Hash Functions, Proofs of Work, Blockchains and Decentralization

CMSC 23280/ECON 23040, Autumn 2018
Lecture 2

David Cash, Harald Uhlig, Ben Zhao

University of Chicago

Class logistics

1. Website for weeks 1-5:
   https://people.cs.uchicago.edu/~davidcash/23280-winter-19/
   (Or: https://david.cash forwards to my page, then click link)

2. Get on Piazza if you were not added automatically

3. Request access to Canvas if you were not added automatically

4. Assignment 1 will be out tonight and due in one week

Lecture 2 Outline

1. Cryptographic Hash Functions
   - Blockchains
   - Proofs of Work
2. Putting DCash “on the blockchain”, with an authority
3. The idea of decentralization
4. Decentralized DCash with an Angel
5. Decentralized DCash via proofs-of-work

Lecture 2 Outline

1. Cryptographic Hash Functions

- Blockchains
- Proofs of Work

2. Putting DCash “on the blockchain”, with an authority

3. The idea of decentralization

4. Decentralized DCash with an Angel

5. Decentralized DCash via proofs-of-work

Hash Functions

Hash functions are frequently used in computer science to store and look-up data.

Hash Functions

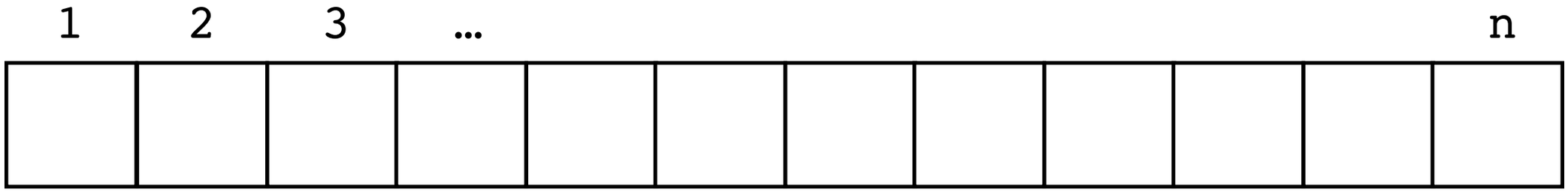
Hash functions are frequently used in computer science to store and look-up data.

Example: Hash table data structure.

Hash Functions

Hash functions are frequently used in computer science to store and look-up data.

Example: Hash table data structure.

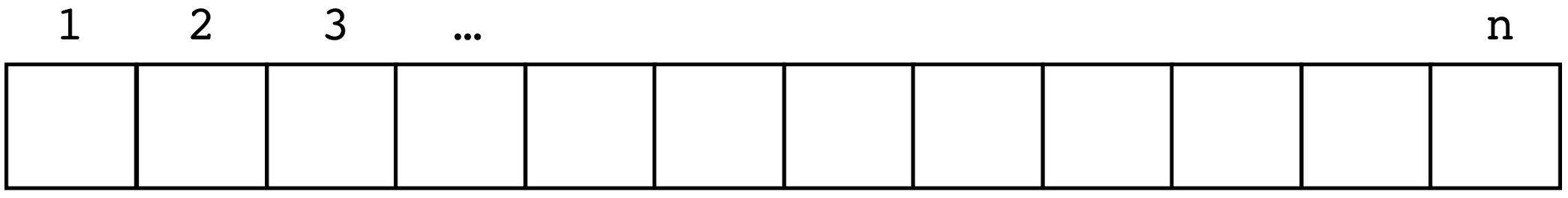


Hash Functions

Hash functions are frequently used in computer science to store and look-up data.

Example: Hash table data structure.

- Use a function `h` that outputs numbers between `1` and `n`
- `n` is number of cells in table, which may be large but not astronomical
- Store `x` at position `h(x)`



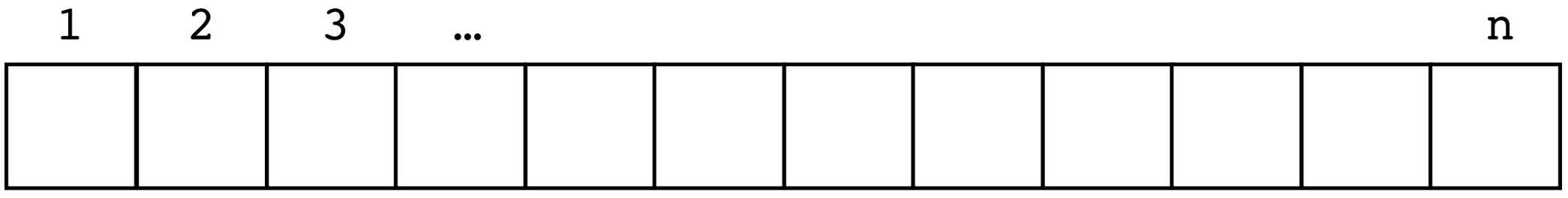
Hash Functions

Hash functions are frequently used in computer science to store and look-up data.

Example: Hash table data structure.

- Use a function `h` that outputs numbers between `1` and `n`
- `n` is number of cells in table, which may be large but not astronomical
- Store `x` at position `h(x)`

`h(x)=2` ⇒ Store x in position 2



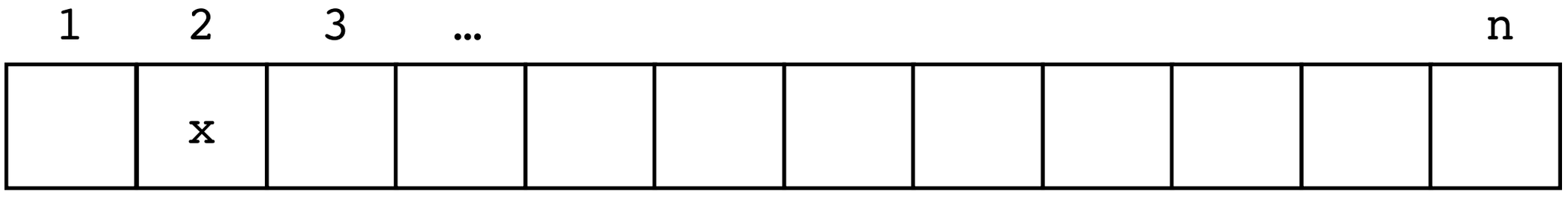
Hash Functions

Hash functions are frequently used in computer science to store and look-up data.

Example: Hash table data structure.

- Use a function `h` that outputs numbers between `1` and `n`
- `n` is number of cells in table, which may be large but not astronomical
- Store `x` at position `h(x)`

`h(x)=2` ⇒ Store x in position 2



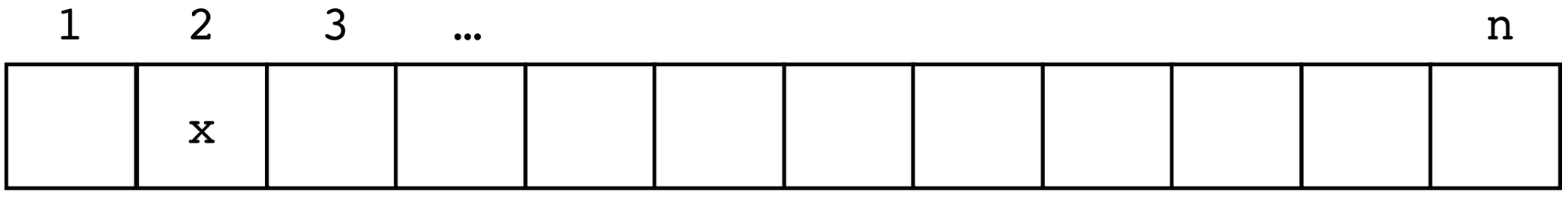
Hash Functions

Hash functions are frequently used in computer science to store and look-up data.

Example: Hash table data structure.

- Use a function `h` that outputs numbers between `1` and `n`
- `n` is number of cells in table, which may be large but not astronomical
- Store `x` at position `h(x)`

`h(x)=2` ⇒ Store x in position 2



- Simple hash functions work, like `h(x)=x mod n` (or slightly more complicated)
- Collisions happen: `x≠x'` but `h(x)=h(x')`, and must be handled by table.

Cryptographic Hash Functions

A **cryptographic hash function** is a function `H` that maps arbitrary strings to fixed-length outputs, *and should also satisfy several security properties*.

Cryptographic Hash Functions

A **cryptographic hash function** is a function `H` that maps arbitrary strings to fixed-length outputs, *and should also satisfy several security properties*.

Hash function `h`

Cryptographic Hash Functions

A **cryptographic hash function** is a function `H` that maps arbitrary strings to fixed-length outputs, *and should also satisfy several security properties*.

Hash function `h`

Cryptographic hash function `H`

Cryptographic Hash Functions

A **cryptographic hash function** is a function `H` that maps arbitrary strings to fixed-length outputs, *and should also satisfy several security properties*.

Cryptographic Hash Functions

A **cryptographic hash function** is a function `H` that maps arbitrary strings to fixed-length outputs, *and should also satisfy several security properties*.

In math notation: $H:\{0,1\}^* \rightarrow \{0,1\}^L$. L is the *output length*.

Cryptographic Hash Functions

A **cryptographic hash function** is a function `H` that maps arbitrary strings to fixed-length outputs, *and should also satisfy several security properties*.

In math notation: $H:\{0,1\}^* \rightarrow \{0,1\}^L$. `L` is the *output length*.

Compared to traditional hash functions, cryptographic hash functions:

Cryptographic Hash Functions

A **cryptographic hash function** is a function `H` that maps arbitrary strings to fixed-length outputs, *and should also satisfy several security properties*.

In math notation: $H:\{0,1\}^* \rightarrow \{0,1\}^L$. `L` is the *output length*.

Compared to traditional hash functions, cryptographic hash functions:

- … have an *astronomical* number of possible outputs
  - Typically `L=256` bits (`=32` bytes) so number of outputs is 2^{256}
  - $2^{256} \approx$ number of atoms in universe!

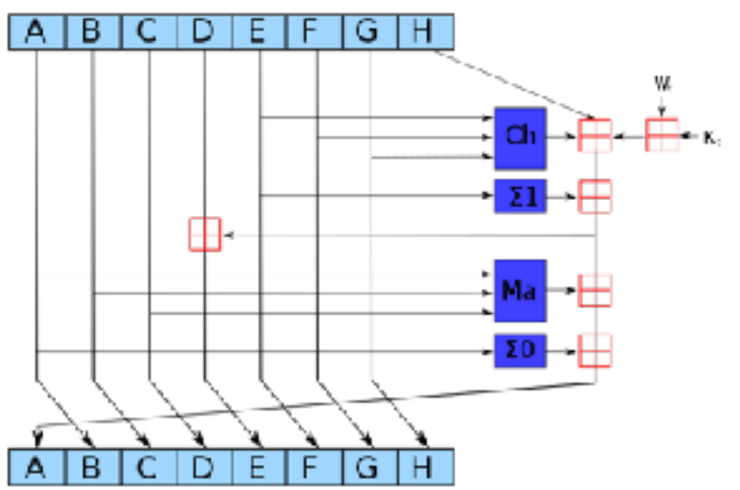
Cryptographic Hash Functions

A **cryptographic hash function** is a function `H` that maps arbitrary strings to fixed-length outputs, *and should also satisfy several security properties*.

In math notation: $H:\{0,1\}^* \rightarrow \{0,1\}^L$. `L` is the *output length*.

Compared to traditional hash functions, cryptographic hash functions:

- … have an *astronomical* number of possible outputs
  - Typically `L=256` bits (`=32` bytes) so number of outputs is 2^{256}
  - $2^{256} \approx$ number of atoms in universe!
- … are relatively complicated (but still fast on a human scale)



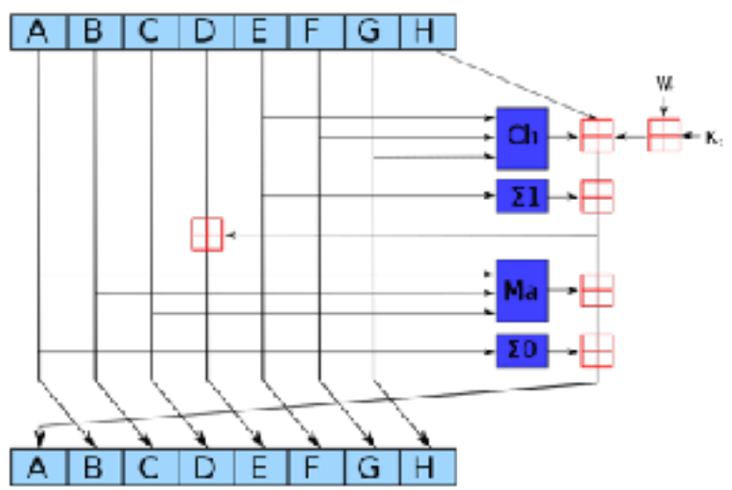
Cryptographic Hash Functions

A **cryptographic hash function** is a function `H` that maps arbitrary strings to fixed-length outputs, *and should also satisfy several security properties*.

In math notation: $H:\{0,1\}^* \rightarrow \{0,1\}^L$. `L` is the *output length*.

Compared to traditional hash functions, cryptographic hash functions:

- … have an *astronomical* number of possible outputs
  - Typically `L=256` bits (`=32` bytes) so number of outputs is 2^{256}
  - $2^{256} \approx$ number of atoms in universe!
- … are relatively complicated (but still fast on a human scale)
- … are much more resilient to adversarial inputs



Aside: Huge, Astronomical, and Depressingly Large

# Steps	Who can do that many?
2^{30}	Your laptop (one day)
2^{56}	Strong computer with GPUs
2^{80}	All computers on Bitcoin network in a few days
2^{128}	US Gov in ??? years, or very large quantum computer*
2^{256}	Nobody ever?
2^{1024}	Nobody ever?

*Not directly comparable but this is an estimate of equivalent power. Quantum computers are most effective against public-key crypto like digital signatures, but they also speed up attacks on hash functions.

Security of Cryptographic Hash Functions

A cryptographic hash function `H` is **collision-resistant** if no feasible attack can find inputs `x`, `x'` such that `x≠x'` but `H(x)=H(x')`.

Security of Cryptographic Hash Functions

A cryptographic hash function `H` is **collision-resistant** if no feasible attack can find inputs `x`, `x'` such that `x≠x'` but `H(x)=H(x')`.

- Collisions always exist when (number of inputs) > (number of outputs) by pigeonhole principle. It should be hard to find them though.

Security of Cryptographic Hash Functions

A cryptographic hash function `H` is **collision-resistant** if no feasible attack can find inputs `x`, `x'` such that `x≠x'` but `H(x)=H(x')`.

- Collisions always exist when (number of inputs) > (number of outputs) by pigeonhole principle. It should be hard to find them though.

If `H` is collision resistant, then `H(x)` is *practically* unique. Other inputs will collide, but if no one can find them, then they don't matter.

Security of Cryptographic Hash Functions

A cryptographic hash function `H` is **collision-resistant** if no feasible attack can find inputs `x`, `x'` such that `x≠x'` but `H(x)=H(x')`.

- Collisions always exist when (number of inputs) > (number of outputs) by pigeonhole principle. It should be hard to find them though.

If `H` is collision resistant, then `H(x)` is *practically* unique. Other inputs will collide, but if no one can find them, then they don't matter.

- This means `H` is *committing*: Suppose you predict the closing prices of the S&P 500 tomorrow, and let `x` be your predictions. You publish `y=H(x)`.

Security of Cryptographic Hash Functions

A cryptographic hash function `H` is **collision-resistant** if no feasible attack can find inputs `x`, `x'` such that `x≠x'` but `H(x)=H(x')`.

- Collisions always exist when (number of inputs) > (number of outputs) by pigeonhole principle. It should be hard to find them though.

If `H` is collision resistant, then `H(x)` is *practically* unique. Other inputs will collide, but if no one can find them, then they don't matter.

- This means `H` is *committing*: Suppose you predict the closing prices of the S&P 500 tomorrow, and let `x` be your predictions. You publish `y=H(x)`.
- After the market closes, you can only reveal the `x` you chose. Finding another `x` to change your prediction amounts to finding a collision.

Real Cryptographic Hash Functions Get Broken

Most crypto (encryption, digital signatures, etc) only gets broken very rarely.
But hash functions are harder to build and *do* get broken.

Real Cryptographic Hash Functions Get Broken

Most crypto (encryption, digital signatures, etc) only gets broken very rarely.
But hash functions are harder to build and *do* get broken.

- MD5 (1992) was broken in 2004 - can now find collisions very quickly.

Real Cryptographic Hash Functions Get Broken

Most crypto (encryption, digital signatures, etc) only gets broken very rarely.
But hash functions are harder to build and *do* get broken.

- MD5 (1992) was broken in 2004 - can now find collisions very quickly.

Xiaoyun Wang (Tsinghua University), 2004

Real Cryptographic Hash Functions Get Broken

Most crypto (encryption, digital signatures, etc) only gets broken very rarely.
But hash functions are harder to build and *do* get broken.

- MD5 (1992) was broken in 2004 - can now find collisions very quickly.

MD5(`d131dd02c5e6eec4693d9a0698aff95c 2fcab58712467eab4004583eb8fb7f89`
`55ad340609f4b30283e488832571415a 085125e8f7cdc99fd91dbdf280373c5b`
`d8823e3156348f5bae6dacd436c919c6 dd53e2b487da03fd02396306d248cda0`
`e99f33420f577ee8ce54b67080a80d1e c69821bcb6a8839396f9652b6ff72a70`)

= MD5(`d131dd02c5e6eec4693d9a0698aff95c 2fcab50712467eab4004583eb8fb7f89`
`55ad340609f4b30283e4888325f1415a 085125e8f7cdc99fd91dbd7280373c5b`
`d8823e3156348f5bae6dacd436c919c6 dd53e23487da03fd02396306d248cda0`
`e99f33420f577ee8ce54b67080280d1e c69821bcb6a8839396f965ab6ff72a70`)

Xiaoyun Wang (Tsinghua University), 2004

Real Cryptographic Hash Functions Get Broken

Most crypto (encryption, digital signatures, etc) only gets broken very rarely.
But hash functions are harder to build and *do* get broken.

- MD5 (1992) was broken in 2004 - can now find collisions very quickly.
- SHA-1 (1995) was broken in 2017 - A big computer can find collisions

MD5(d131dd02c5e6eec4693d9a0698aff95c 2fcab58712467eab4004583eb8fb7f89
55ad340609f4b30283e488832571415a 085125e8f7cdc99fd91dbdf280373c5b
d8823e3156348f5bae6dacd436c919c6 dd53e2b487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080a80d1e c69821bcb6a8839396f9652b6ff72a70)

= MD5(d131dd02c5e6eec4693d9a0698aff95c 2fcab50712467eab4004583eb8fb7f89
55ad340609f4b30283e4888325f1415a 085125e8f7cdc99fd91dbd7280373c5b
d8823e3156348f5bae6dacd436c919c6 dd53e23487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080280d1e c69821bcb6a8839396f965ab6ff72a70)

Xiaoyun Wang (Tsinghua University), 2004

Real Cryptographic Hash Functions Get Broken

Most crypto (encryption, digital signatures, etc) only gets broken very rarely.
But hash functions are harder to build and *do* get broken.

- MD5 (1992) was broken in 2004 - can now find collisions very quickly.
- SHA-1 (1995) was broken in 2017 - A big computer can find collisions
- SHA-256/SHA-512 (2001) are not known to be broken

MD5(d131dd02c5e6eec4693d9a0698aff95c 2fcab58712467eab4004583eb8fb7f89
55ad340609f4b30283e488832571415a 085125e8f7cdc99fd91dbdf280373c5b
d8823e3156348f5bae6dacd436c919c6 dd53e2b487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080a80d1e c69821bcb6a8839396f9652b6ff72a70)

= MD5(d131dd02c5e6eec4693d9a0698aff95c 2fcab50712467eab4004583eb8fb7f89
55ad340609f4b30283e4888325f1415a 085125e8f7cdc99fd91dbd7280373c5b
d8823e3156348f5bae6dacd436c919c6 dd53e23487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080280d1e c69821bcb6a8839396f965ab6ff72a70)

Xiaoyun Wang (Tsinghua University), 2004

Real Cryptographic Hash Functions Get Broken

Most crypto (encryption, digital signatures, etc) only gets broken very rarely.
But hash functions are harder to build and *do* get broken.

- MD5 (1992) was broken in 2004 - can now find collisions very quickly.
- SHA-1 (1995) was broken in 2017 - A big computer can find collisions
- SHA-256/SHA-512 (2001) are not known to be broken
- SHA-3 (2015) is new and not known to be broken

MD5(d131dd02c5e6eec4693d9a0698aff95c 2fcab58712467eab4004583eb8fb7f89
55ad340609f4b30283e488832571415a 085125e8f7cdc99fd91dbdf280373c5b
d8823e3156348f5bae6dacd436c919c6 dd53e2b487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080a80d1e c69821bcb6a8839396f9652b6ff72a70)

= MD5(d131dd02c5e6eec4693d9a0698aff95c 2fcab50712467eab4004583eb8fb7f89
55ad340609f4b30283e4888325f1415a 085125e8f7cdc99fd91dbd7280373c5b
d8823e3156348f5bae6dacd436c919c6 dd53e23487da03fd02396306d248cda0
e99f33420f577ee8ce54b67080280d1e c69821bcb6a8839396f965ab6ff72a70)

Xiaoyun Wang (Tsinghua University), 2004

Real Cryptographic Hash Functions Get Broken

Most crypto (encryption, digital signatures, etc) only gets broken very rarely.
But hash functions are harder to build and *do* get broken.

- MD5 (1992) was broken in 2004 - can now find collisions very quickly.
- SHA-1 (1995) was broken in 2017 - A big computer can find collisions
- SHA-256/SHA-512 (2001) are not known to be broken
- SHA-3 (2015) is new and not known to be broken

MD5(`d131dd02c5e6eec4693d9a0698aff95c 2fcab58712467eab4004583eb8fb7f89`
`55ad340609f4b30283e488832571415a 085125e8f7cdc99fd91dbdf280373c5b`
`d8823e3156348f5bae6dacd436c919c6 dd53e2b487da03fd02396306d248cda0`
`e99f33420f577ee8ce54b67080a80d1e c69821bcb6a8839396f9652b6ff72a70`)

= MD5(`d131dd02c5e6eec4693d9a0698aff95c 2fcab50712467eab4004583eb8fb7f89`
`55ad340609f4b30283e4888325f1415a 085125e8f7cdc99fd91dbd7280373c5b`
`d8823e3156348f5bae6dacd436c919c6 dd53e23487da03fd02396306d248cda0`
`e99f33420f577ee8ce54b67080280d1e c69821bcb6a8839396f965ab6ff72a70`)

Xiaoyun Wang (Tsinghua University), 2004

You can ignore the alphabet soup of hash function names. Bitcoin uses SHA256.

Lecture 2 Outline

1. Cryptographic Hash Functions

   - **Blockchains**

   - Proofs of Work

2. Putting DCash “on the blockchain”, with an authority

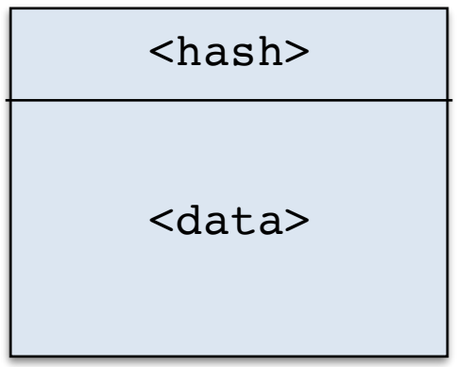
3. The idea of decentralization

4. Decentralized DCash with an Angel

5. Decentralized DCash via proofs-of-work

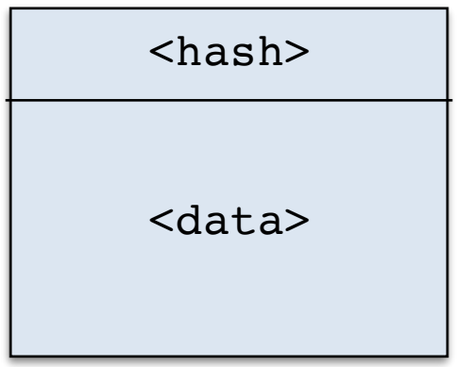
Hash Functions and Blocks

Definition: A *block* is a data structure with two fields: Hash and Data.

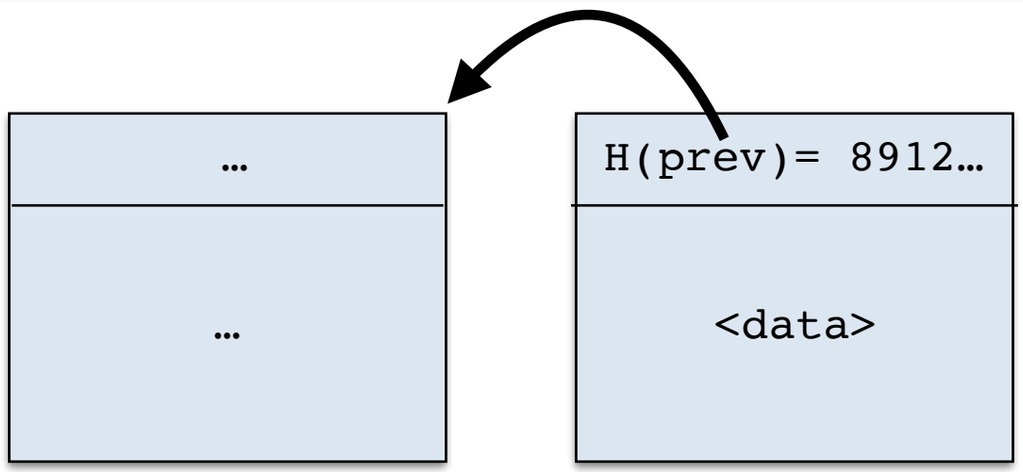


Hash Functions and Blocks

Definition: A *block* is a data structure with two fields: Hash and Data.

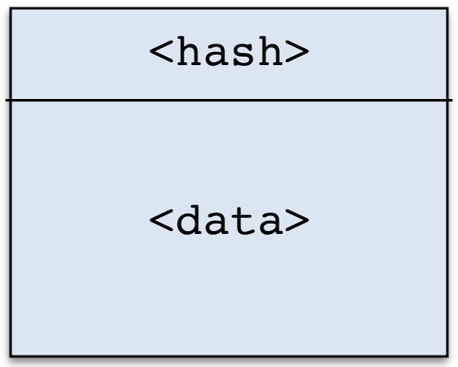


Definition: If the hash field of a block is `H(x)`, where `x` is the another block, we say that the block *points to* `x` and we indicate this in a diagram as:

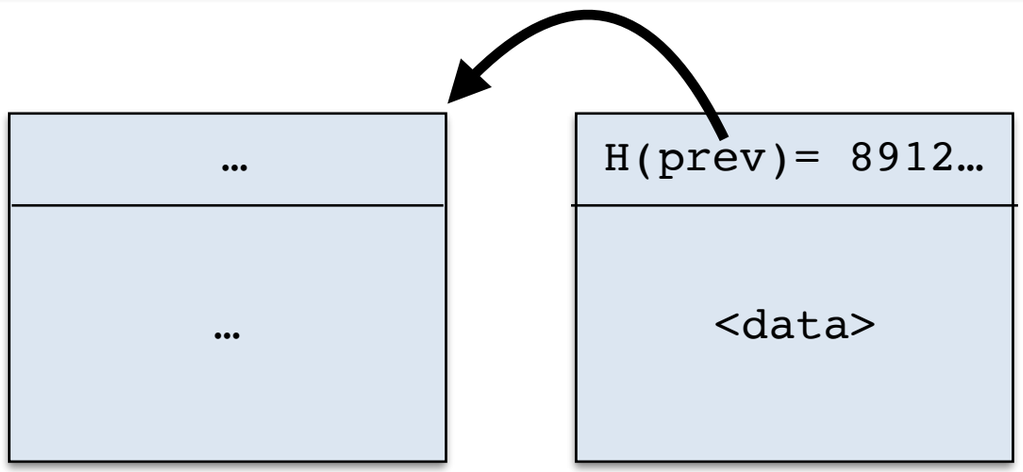


Hash Functions and Blocks

Definition: A *block* is a data structure with two fields: Hash and Data.



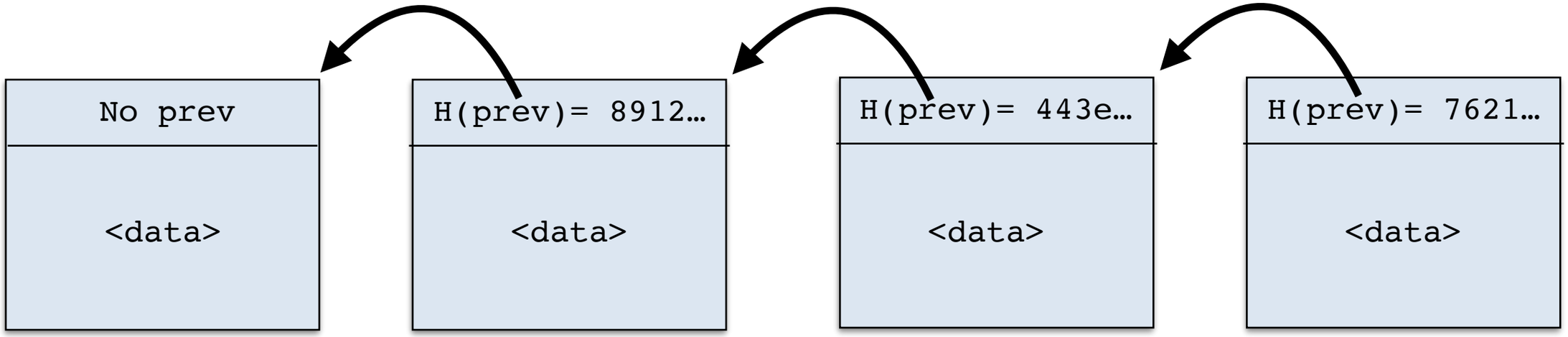
Definition: If the hash field of a block is `H(x)`, where `x` is the another block, we say that the block *points to* `x` and we indicate this in a diagram as:



- Note: Input to H should include *both* fields of previous block (hash *and* data)

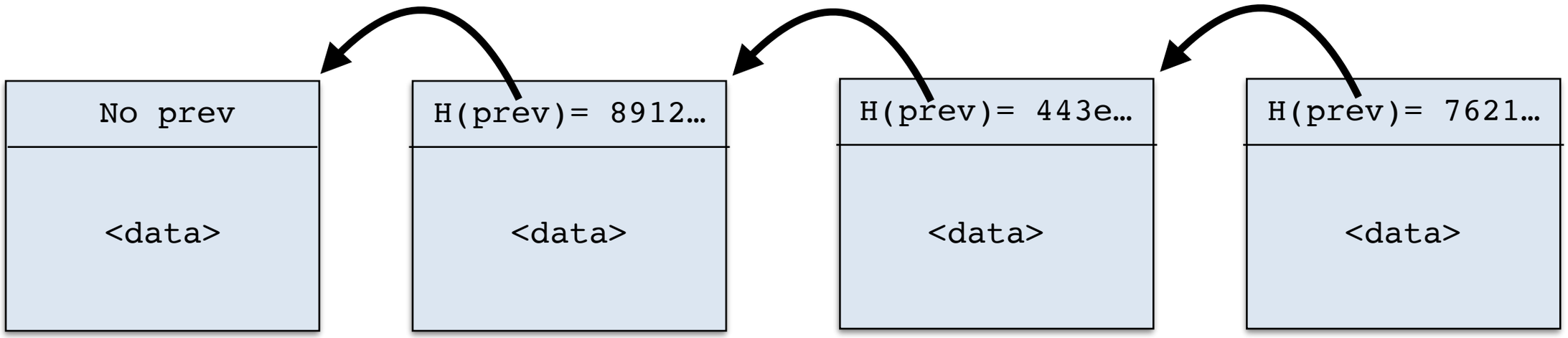
Blockchains

Definition: A *blockchain* is linear sequence of blocks that point to each other.



Blockchains

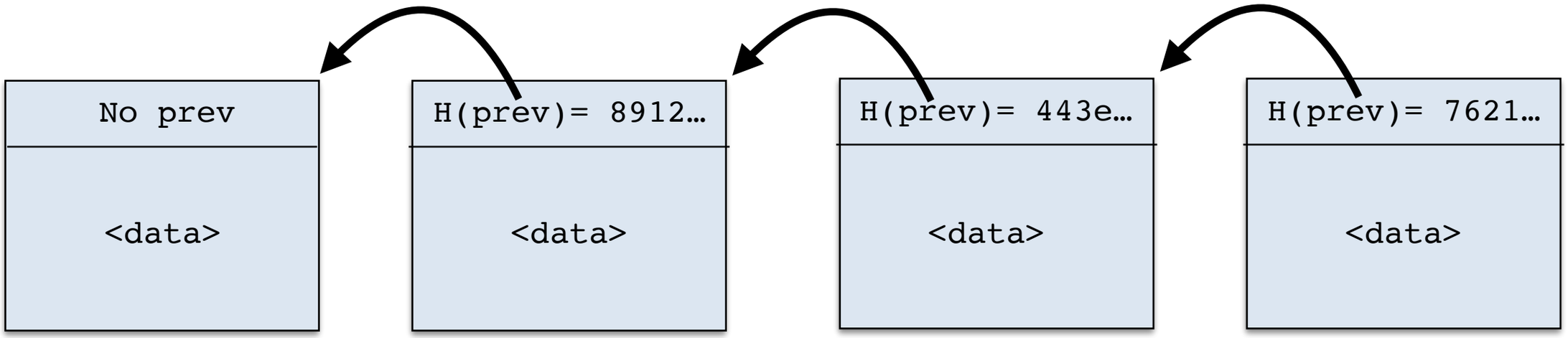
Definition: A *blockchain* is linear sequence of blocks that point to each other.



Given the last block, it's easy to add a block to a blockchain. Just evaluating `H` on the last block and put the result in your new block's hash field.

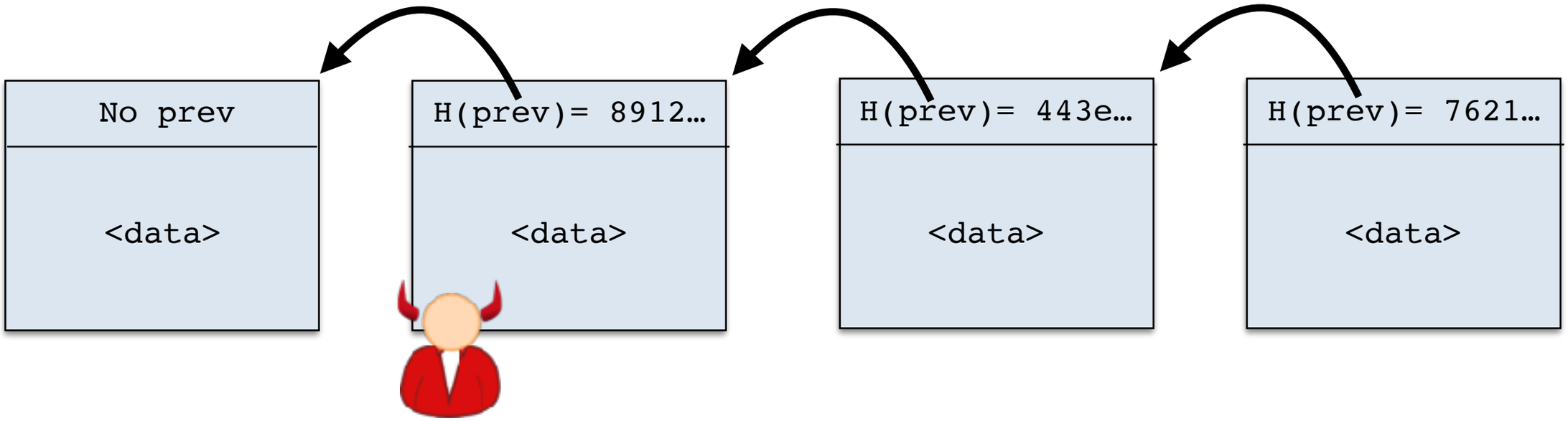
Blockchains Integrity

Blockchain integrity property: If you know the last hash field (`7612…` in example), you can check integrity of data in entire blockchain



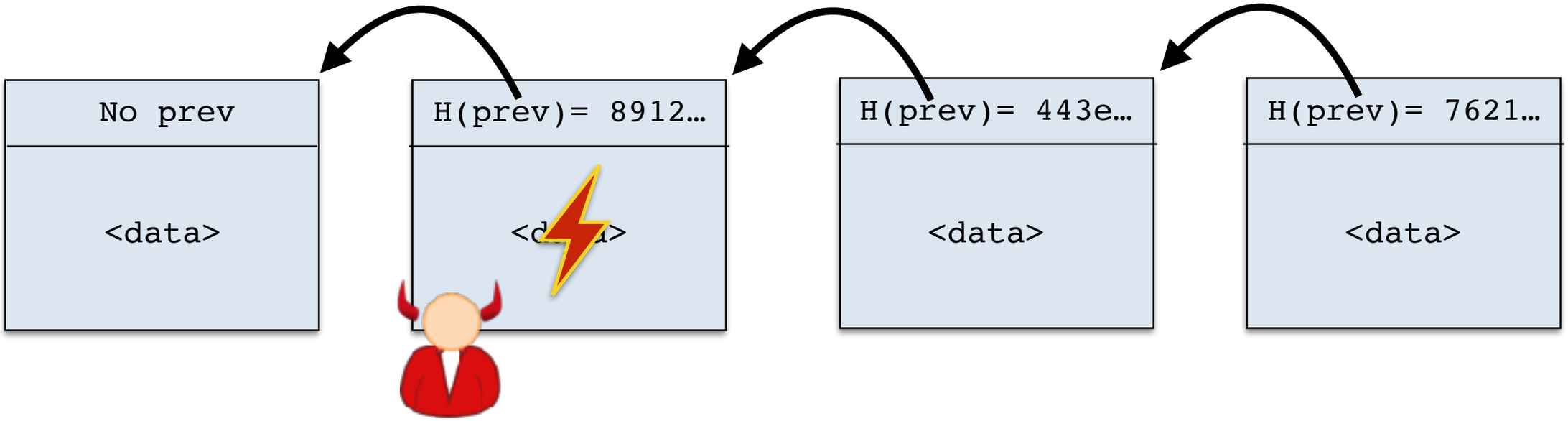
Blockchains Integrity

Blockchain integrity property: If you know the last hash field (`7612…` in example), you can check integrity of data in entire blockchain



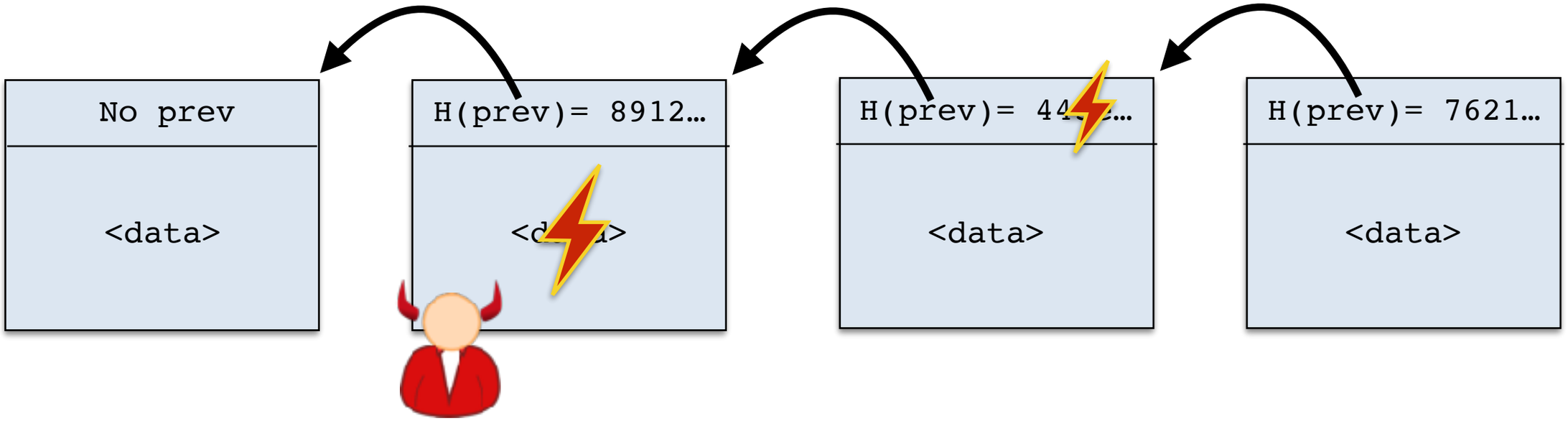
Blockchains Integrity

Blockchain integrity property: If you know the last hash field (`7612…` in example), you can check integrity of data in entire blockchain



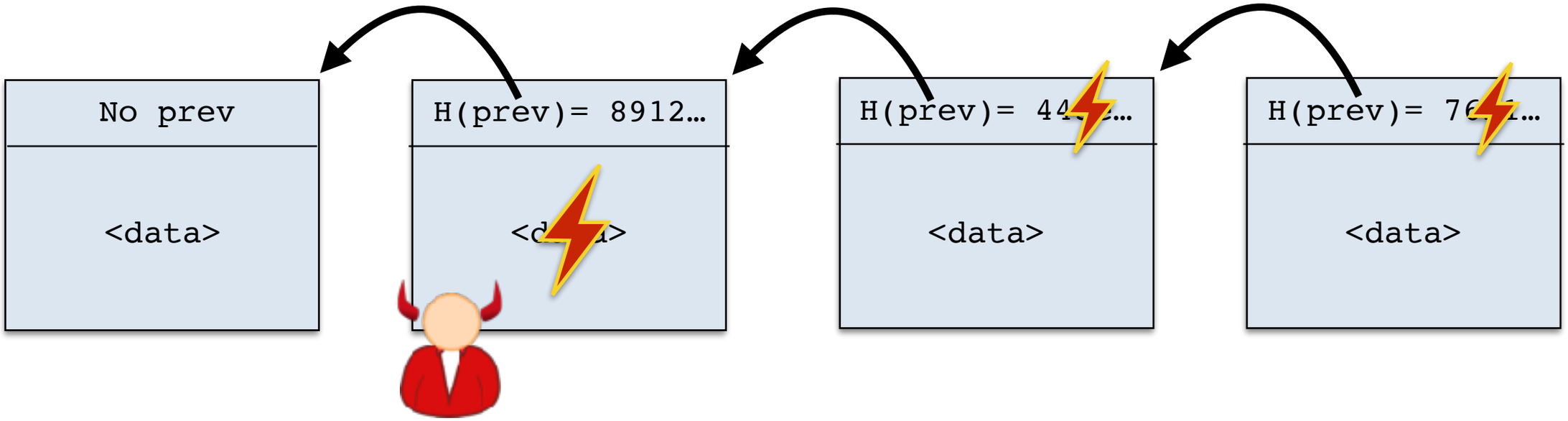
Blockchains Integrity

Blockchain integrity property: If you know the last hash field (`7612…` in example), you can check integrity of data in entire blockchain



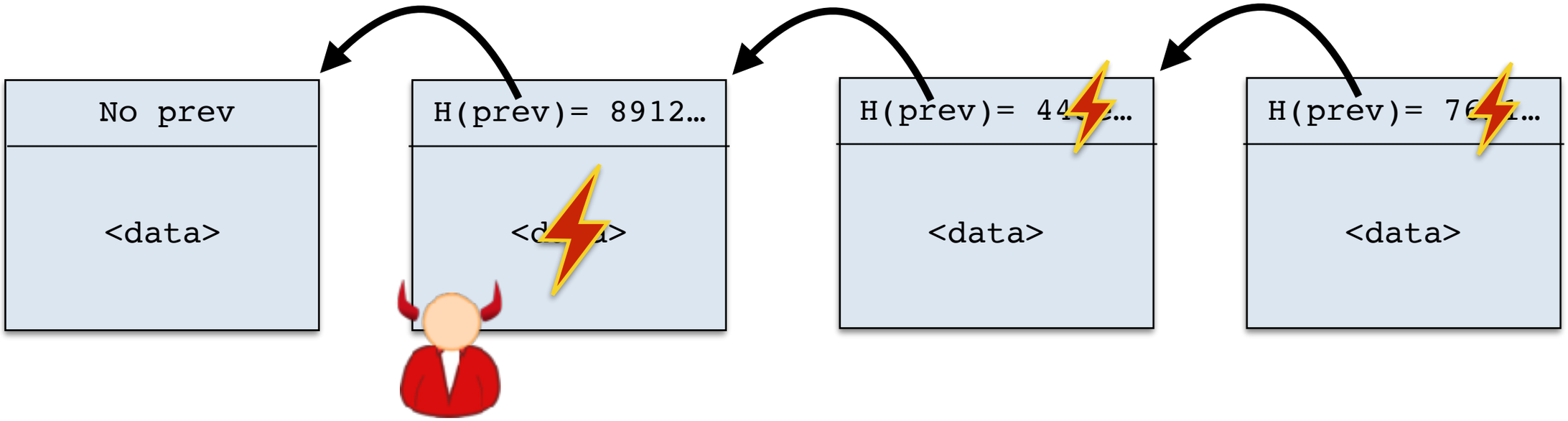
Blockchains Integrity

Blockchain integrity property: If you know the last hash field (`7612…` in example), you can check integrity of data in entire blockchain



Blockchains Integrity

Blockchain integrity property: If you know the last hash field (`7612…` in example), you can check integrity of data in entire blockchain



Theorem that can be formalized proved: Changing/deleting/adding a block without changing the final hash requires finding a collision in `H` (which we believe is infeasible).

Lecture 2 Outline

1. Cryptographic Hash Functions

   - Blockchains

   - **Proofs of Work**

2. Putting DCash “on the blockchain”, with an authority

3. The idea of decentralization

4. Decentralized DCash with an Angel

5. Decentralized DCash via proofs-of-work

Hash Security Beyond Collision Resistance

General security goal: Input/output behavior of function $H(\cdot)$ should look totally random to any feasible analysis.

Hash Security Beyond Collision Resistance

General security goal: Input/output behavior of function `H(·)` should look totally random to any feasible analysis.

- No detectable patterns/biases in output bits
- If `x≠x'` then `H(x)` and `H(x')` should look like *independent* random bit strings
- Note: The function `H` is itself fixed and not random. It should *look* random.

Hash Security Beyond Collision Resistance

General security goal: Input/output behavior of function `H(·)` should look totally random to any feasible analysis.

- No detectable patterns/biases in output bits
- If `x≠x'` then `H(x)` and `H(x')` should look like *independent* random bit strings
- Note: The function `H` is itself fixed and not random. It should *look* random.

Example consequence of this security: For each `x`, the "probability" that `y=H(x)` starts with at least `z` zeros is $(1/2)^z$

Hash Security Beyond Collision Resistance

General security goal: Input/output behavior of function `H(·)` should look totally random to any feasible analysis.

- No detectable patterns/biases in output bits
- If `x≠x'` then `H(x)` and `H(x')` should look like *independent* random bit strings
- Note: The function `H` is itself fixed and not random. It should *look* random.

Example consequence of this security: For each `x`, the "probability" that `y=H(x)` starts with at least `z` zeros is $(1/2)^z$

- Probability `1/2` that it starts with at least one zero, `1/4` that it starts with at least two zeros, `1/8` for three, `1/16` for four, etc
- Once `z≈80` this probability very small by computer standards
- In probability jargon: Number of starting zeros is geometric with parameter `1/2`.

Hash Function Application: Proofs of Work Puzzles

Definition: A *proof-of-work puzzle* consists of an input string `x` and a hardness parameter `z`. To solve the puzzle you must find a string `c` such that `H(x,c)` starts with `z` zeros.

Hash Function Application: Proofs of Work Puzzles

Definition: A *proof-of-work puzzle* consists of an input string **`x`** and a hardness parameter **`z`**. To solve the puzzle you must find a string **`c`** such that **`H(x,c)`** starts with **`z`** zeros.

Hardness parameter: Integer **`z`**
Input: string **`x`**
Solution: string **`c`** such that **`H(x,c)`** starts with **`z`** zeros

Hash Function Application: Proofs of Work Puzzles

Definition: A *proof-of-work puzzle* consists of an input string **x** and a hardness parameter **z**. To solve the puzzle you must find a string **c** such that **H(x,c)** starts with **z** zeros.

Hardness parameter: Integer **z**
Input: string **x**
Solution: string **c** such that **H(x,c)** starts with **z** zeros

If **H** is a secure hash function then best algorithm just tries different values of c until it gets lucky:

Canonical algorithm: On input **x**:

```
For c = 0,1,2,…
  If H(x,c) starts with z zeros: Output c
```

Hash Function Application: Proofs of Work Puzzles

Definition: A *proof-of-work puzzle* consists of an input string **x** and a hardness parameter **z**. To solve the puzzle you must find a string **c** such that **H(x,c)** starts with **z** zeros.

Hardness parameter: Integer **z**
Input: string **x**
Solution: string **c** such that **H(x,c)** starts with **z** zeros

If **H** is a secure hash function then best algorithm just tries different values of c until it gets lucky:

Canonical algorithm: On input **x**:

```
For c = 0,1,2,…
  If H(x,c) starts with z zeros: Output c
```

- Canonical algorithm evaluates hash 2^z times *on average*

Hash Function Application: Proofs of Work Puzzles

Definition: A *proof-of-work puzzle* consists of an input string **x** and a hardness parameter **z**. To solve the puzzle you must find a string **c** such that **H(x,c)** starts with **z** zeros.

Hardness parameter: Integer **z**
Input: string **x**
Solution: string **c** such that **H(x,c)** starts with **z** zeros

If **H** is a secure hash function then best algorithm just tries different values of c until it gets lucky:

Canonical algorithm: On input **x**:

```
For c = 0,1,2,…
  If H(x,c) starts with z zeros: Output c
```

- Canonical algorithm evaluates hash 2^z times *on average*
- **z=20** is quick to solve, **z=70** is solvable only by powerful computers

Proofs-of-Work for Anti-Spam (Not used, sadly)

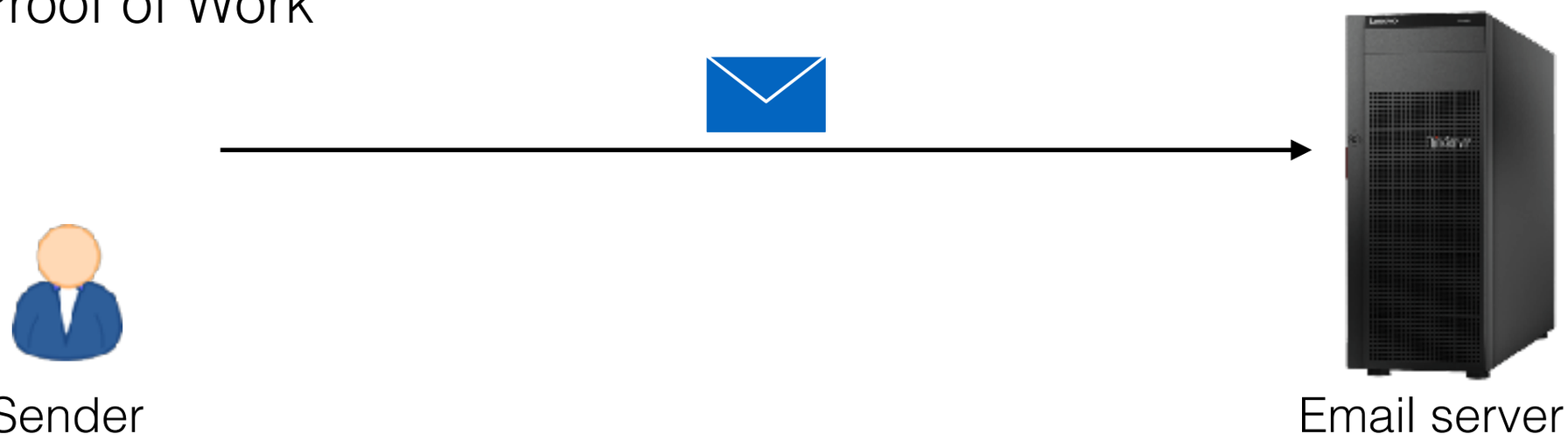
POW = Proof of Work

Sender

Email server

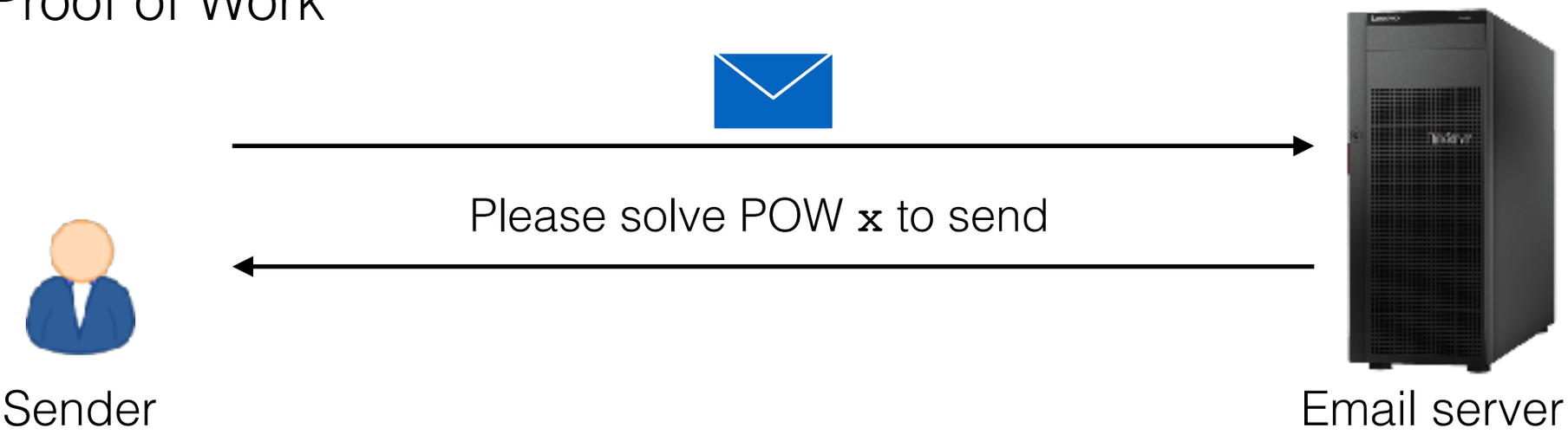
Proofs-of-Work for Anti-Spam (Not used, sadly)

POW = Proof of Work



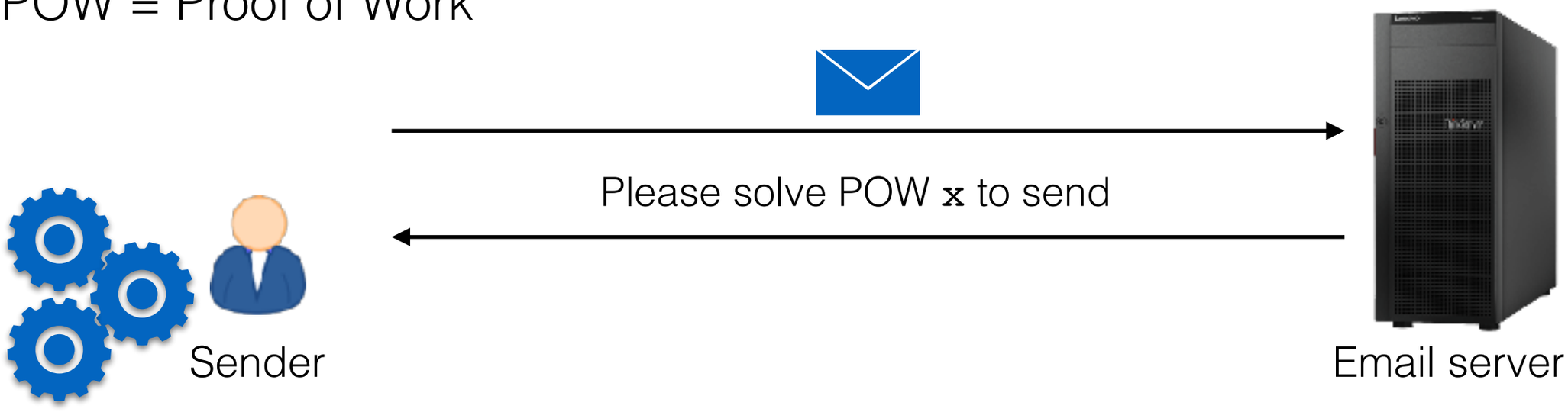
Proofs-of-Work for Anti-Spam (Not used, sadly)

POW = Proof of Work



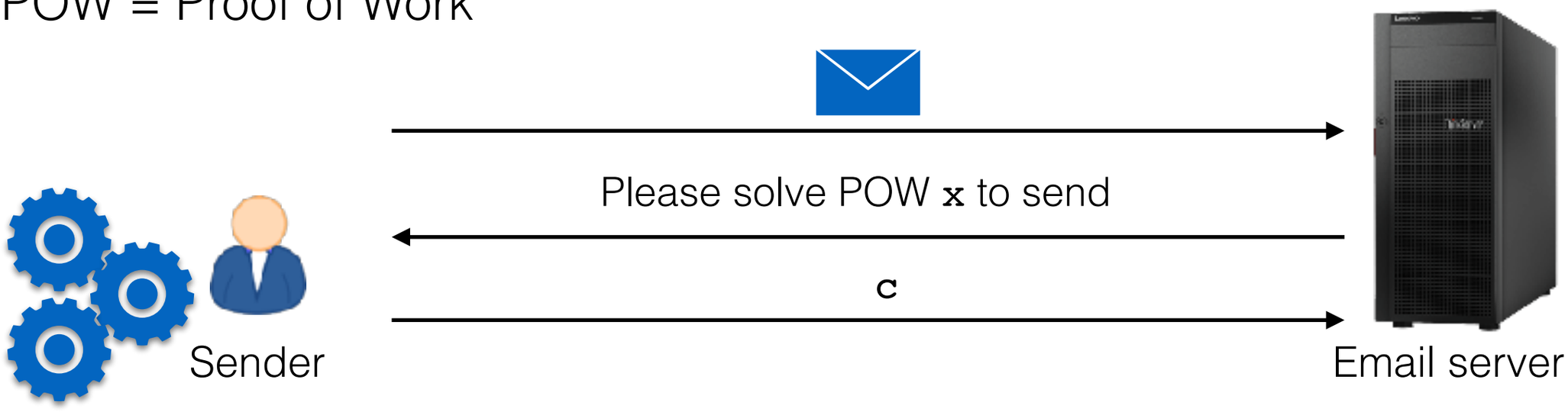
Proofs-of-Work for Anti-Spam (Not used, sadly)

POW = Proof of Work



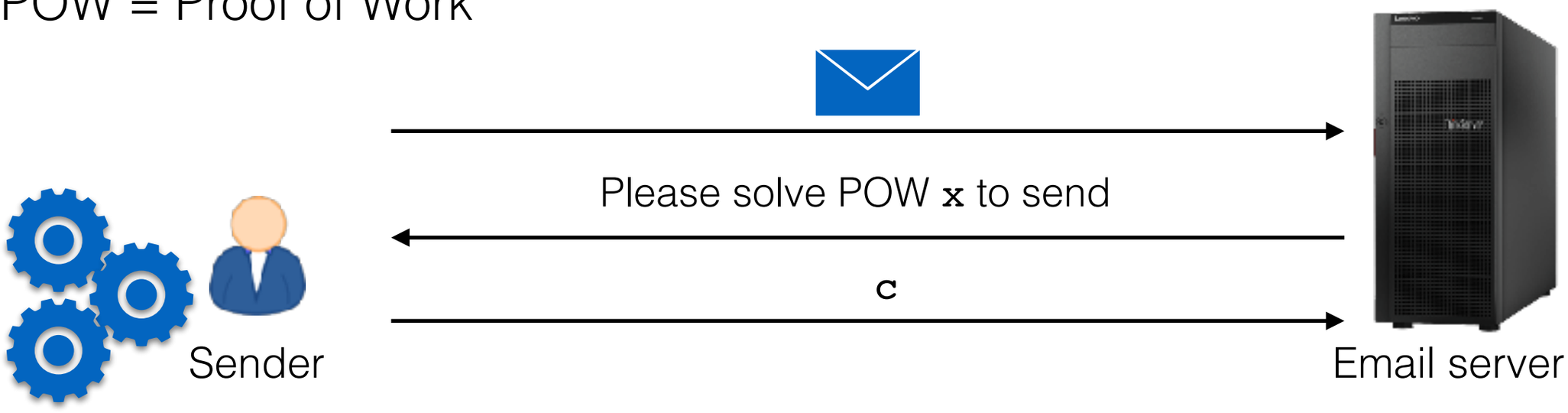
Proofs-of-Work for Anti-Spam (Not used, sadly)

POW = Proof of Work



Proofs-of-Work for Anti-Spam (Not used, sadly)

POW = Proof of Work

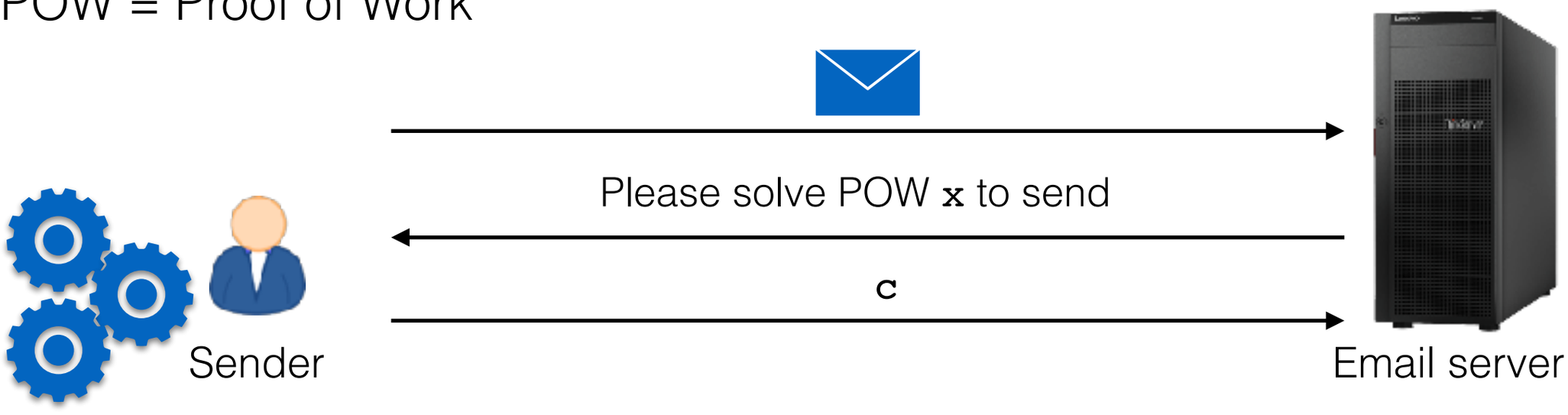


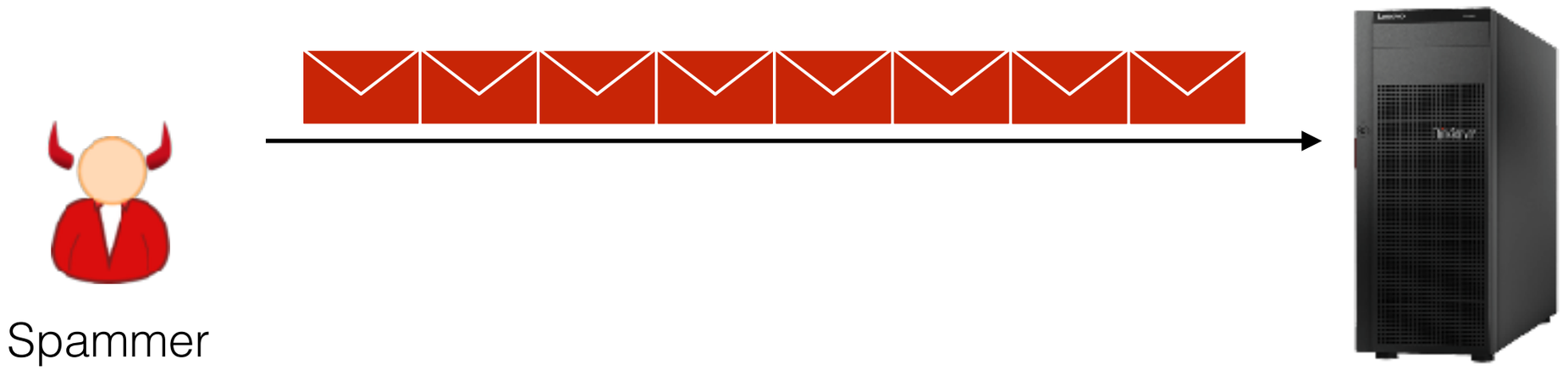
Spammer

Email server

Proofs-of-Work for Anti-Spam (Not used, sadly)

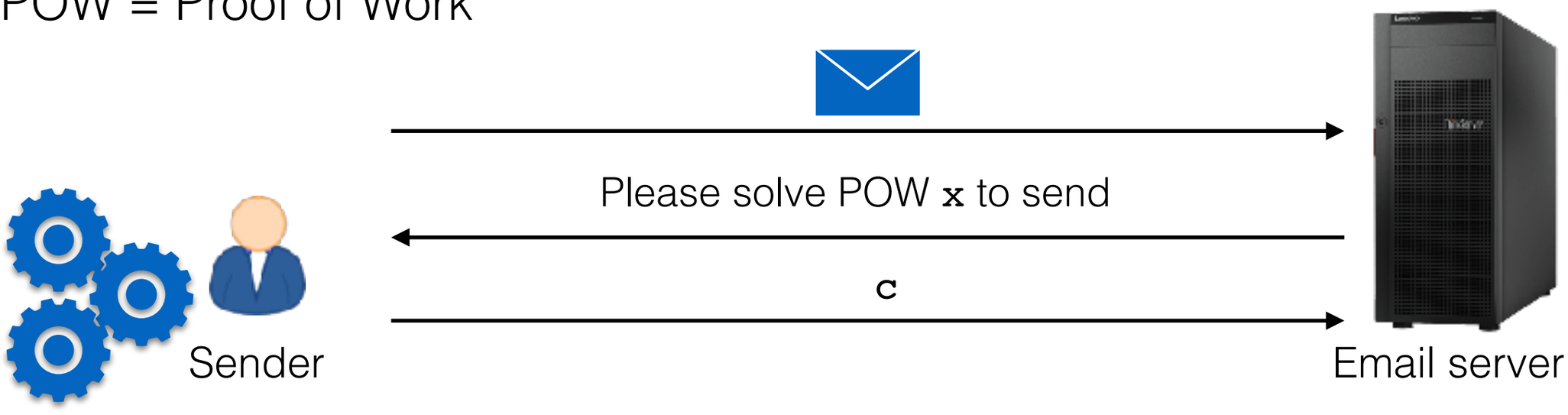
POW = Proof of Work

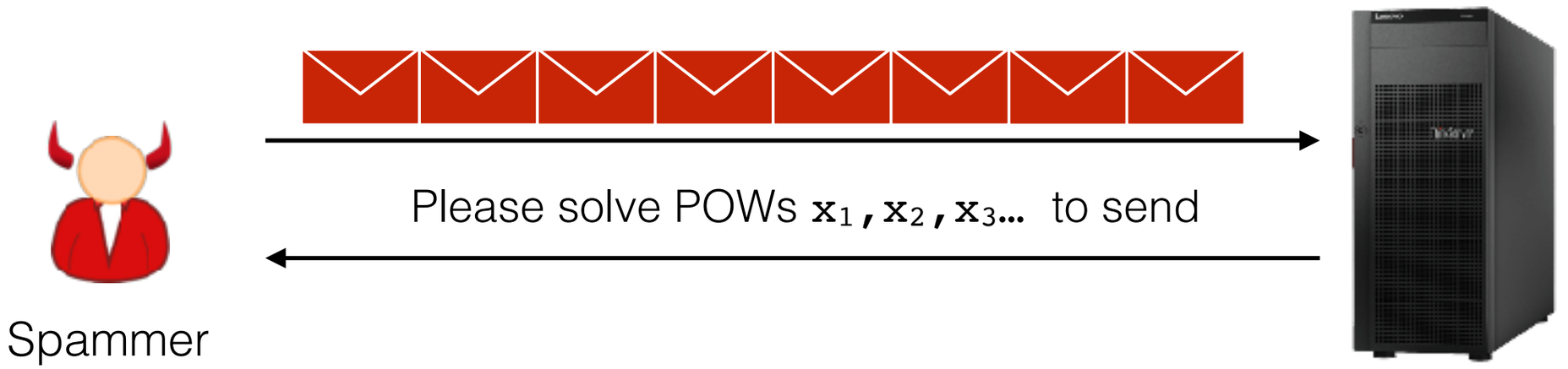




Proofs-of-Work for Anti-Spam (Not used, sadly)

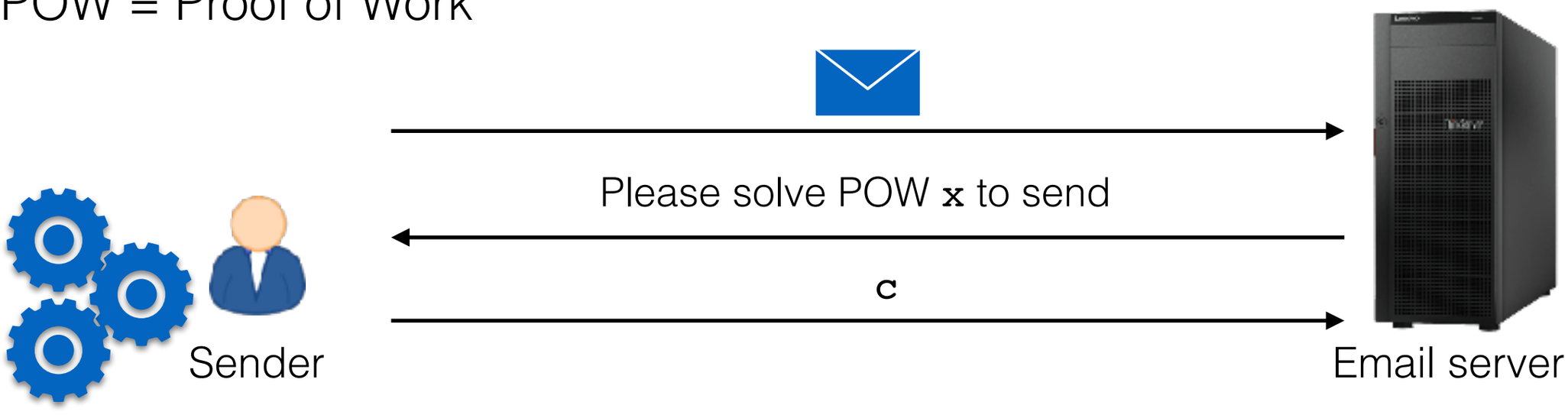
POW = Proof of Work

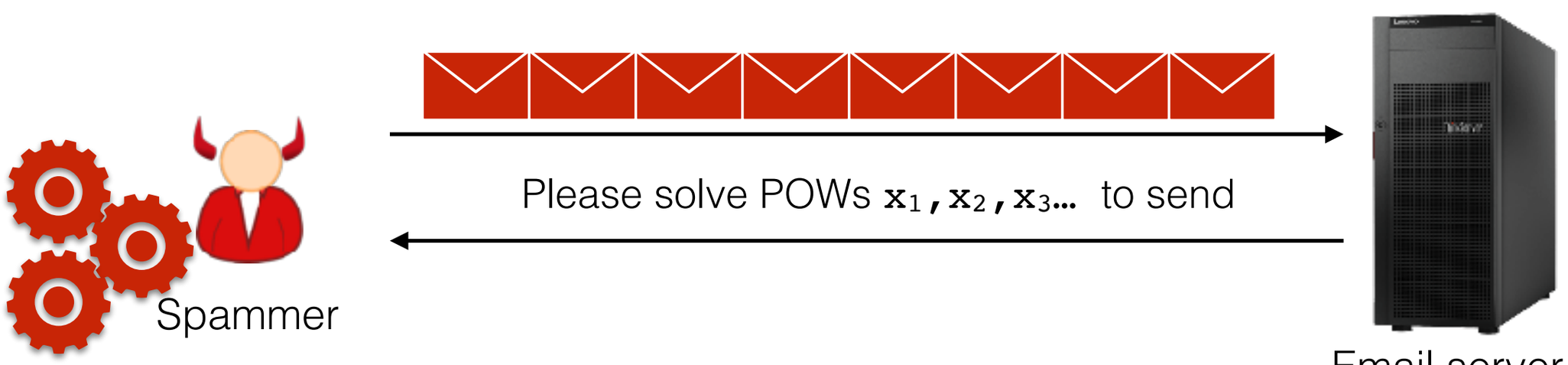




Proofs-of-Work for Anti-Spam (Not used, sadly)

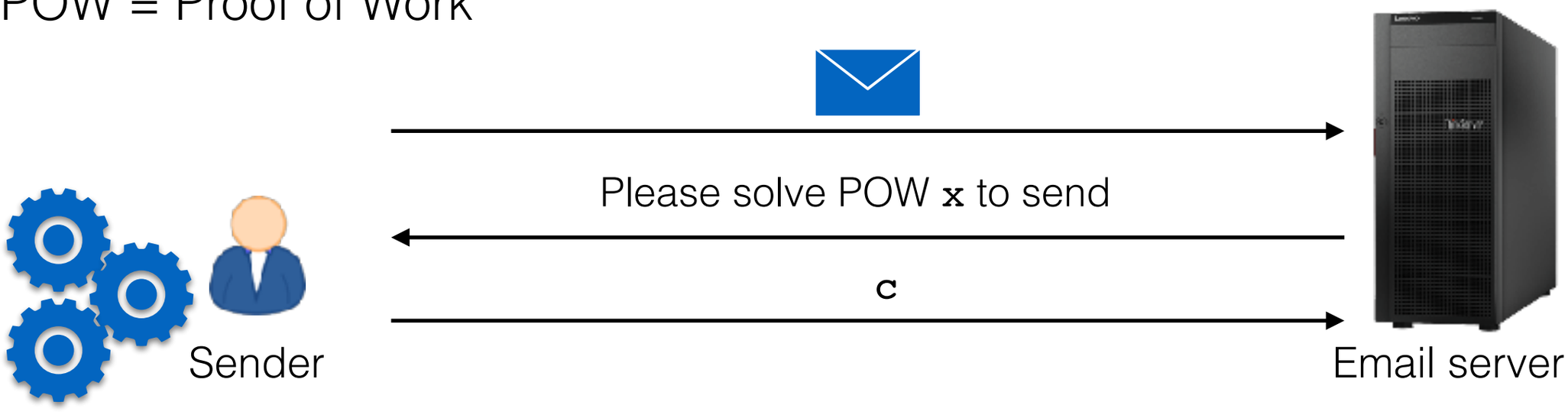
POW = Proof of Work

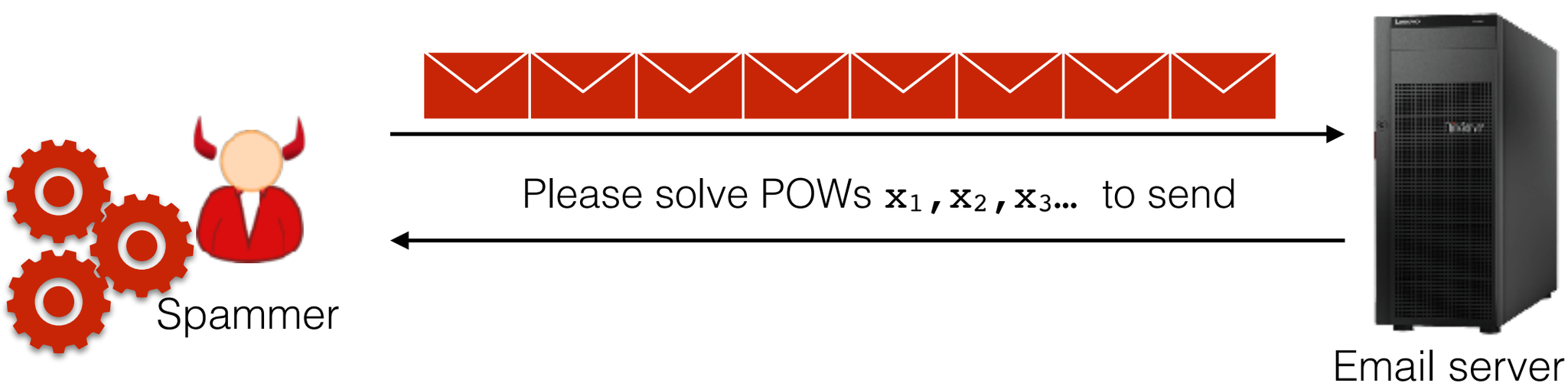




Proofs-of-Work for Anti-Spam (Not used, sadly)

POW = Proof of Work

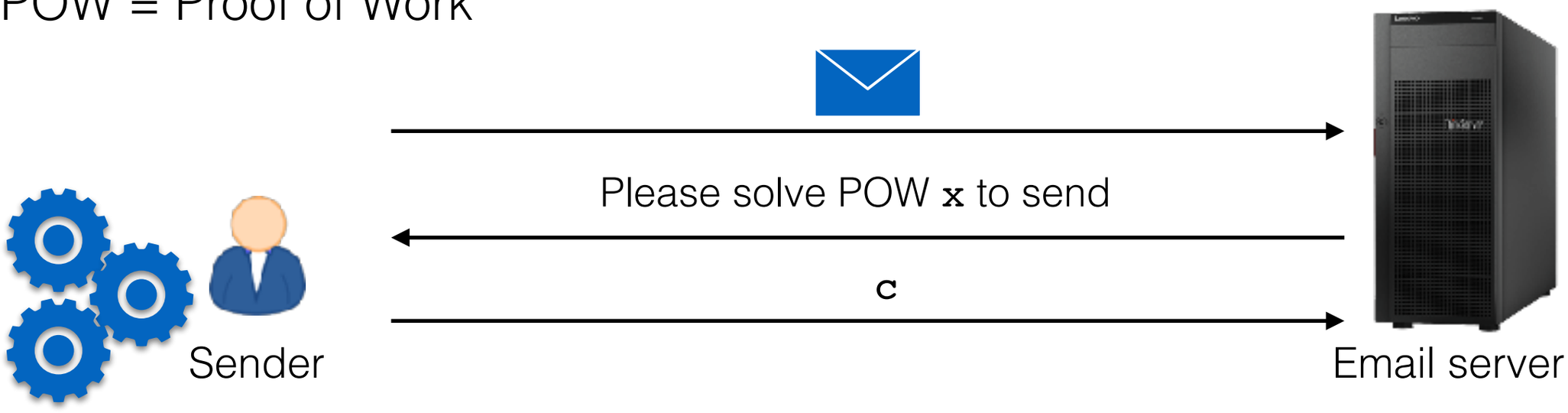


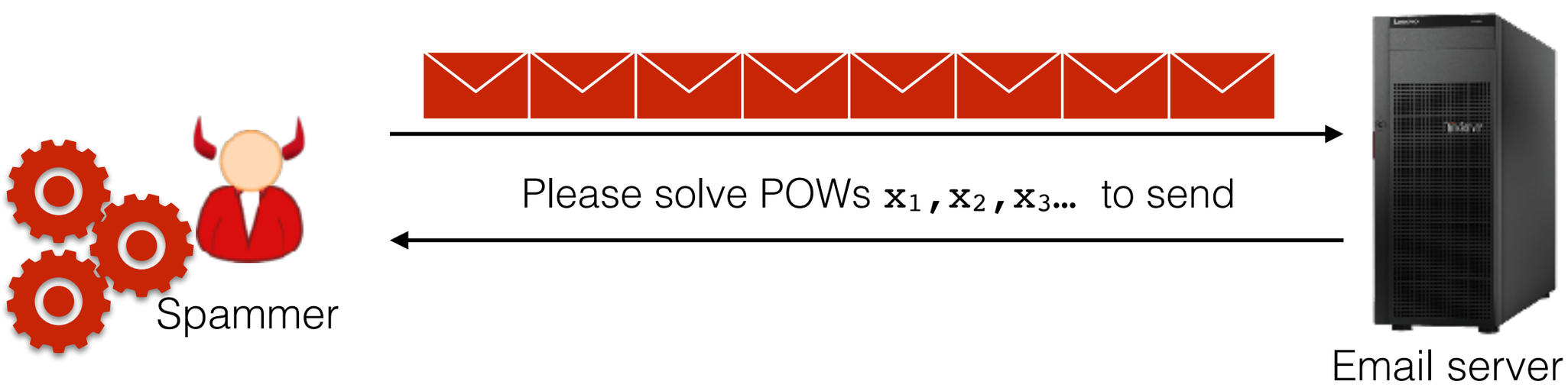


- Evaluating POW takes time/money/compute hardware for good and bad parties

Proofs-of-Work for Anti-Spam (Not used, sadly)

POW = Proof of Work





- Evaluating POW takes time/money/compute hardware for good and bad parties
- Set hardness parameter z so that:
  1. Normal senders can solve one puzzle quickly, without noticing the work
  2. It is unprofitable for spammers to solve millions of puzzles

Lecture 2 Outline

1. Cryptographic Hash Functions

   - Blockchains

   - Proofs of Work

2. Putting DCash “on the blockchain”, with an authority

3. The idea of decentralization

4. Decentralized DCash with an Angel

5. Decentralized DCash via proofs-of-work

Recalling DCash

Initialization: Ben, Emily, and David all generate keys for digital signatures

David's verification key: PK_{david} = 5e7843…
Ben's verification key: PK_{ben} = 88f01e…
Emily' verification key: PK_{emily} = 16823a…

TranID	From	To	Amount	Signature
1	88f01e…	16823a…	1	91a001…
2	5e7843…	16823a…	2	2c3118…
3	88f01e…	5e7843…	3	7623a6…
4	16823a…	5e7843…	6	987234…
5	88f01e…	5e7843…	1	234b98…

Recalling DCash

Initialization: Ben, Emily, and David all generate keys for digital signatures

David's verification key: PK_{david} = 5e7843…
Ben's verification key: PK_{ben} = 88f01e…
Emily' verification key: PK_{emily} = 16823a…

TranID	From	To	Amount	Signature
1	88f01e…	16823a…	1	91a001…
2	5e7843…	16823a…	2	2c3118…
3	88f01e…	5e7843…	3	7623a6…
4	16823a…	5e7843…	6	987234…
5	88f01e…	5e7843…	1	234b98…

- In the terminology of the start of Chapter 3, this is an "account-based ledger"

Recalling DCash

Initialization: Ben, Emily, and David all generate keys for digital signatures

David's verification key: PK_{david} = 5e7843…
Ben's verification key: PK_{ben} = 88f01e…
Emily' verification key: PK_{emily} = 16823a…

TranID	From	To	Amount	Signature
1	88f01e…	16823a…	1	91a001…
2	5e7843…	16823a…	2	2c3118…
3	88f01e…	5e7843…	3	7623a6…
4	16823a…	5e7843…	6	987234…
5	88f01e…	5e7843…	1	234b98…

- In the terminology of the start of Chapter 3, this is an "account-based ledger"
- To determine if a transaction is valid, we must rerun entire history of ledger

DCash 2.0 (a.k.a. Scroogecoin, Text section 1.5)

- Move from “account-based ledger” to “transaction-based ledger”

DCash 2.0 (a.k.a. Scroogecoin, Text section 1.5)

- Move from “account-based ledger” to “transaction-based ledger”
- Store transactions in a blockchain managed by a semi-trusted authority

DCash 2.0 (a.k.a. Scroogecoin, Text section 1.5)

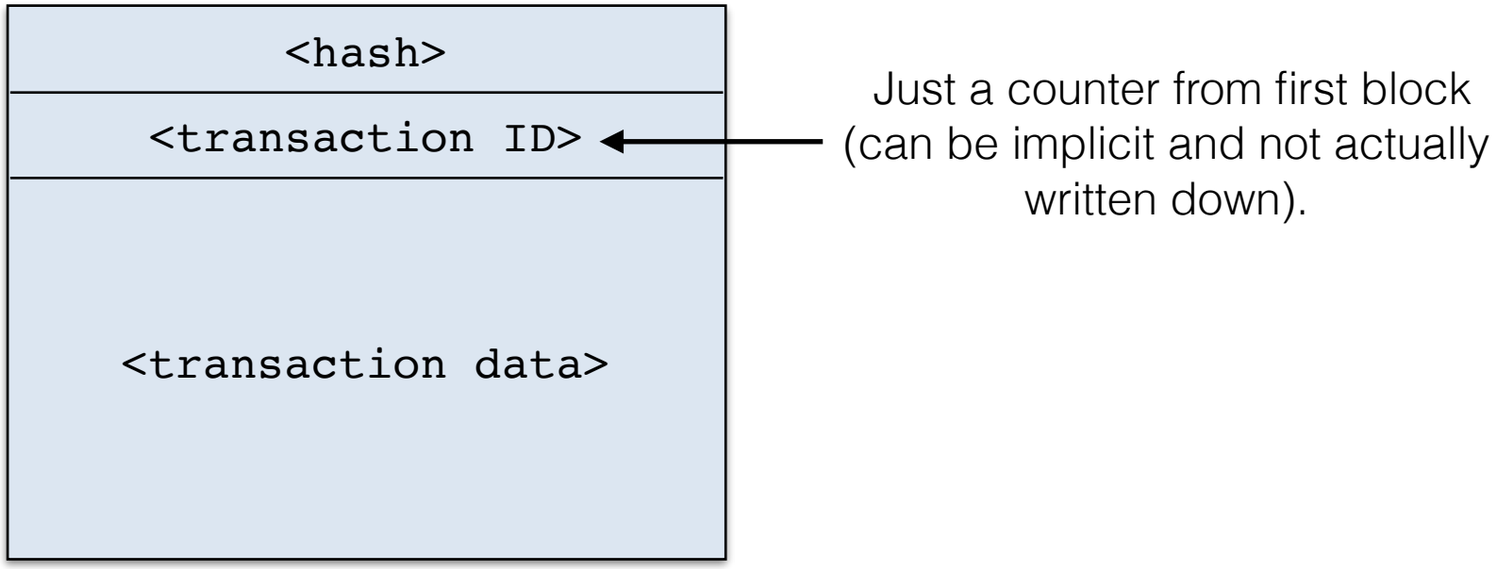
- Move from “account-based ledger” to “transaction-based ledger”
- Store transactions in a blockchain managed by a semi-trusted authority

Authority’s responsibilities:

1. Publish blockchain contents publicly.
2. Create coins and assign them to owners (public keys) at will.
3. Receive transactions notifications and commit them to the blockchain.

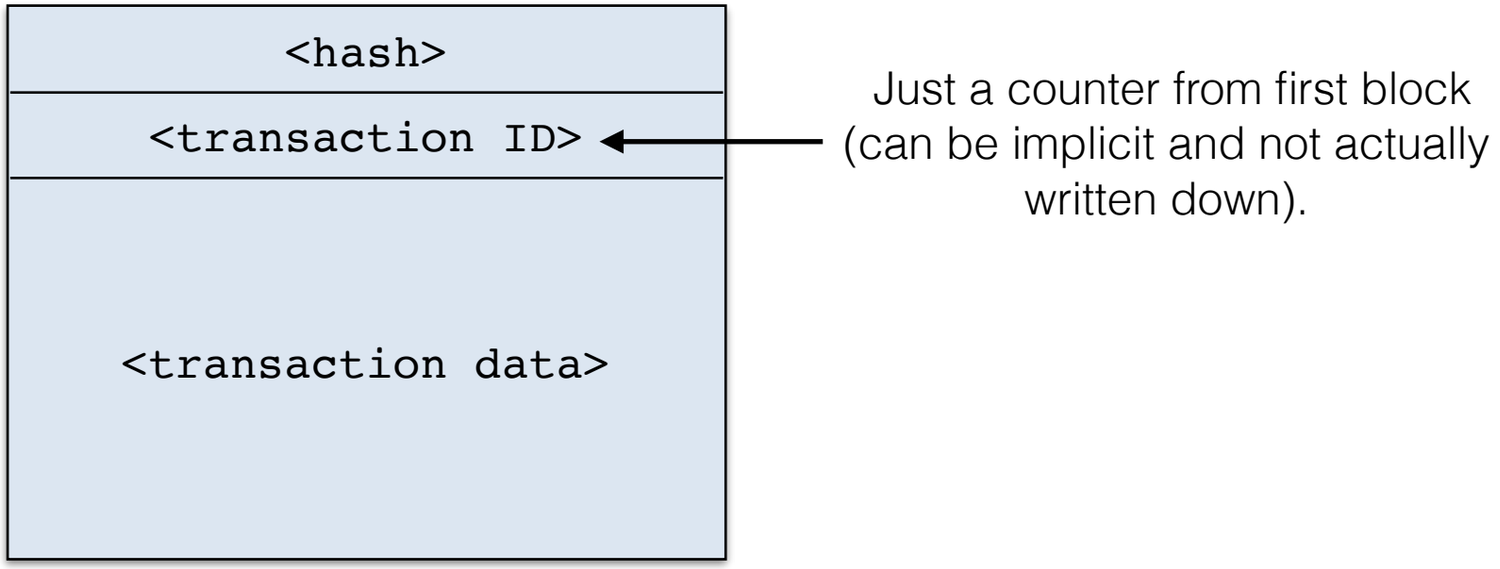
DCash 2.0 Blocks, Transactions, and Coins

- One transaction per block



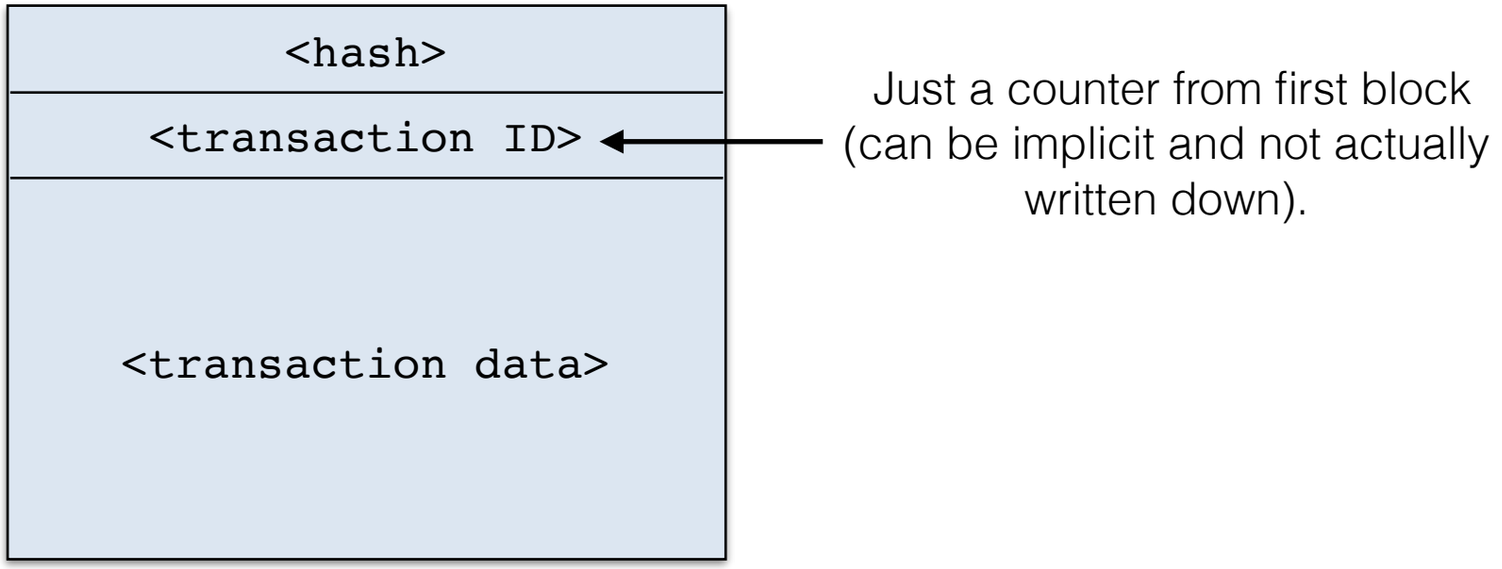
DCash 2.0 Blocks, Transactions, and Coins

- One transaction per block
- Each transaction *consumes* some coins and *creates* new coins



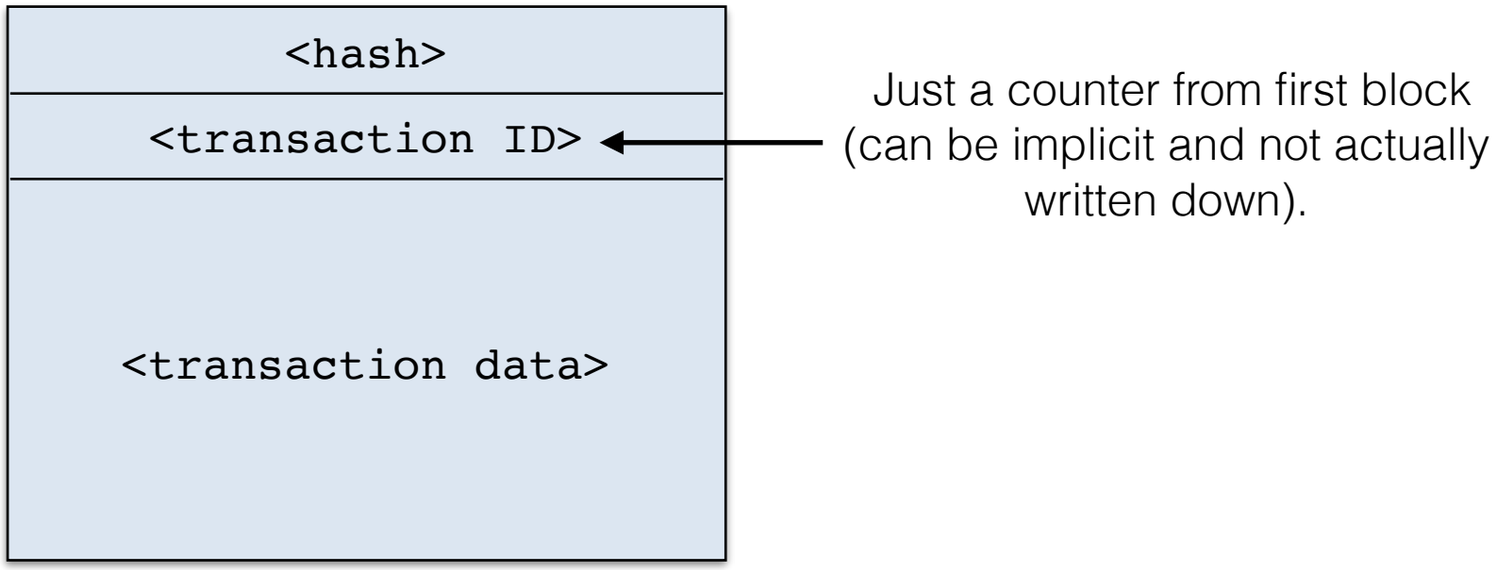
DCash 2.0 Blocks, Transactions, and Coins

- One transaction per block
- Each transaction *consumes* some coins and *creates* new coins
- Each coin is created and consumed once. A created coin is *unspent* until it is consumed



DCash 2.0 Blocks, Transactions, and Coins

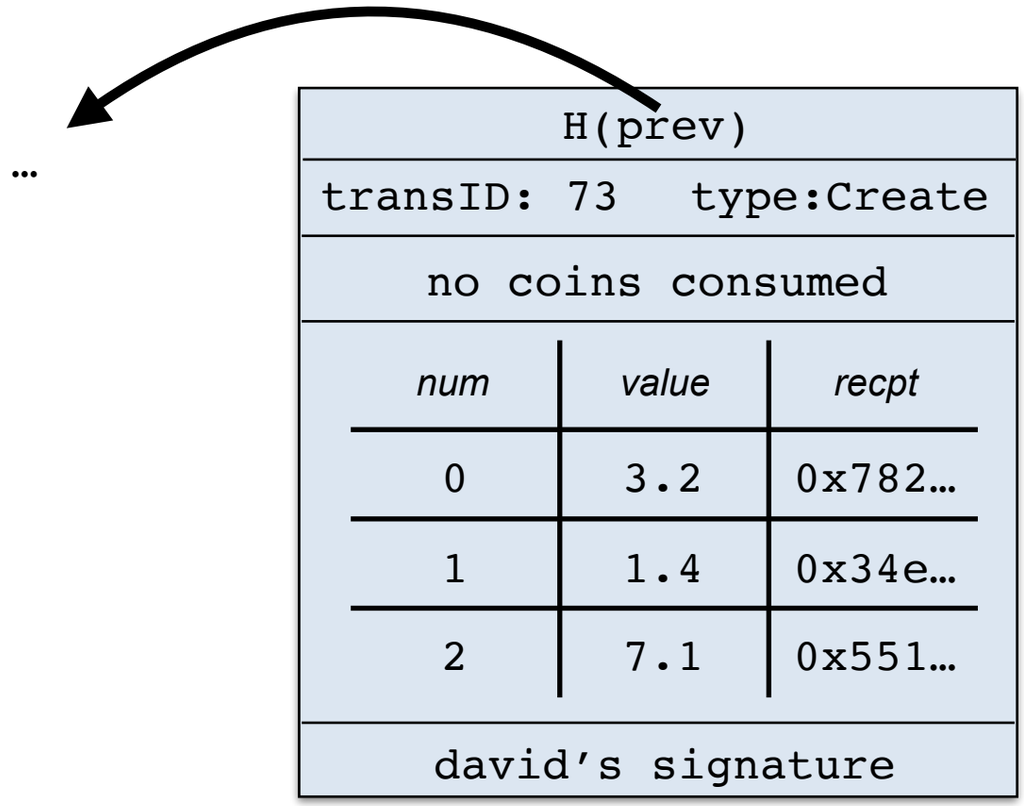
- One transaction per block
- Each transaction *consumes* some coins and *creates* new coins
- Each coin is created and consumed once. A created coin is *unspent* until it is consumed



- Two types of transactions: CreateCoins and PayCoins

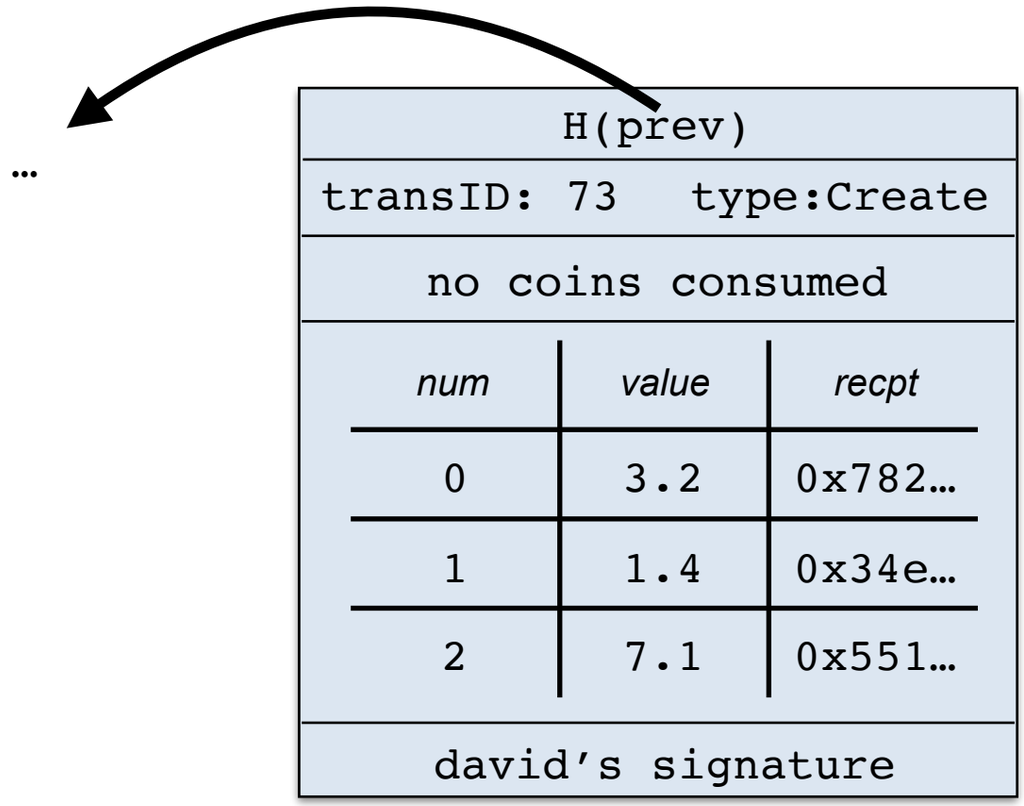
First Type of Transaction: CreateCoins

- CreateCoins transactions… create coins.
- Authority decides when to create coins and who gets them



First Type of Transaction: CreateCoins

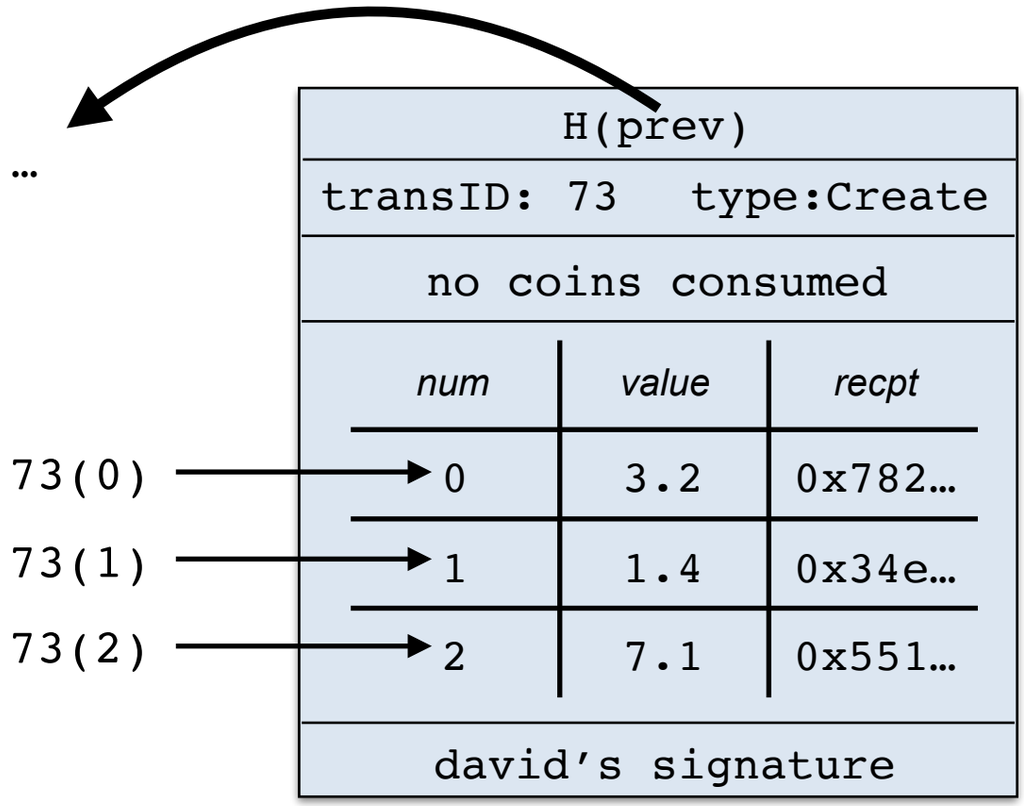
- CreateCoins transactions… create coins.
- Authority decides when to create coins and who gets them



- Every coin has a **`coinID`** consisting of **`transID`** and an index starting at zero.
  - **`coinIDs`** are unique and never reused

First Type of Transaction: CreateCoins

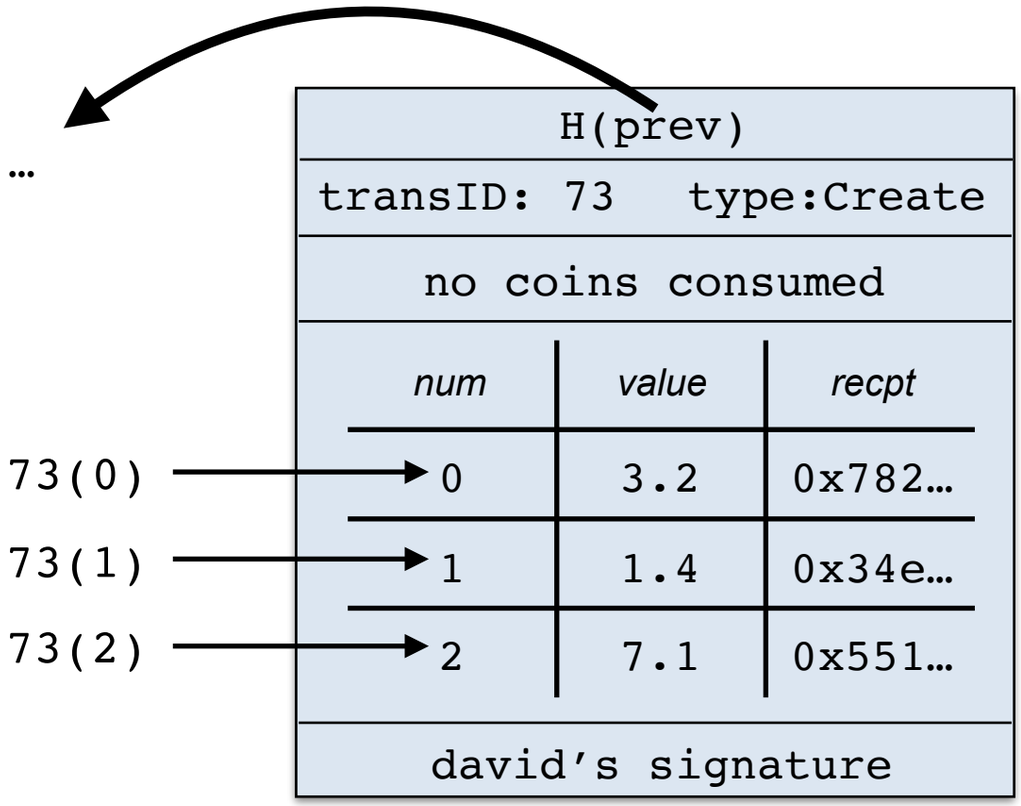
- CreateCoins transactions… create coins.
- Authority decides when to create coins and who gets them



- Every coin has a **`coinID`** consisting of **`transID`** and an index starting at zero.
  - **`coinIDs`** are unique and never reused

First Type of Transaction: CreateCoins

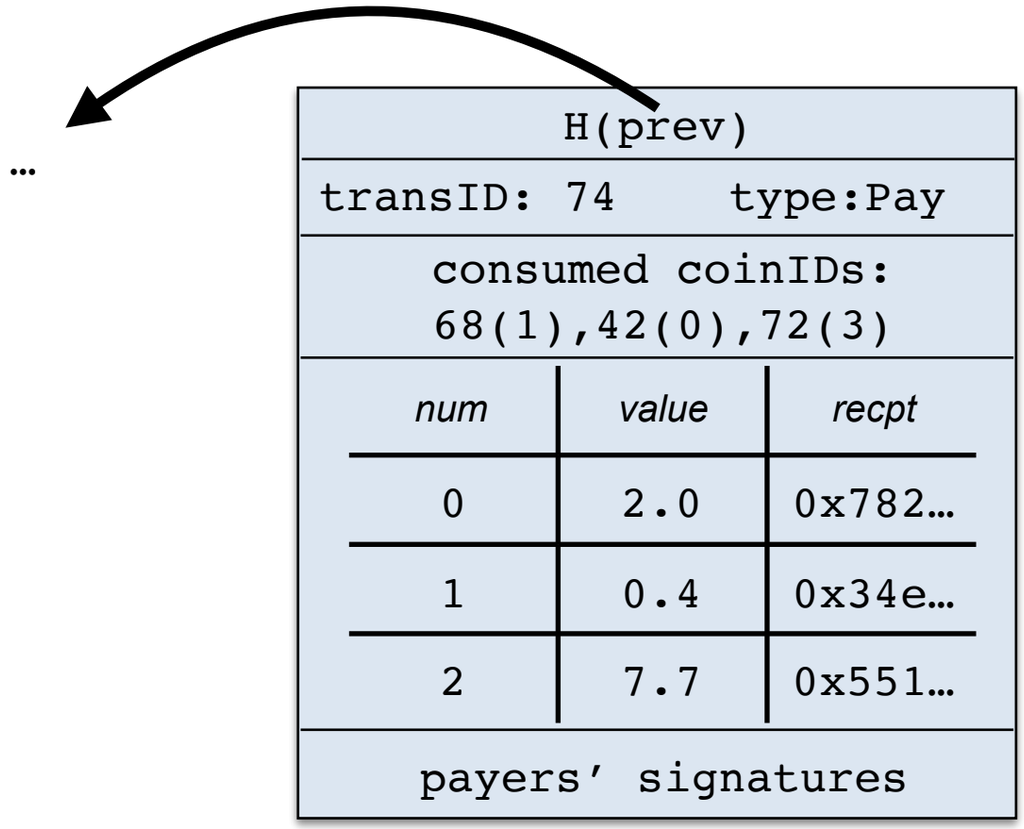
- CreateCoins transactions… create coins.
- Authority decides when to create coins and who gets them



- Every coin has a **`coinID`** consisting of **`transID`** and an index starting at zero.
  - **`coinIDs`** are unique and never reused
- Coins can have different values.

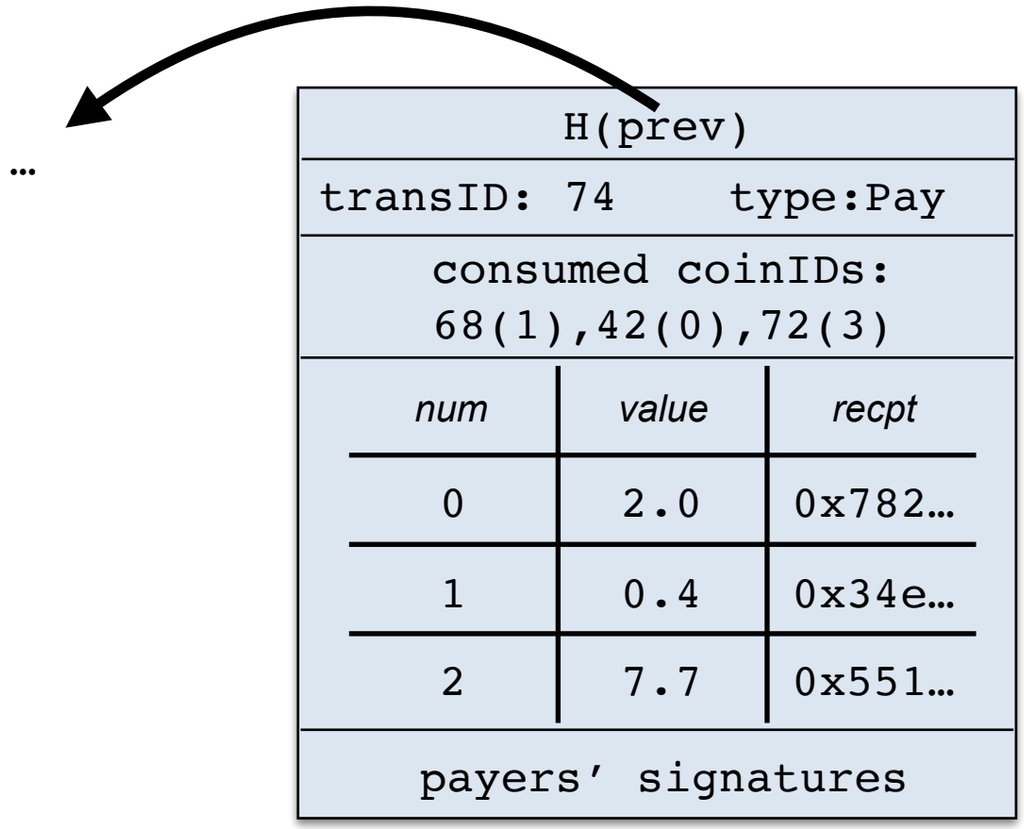
Second Type of Transaction: PayCoins

- PayCoins transactions consume some number of coins and create new coins owned by (potentially) different keys
- Payers must sign transaction



Second Type of Transaction: PayCoins

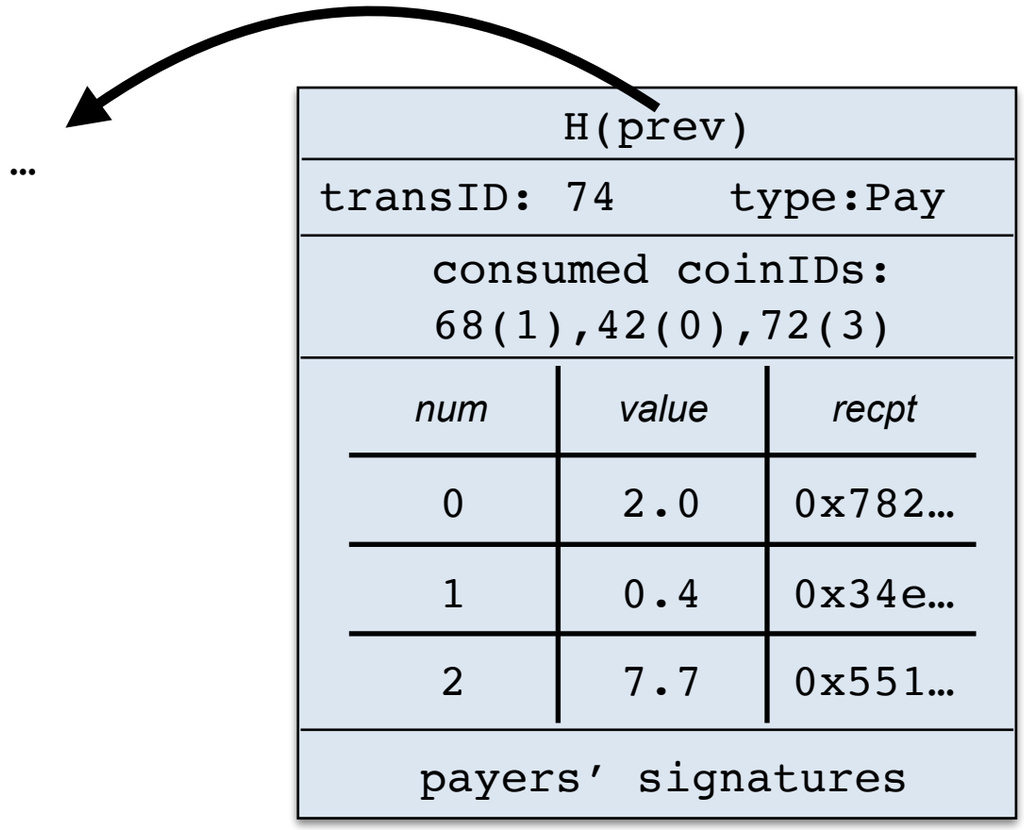
- PayCoins transactions consume some number of coins and create new coins owned by (potentially) different keys
- Payers must sign transaction



- Consumed coins should sum to value as created coins

Second Type of Transaction: PayCoins

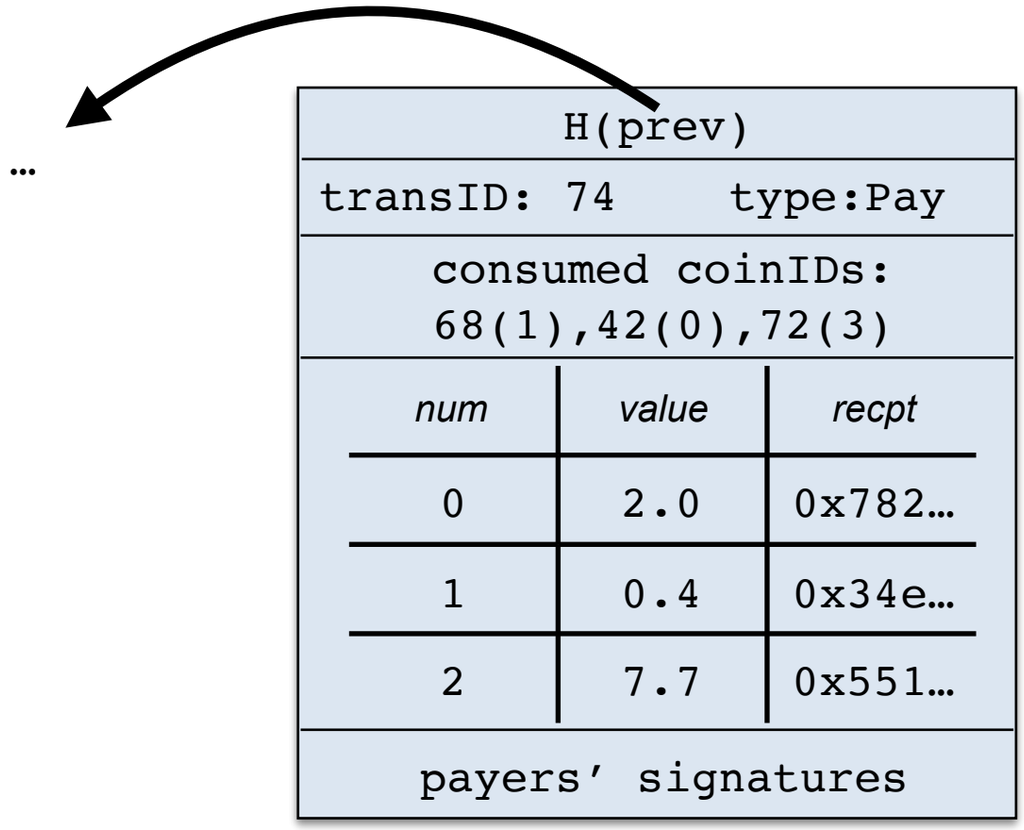
- PayCoins transactions consume some number of coins and create new coins owned by (potentially) different keys
- Payers must sign transaction



- Consumed coins should sum to value as created coins
- Intuitively, consumed coins `68(1),42(0),72(3)` are destroyed (melted down) and coins `74(0),74(1),74(2)` are newly created with possibly different owners.

Second Type of Transaction: PayCoins

- PayCoins transactions consume some number of coins and create new coins owned by (potentially) different keys
- Payers must sign transaction



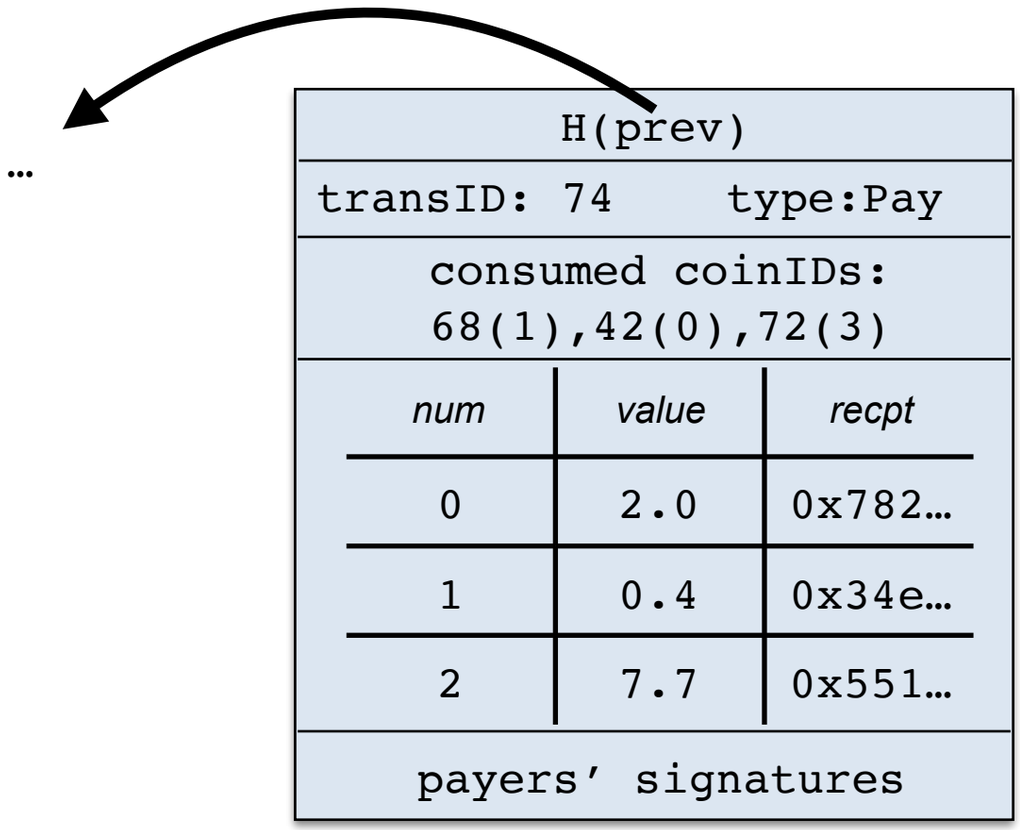
- Consumed coins should sum to value as created coins
- Intuitively, consumed coins `68(1),42(0),72(3)` are destroyed (melted down) and coins `74(0),74(1),74(2)` are newly created with possibly different owners.
- This is "transaction oriented": Each transaction says where its funds came from.

Valid transactions in DCash 2.0

- CreateCoins transactions are valid if the authority signed them and the hash matches the previous block

- PayCoins transactions are valid if:
  1. Consumed coins were indeed created previously
  2. Consumed coins have not been consumed in a previous block
  3. Consumed coins sum to same value as created created coins
  4. Signatures from payers are all valid
  5. Hash matches previous block

Quickly validating PayCoins Transactions



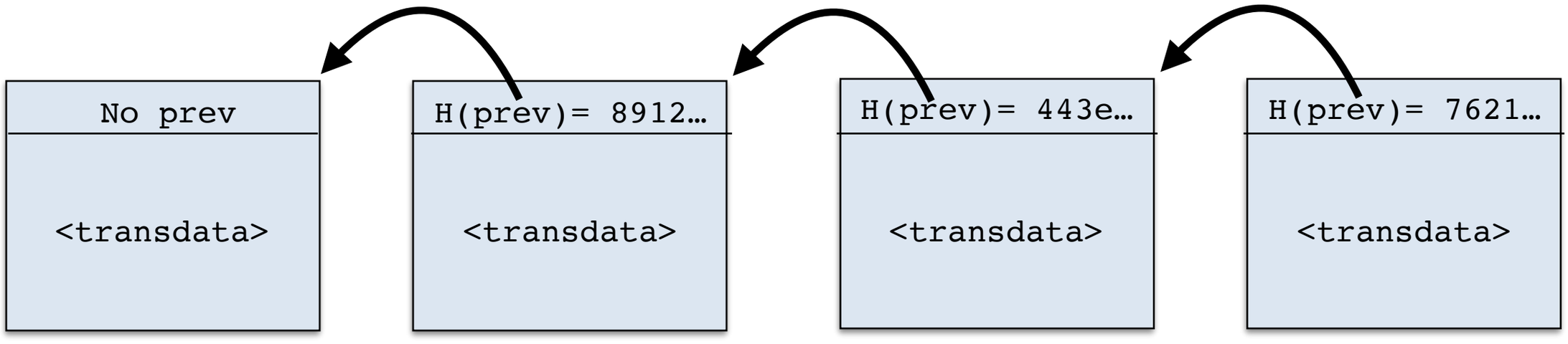
1. Step backwards and check if consumed coins were spent *since transaction 42*
2. Look up consumed coins, sum up their values, compare to output sum
3. Check sigs (using public keys of owners of consumed coin)

- This is easier than with our original ledger: We don't have to compute how much value each account has.

Tricks with PayCoins Transactions

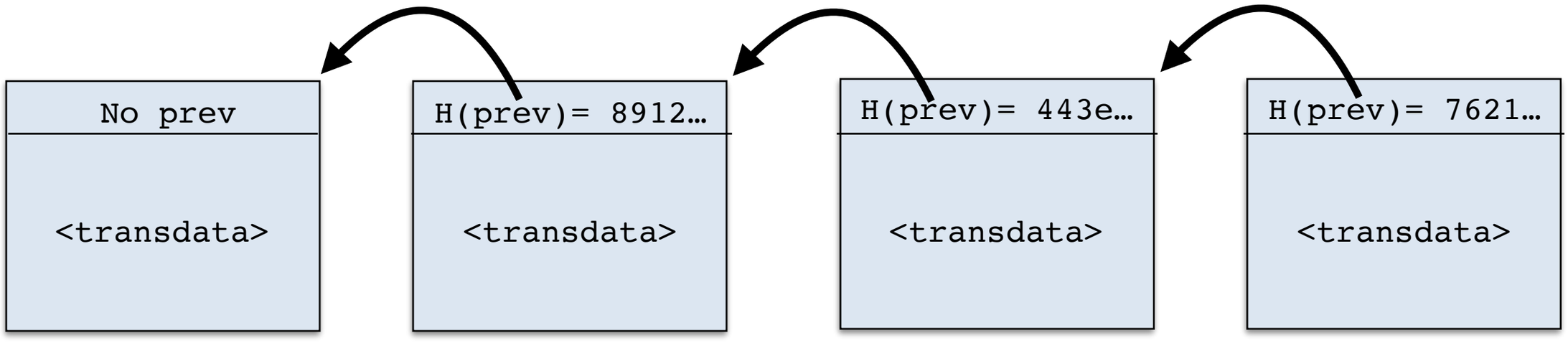
- Can consume a coin, pay someone, and pay yourself the change.
- Can split coins by paying yourself twice in one transaction.
- Easy to extend DCash 2.0 to allow multiple transactions in a block.

(Semi-) Trust in the DCash Authority



- Can the authority steal coins?

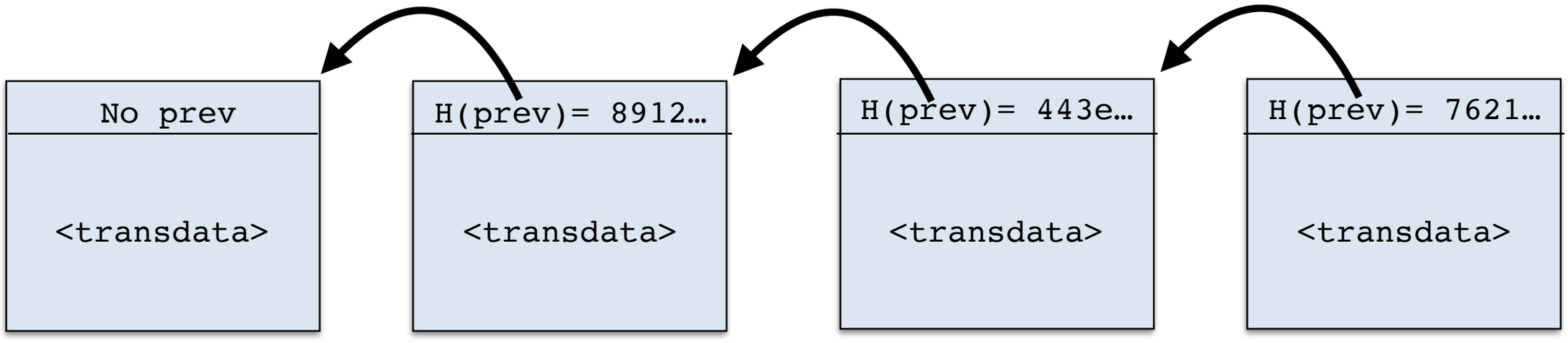
(Semi-) Trust in the DCash Authority



- Can the authority steal coins?

  No, this requires forging a signature.

(Semi-) Trust in the DCash Authority

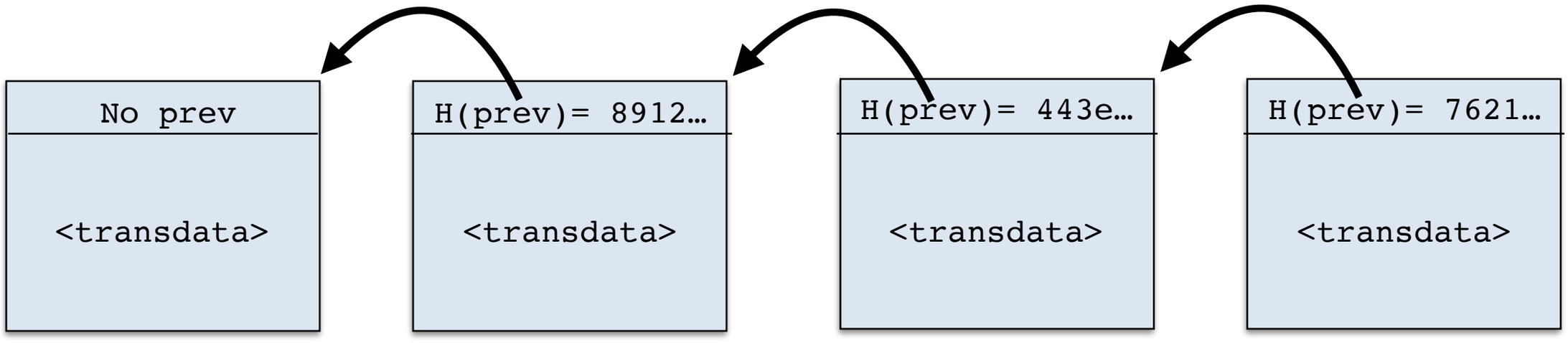


- Can the authority steal coins?

  No, this requires forging a signature.

- Can the authority delete transactions?

(Semi-) Trust in the DCash Authority



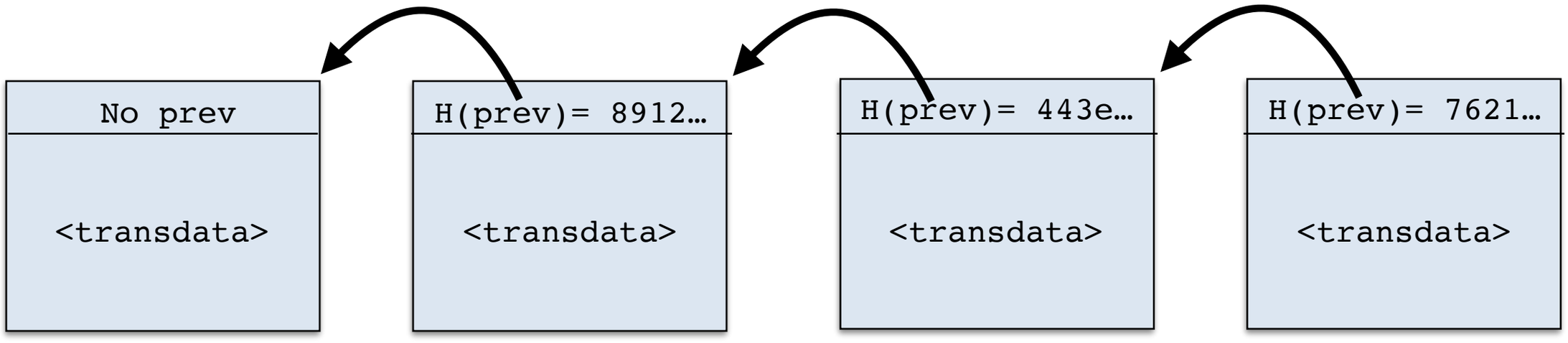
- Can the authority steal coins?

  No, this requires forging a signature.

- Can the authority delete transactions?

  No, this requires modifying the blockchain and will be detected by users.

(Semi-) Trust in the DCash Authority



- Can the authority steal coins?

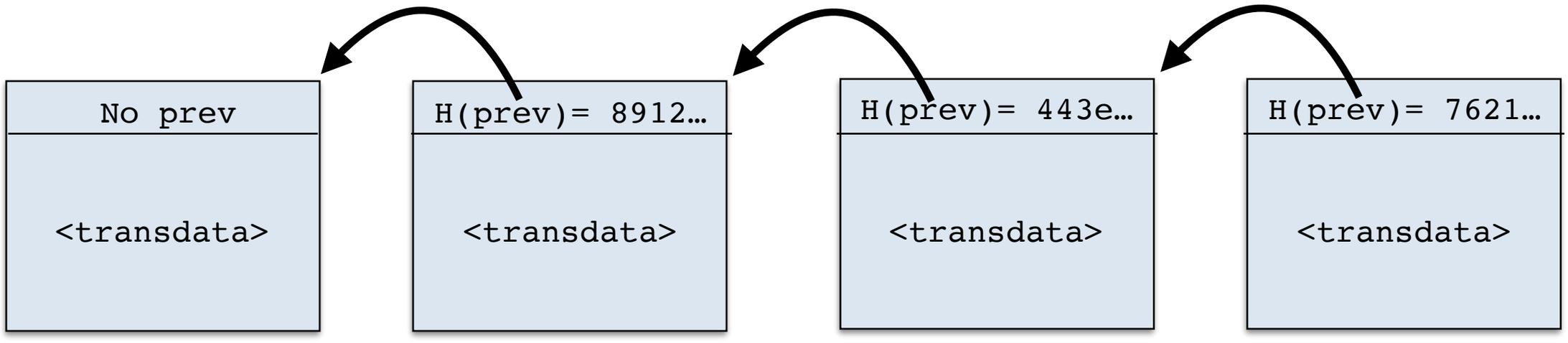
  No, this requires forging a signature.

- Can the authority delete transactions?

  No, this requires modifying the blockchain and will be detected by users.

But the authority *can*:

(Semi-) Trust in the DCash Authority



- Can the authority steal coins?

  No, this requires forging a signature.

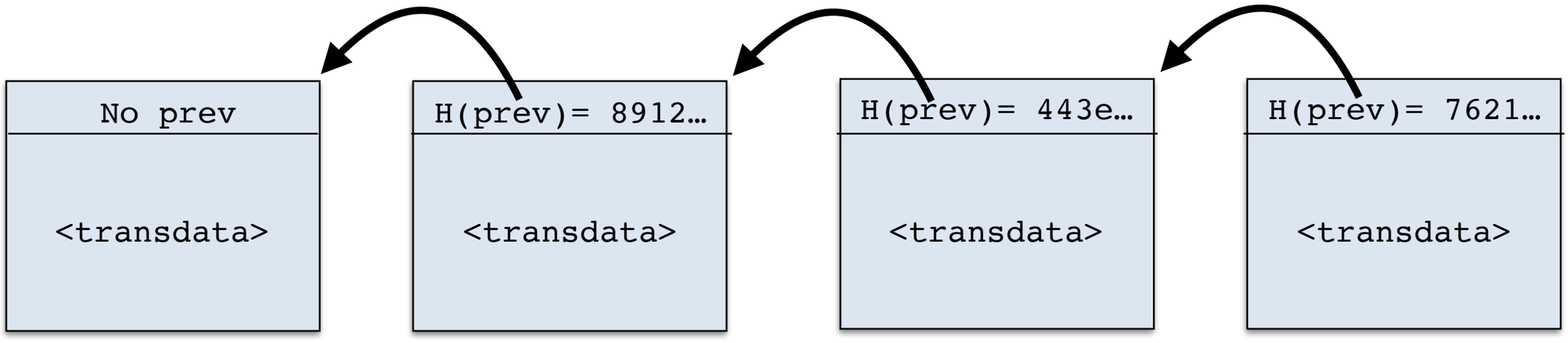
- Can the authority delete transactions?

  No, this requires modifying the blockchain and will be detected by users.

But the authority *can*:

1. Create as many coins as it likes.

(Semi-) Trust in the DCash Authority



- Can the authority steal coins?

  No, this requires forging a signature.

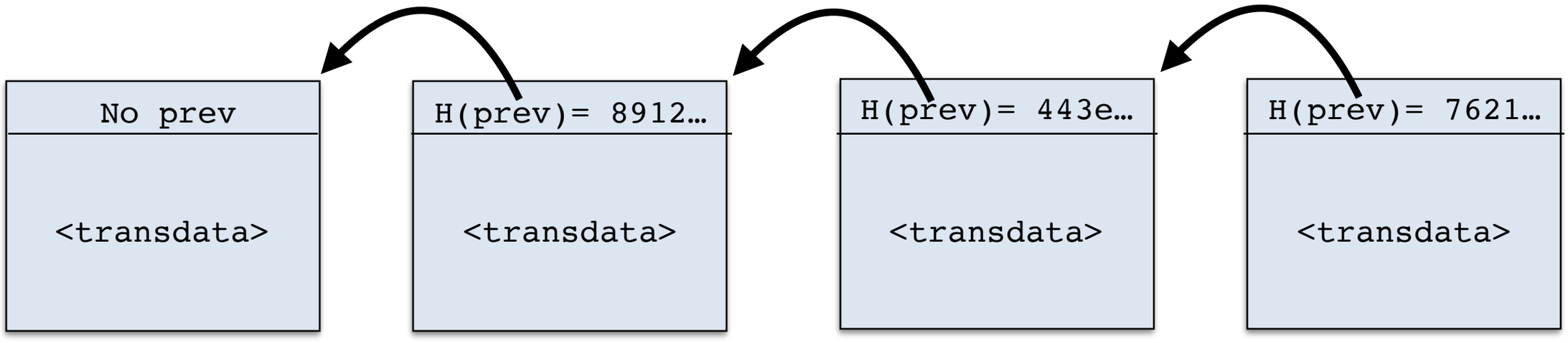
- Can the authority delete transactions?

  No, this requires modifying the blockchain and will be detected by users.

But the authority *can*:

1. Create as many coins as it likes.
2. Refuse transactions, locking people out or extorting them.

(Semi-) Trust in the DCash Authority



- Can the authority steal coins?

  No, this requires forging a signature.

- Can the authority delete transactions?

  No, this requires modifying the blockchain and will be detected by users.

But the authority *can*:

1. Create as many coins as it likes.
2. Refuse transactions, locking people out or extorting them.
3. Walk away from the entire affair, rendering all coins worthless.

Lecture 2 Outline

1. Cryptographic Hash Functions

   - Blockchains

   - Proofs of Work

2. Putting DCash “on the blockchain”, with an authority

3. The idea of decentralization

4. Decentralized DCash with an Angel

5. Decentralized DCash via proofs-of-work

Decentralized Technologies vs Centralized

Decentralized Technologies vs Centralized

Decentralized

1. The Internet
2. Email
3. The Web
4. Git

Decentralized Technologies vs Centralized

Decentralized	Centralized
1. The Internet	1. AOL
2. Email	2. Facebook
3. The Web	3. Piazza
4. Git	4. DNS

Decentralized Technologies vs Centralized

Decentralized

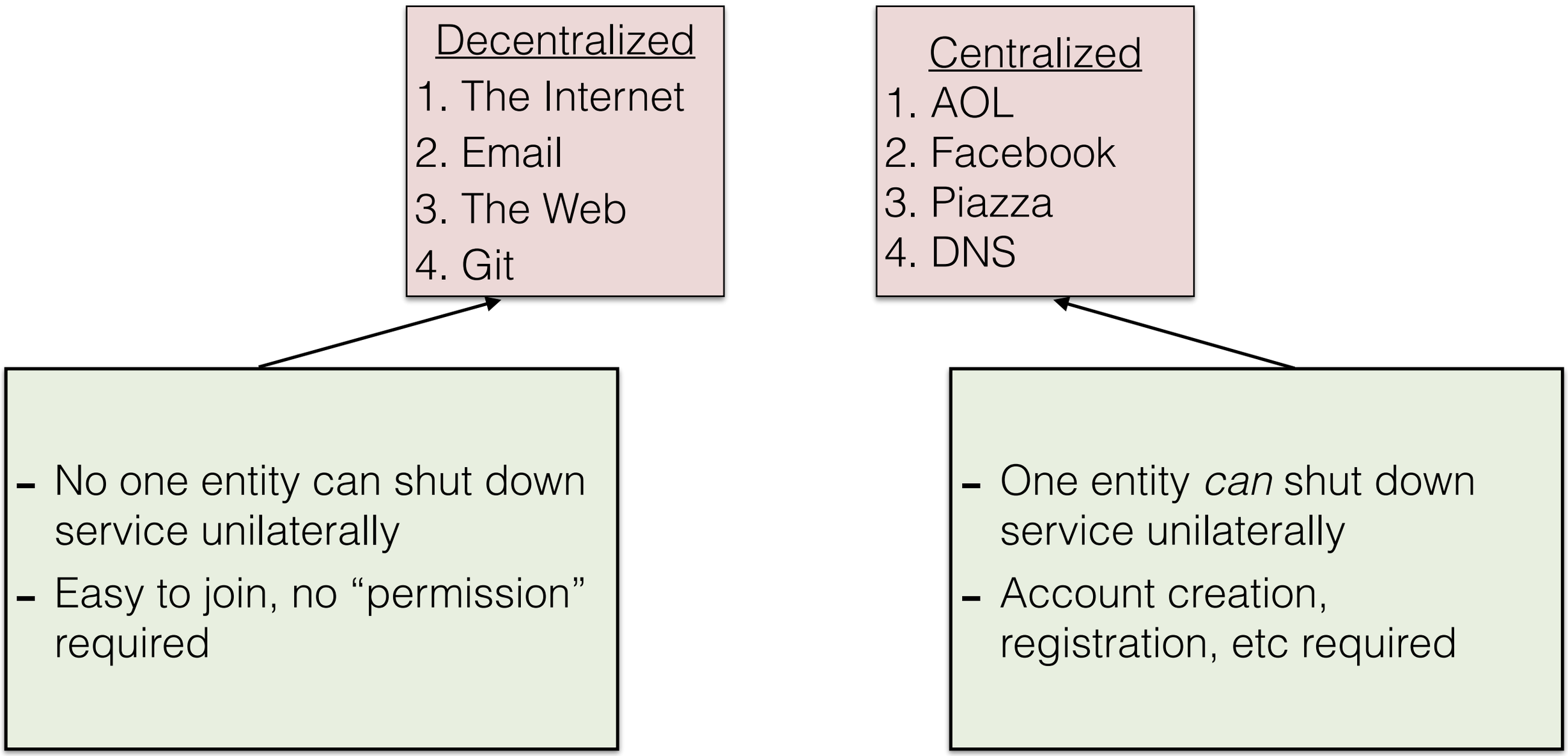
1. The Internet
2. Email
3. The Web
4. Git

Centralized

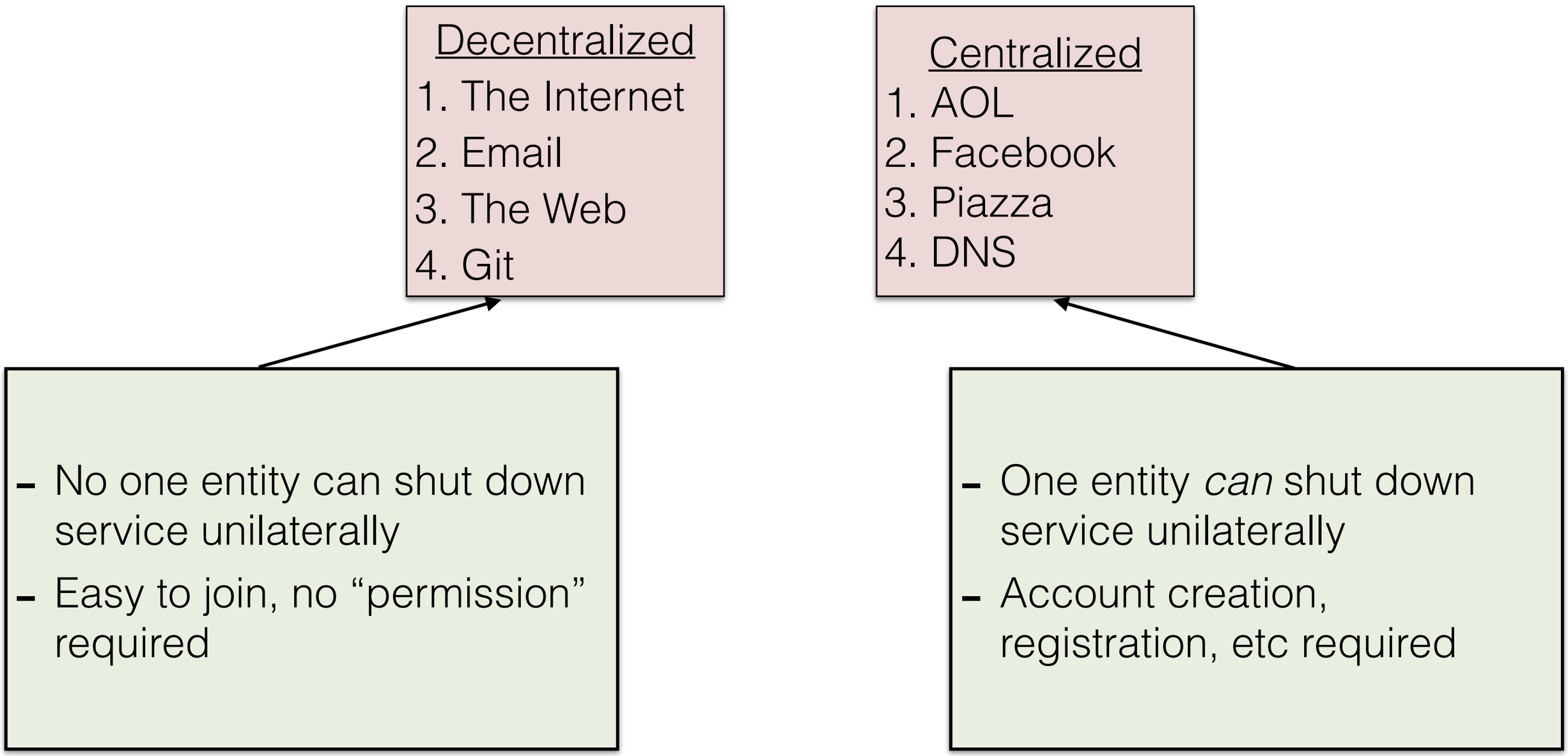
1. AOL
2. Facebook
3. Piazza
4. DNS

- No one entity can shut down service unilaterally
- Easy to join, no “permission” required

Decentralized Technologies vs Centralized



Decentralized Technologies vs Centralized



Note: A system may run on thousands of different computers yet still be centralized and controlled by one organization.

Decentralization as Studied in Computer Science

- Computer scientists of studied decentralized systems since the 1970's.
- The news was mostly bad: In many models it is impossible to distribute decisions
- The relevant problem for us called *distributed concensus*

Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

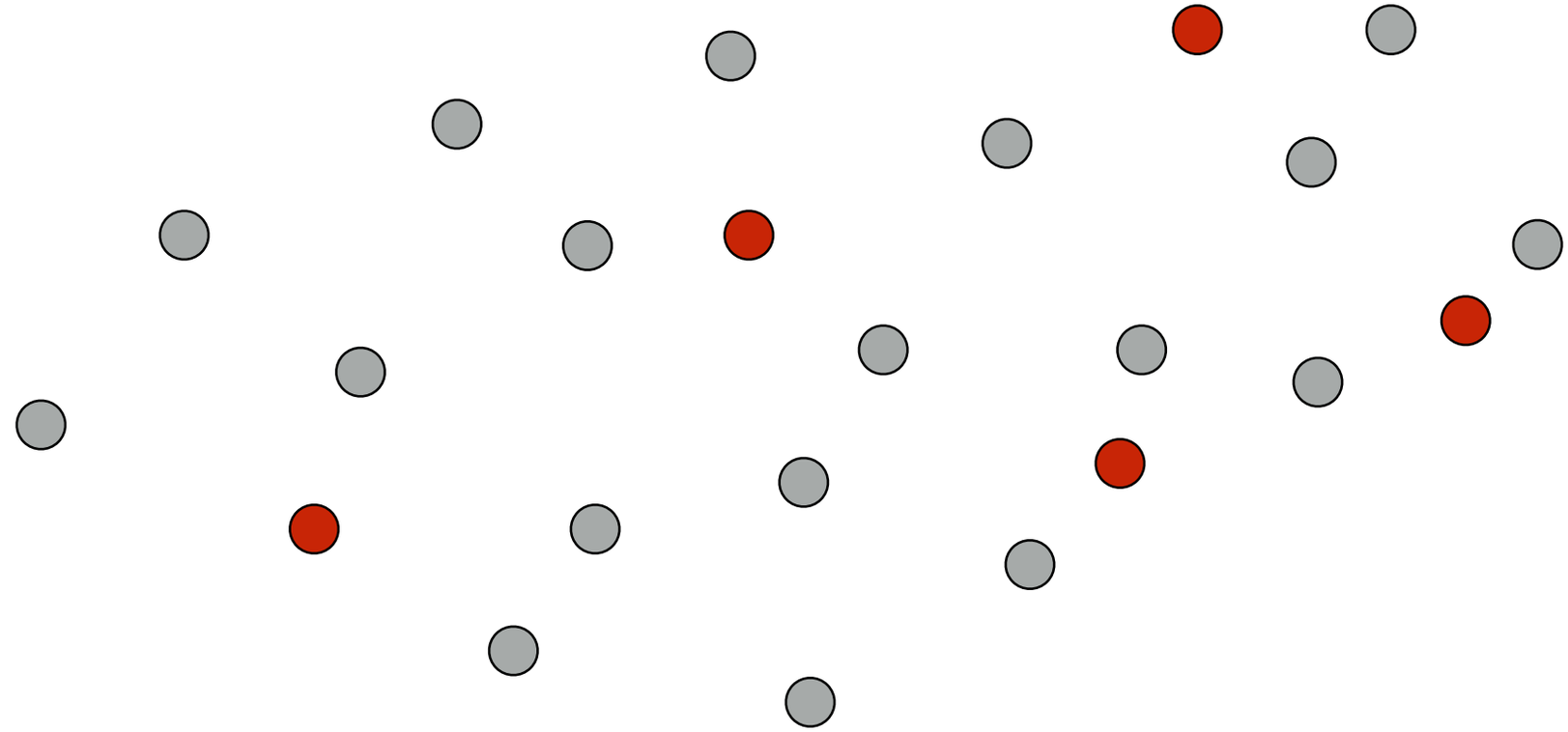
MICHAEL S. PATERSON

University of Warwick, Coventry, England

Abstract. The consensus problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable processes to agree on a binary value. In this paper, it is shown that every protocol for this problem has the possibility of nontermination, even with only one faulty process. By way of contrast, solutions are known for the synchronous case, the "Byzantine Generals" problem.

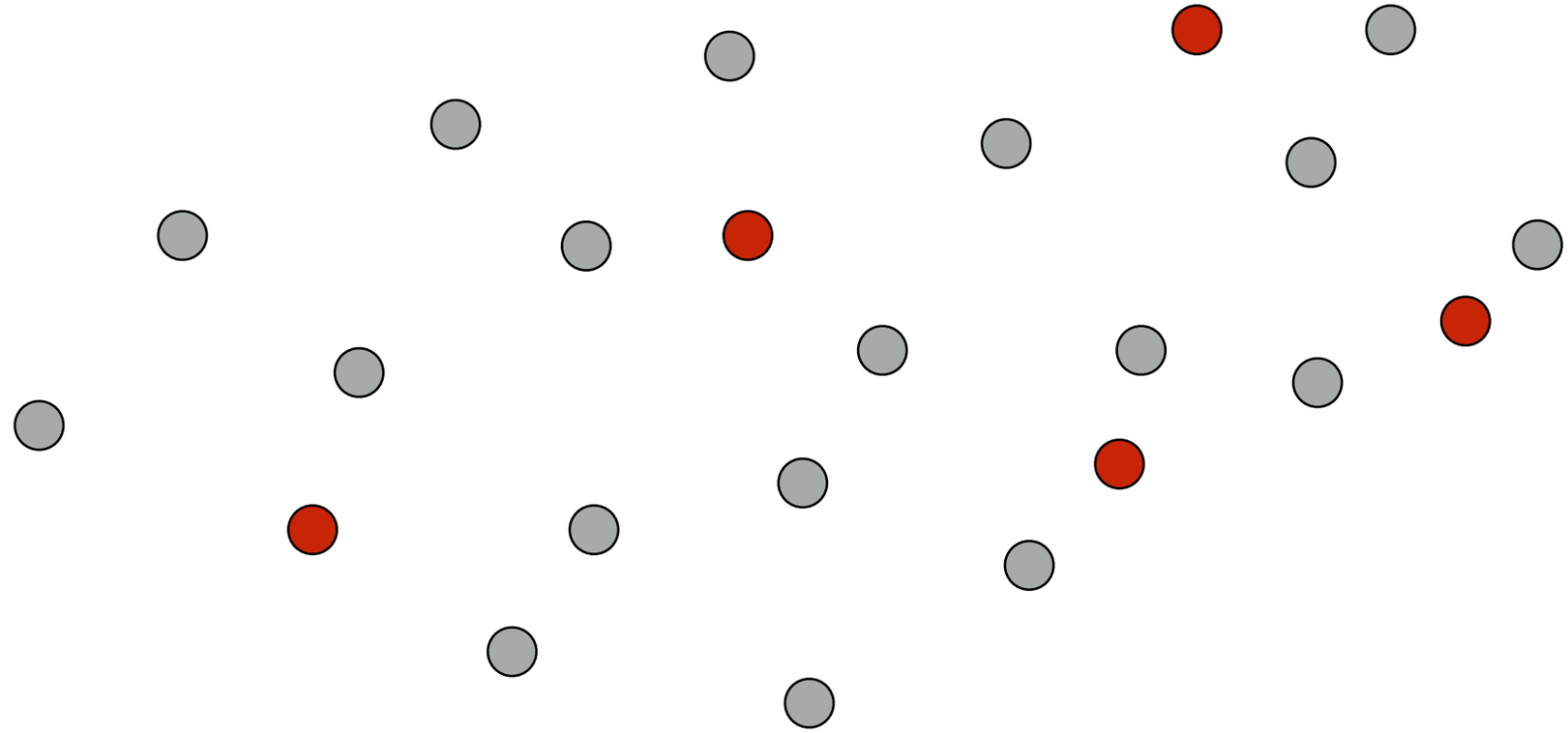
Distributed Consensus

- Several nodes communicate asynchronously
- Every node is either *honest* or *malicious*
- Honest nodes supply input values and follow specified protocol



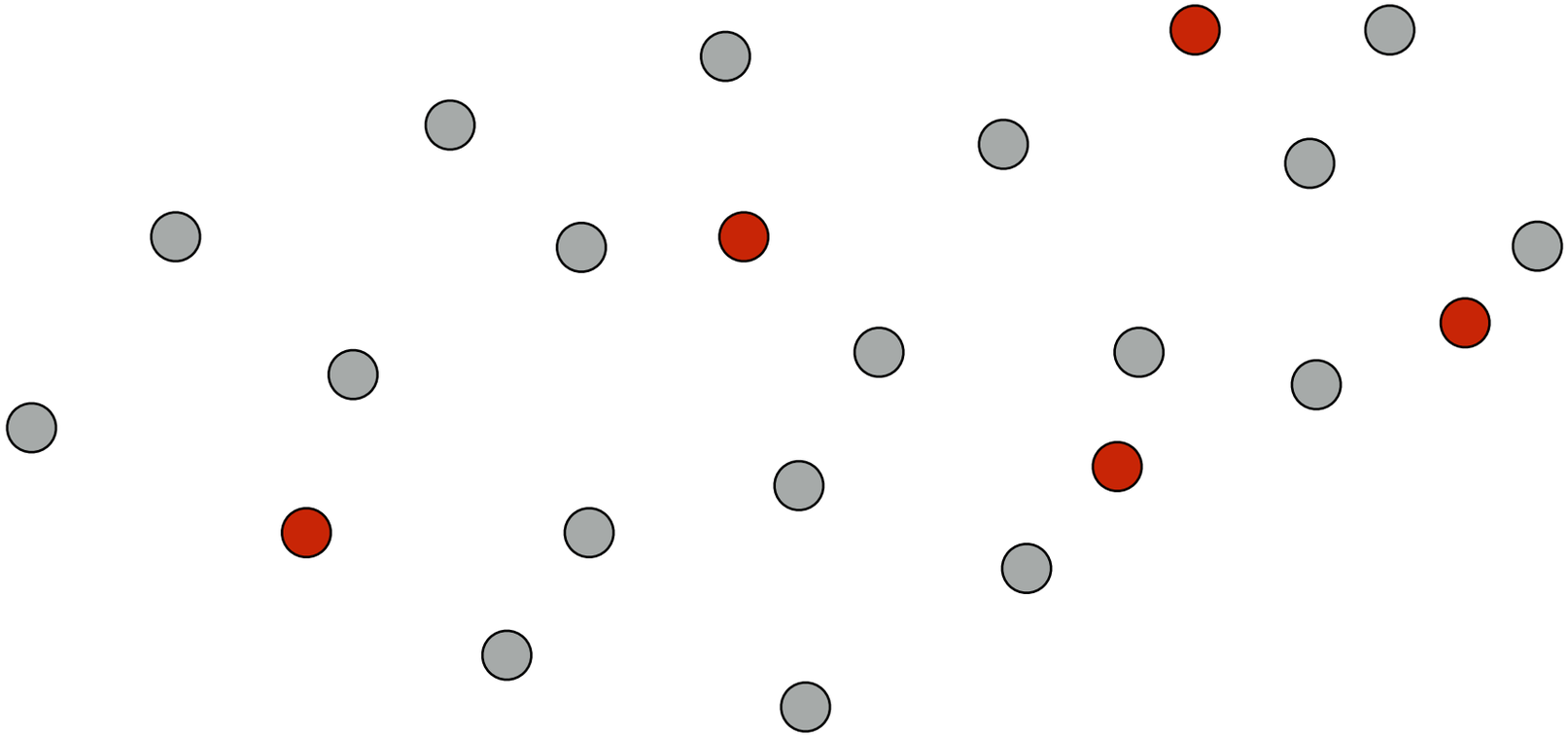
Distributed Consensus

- Several nodes communicate asynchronously
- Every node is either *honest* or *malicious*
- Honest nodes supply input values and follow specified protocol



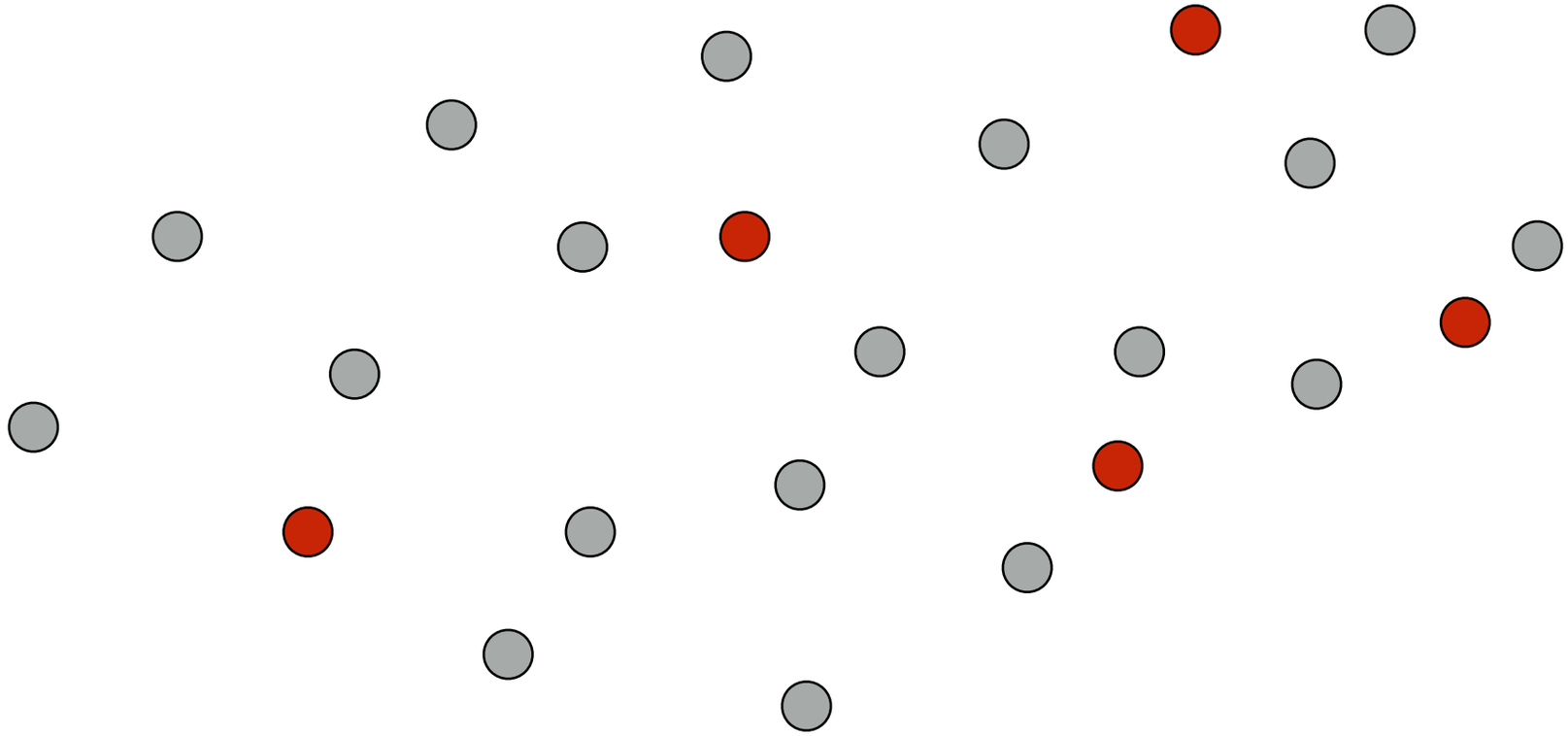
Distibuted consensus problem:
At start: All honest nodes have input value
At end: Protocol terminates and
1. All honest nodes agree on same value
2. The agreed-on value originated from an honest node.

Distributed Consensus of Blockchain State



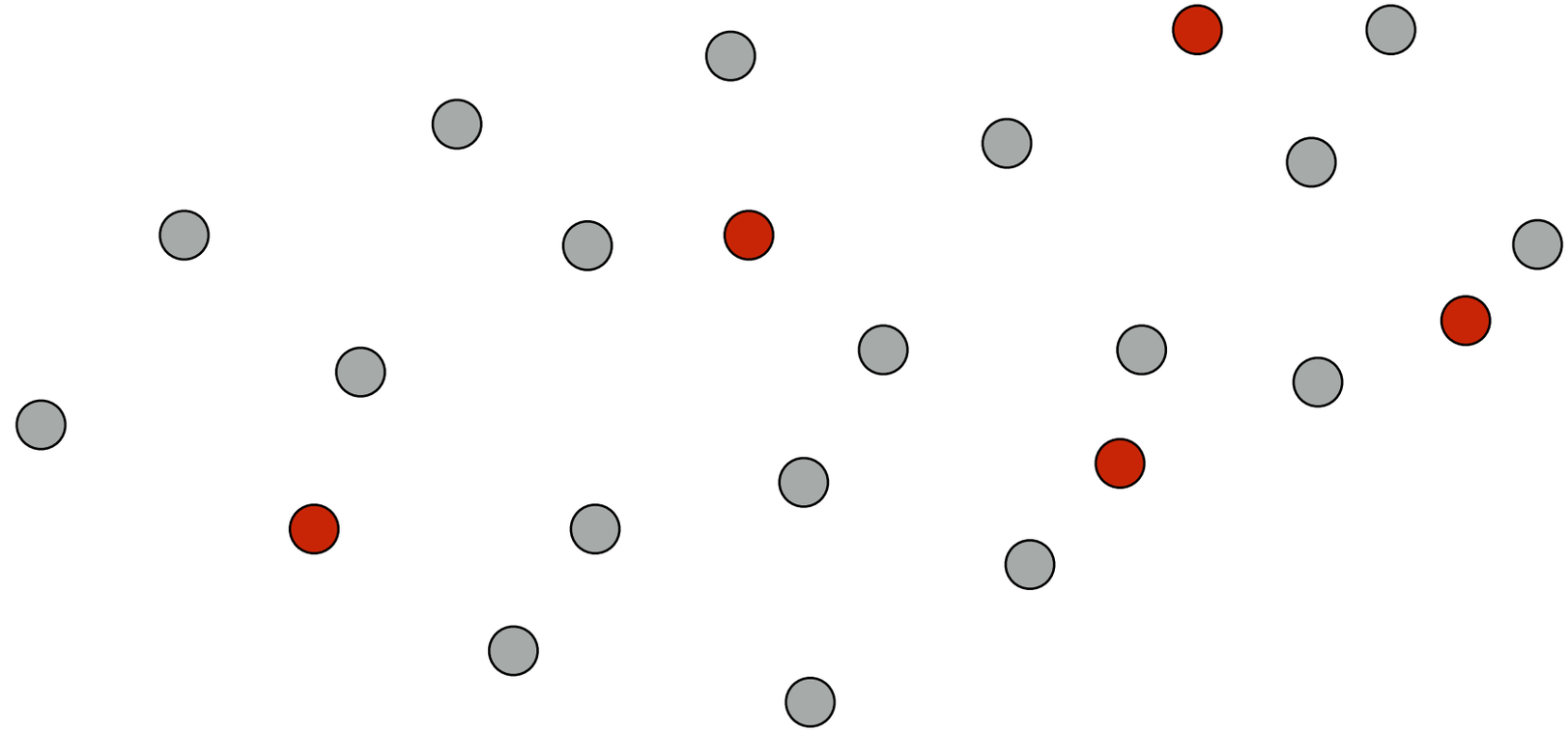
Distributed Consensus of Blockchain State

- Nodes connected to network broadcast transaction information



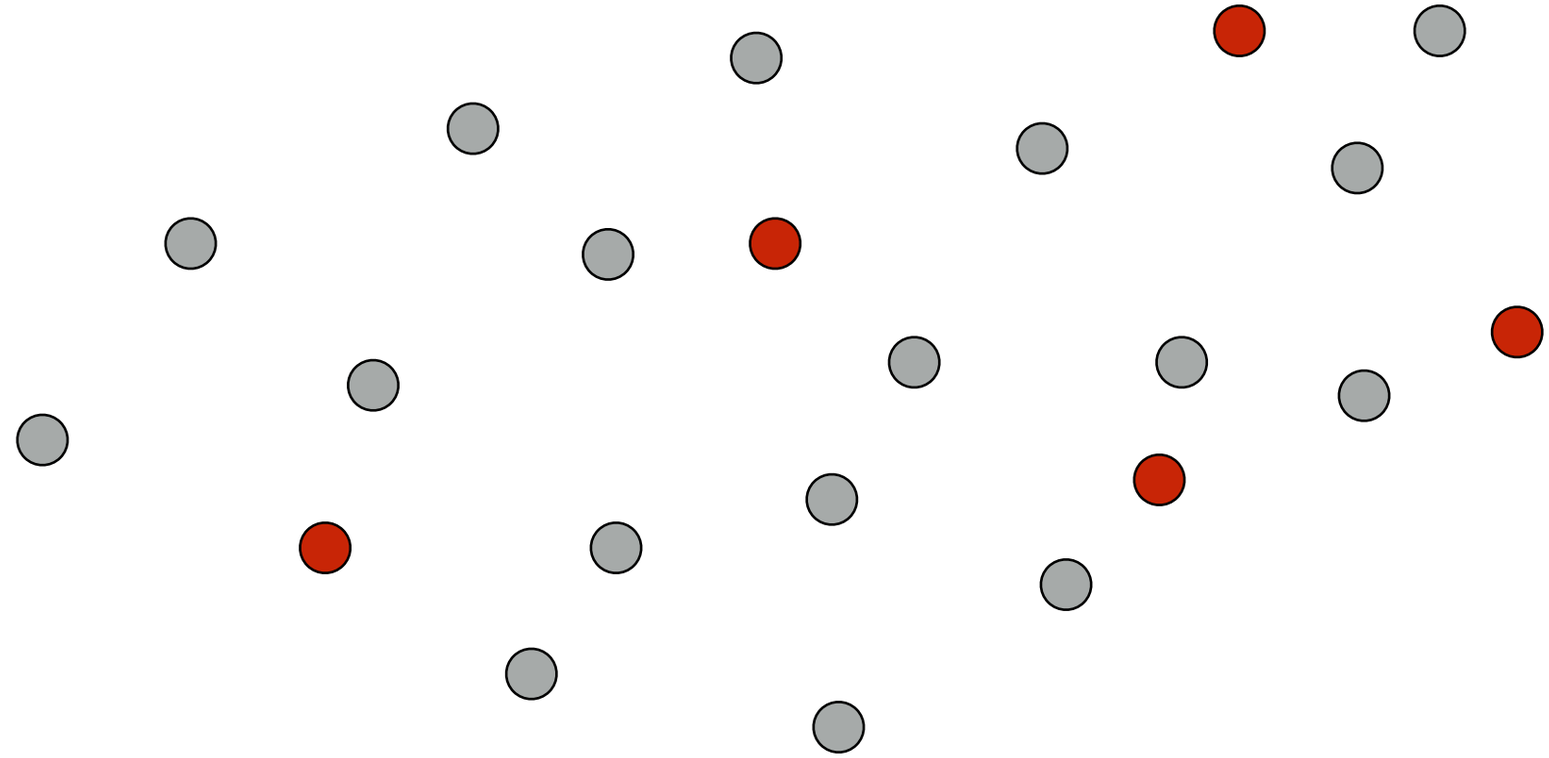
Distributed Consensus of Blockchain State

- Nodes connected to network broadcast transaction information
- Specified protocol says how to decide on the state of blockchain, but some nodes are malicious and don't follow protocol.



Distributed Consensus of Blockchain State

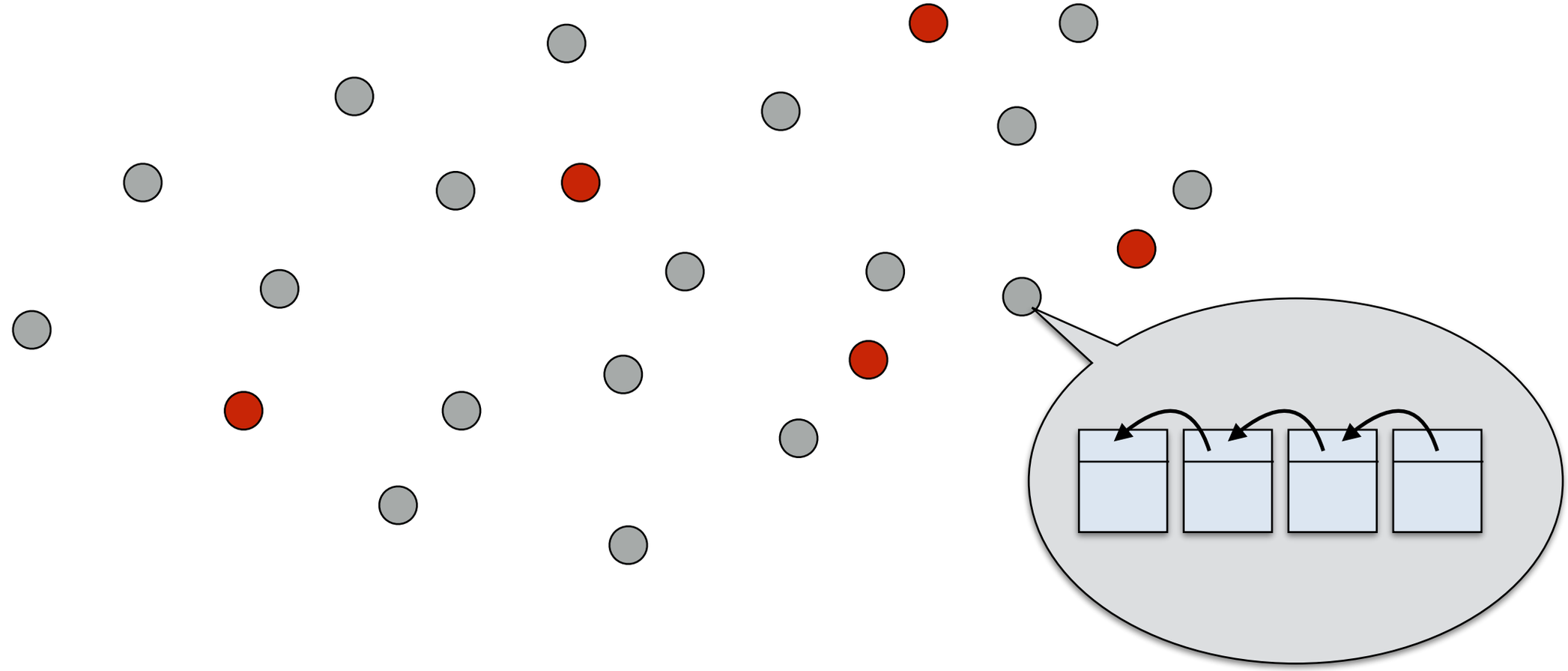
- Nodes connected to network broadcast transaction information
- Specified protocol says how to decide on the state of blockchain, but some nodes are malicious and don't follow protocol.



- Each honest node will have a view of the blockchain.

Distributed Consensus of Blockchain State

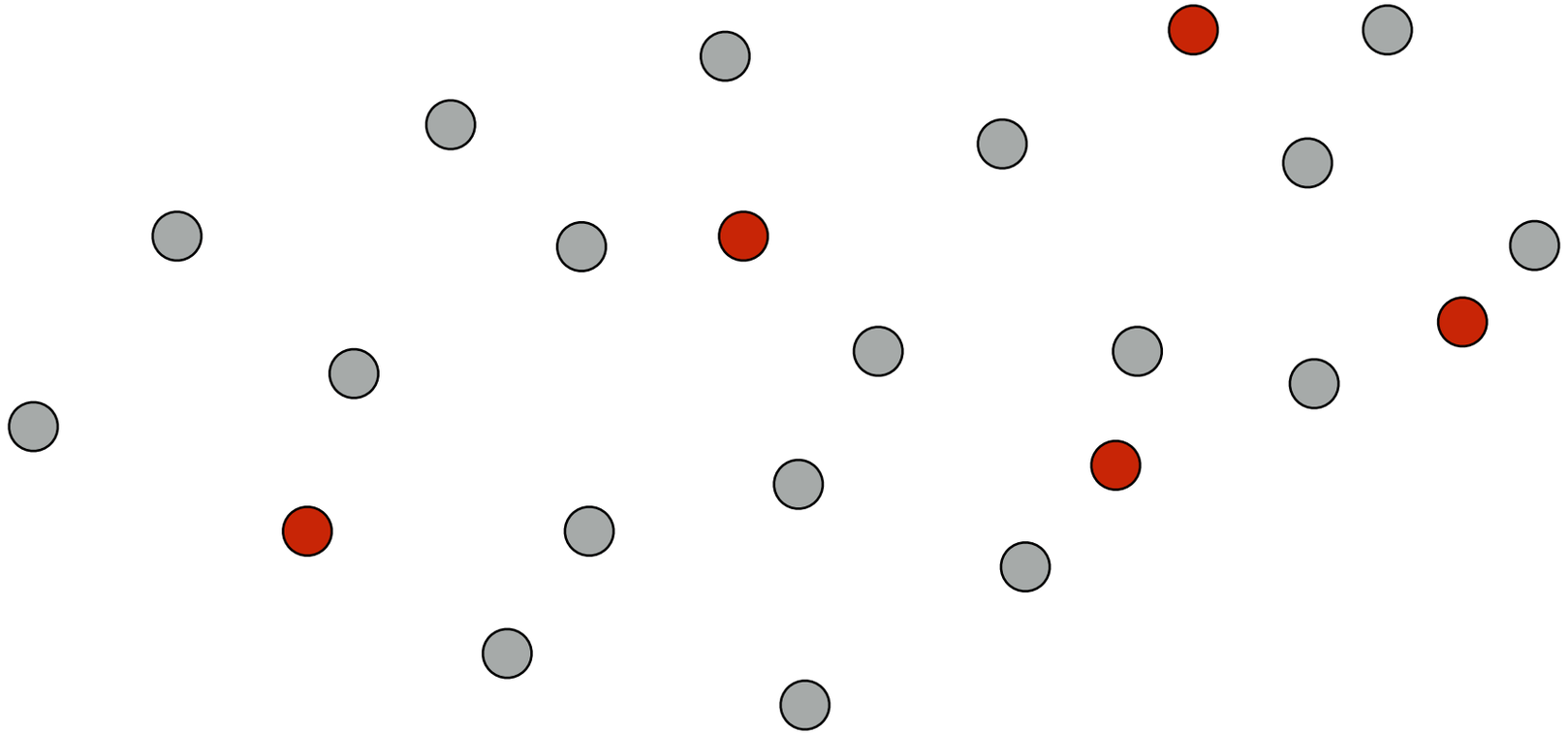
- Nodes connected to network broadcast transaction information
- Specified protocol says how to decide on the state of blockchain, but some nodes are malicious and don't follow protocol.



- Each honest node will have a view of the blockchain.

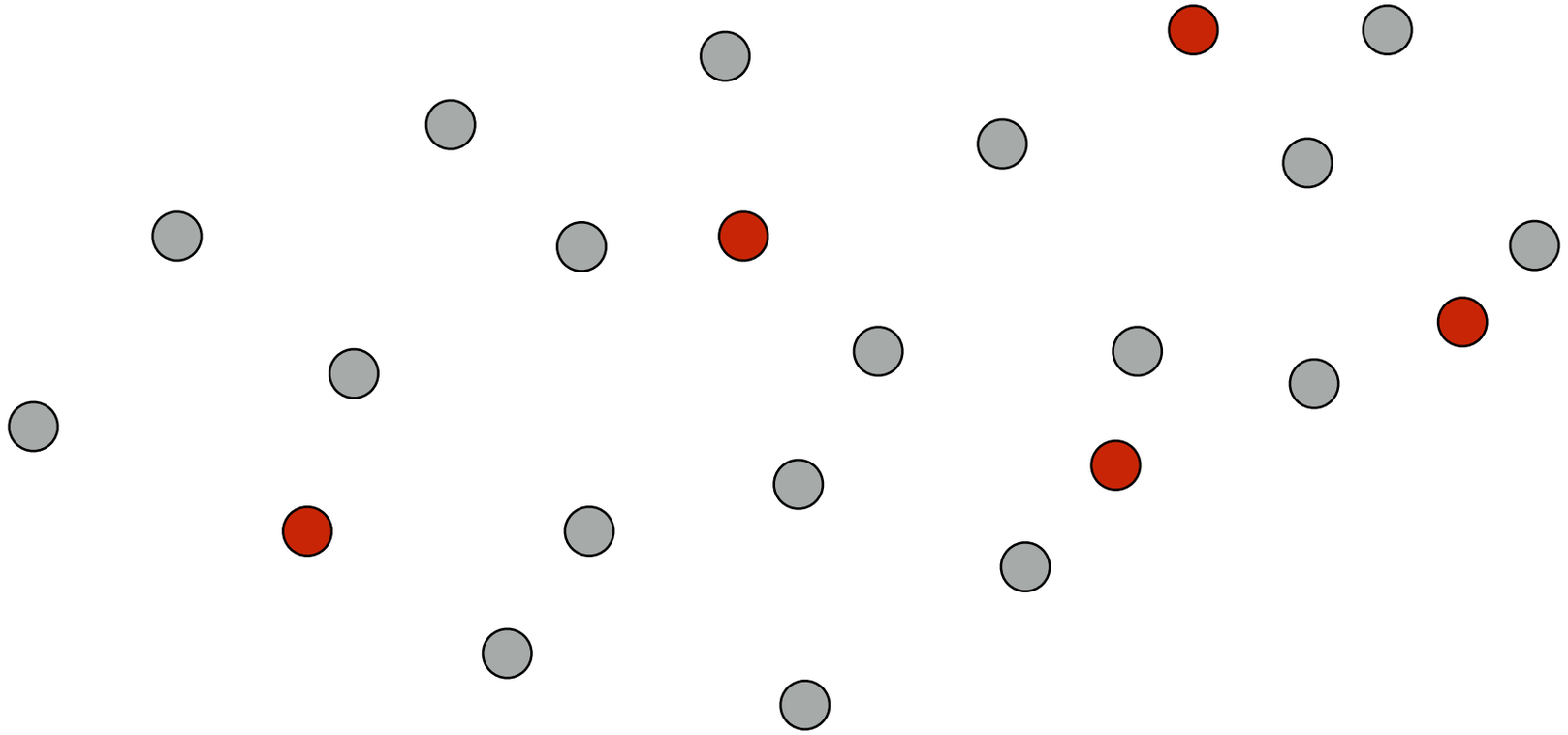
Identities and Sybil Attacks

- In Bitcoin, identities are not assigned by an authority. Anyone can create a node any time.



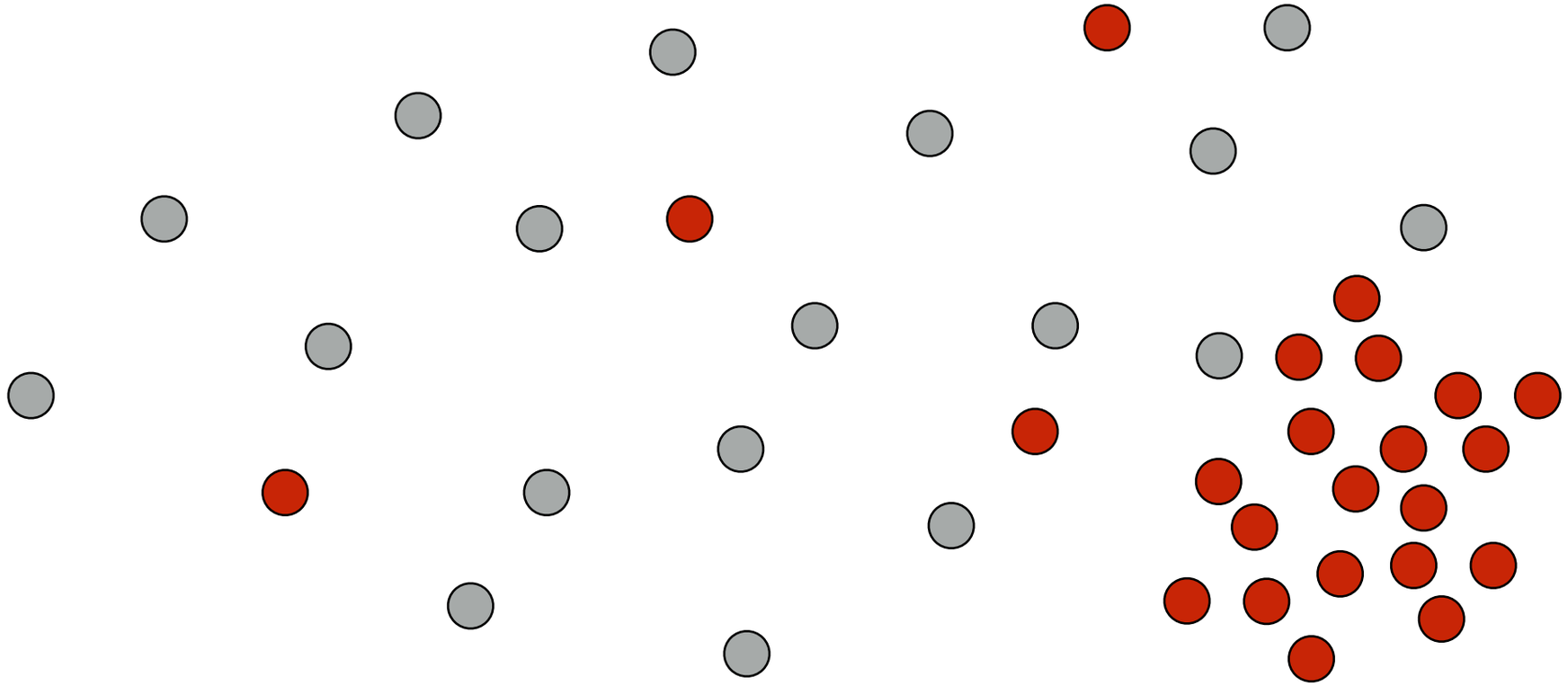
Identities and Sybil Attacks

- In Bitcoin, identities are not assigned by an authority. Anyone can create a node any time.
- It's easy for an adversary to create *tons* of nodes. This is called a *Sybil attack*.



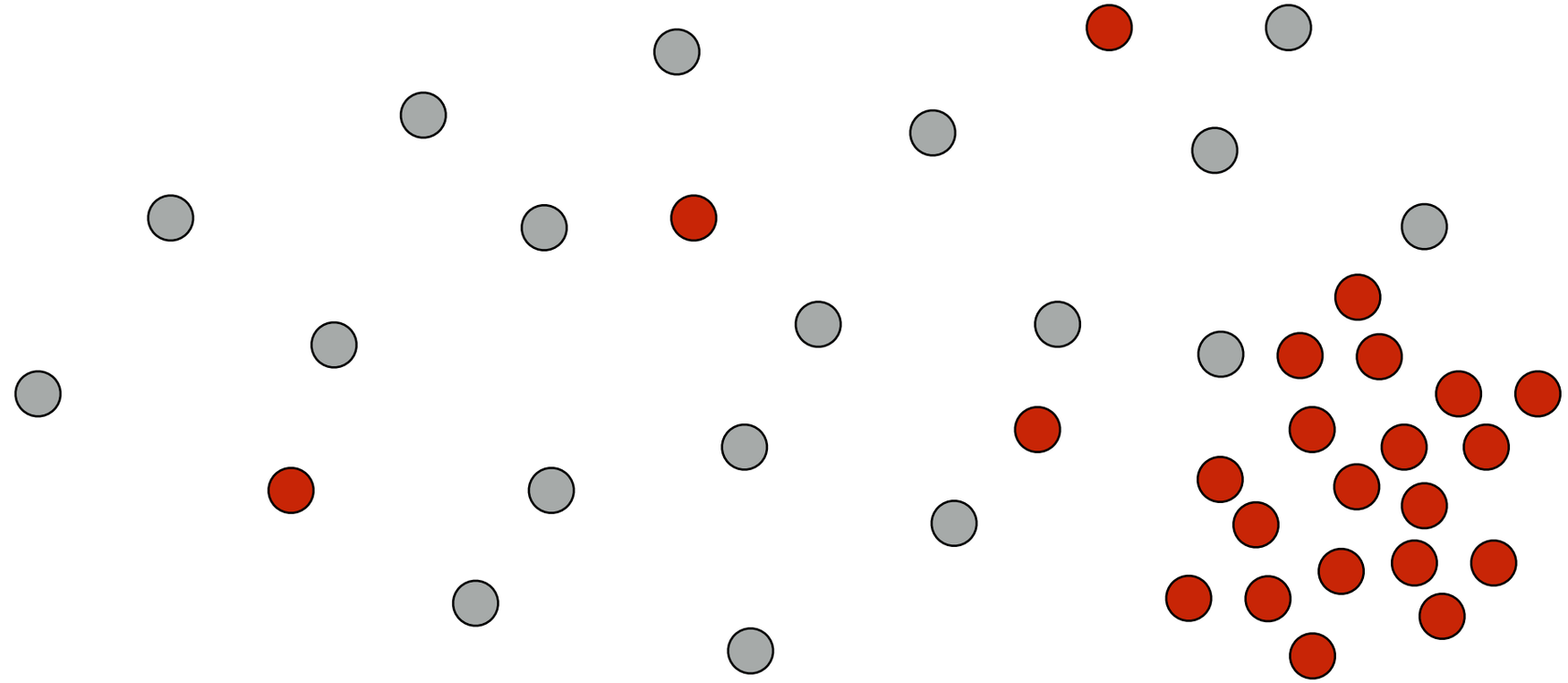
Identities and Sybil Attacks

- In Bitcoin, identities are not assigned by an authority. Anyone can create a node any time.
- It's easy for an adversary to create *tons* of nodes. This is called a *Sybil attack*.



Identities and Sybil Attacks

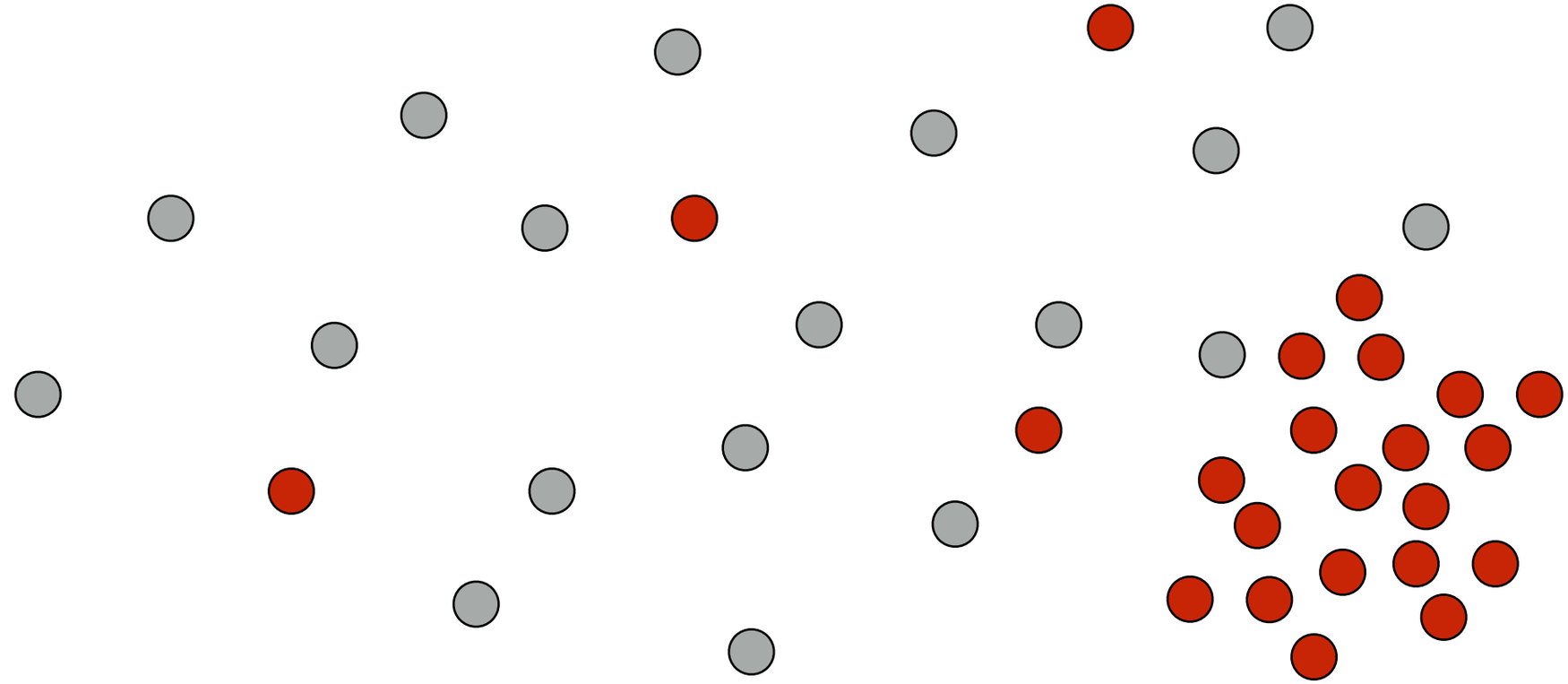
- In Bitcoin, identities are not assigned by an authority. Anyone can create a node any time.
- It's easy for an adversary to create *tons* of nodes. This is called a *Sybil attack*.



- Lesson: Without an authority to vet nodes, we can't count on malicious nodes being in the minority.

Identities and Sybil Attacks

- In Bitcoin, identities are not assigned by an authority. Anyone can create a node any time.
- It's easy for an adversary to create *tons* of nodes. This is called a *Sybil attack*.



- Lesson: Without an authority to vet nodes, we can't count on malicious nodes being in the minority.

Goal: Have all honest nodes agree on one of their views of the blockchain.

Lecture 2 Outline

1. Cryptographic Hash Functions
   - Blockchains
   - Proofs of Work
2. Putting DCash “on the blockchain”, with an authority
3. The idea of decentralization
4. **Decentralized DCash with an Angel**
5. Decentralized DCash via proofs-of-work

DCash 3.0: Consensus with an angel (text section 2.3)

- In this protocol, time is divided into rounds.
- Every node maintains its own personal “view” of the blockchain.
- We start with an unrealistic model that depends on an imaginary *angel*.

DCash 3.0: Consensus with an angel (text section 2.3)

- In this protocol, time is divided into rounds.
- Every node maintains its own personal "view" of the blockchain.
- We start with an unrealistic model that depends on an imaginary *angel*.

Angel's magic power: Able to select a random node from the network and announce its identity to everyone. The chosen node can add a block to its personal view. Nobody else can add a block to their view.

Exception: Malicious nodes are a hive-mind and share one personal view.

DCash 3.0: Consensus with an angel (text section 2.3)

- In this protocol, time is divided into rounds.
- Every node maintains its own personal "view" of the blockchain.
- We start with an unrealistic model that depends on an imaginary *angel*.

Angel's magic power: Able to select a random node from the network and announce its identity to everyone. The chosen node can add a block to its personal view. Nobody else can add a block to their view.

Exception: Malicious nodes are a hive-mind and share one personal view.

- At the end of each round, our protocol will use angel to choose a random leader.

DCash 3.0: Consensus with an angel (text section 2.3)

- In this protocol, time is divided into rounds.
- Every node maintains its own personal "view" of the blockchain.
- We start with an unrealistic model that depends on an imaginary *angel*.

Angel's magic power: Able to select a random node from the network and announce its identity to everyone. The chosen node can add a block to its personal view. Nobody else can add a block to their view.

Exception: Malicious nodes are a hive-mind and share one personal view.

- At the end of each round, our protocol will use angel to choose a random leader.

This protocol doesn't exist in reality! We will build the angel and personal views using POWs later.

DCash 3.0: Consensus with an angel (text section 2.3)

DCash 3.0: Consensus with an angel (text section 2.3)

Consensus protocol with angel:

DCash 3.0: Consensus with an angel (text section 2.3)

Consensus protocol with angel:

1. Transactions are broadcast to everyone

DCash 3.0: Consensus with an angel (text section 2.3)

Consensus protocol with angel:

1. Transactions are broadcast to everyone
2. Each node collects transactions into a block

DCash 3.0: Consensus with an angel (text section 2.3)

Consensus protocol with angel:

1. Transactions are broadcast to everyone
2. Each node collects transactions into a block
3. At end of round, angel picks the leader

DCash 3.0: Consensus with an angel (text section 2.3)

Consensus protocol with angel:

1. Transactions are broadcast to everyone
2. Each node collects transactions into a block
3. At end of round, angel picks the leader
4. The leader adds a block of valid transactions to their personal view.

DCash 3.0: Consensus with an angel (text section 2.3)

Consensus protocol with angel:

1. Transactions are broadcast to everyone
2. Each node collects transactions into a block
3. At end of round, angel picks the leader
4. The leader adds a block of valid transactions to their personal view.
5. The leader announces their personal view of the blockchain.

DCash 3.0: Consensus with an angel (text section 2.3)

Consensus protocol with angel:

1. Transactions are broadcast to everyone
2. Each node collects transactions into a block
3. At end of round, angel picks the leader
4. The leader adds a block of valid transactions to their personal view.
5. The leader announces their personal view of the blockchain.
6. Everyone else accepts the announced view as their own if:

DCash 3.0: Consensus with an angel (text section 2.3)

Consensus protocol with angel:

1. Transactions are broadcast to everyone
2. Each node collects transactions into a block
3. At end of round, angel picks the leader
4. The leader adds a block of valid transactions to their personal view.
5. The leader announces their personal view of the blockchain.
6. Everyone else accepts the announced view as their own if:
    a. The transactions in this view are all valid

DCash 3.0: Consensus with an angel (text section 2.3)

Consensus protocol with angel:

1. Transactions are broadcast to everyone
2. Each node collects transactions into a block
3. At end of round, angel picks the leader
4. The leader adds a block of valid transactions to their personal view.
5. The leader announces their personal view of the blockchain.
6. Everyone else accepts the announced view as their own if:
   a. The transactions in this view are all valid
   b. The announced chain has **more blocks** than their view.

DCash 3.0: Consensus with an angel (text section 2.3)

Consensus protocol with angel:

1. Transactions are broadcast to everyone
2. Each node collects transactions into a block
3. At end of round, angel picks the leader
4. The leader adds a block of valid transactions to their personal view.
5. The leader announces their personal view of the blockchain.
6. Everyone else accepts the announced view as their own if:
   a. The transactions in this view are all valid
   b. The announced chain has **more blocks** than their view.

Note 1: When a node is chosen, it can always add a block to its view. It's up to the other nodes to accept it or not.

DCash 3.0: Consensus with an angel (text section 2.3)

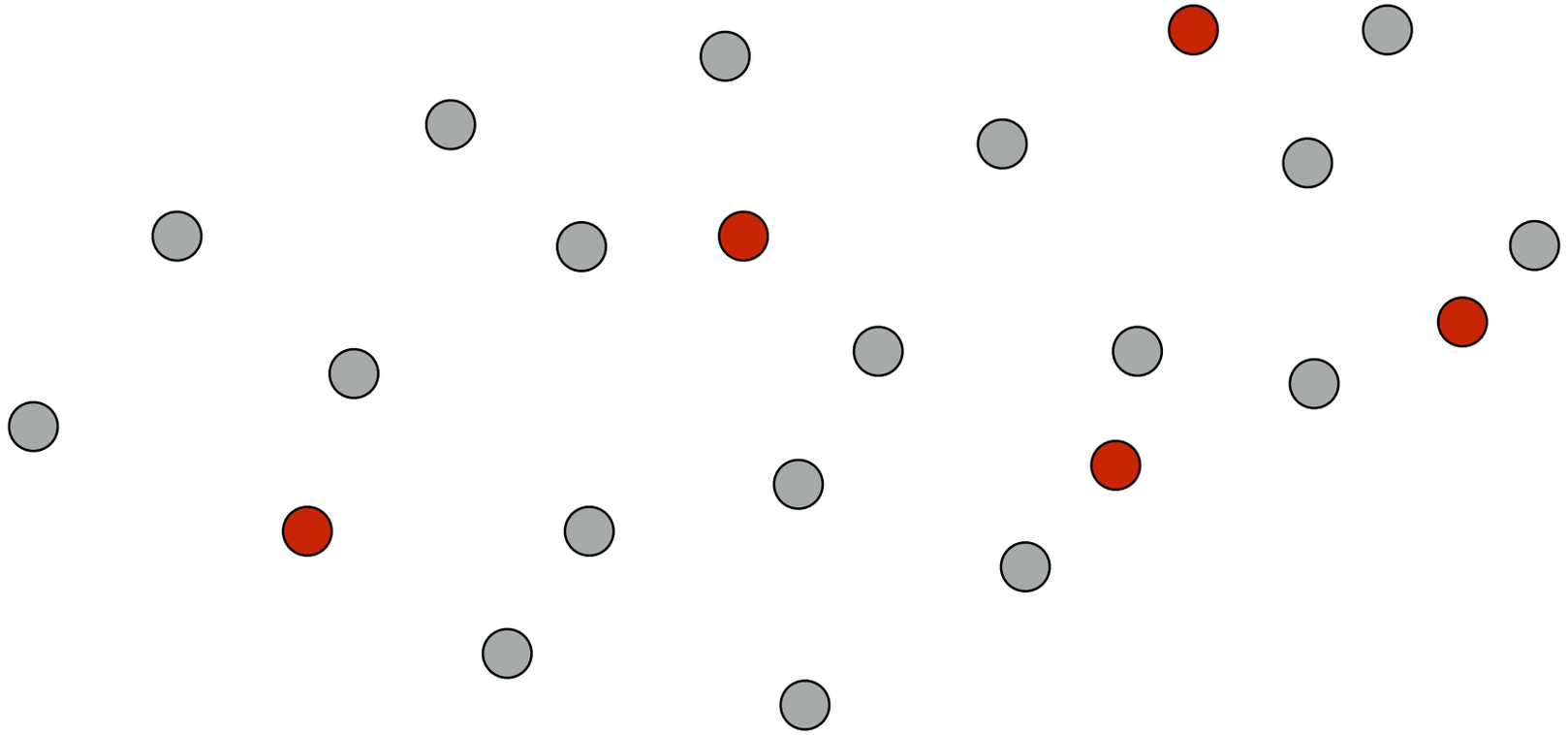
Consensus protocol with angel:

1. Transactions are broadcast to everyone
2. Each node collects transactions into a block
3. At end of round, angel picks the leader
4. The leader adds a block of valid transactions to their personal view.
5. The leader announces their personal view of the blockchain.
6. Everyone else accepts the announced view as their own if:
   a. The transactions in this view are all valid
   b. The announced chain has **more blocks** than their view.

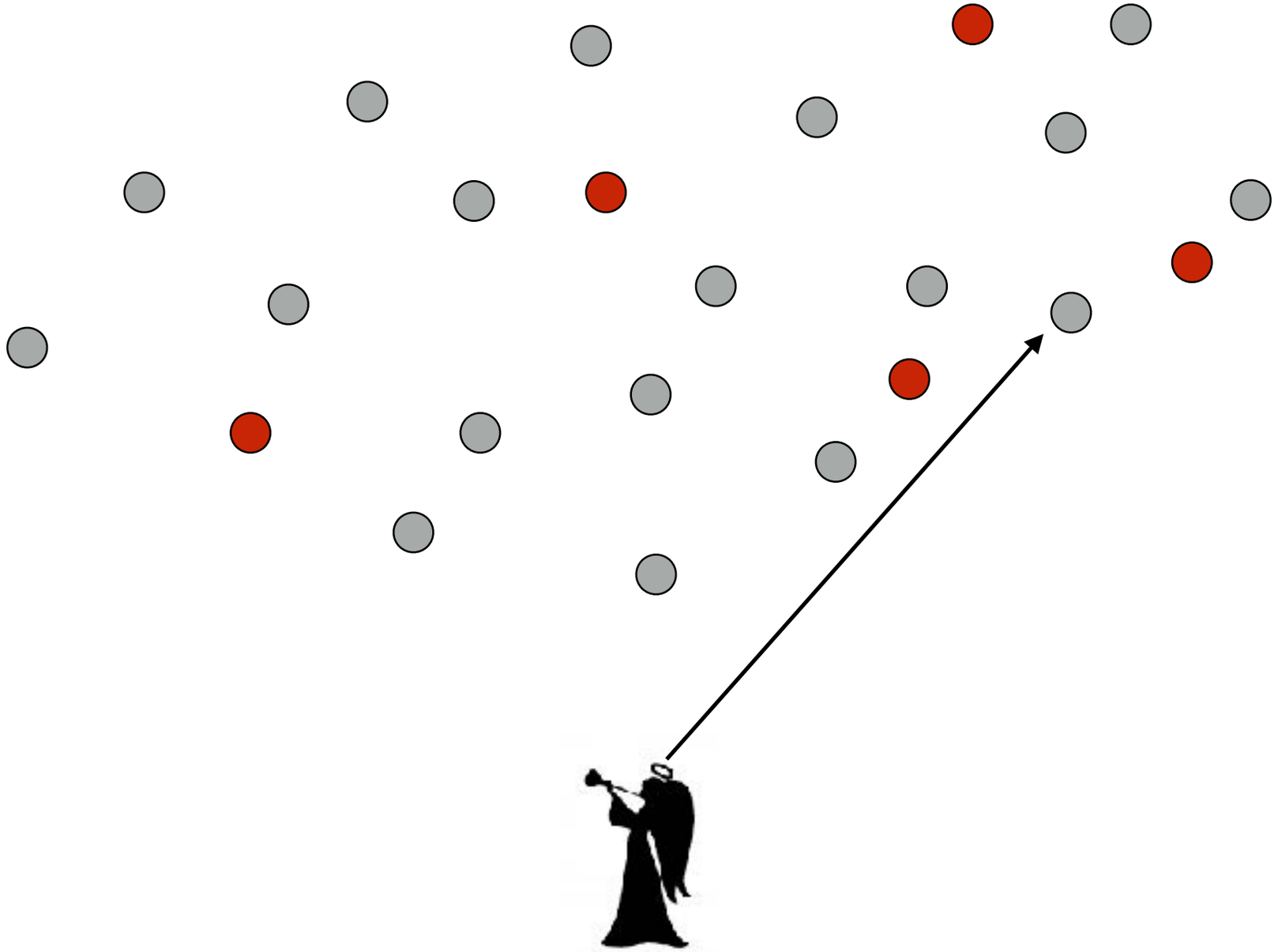
Note 1: When a node is chosen, it can always add a block to its view. It's up to the other nodes to accept it or not.

Note 2: A node may *only* add a block to its view when the angel chooses it.

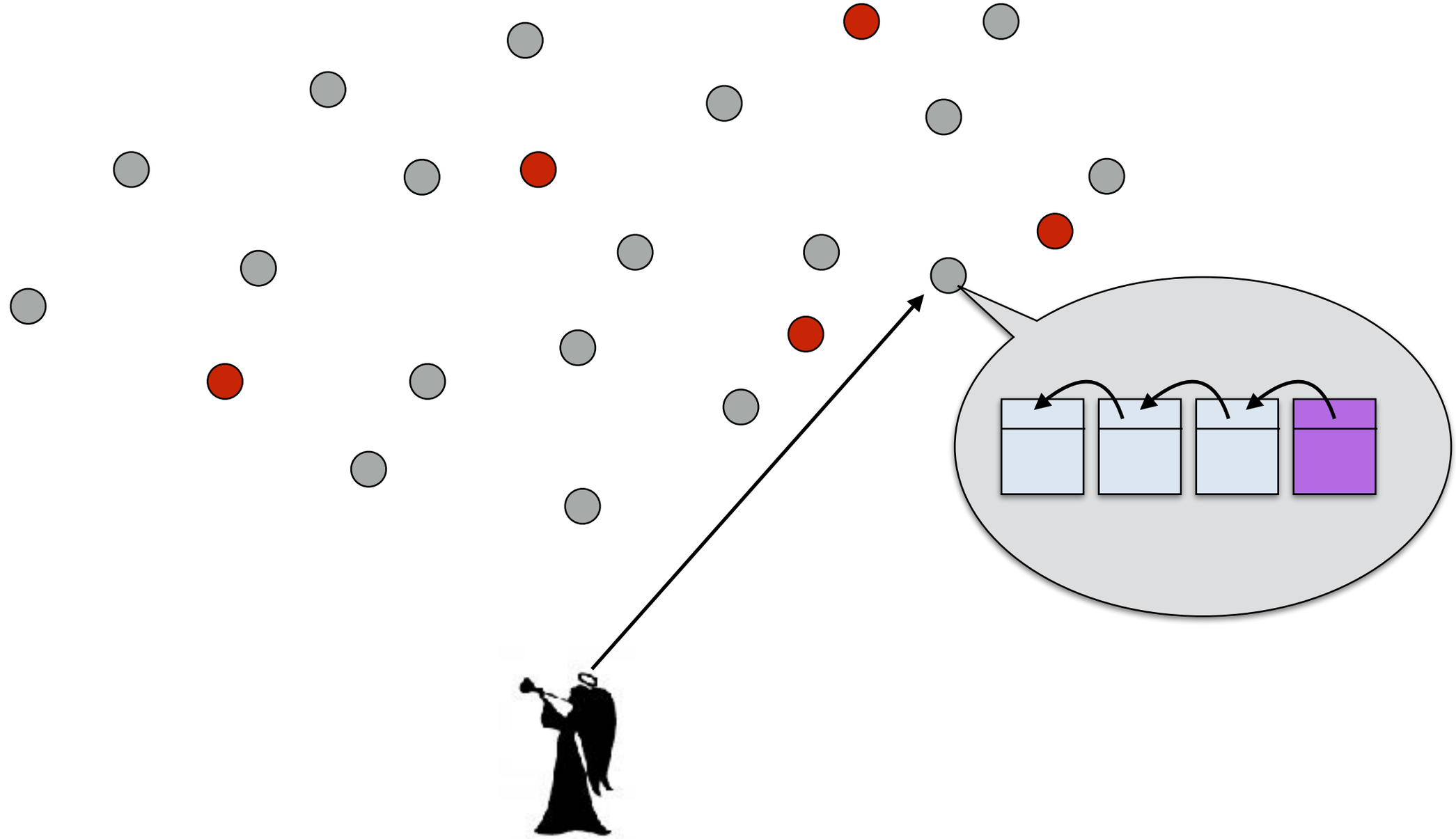
Example: Angel picks an honest node



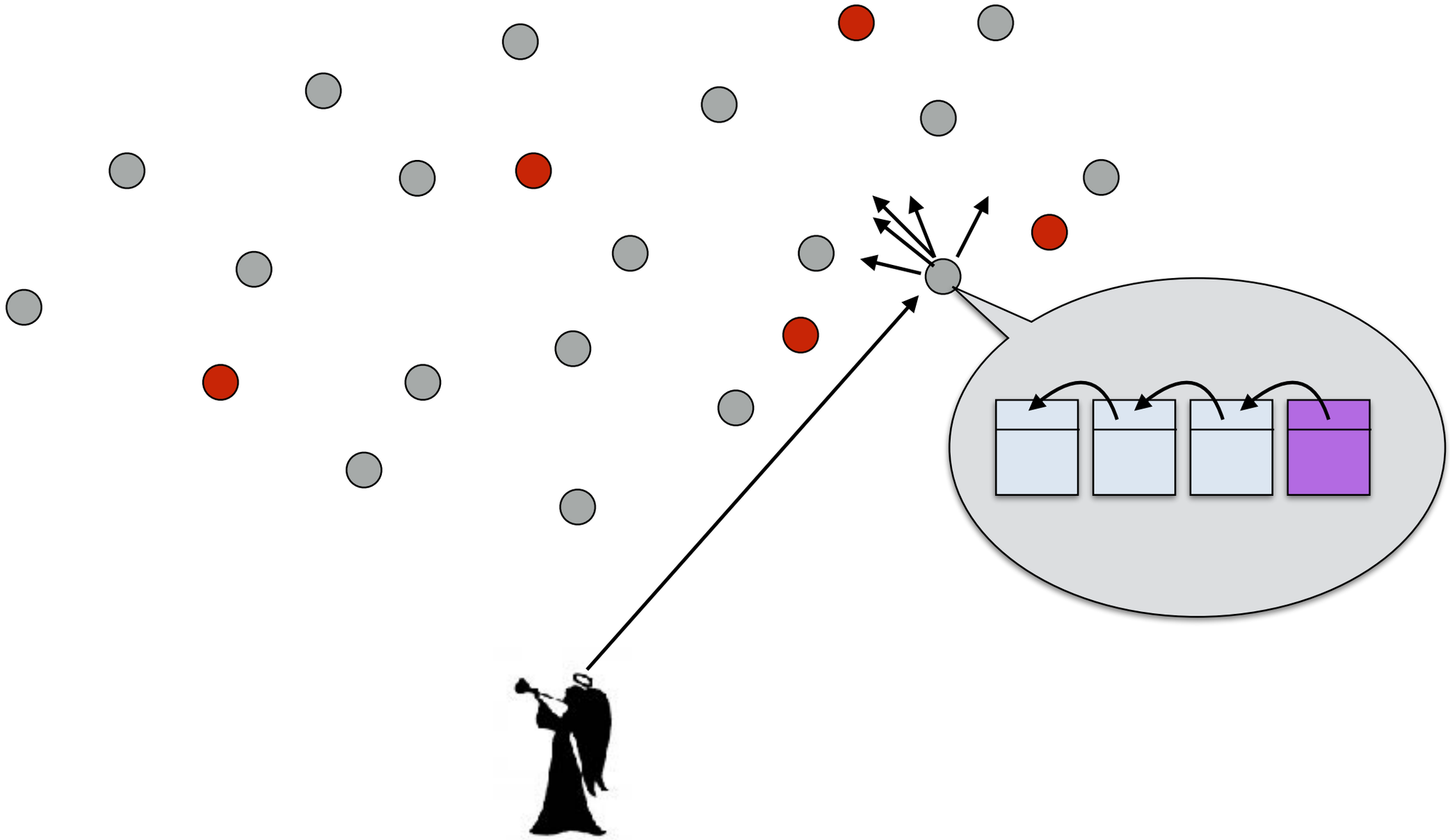
Example: Angel picks an honest node



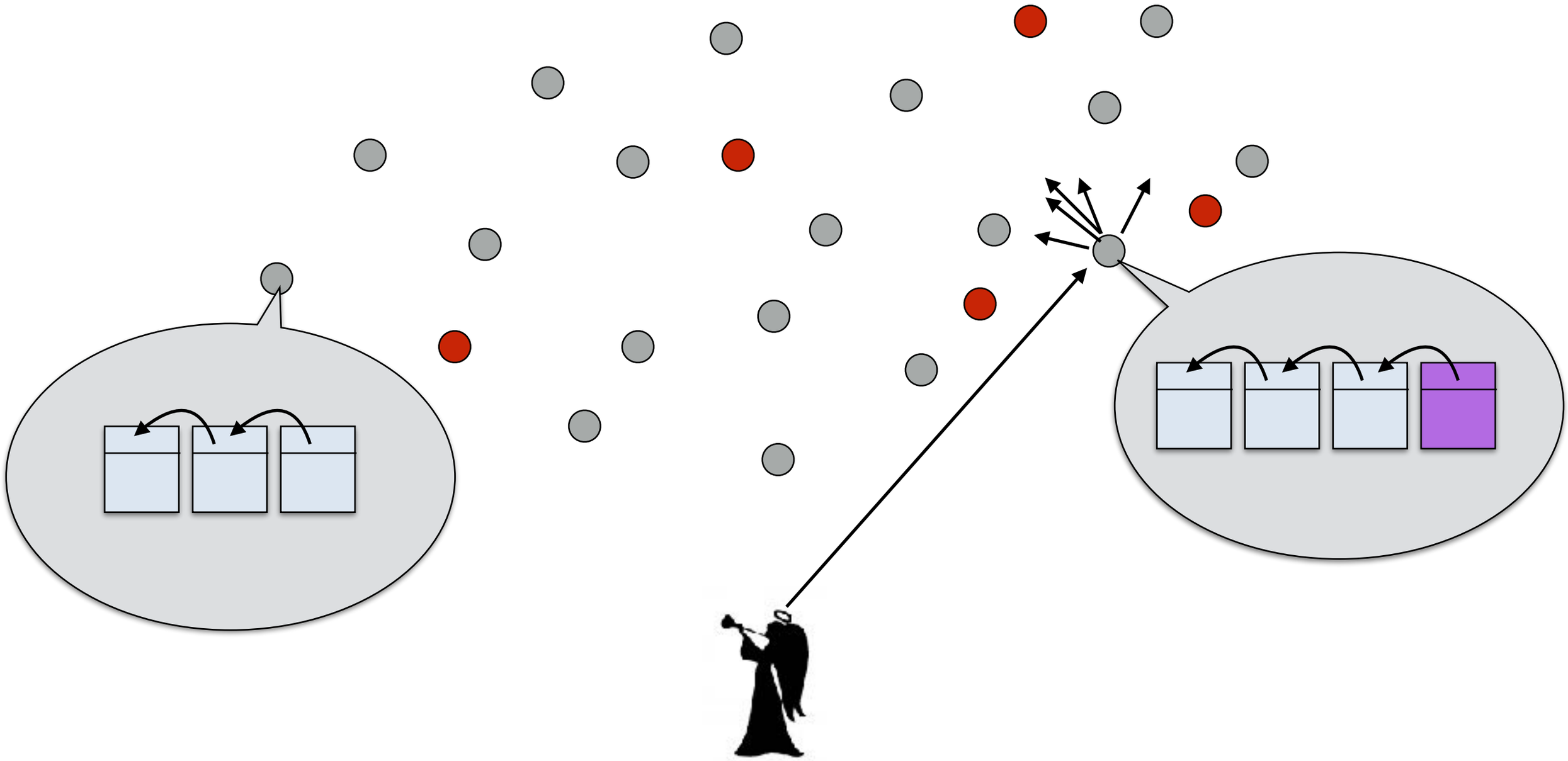
Example: Angel picks an honest node



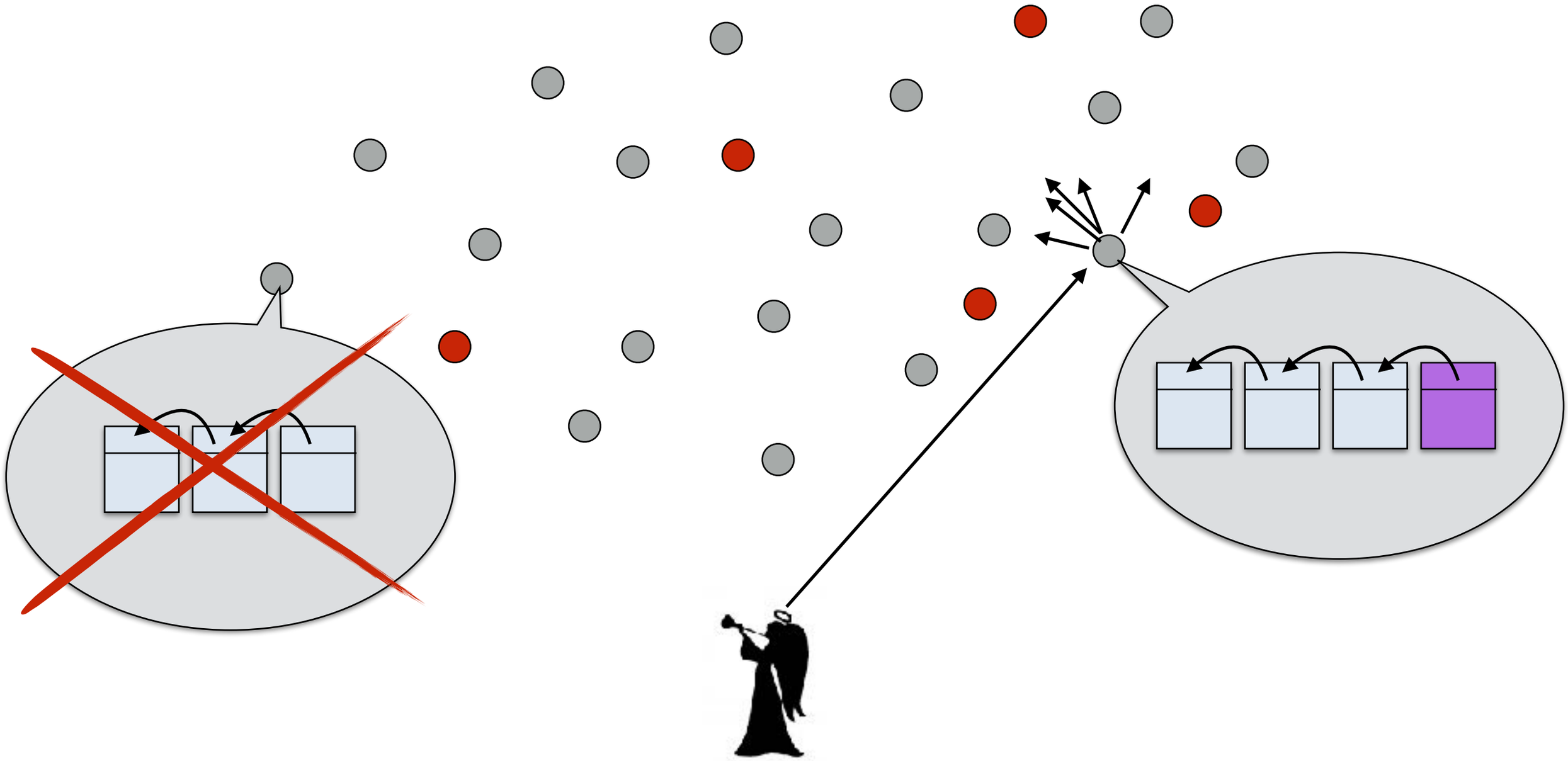
Example: Angel picks an honest node



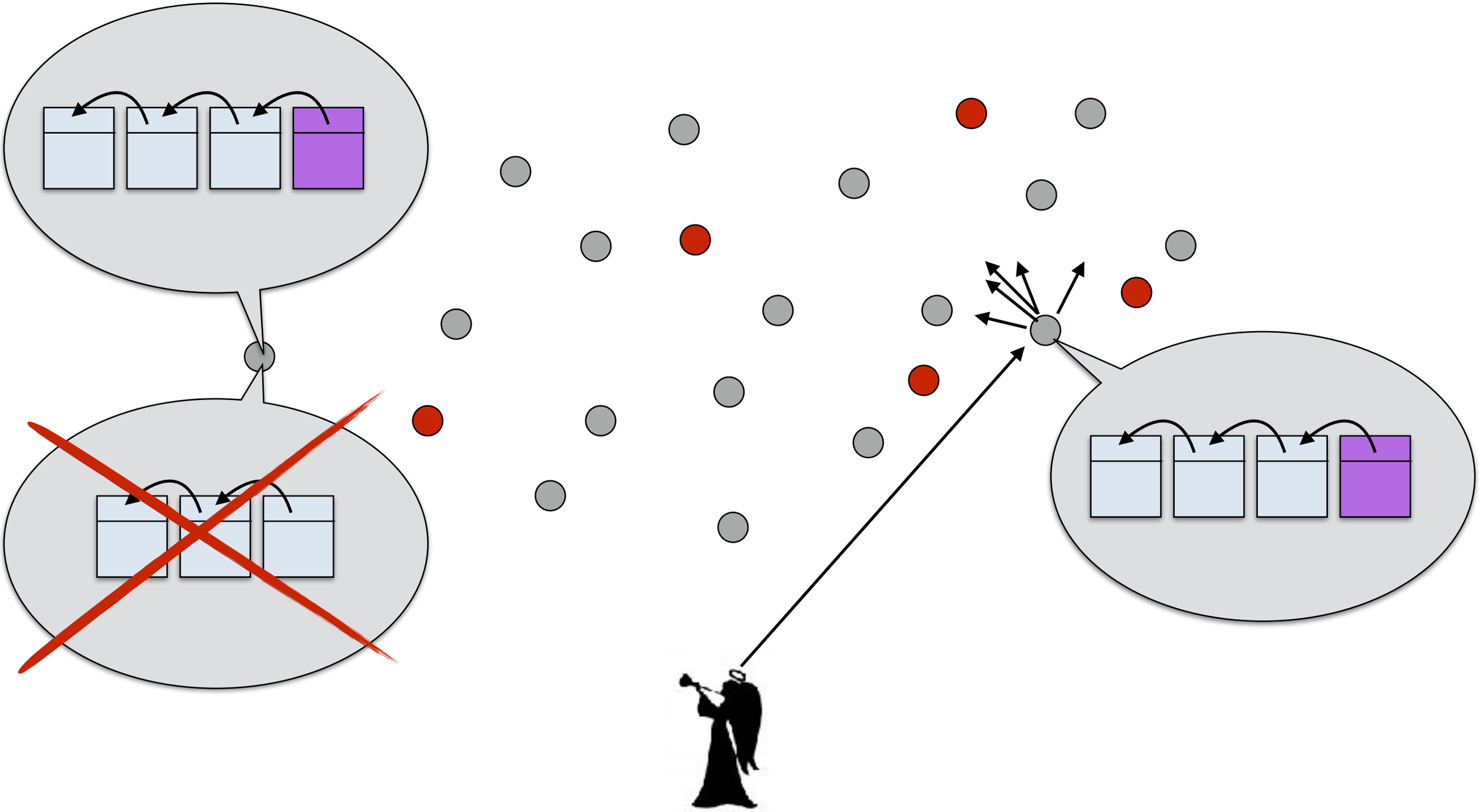
Example: Angel picks an honest node



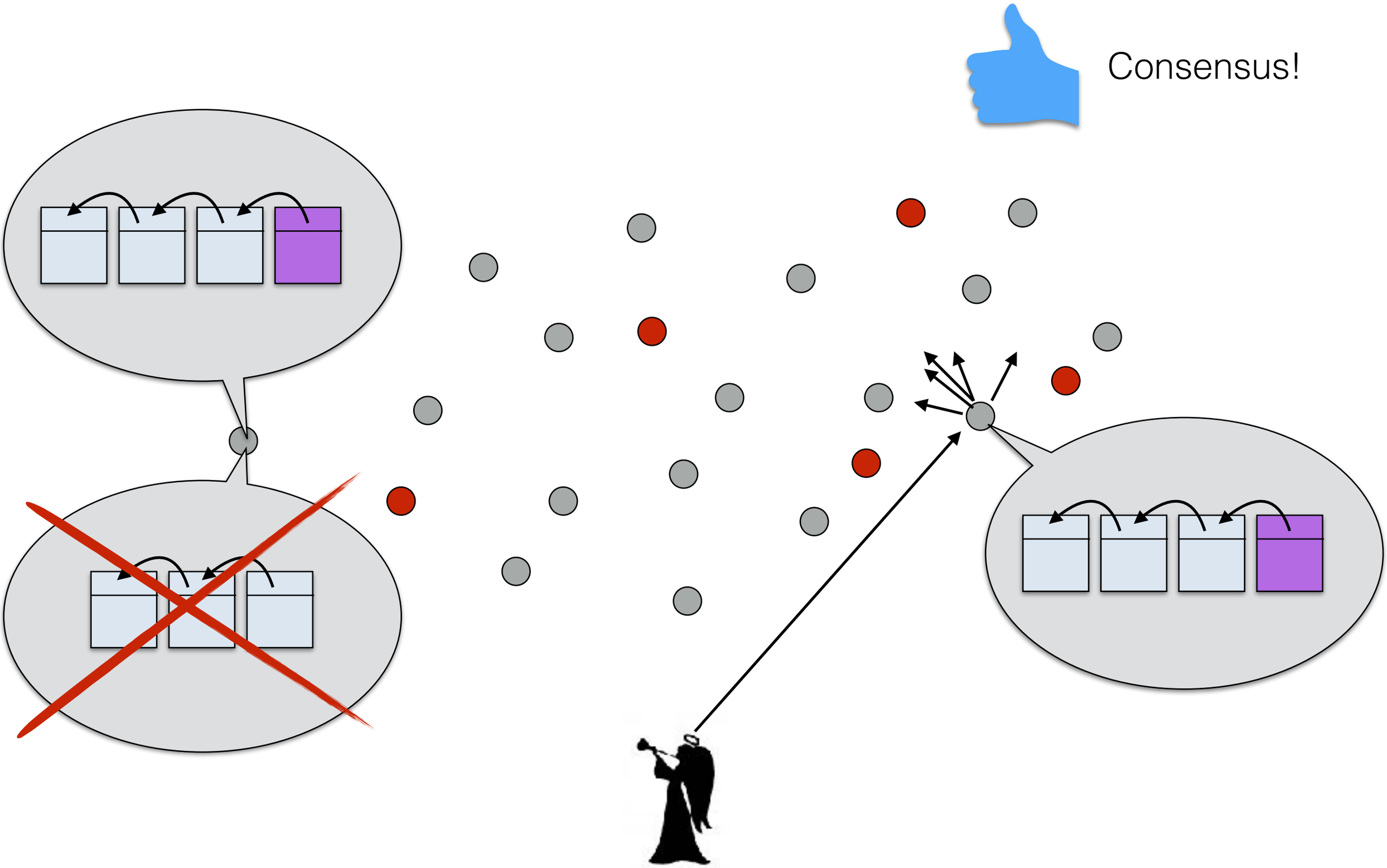
Example: Angel picks an honest node



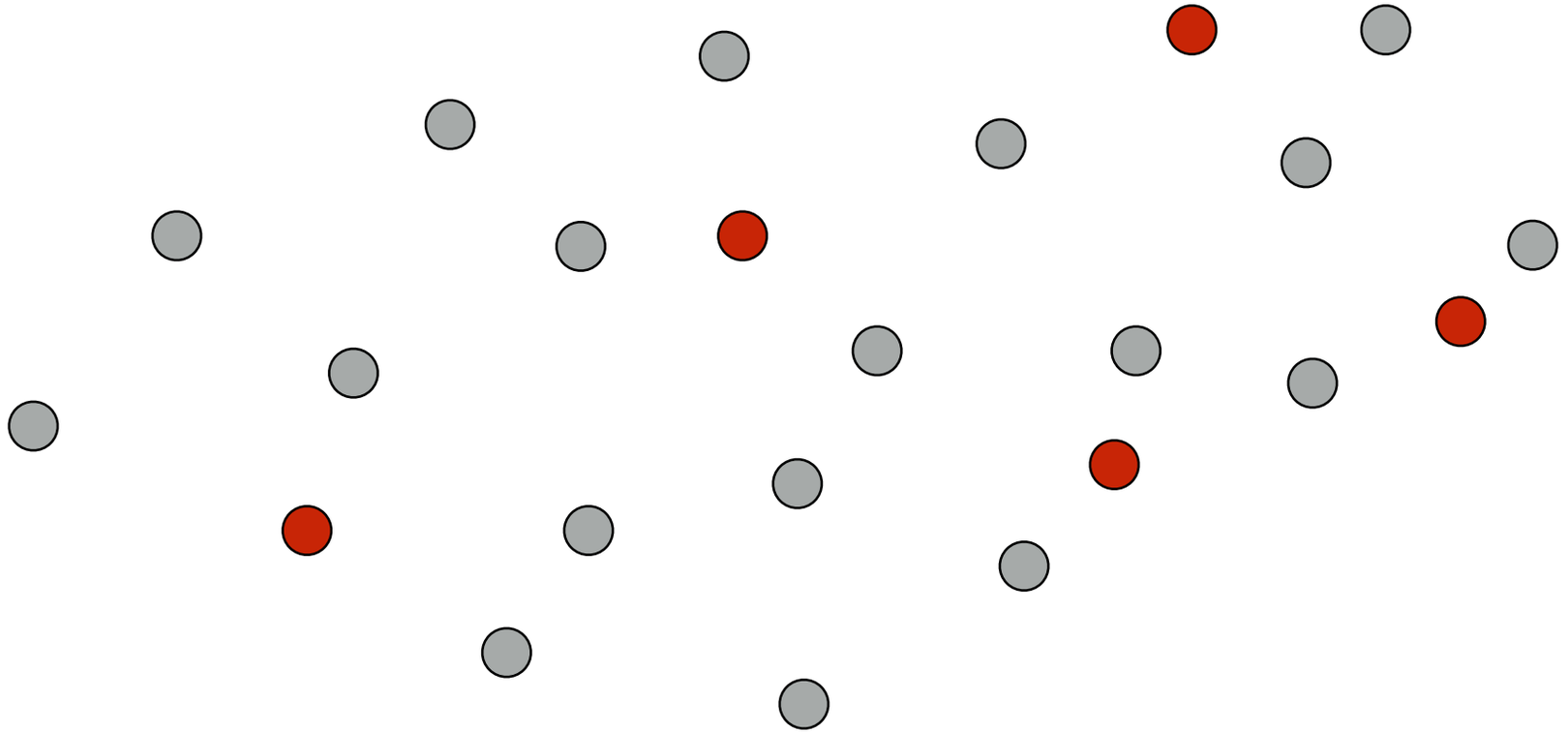
Example: Angel picks an honest node



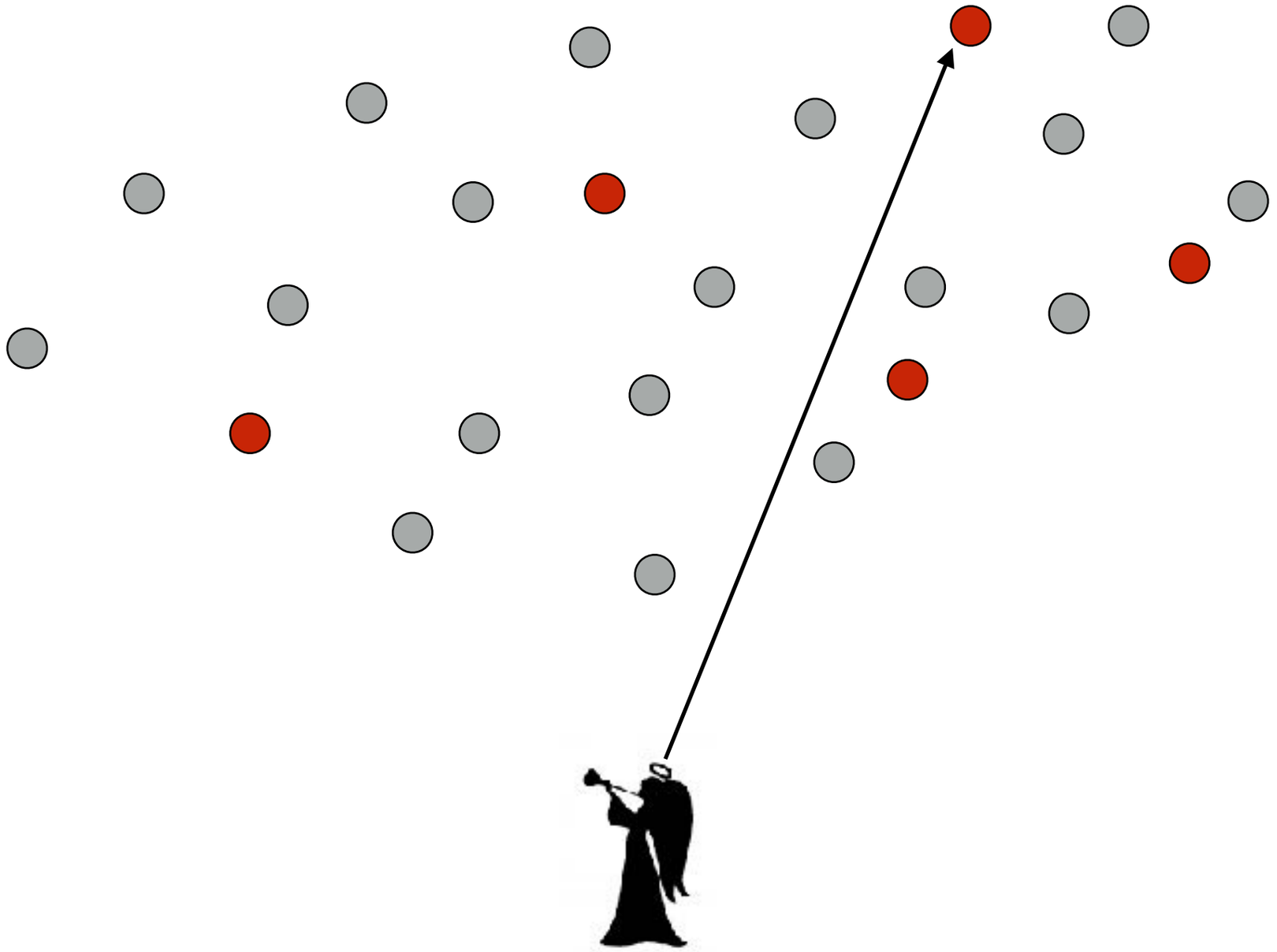
Example: Angel picks an honest node



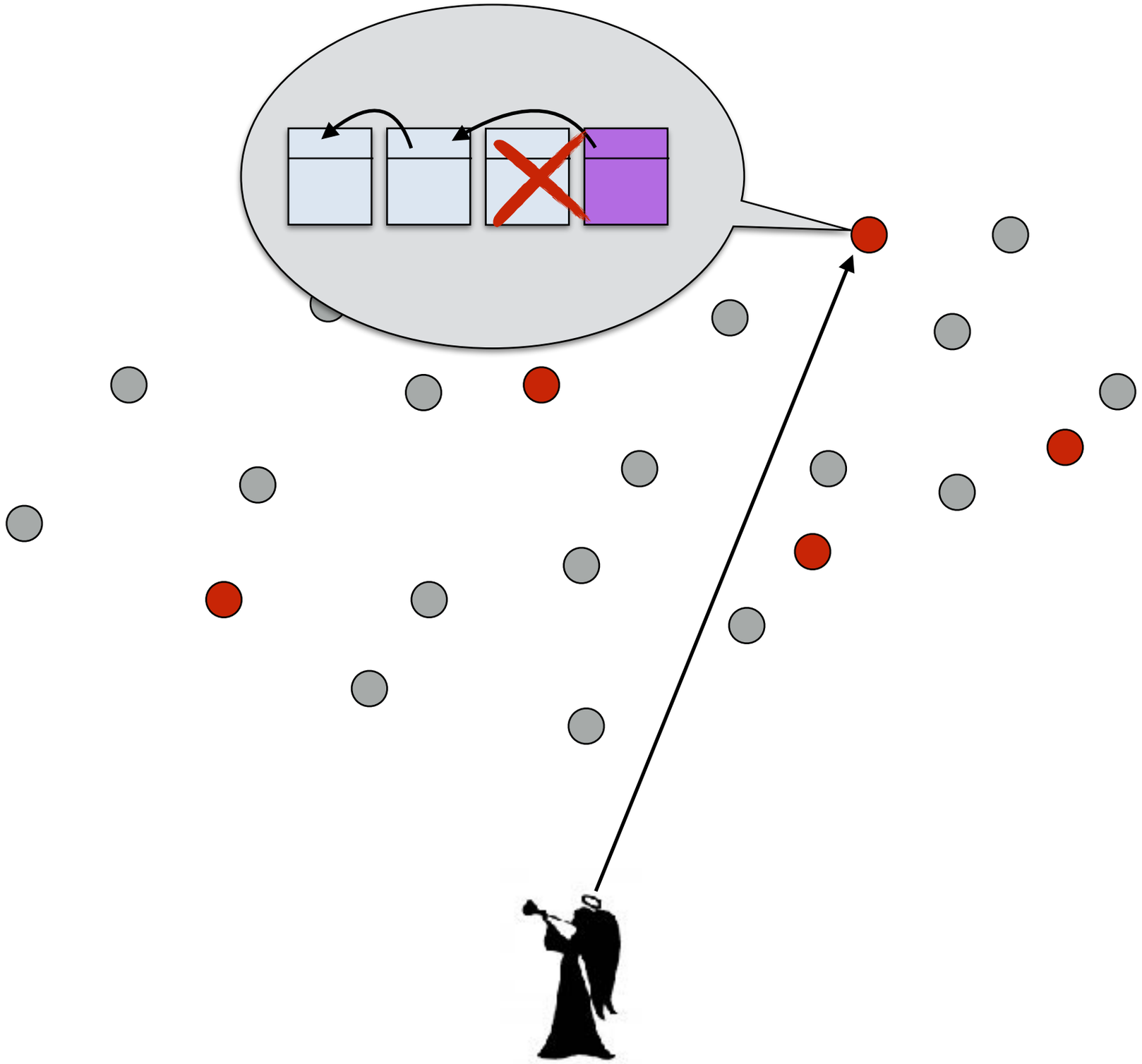
Example: Angel picks an malicious node



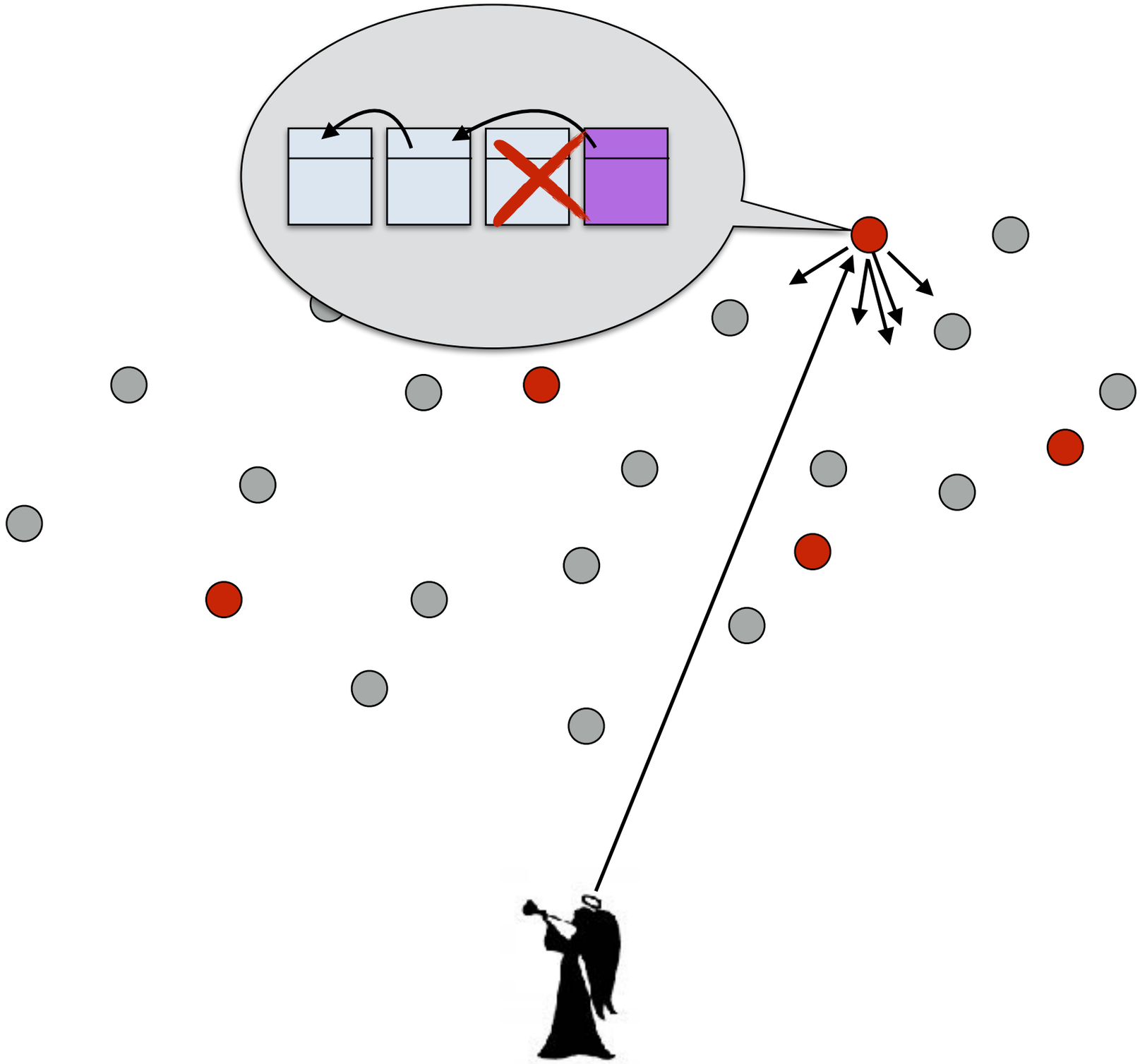
Example: Angel picks an malicious node



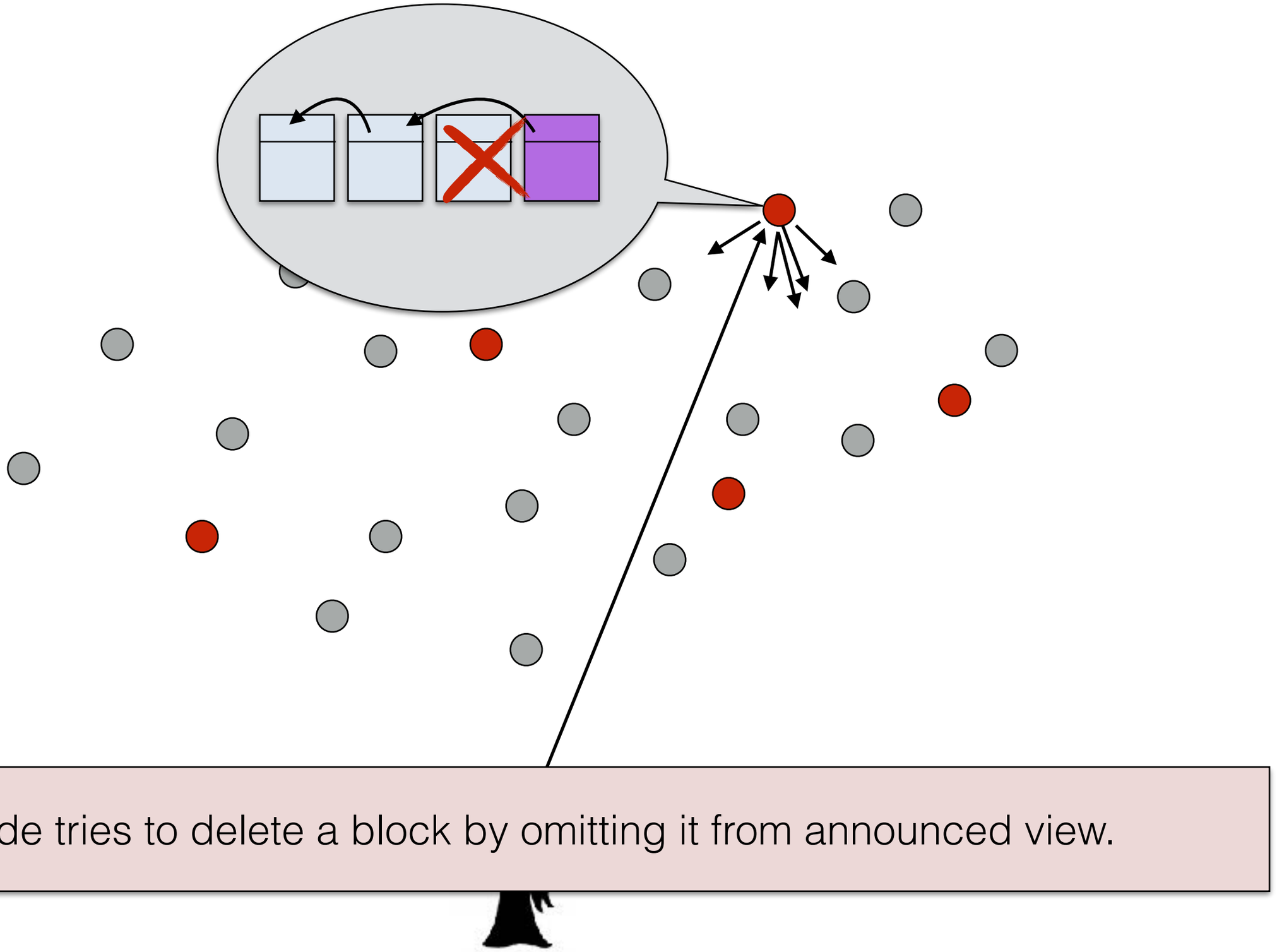
Example: Angel picks an malicious node



Example: Angel picks an malicious node

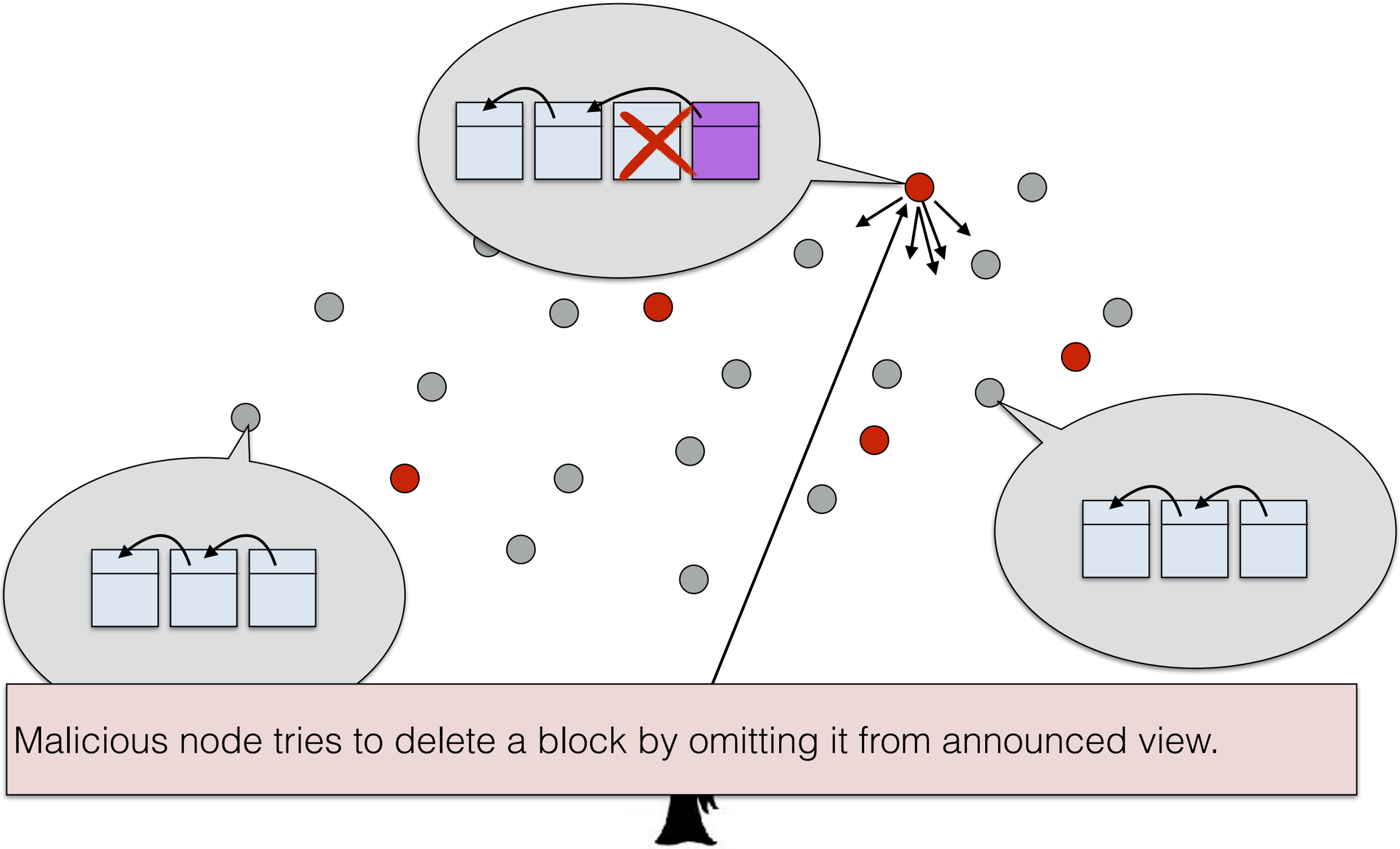


Example: Angel picks an malicious node



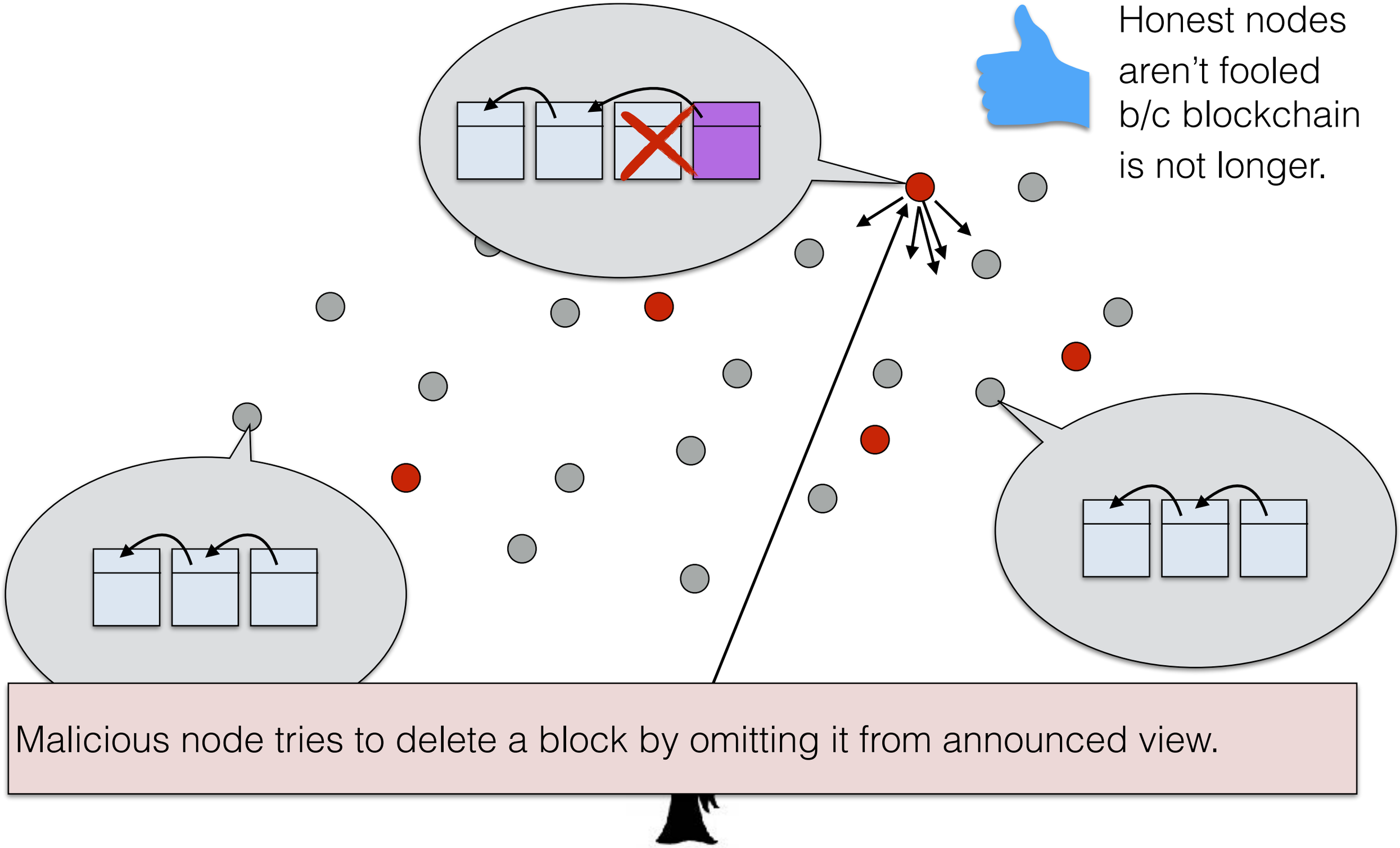
Malicious node tries to delete a block by omitting it from announced view.

Example: Angel picks an malicious node



Malicious node tries to delete a block by omitting it from announced view.

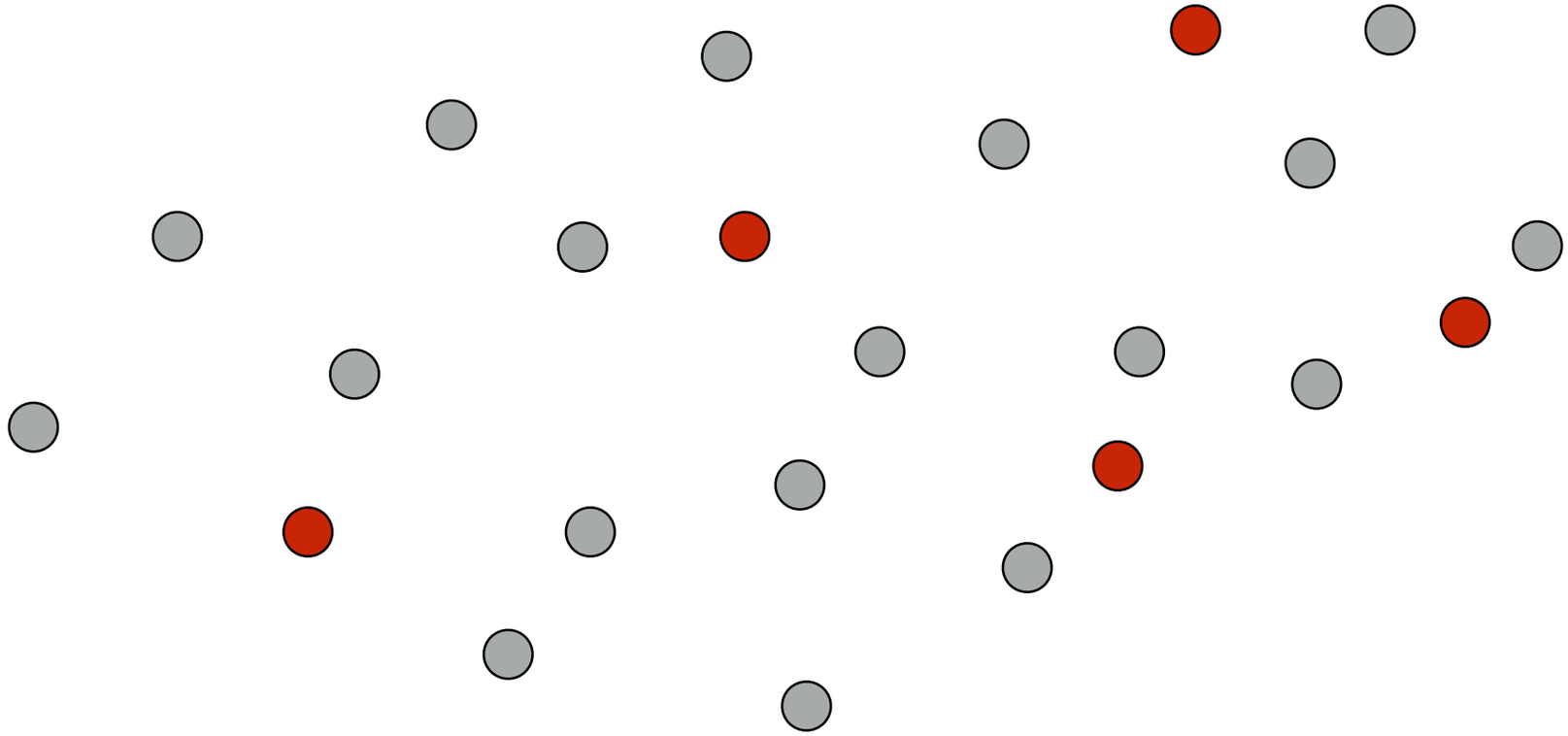
Example: Angel picks an malicious node



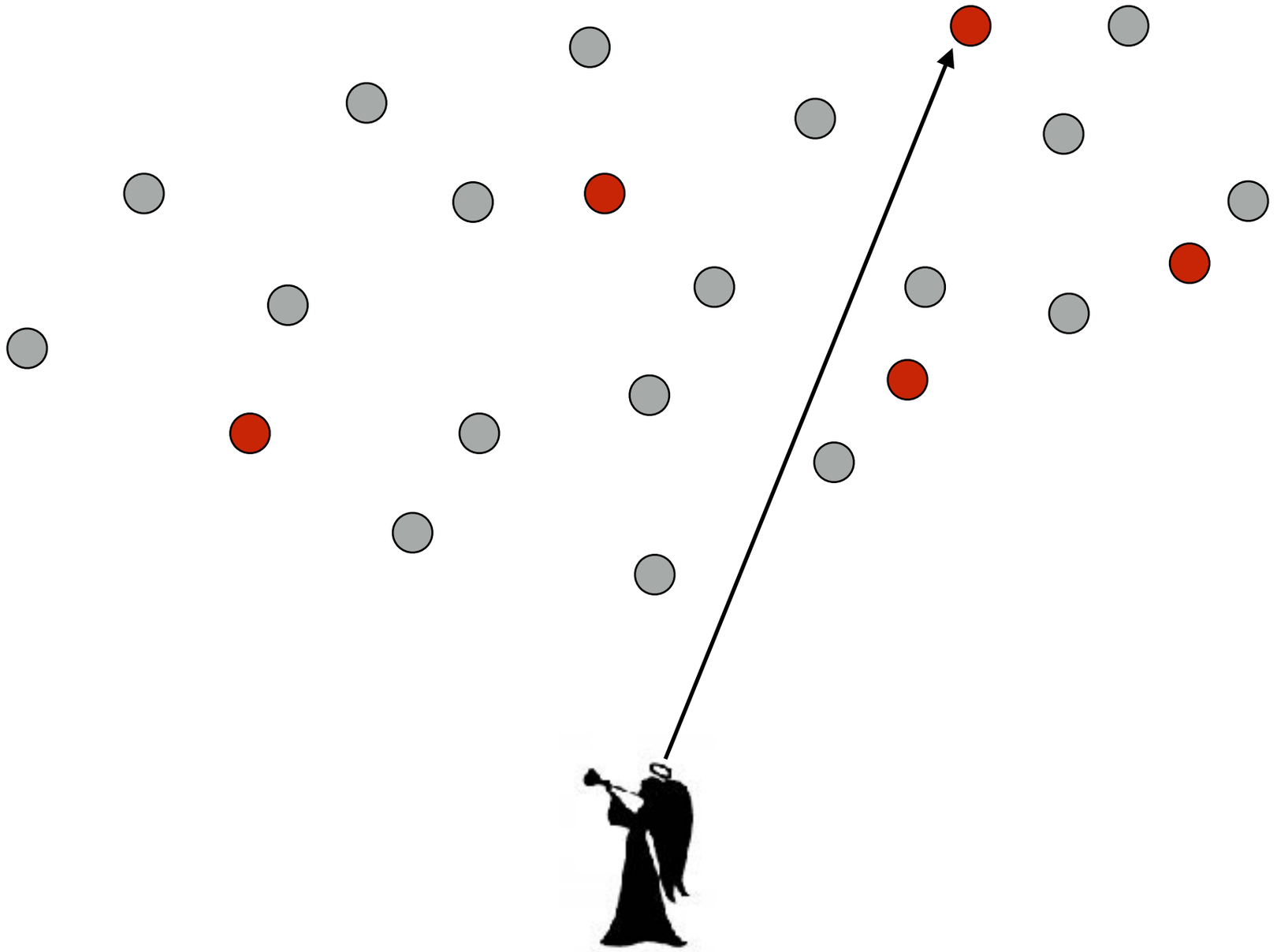
Consensus!
Honest nodes aren't fooled b/c blockchain is not longer.

Malicious node tries to delete a block by omitting it from announced view.

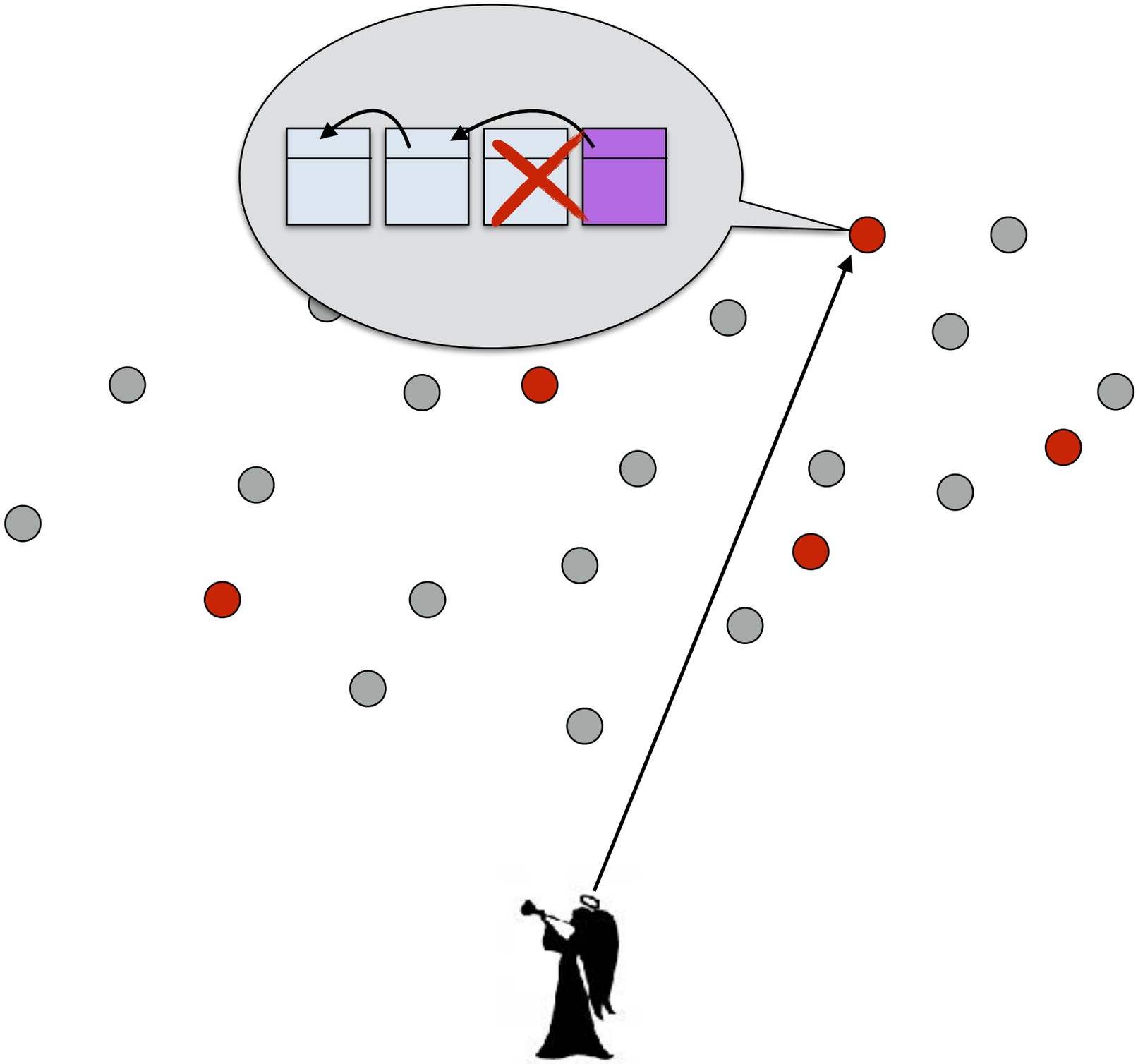
Example: Angel picks an malicious node twice



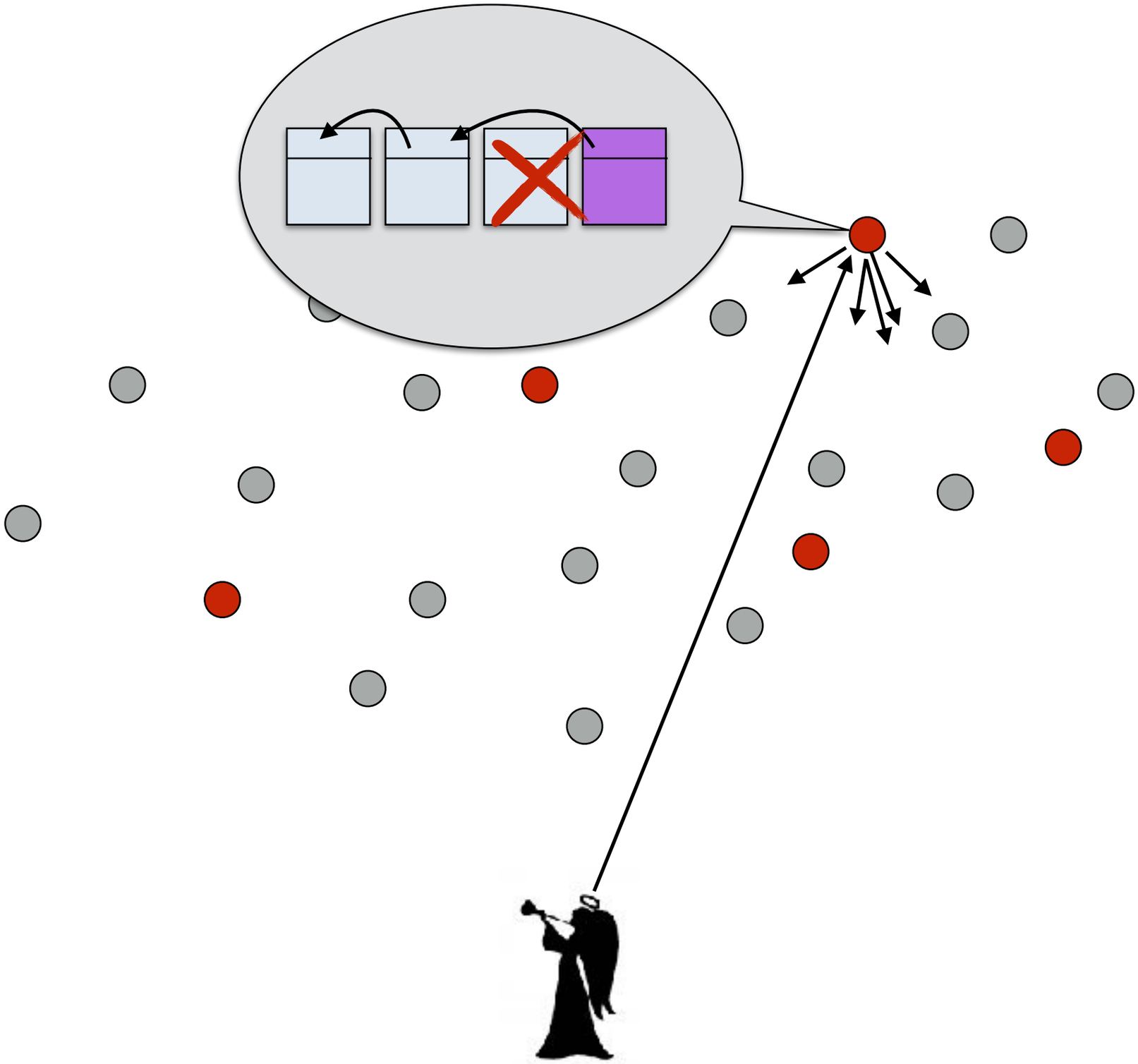
Example: Angel picks an malicious node twice



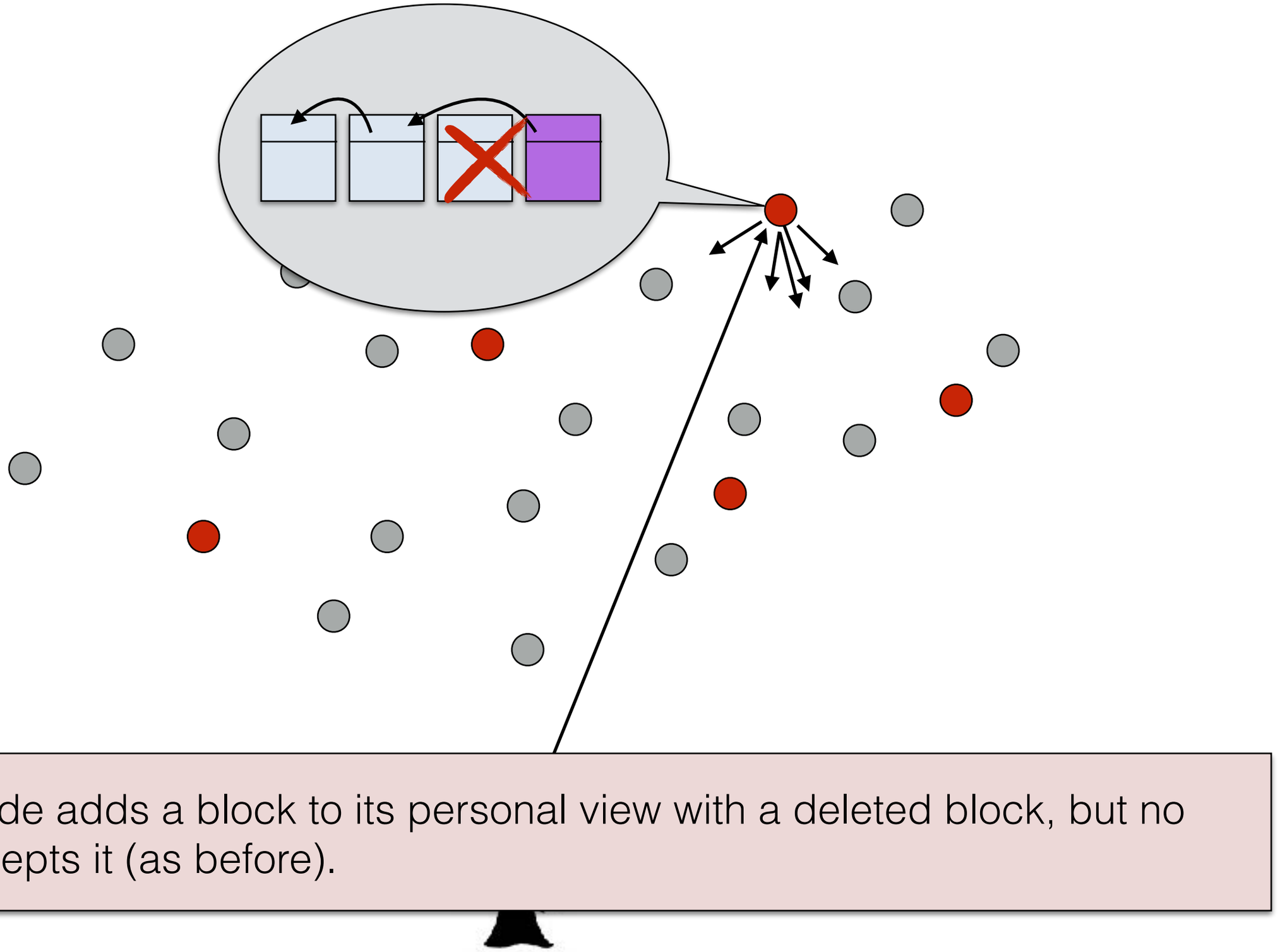
Example: Angel picks an malicious node twice



Example: Angel picks an malicious node twice

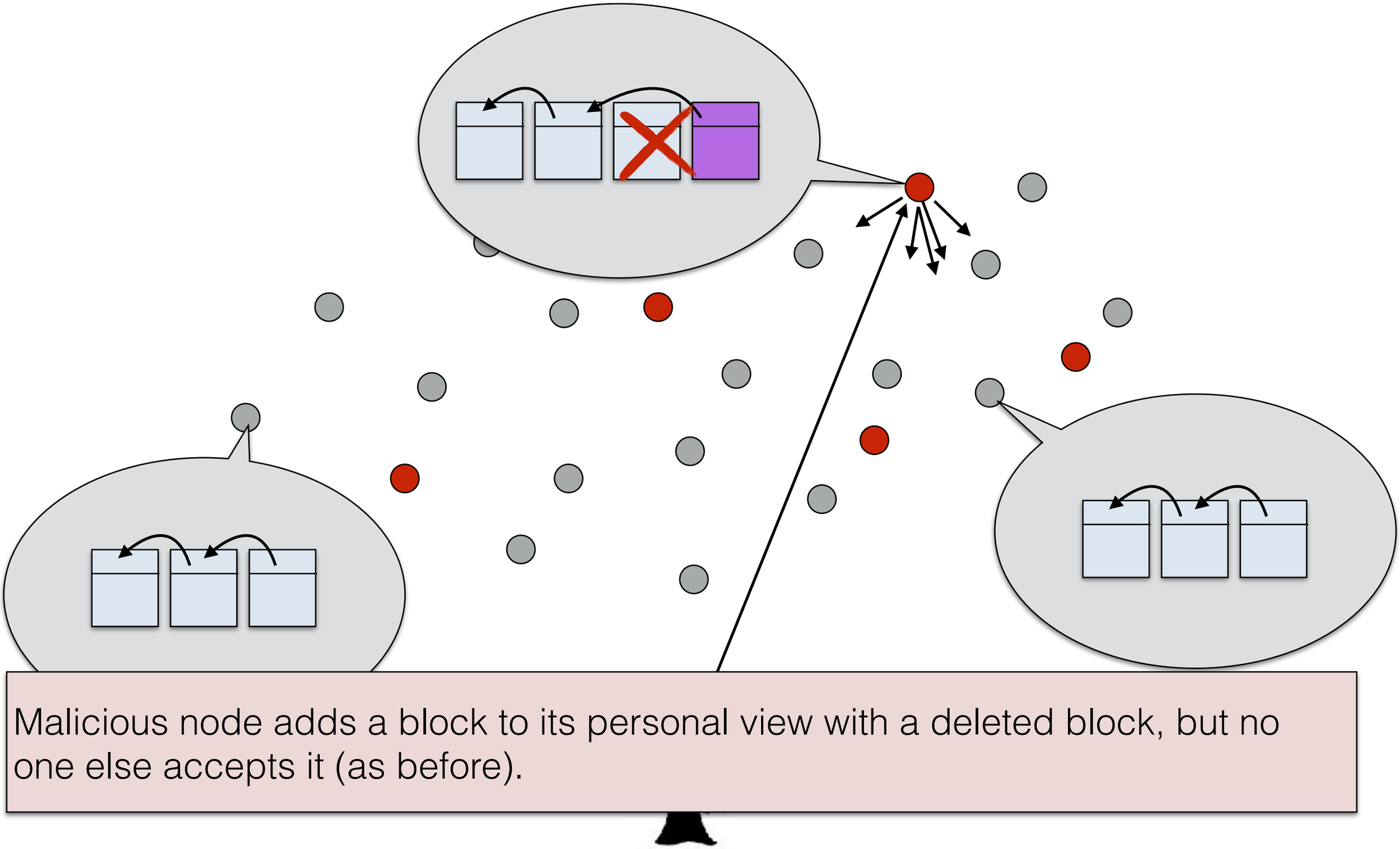


Example: Angel picks an malicious node twice



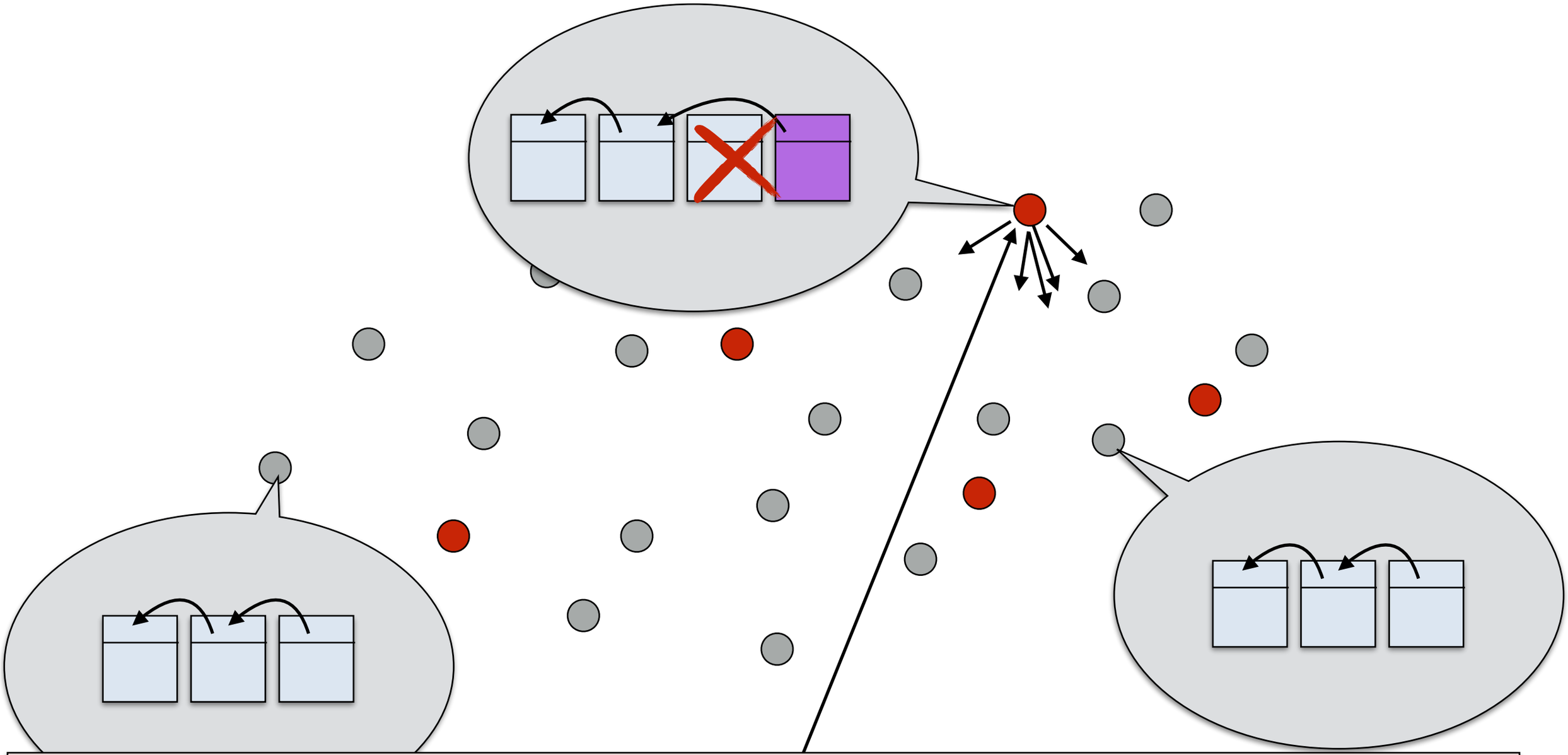
Malicious node adds a block to its personal view with a deleted block, but no one else accepts it (as before).

Example: Angel picks an malicious node twice



Malicious node adds a block to its personal view with a deleted block, but no one else accepts it (as before).

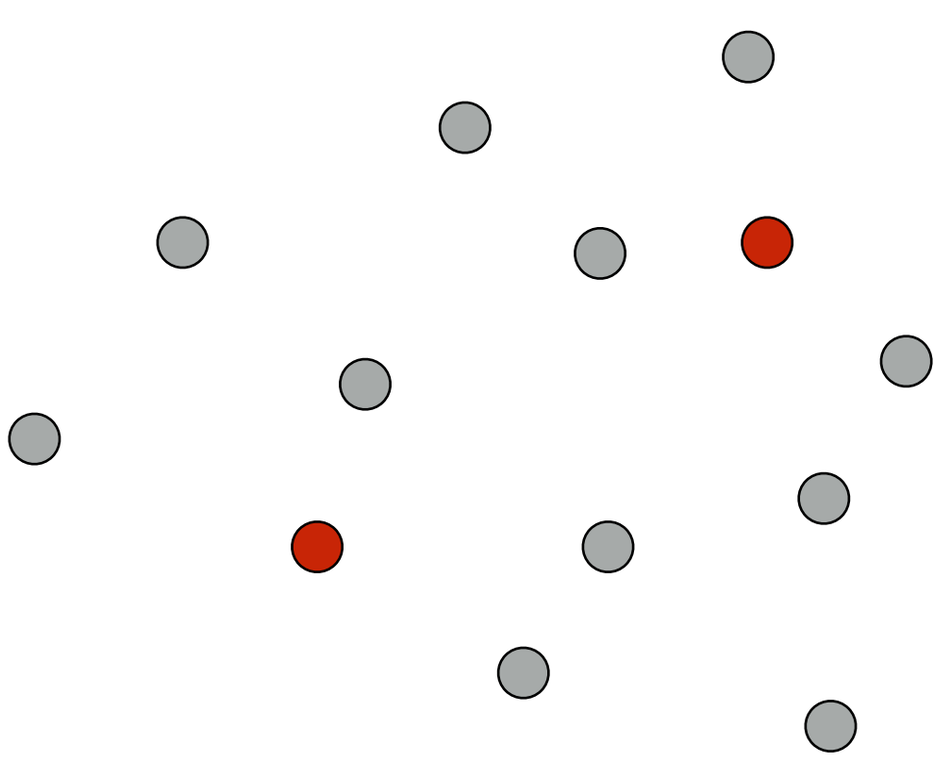
Example: Angel picks an malicious node twice



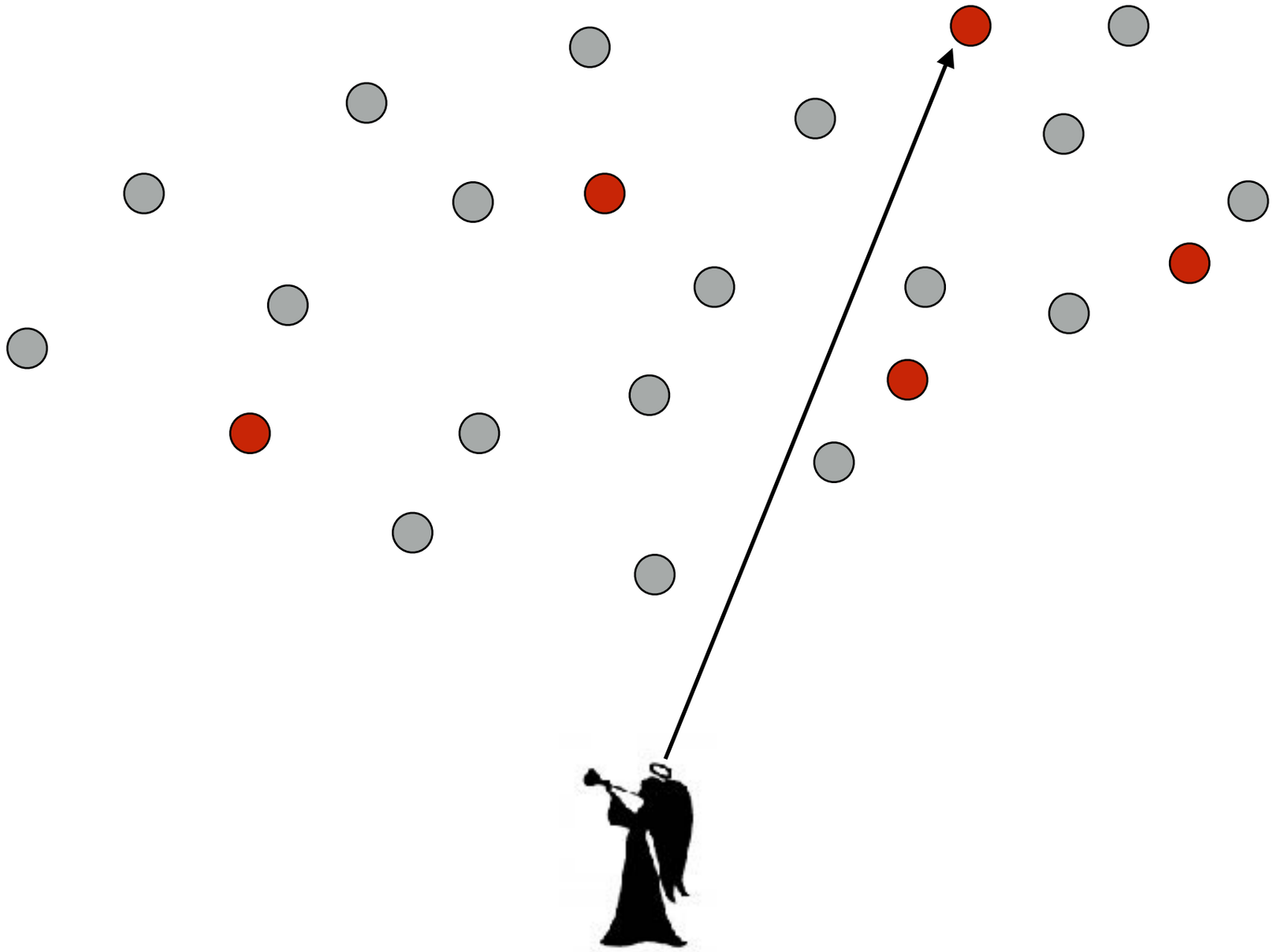
Malicious node adds a block to its personal view with a deleted block, but no one else accepts it (as before).

However, the malicious node adds the block to its view of the chain.

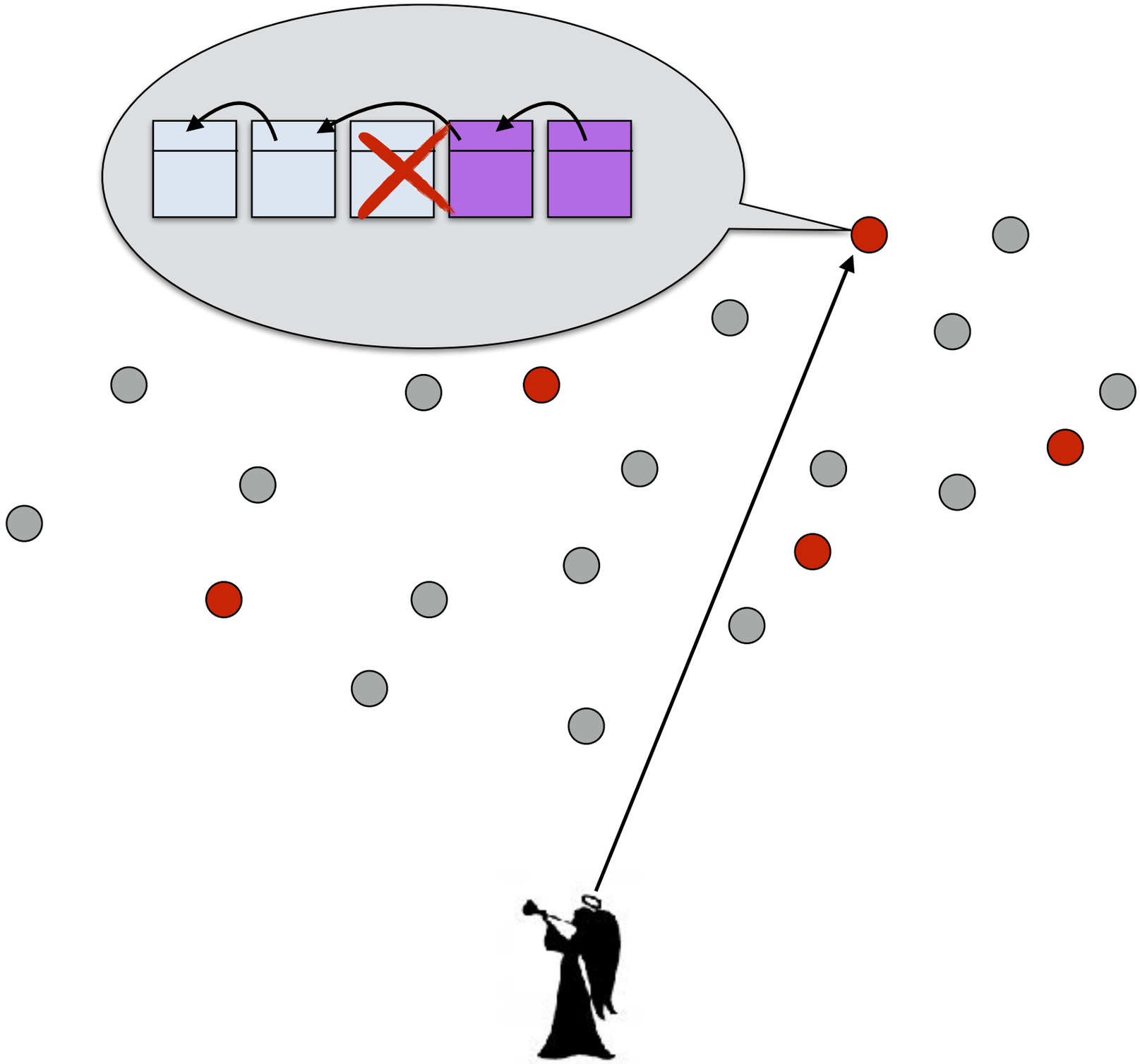
Example: Angel picks an malicious node twice



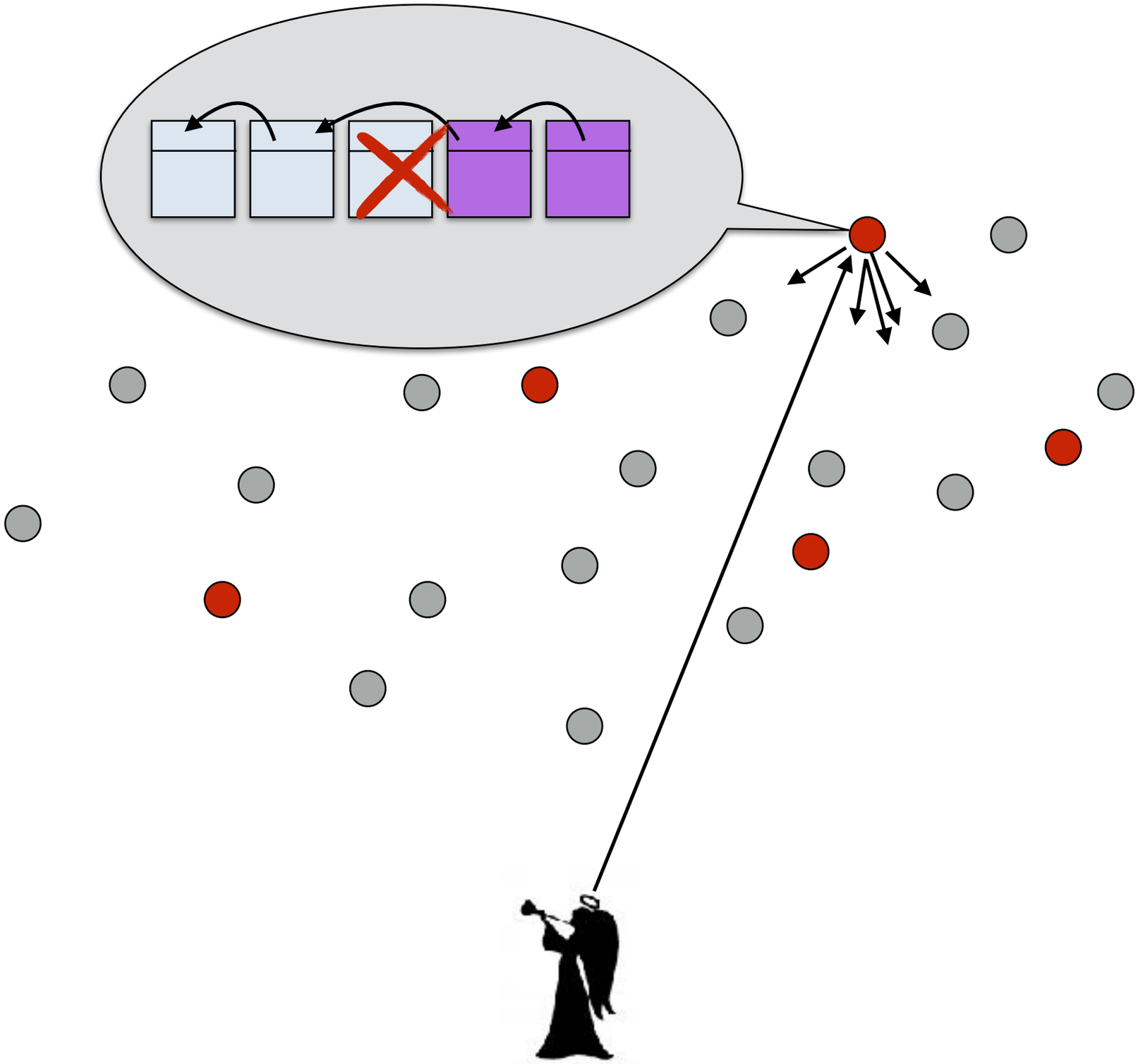
Example: Angel picks an malicious node twice



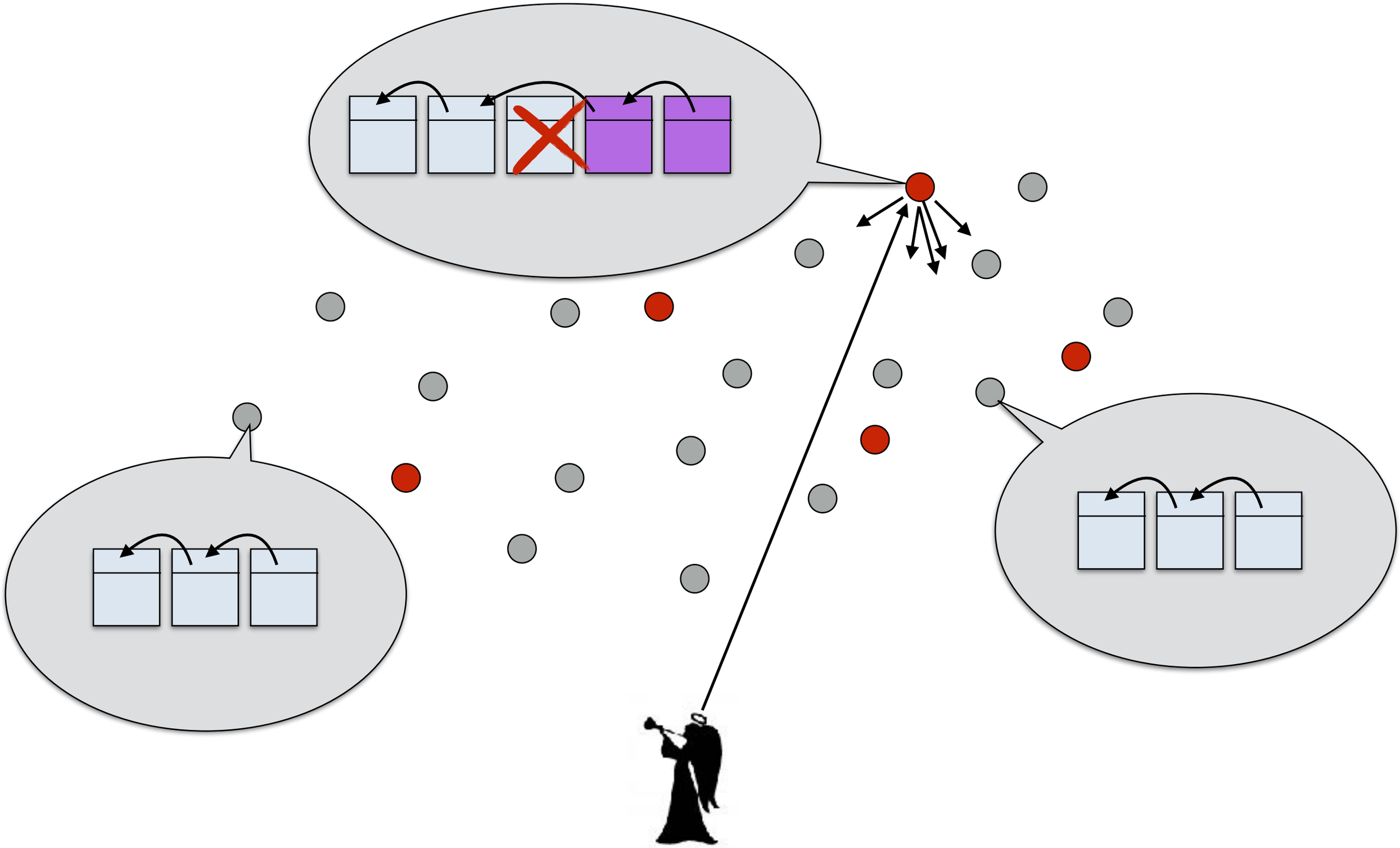
Example: Angel picks an malicious node twice



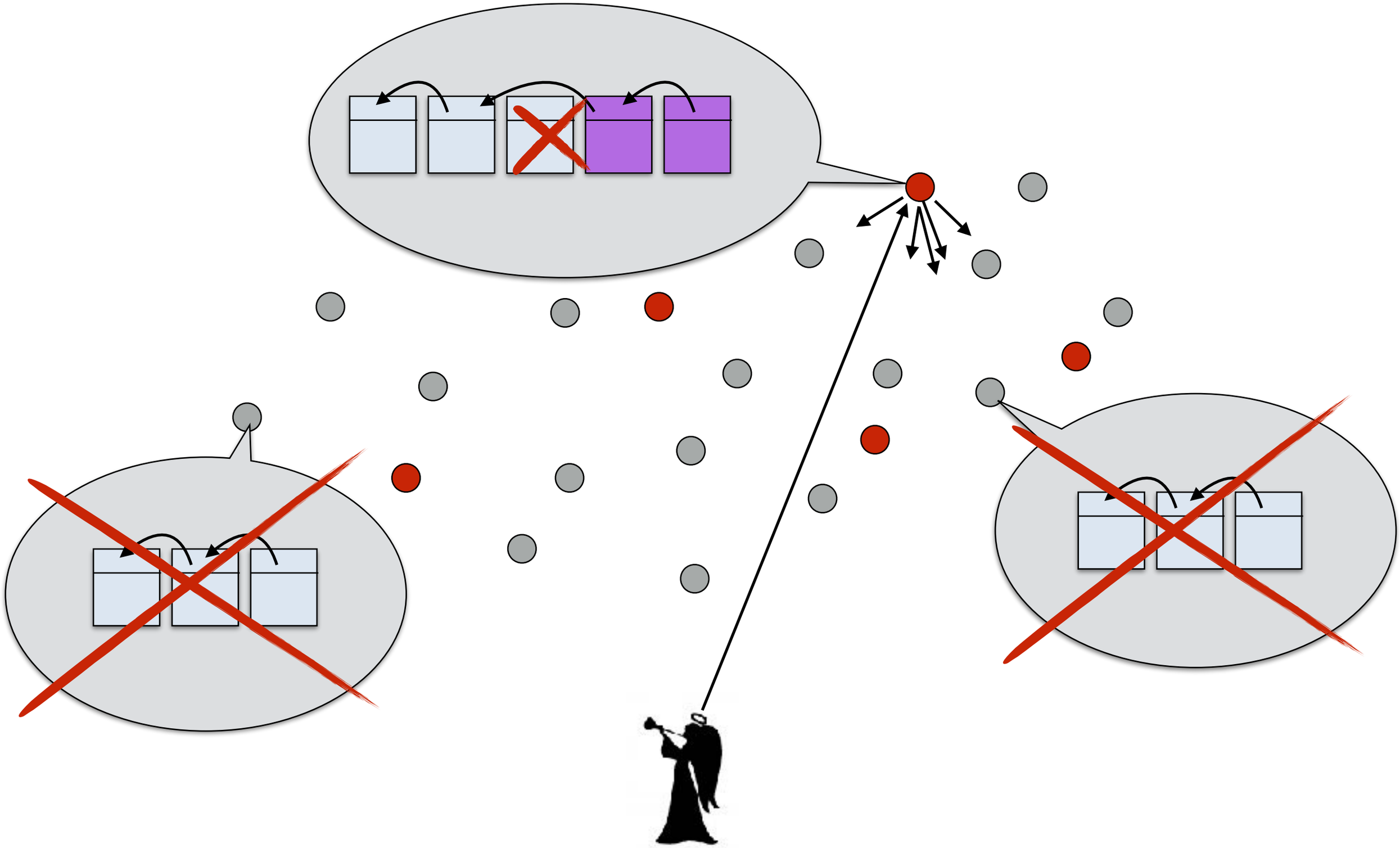
Example: Angel picks an malicious node twice



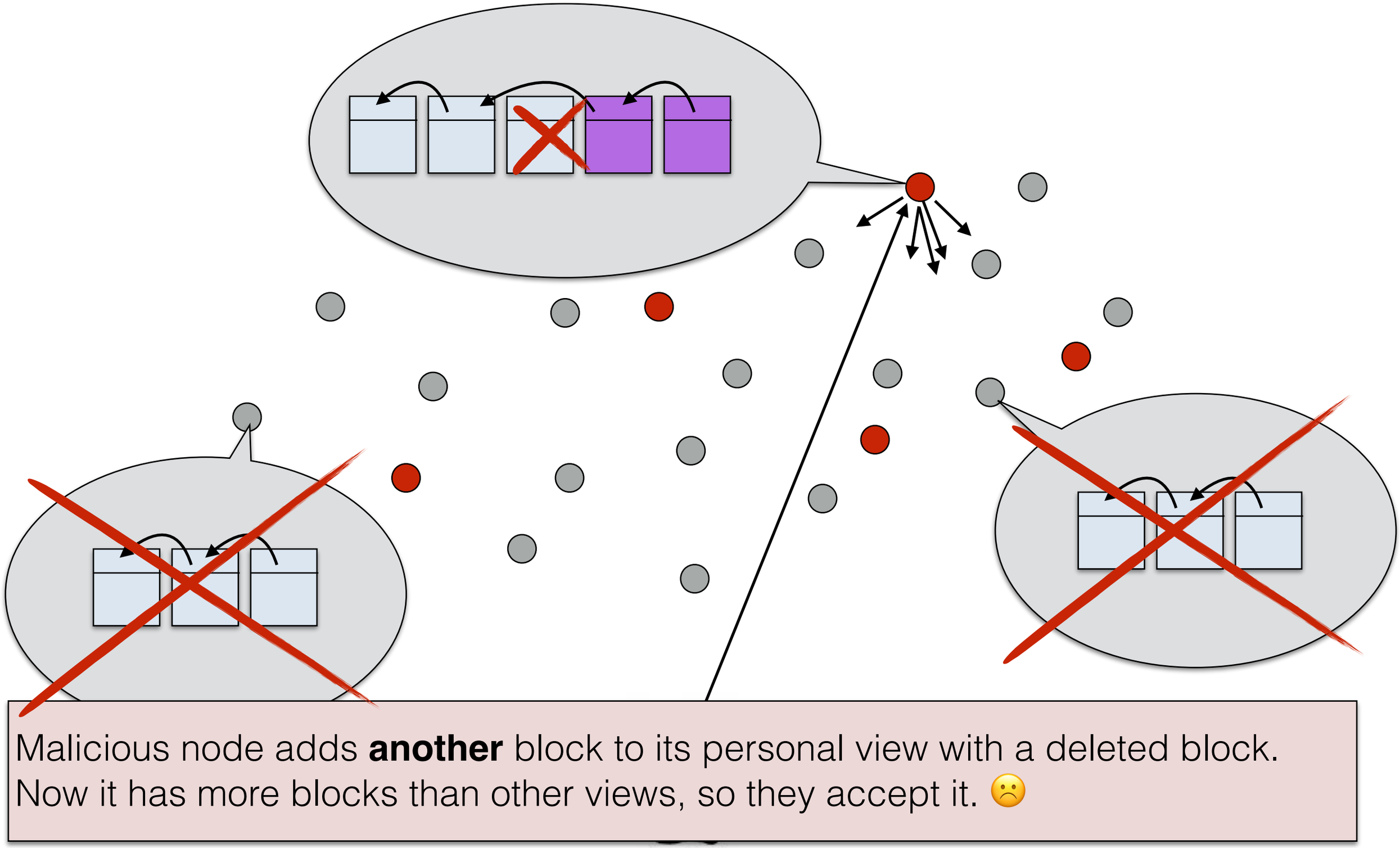
Example: Angel picks an malicious node twice



Example: Angel picks an malicious node twice

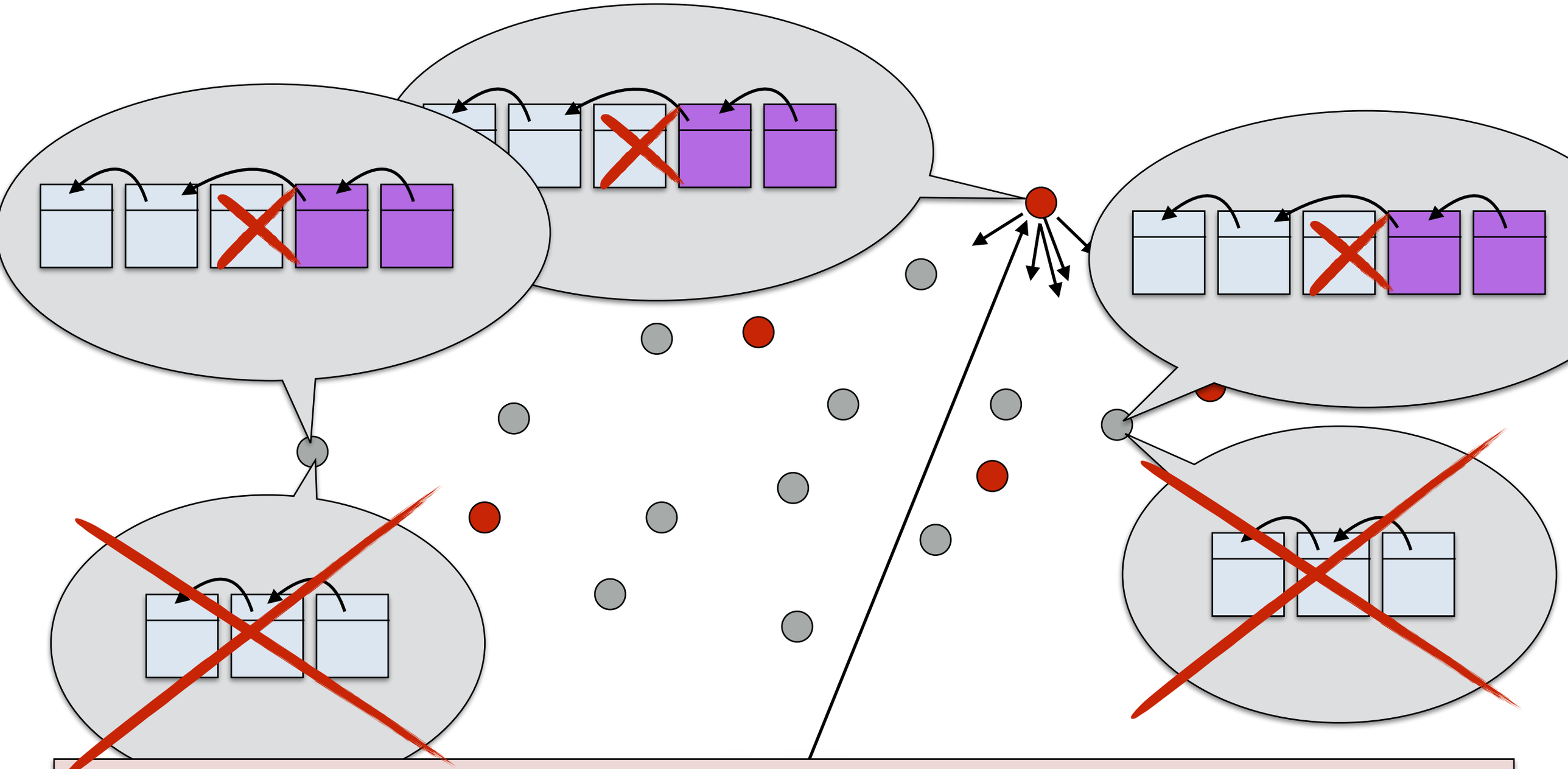


Example: Angel picks an malicious node twice



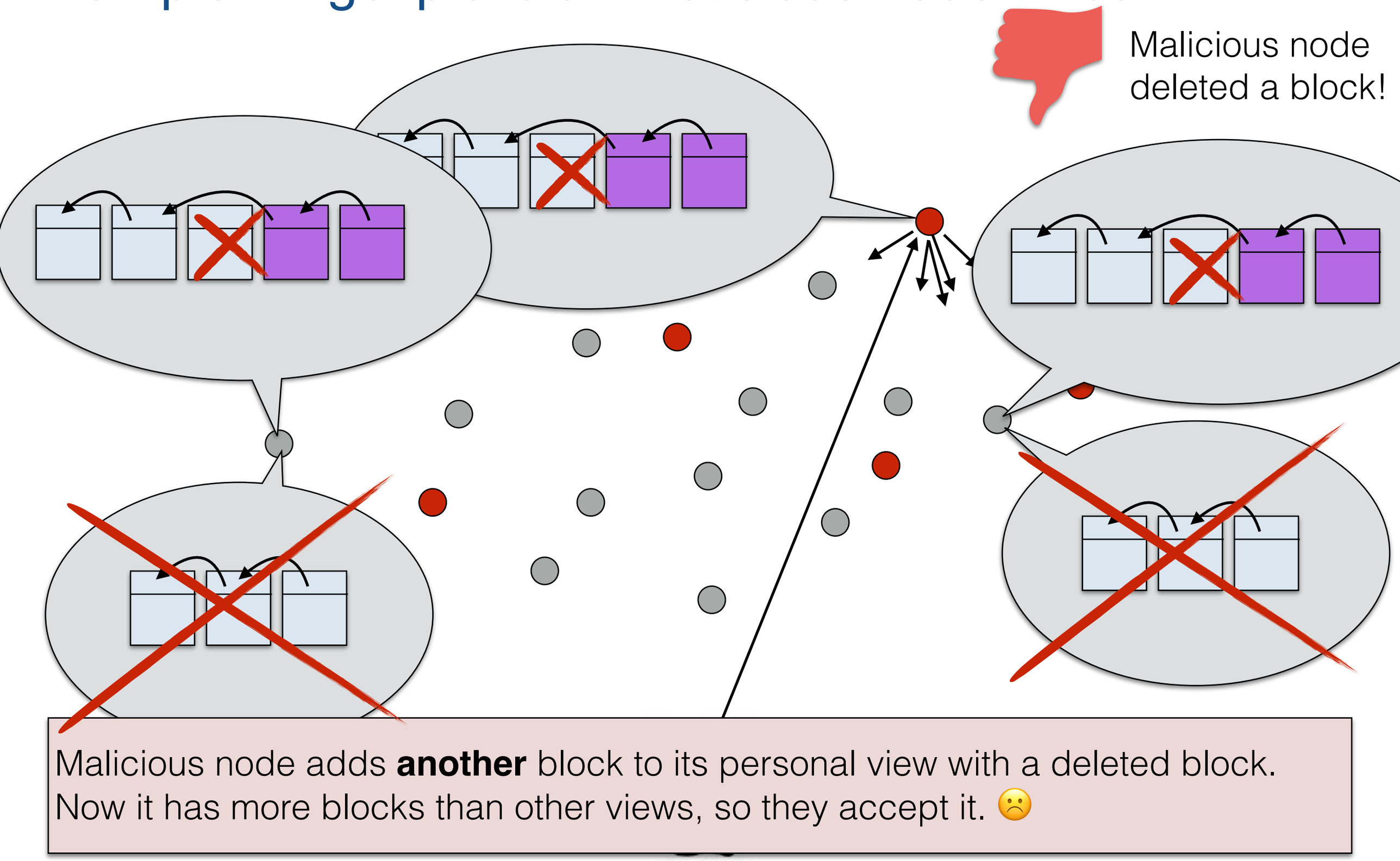
Malicious node adds **another** block to its personal view with a deleted block.
Now it has more blocks than other views, so they accept it. ☹

Example: Angel picks an malicious node twice



Malicious node adds **another** block to its personal view with a deleted block.
Now it has more blocks than other views, so they accept it. ☹

Example: Angel picks an malicious node twice



Malicious node adds **another** block to its personal view with a deleted block.
Now it has more blocks than other views, so they accept it. ☹

Intuition for Consensus

- Malicious nodes need to grow a longer chain in order to cheat.
- Being picked several times allows them to grow their chain.
- But when honest nodes are picked, the chain grow and others accept it.

Intuition for Consensus

- Malicious nodes need to grow a longer chain in order to cheat.
- Being picked several times allows them to grow their chain.
- But when honest nodes are picked, the chain grow and others accept it.

Theorem: Assume the majority of nodes are honest. Once the honest nodes are “ahead” of the malicious nodes, the chance the malicious nodes will ever “catch up” decreases exponentially in the size of the lead.

Intuition for Consensus

- Malicious nodes need to grow a longer chain in order to cheat.
- Being picked several times allows them to grow their chain.
- But when honest nodes are picked, the chain grow and others accept it.

Theorem: Assume the majority of nodes are honest. Once the honest nodes are “ahead” of the malicious nodes, the chance the malicious nodes will ever “catch up” decreases exponentially in the size of the lead.

- After several blocks have been added, nodes can be confident that past blocks can’t be deleted by malicious nodes — They’re too far behind to catch up.

Intuition for Consensus

- Malicious nodes need to grow a longer chain in order to cheat.
- Being picked several times allows them to grow their chain.
- But when honest nodes are picked, the chain grow and others accept it.

Theorem: Assume the majority of nodes are honest. Once the honest nodes are “ahead” of the malicious nodes, the chance the malicious nodes will ever “catch up” decreases exponentially in the size of the lead.

- After several blocks have been added, nodes can be confident that past blocks can’t be deleted by malicious nodes — They’re too far behind to catch up.

Going forward: “Confirmations” for a transaction are subsequent blocks that follow the transaction’s block. More confirmations mean the transaction is “more official”.

Problem to solve later: Sybil attacks

Vulnerable to Sybil attacks: If there are a ton of malicious nodes, then the probability a malicious node is chosen is high.

Probability that angel picks honest node = # honest / total

Probability that angel picks malicious node = # malicious / total

Problem to solve later: Sybil attacks

Vulnerable to Sybil attacks: If there are a ton of malicious nodes, then the probability a malicious node is chosen is high.

Probability that angel picks honest node = # honest / total

Probability that angel picks malicious node = # malicious / total

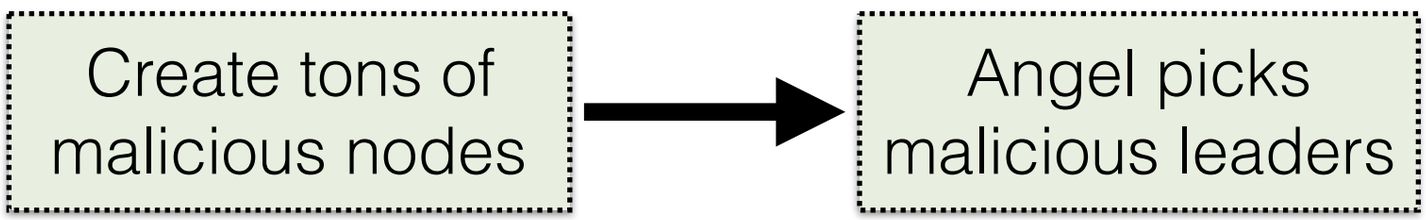
Create tons of malicious nodes

Problem to solve later: Sybil attacks

Vulnerable to Sybil attacks: If there are a ton of malicious nodes, then the probability a malicious node is chosen is high.

Probability that angel picks honest node = # honest / total

Probability that angel picks malicious node = # malicious / total



Problem to solve later: Sybil attacks

Vulnerable to Sybil attacks: If there are a ton of malicious nodes, then the probability a malicious node is chosen is high.

Probability that angel picks honest node = # honest / total

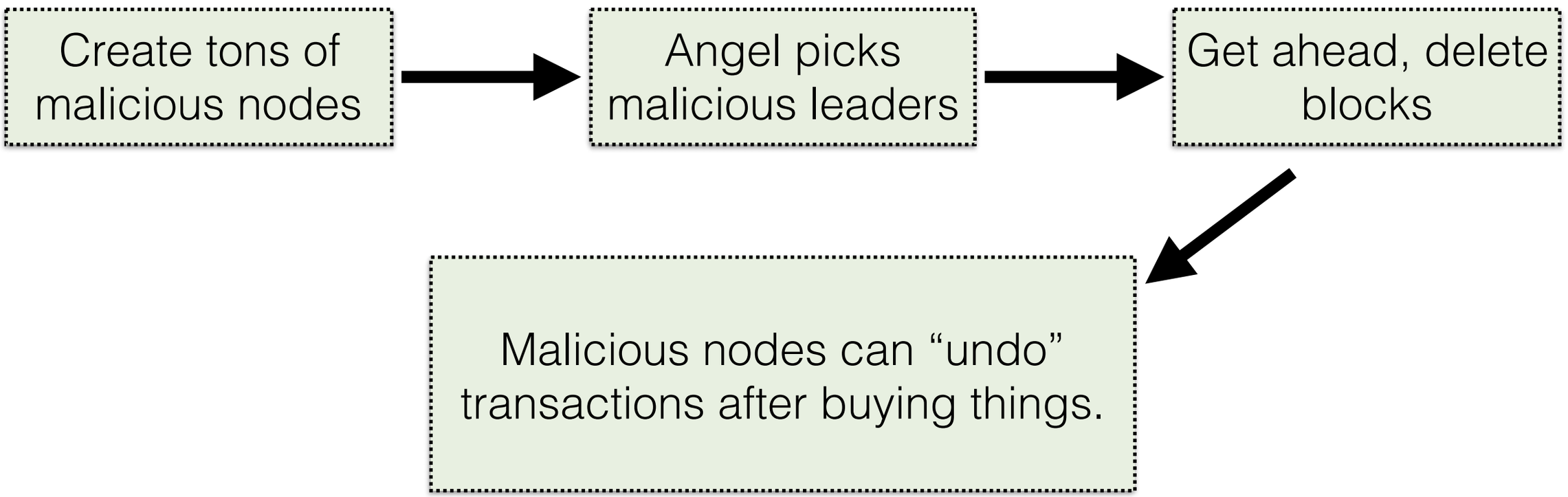
Probability that angel picks malicious node = # malicious / total

Problem to solve later: Sybil attacks

Vulnerable to Sybil attacks: If there are a ton of malicious nodes, then the probability a malicious node is chosen is high.

Probability that angel picks honest node = # honest / total

Probability that angel picks malicious node = # malicious / total



Lecture 2 Outline

1. Cryptographic Hash Functions
   - Blockchains
   - Proofs of Work
2. Putting DCash “on the blockchain”, with an authority
3. The idea of decentralization
4. Decentralized DCash with an Angel
5. **Decentralized DCash via proofs-of-work — Next time.**

The End