Blockchains & Cryptocurrencies

Consensus & Towards Bitcoin

Instructor: Matthew Green & Abhishek Jain
Johns Hopkins University - Spring 2023

Many slides based on NBFMG

Housekeeping

- Readings, NBFMG
- Please keep up with the readings!
- A1 will be out today (after class)
  - On the course syllabus and a note in Piazza
  - **https://piazza.com/class/ldacuez4oqgrt**

News?

If you were a *market maker* on FTX, though, you *were* allowed to have a negative balance: Effectively, FTX would lend you the money so you could open a position without depositing the money first, or have the market move against you without instant liquidation. In FTX's code, most accounts had a "borrow" flag set to zero, meaning that they could not have negative balances, but some 4,000 accounts had the borrow flag set to some positive number, meaning that FTX would lend them the money up to some credit limit. Of those 4,000 accounts, 41 had credit limits of $1 million to $150 million. One — Alameda — had a higher limit. Alameda's limit was $65 billion. (Slide 18 shows a code snippet, showing that the actual limit was $65,355,999,994.) "FTX will allow Alameda to have a negative balance of up to $65 billion" is functionally equivalent to "Alameda can use as much of FTX's customer money as it wants."

Alameda — had a higher limit.
ameda's limit was $65 billion. (Slide 18
ows a code snippet, showing that the
tual limit was $65,355,999,994.) "FTX will
ow Alameda to have a negative balance
up to $65 billion" is functionally
uivalent to "Alameda can use as much of
X's customer money as it wants."

News?

```
'id': {0: 5, 1: 9, 2: 149158294},
'username': {0: 'ETH1NX', 1: 'info@alameda-research.com', 2: 'info@alameda-research.com/f
'min_imf': {0: Decimal('0.05000000'), 1: Decimal('0.05000000'), 2: Decimal('0.10000000')}
'maker_fee': {0: Decimal('0.00010000'), 1: Decimal('-0.00010000'), 2: Decimal('0.00018000
'taker_fee': {0: Decimal('0.00030000'), 1: Decimal('0.00015000'), 2: Decimal('0.00063000'
'liquidating': {0: False, 1: False, 2: False},
'backstop_provider': {0: False, 1: True, 2: False},
'borrow': {0: Decimal('0E-8'), 1: Decimal('65355999994.00000000'), 2: Decimal('0E-8')},
'can_withdraw_below_borrow': {0: False, 1: True, 2: False},
'allow_negative': {0: False, 1: True, 2: True},
'can_trade_futures': {0: True, 1: True, 2: True},
'use_ftt_collateral': {0: True, 1: True, 2: True},
'ignore_imf_factors': {0: True, 1: False, 2: False},
'fee_voucher': {0: Decimal('0E-8'), 1: Decimal('0E-8'), 2: Decimal('0E-8')},
'charge_interest_on_negative_usd': {0: False, 1: False, 2: False},
'spot_margin_enabled': {0: False, 1: False, 2: False},
'spot_margin_lending_enabled': {0: False, 1: True, 2: False},
'account_type': {0: None, 1: None, 2: None},
```

News?

`decimal` — Decimal fixed point and floating point arithmetic

Source code: Lib/decimal.py

The `decimal` module provides support for fast correctly rounded decimal floating point arithmetic. It offers several advantages over the `float` datatype:

- Decimal "is based on a floating-point model which was designed with people in mind, and necessarily has a paramount guiding principle – computers must provide an arithmetic that works in the same way as the arithmetic that people learn at school." – excerpt from the decimal arithmetic specification.

- Decimal numbers can be represented exactly. In contrast, numbers like `1.1` and `2.2` do not have exact representations in binary floating point. End users typically would not expect `1.1 + 2.2` to display as `3.3000000000000003` as it does with binary floating point.

Today

- We're going to talk about "consensus"
- What the heck is consensus, how do you accomplish it, what's the point?
- Then we'll evolve towards Bitcoin

Review: hashes / signatures

Reminder: hash functions

- Take as input an arbitrary-length string
- Output a (shorter) fixed-size string

Cryptographic hash function security:

Reminder: hash functions

- Take as input an arbitrary-length string
- Output a (shorter) fixed-size string

Cryptographic hash function security:

- Collision-resistant
- Pre-image resistant
- “Random oracle”-like (for some cases)

Reminder: hash functions

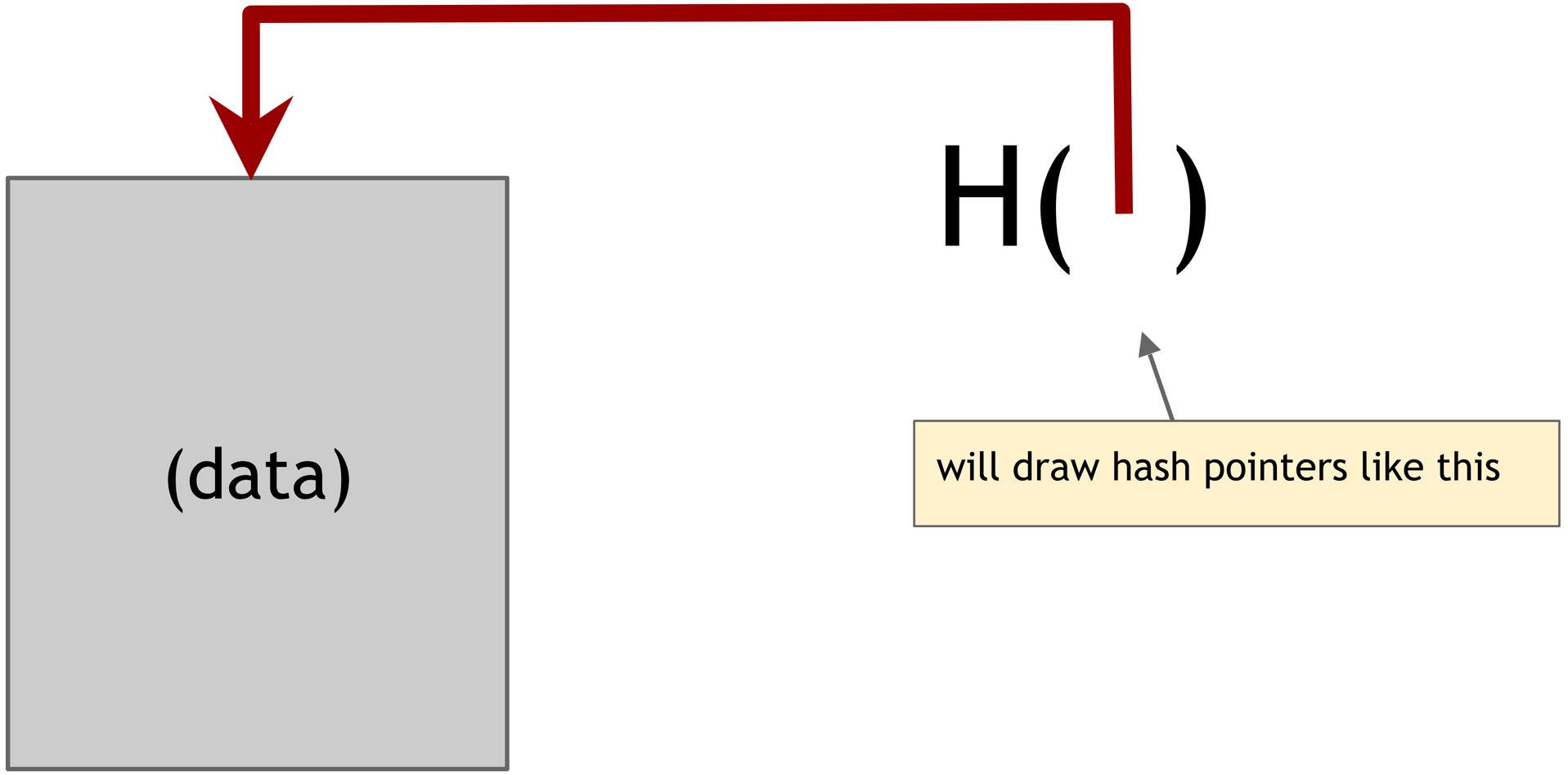
- Some examples (current+historical):
  - MD5 ❌
  - SHA1 ❌
  - SHA2 family ✅
    SHA256, SHA512, SHA384 (also SHA224 😒)
  - SHA3 (AKA “Keccak”) ✅
  - Blake2 ✅

Hash pointer

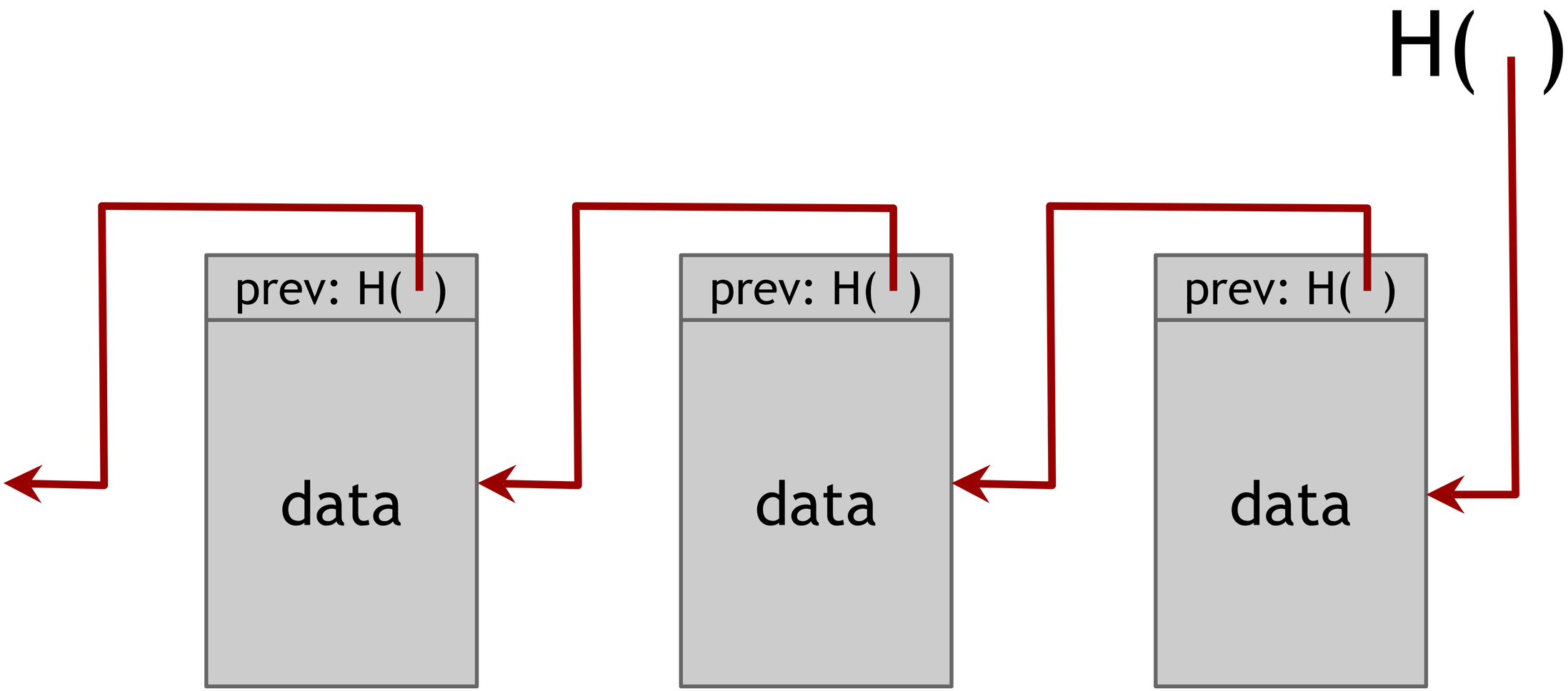
- pointer to where some info is stored, *and*
- cryptographic hash of the info

If we have a hash pointer, we can

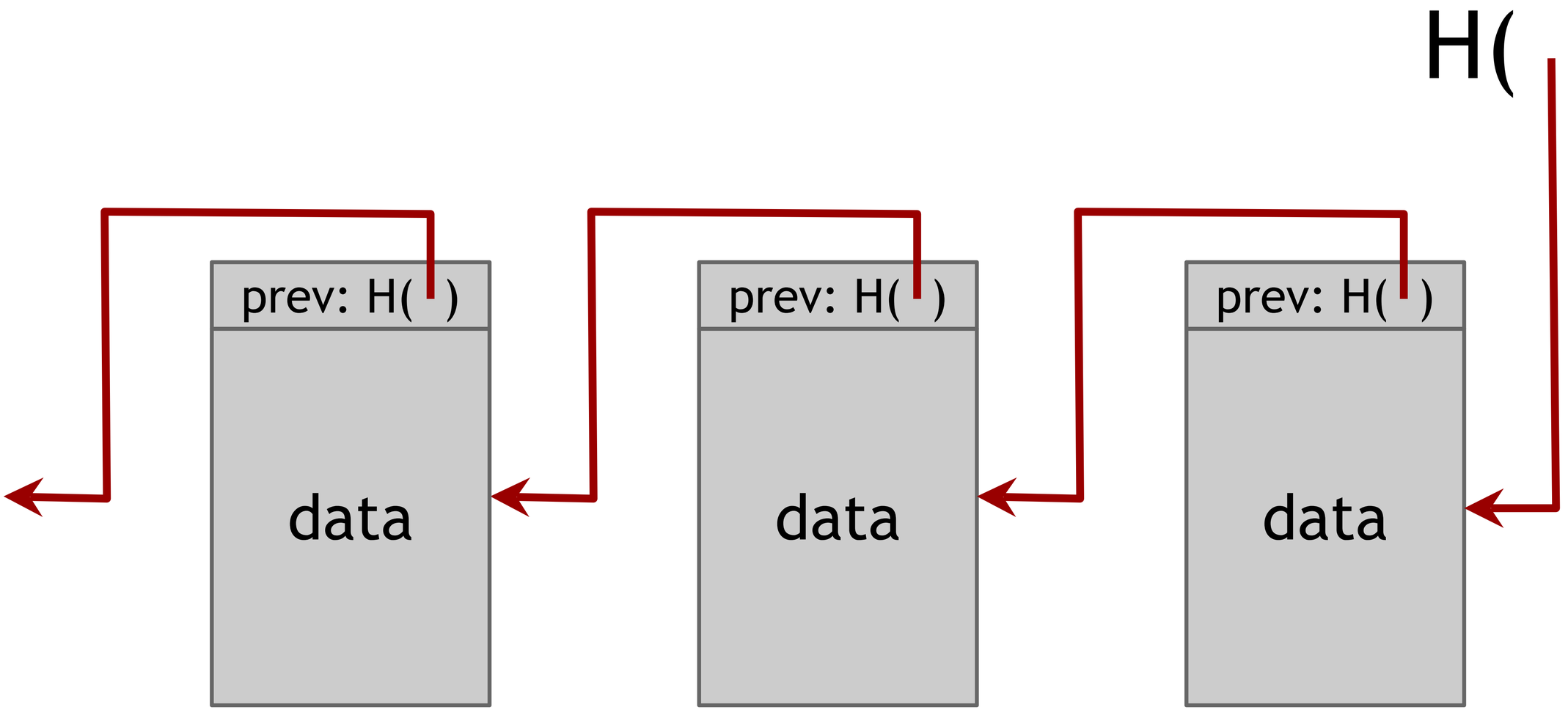
- ask to get the info back, and
- verify that it hasn’t changed


H( )
(data)
will draw hash pointers like this

Building data structures with hash pointers

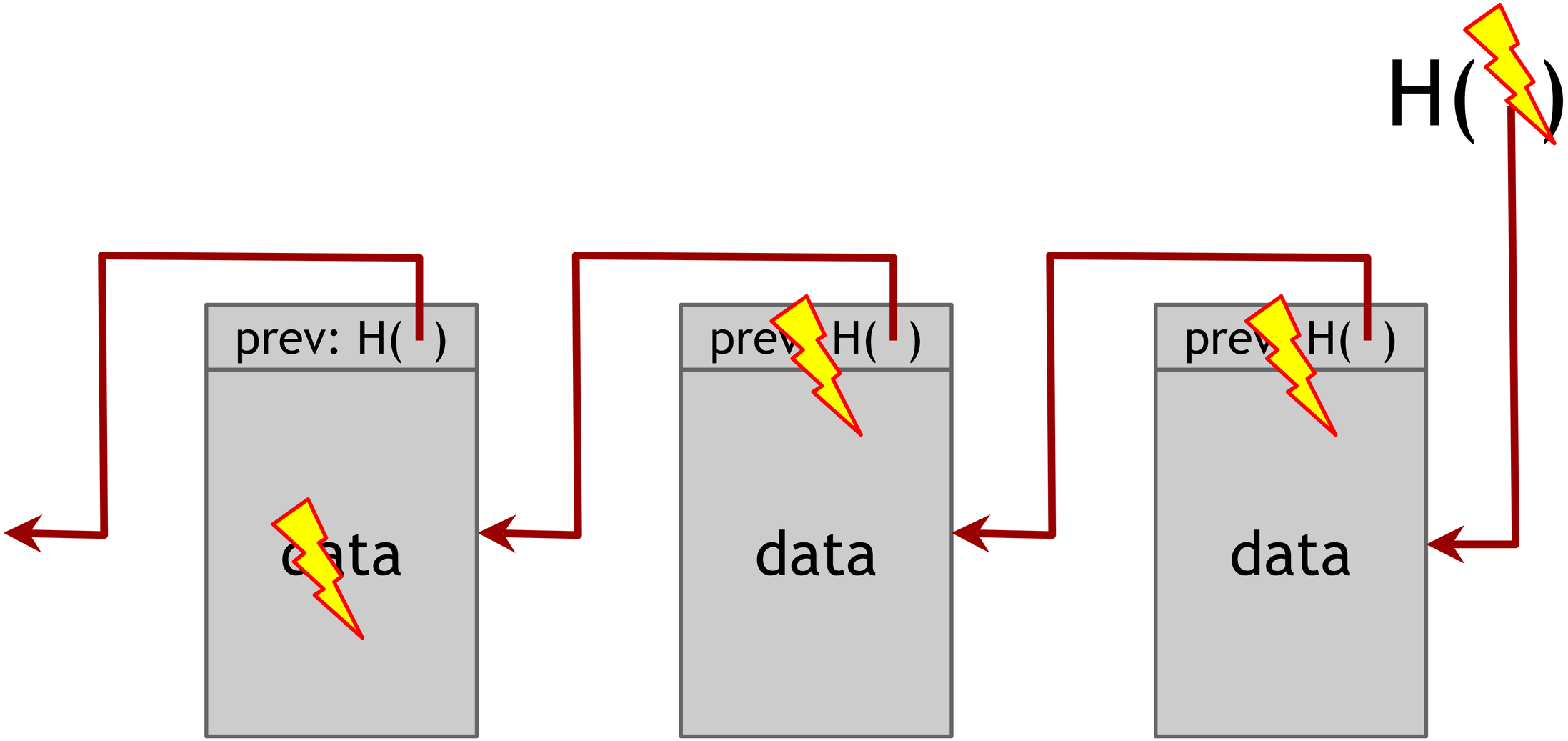

H( )
prev: H( )
data
prev: H( )
data
prev: H( )
data

Linked list with hash pointers = “Blockchain”



use case: tamper-evident log

detecting tampering



use case: tamper-evident log

Digital signatures

- $(sk, pk) \leftarrow keygen(1^k)$

  sk: secret signing key

  pk: public verification key

Security parameter

randomized algorithm

- $sig \leftarrow sign(sk, message)$

Typically randomized

- $isValid \leftarrow verify(pk, message, sig)$

Requirements for signatures

- Correctness: “valid signatures verify”
  - verify(pk, message, sign(sk, message)) == true
- Unforgeability under chosen-message attacks (UF-CMA): “can’t forge signatures”
  - adversary who knows pk, and gets to see signatures on messages of his choice, can’t produce a verifiable signature on another message

Review: cash problems

- **Double spending**
  - To capture double spending you need an online (networked) party that must be trusted
- **Authentication / Authentication**
  - How do I prove that I am the owner of currency & thus authorized to transact with it?
- **Origin/Issuance**
  - How is new currency created?

Partial approach

- Let's not dispense with our centralized approach (just yet)
  - We will, however, reduce the number of our assumptions
  - Our new assumption is that there is a **centralized** party that can maintain a ledger
  - This centralized party also can create ("mint") new currency and assign it to be owned by users

GoofyCoin

Goofy can create new coins

signed by pk_{Goofy}

CreateCoin [uniqueCoinID]

A coin’s owner can spend it.

signed by pk_{Goofy}

Pay to pk_{Alice} H( )

signed by pk_{Goofy}

CreateCoin [uniqueCoinID]

The recipient can pass on the coin again.

signed by pk_{Alice}

Pay to pk_{Bob} : H( )

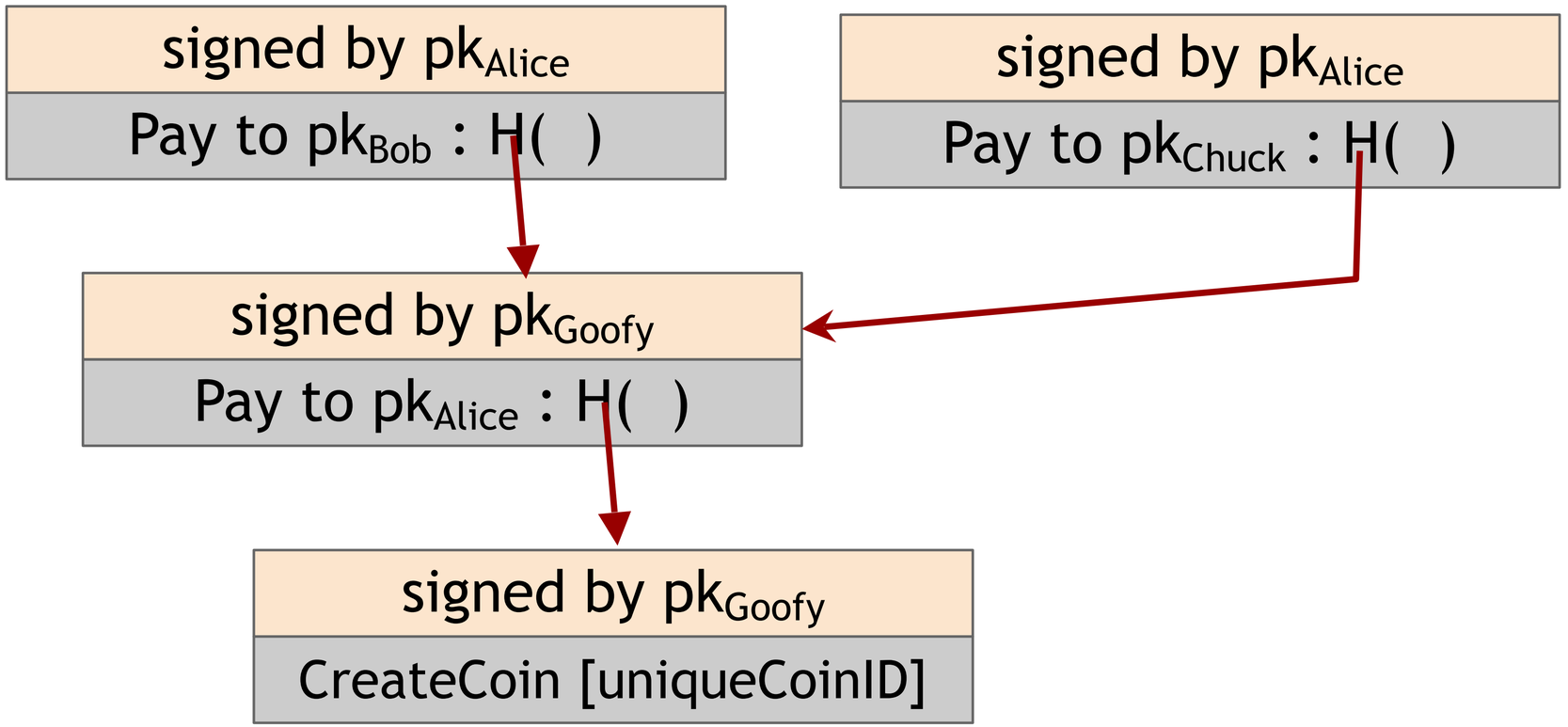
signed by pk_{Goofy}

Pay to pk_{Alice} : H( )

signed by pk_{Goofy}

CreateCoin [uniqueCoinID]

double-spending attack



double-spending attack

This is the main design challenge in digital currency

How do we solve this?

How do we solve this?

- Simplest answer: send all transactions to an atomic, append-only **centralized** ledger
- Have the ledger provide a definite ordering for transactions
  - If two transactions conflict, simply disallow the later one
- No TX is valid unless the ledger has "approved" and ordered it

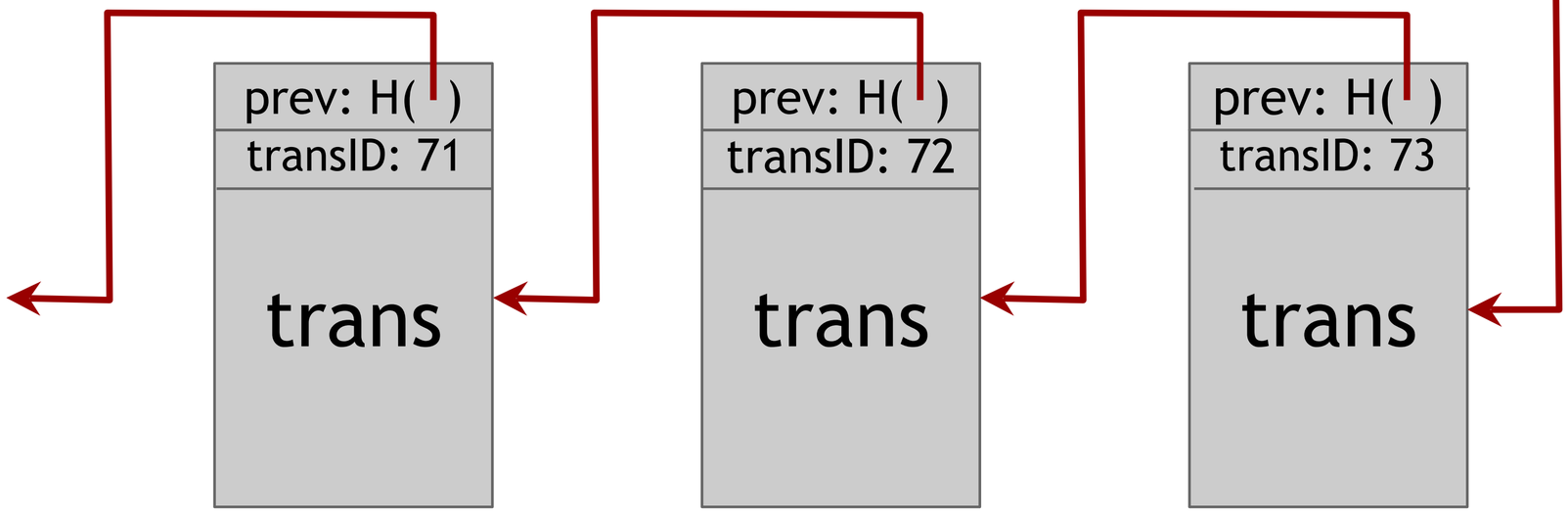
ScroogeCoin

Scrooge publishes a history of all transactions in an “append-only” ledger

H( )

Sig

Implement the ledger using a block chain, signed by Scrooge



optimization: put multiple transactions in the same block

CreateCoins transaction creates new coins

Valid, because I said so.

transID: 73 type:CreateCoins		
coins created		
num	*value*	*recipient*
0	3.2	0x...
1	1.4	0x...
2	7.1	0x...

coinID 73(0)

coinID 73(1)

coinID 73(2)

signature

CreateCoins transaction creates new coins

Valid, because I said so.

transID: 73 type:CreateCoins		
coins created		
num	*value*	*recipient*
0	3.2	0x...
1	1.4	0x...
2	7.1	0x...

These are public keys!

coinID 73(0)

coinID 73(1)

coinID 73(2)

signature

PayCoins transaction consumes (and destroys) some coins,
and creates new coins of the same total value

transID: 73 type:PayCoins

consumed coinIDs:
68(1), 42(0), 72(3)

coins created

num	*value*	*recipient*
0	3.2	0x...
1	1.4	0x...
2	7.1	0x...

signatures

Valid if:
-- consumed coins valid,
-- not already consumed,
-- total value out = total value in, and
-- signed by owners of all consumed coins

One signature for each consumed coin

Immutable coins

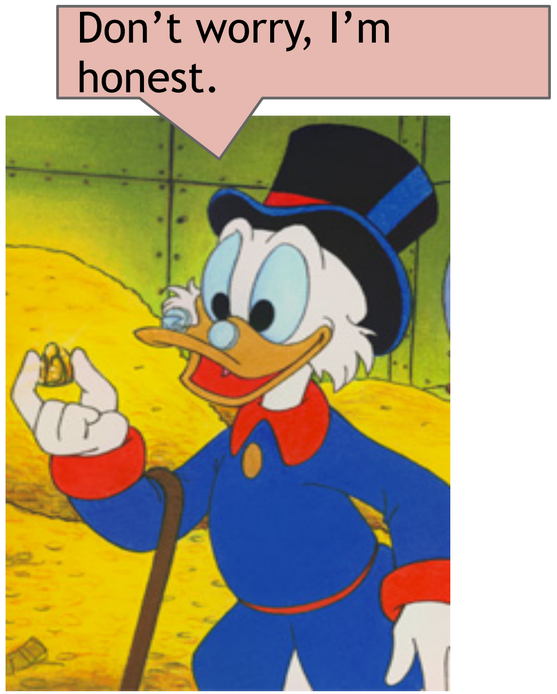
Coins can’t be transferred, subdivided, or combined.

But: you can get the same effect by using transactions

to subdivide: create new transaction

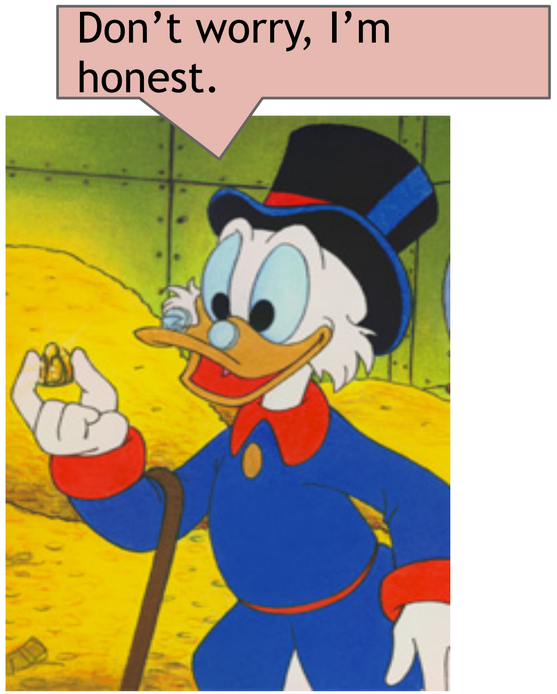
consume your coin

pay out two new coins to yourself



Crucial question:

Can we descroogify the currency, and operate without any central, trusted party?



Crucial question:

Can we descroogify the currency, and operate without any central, trusted party?

Related question:

Why do we <u>need</u> to do this?

Centralization vs. Decentralization

- **Competing paradigms that underlie many technologies**

- Decentralized != Distributed
  (as in distributed system) but we'll often use them as synonyms

Centralization vs. Decentralization

- Examples:
  - email?
  - WWW?
  - DNS?
- What about software development?

Aspects of decentralization in Bitcoin

1. Who maintains the ledger?
2. Who has authority over which transactions are valid?
3. Who creates (and obtains) new bitcoins?
4. Who determines how the rules change?
5. How do these coins acquire monetary value?

Aspects of decentralization in Bitcoin

Peer-to-peer network:

open to anyone, low barrier to entry

high node churn (nodes can come and go)

Mining:

open to anyone, but inevitable concentration of power

often seen as undesirable

Updates to software:

core developers trusted by community, have great power

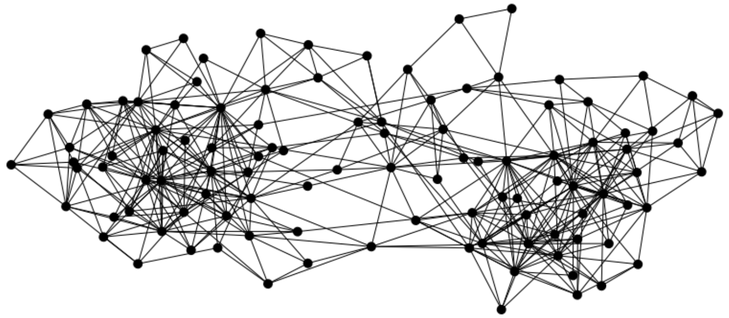
Distributed consensus

Bitcoin is a peer-to-peer system

When Alice wants to pay Bob:
she broadcasts the transaction to all Bitcoin nodes

signed by Alice

Pay to pk_{Bob} : H( )



Note: Bob's computer is not in the picture

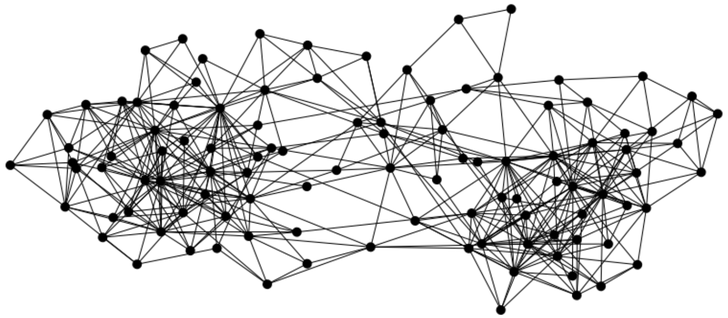
Bitcoin is a peer-to-peer system

This network is a fill/flood style P2P network: all nodes perform basic validation, then relay to their peers

This introduces bootstrapping, spam and DoS problems, which are dealt with through “seeders” and “reputation” scores

signed by Alice

Pay to pk_{Bob} : H( )

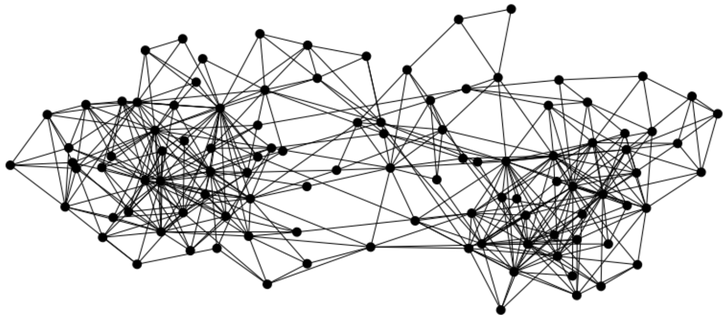


Why aren’t we done here?

Why can’t we just trust this system to eliminate invalid blocks, and give everyone a robust view of the Tx history?

signed by Alice

Pay to pk_{Bob} : H( )



Bitcoin’s key challenge

Key technical challenge of decentralized e-cash: distributed consensus

or: how do all of these nodes agree on an ordered history of transactions?

Defining distributed consensus

The protocol terminates and all honest nodes decide on the same **value**

This value must have been proposed by some honest node

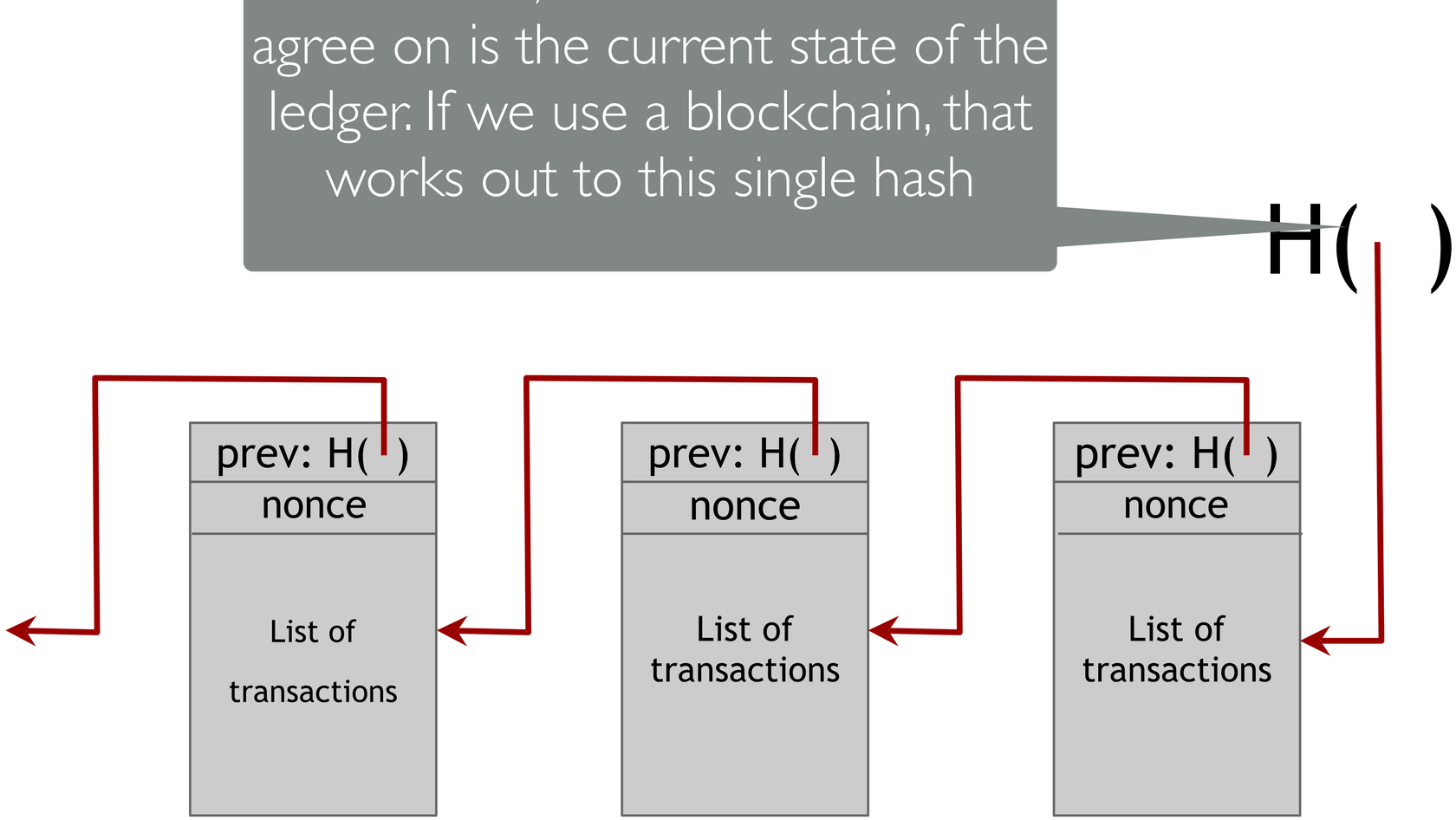
Defining distributed consensus

Q: What is this "value" in Bitcoin?

The protocol terminates and all honest nodes decide on the same **value**

This value must have been proposed by some honest node

A: In Bitcoin, the value we want to agree on is the current state of the ledger. If we use a blockchain, that works out to this single hash

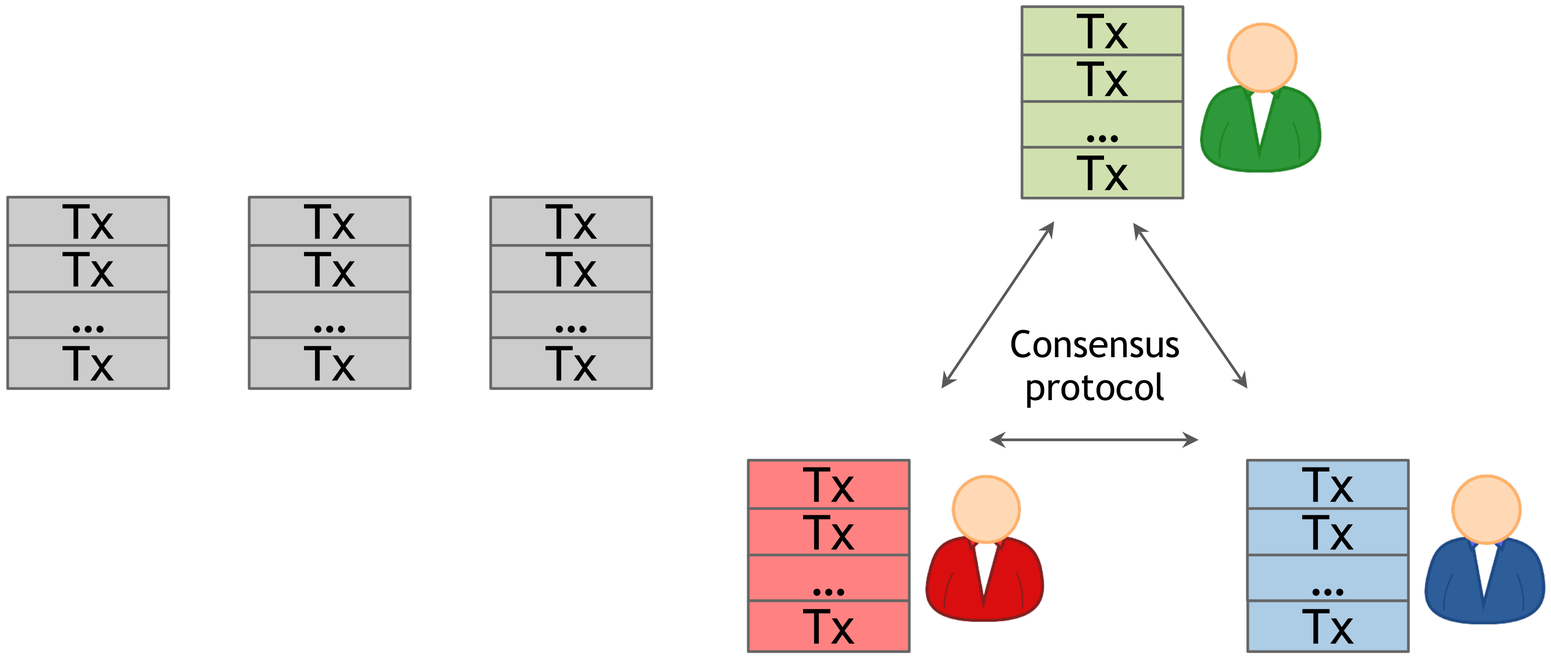


How consensus could work in Bitcoin

At any given time:

- All nodes have a sequence of blocks of transactions they've reached consensus on
- (Blocks are also distributed via p2p network)
- Each node has a set of outstanding transactions it's heard about

How consensus could work in Bitcoin



OK to select any valid block, even if proposed by only one node

Why consensus is hard

Nodes may crash
Nodes may be malicious

Network is imperfect

- Not all pairs of nodes connected
- Faults in network (“partitioning”)
- Latency

No notion of global time