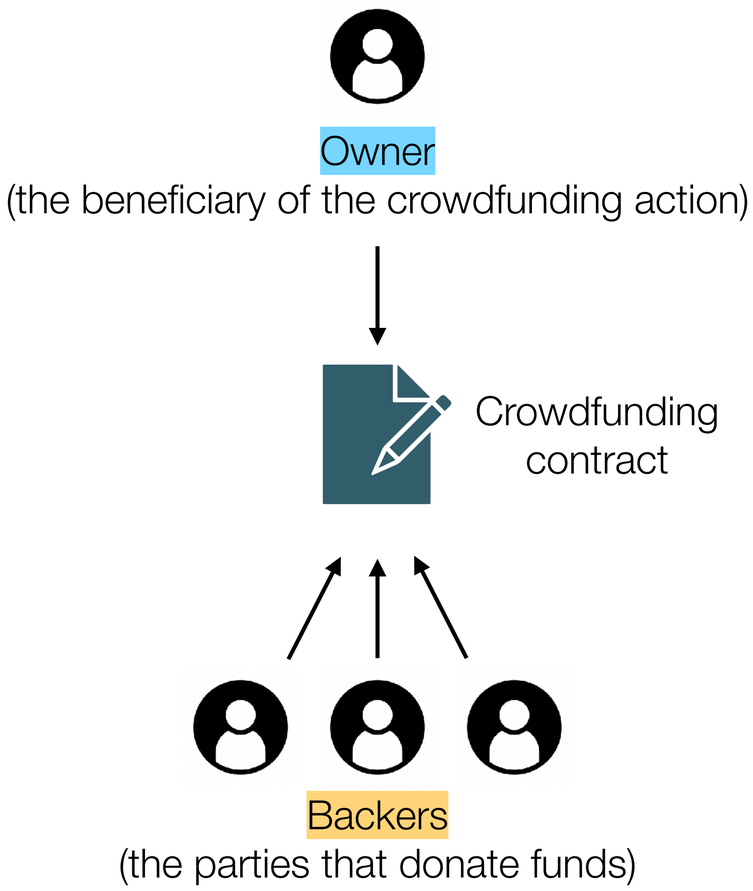
```
contract NameRegistry {

    mapping (string => address) public registry;

    constructor() {}

    function claimName(string name) public payable {
        require(msg.value >= 1 ether);
        if (registry[name] == address(0)) {
            registry[name] = msg.sender;
        }
    }

    function ownerOf(string name) public view {
        return registry[name];
    }
}
```

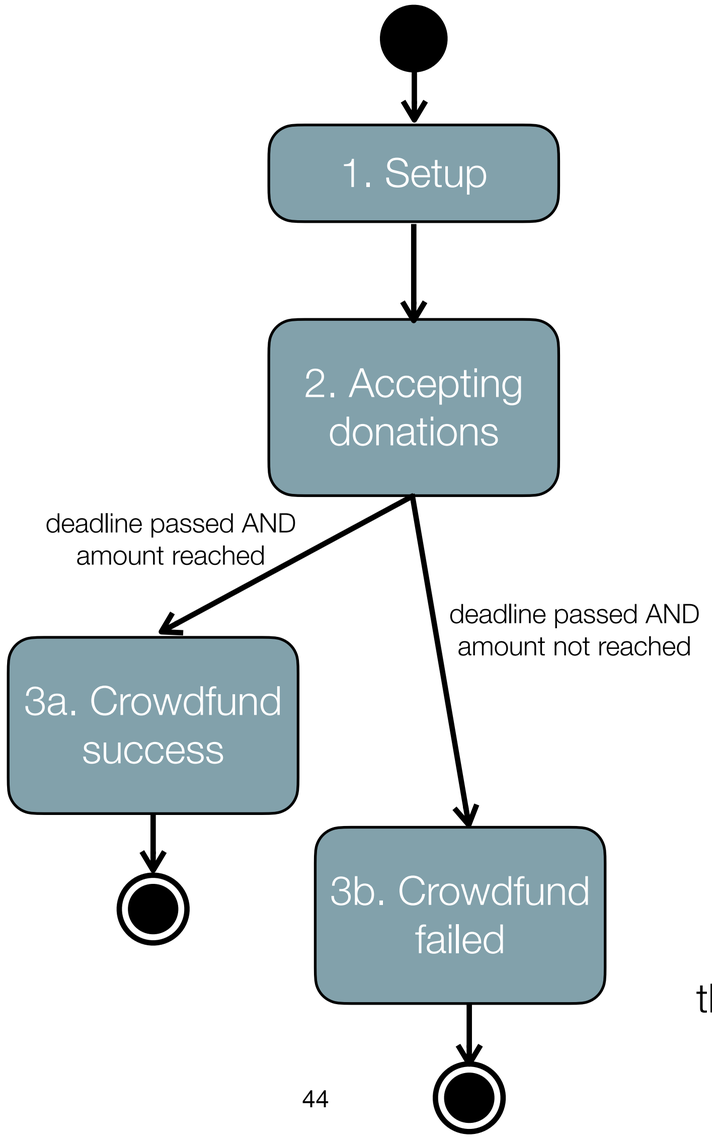
Anyone can lookup ownership of names by calling the **`ownerOf()`** function.

Since the function is read-only (marked as **`view`**), it can also be called locally by a client without creating a transaction and without broadcasting it to the network.

Remix demo https://remix.ethereum.org/

A more complete example: a crowdfunding contract





Step 1: the owner creates the contract, stating target amount + funding deadline (which **cannot be changed** afterwards)

Step 2: backers can donate money (**deposit** funds into the contract) IF the funding deadline has not yet passed

Step 3a (crowdfunding successful): the owner can claim the funds (**withdraw** funds from the contract) IF the funding deadline has passed AND the minimum target amount has been met

Step 3b (crowdfunding failed): backers can reclaim their donations (**withdraw** funds from the contract) IF the funding deadline has passed AND the minimum target amount has **not** been met

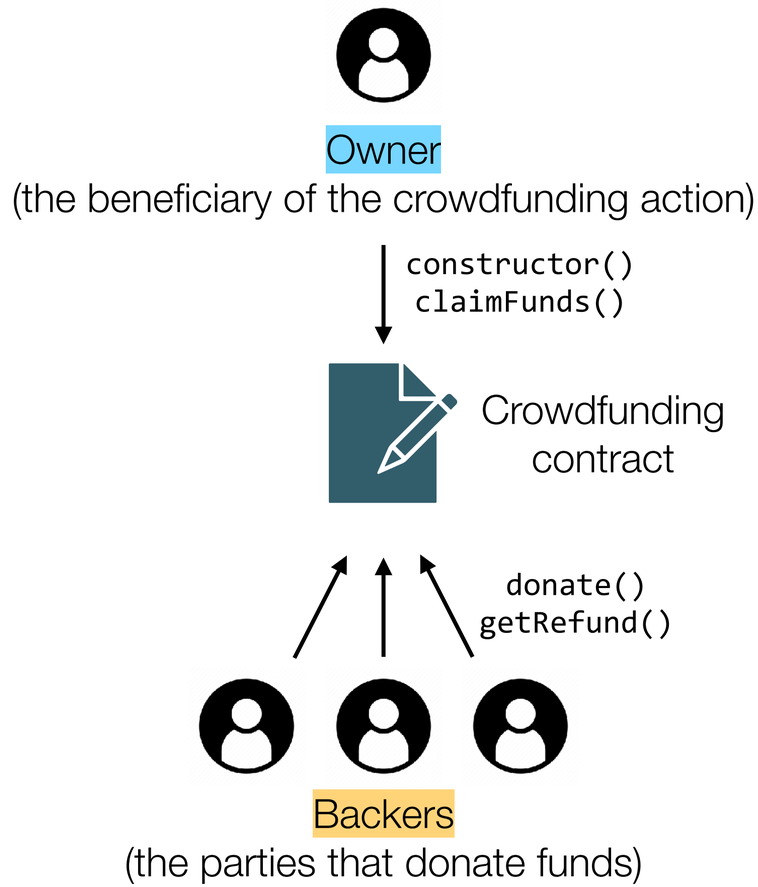
Crowdfunding contract: Solidity source code

```
contract Crowdfunding {
    address public owner;    // the beneficiary address
    uint256 public deadline; // campaign deadline in number of days
    uint256 public goal;     // funding goal in ether
    mapping (address => uint256) public backers; // the share of each backer

    constructor(uint256 numberOfDays, uint256 _goal) {
        owner = msg.sender;
        deadline = block.timestamp + (numberOfDays * 1 days);
        goal = _goal;

    }
    function donate() public payable {
        require(block.timestamp < deadline); // before the fundraising deadline
        backers[msg.sender] += msg.value;

    }

    function claimFunds() public {
        require(address(this).balance >= goal); // funding goal met
        require(block.timestamp >= deadline); // after the withdrawal period
        require(msg.sender == owner);
        payable(msg.sender).transfer(address(this).balance);
    }
    function getRefund() public {
        require(address(this).balance < goal); // campaign failed: goal not met
        require(block.timestamp >= deadline); // in the withdrawal period
        uint256 donation = backers[msg.sender];
        backers[msg.sender] = 0;
        payable(msg.sender).transfer(donation);
    }
}
```

(Based on: Ilya Sergey, “The next 700 smart contract languages”, Principles of Blockchain Systems 2021)

Crowdfunding contract: Solidity source code

```
contract Crowdfunding {
    address public owner;    // the beneficiary address
    uint256 public deadline; // campaign deadline in number of days
    uint256 public goal;     // funding goal in ether
    mapping (address => uint256) public backers; // the share of each backer

    constructor(uint256 numberOfDays, uint256 _goal) {
        owner = msg.sender;
        deadline = block.timestamp + (numberOfDays * 1 days);
        goal = _goal;
    }
    function donate() public payable {
        require(block.timestamp < deadline); // before the fundraising deadline
        backers[msg.sender] += msg.value;
    }
    function claimFunds() public {
        require(address(this).balance >= goal); // funding goal met
        require(block.timestamp >= deadline); // after the withdrawal period
        require(msg.sender == owner);
        payable(msg.sender).transfer(address(this).balance);
    }
    function getRefund() public {
        require(address(this).balance < goal); // campaign failed: goal not met
        require(block.timestamp >= deadline); // in the withdrawal period
        uint256 donation = backers[msg.sender];
        backers[msg.sender] = 0;
        payable(msg.sender).transfer(donation);
    }
}
```



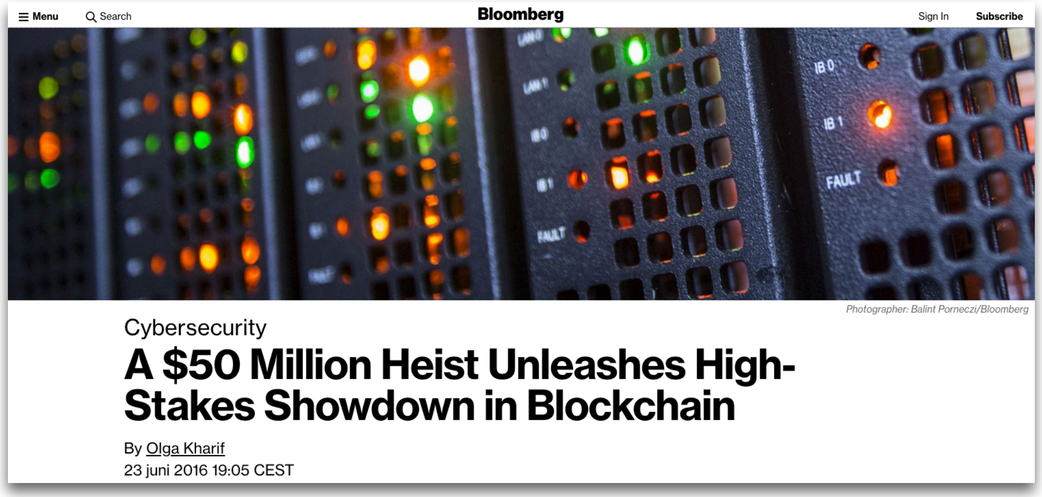
Crowdfunding contract: Solidity source code

```
contract Crowdfunding {

    address public owner;    // the beneficiary address
    uint256 public deadline; // campaign deadline in number of days
    uint256 public goal;     // funding goal in ether
    mapping (address => uint256) public backers; // the share of each backer

    constructor(uint256 numberOfDays, uint256 _goal) {
        owner = msg.sender;
        deadline = block.timestamp + (numberOfDays * 1 days);
        goal = _goal;

    }
    function donate() public payable {
        require(block.timestamp < deadline); // before the fundraising deadline
        backers[msg.sender] += msg.value;

    }

    function claimFunds() public {
        require(address(this).balance >= goal); // funding goal met
        require(block.timestamp >= deadline); // after the withdrawal period
        require(msg.sender == owner);
        payable(msg.sender).transfer(address(this).balance);
    }
    function getRefund() public {
        require(address(this).balance < goal); // campaign failed: goal not met
        require(block.timestamp >= deadline); // in the withdrawal period
        uint256 donation = backers[msg.sender];
        backers[msg.sender] = 0;
        payable(msg.sender).transfer(donation);
    }
}
```

Instructions to deposit and withdraw money (ether)

Writing correct smart contracts is a risky business

The DAO Hack (2016)



~$50 million stolen

cause: a “re-entrancy” bug: a function was called unexpectedly when the contract state was not yet properly updated

Parity freeze bug (2017)

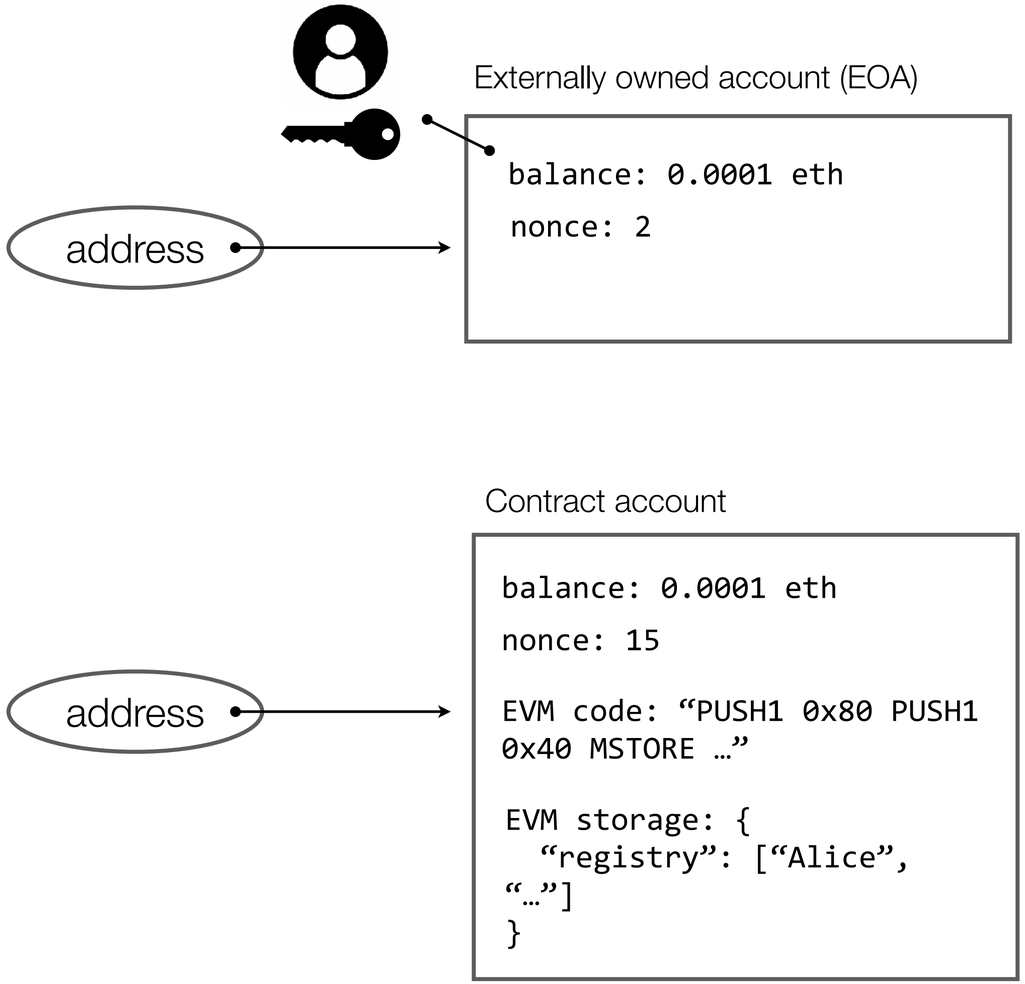
~$280 million accidentally frozen

cause: forgot to initialize a field in the constructor function

Ethereum: accounts, transactions and blocks

Ethereum accounts

- **Externally-owned** accounts (EOAs):
  - Associated with a public-private key pair
  - Account state = ether balance + nonce (see later)
  - Account address based on hash(public key)
- **Contract** accounts:
  - Not associated with a public-private key pair
  - Contract account = ether balance + nonce + storage + code
  - Account address based on hash(sender, nonce)

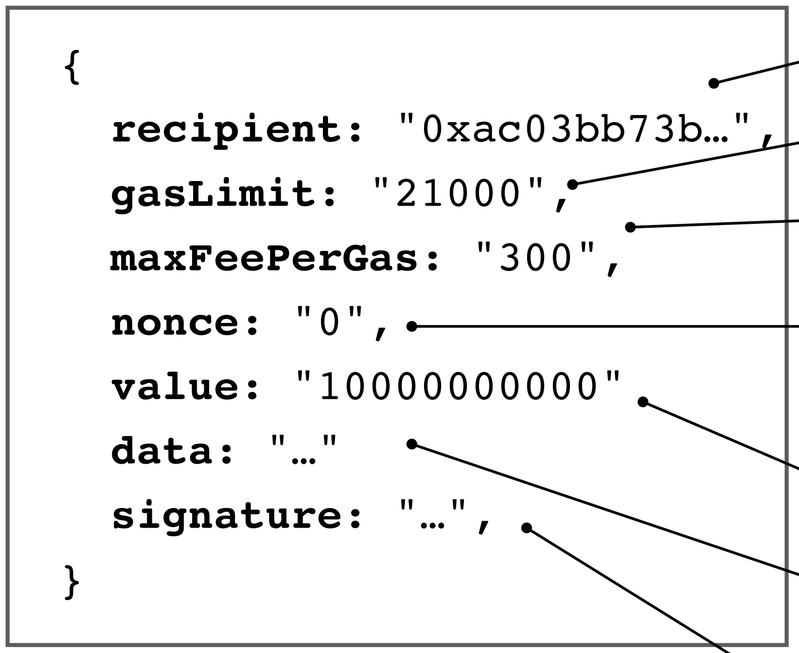


Ethereum transactions

- Ethereum supports three kinds of transactions:
  - 1. Simple **payment** transactions: a transfer of ether from one account to another (similar to Bitcoin and most other blockchain networks)
  - 2. Transactions that **deploy** new contract code to the blockchain. The transaction includes the **bytecode** to be stored in the new contract's account.
  - 3. Transactions that **call** functions on previously deployed smart contracts. A call transaction includes a function **selector** (to uniquely identify a function by name and argument types) and the function **arguments**.

Ethereum transactions

An Ethereum transaction



Recipient address (EOA or contract address). If absent, the transaction Represents a contract deployment.

Maximum gas cost the sender is willing to pay (in units of gas)

Maximum gas price the sender is willing to pay (in wei, 10^{-18} ether)

A sequentially incrementing counter *n* indicating this is the n^{th} transaction originating from this sender. If this *n* is not equal to the nonce stored in the sender's account + 1, the transaction is rejected. This prevents **replay attacks** where an attacker could otherwise **copy and resubmit** the sender's transaction to duplicate its effect.

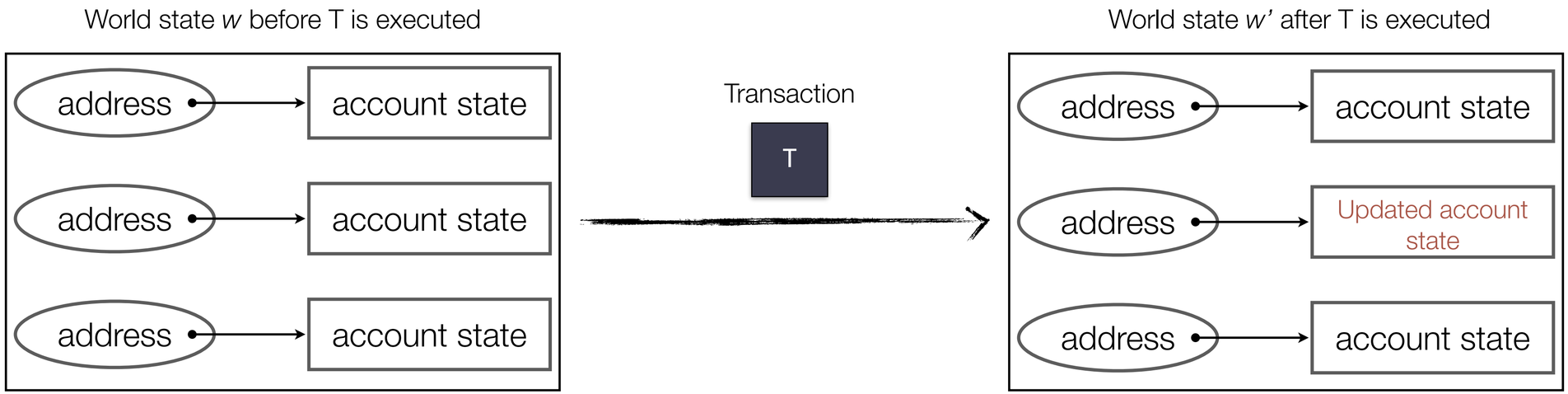
The amount of ether to be transferred to the recipient (in wei, 10^{-18} ether)

Optional data payload. For contract **deployment** transactions, contains the contract's bytecode. For contract function **calls**, contains the called function signature + function arguments ("calldata")

Digital signature of the sender. A valid signature confirms the sender has authorized the transaction.

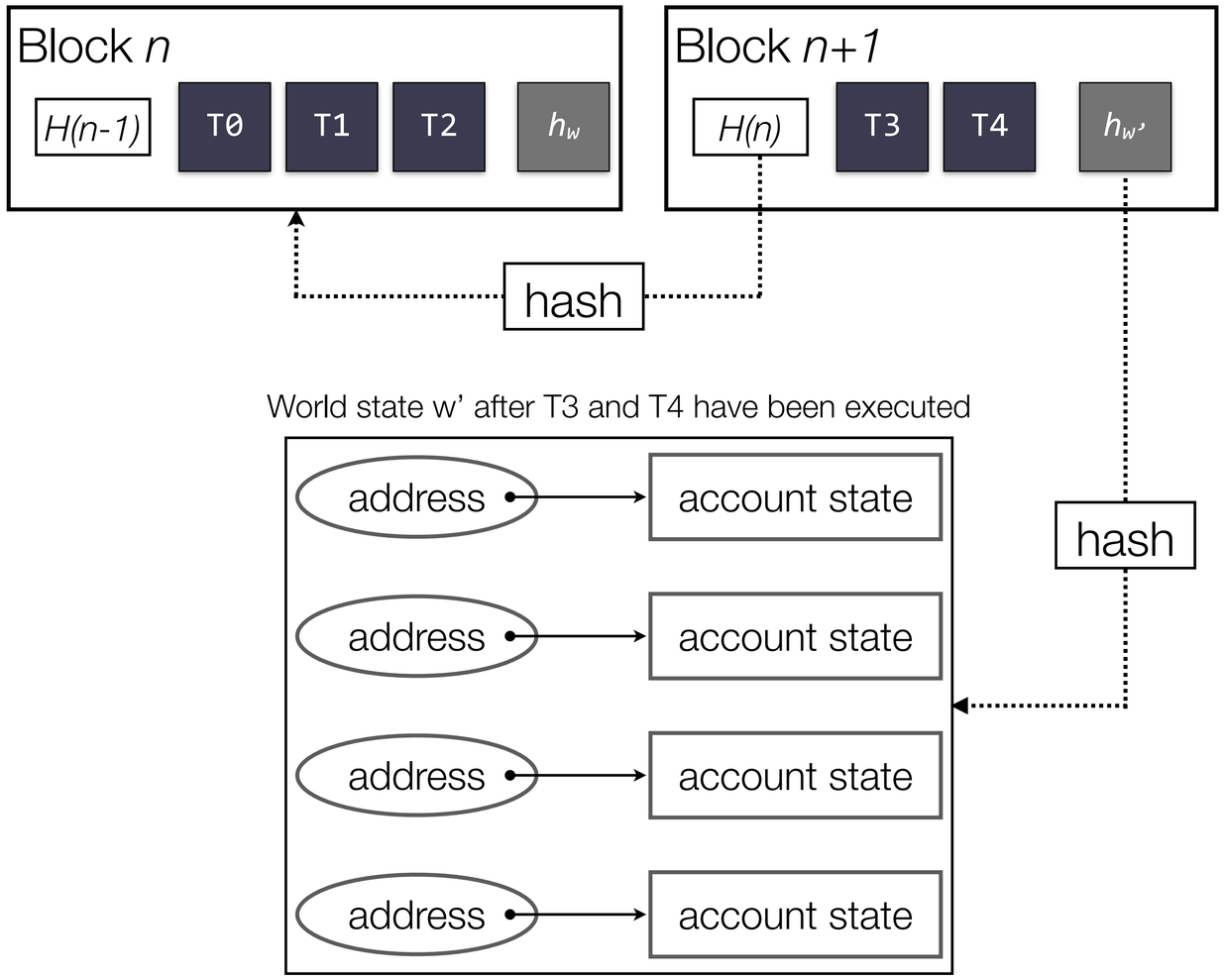
Ethereum transactions and “world state”

- The Ethereum “world state” is a mapping from account addresses to account state
- Every transaction T updates the current world state *w* to produce the next world state *w’*



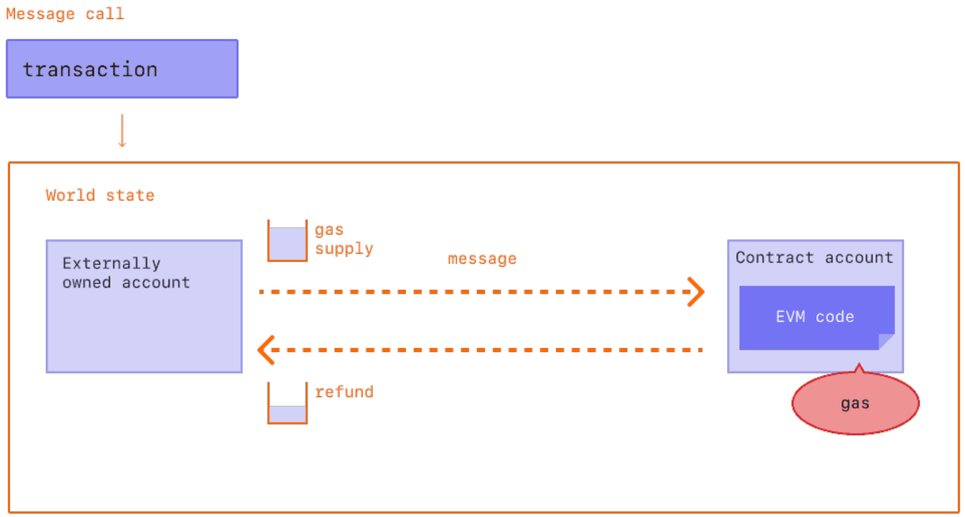
Ethereum blocks and hash pointers

- Each block contains a hash pointer to the entire updated world state
- The cryptographic hash represents a *commitment* to the world state (once the hash is published, the world state can no longer be changed without also visibly changing the hash)
- In practice, clever tree-structures ("merkle patricia trees") are used so that updates to the world state can be made efficiently, and the new hash can be computed incrementally from the previous world state *w* and the new world state *w'*



Ethereum gas fees

- Transaction fees in Ethereum are paid in a dedicated "unit of computation" called **gas**.
- The **gas price** is the price of 1 unit of gas in ether.
  - It is determined dynamically based on supply (available space in a block) and demand (transactions waiting to be processed).
  - The more the network becomes congested, the higher the gas price will be.
- The **gas fee** of a transaction (in ether) = total gas cost of the function call (in units of gas) x the gas price (in ether)



(Source: Ethereum documentation at https://ethereum.org/)

Ethereum gas fees

- The gas fee depends on the type and number of instructions executed by the contract.
- Each opcode in the EVM has a **gas cost.** Some opcodes have a **fixed** gas cost, others have **variable** gas cost. Basic arithmetic is cheap. Reading/writing to storage is expensive.
- The gas cost is computed in real-time as the EVM is executing each opcode in the function.
- A transaction must set a **maximum gas limit**. The gas limit ensures that a function call always has a **finite** execution time.
- If the function call “runs **out of gas**” (the gas cost exceeds the gas limit), then the transaction is aborted. The sender must still pay the gas fee even if the transaction is aborted! (Why?)
- Ethereum also sets a global maximum **gas limit** on each **block.** This puts an upper bound on the amount of transactions (and the amount of computation) to be processed per block.
- The gas limit **protects** the network **against denial of service** attacks.

Example EVM opcode gas costs

opcode	gas cost	description
ADD	3	Add two numbers
MUL	5	Multiply two numbers
AND	3	Bitwise AND operation
SLOAD	2100 or 100, depending on cache	Load word from storage
SSTORE	min 100 can be > 20.000	Save word to storage
JUMP	8	Update program counter
PUSH1	3	Place 1-byte item on stack
…	…	

(Source: https://evm.codes)

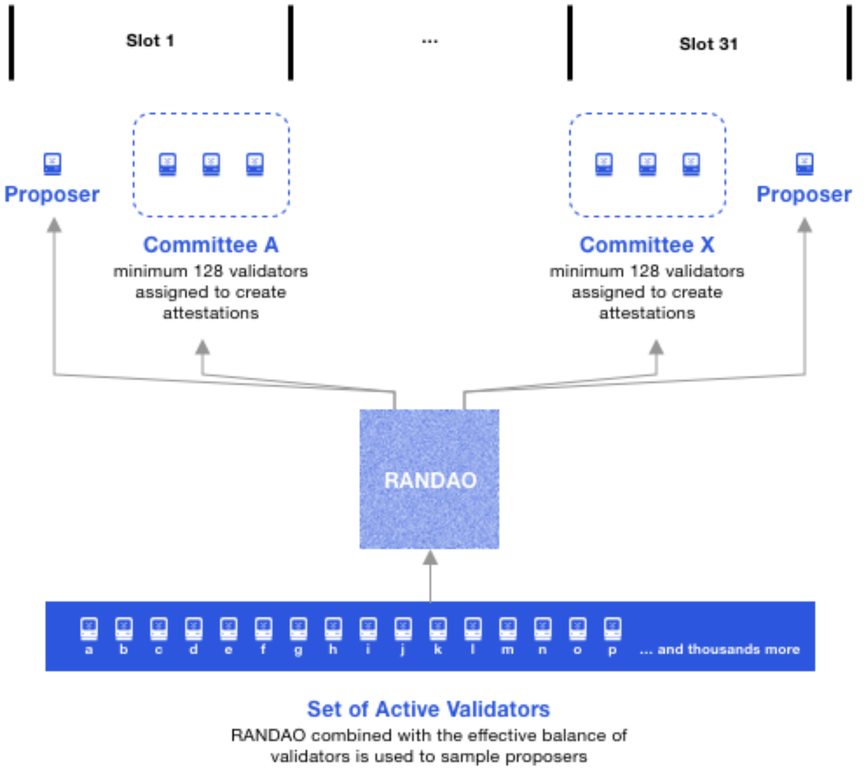
Ethereum: Proof-of-Stake consensus in a nutshell

- Ethereum uses Proof-of-Stake consensus (*): a validator's voting power is proportional to its **stake**
- Nodes have to **stake minimum 32 ether** into a deposit contract as collateral to become a validator.
- Some or all of a validator's stake can be destroyed if it is found to be dishonest. Nodes thus have **strong economic incentives** to remain honest.
- Validators follow a fixed protocol to take turns in proposing new blocks or attesting to the validity of new blocks.
- Ethereum works with fixed time slots. New blocks are created **every 12-15 seconds** on average (vs ~10 minutes in Bitcoin).
- The Ethereum network processes an average of ~12 transactions per second (data from late 2023).

(*) Before Sept. 6, 2022 Ethereum used Proof-of-Work consensus, but with a different PoW algorithm and parameters than Bitcoin.

Ethereum: Proof-of-Stake consensus in a nutshell

- In every time slot (every 12 seconds) a validator is **randomly selected** to be the block **proposer** and another group of nodes is randomly selected to form a **committee**.
- The chance to get elected as the block proposer is **proportional to a validator's staked funds**.
- The **randomness** needed to elect the next proposer is generated in a decentralized manner so that no single node can influence the result (using a mechanism known as "RANDAO" the details of which we will not discuss here).
- The block **proposer** bundles transactions, executes them and computes the new world state. They wrap this information into a block and broadcast it to the committee.
- When validators in the **committee** receive the block, they re-execute the transactions to ensure they reach the identical new world state. If they agree, they **attest** to the validity of the block by signing it. Signatures from a 2/3 majority of the committee are needed to mark a block as "final".
- If a validator sees two conflicting blocks for the same slot they pick the one supported by the **most staked ether**.



(Image credit: Consensys. Source: consensys.net, February 2020)

Summary

Ethereum case study: summary

- Ethereum is a **“programmable” blockchain**: create and call “smart contract” code stored on the blockchain
- **Decentralized applications** are Web applications where (part of) the back-end logic is implemented as a smart contract.
- **Smart contracts** are programs that can send and receive “money” (digital assets)
- Ethereum supports three kinds of **transactions**: payment transactions (transfer ether), deploy new smart contracts, or call a function on a previously deployed smart contract.
- Ethereum nodes keep track of a **“world state”**: a mapping from account addresses to account state. Transactions can update the state.
- Ethereum uses a lottery-based **“Proof-of-Stake”** consensus algorithm: validators are chosen to produce blocks proportional to their “stake” (an amount of ether tokens they must deposit in a contract)