Blockchains & Cryptocurrencies

Smart Contracts & Ethereum II

Instructor: Matthew Green
Fall 2024

News?

Trump involved in Bitcoin transaction in New York's PubKey bar

Former President Trump assisted in completing a Lightning transaction to buy burgers for fans.

Liam 'Akiba' Wright
Sep. 19, 2024 at 10:44 am UTC

3 min read
Updated: Sep. 19, 2024 at 10:44 am UTC

Gage Skidmore / CC BY-SA 2.0 / Flickr. Remixed by CryptoSlate

NEWS 4 MIN READ

Donald Trump's Messy Crypto Burger Purchase Shows Web3 Isn't Ready for Primetime

PUBLISHED 4 DAYS AGO
BY EDDIE MITCHELL VERIFIED BY INSHA ZIA

SHARE ON

MOST POPULAR

CRYPTO 3 DAYS AGO

Catizen (CATI) Token

Donald Trump paid a visit to New York Greenwich PubKey bar the other day **and bought burgers** using the Lightning Network (LN) and Zaprite. It wasn't exactly a great showcase of the technology, and is more likely to be **mocked** than to motivate new converts.

What will it take to get greater Lightning adoption? Viktor Ihnatiuk runs a Bitcoin-focused venture studio called Boosty Labs that's tackling this question on a couple fronts.

"Lightning Network has a chicken-and-egg problem — they don't have enough transactions because they don't have any public liquidity and vice versa," Ihnatiuk told Blockworks.

The Kraken crypto exchange recently removed support for Lightning deposits and withdrawals, which a spokesperson **told** Protos was "the result of technical changes."

News?

ZOLTAN VARDAI

SEP 09, 2024

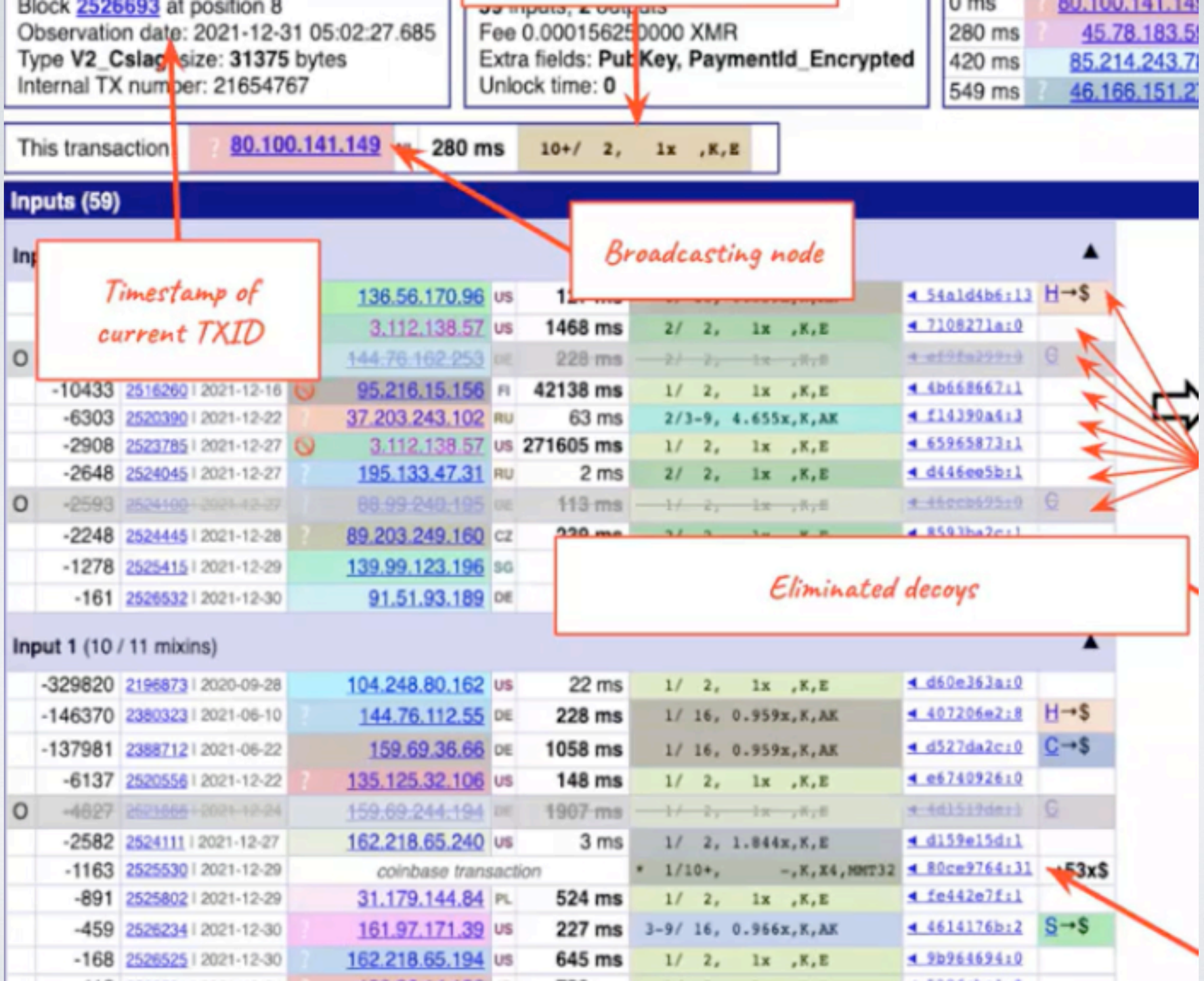
Leaked Chainalysis video suggests Monero transactions may be traceable

A copy of the now-deleted Monero tracing video was shared with Cointelegraph, and it suggests the firm can trace XMR transactions and associated IP addresses.

Can Chainalysis track Monero IPs via its own “malicious nodes?”

Images of the leaked video reemerged on the social media platform Reddit, posted by pseudonymous user u/__lt__.

The user claims that the leaked video shows how Chainalysis can track transactions back to 2021 via its own “malicious” Monero nodes. The user wrote that the company had likely:


Block 2526693 at position 8
Observation date: 2021-12-31 05:02:27.685
Type V2_Cslag, size: 31375 bytes
Internal TX number: 21654767
Fee 0.000156250000 XMR
Extra fields: PubKey, PaymentId_Encrypted
Unlock time: 0
0 ms
280 ms
420 ms
549 ms
This transaction
80.100.141.149
280 ms
Inputs (59)
Timestamp of current TXID
Broadcasting node
Eliminated decoys
Input 1 (10 / 11 mixins)
coinbase transaction

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.

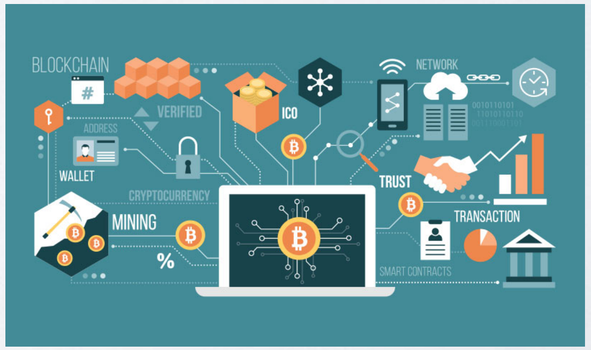
Obsolete Historical Data

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

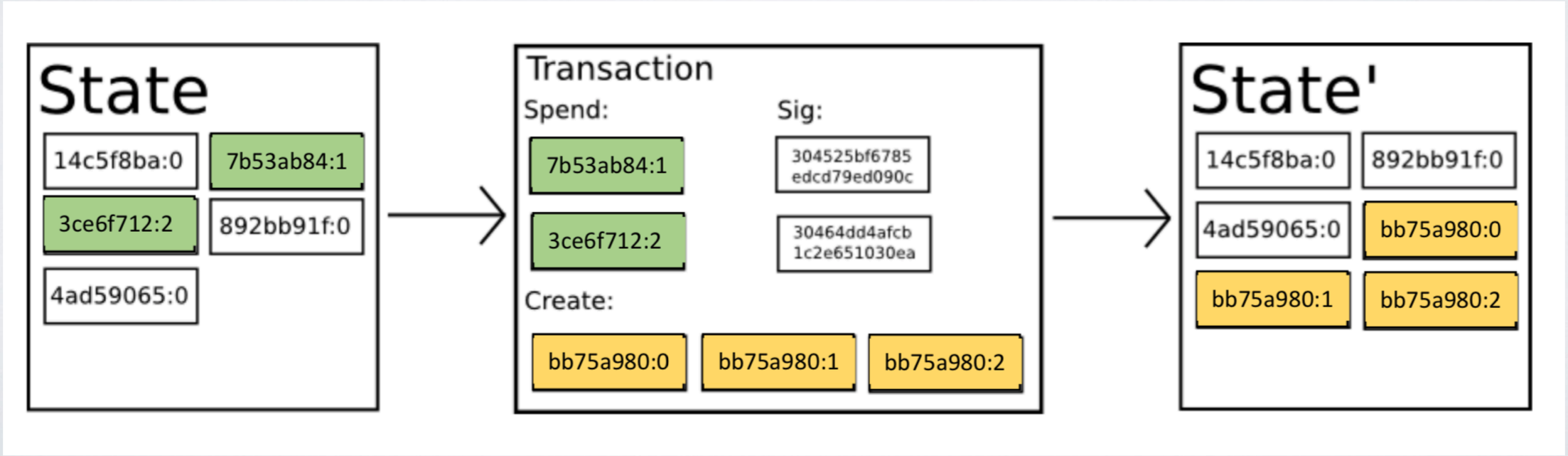
- From Bitcoin to smart contracts
- Ethereum
- Applications of smart contracts



Bitcoin transactions as “state transition”

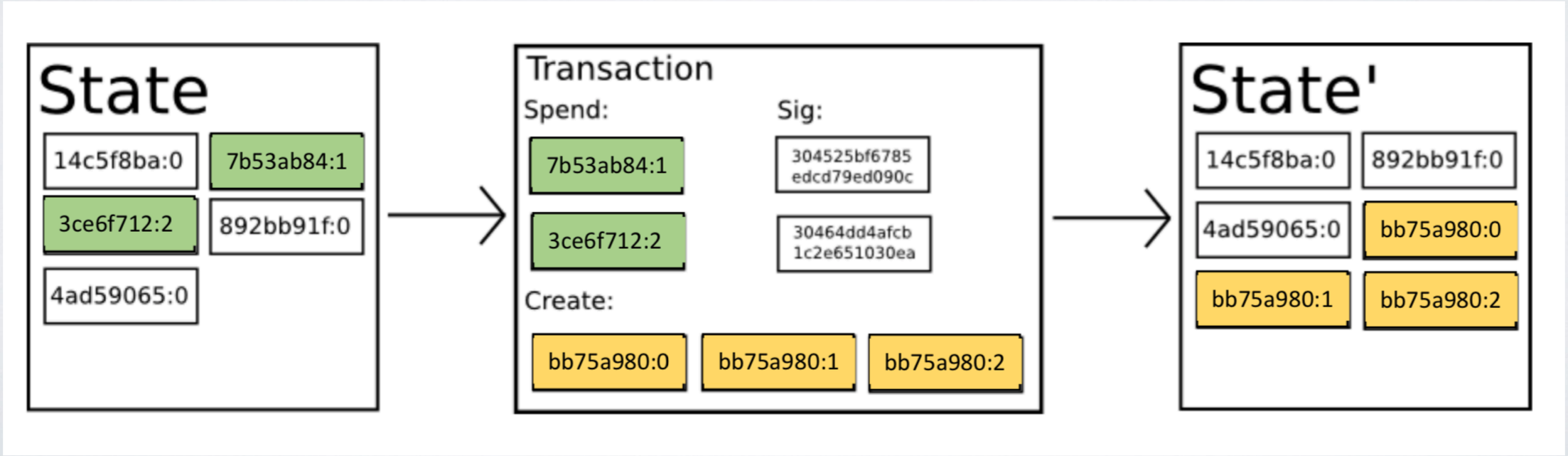
- **Let’s consider each Bitcoin transaction as a state transition function**
  - What is the input state?
  - What is the output state?
  - What does a Transaction do to the state?

- 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)

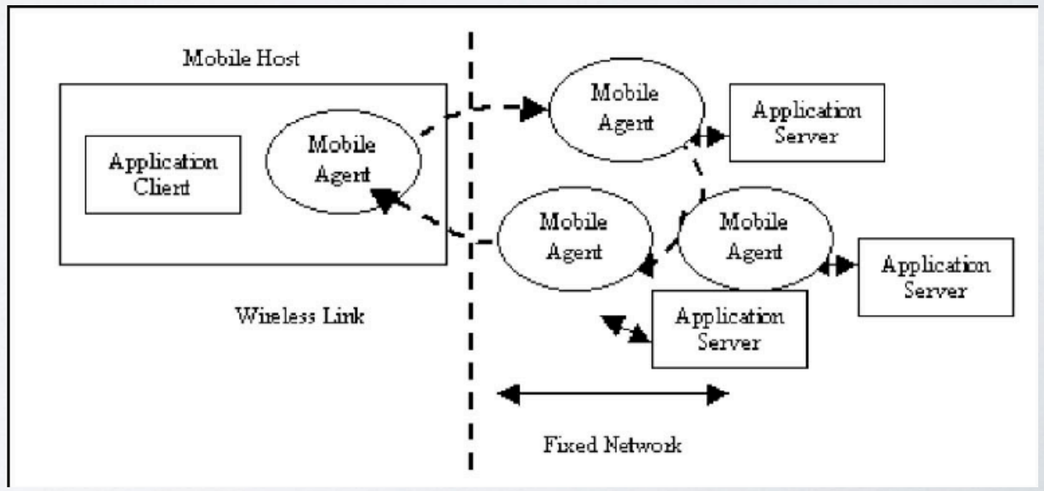


“Smart contracts”

- 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”

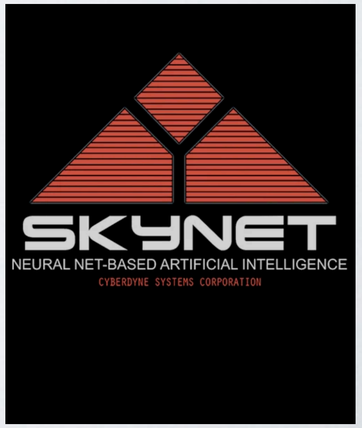
“Mobile Agents”

- 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

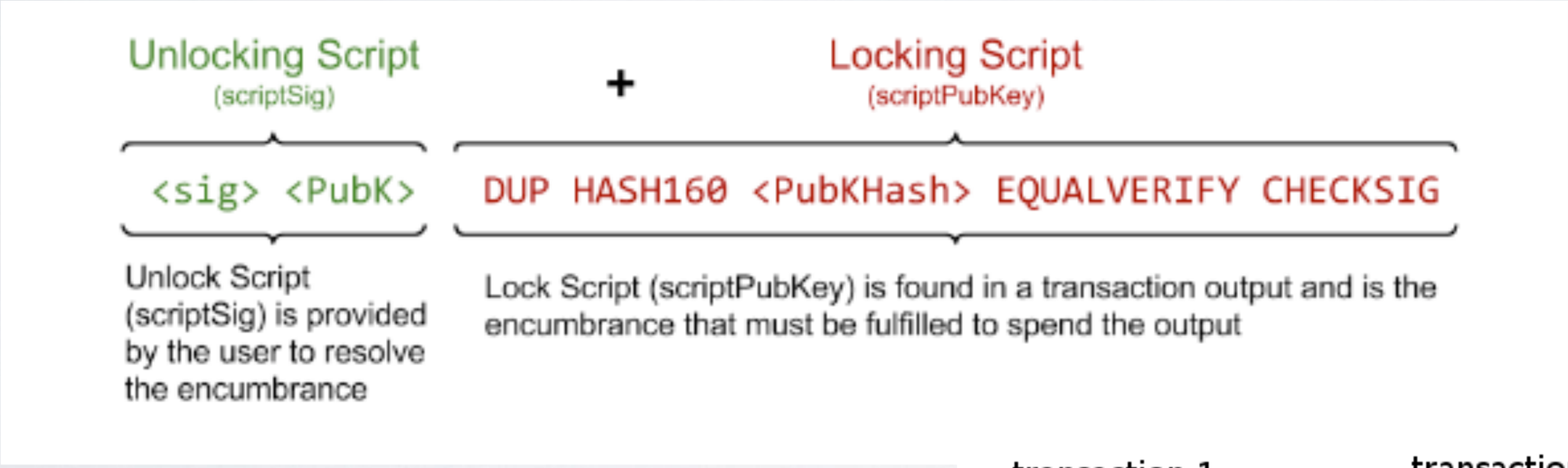


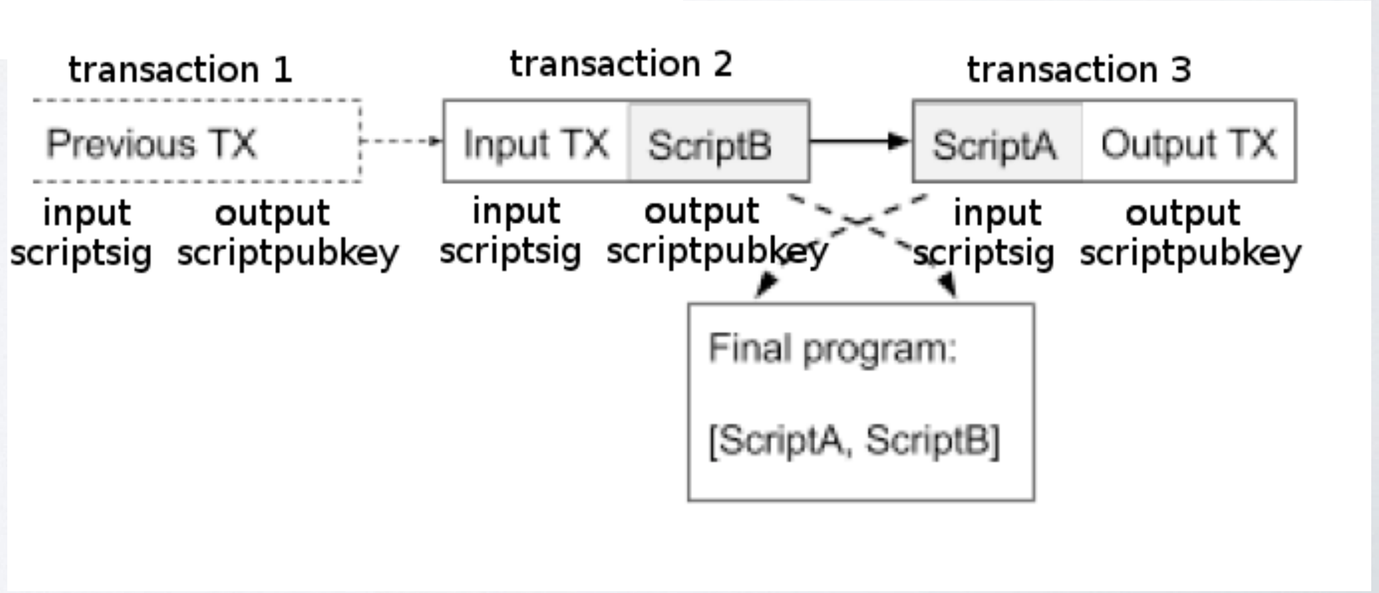
Two visions, same goal

- 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?



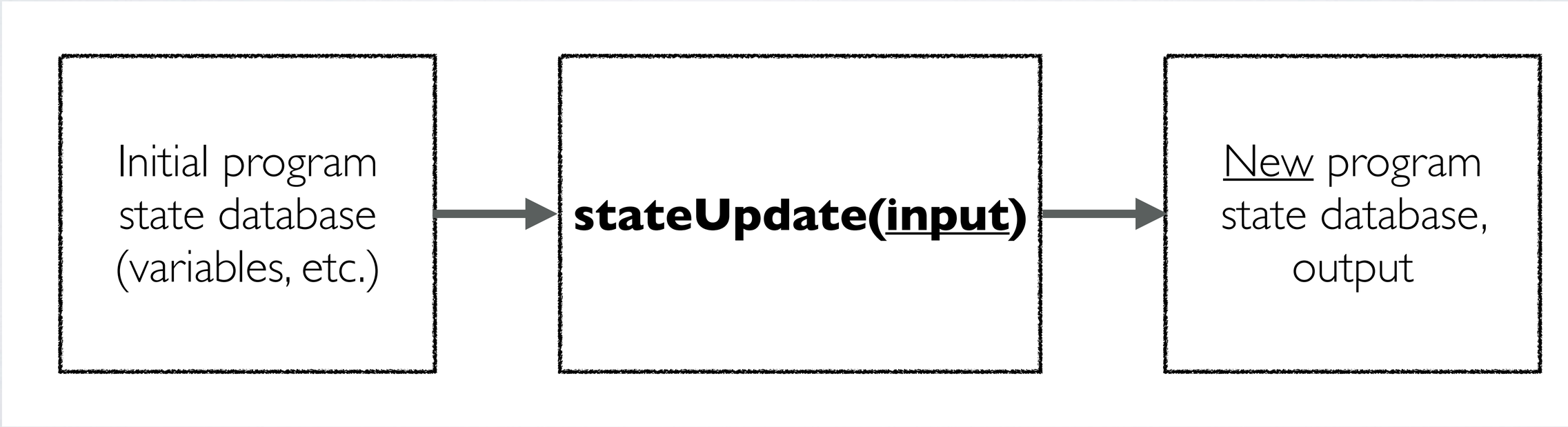


“Towards Ethereum”

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

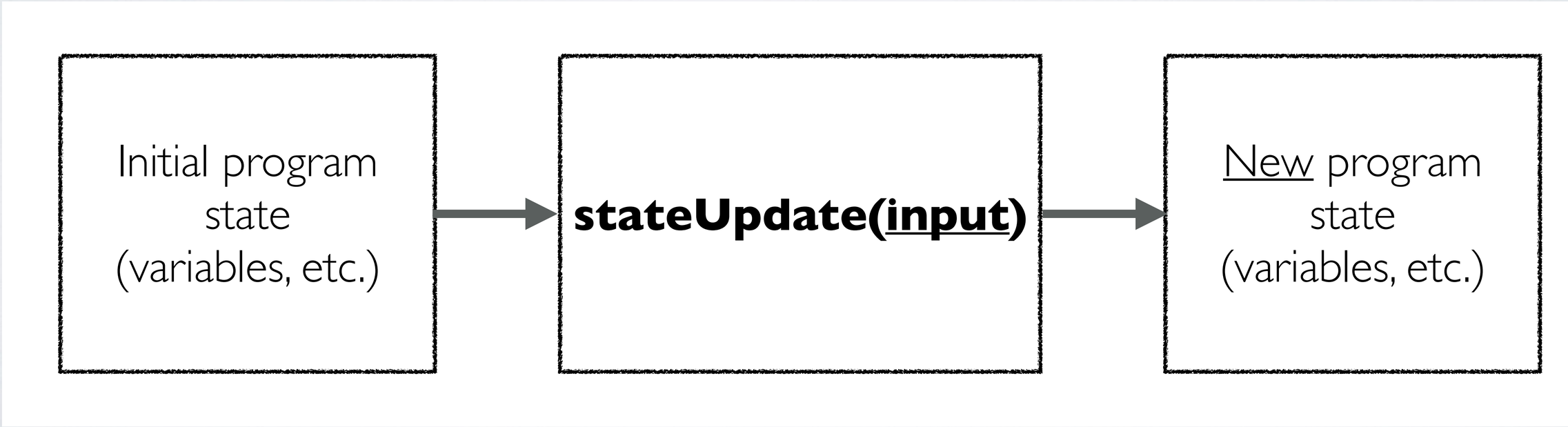
Generalizing programs

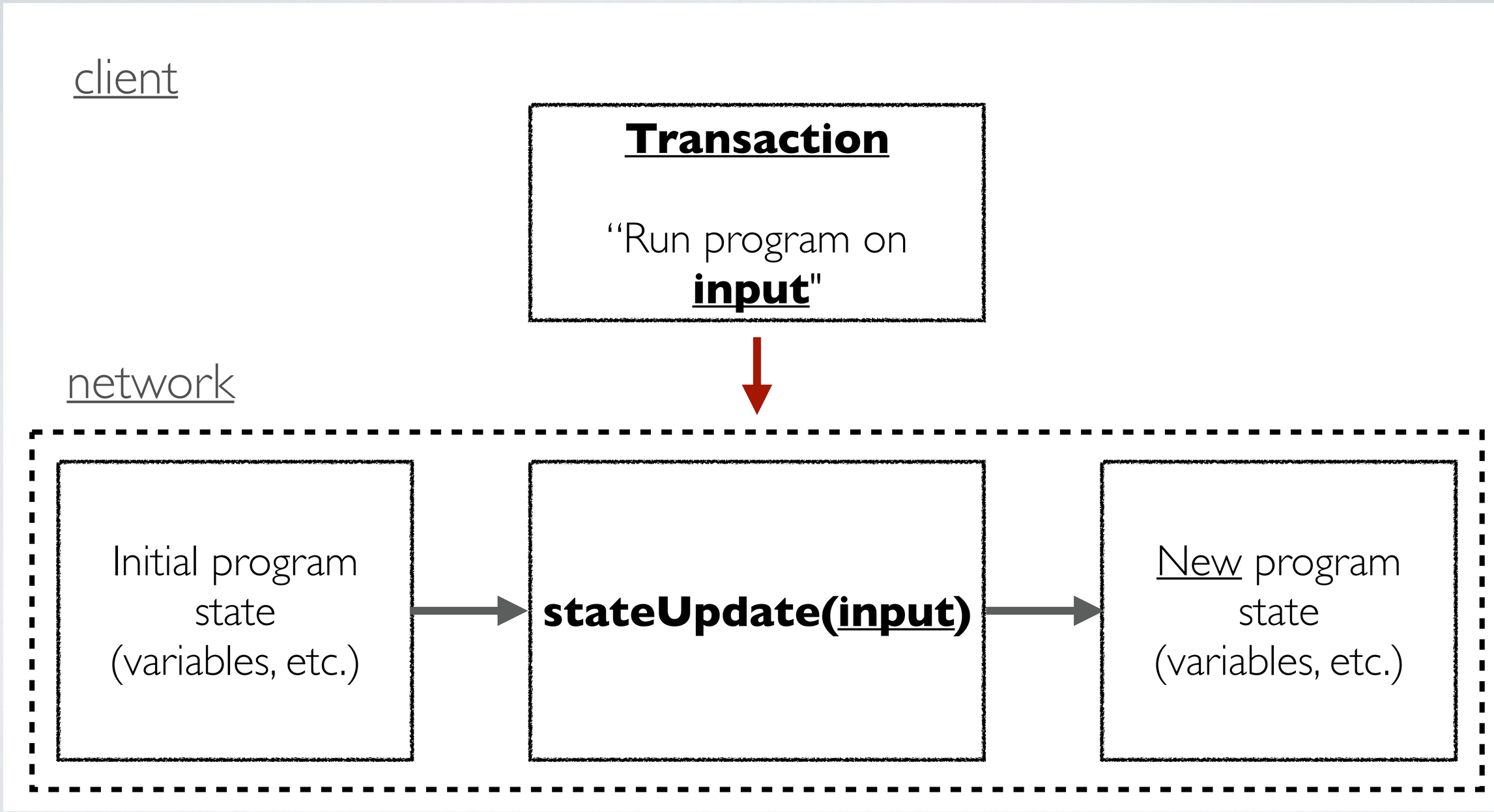
- Programs can be written as state update functions
  - Input: previous state of program, some "call input"
  - Output: new updated state for program, maybe some return value

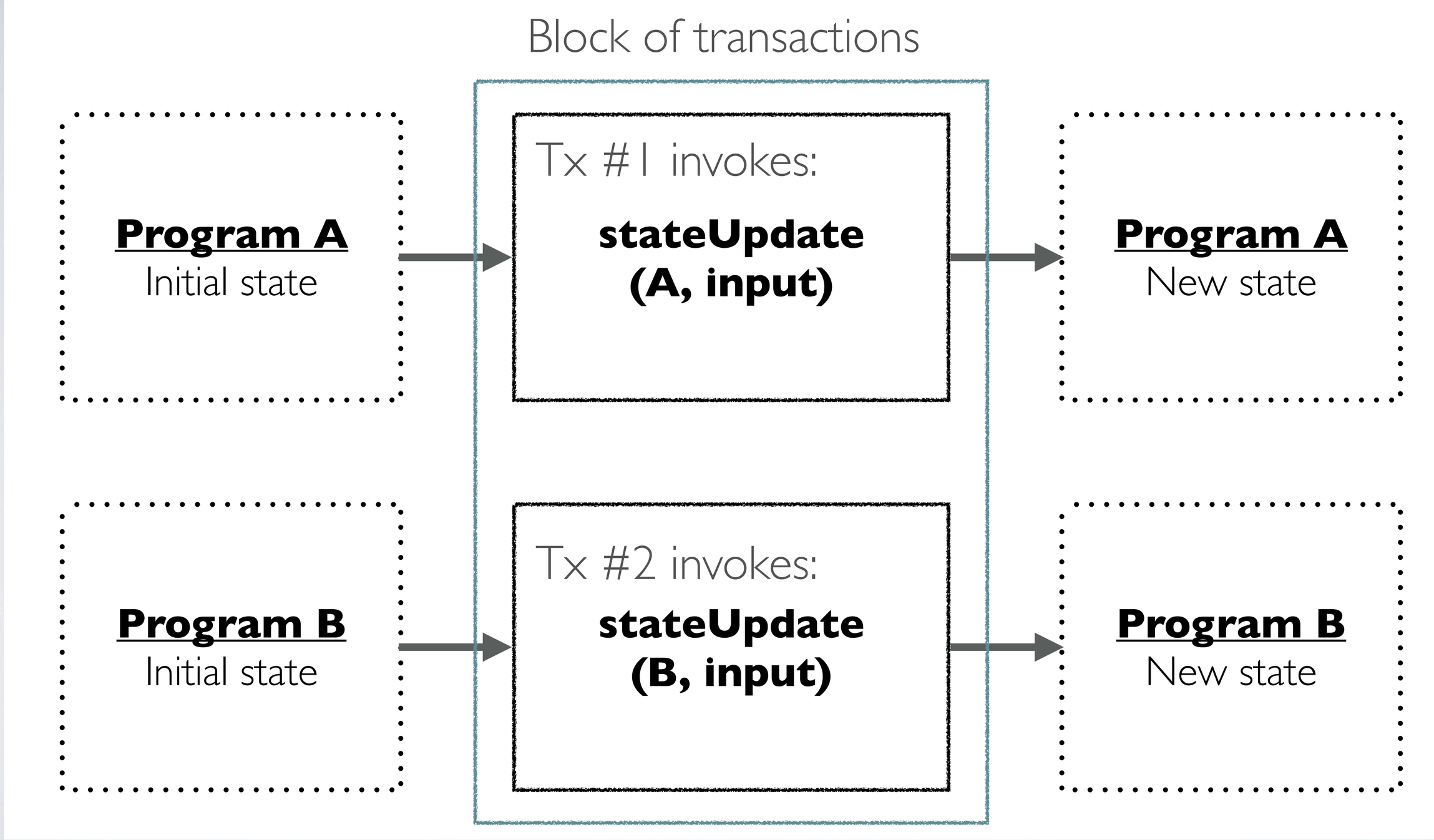


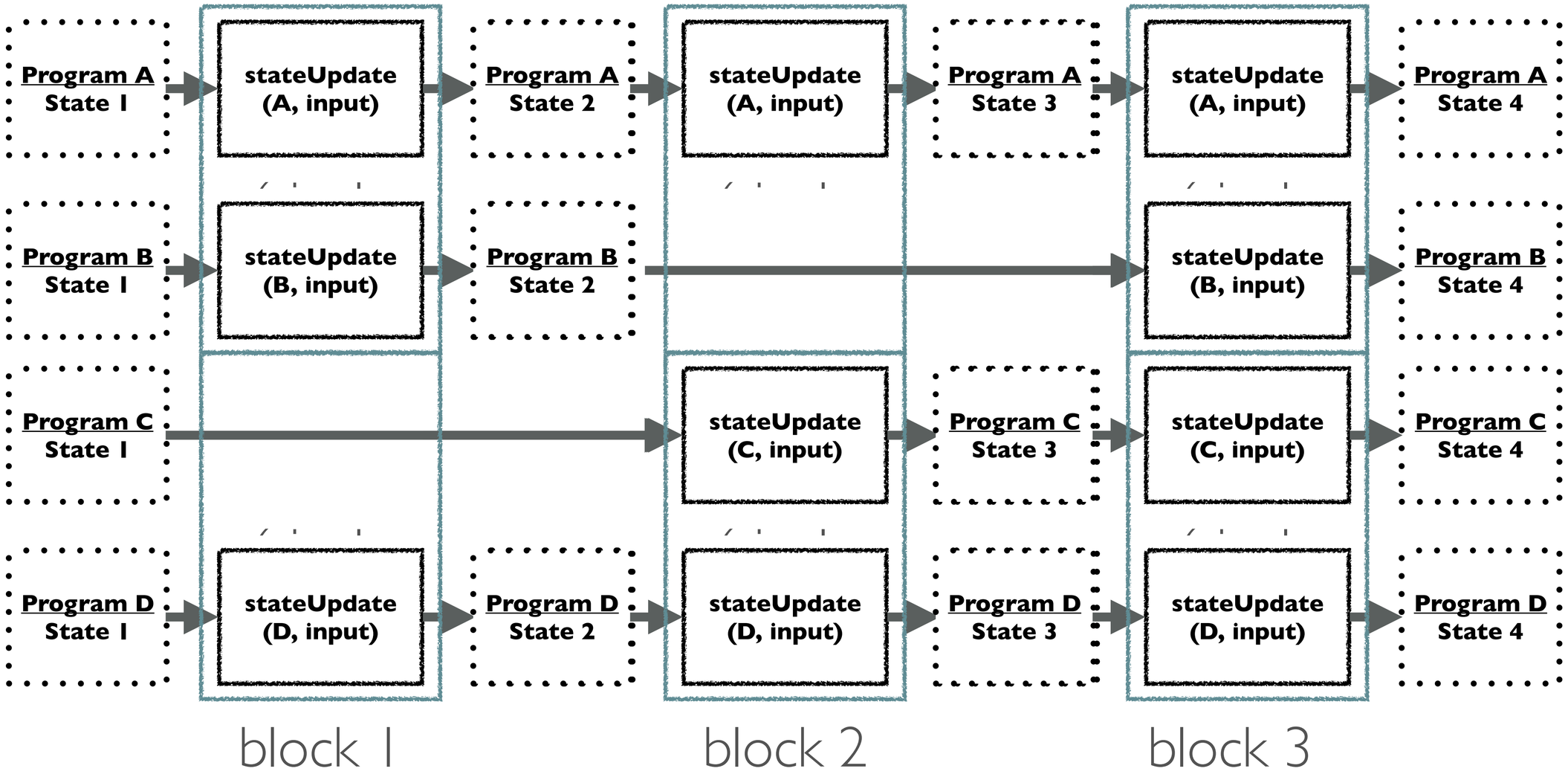
Blockchains for programs

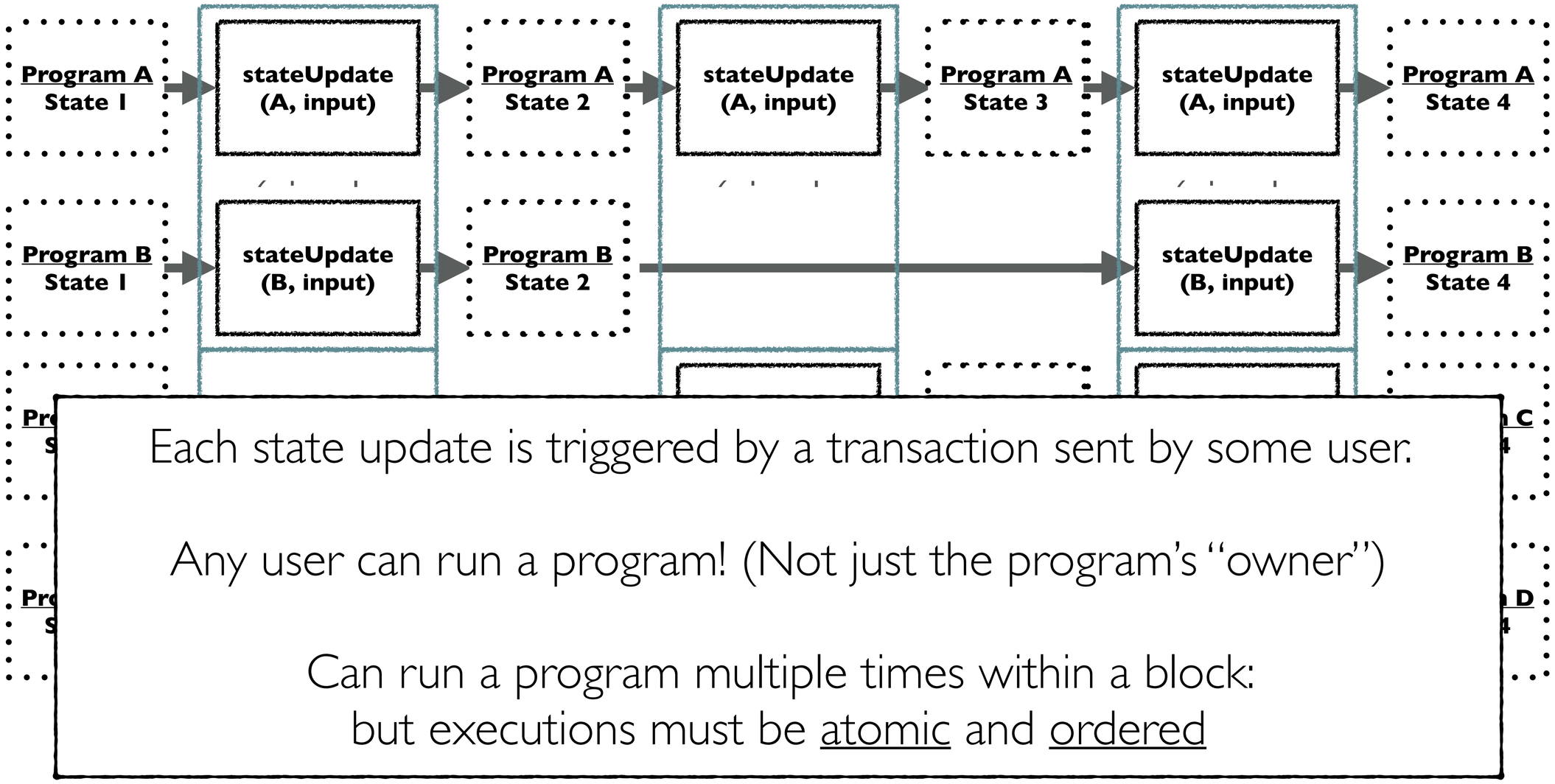
- Assume the blockchain has the program and a database
  - Each transaction simply "invokes" the program on **input**
  - The blockchain runs the program and updates its state



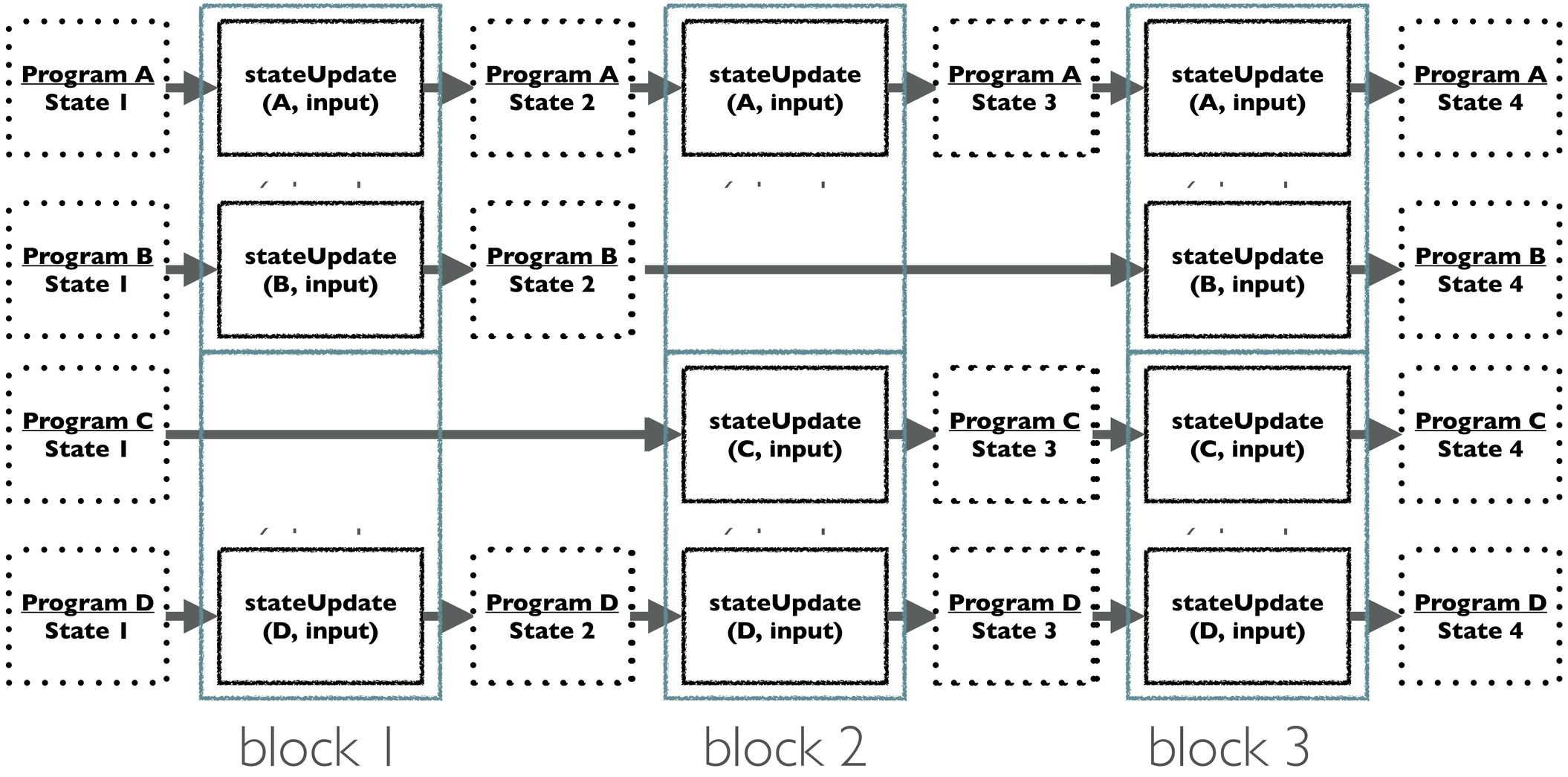

client
Transaction
"Run program on input"
network
Initial program state (variables, etc.)
stateUpdate(input)
New program state (variables, etc.)


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

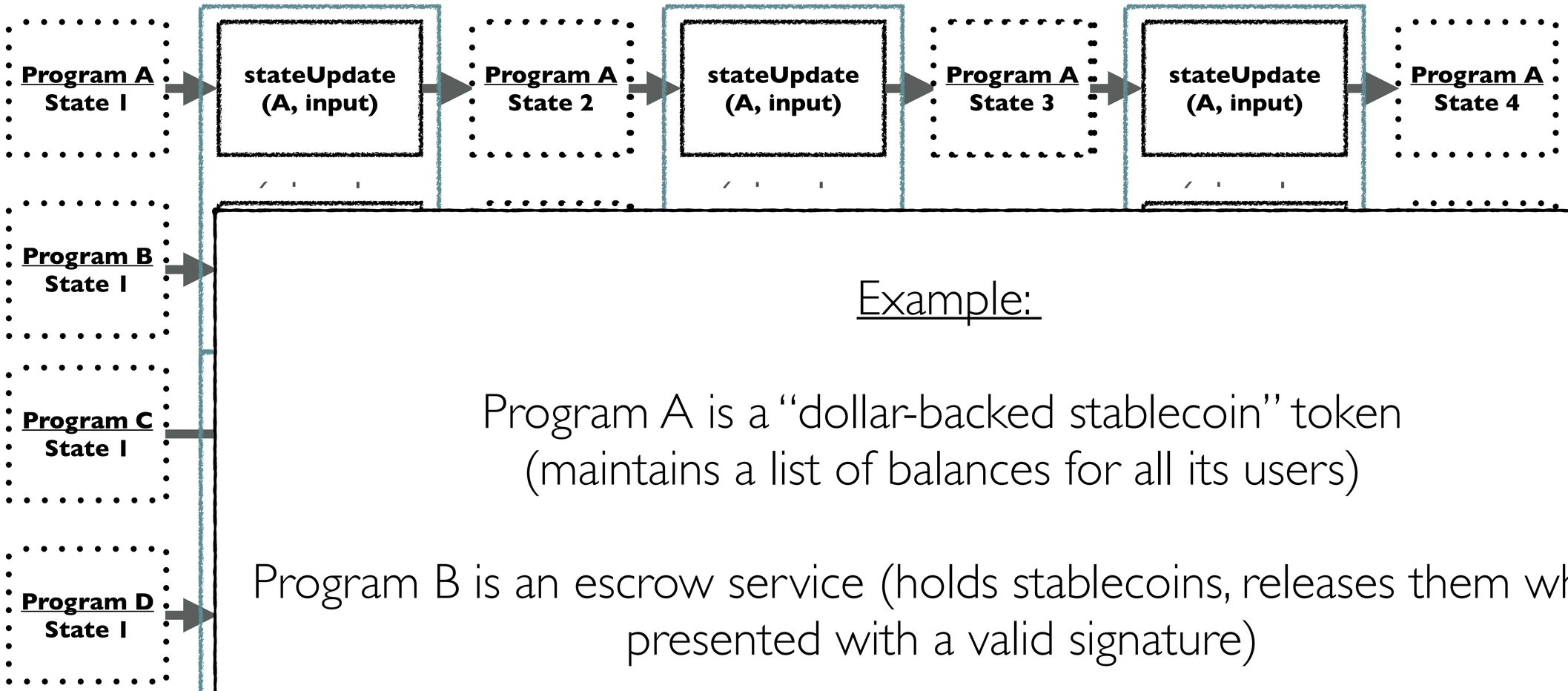

Program A
State 1
stateUpdate
(A, input)
Program A
State 2
stateUpdate
(A, input)
Program A
State 3
stateUpdate
(A, input)
Program A
State 4
Program B
State 1
stateUpdate
(B, input)
Program B
State 2
stateUpdate
(B, input)
Program B
State 4
Program C
State 1
stateUpdate
(C, input)
Program C
State 3
stateUpdate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



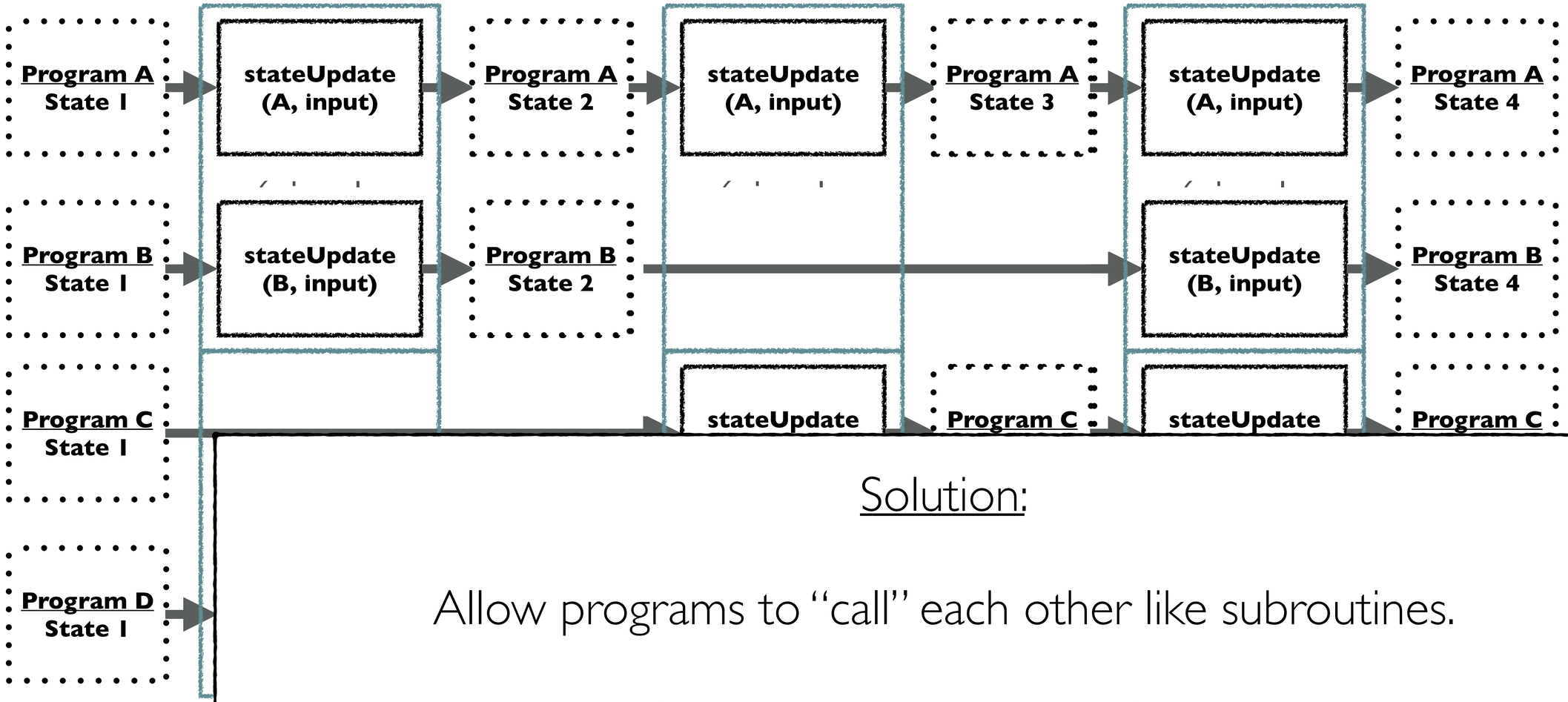
Example:

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?

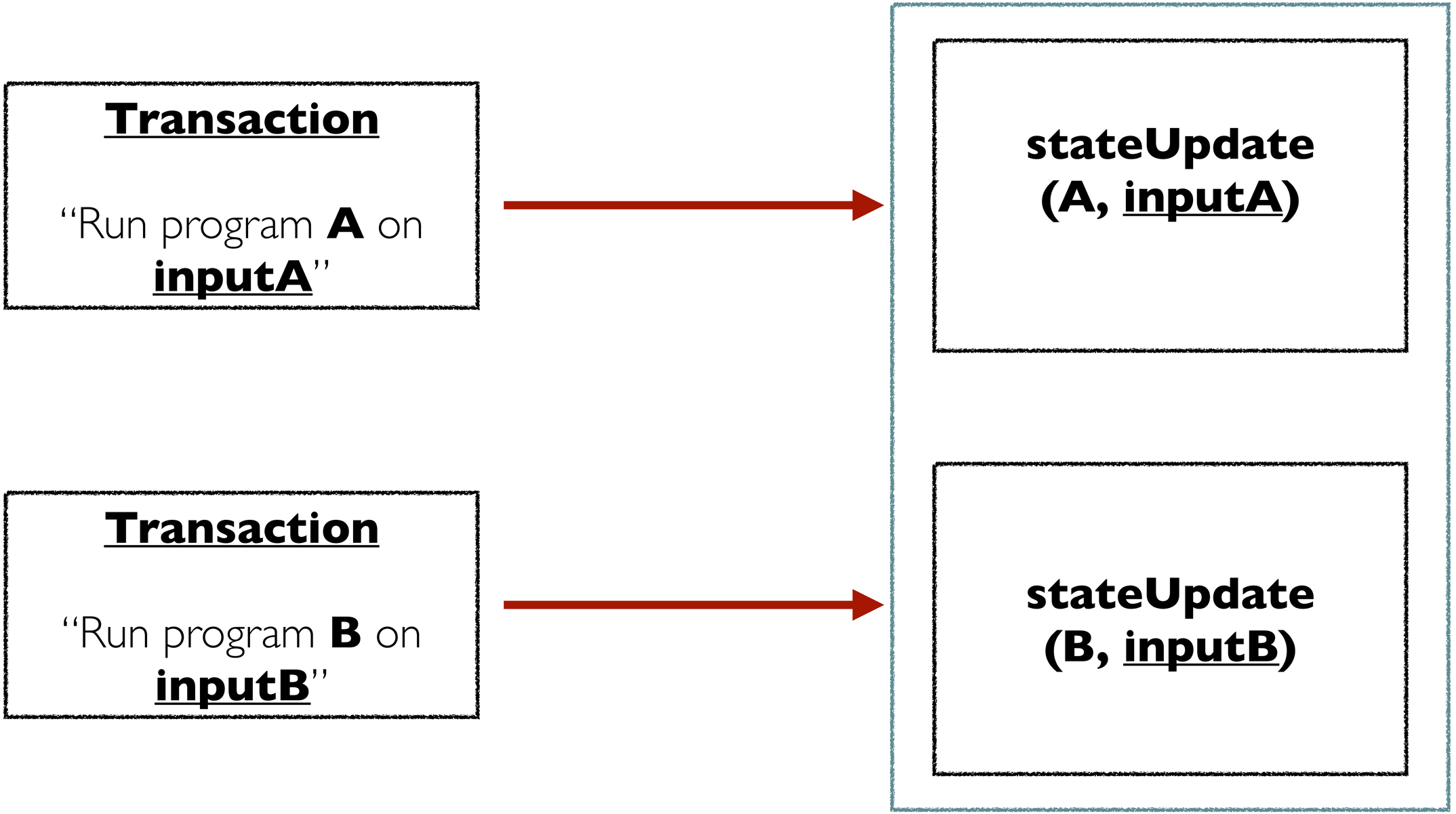


Solution:

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"

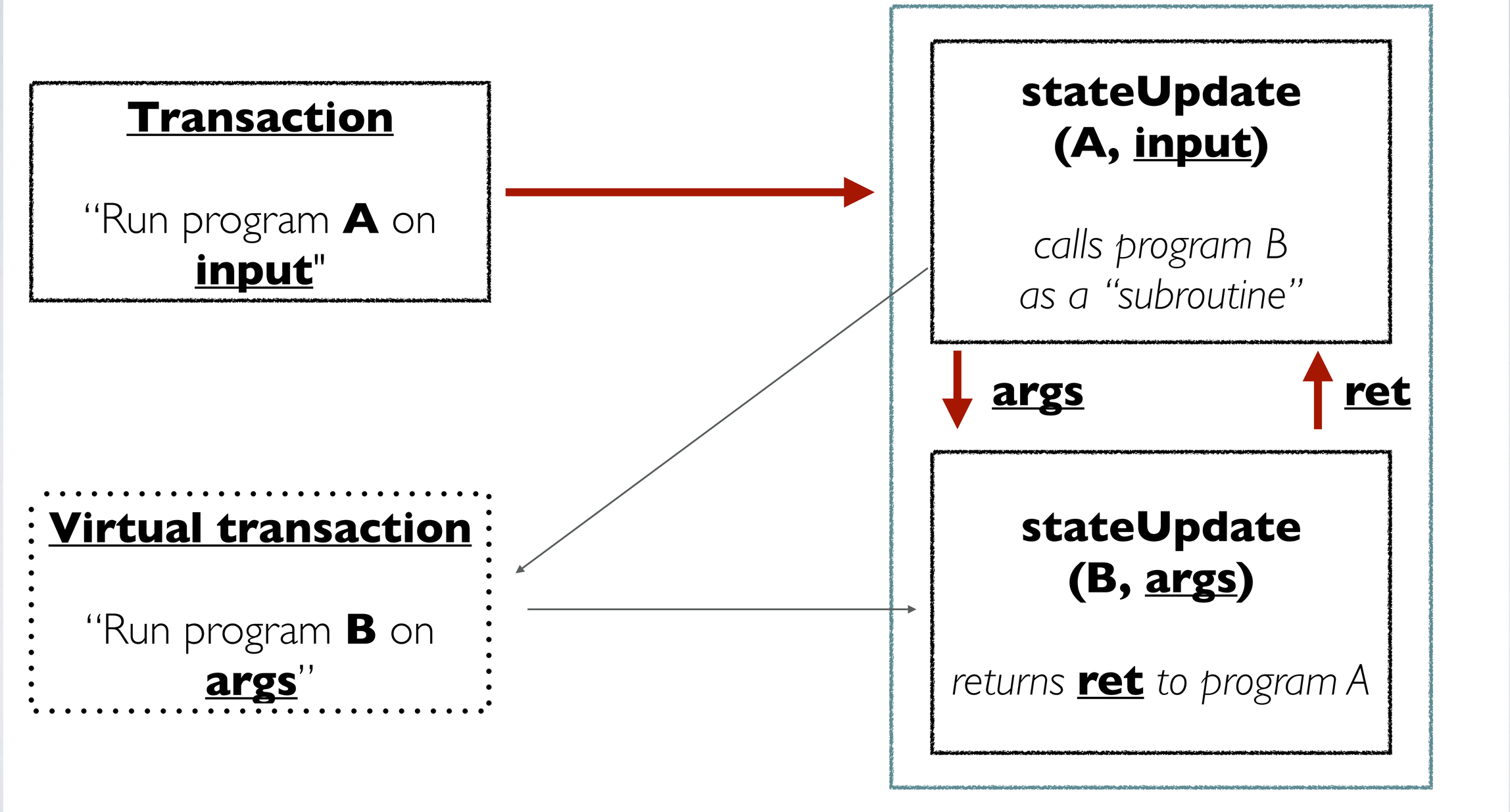
args

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

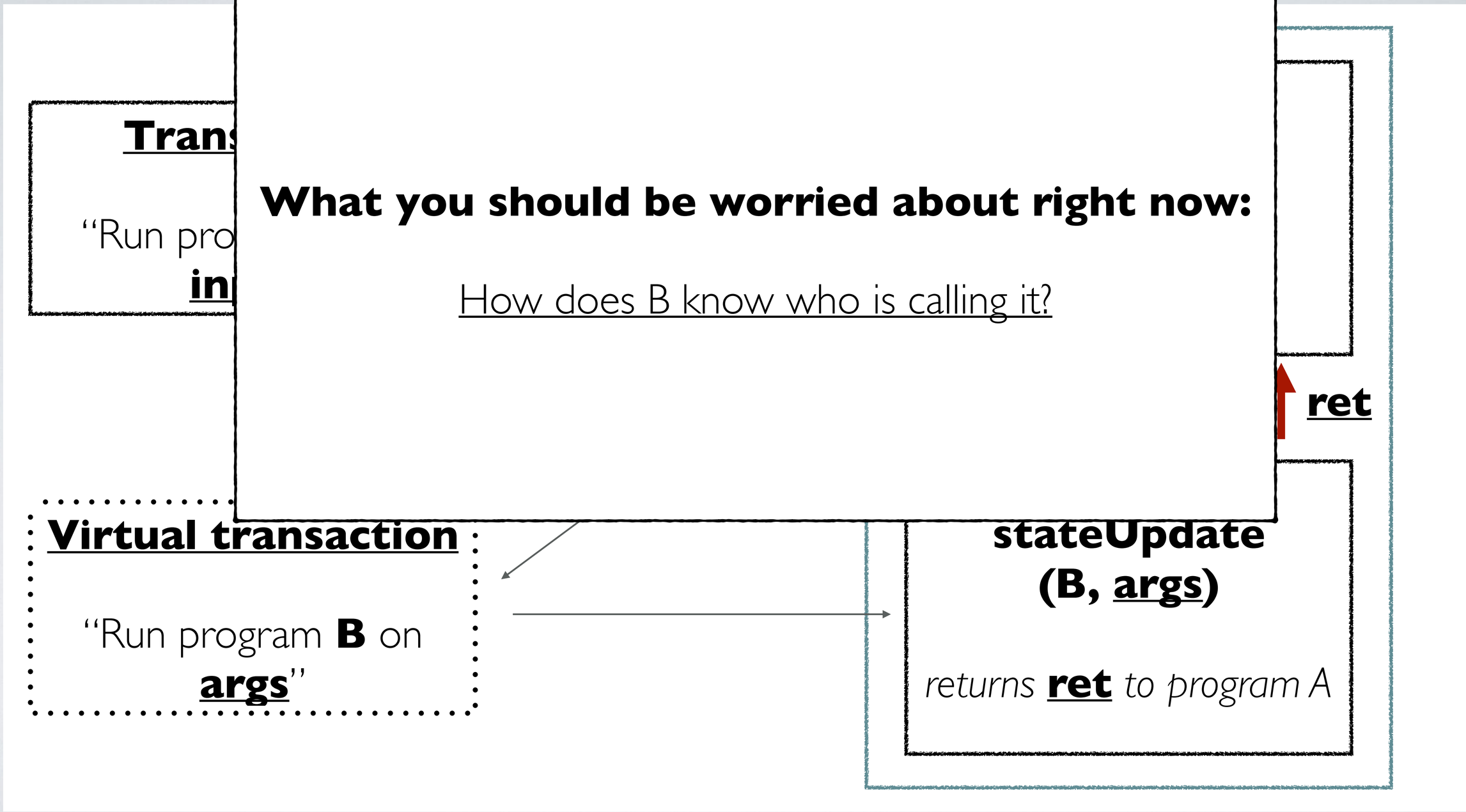
ret

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

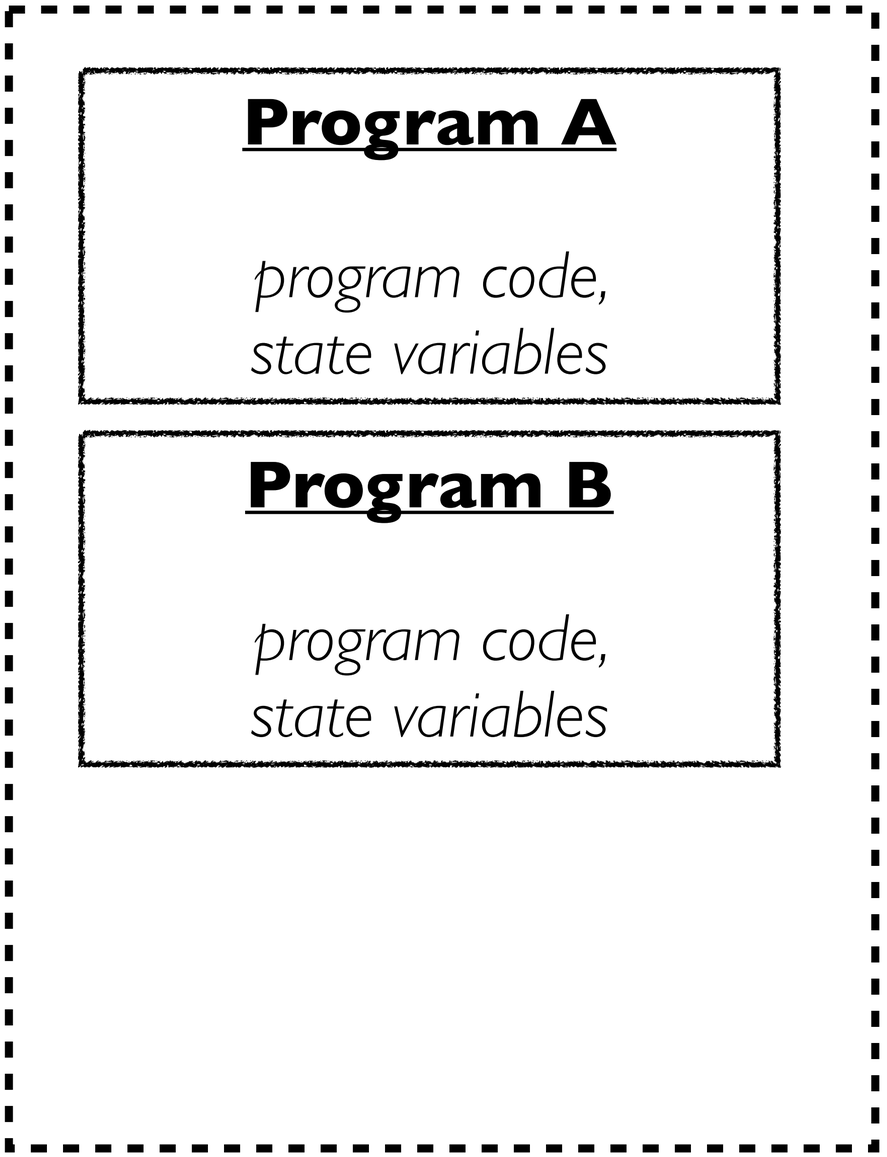
You should have many questions

- 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?

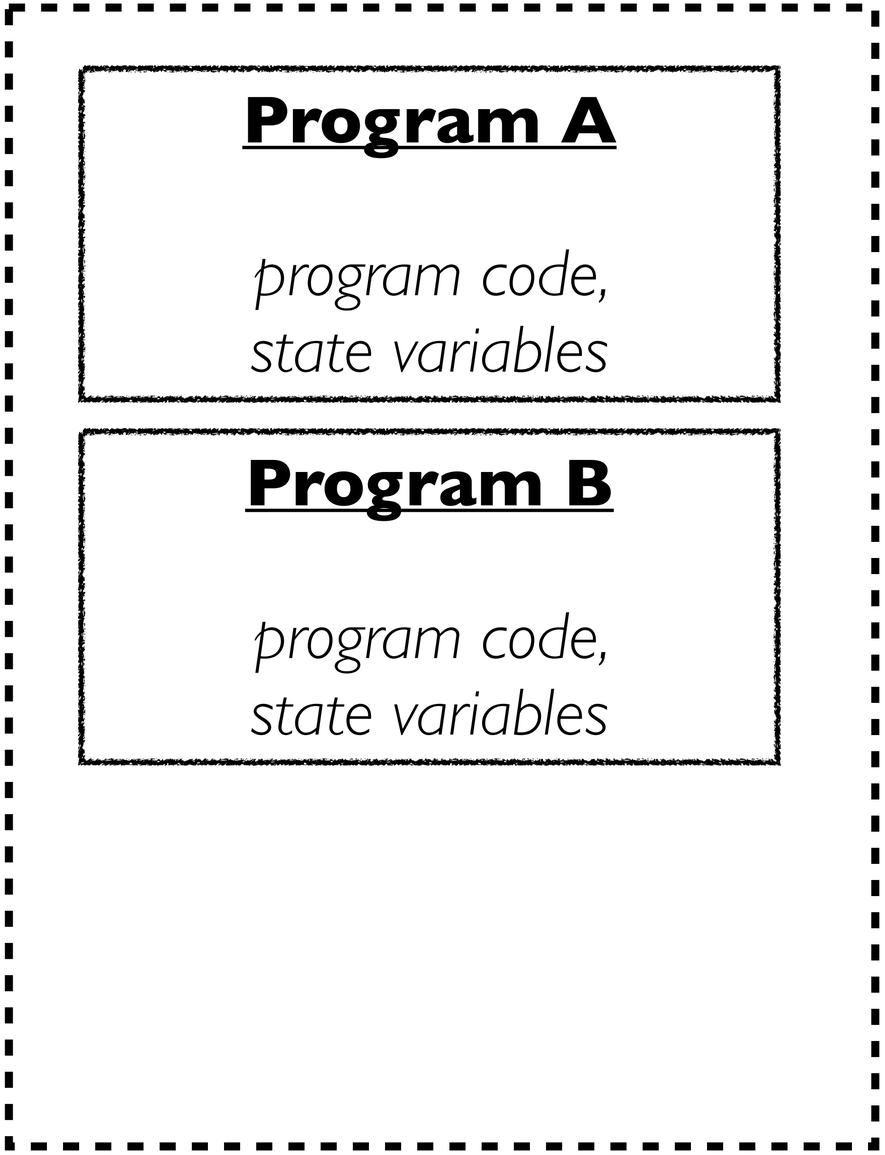
- The network maintains a database
- The database contains one "record" for each program, containing code for its stateUpdate function and all variables/data
- Users can run any program by sending an "execute" transaction (just renaming what we already saw)

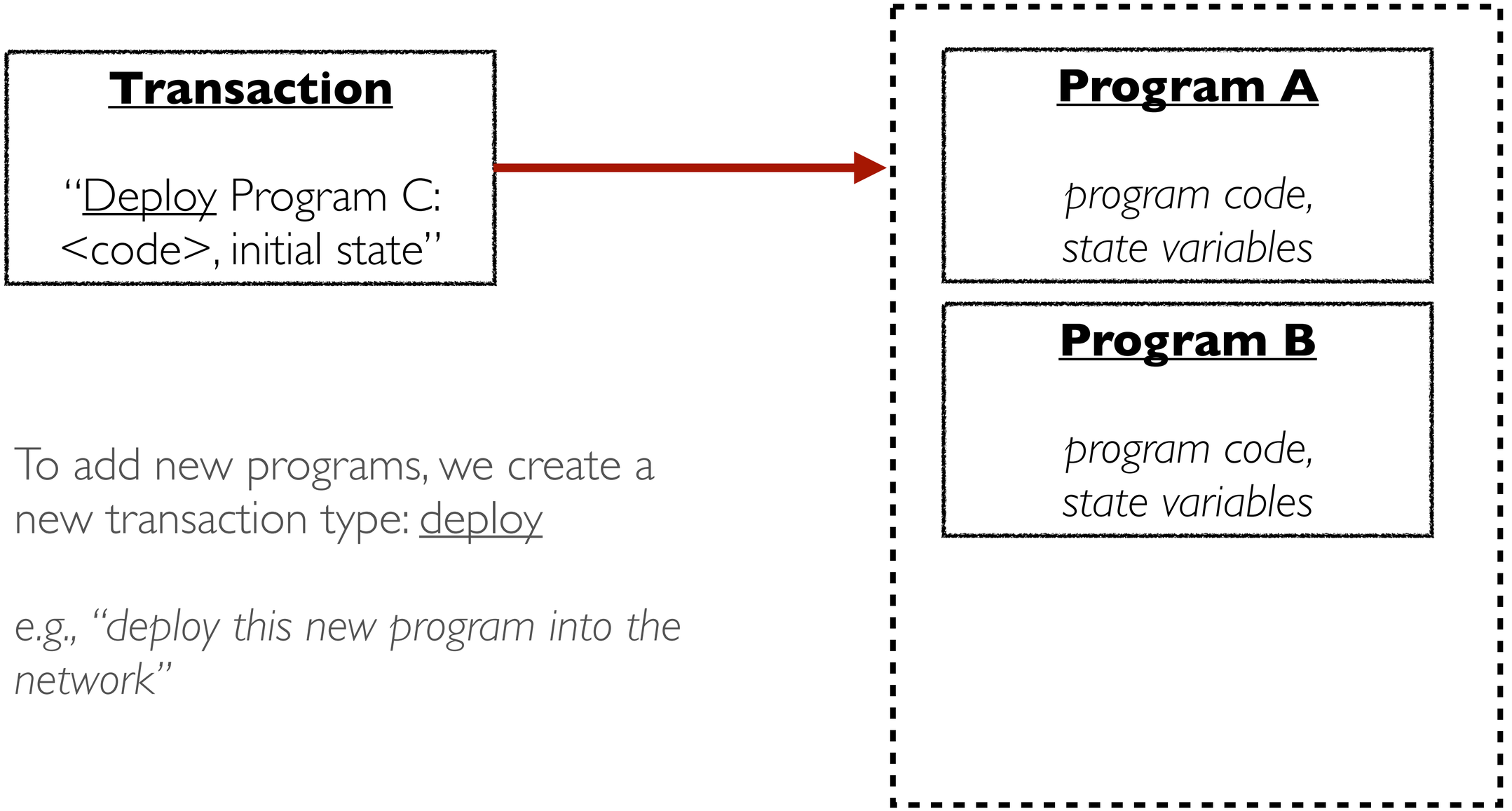
  e.g., "Execute Program B on this input"



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

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

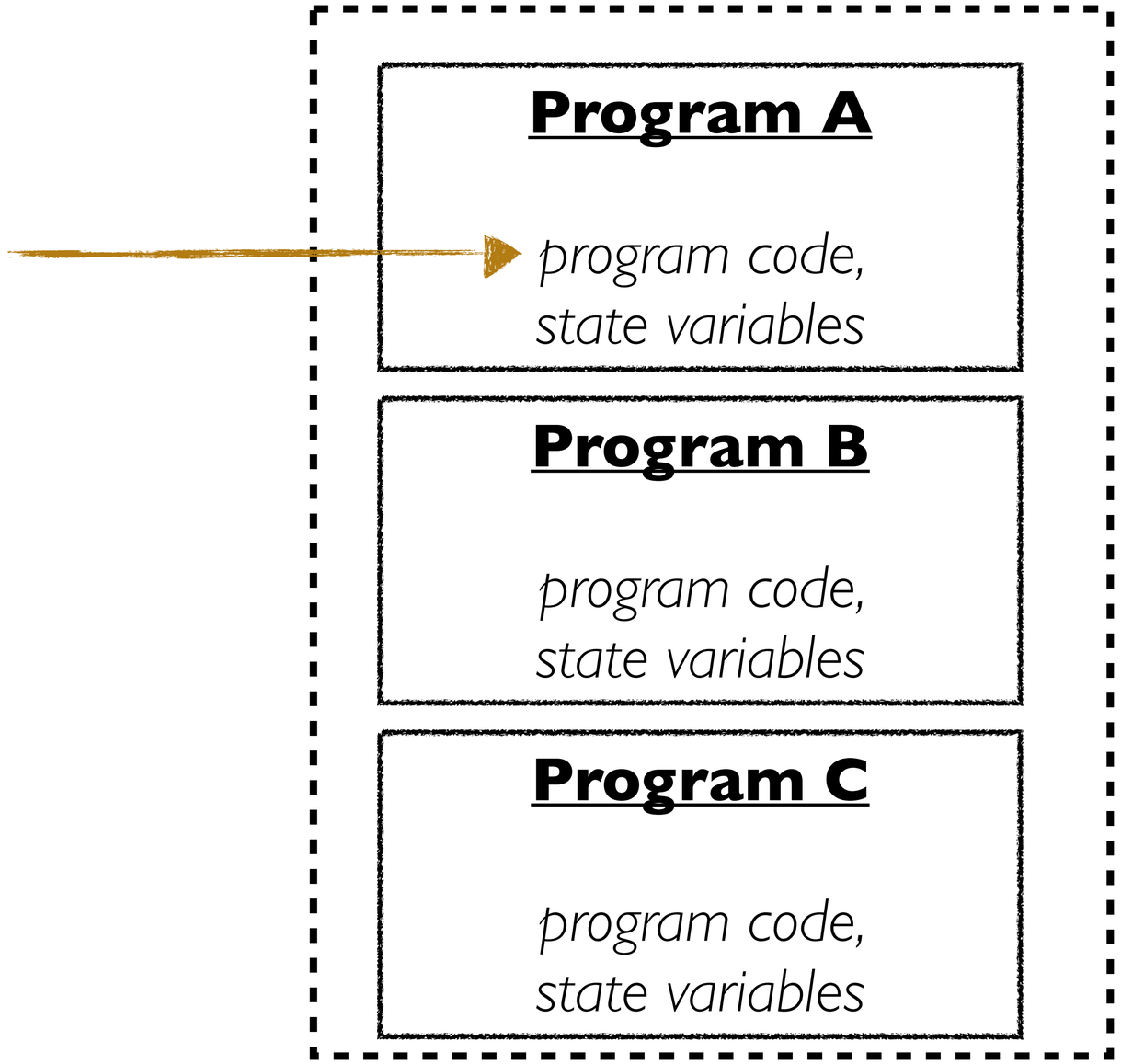




- To add new programs, we create a new transaction type: deploy

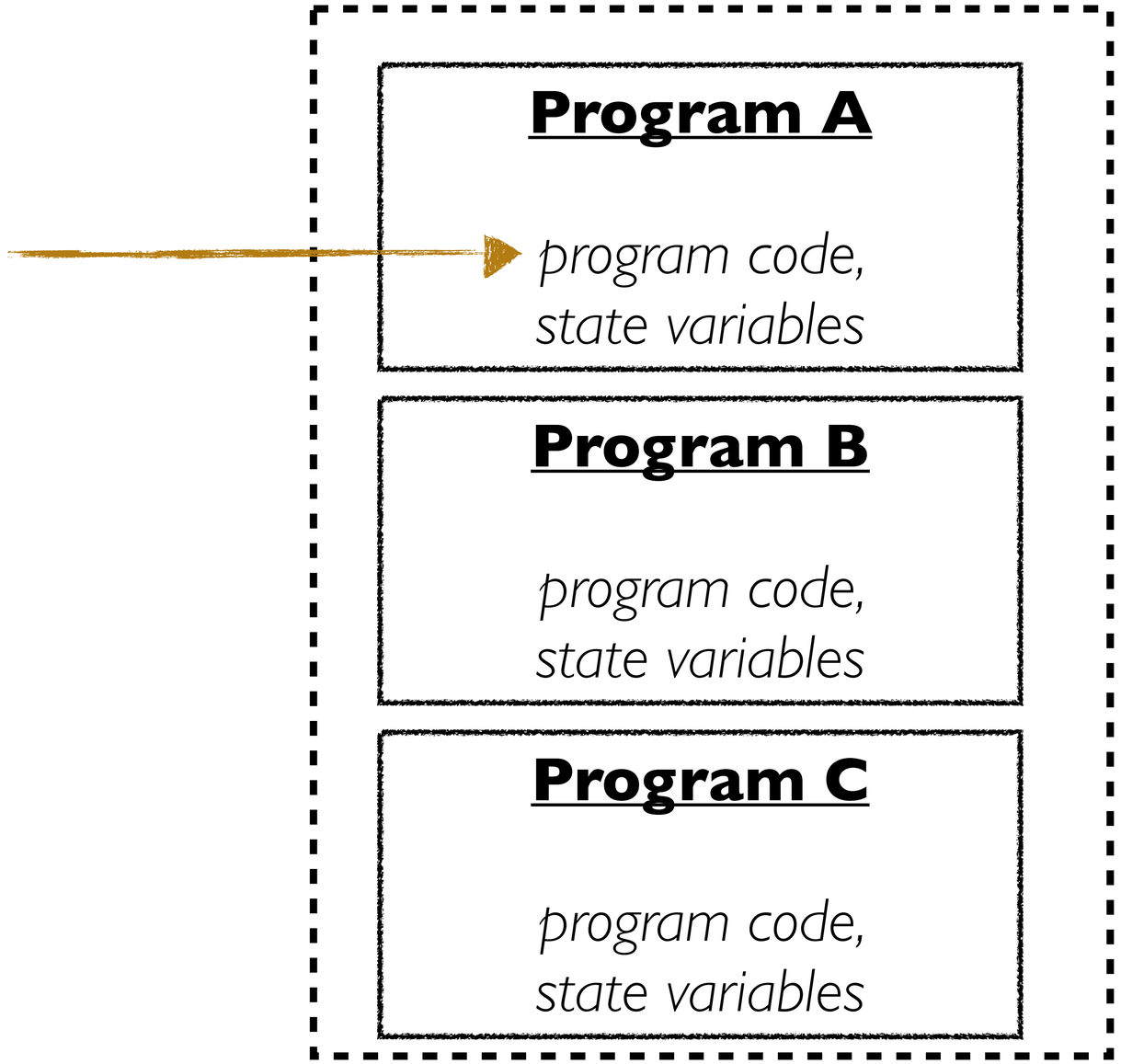
  e.g., "deploy this new program into the network"

Q: What should this code be written in?



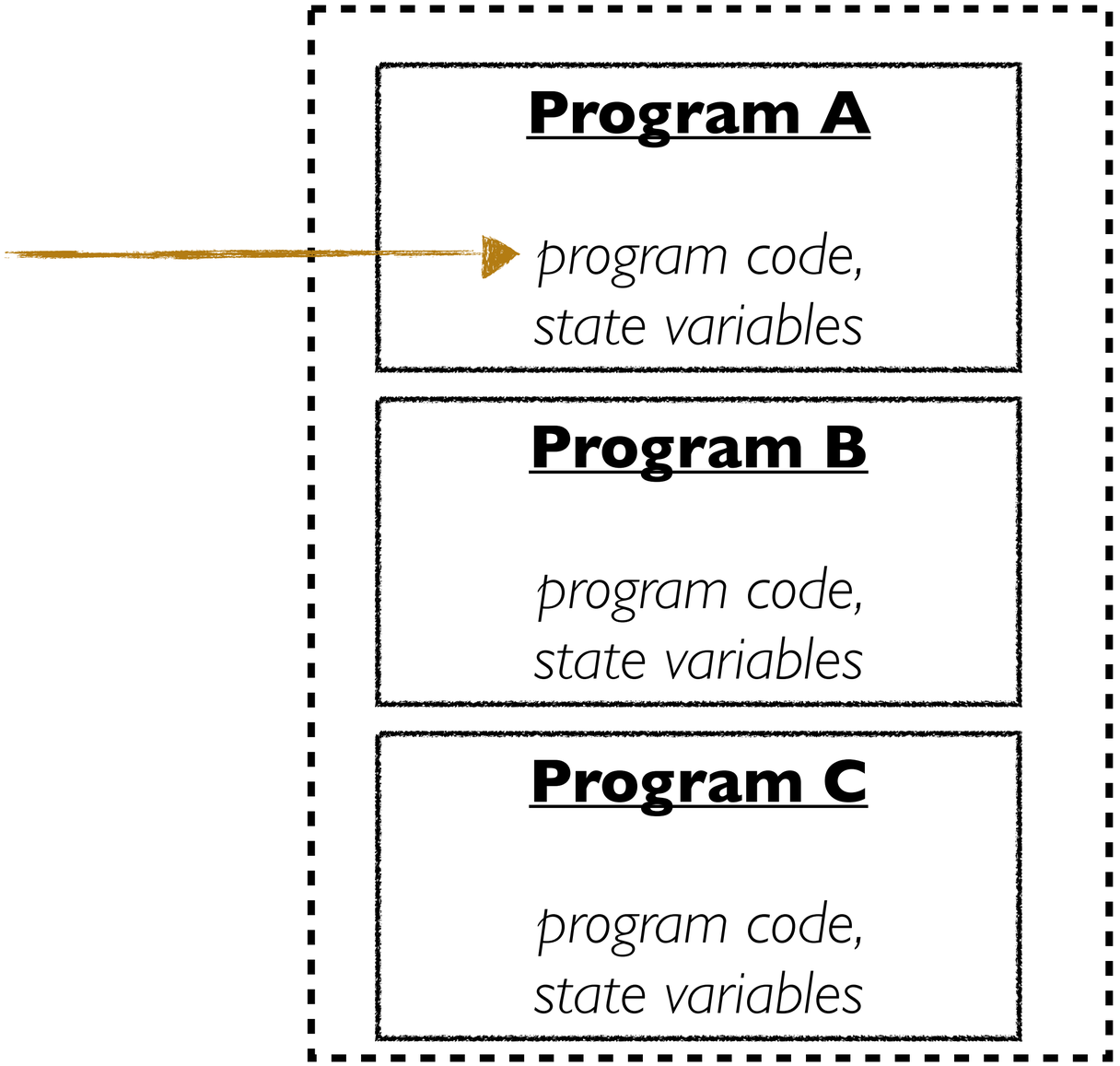
Q: What should this code be written in?

A: Javascript



Q: What should this code be written in?

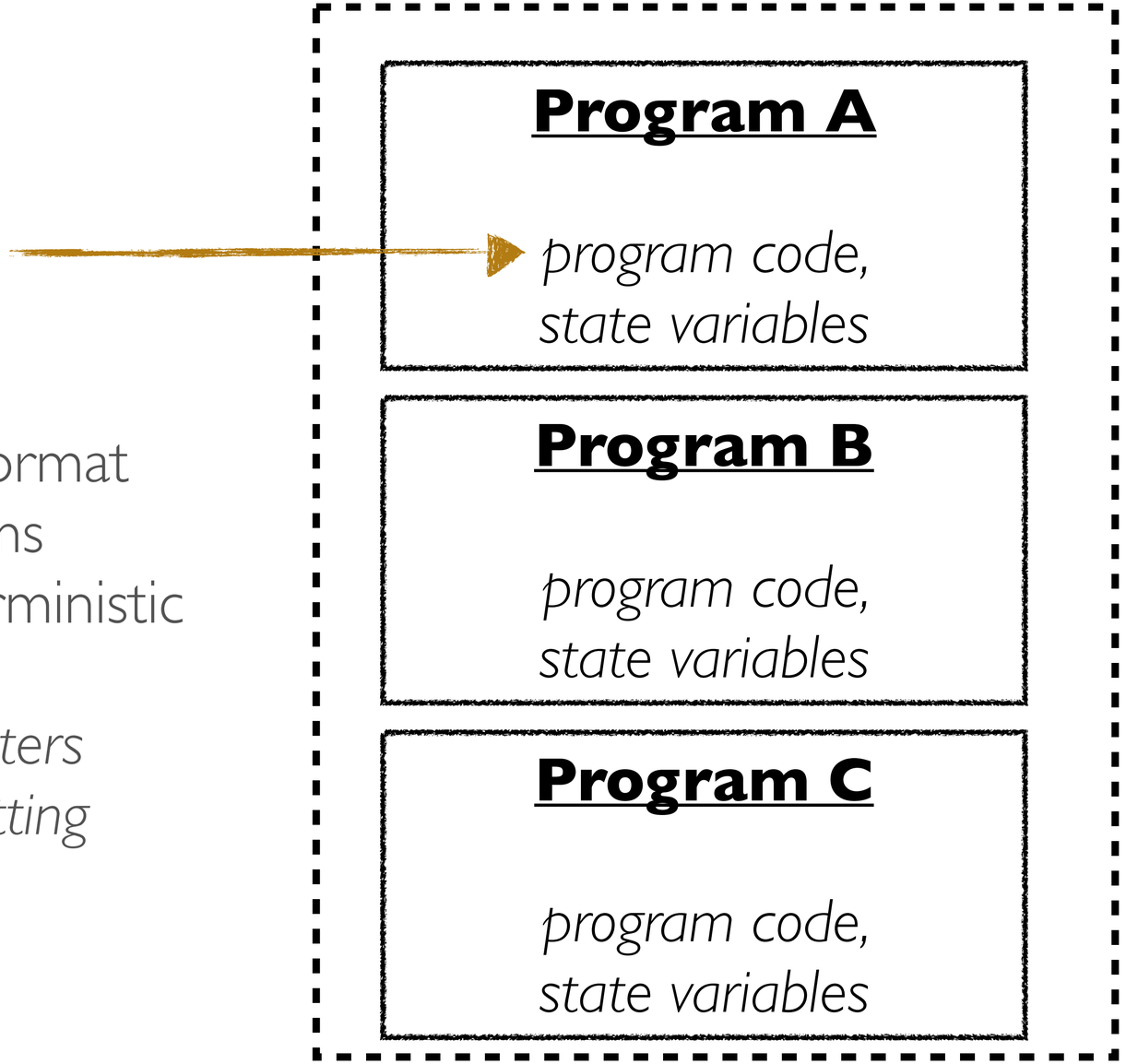
A: Javascript



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

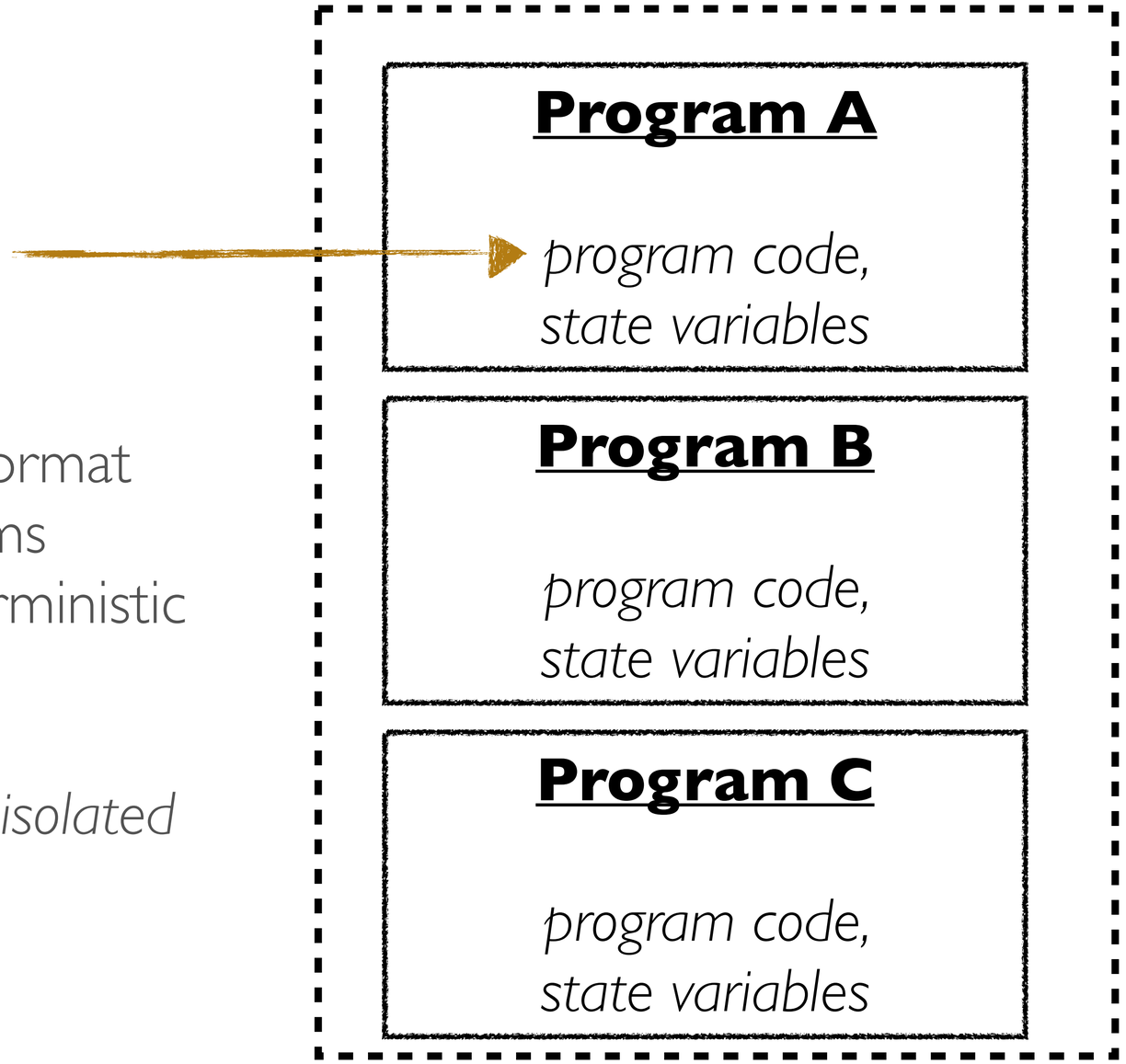
(We can't have different computers running the same code and getting different results!)



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

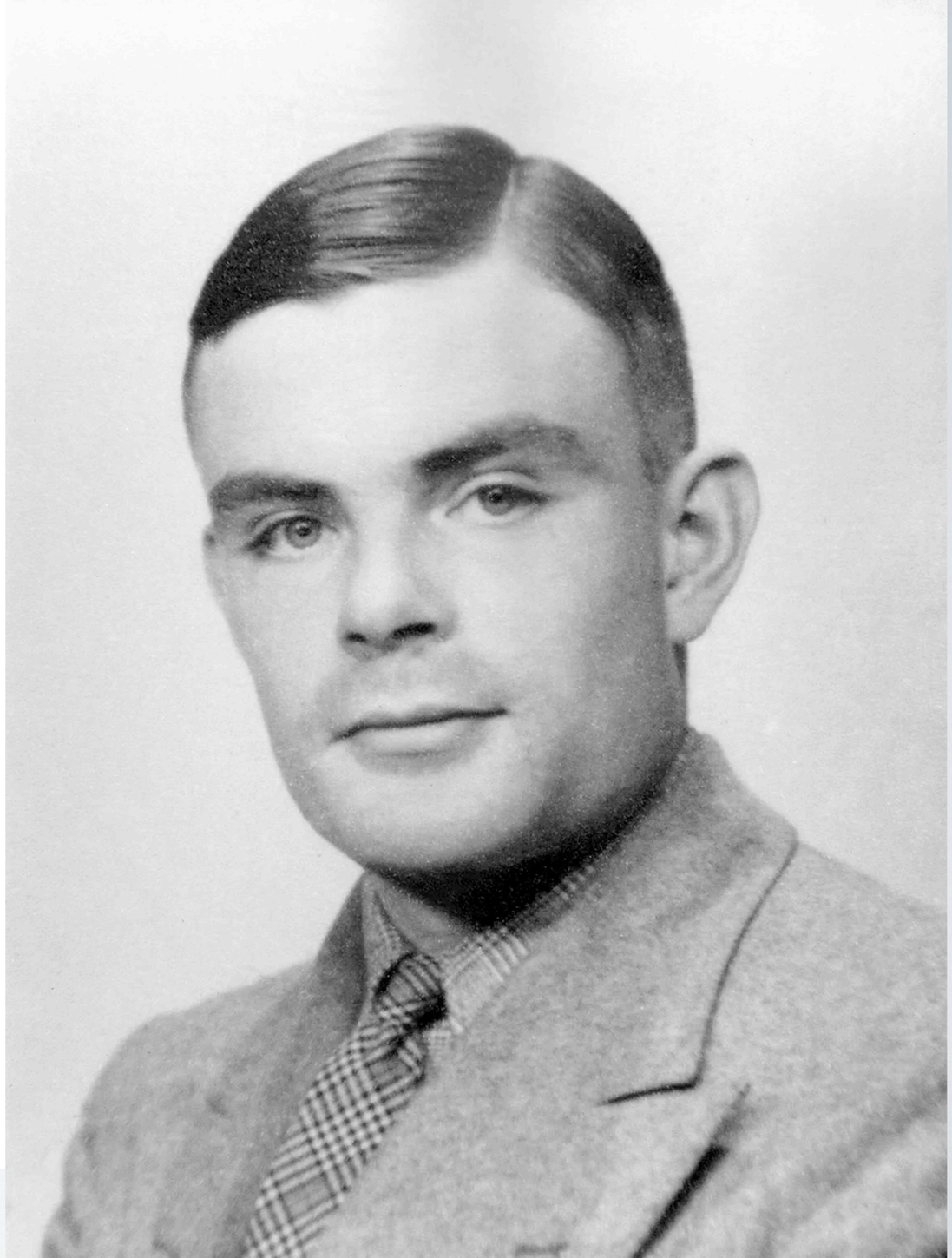
(Also should be memory-safe and easily contained within an isolated virtual machine or sandbox.)



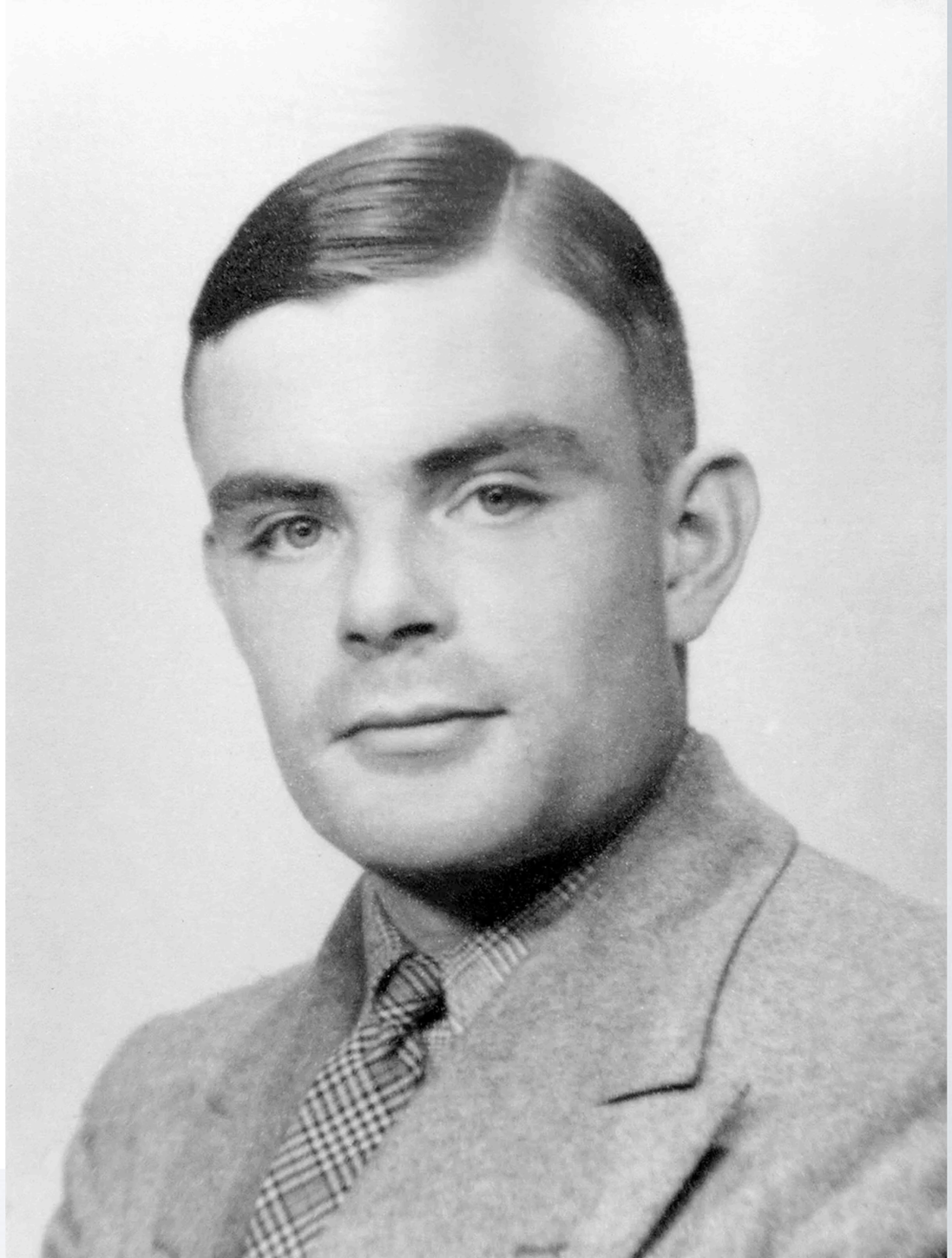
What if running a program takes too long?

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

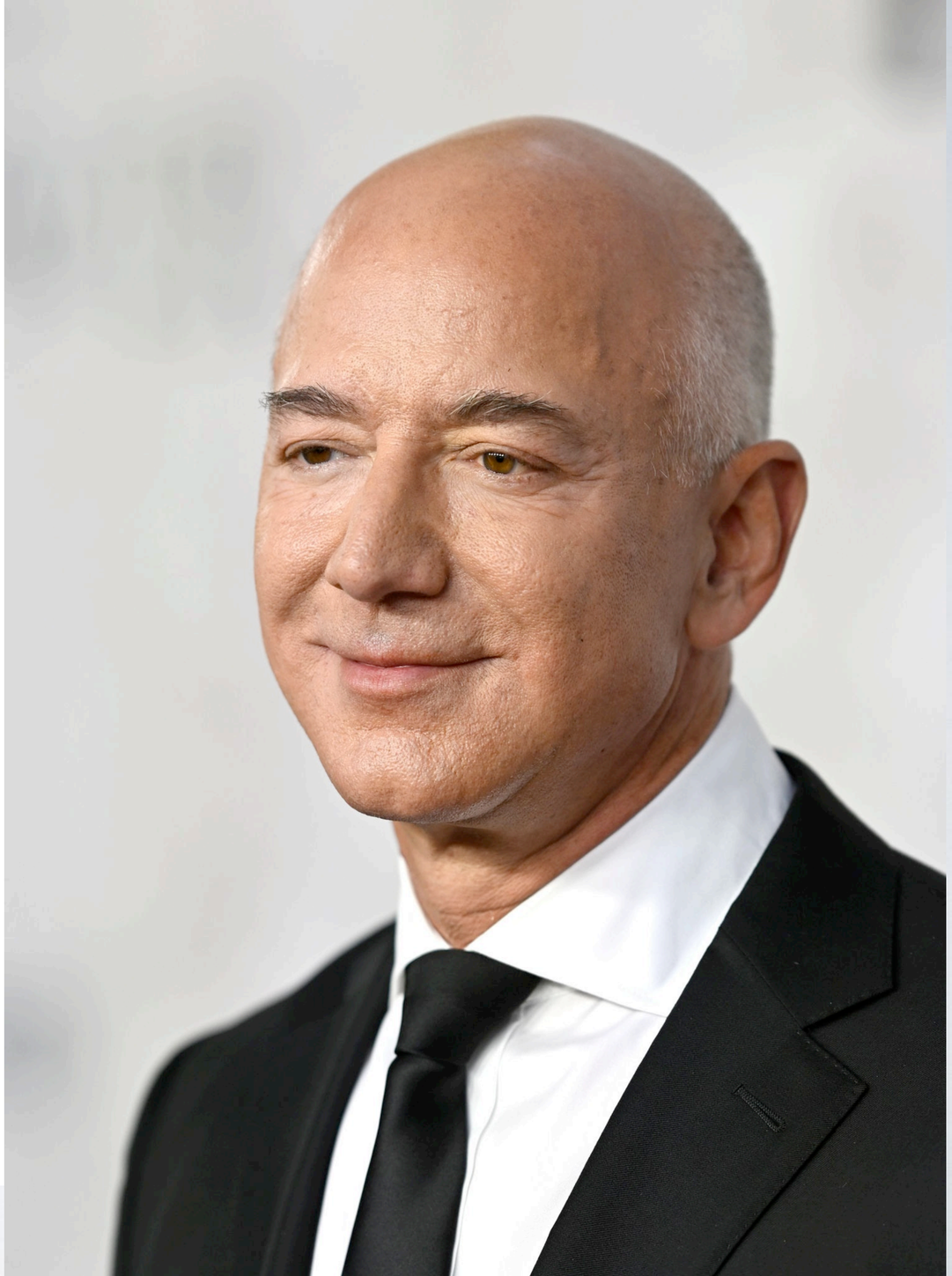
- We cannot allow each stateUpdate to run for an unlimited number of cycles
  - Otherwise people will gum up the network and cause it to “halt”!
  - We could solve the halting problem…
- Ditto with state variables:
  - Programs can’t be allowed to just fill up the database with junk



- Answer #1: cap resource consumption
  - Put a "max cycles allowed" limit on each stateUpdate function
  - When a program runs for too many steps, abort it
    - What do we do with any partial state changes it's made?
  - We can place similar caps on database (storage) consumption
    - However, this might be annoying!



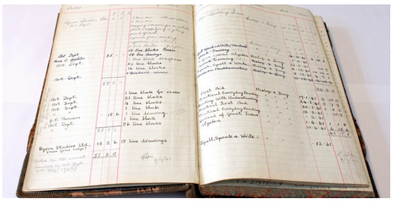
- Answer #2: charge for resource consumption
  - If we have cryptocurrency in this network, we can charge people for compute and storage resources!
    - We'll need a universal system for measuring compute cycles (don't want measurement to be wildly varying for different processors)
    - We may also need a "fee market" that adjusts the price of resources when the network gets congested



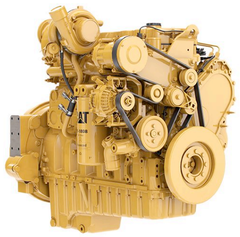
How does this network stay in consensus?

- Network with single node (N=1)
  - One computer, has a database with program code and program state, ordering logic, VM
    - Receives transactions
    - Orders them into blocks (like Bitcoin)
    - Executes each transaction
    - Updates the database

Database
(program code/state, blocks, transactions)



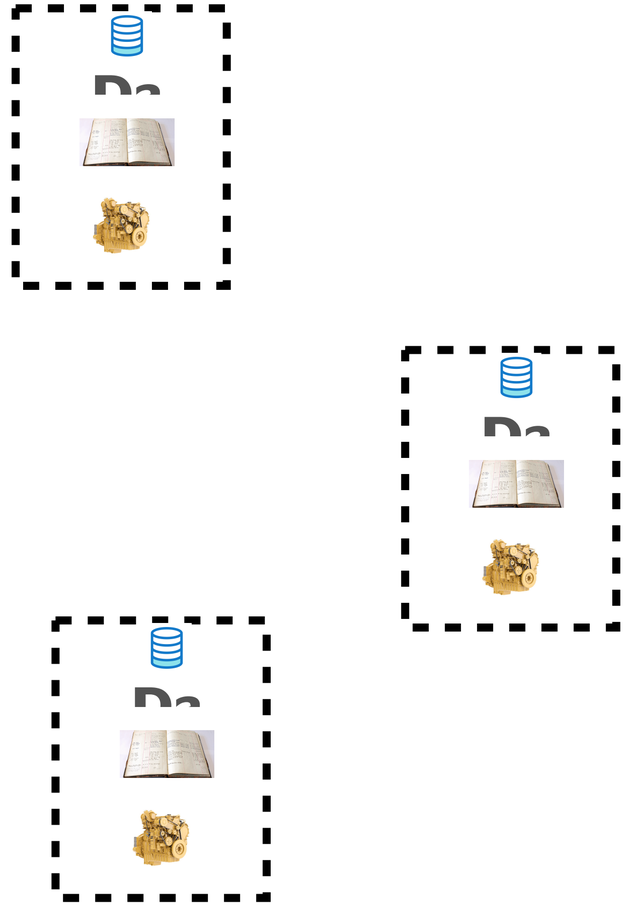
Transaction ordering



Execution engine (VM)

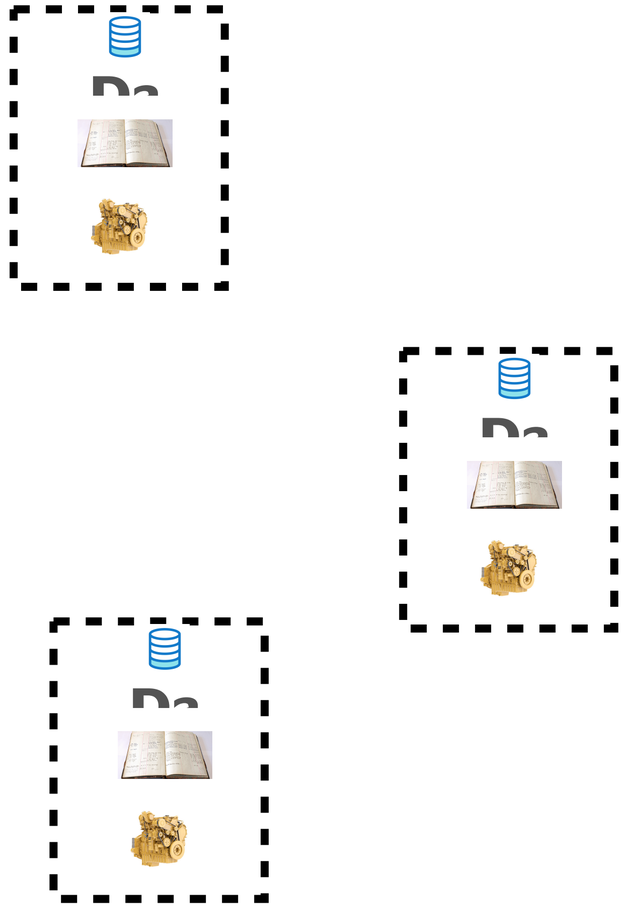
Example node

How do many nodes stay in sync?



How do many nodes stay in sync?

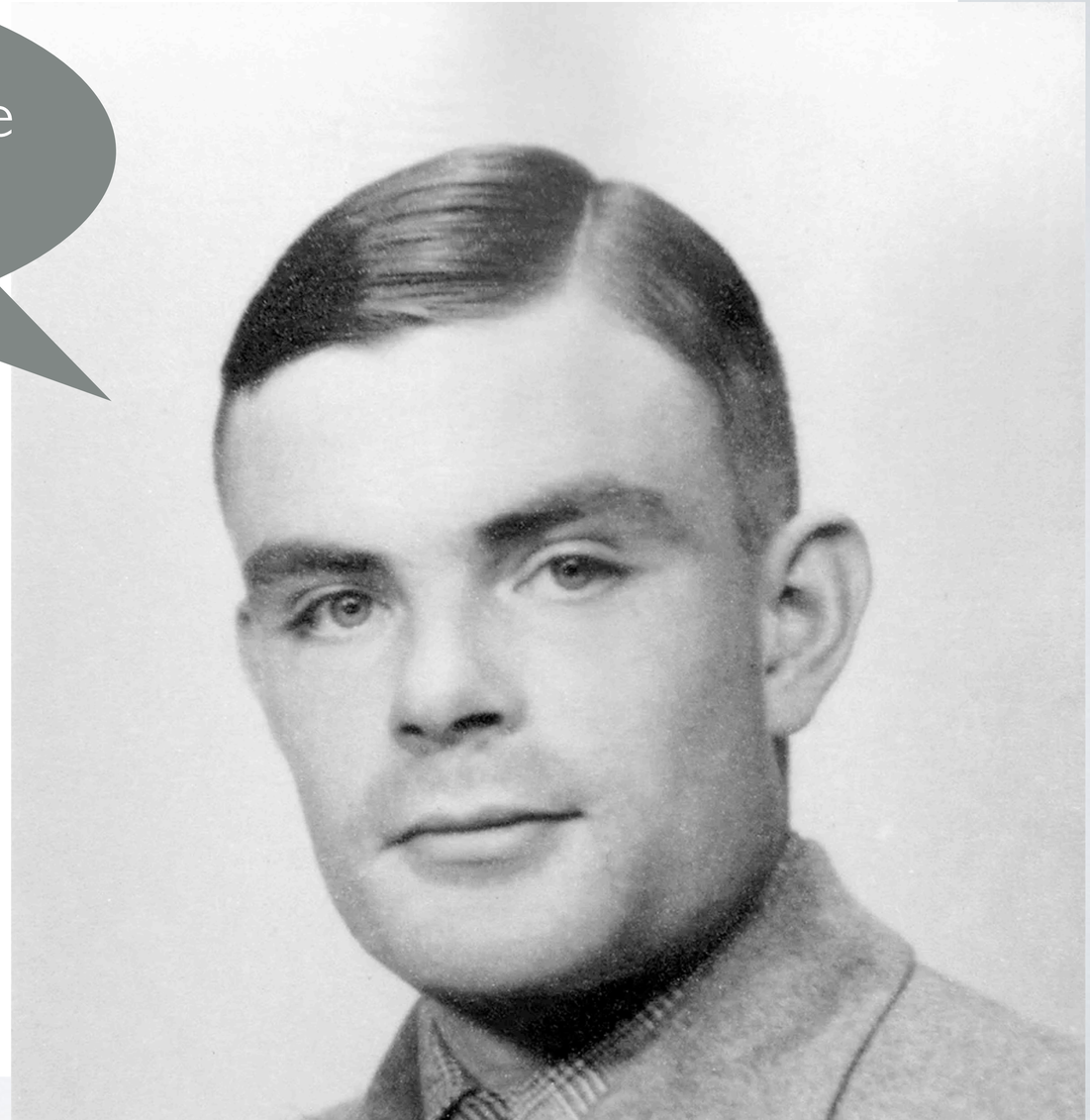
- Idea: just agree on transaction ordering?
  - If all nodes start from the same state (empty database)
  - And all nodes agree on a unique transaction ledger, execute transactions in that order (e.g., Bitcoin approach)
  - And all transaction execution is deterministic and completely repeatable
  - **Then… in theory, every node should always end up agreeing on the same overall state!**



How do many nodes stay in sync?

- Idea: just agree on transaction[illegible]?
  - If all nodes start fr[illegible] (empty database)
  - And all nodes agree on a unique transaction ledger, execute transactions in that order (e.g., Bitcoin approach)
  - And all transaction execution is <u>deterministic</u> and completely repeatable
  - Then… in theory, every node should always end up agreeing on the database!

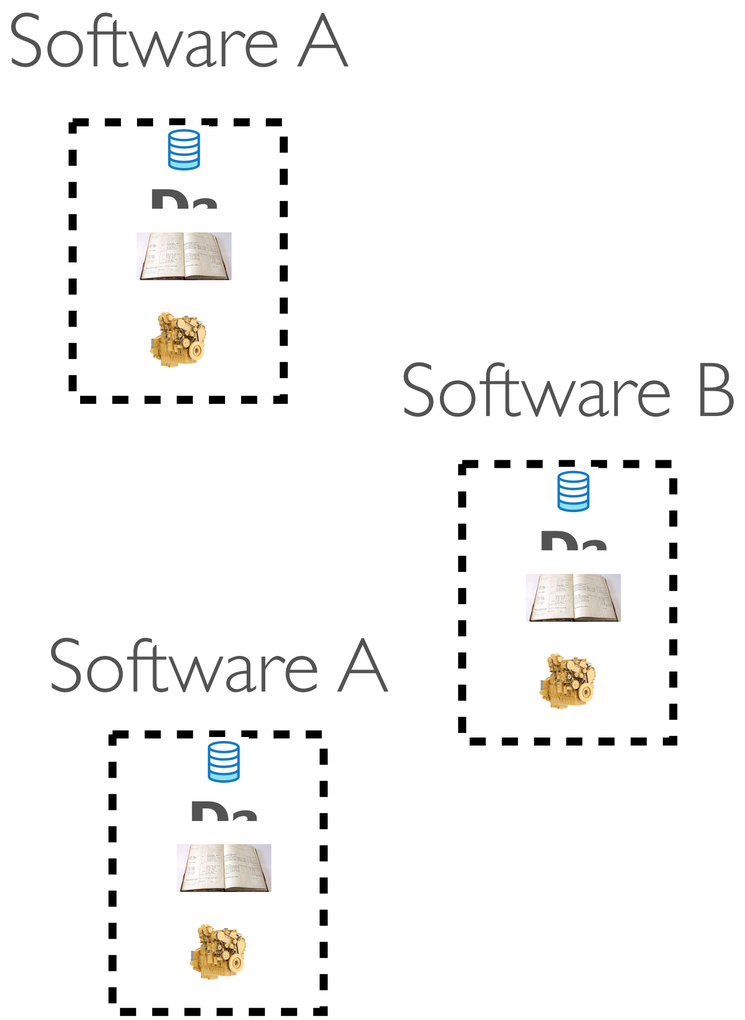
Does this give you the willies?



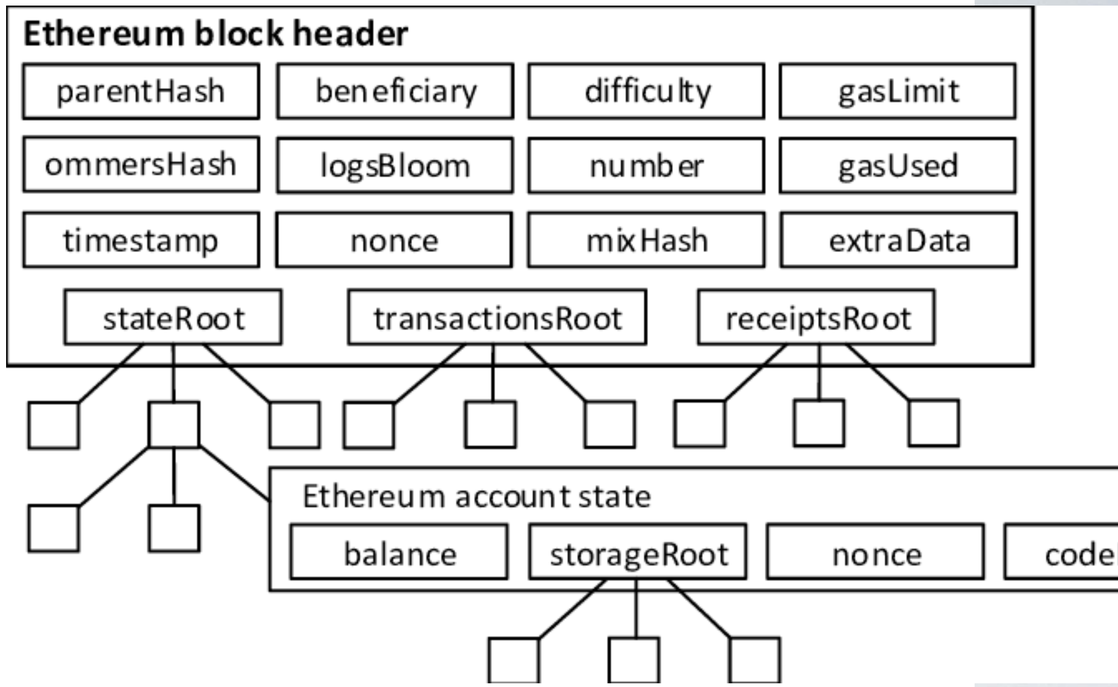
- What if something goes wrong?
  - All nodes can achieve consensus on the contents of the transaction ledger (aka Blockchain)

    Nakamoto consensus: nodes compare the most recent hash of the blockchain!

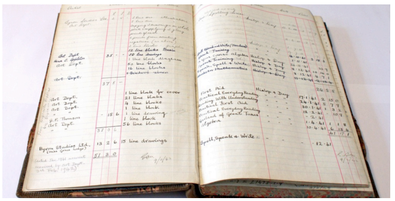
  - But what if A's execution gets a different result than B's execution engine?
    - Scary! The transactions might be in consensus, but the database contents might not be. How do we detect this?



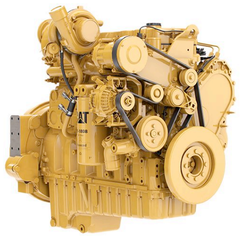
- Answer: hash the database too!
  - We build a hash tree over all the database records
  - Each time we update the database (write things out), we update the hash tree
  - When a transaction runs, we store these hash trees into the block
  - (In theory, these trees can enable light clients to verify execution.)



Database
(program code/state, blocks, transactions)



Transaction ordering



Execution engine (VM)

Example node

Ethereum

- Proposed in 2013 by Vitalik Buterin
  - Basic idea: extend Bitcoin by adding Turing-complete scripting, with full access to chain state ("smart contracts")
  - Scripts run inside of an Ethereum virtual machine (EVM), can call other scripts & each other (recursively)
  - Includes a native token (ETH) to pay for transactions, but users can create additional tokens using contracts