Blockchains & Cryptocurrencies

Smart Contracts & Ethereum

Instructor: Matthew Green & Abhishek Jain
Spring 2024

News?

News?

Technology

Ethereum Devs Poised to Split Blockchain's Next Big Upgrade, 'Pectra,' in Two

On Thursday, Ethereum developers will decide if Pectra will be split into two forks. If developers agree on the split, the first package could come in 2025, as early as February.

By Margaux Nijkerk Sep 18, 2024 at 7:00 a.m. EDT

Verkle Trees and Statelessness

EIP-2935 introduces Verkle trees and enables statelessness in Pectra. Verkle trees eliminate the need for nodes to store the network's state locally, and thereby greatly reduce the computational burden on validators.

Validator Light Clients

In addition to Verkle trees and statelessness, there are three additional 'parallel upgrades' likely to be included in Pectra. These upgrades include the development of validator light clients, which allow users to validate the network without downloading the entire Ethereum blockchain.

Light clients aim to enhance Ethereum's decentralisation by enabling validation through "resource-constrained devices like tablets and cell phones," according to Ethereum developers.

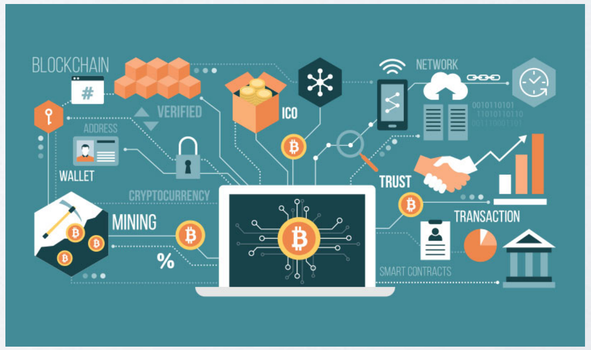
Obsolete Historical Data

With nodes no longer required to store Ethereum's full block history, EIP-4444 formalises the deletion of historical data from full nodes after a specified time period, further reducing computational demands and improving node decentralisation.

"The amount of data needed to run a node would decrease from multiple terabytes to... potentially running a node in RAM," said Buterin.

Today

- From Bitcoin to smart contracts
- Ethereum
- Applications of smart contracts



Bitcoin blocks

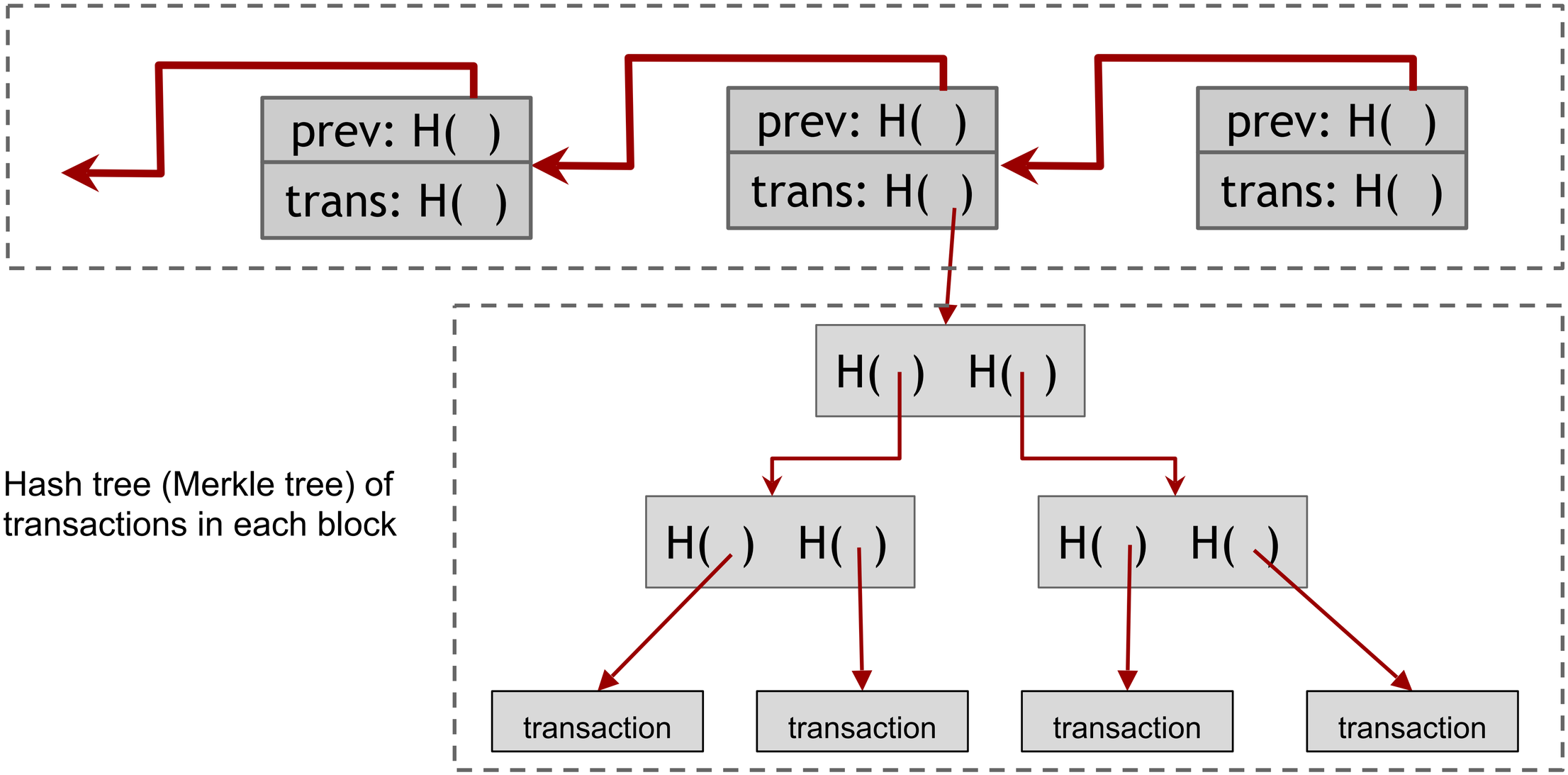
Bitcoin blocks

Why bundle transactions together?

- Single unit of work for miners
- Limit length of hash-chain of blocks
  - Faster to verify history

Bitcoin block structure

Hash chain of blocks



Hash tree (Merkle tree) of transactions in each block

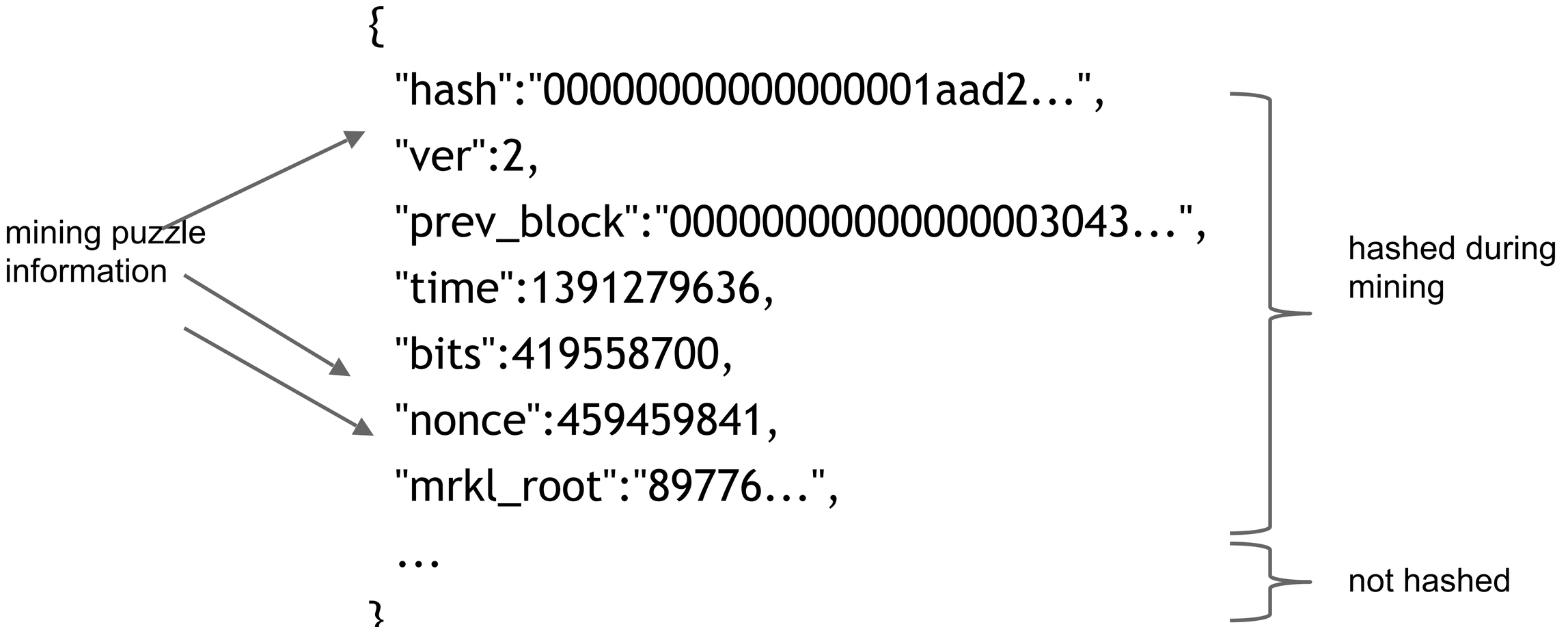
The real deal: a Bitcoin block

block header

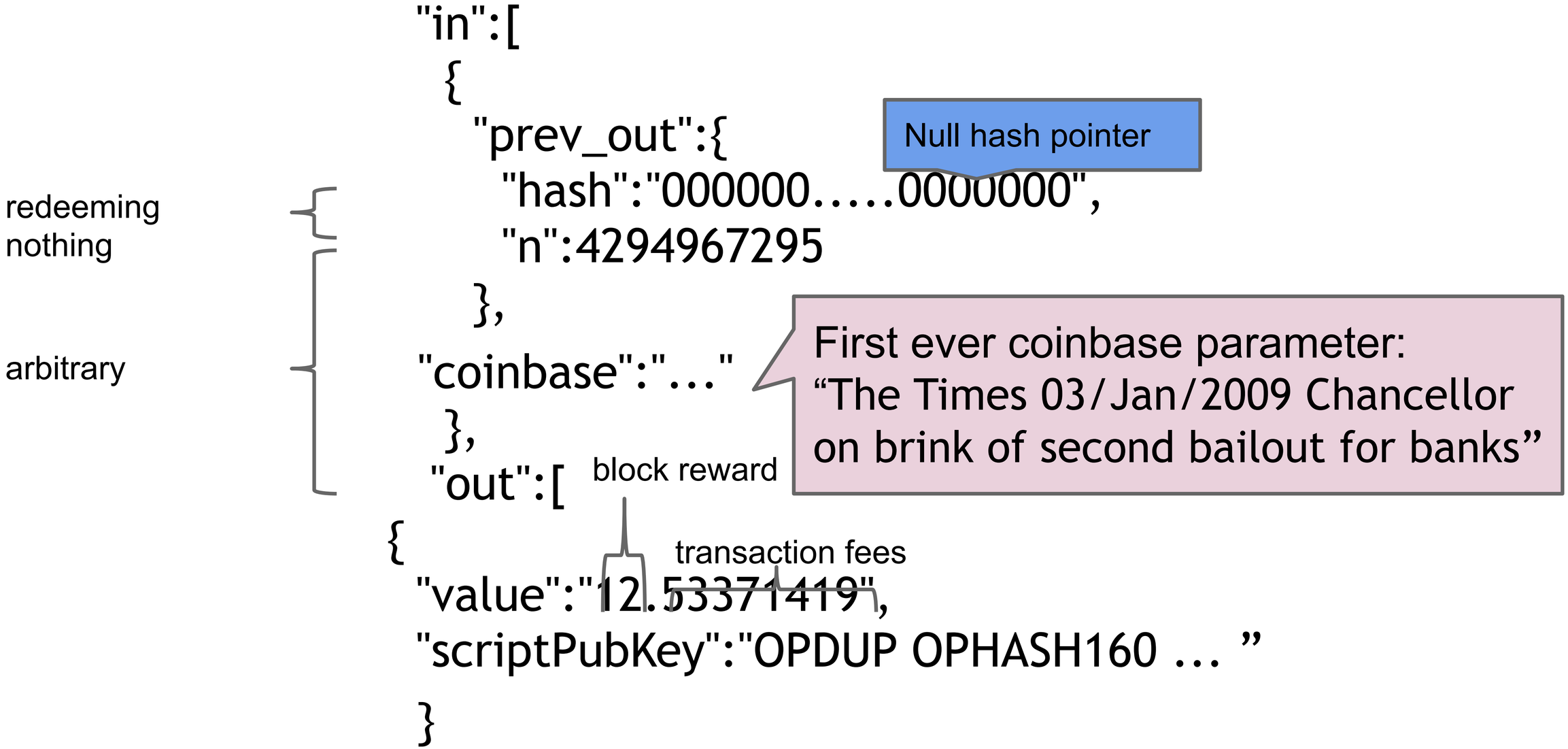
transaction data


```
{
  "hash":"00000000000000001aad2...",
  "ver":2,
  "prev_block":"00000000000000003043...",
  "time":1391279636,
  "bits":419558700,
  "nonce":459459841,
  "mrkl_root":"89776...",
  "n_tx":354,
  "size":181520,
  "tx":[
    ...
  ],
  "mrkl_tree":[
    "6bd5eb25...",
    ...
    "89776cdb..."
  ]
}
```

The real deal: a Bitcoin block header



The real deal: coinbase transaction



See for yourself!

Transaction View information about a bitcoin transaction

151b750d1f13e76d84e82b34b12688811b23a8e3119a1cba4b4810f9b0ef408d

1KryFUt9tXHvaoCYTNPbqpWPJKQ717YmL5

1KvrdrQ3oGqMAiDTMEYCcdDSnVaGNW2YZh 1.0194 BTC
1KryFUt9tXHvaoCYTNPbqpWPJKQ717YmL5 3.458 BTC

9 Confirmations 4.4774 BTC

Summary	
Size	257 (bytes)
Received Time	2014-08-05 01:55:25
Included In Blocks	314018 (2014-08-05 02:00:40 +5 minutes)
Confirmations	9 Confirmations
Relayed by IP	Blockchain.info
Visualize	View Tree Chart

Inputs and Outputs	
Total Input	4.4775 BTC
Total Output	4.4774 BTC
Fees	0.0001 BTC
Estimated BTC Transacted	1.0194 BTC
Scripts	Show scripts & coinbase

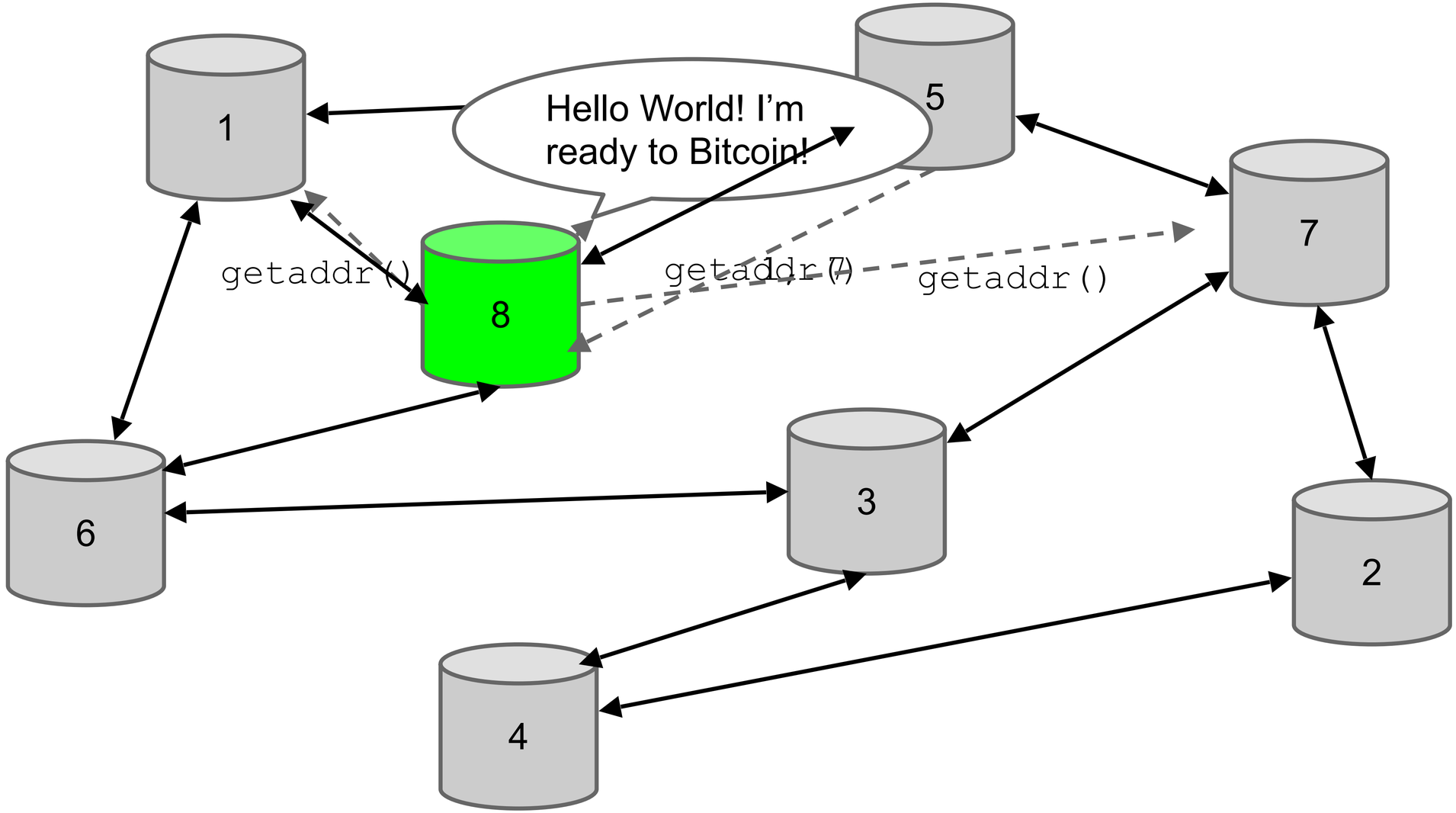
blockchain.info (and many other sites)

The Bitcoin network

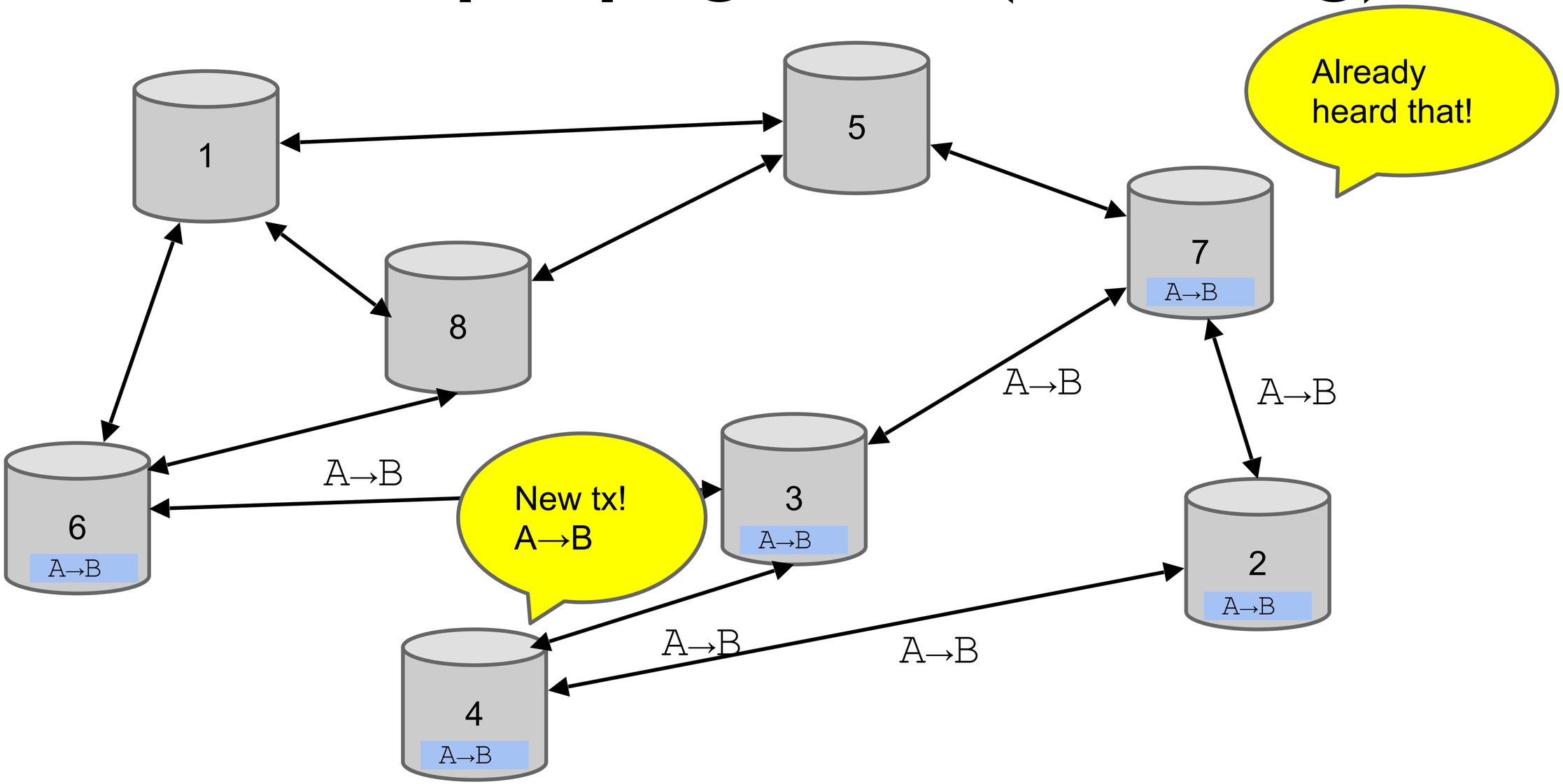
Bitcoin P2P network

- Ad-hoc protocol (runs on TCP port 8333)
- Ad-hoc network with random topology
- All nodes are equal
- New nodes can join at any time
- Forget non-responding nodes after 3 hr

Joining the Bitcoin P2P network

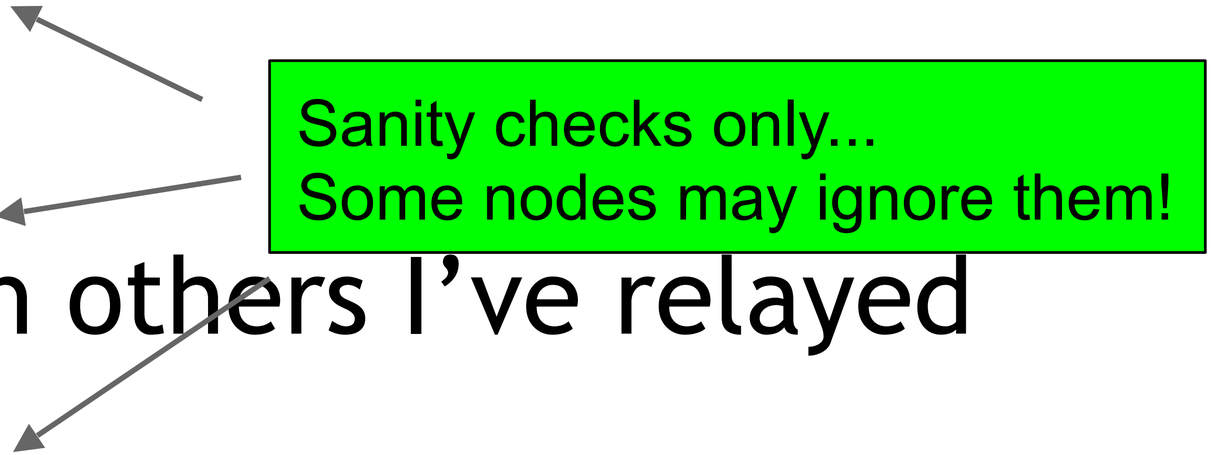


Transaction propagation (flooding)



Should I relay a proposed transaction?

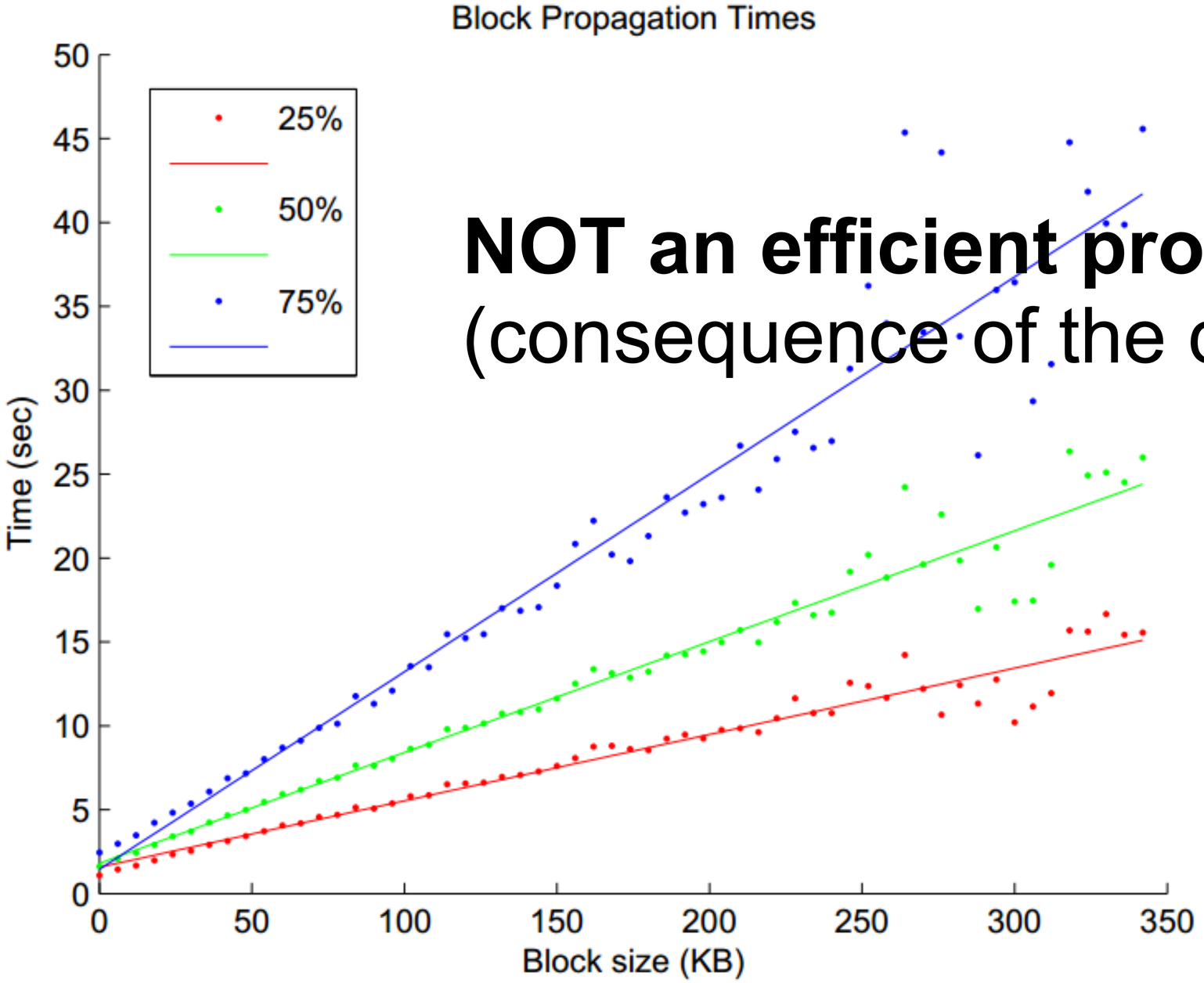
- Transaction valid with current block chain
- (default) script matches a whitelist
  - Avoid unusual scripts
- Haven’t seen before
  - Avoid infinite loops
- Doesn’t conflict with others I’ve relayed
  - Avoid double-spends



Race conditions

Transactions or blocks may *conflict*

- Default behavior: accept what you hear first
- Network position matters
- Miners may implement other logic!



NOT an efficient protocol
(consequence of the design)

Source: Yonatan Sompolinsky and Aviv Zohar: “Accelerating Bitcoin’s Transaction Processing” 2014

Fully-validating nodes

- Permanently connected
- Store entire block chain
- Hear and forward every node/transaction

Thin/SPV clients (not fully-validating)

Idea: don’t store everything

- Store block headers only
- Request transactions as needed
  - To verify incoming payment
- Trust fully-validating nodes

Limitations & improvements

Hard-coded limits in Bitcoin

- 10 min. average creation time per block
- 1 M bytes in a pre-Segwit block
  - Now larger (~4MB? With Segwit)
- 21M total bitcoins maximum
- 50,25,12.5... bitcoin mining reward

These affect economic balance of power too much to change now

Cryptographic limits in Bitcoin

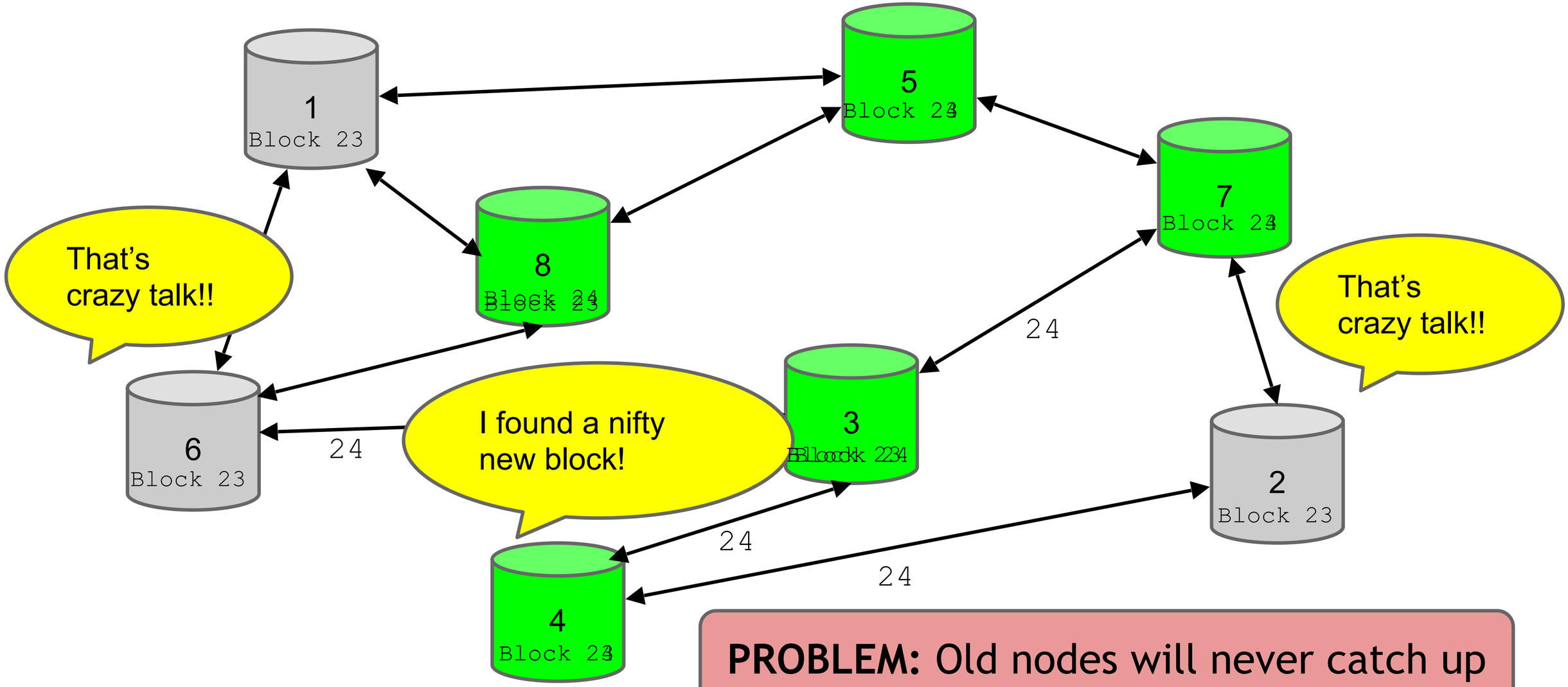
- Only 1 signature algorithm (ECDSA/P256)
- Hard-coded hash functions

Some of these crypto primitives used here might break by 2040 (e.g., collision-found in hash function, or powerful quantum computer breaks ECDSA)...

Why not update Bitcoin software to overcome these limitations?

- Many of these changes require “hard forks”, which are currently considered unacceptable

“Hard-forking” changes to Bitcoin



PROBLEM: Old nodes will never catch up

Soft forks

Observation: we can add new features which only *limit* the set of valid transactions

Need majority of nodes to enforce new rules

Old nodes will approve

RISK: Old nodes might mine now-invalid blocks

Soft fork example: pay to script hash

<signature>
<<pubkey> OP_CHECKSIG>

OP_HASH160
<hash of redemption script>
OP_EQUAL

Old nodes will just approve the hash, not run the embedded script

Hard forks

- New op codes
- Changes to size limits
- Changes to mining rate
- Many small bug fixes

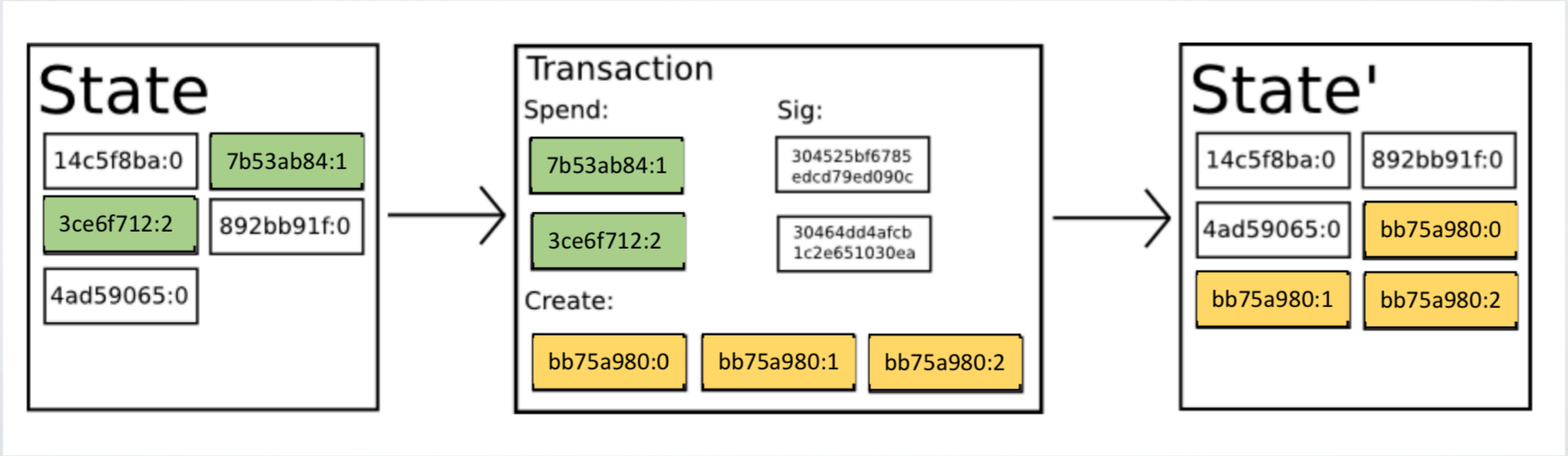
Currently seem unlikely to happen

Many of these issues addressed by Altcoins

Bitcoin transactions as “state transition”

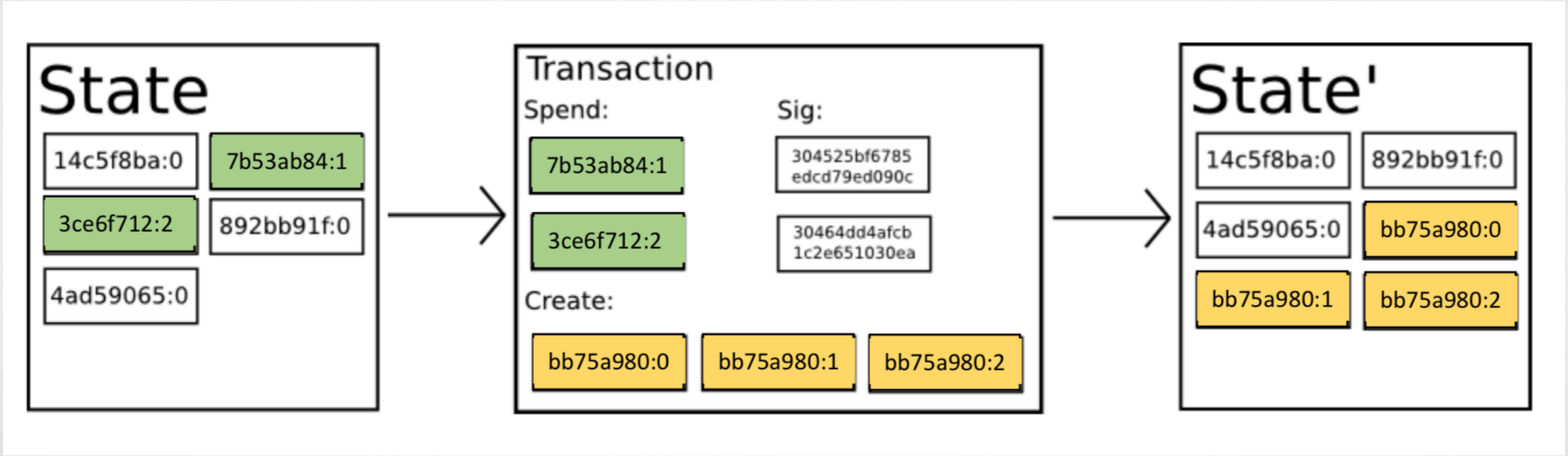
- **Let’s consider each Bitcoin transaction as a state transition function**
  - What is the input state?
  - What is the output state?
  - What does a Transaction do to the state?

- Input state: list of coins available for spending (UTXO set)
- Transaction: set of instructions for updating the UTXO set
- Output state: Updated UTXO set



- Input state: list of coins available for spending (UTXO set)
- Transaction: set of instructions for updating the UTXO set
- Output state: Updated UTXO set

  (script determines if a single UTXO is satisfied)

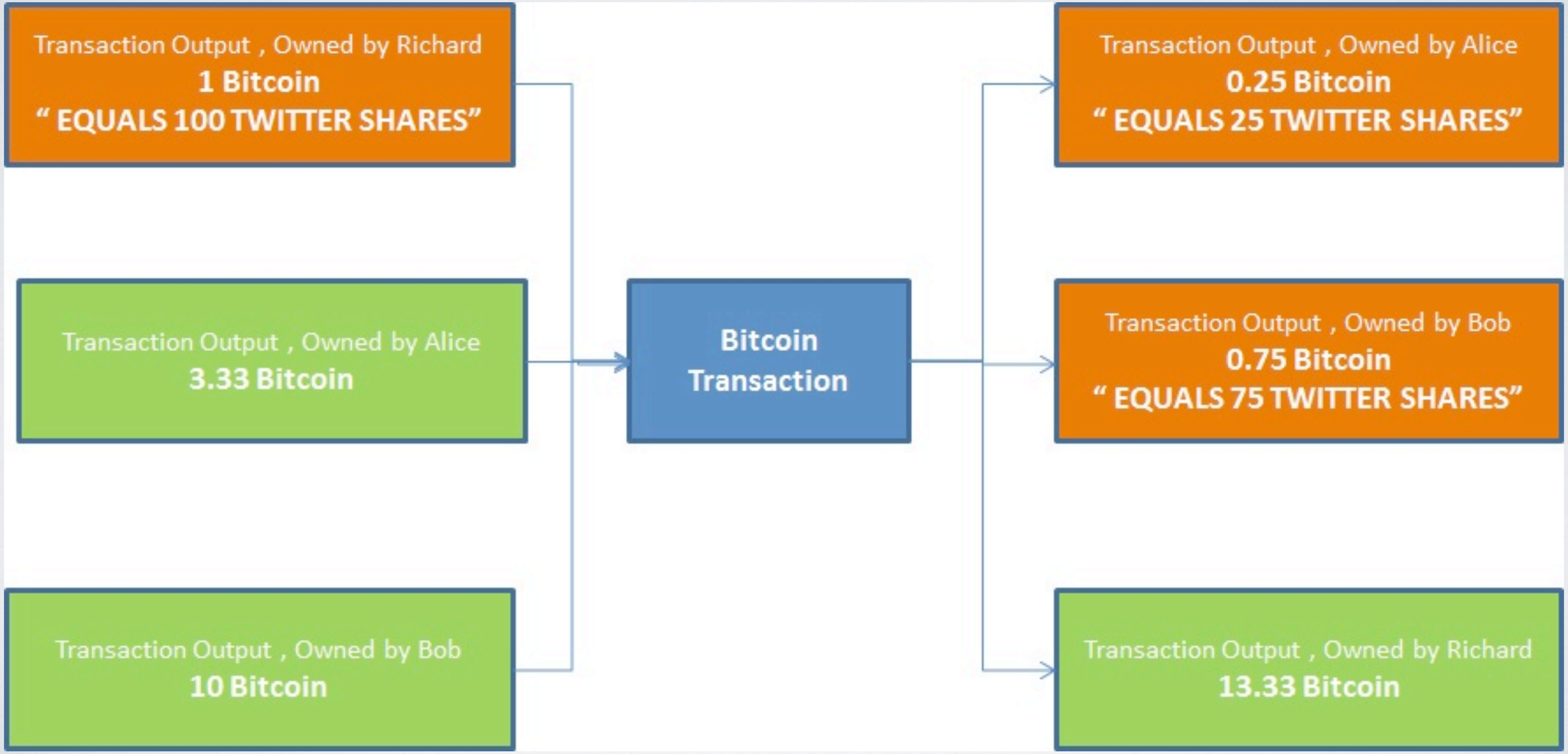


What if we want more powerful programming capabilities?

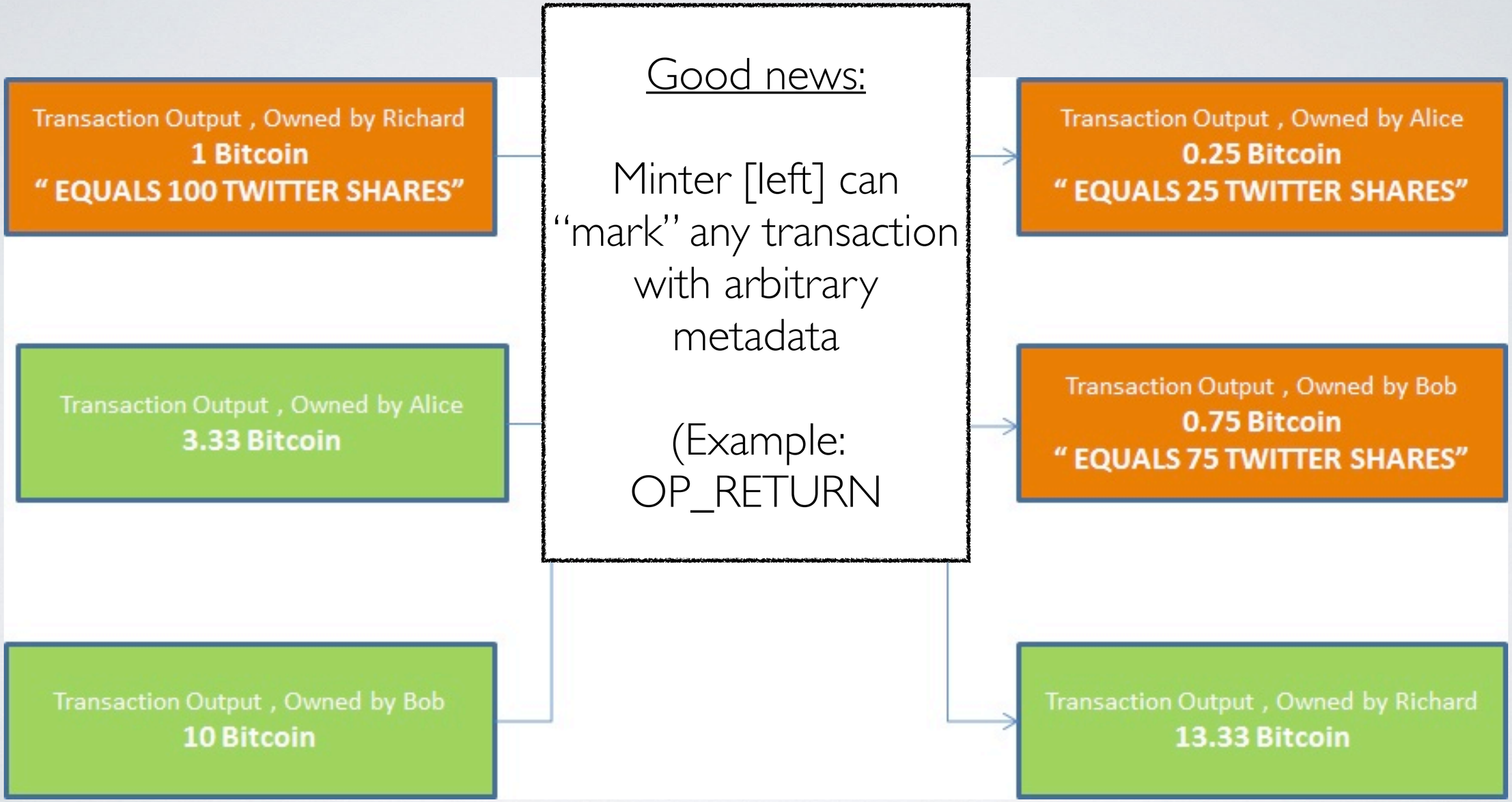
Example: custom tokens

- Bitcoin supports a single currency (BTC)
  - You can spin up a new network (e.g., Litecoin, Dogecoin, etc.)
  - Can we support a second currency on Bitcoin?
    - Applications: coupons, stablecoins, NFTs
    - Major calls: **Mint** (e.g., centralized party), **Burn**, **Pay**

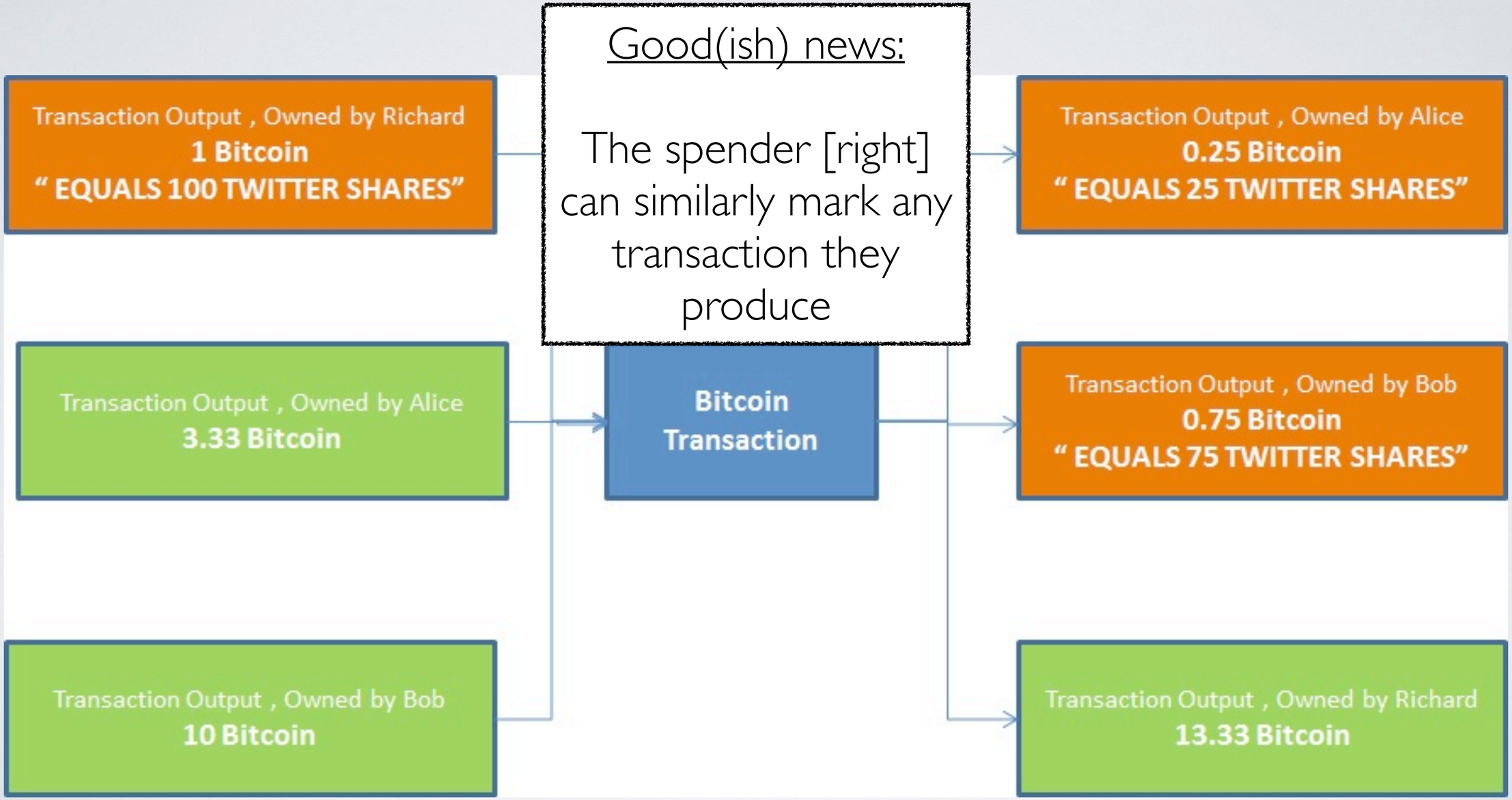
“Colored coins” on Bitcoin



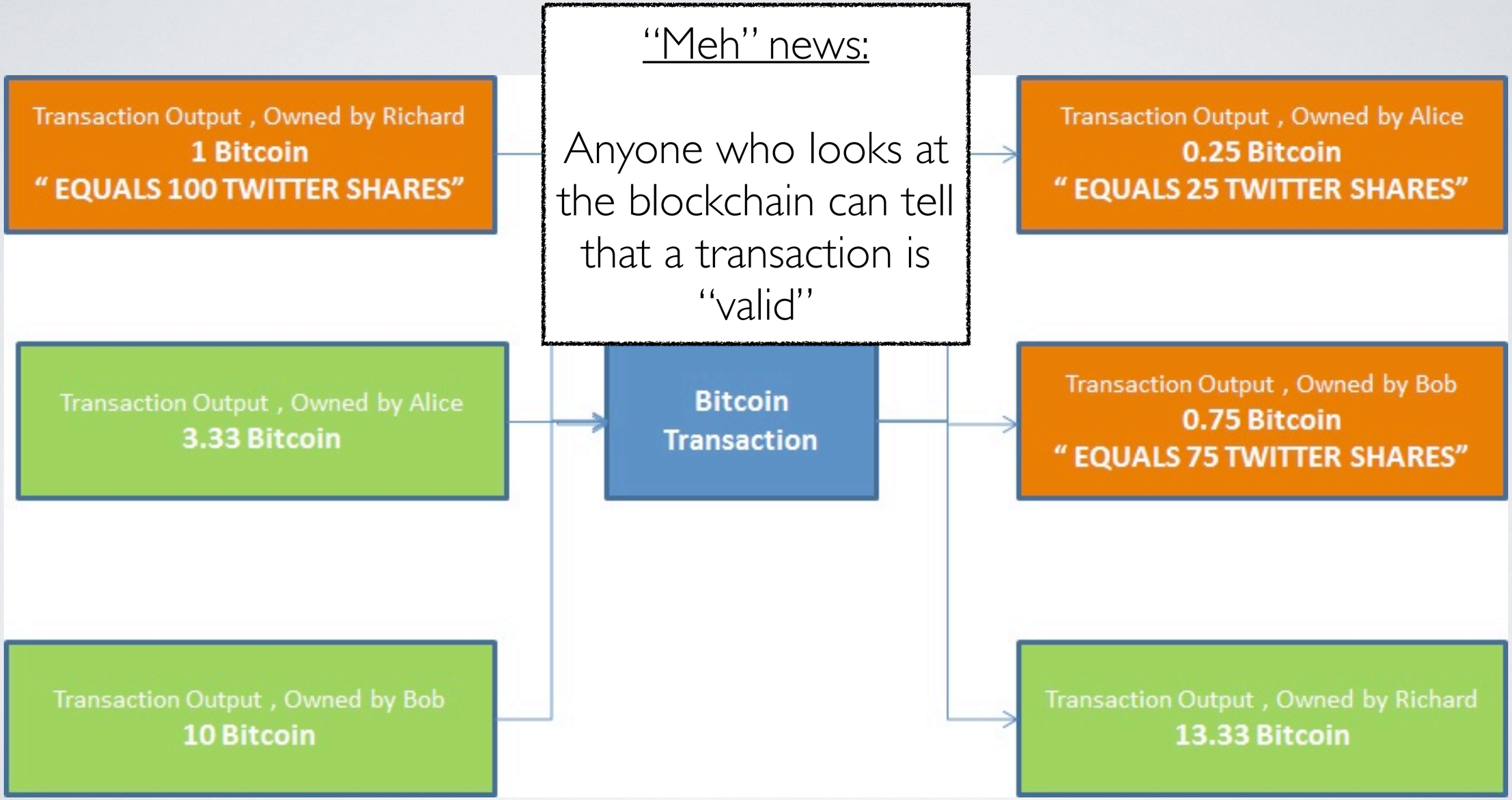
“Colored coins” on Bitcoin



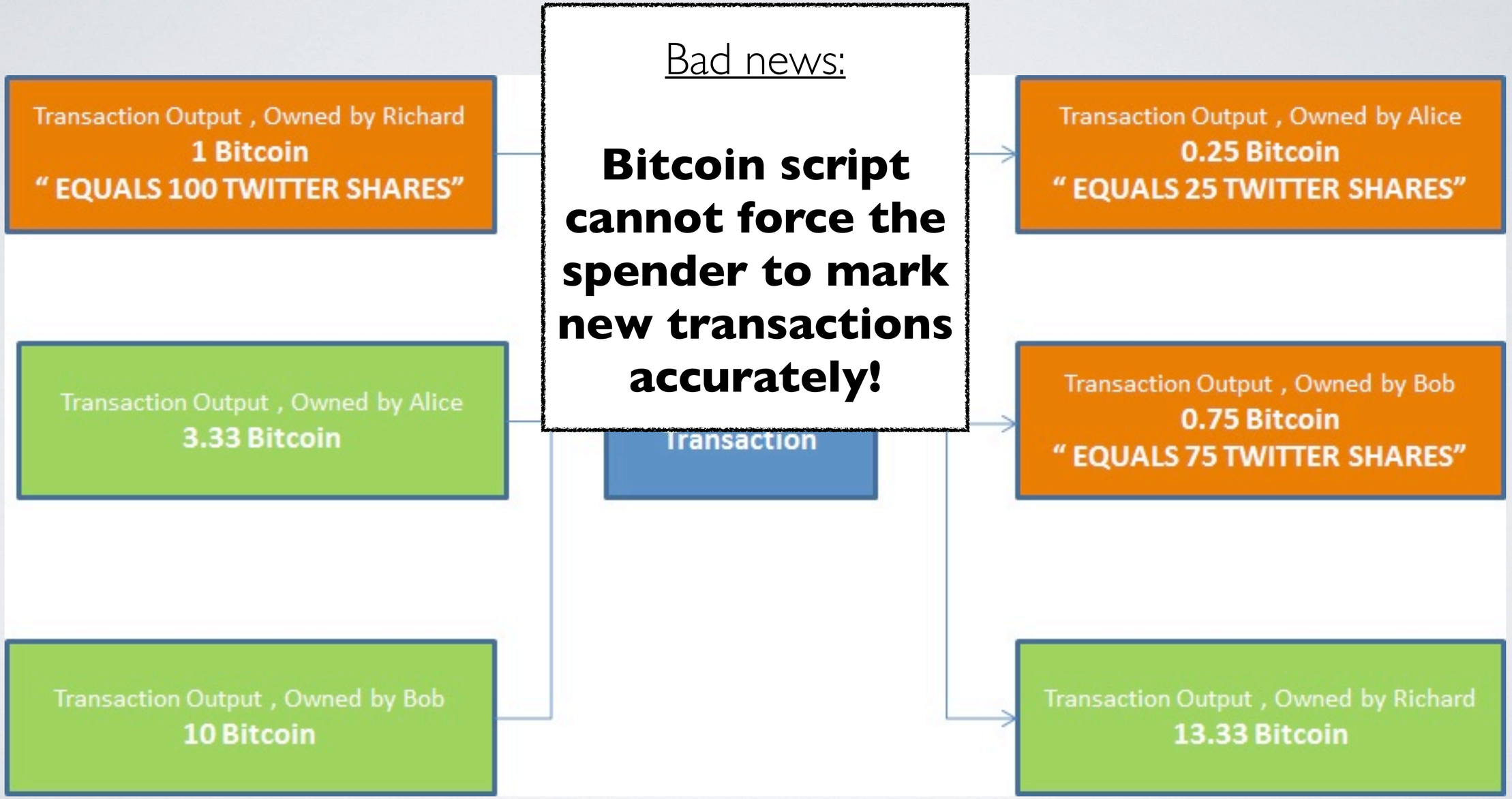
“Colored coins” on Bitcoin



"Colored coins" on Bitcoin



“Colored coins” on Bitcoin



Example 2: prediction market

- Simple program that executes the following:
  - Accepts "bets" on one of two possible outcomes (e.g., Presidential election outcomes)
  - Maintains odds based on all previous bets (simple calculation)
  - Receives a signed message from a pre-chosen judge (e.g., "Candidate A wins!")
  - Pays funds to all winners

Example 3: decentralized exchange

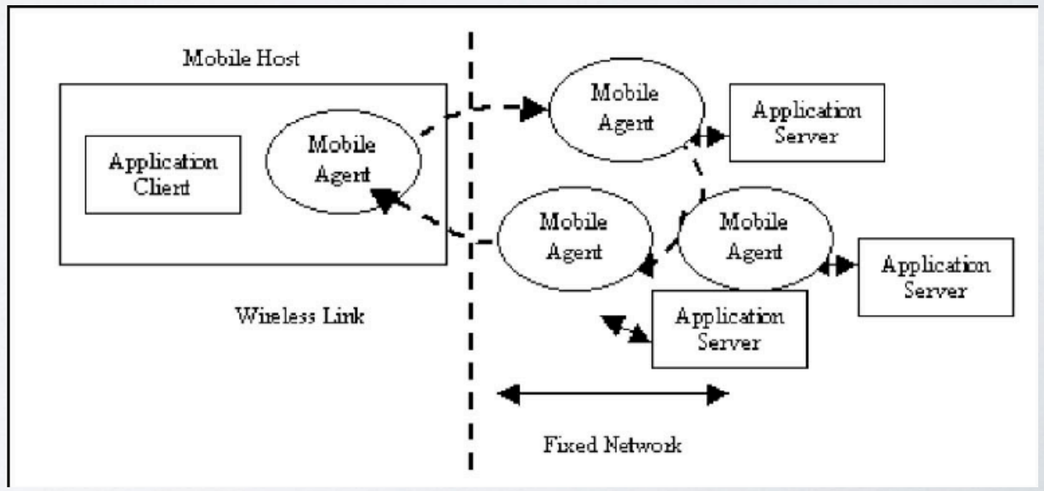
- Assume we have at least two different "tokens"
  - Accept deposits of Token A/Token B
    - Keep a separate for both tokens for each user
  - Build a program that can accept "bids" and "asks" to exchange Token A for Token B
  - Match bids to complete trades
  - Allow withdrawals of the exchanged tokens

“Smart contracts”

- Idea proposed by Nick Szabo (1994)
  - If two users wish to establish a legal contract (e.g., escrow funds, pay out on specific conditions) they can write those conditions in software
  - The software will run autonomously, respond to inputs, evaluate the conditions
  - Contract program can receive and pay out funds according to its programming
  - “Code is law”

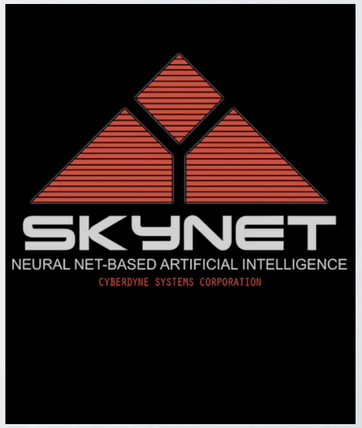
“Mobile Agents”

- Old idea from distributed computing (1990s)
  - Built software programs (“agents”) that can move between servers on a distributed computing system
  - Agents are survivable, can operate even if some computers go down
  - Agent must be portable, secure, and may need to compensate operators for resources consumed

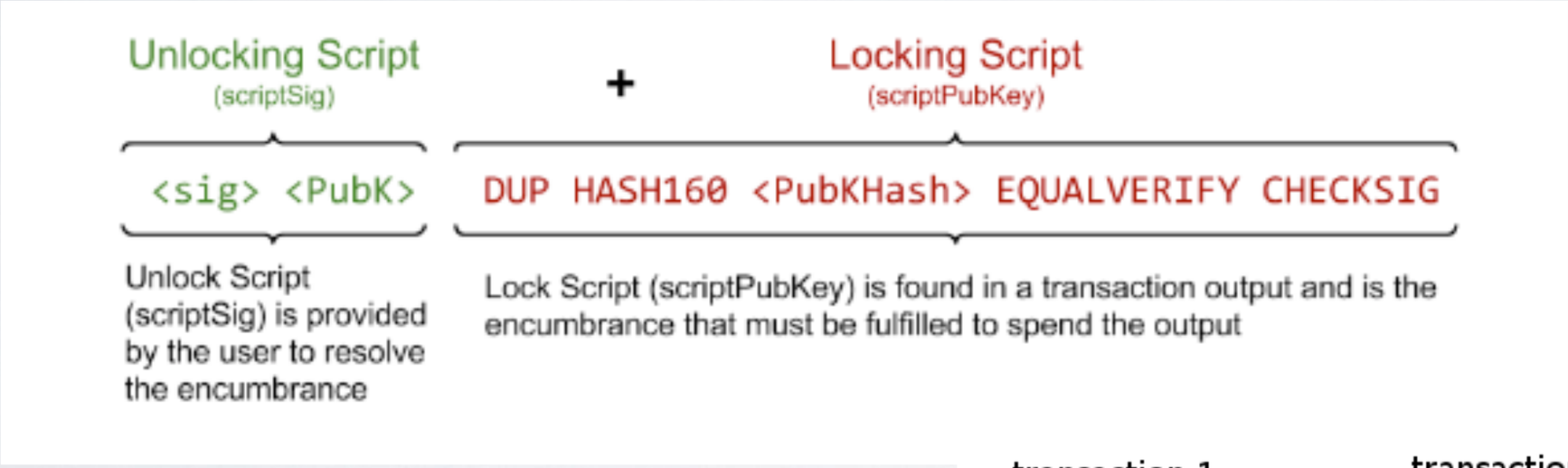


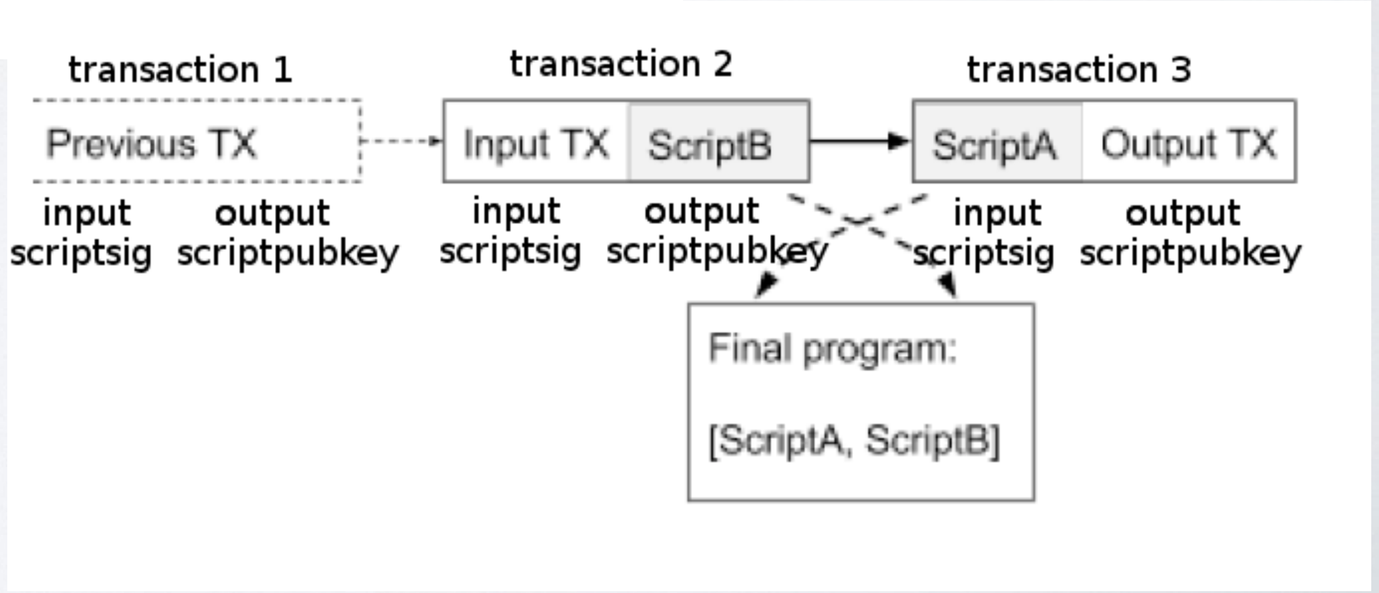
Two visions, same goal

- Run software on a computing network
  - Software is run by volunteer nodes (who come and go)
  - Software is accurate and can't be tampered with
  - Software can compensate users for the resource usage (e.g., payments)
  - Software can do useful things:
    e.g., control the disbursement of funds



Can we implement this on Bitcoin?





Can we implement this on Bitcoin?

- Each UTXO has a script. However:
  - Each script only runs once (when the UTXO is consumed)
  - **Scripts do not have access to "global" state** (they only have access to scriptSig and scriptPubKey data plus some limited global data like "block height".)
  - Scripts are not Turing complete (even in a limited sense)
  - Limited script opcodes (security reasons)

“Smart contracts”

- Idea proposed by Nick Szabo (1994)
  - Bitcoin script is a ‘contract’ in the sense that it provides programmable conditions for redeeming a coin
  - However, conditions are highly limited
    - Example: pay out a coin iff a user has a signing key
    - Example: implement a second currency/asset
    - Example: pay out a coin iff a candidate wins the US election
    - Example: pay out a coin iff a majority of users votes to invest in a service

Bitcoin ✅

Bitcoin ✅👎

Bitcoin 👎

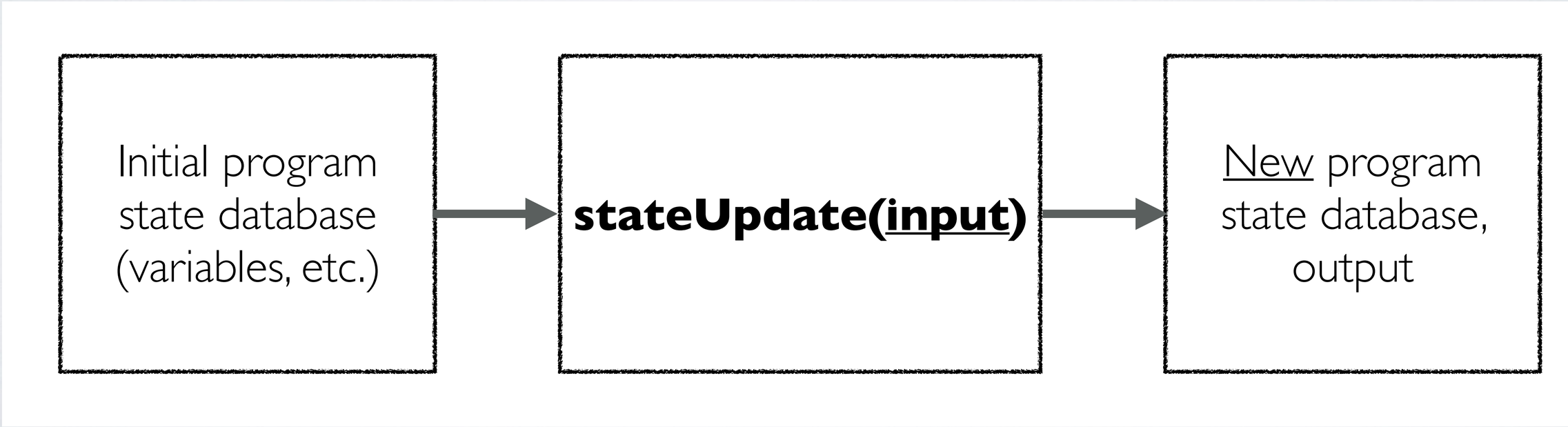
Bitcoin 👎

“Towards Ethereum”

Note: system we will discuss next is almost but not entirely unlike Ethereum.

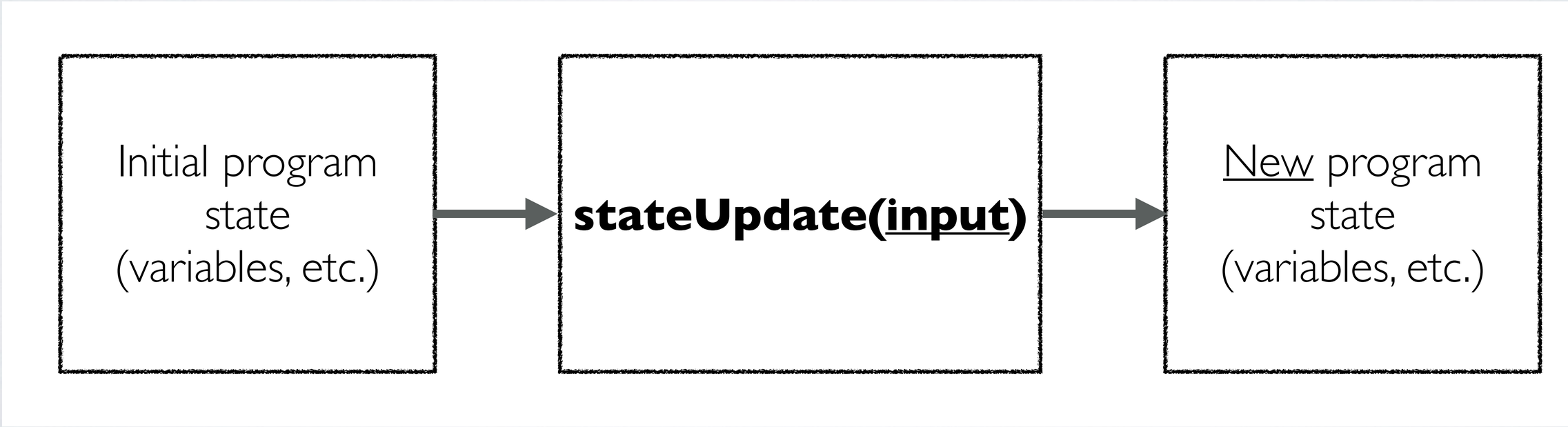
Generalizing programs

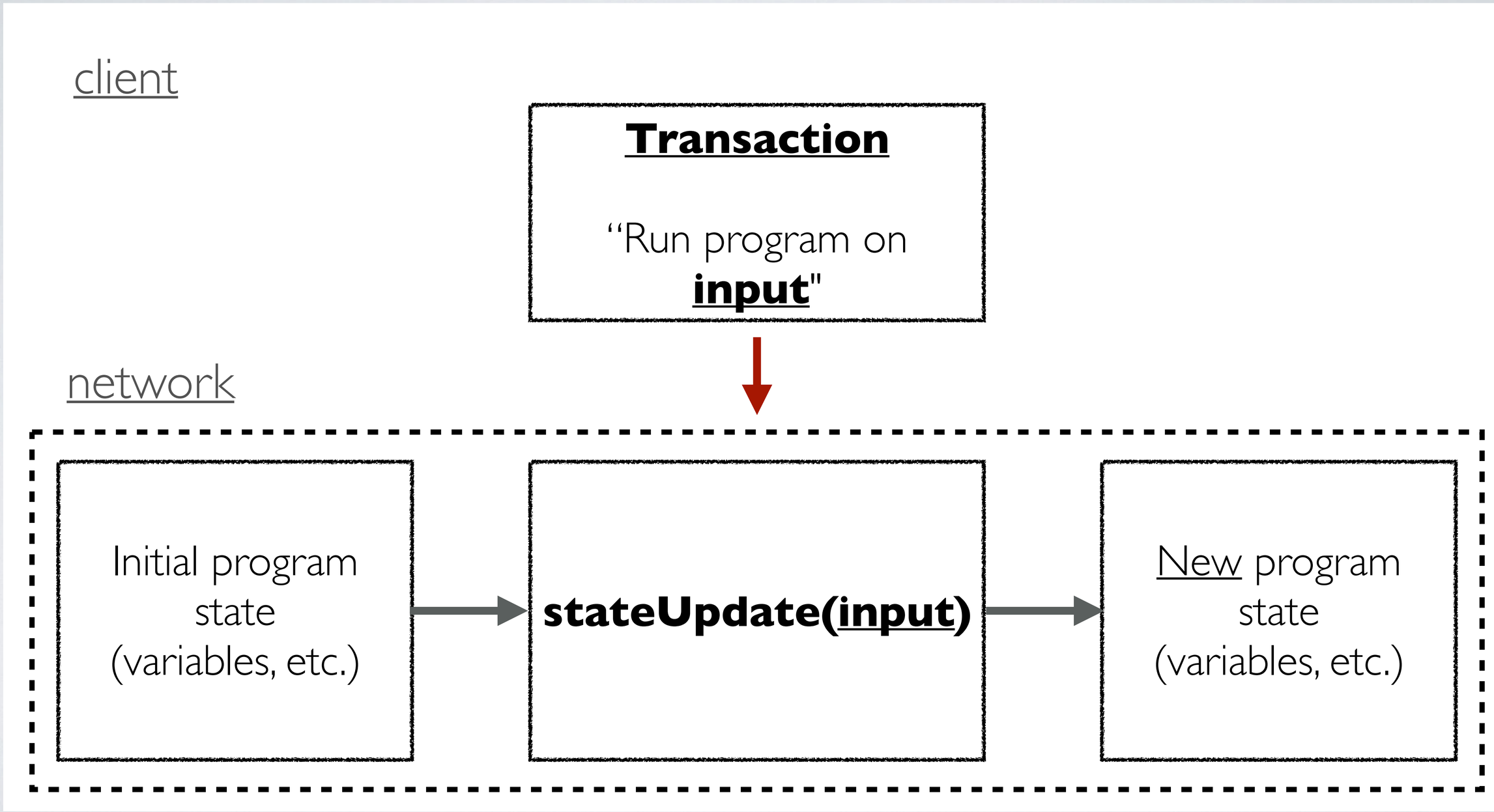
- Programs can be written as state update functions
  - Input: previous state of program, some "call input"
  - Output: new updated state for program, maybe some return value

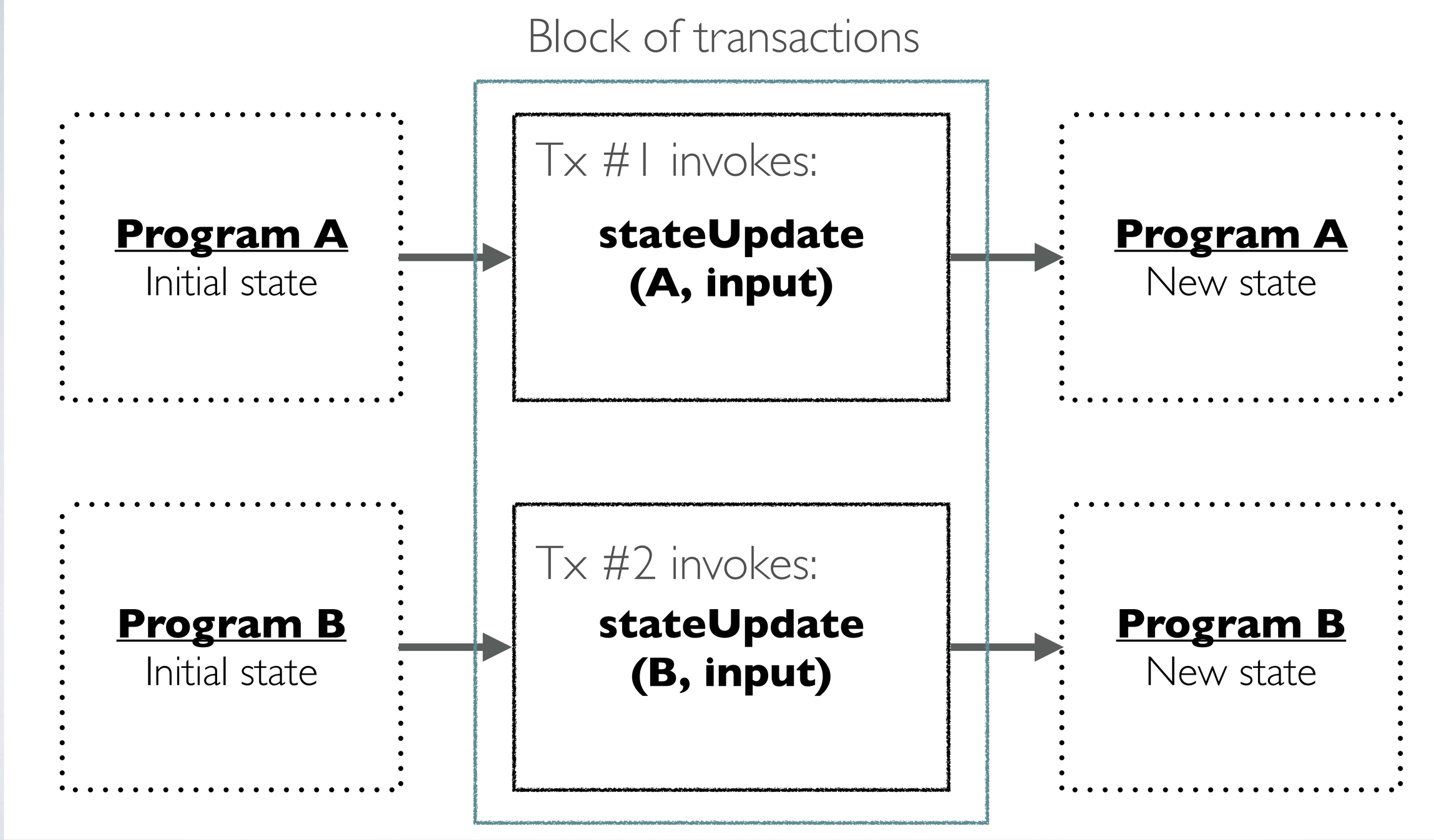


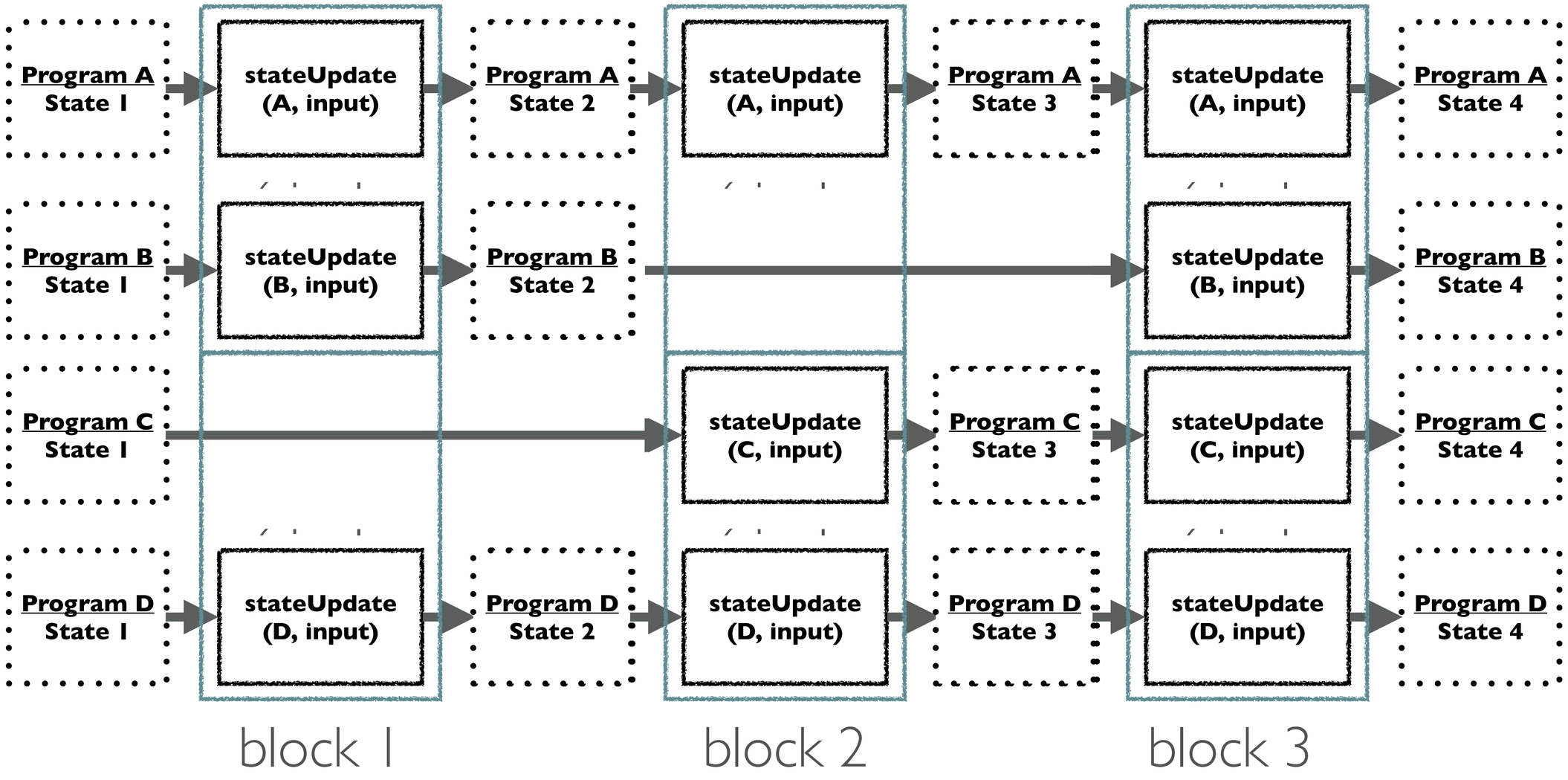
Blockchains for programs

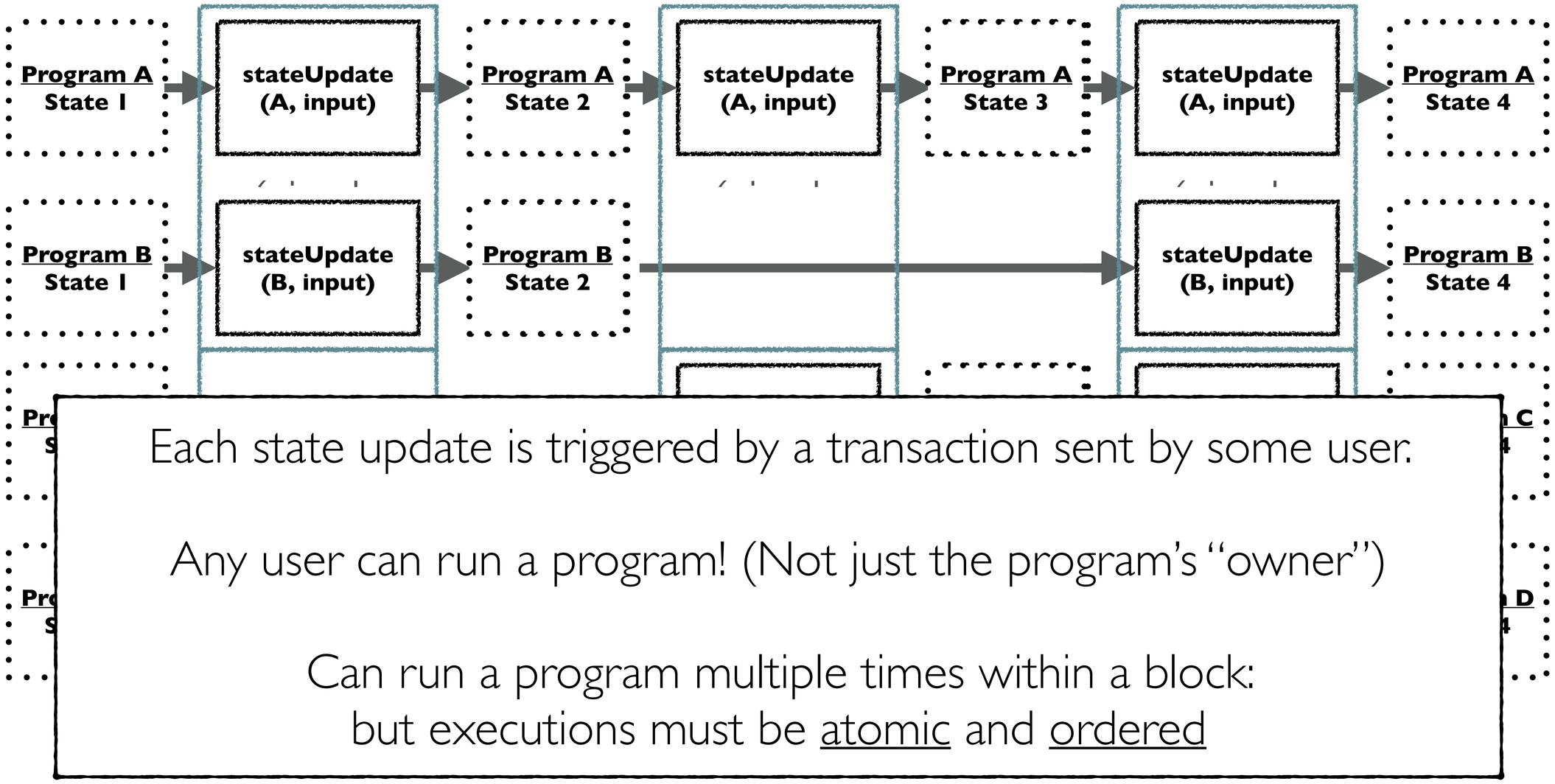
- Assume the blockchain has the program and a database
  - Each transaction simply "invokes" the program on **input**
  - The blockchain runs the program and updates its state



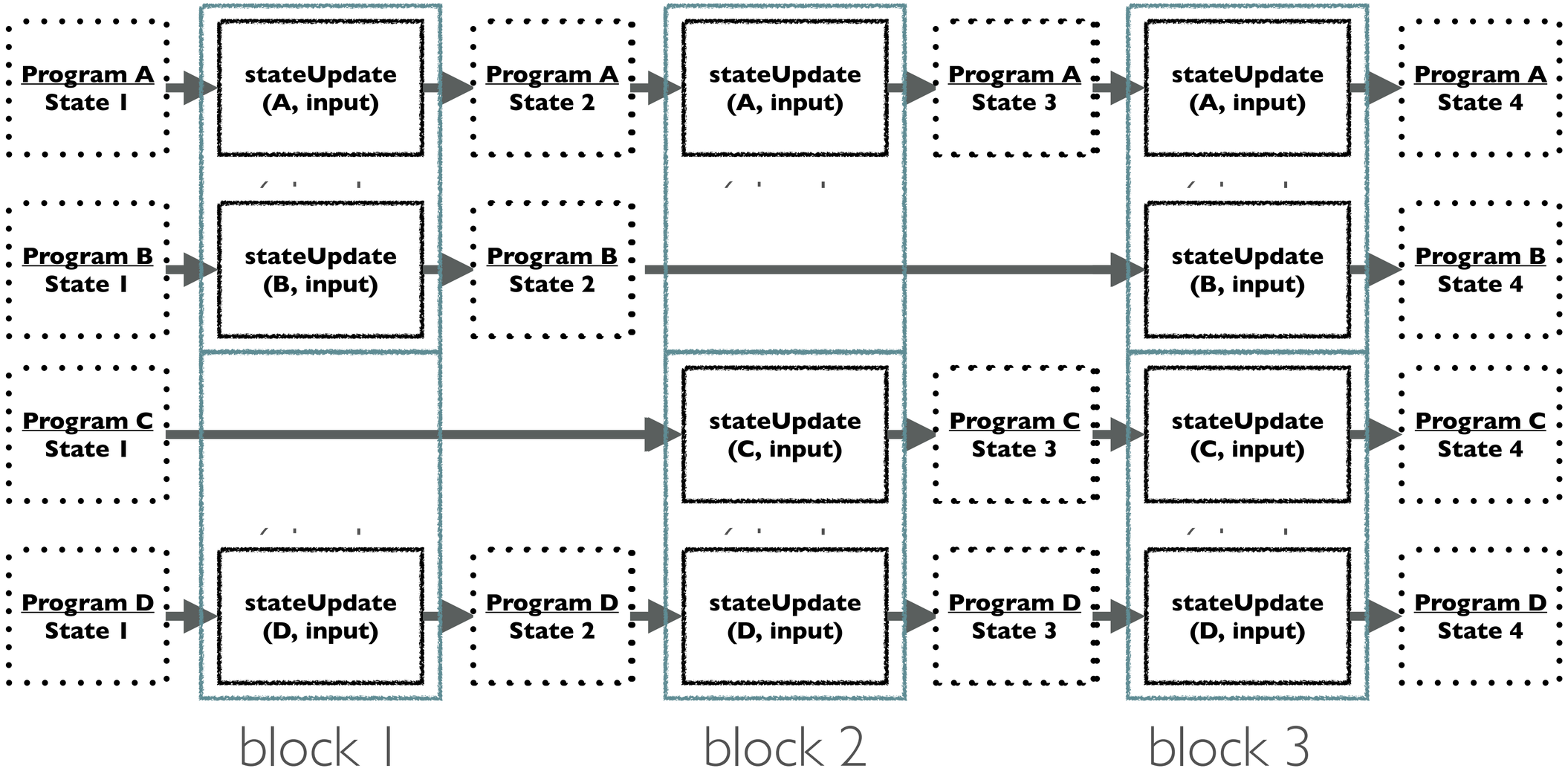

client
Transaction
"Run program on input"
network
Initial program state (variables, etc.)
stateUpdate(input)
New program state (variables, etc.)


Block of transactions
Program A
Initial state
Tx #1 invokes:
stateUpdate
(A, input)
Program A
New state
Program B
Initial state
Tx #2 invokes:
stateUpdate
(B, input)
Program B
New state

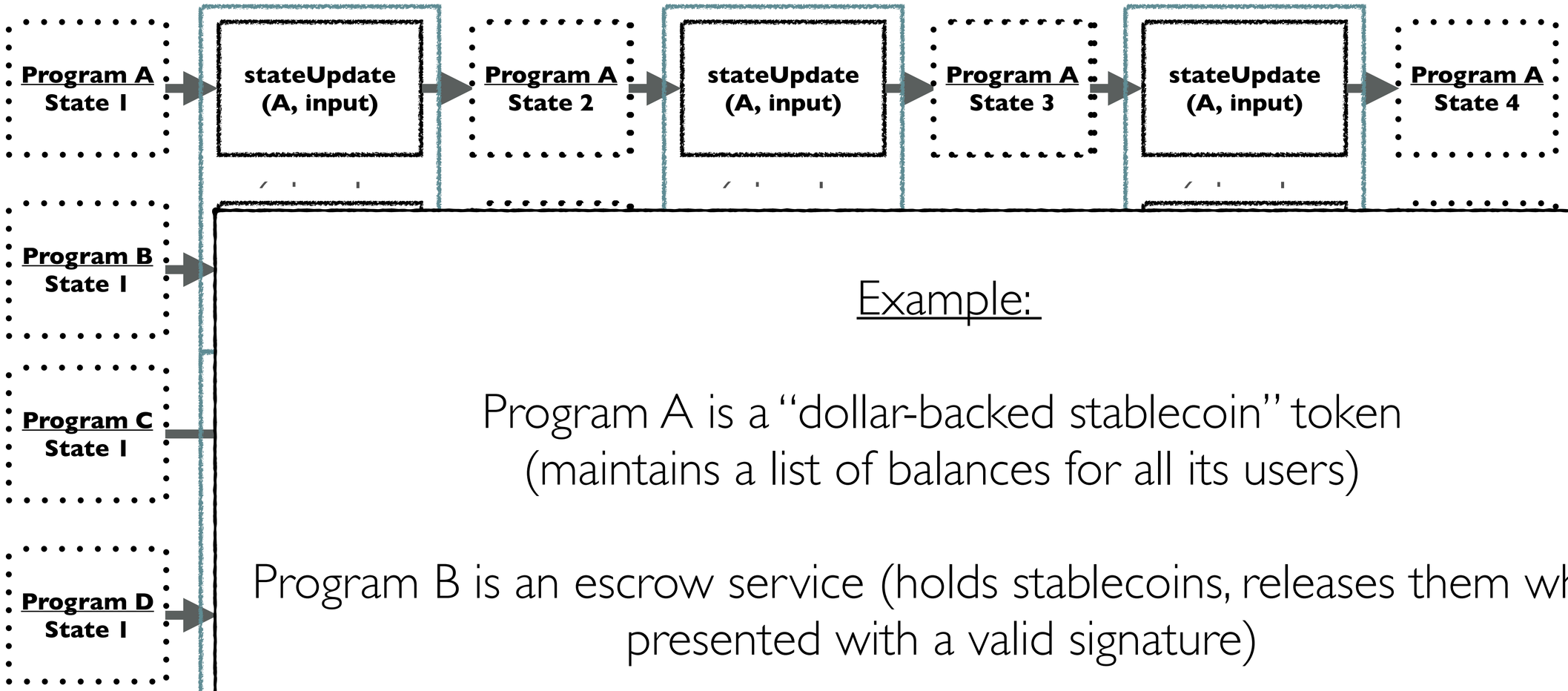

Program A
State 1
stateUpdate
(A, input)
Program A
State 2
stateUpdate
(A, input)
Program A
State 3
stateUpdate
(A, input)
Program A
State 4
Program B
State 1
stateUpdate
(B, input)
Program B
State 2
stateUpdate
(B, input)
Program B
State 4
Program C
State 1
stateUpdate
(C, input)
Program C
State 3
stateUpdate
(C, input)
Program C
State 4
Program D
State 1
stateUpdate
(D, input)
Program D
State 2
stateUpdate
(D, input)
Program D
State 3
stateUpdate
(D, input)
Program D
State 4
block 1
block 2
block 3


Program A
State 1
stateUpdate
(A, input)
Program A
State 2
stateUpdate
(A, input)
Program A
State 3
stateUpdate
(A, input)
Program A
State 4
Program B
State 1
stateUpdate
(B, input)
Program B
State 2
stateUpdate
(B, input)
Program B
State 4
Each state update is triggered by a transaction sent by some user.
Any user can run a program! (Not just the program's "owner")
Can run a program multiple times within a block:
but executions must be atomic and ordered

What if program A wants to alter the state of program B?



What if program A wants to alter the state of program B?



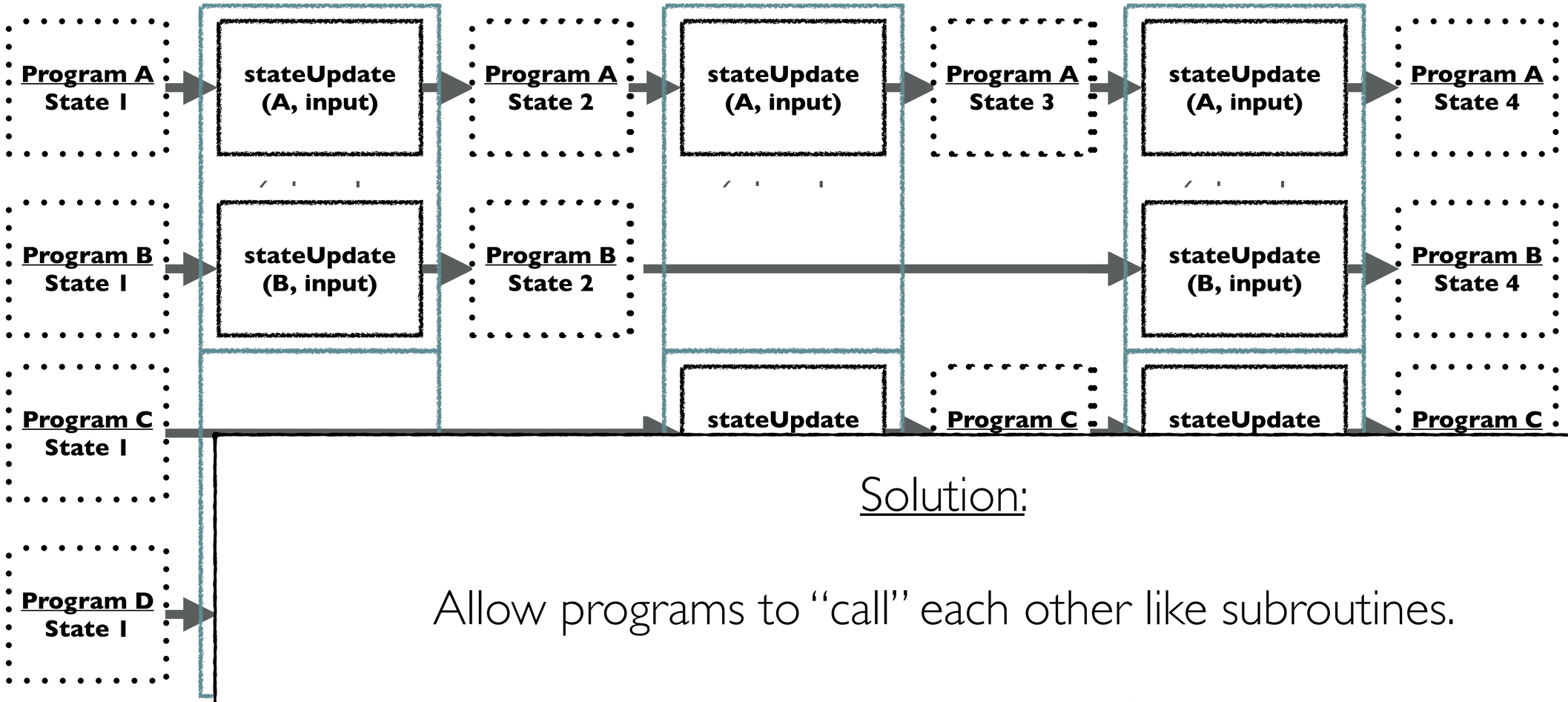
Example:

Program A is a "dollar-backed stablecoin" token
(maintains a list of balances for all its users)

Program B is an escrow service (holds stablecoins, releases them when presented with a valid signature)

How does Program B tell Program A to "pay" someone?

What if program A wants to alter the state of program B?

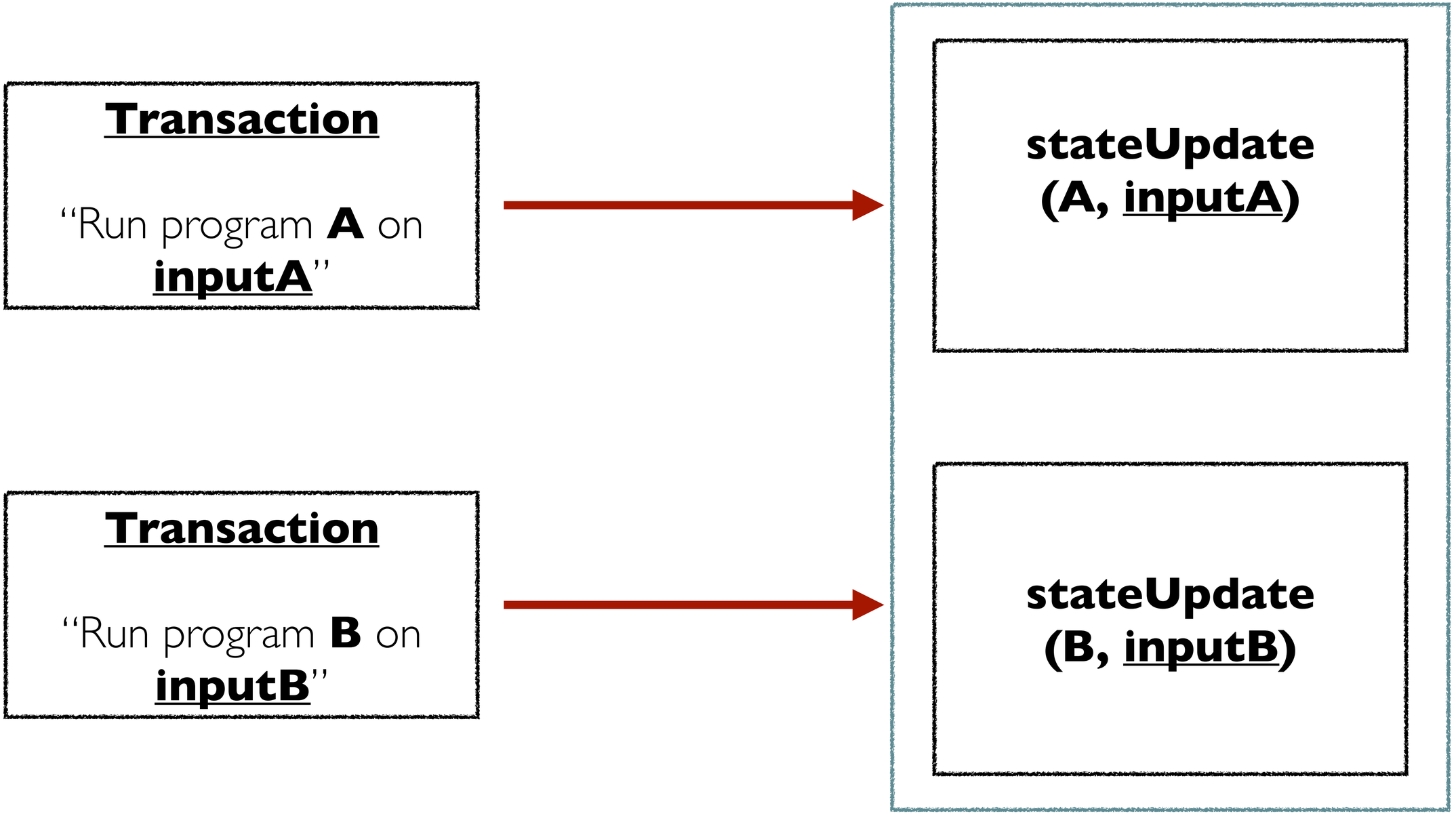


Solution:

Allow programs to "call" each other like subroutines.

Program B can tell A to "pay" another user (e.g., reduce one account balance, increase another user's account balance.)

External transactions



Transaction

"Run program **A** on **input**"

stateUpdate (A, input)

calls program B as a "subroutine"

stateUpdate (B, args)

Transaction

"Run program **A** on **input**"

stateUpdate (A, input)

calls program B as a "subroutine"

args

stateUpdate (B, args)

1. An external user calls program A
2. Program A calls program B as a subroutine

Transaction

"Run program **A** on **input**"

stateUpdate (A, input)

calls program B as a "subroutine"

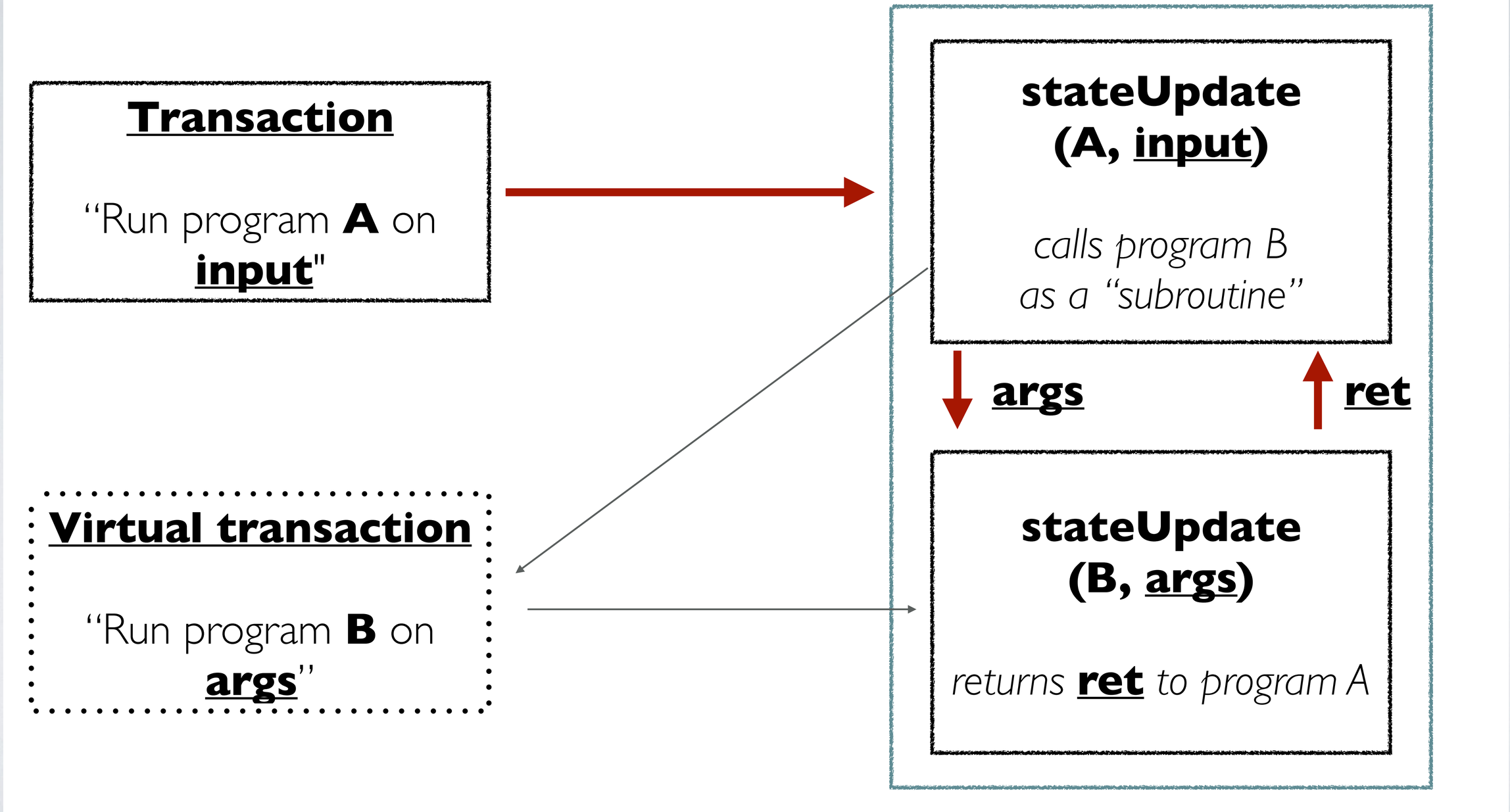
args

ret

stateUpdate (B, args)

returns **ret** *to program A*

1. An external user calls program A
2. Program A calls program B as a subroutine
3. Program B returns **ret** to A (and updates its own state)


Transaction
"Run program A on input"
stateUpdate (A, input)
calls program B as a "subroutine"
args
ret
Virtual transaction
"Run program B on args"
stateUpdate (B, args)
returns ret to program A

What you should be worried about right now:

When does B actually run?

Immediately?

Does A get "paused" and then B runs (maybe later) and then B generates a virtual transaction to start A up where it left off?

What if the call fails: can A get "stuck" forever?

Trans

"Run pro

inp

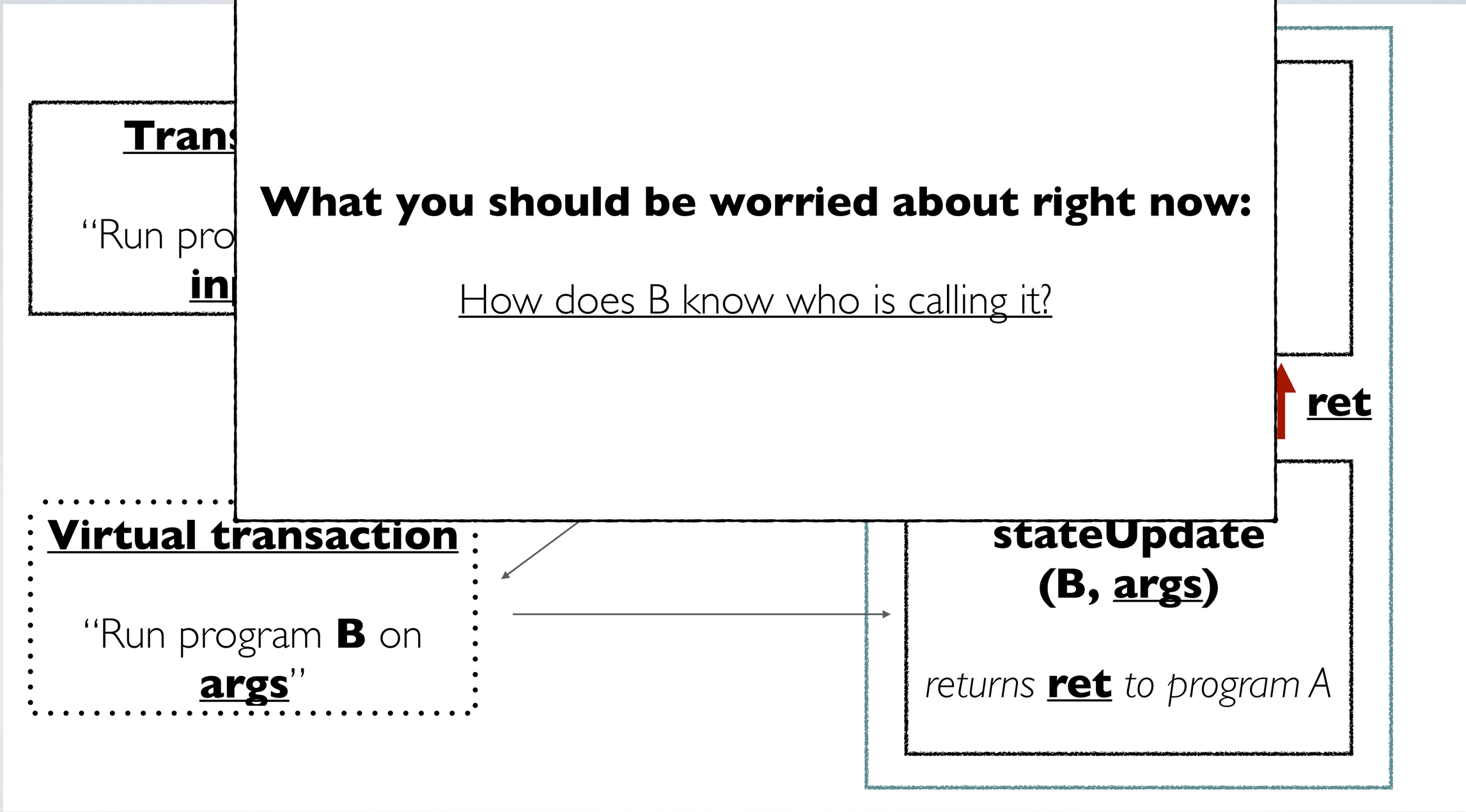
ret

Virtual t

"Run program **B** on **args**"

(B, args)

returns **ret** *to program A*


Trans
"Run pro
in
What you should be worried about right now:
How does B know who is calling it?
ret
Virtual transaction
"Run program B on args"
stateUpdate (B, args)
returns ret to program A

What you should be worried about right now:

How does B know who is calling it?

For real transactions, we can use addresses (public keys) and sign transactions with digital signatures (like Bitcoin)

For "virtual transactions", does the other program have an "address" and can it sign things?

Trans

"Run pro

inp

ret

Virtual transaction

"Run program **B** on **args**"

stateUpdate (B, args)

returns **ret** *to program A*

You should have many questions

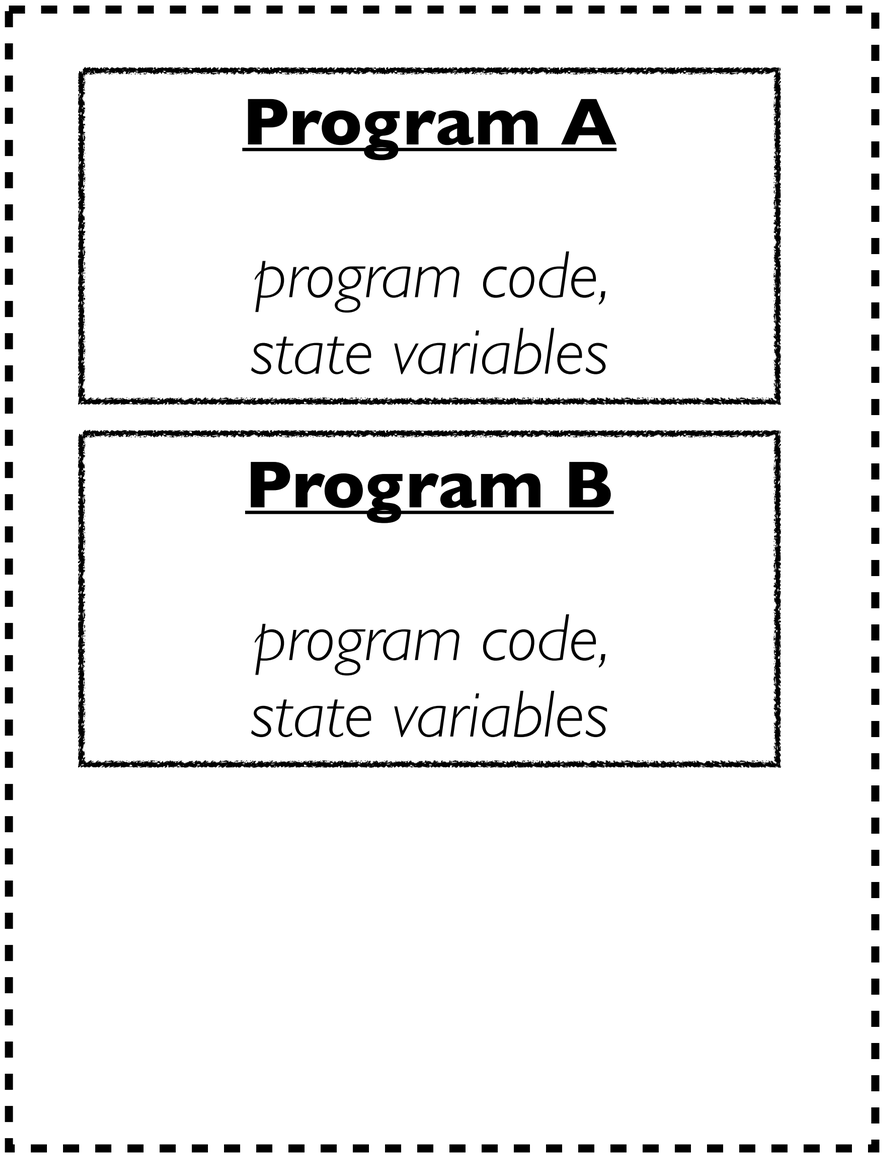
You should have many questions

- Where do these programs come from?
  - E.g., how do I submit a new program to the system
- What if running a program takes too long? Uses too much state?
- What language are these programs written in?
- Where does all the program state get stored?
- If this is a Bitcoin-like system, how do many computers stay in consensus?

Where do the programs come from?

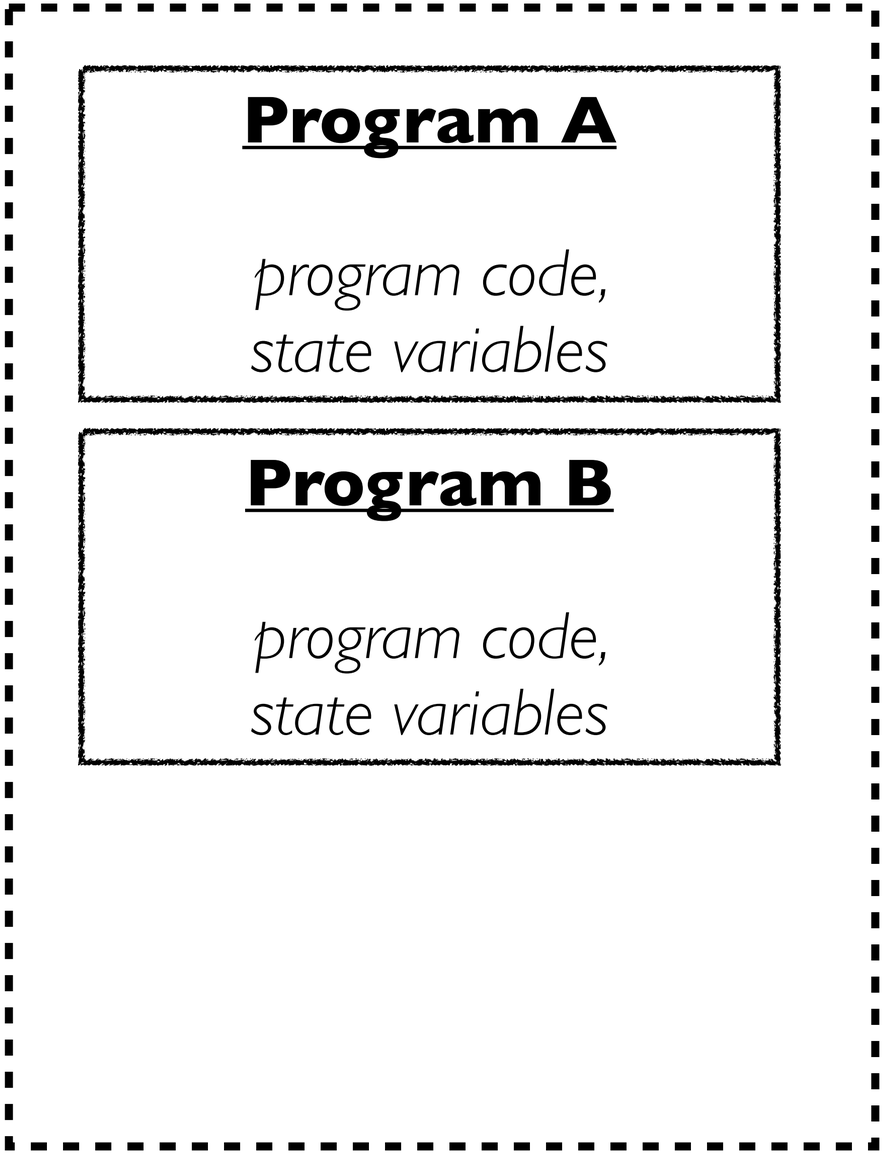
- The network maintains a database
- The database contains one "record" for each program, containing code for its stateUpdate function and all variables/data
- Users can run any program by sending an "execute" transaction (just renaming what we already saw)

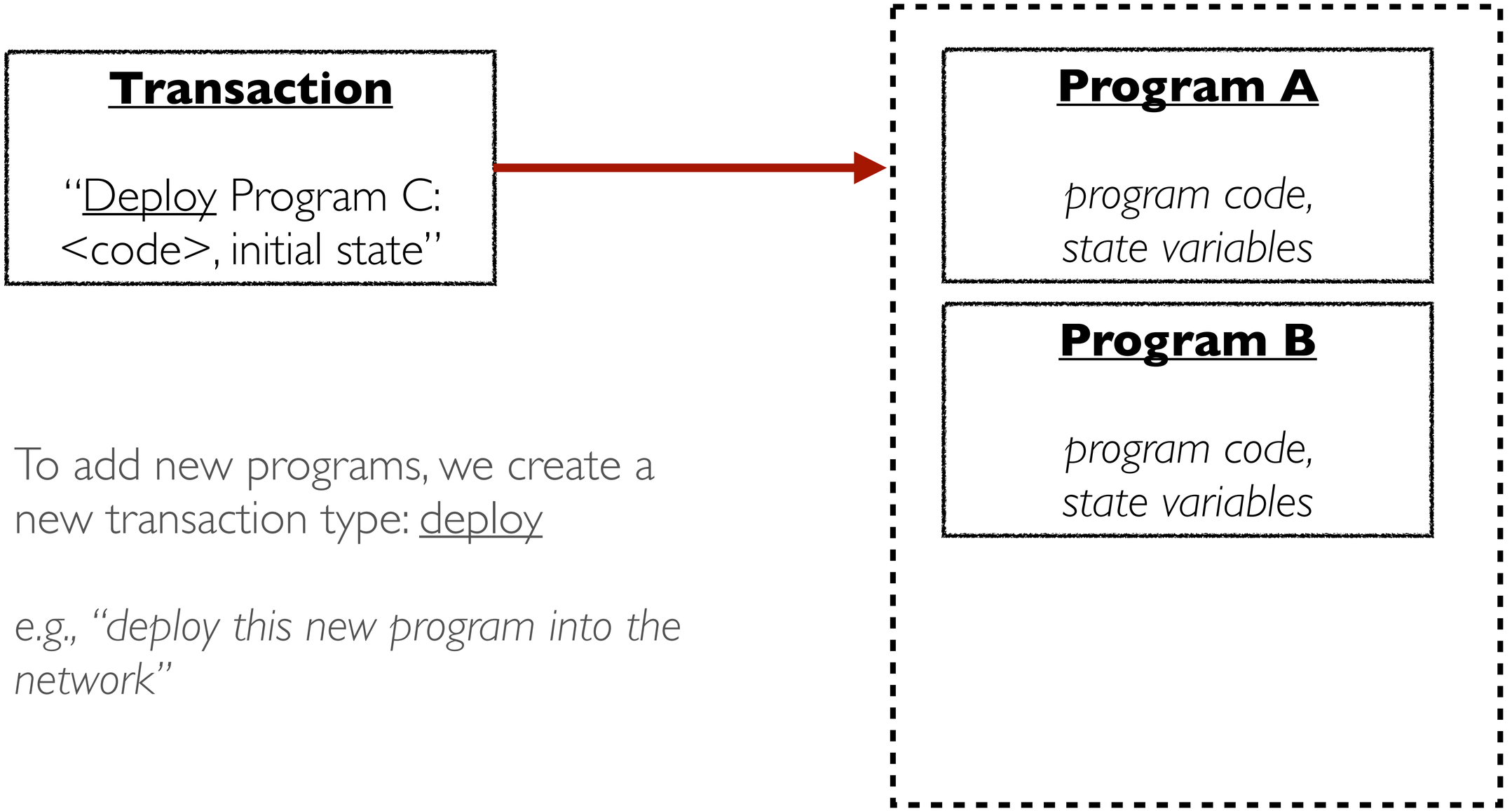
  e.g., "Execute Program B on this input"



- To add new programs, we create a new transaction type: deploy

  e.g., "deploy this new program into the network"

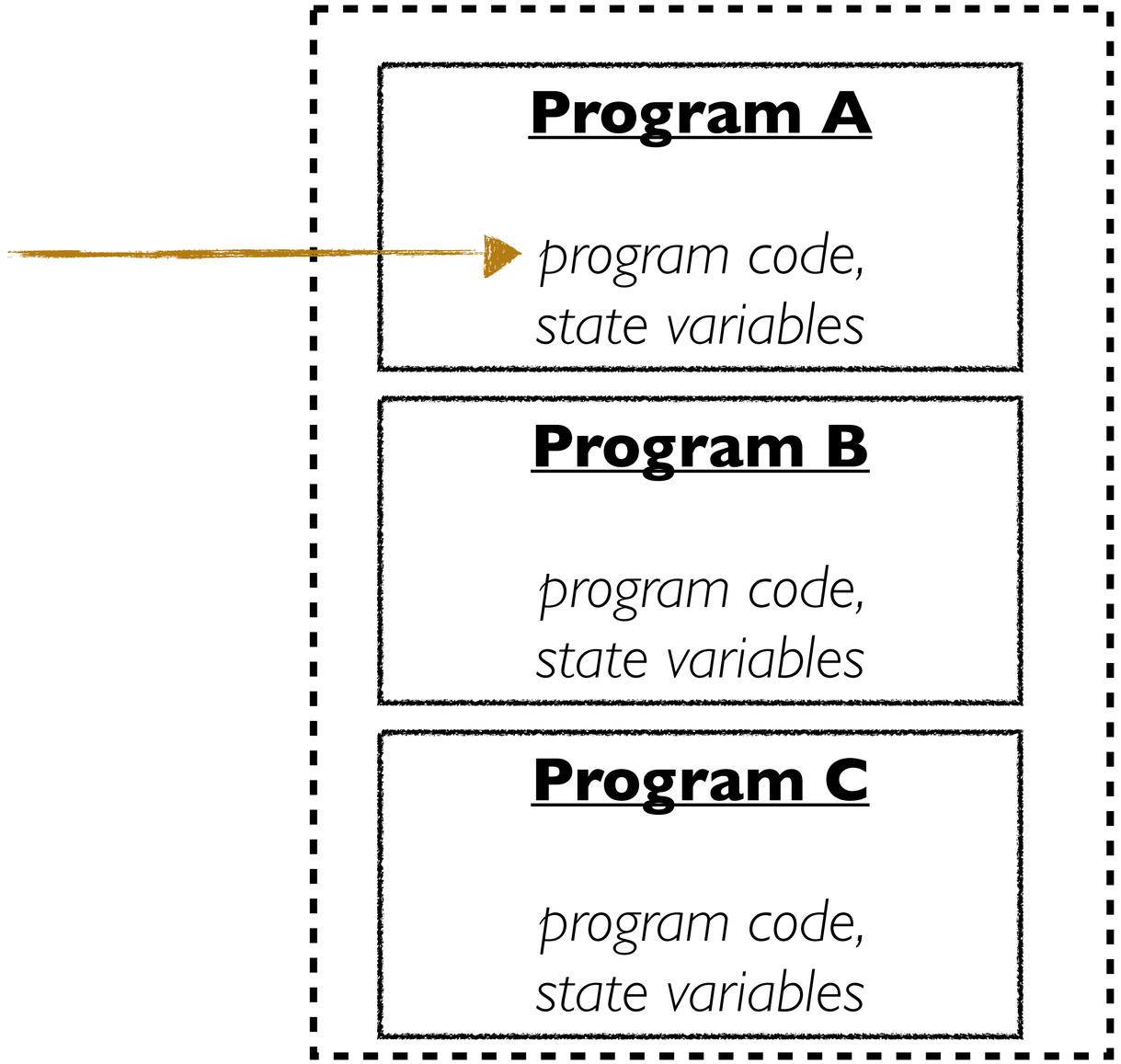




- To add new programs, we create a new transaction type: deploy

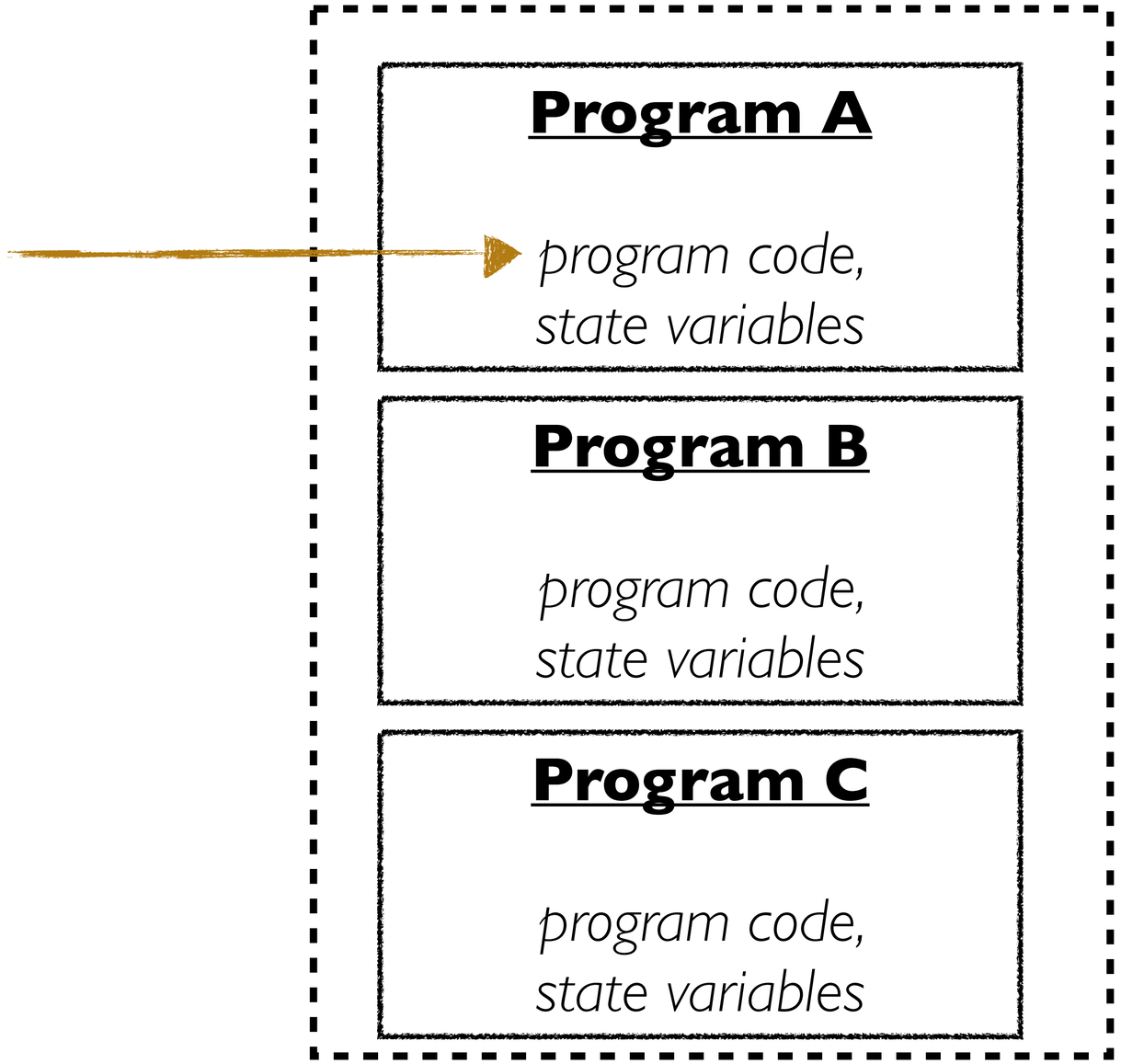
  e.g., "deploy this new program into the network"

Q: What should this code be written in?



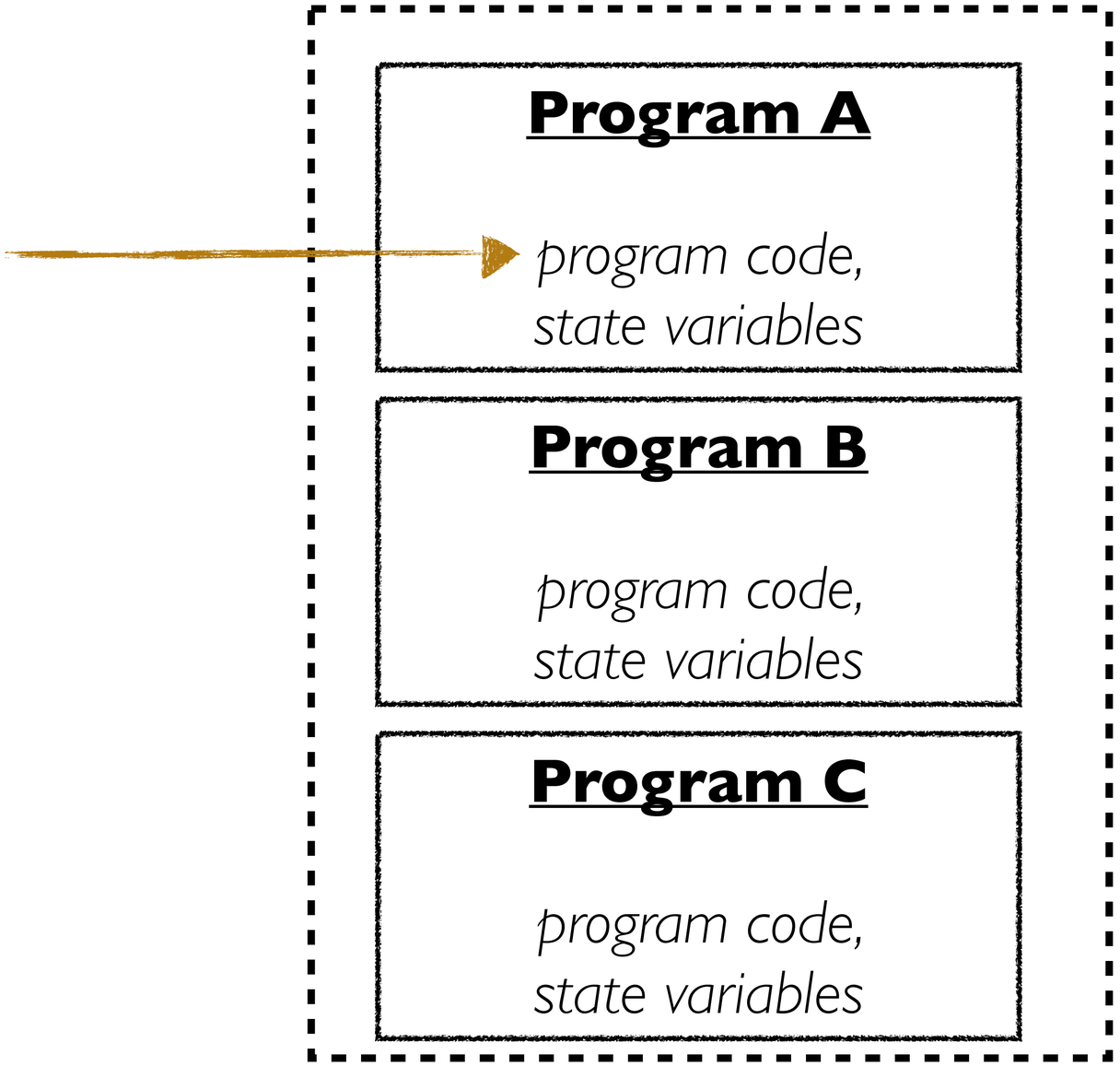
Q: What should this code be written in?

A: Javascript



Q: What should this code be written in?

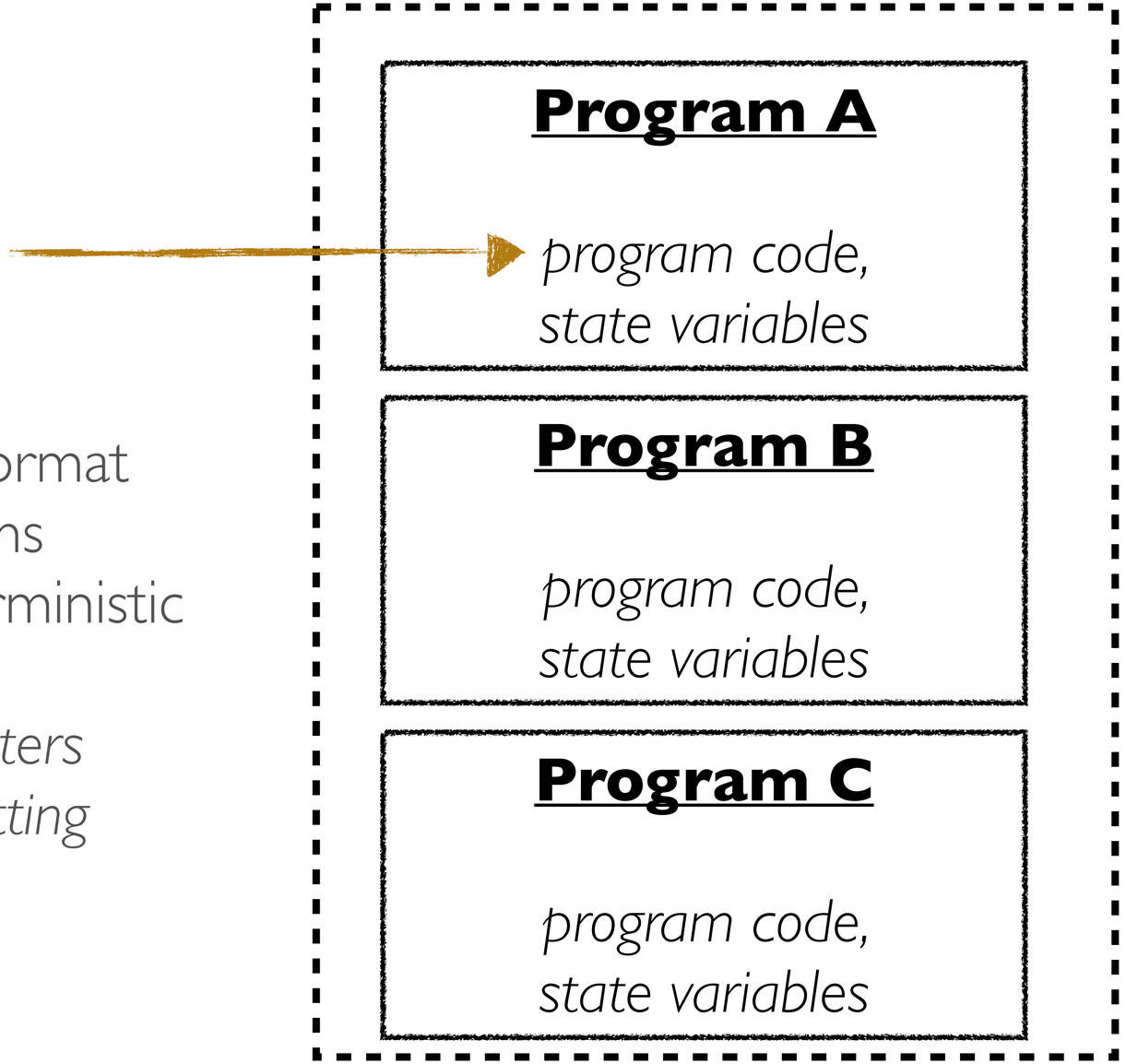
A: Javascript



Q: What should this code be written in?

A: Some kind of portable code format that runs across many platforms and is well-specified and deterministic

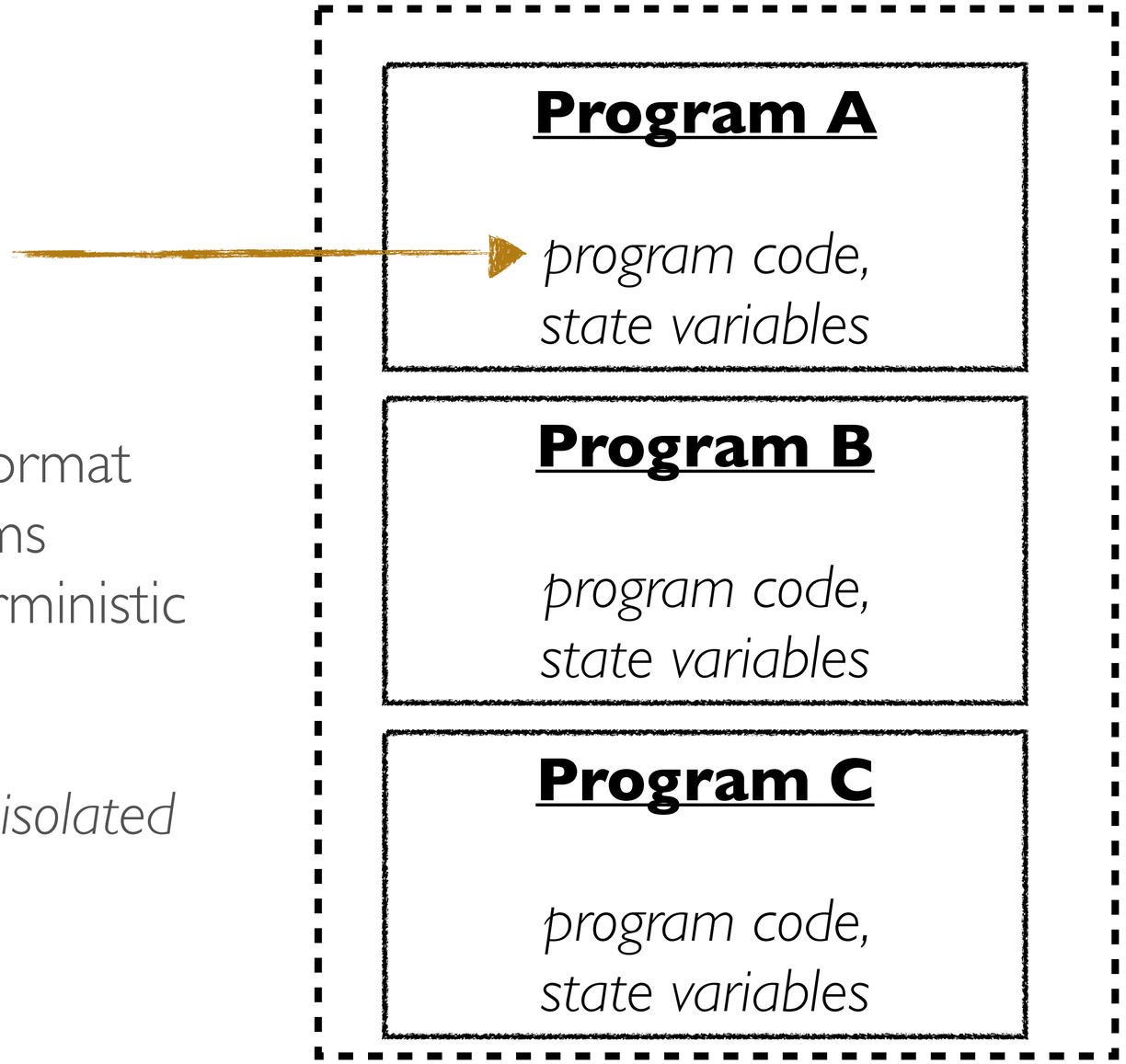
(We can't have different computers running the same code and getting different results!)



Q: What should this code be written in?

A: Some kind of portable code format that runs across many platforms and is well-specified and deterministic

(Also should be memory-safe and easily contained within an isolated virtual machine or sandbox.)



What if running a program takes too long?

What if a program "fills up" the database with junk?