Engineering Distributed Systems that We Can Trust (and also Run)

Ilya Sergey

ilyasergey.net

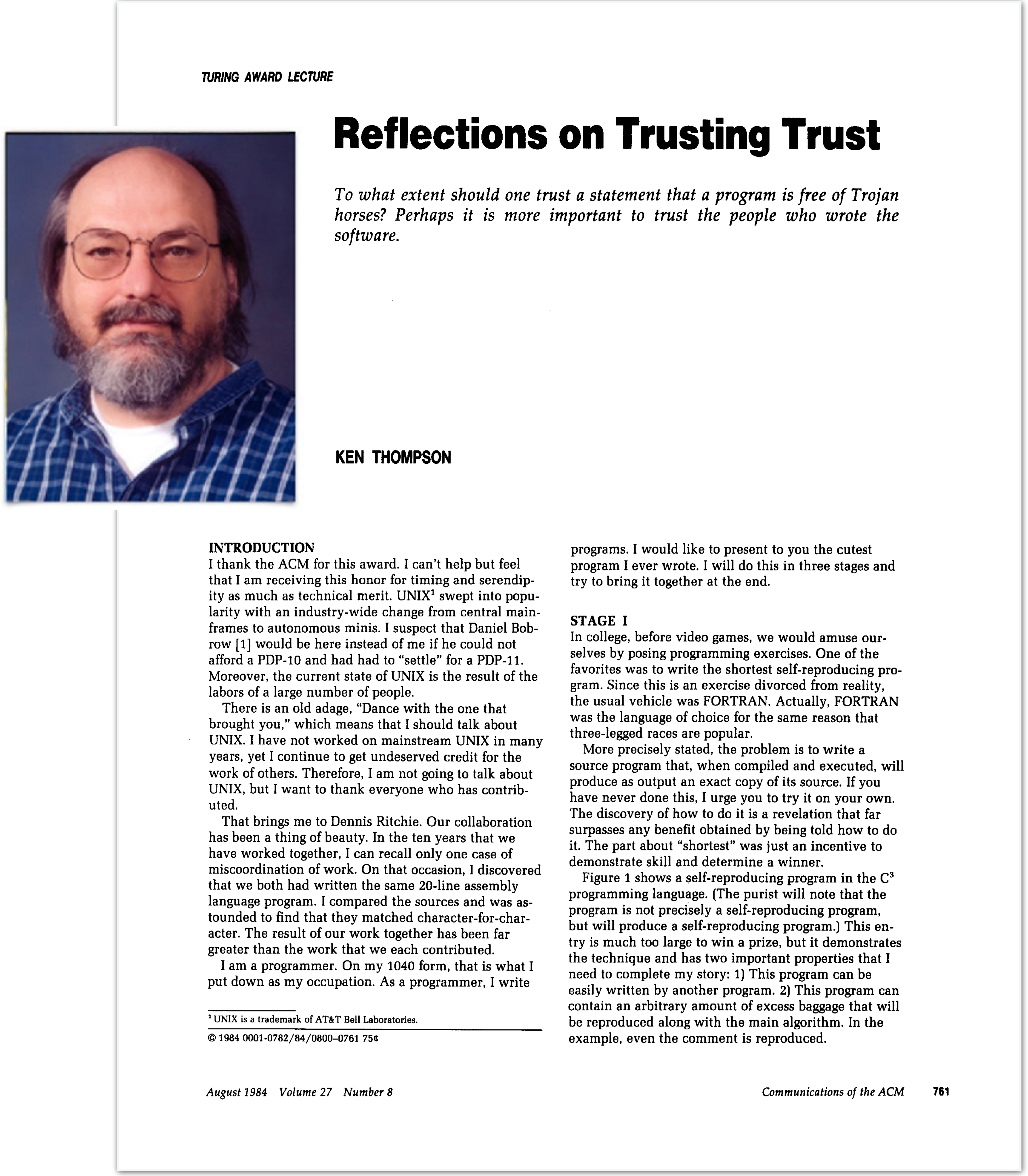
Yale NUS College

NUS National University of Singapore

zilliqa

Trust, n. /trʌst/

Firm belief in the reliability, truth,
or ability of someone or something;
confidence or faith in a person or thing,
or in an attribute of a person or thing.



TURING AWARD LECTURE

Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

KEN THOMPSON

INTRODUCTION
I thank the ACM for this award. I can't help but feel that I am receiving this honor for timing and serendipity as much as technical merit. UNIX[1] swept into popularity with an industry-wide change from central mainframes to autonomous minis. I suspect that Daniel Bobrow [1] would be here instead of me if he could not afford a PDP-10 and had had to "settle" for a PDP-11. Moreover, the current state of UNIX is the result of the labors of a large number of people.

There is an old adage, "Dance with the one that brought you," which means that I should talk about UNIX. I have not worked on mainstream UNIX in many years, yet I continue to get undeserved credit for the work of others. Therefore, I am not going to talk about UNIX, but I want to thank everyone who has contributed.

That brings me to Dennis Ritchie. Our collaboration has been a thing of beauty. In the ten years that we have worked together, I can recall only one case of miscoordination of work. On that occasion, I discovered that we both had written the same 20-line assembly language program. I compared the sources and was astounded to find that they matched character-for-character. The result of our work together has been far greater than the work that we each contributed.

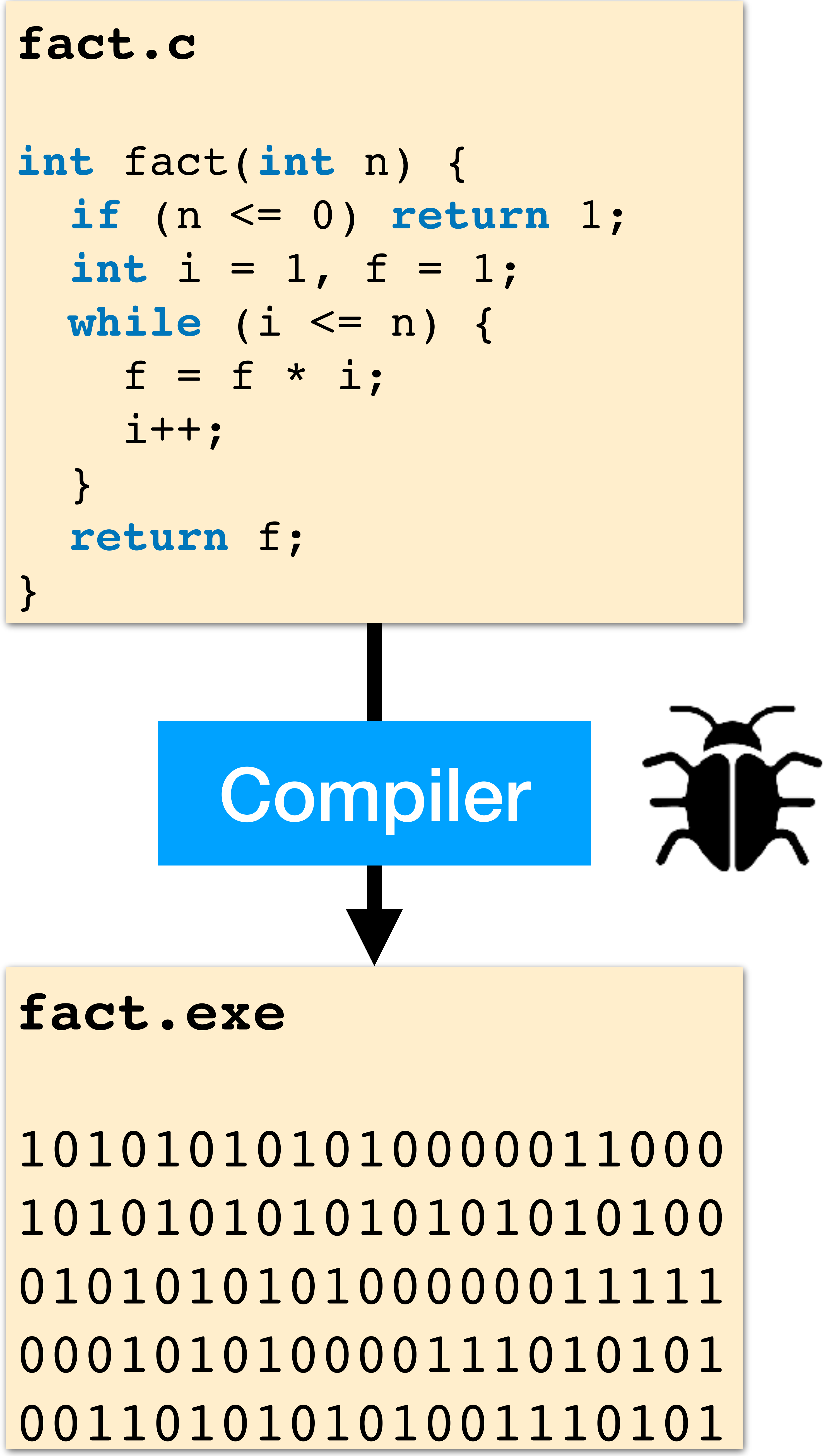
I am a programmer. On my 1040 form, that is what I put down as my occupation. As a programmer, I write programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and try to bring it together at the end.

STAGE I
In college, before video games, we would amuse ourselves by posing programming exercises. One of the favorites was to write the shortest self-reproducing program. Since this is an exercise divorced from reality, the usual vehicle was FORTRAN. Actually, FORTRAN was the language of choice for the same reason that three-legged races are popular.

More precisely stated, the problem is to write a source program that, when compiled and executed, will produce as output an exact copy of its source. If you have never done this, I urge you to try it on your own. The discovery of how to do it is a revelation that far surpasses any benefit obtained by being told how to do it. The part about "shortest" was just an incentive to demonstrate skill and determine a winner.

Figure 1 shows a self-reproducing program in the C[3] programming language. (The purist will note that the program is not precisely a self-reproducing program, but will produce a self-reproducing program.) This entry is much too large to win a prize, but it demonstrates the technique and has two important properties that I need to complete my story: 1) This program can be easily written by another program. 2) This program can contain an arbitrary amount of excess baggage that will be reproduced along with the main algorithm. In the example, even the comment is reproduced.

[1] UNIX is a trademark of AT&T Bell Laboratories.



TURING AWARD LECTURE

Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

KEN THOMPSON

INTRODUCTION

I thank the ACM for this award. I can't help but feel that I am receiving this honor for timing and serendipity as much as technical merit. UNIX[1] swept into popularity with an industry-wide change from central mainframes to autonomous minis. I suspect that Daniel Bobrow [1] would be here instead of me if he could not afford a PDP-10 and had had to "settle" for a PDP-11. Moreover, the current state of UNIX is the result of the labors of a large number of people.

There is an old adage, "Dance with the one that brought you," which means that I should talk about UNIX. I have not worked on mainstream UNIX in many years, yet I continue to get undeserved credit for the work of others. Therefore, I am not going to talk about UNIX, but I want to thank everyone who has contributed.

That brings me to Dennis Ritchie. Our collaboration has been a thing of beauty. In the ten years that we have worked together, I can recall only one case of miscoordination of work. On that occasion, I discovered that we both had written the same 20-line assembly language program. I compared the sources and was astounded to find that they matched character-for-character. The result of our work together has been far greater than the work that we each contributed.

I am a programmer. On my 1040 form, that is what I put down as my occupation. As a programmer, I write programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and try to bring it together at the end.

STAGE I

In college, before video games, we would amuse ourselves by posing programming exercises. One of the favorites was to write the shortest self-reproducing program. Since this is an exercise divorced from reality, the usual vehicle was FORTRAN. Actually, FORTRAN was the language of choice for the same reason that three-legged races are popular.

More precisely stated, the problem is to write a source program that, when compiled and executed, will produce as output an exact copy of its source. If you have never done this, I urge you to try it on your own. The discovery of how to do it is a revelation that far surpasses any benefit obtained by being told how to do it. The part about "shortest" was just an incentive to demonstrate skill and determine a winner.

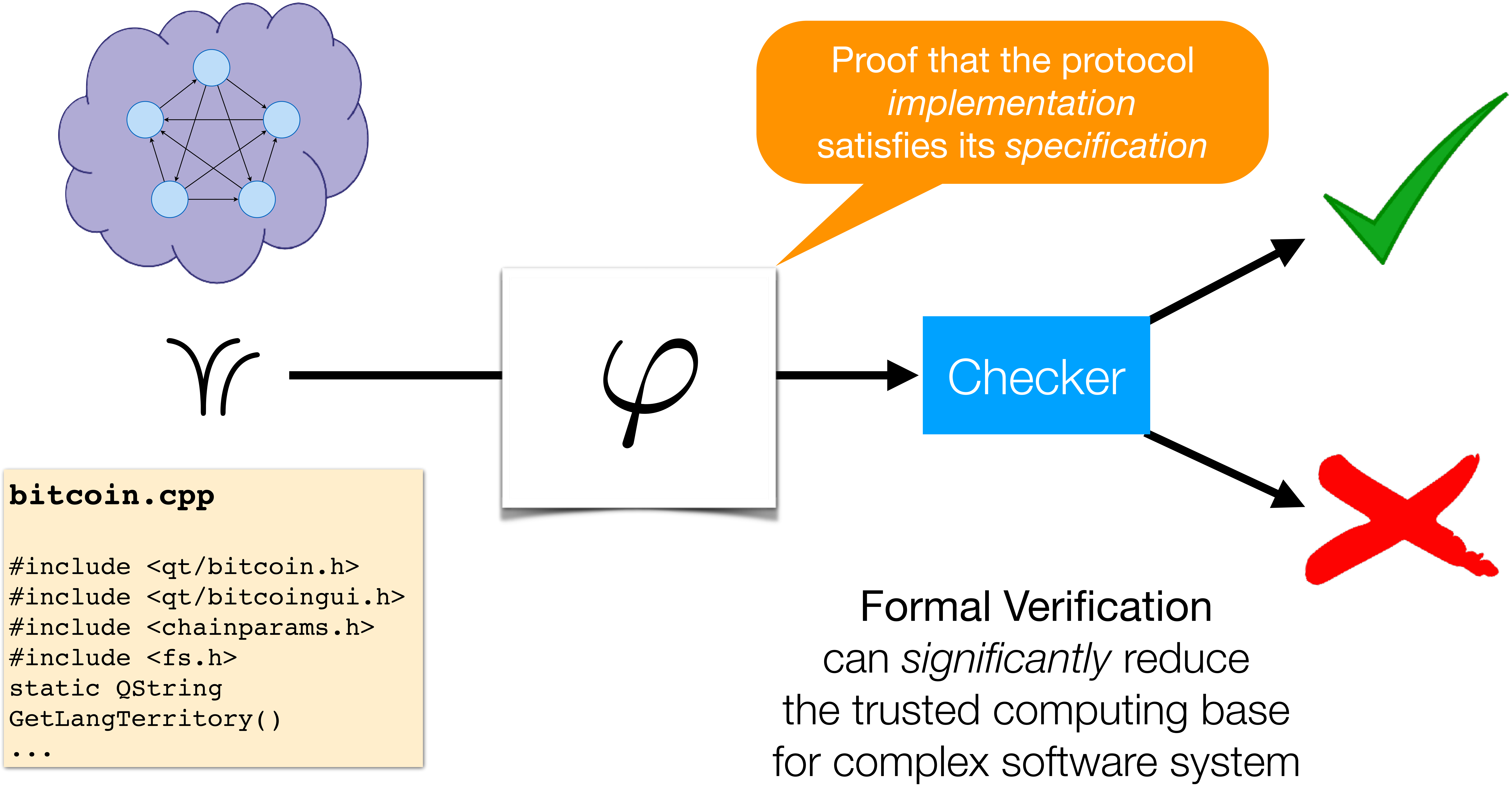
Figure 1 shows a self-reproducing program in the C[3] programming language. (The purist will note that the program is not precisely a self-reproducing program, but will produce a self-reproducing program.) This entry is much too large to win a prize, but it demonstrates the technique and has two important properties that I need to complete my story: 1) This program can be easily written by another program. 2) This program can contain an arbitrary amount of excess baggage that will be reproduced along with the main algorithm. In the example, even the comment is reproduced.

[1] UNIX is a trademark of AT&T Bell Laboratories.

The moral is obvious. You can't trust code that you did not totally create yourself. [...] No amount of source-level verification or scrutiny will protect you from using untrusted code.

Shall we Trust Blockchain Implementations?

- Vulnerabilities due to **high-level** protocol design:
  - **EOS** Transaction Congestion Attack (Jan 2019):
    Different priorities in transaction processing resulted in adversarial denial-of-service.
- **Low-level** implementation bugs violating the protocol.
  - **Bitcoin** Value Overflow Incident (August 2010):
    due to data structure bug, the implementation accepts a malformed transaction.
  - **Bitcoin** Accidental Hard Fork (March 2013):
    Switching from BerkeleyDB to LevelDB unintentionally redefined the consensus.
  - **Bitcoin** Inflation Bug (September 2018):
    Faulty transaction processing allows for denial-of-service attack (unexploited).


Proof that the protocol *implementation* satisfies its *specification*
φ
Checker
bitcoin.cpp
#include <qt/bitcoin.h>
#include <qt/bitcoingui.h>
#include <chainparams.h>
#include <fs.h>
static QString
GetLangTerritory()
...
Formal Verification
can *significantly* reduce
the trusted computing base
for complex software system

Formal Verification

Proving Correctness of algorithms or software artefacts

with respect to a given rigorous specification

using mathematical reasoning.

Formal Verification

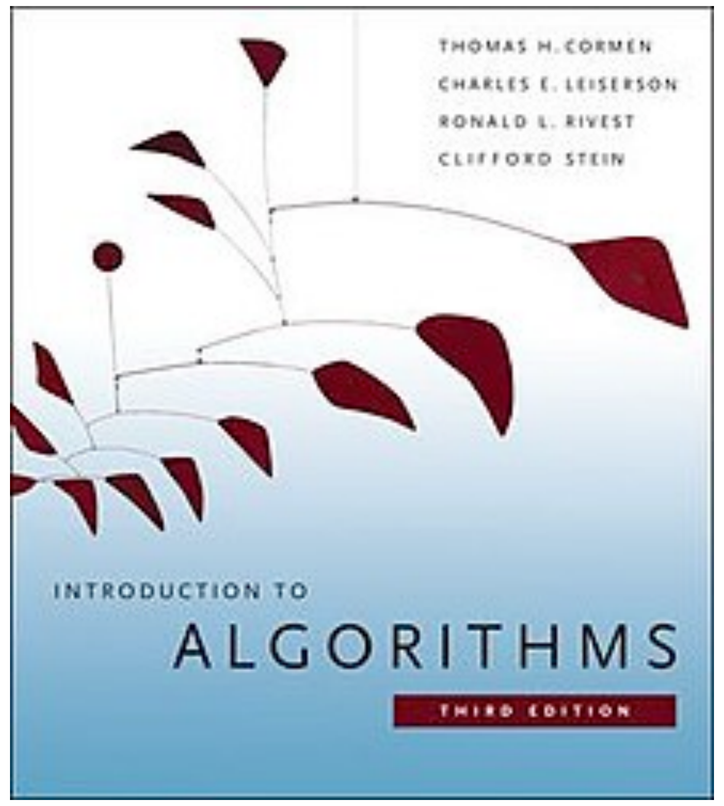
Proving Correctness of algorithms or software artefacts

with respect to a given rigorous specification

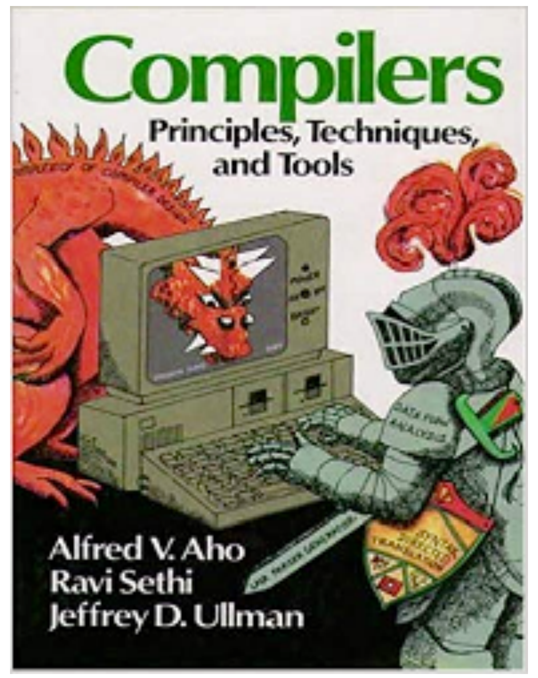
using mathematical reasoning.

Correctness-critical software

- Implementations of textbook algorithms

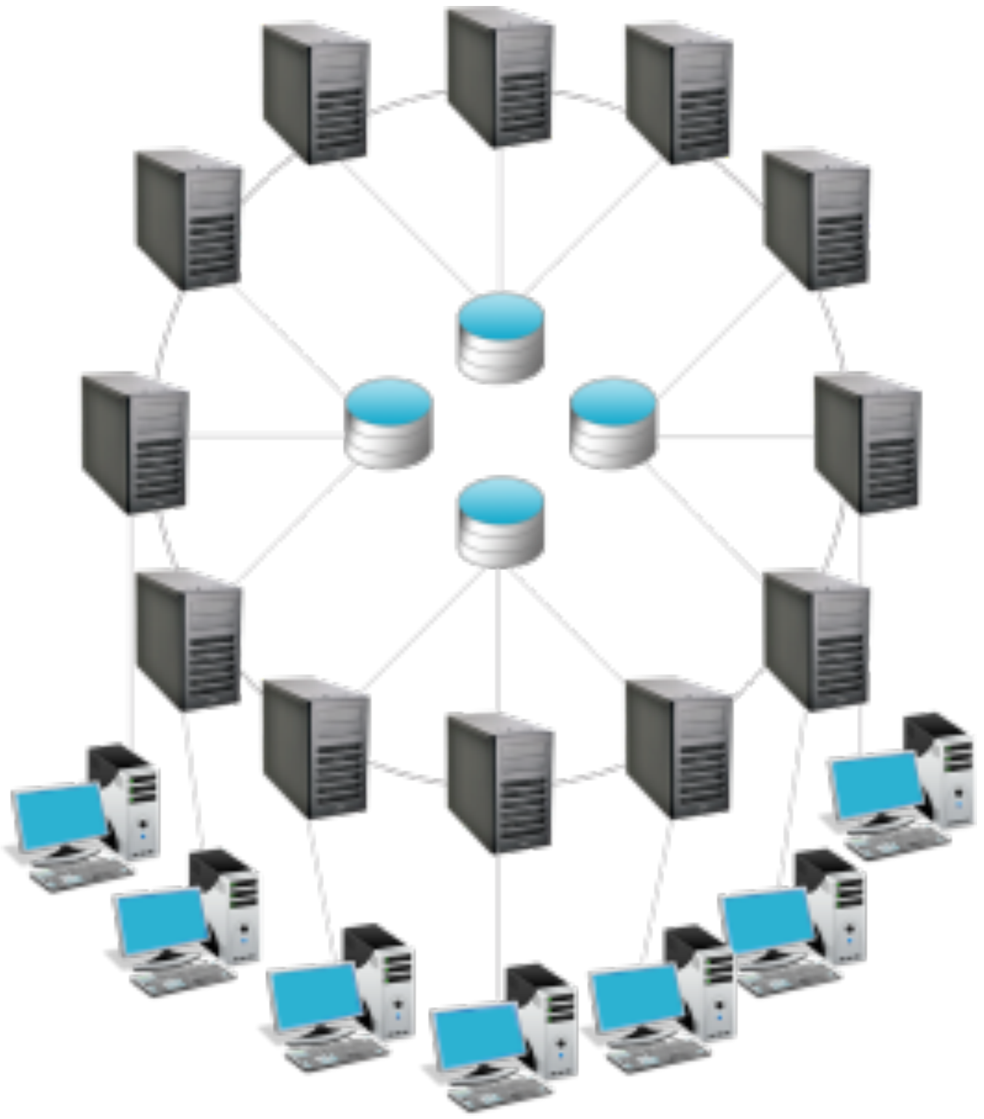


- Operational Systems



- Compilers

- Distributed Systems and their Applications



Formal Verification

Proving Correctness of algorithms or software artefacts

with respect to a given rigorous specification

using mathematical reasoning.

Formal Verification

Proving Correctness of algorithms or software artefacts

with respect to a given rigorous specification

using mathematical reasoning.

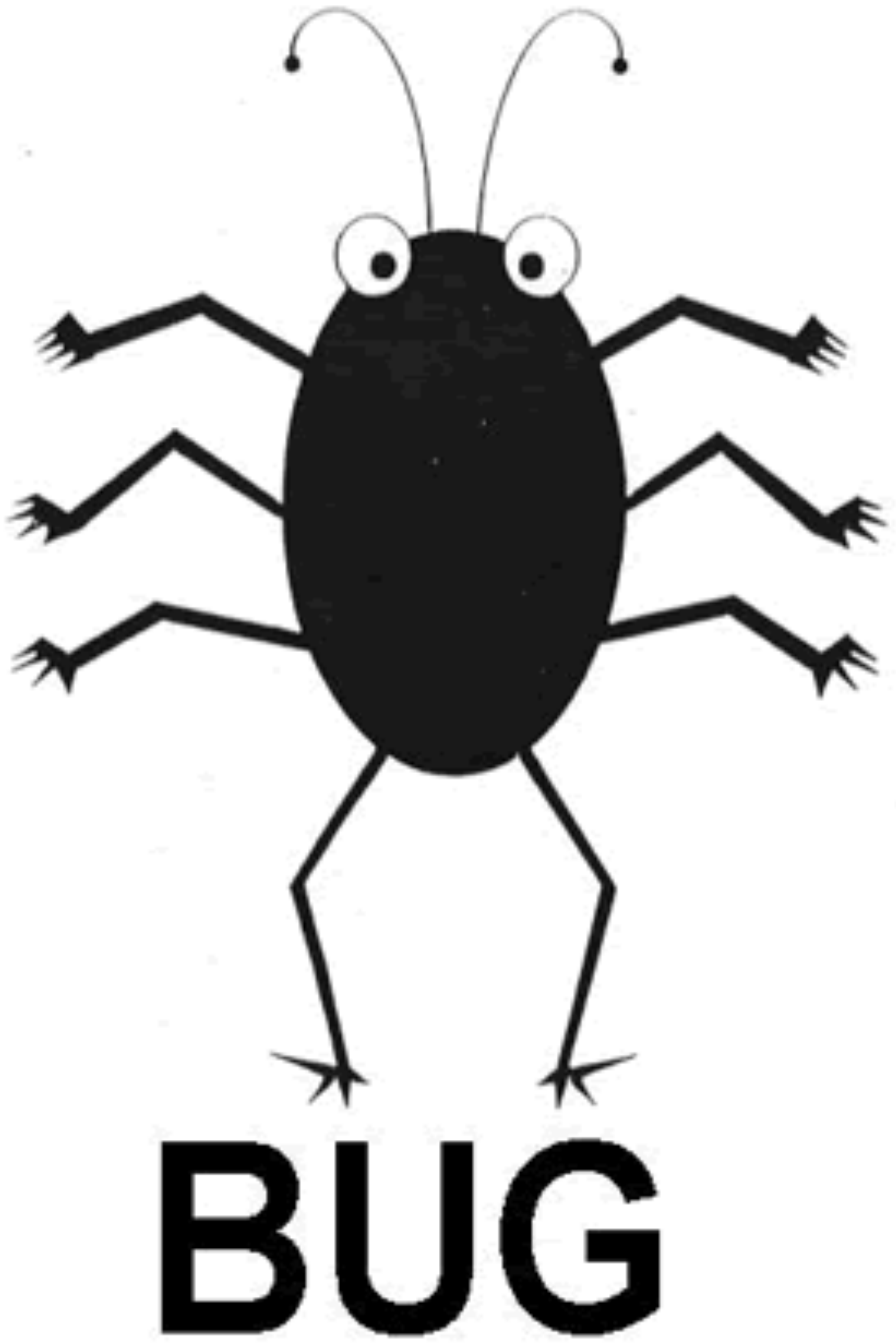
Formal Verification $\neq$ Testing

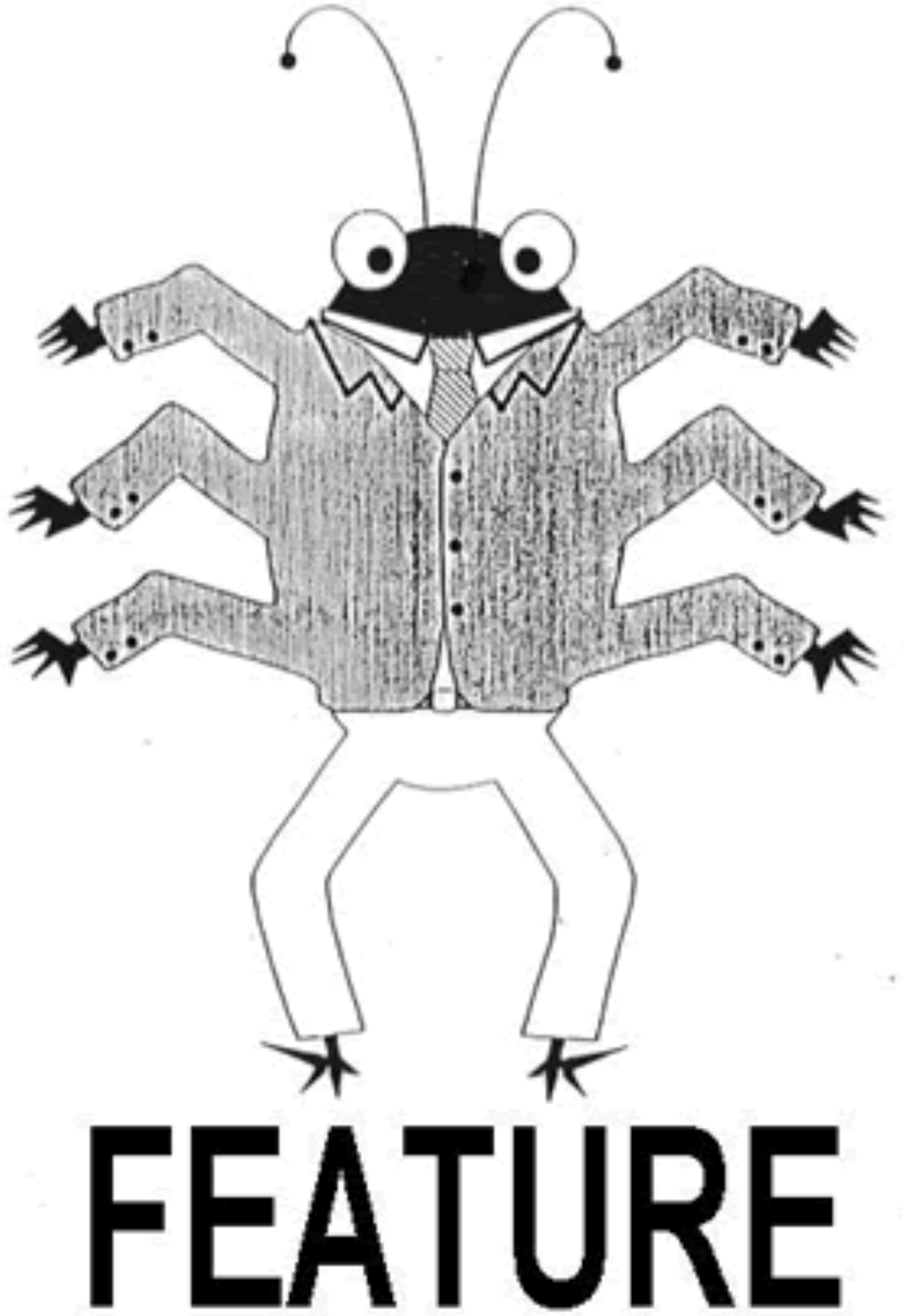
*"Program testing can be used to show the **presence** of bugs, but never to show their **absence**!"*

Edsger W. Dijkstra

But the bugs are in the eye of the beholder!

But the bugs are in the eye of the beholder!





specification

Formal Verification

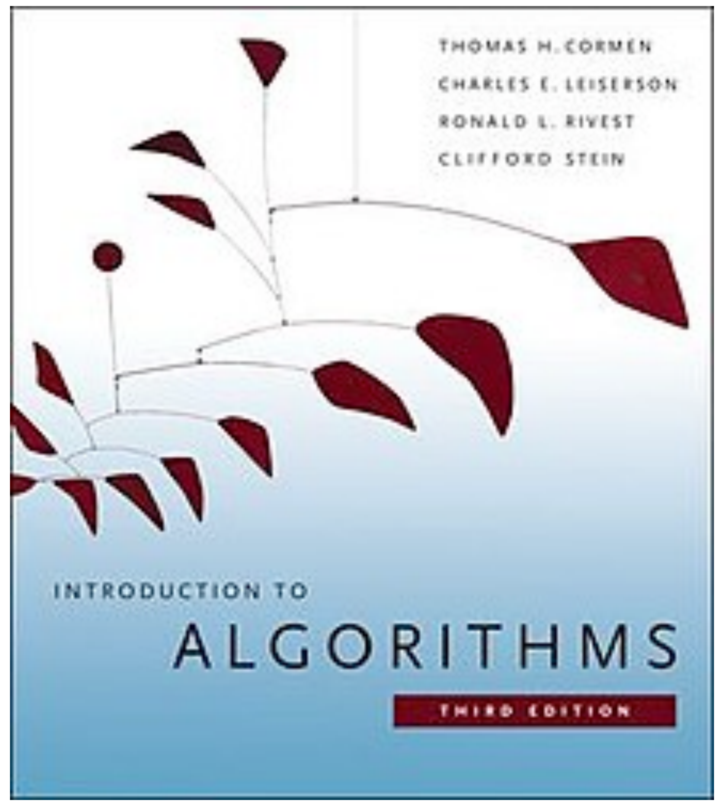
Proving correctness of algorithms or software artefacts

with respect to a given rigorous specification

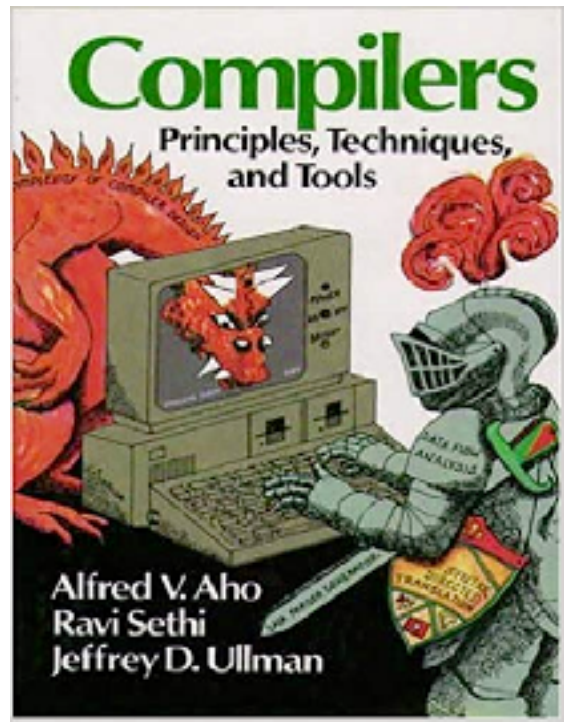
using mathematical reasoning.

Correctness-critical software

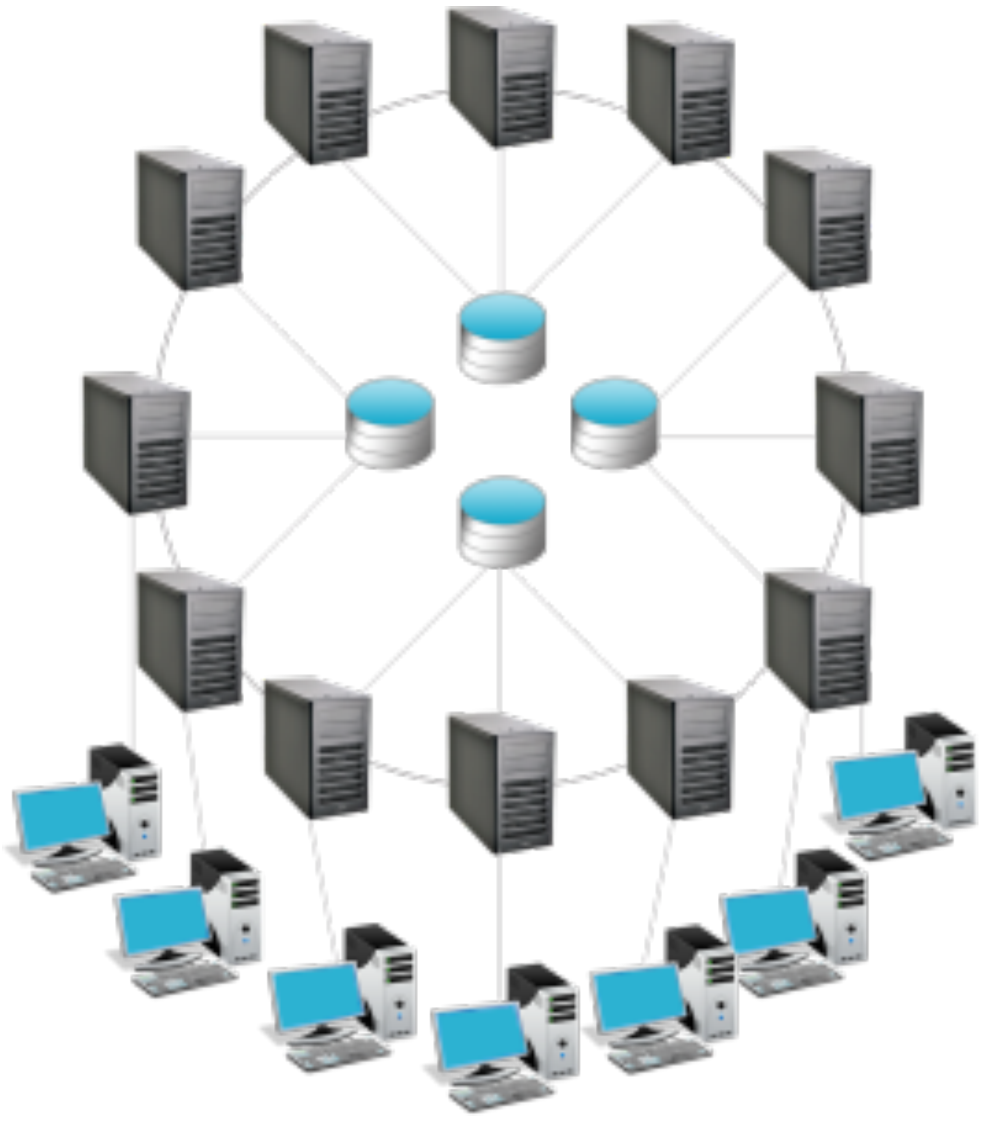
- Implementations of textbook algorithms



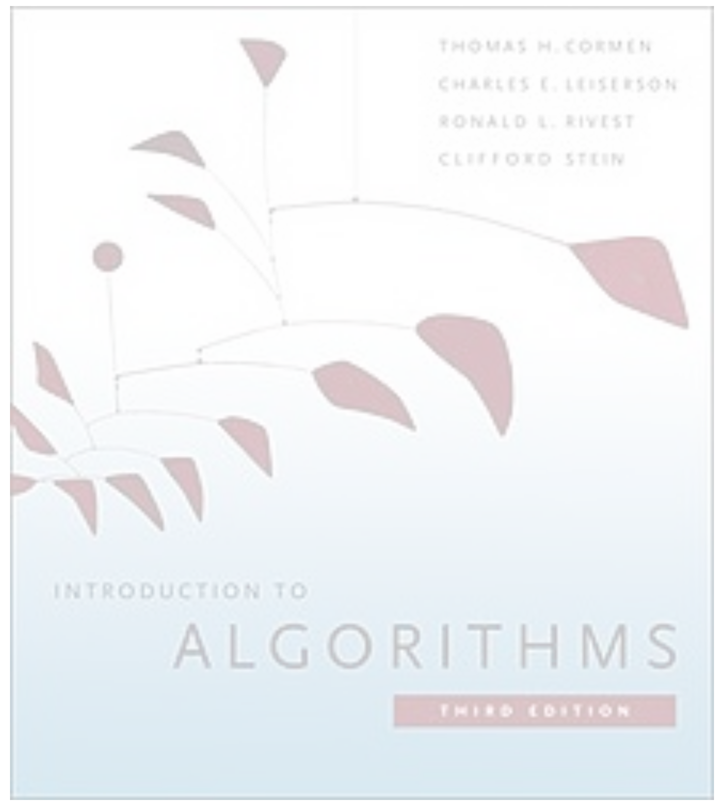
- Operational Systems



- Compilers
- Distributed systems and their applications

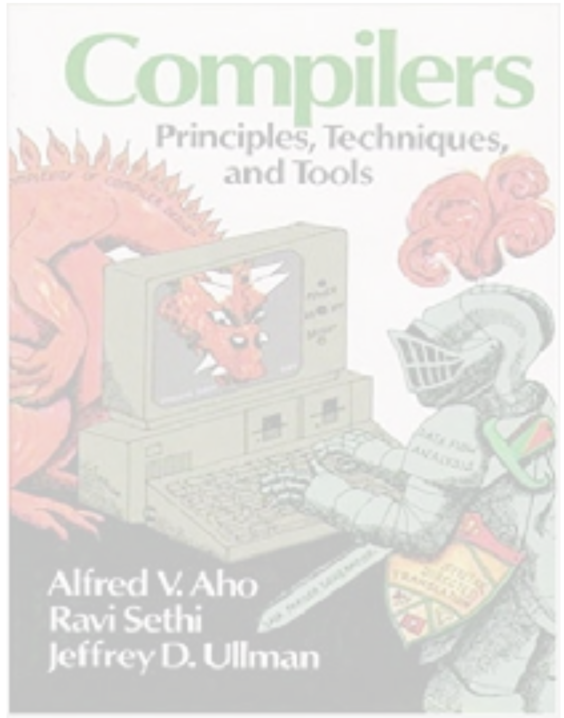


Correctness-critical software

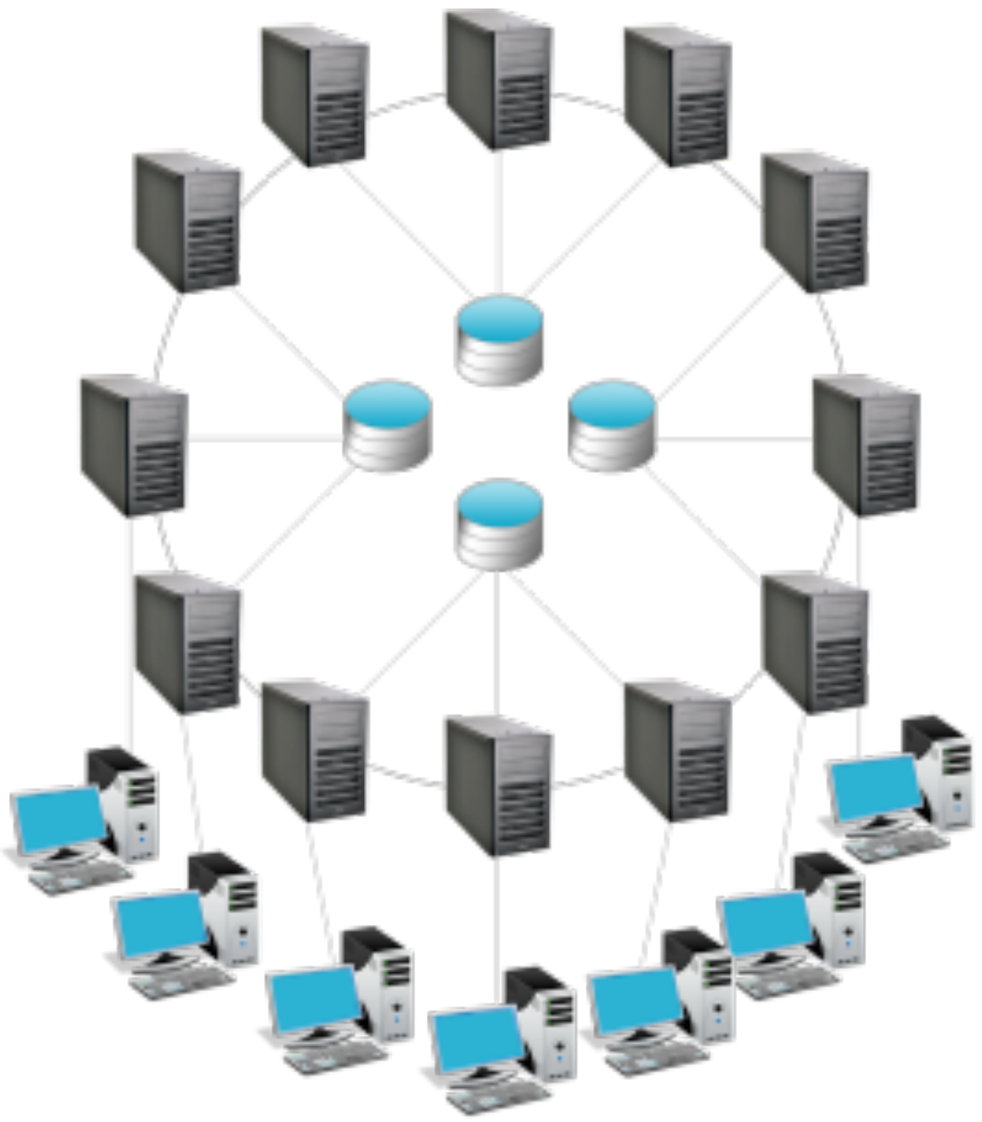


- Implementations of textbook algorithms

- Operational Systems



- Compilers
- Distributed systems and their applications



Running Example:

Toychain

CPP'18

Mechanising Blockchain Consensus

George Pîrlea
University College London, UK
george.pirlea.15@ucl.ac.uk

Ilya Sergey
University College London, UK
i.sergey@ucl.ac.uk

Abstract

We present the first formalisation of a blockchain-based distributed consensus protocol with a proof of its consistency mechanised in an interactive proof assistant.

Our development includes a reference mechanisation of the *block forest* data structure, necessary for implementing provably correct per-node protocol logic. We also define a model of a network, implementing the protocol in the form of a replicated state-transition system. The protocol's executions are modeled via a small-step operational semantics for asynchronous message passing, in which packages can be rearranged or duplicated.

1 Introduction

The notion of decentralised blockchain-based consensus is a tremendous success of the modern science of distributed computing, made possible by the use of basic cryptography, and enabling many applications, including but not limited to cryptocurrencies, smart contracts, application-specific arbitration, voting, *etc.*

In a nutshell, the idea of a distributed consensus protocol based on *blockchains*, or *transaction ledgers*,[1] is rather simple. In all such protocols, a number of stateful nodes (participants) are communicating with each other in an asynchronous message-passing style. In a message, a node (a)

See also ***Towards Mechanising Probabilistic Properties of a Blockchain*** by Gopinathan and Sergey (2019).

What blockchain protocol does

$$\{tx_1, tx_3, tx_5, tx_4, tx_2\}$$

transactions
can be *anything*

- transforms a **set** of transactions into a *globally-agreed* **sequence**
- "distributed timestamp server" (Nakamoto2008)

blockchain
consensus protocol

$$tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2$$

$$\{tx_1, tx_3, tx_5, tx_4, tx_2\}$$

$$\downarrow$$

$$[tx_5, tx_3] \to [tx_4] \to [tx_1, tx_2]$$

$$\downarrow$$

$$tx_5 \to tx_3 \to tx_4 \to tx_1 \to tx_2$$

$$\{tx_1, tx_3, tx_5, tx_4, tx_2\}$$

$$\downarrow$$

$$[tx_5, tx_3] \leftarrow [tx_4] \leftarrow [tx_1, tx_2]$$

$$\downarrow$$

$$tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2$$

$$\{tx_1, tx_3, tx_5, tx_4, tx_2\}$$

$$[] \leftarrow [tx_5, tx_3] \leftarrow [tx_4] \leftarrow [tx_1, tx_2]$$

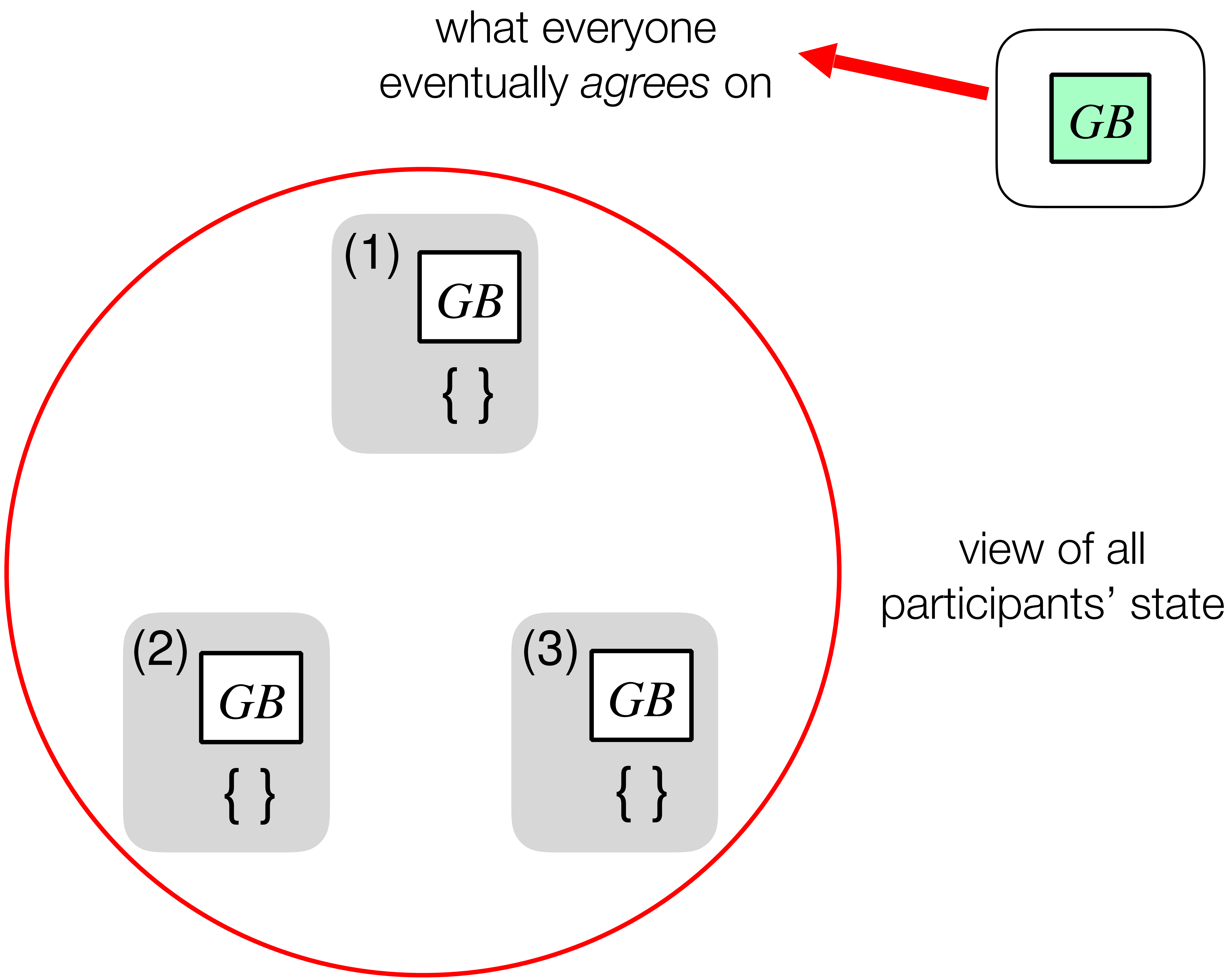
GB = genesis block

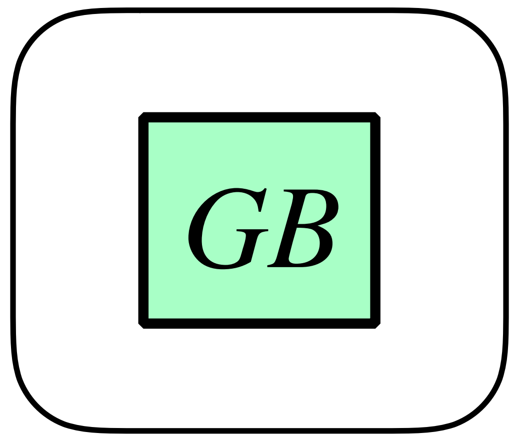
$$tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2$$

How it works

- **distributed**
  - multiple <u>nodes</u>

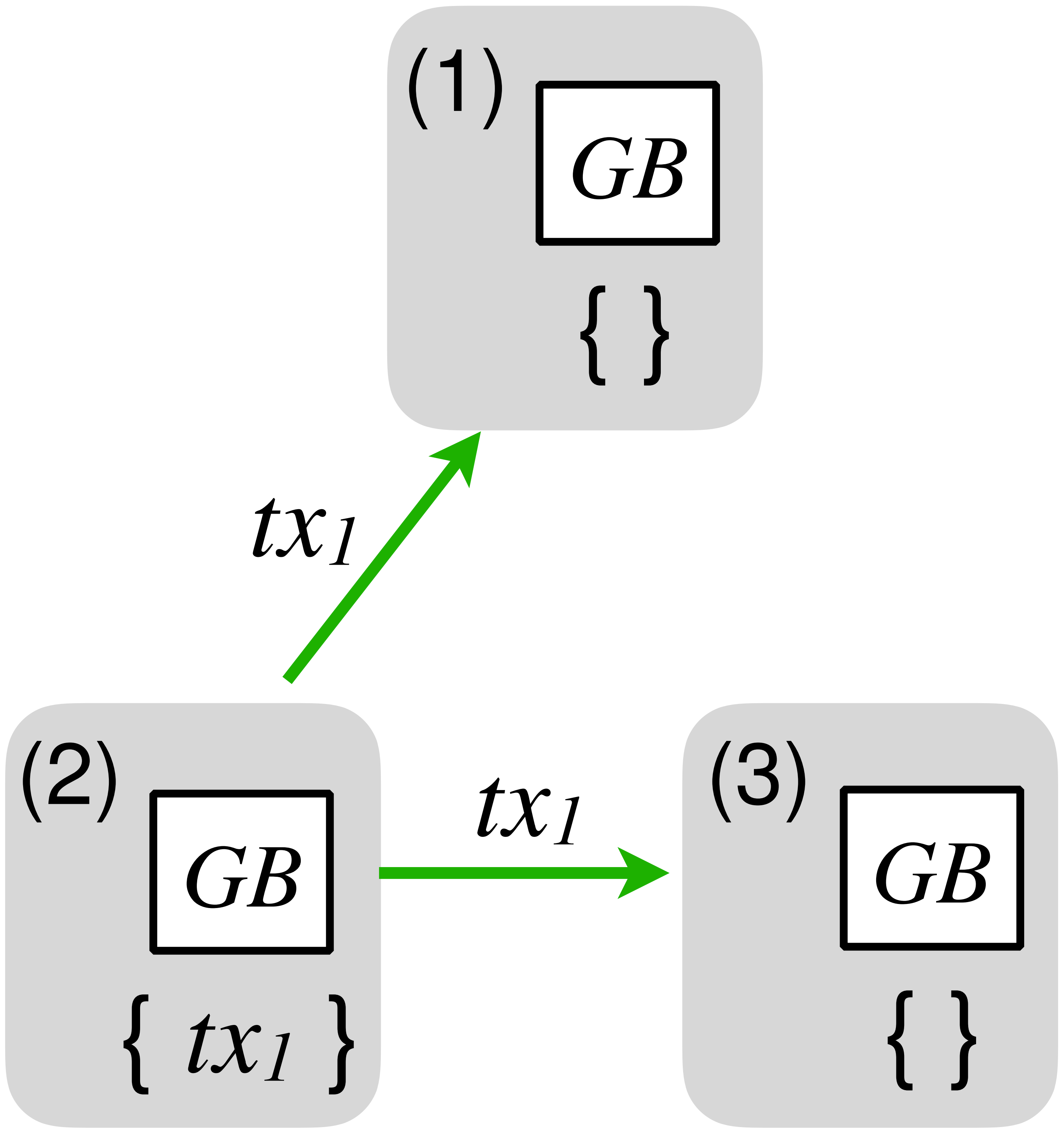
- all start with same GB





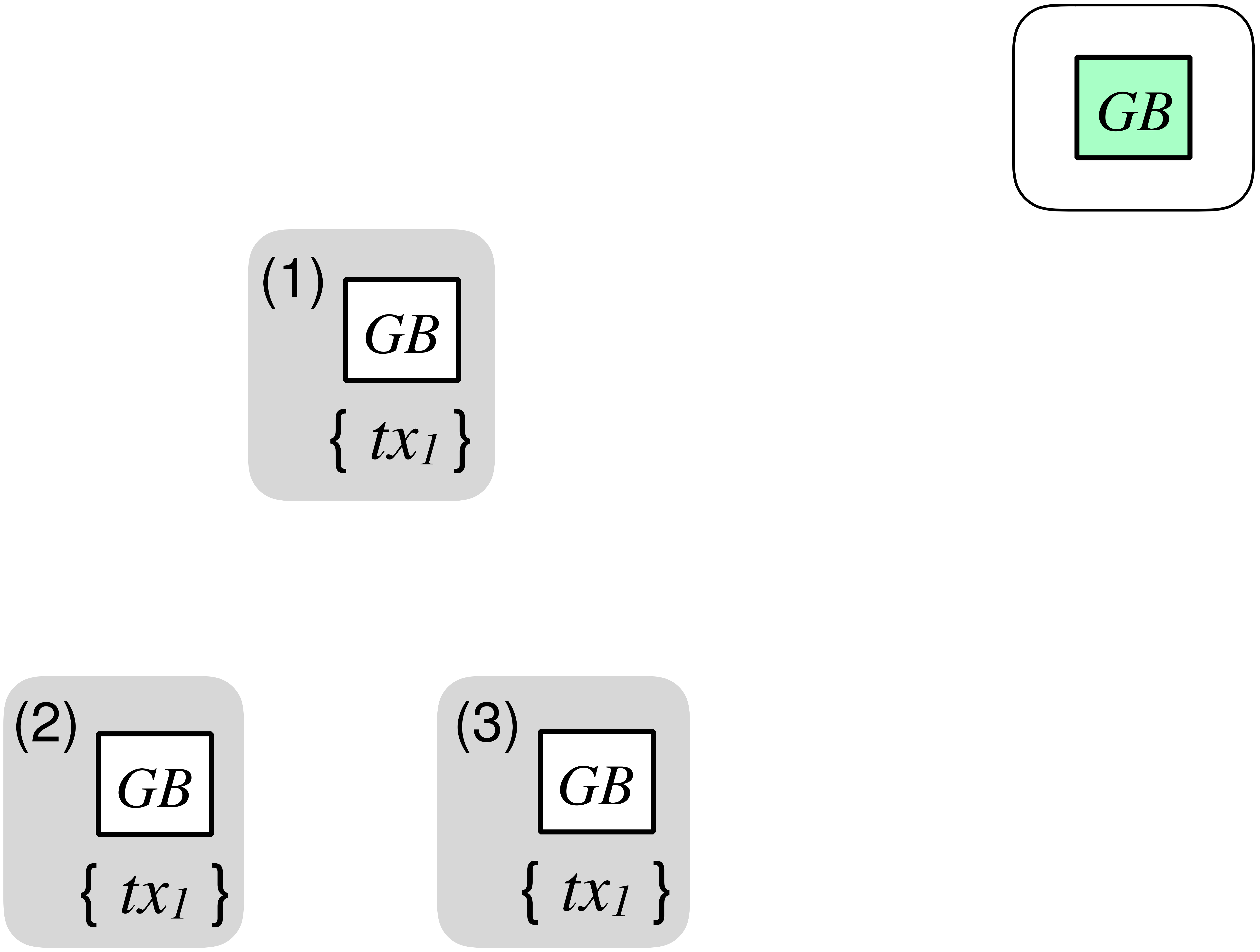
- **distributed**
  - multiple nodes
  - message-passing over a network

- all start with same GB



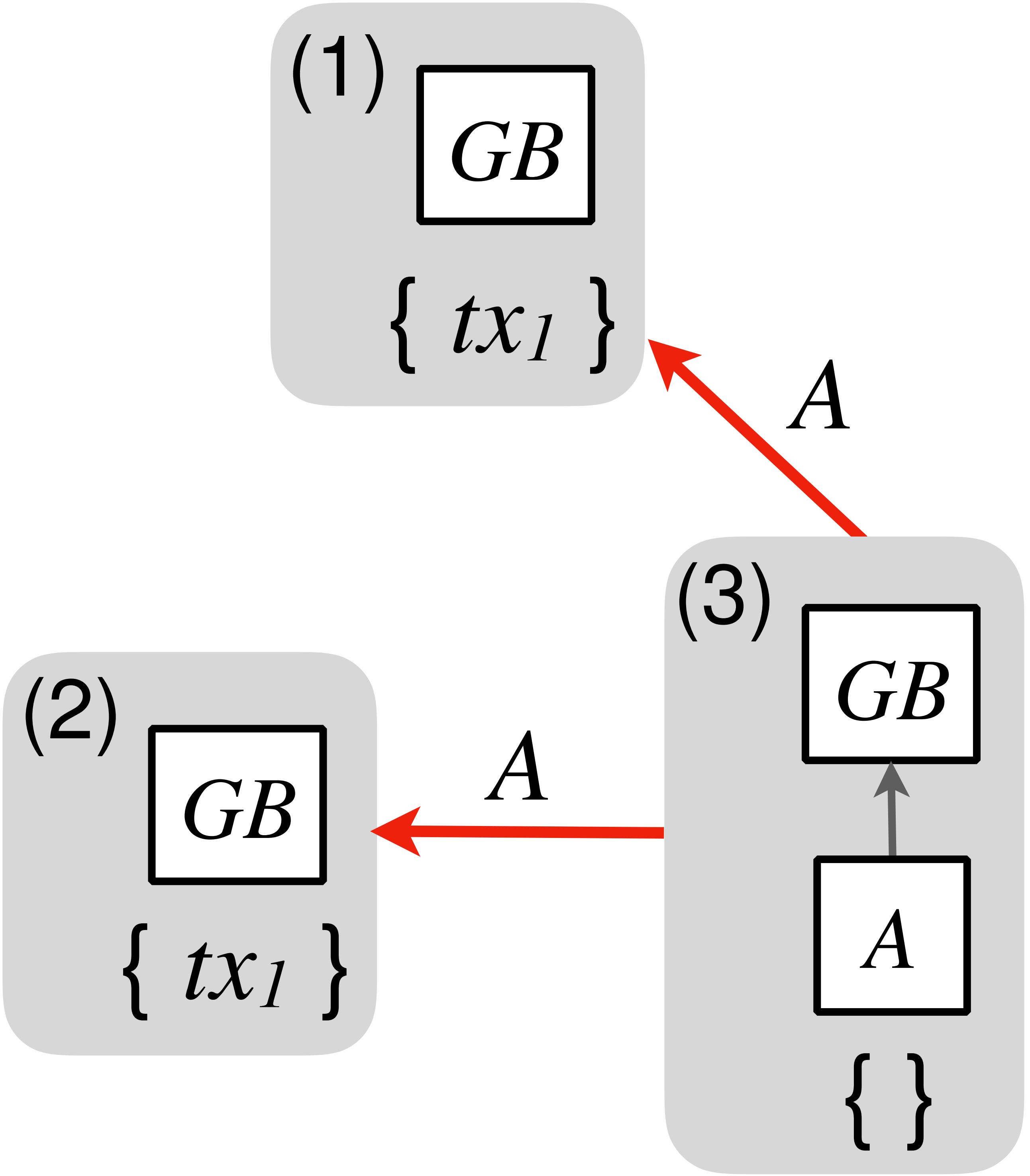
- **distributed**
  - multiple nodes
  - message-passing over a network

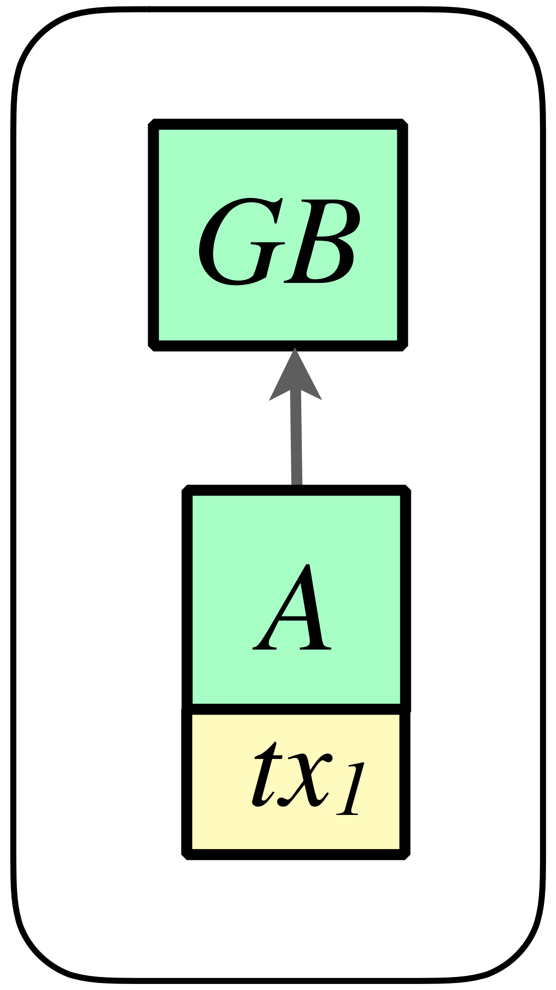
- all start with same GB
- have a transaction pool



- **distributed**
  - multiple nodes
  - message-passing over a network

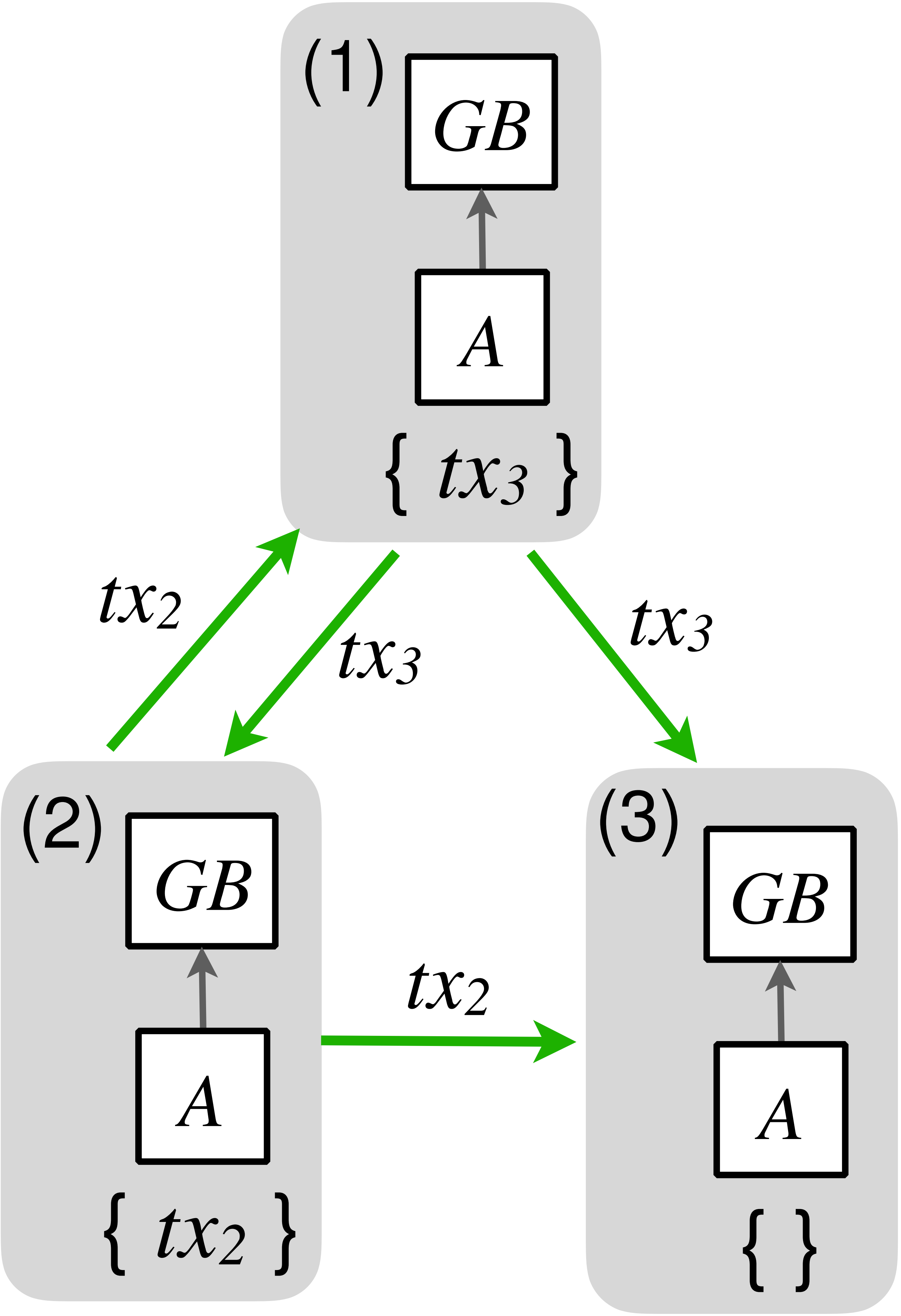
- all start with same GB
- have a transaction pool
- can <u>mint blocks</u>

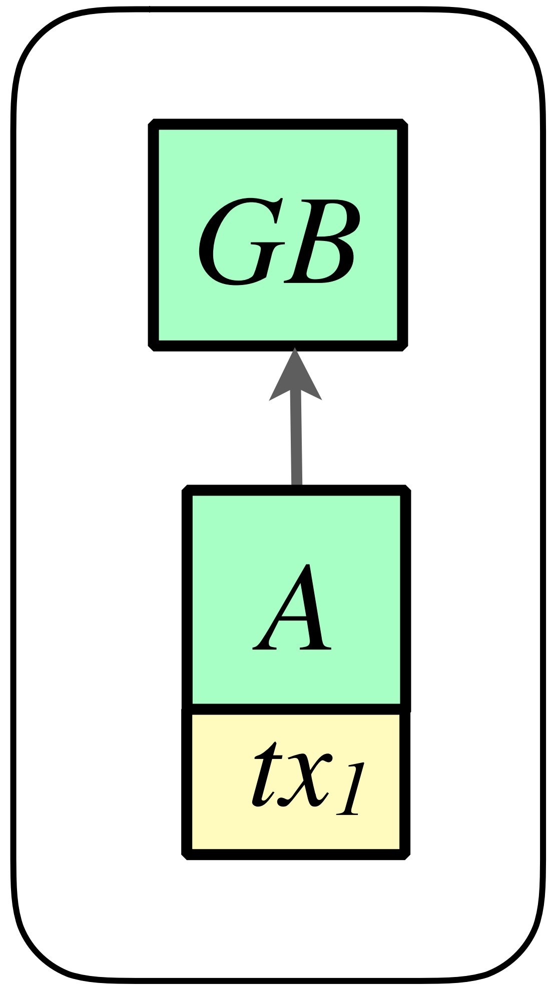




- **distributed** => <u>concurrent</u>
  - multiple nodes
  - message-passing over a network

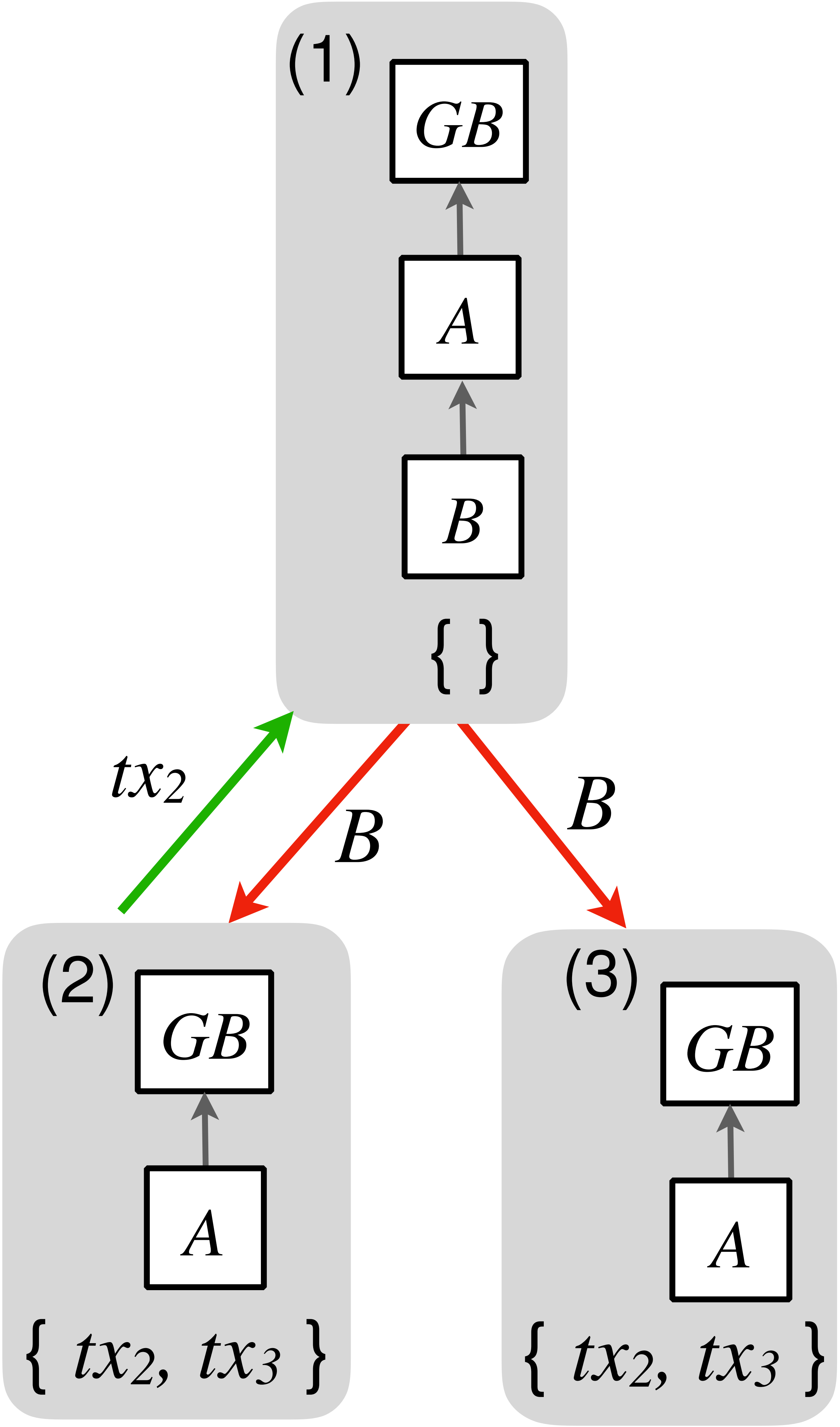
- multiple transactions can be issued and propagated concurrently

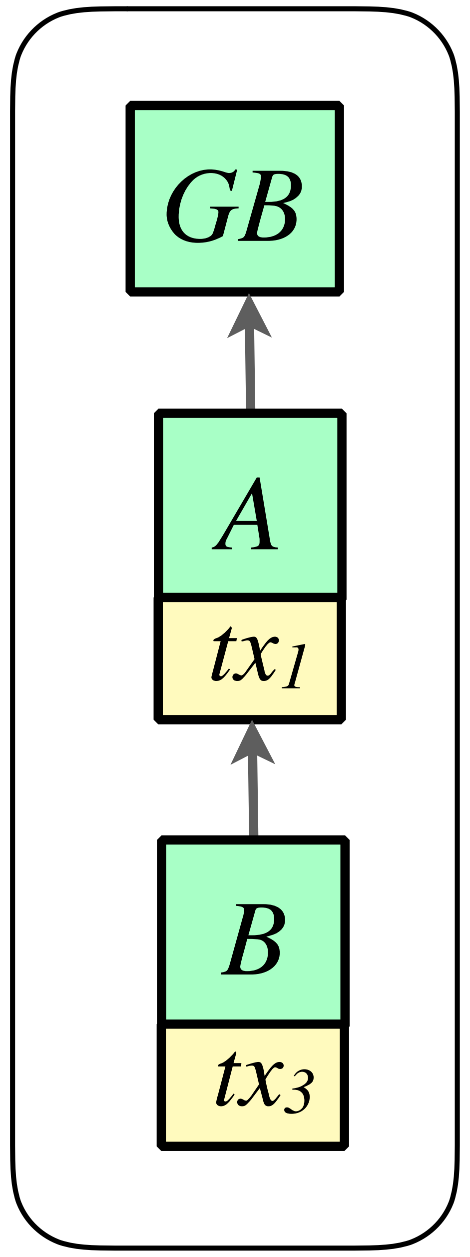




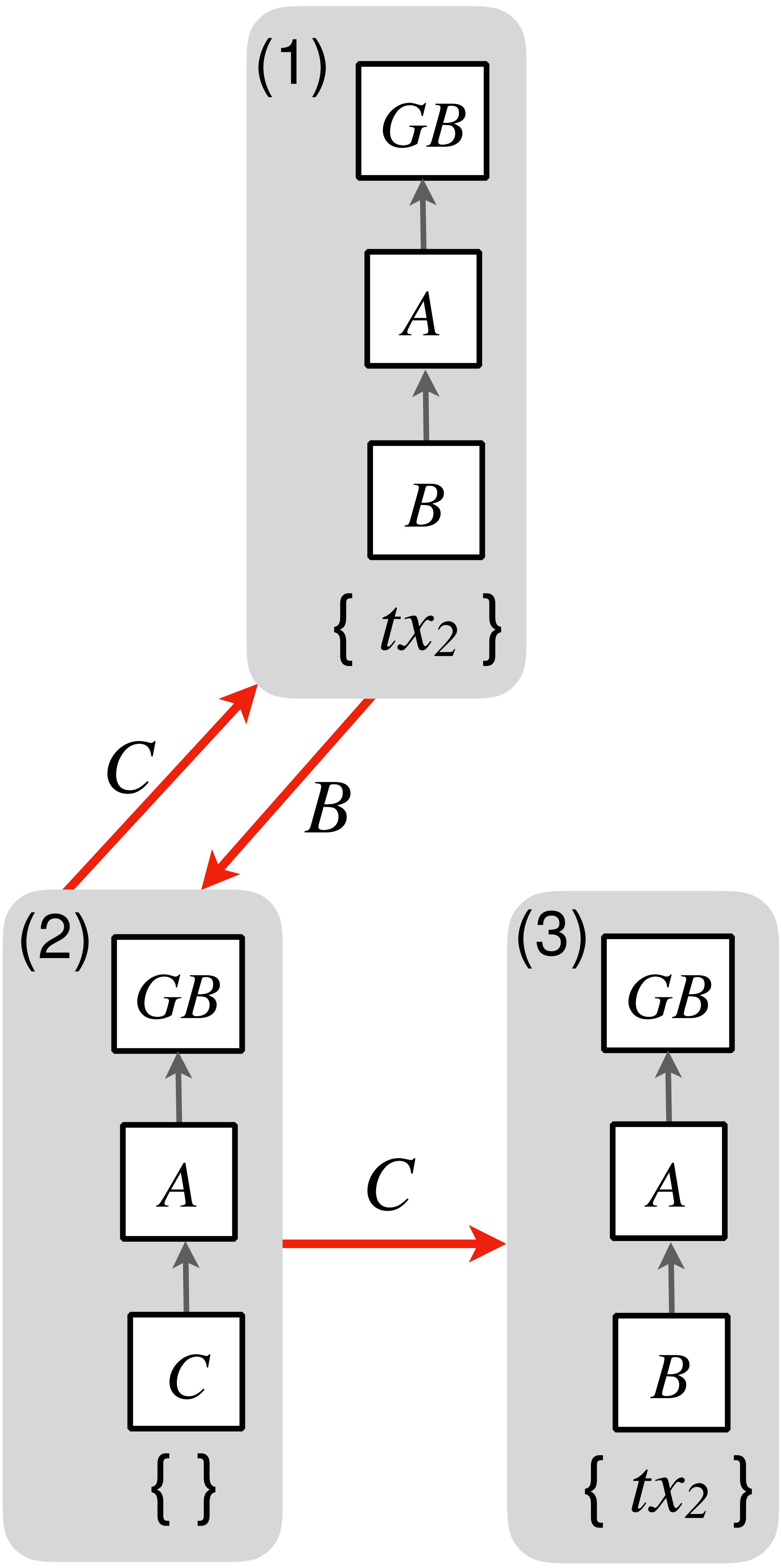
- **distributed** => <u>concurrent</u>
  - multiple nodes
  - message-passing over a network

- blocks can be minted without full knowledge of all transactions

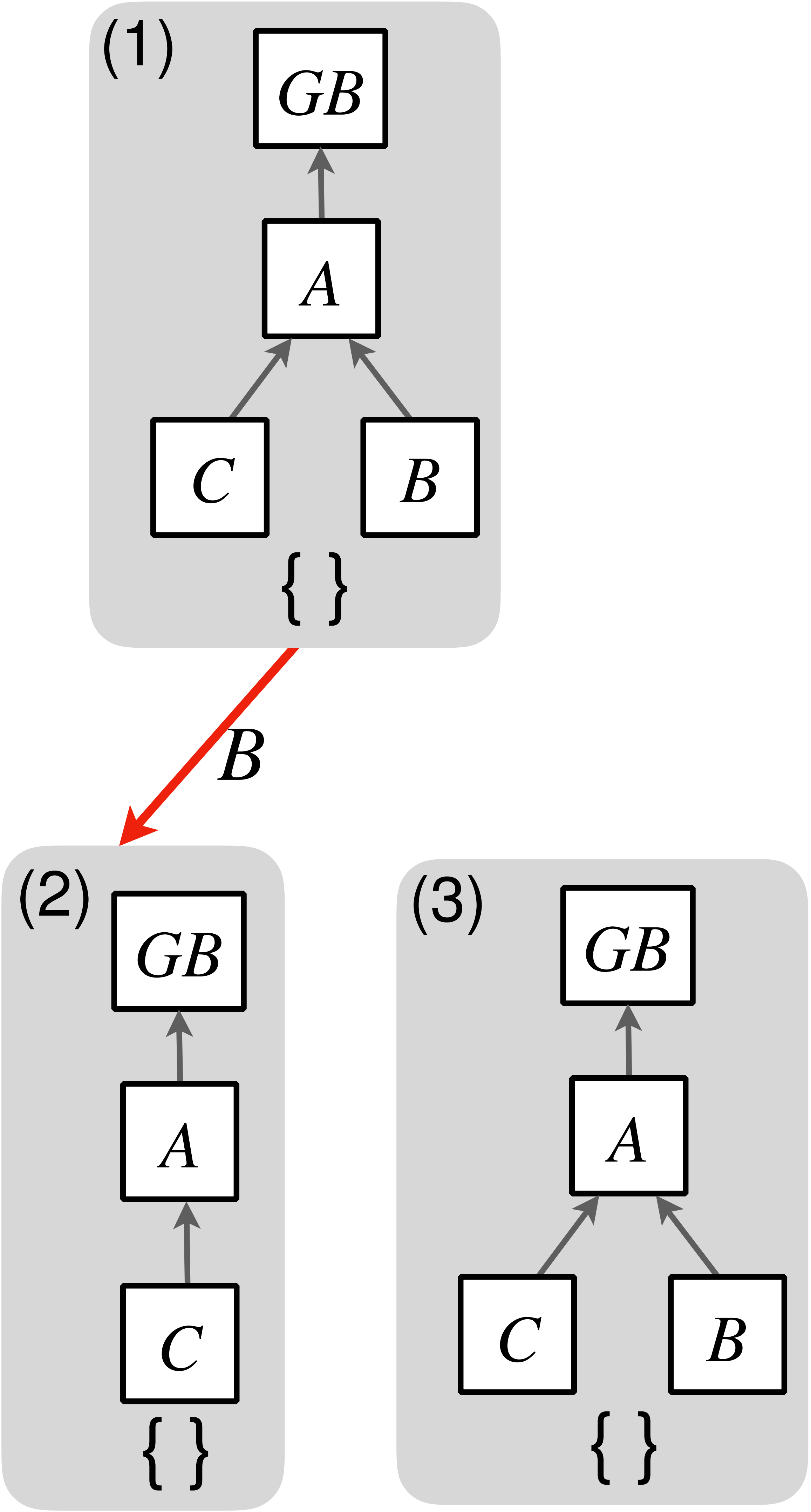




- chain fork has happened, but nodes don’t know



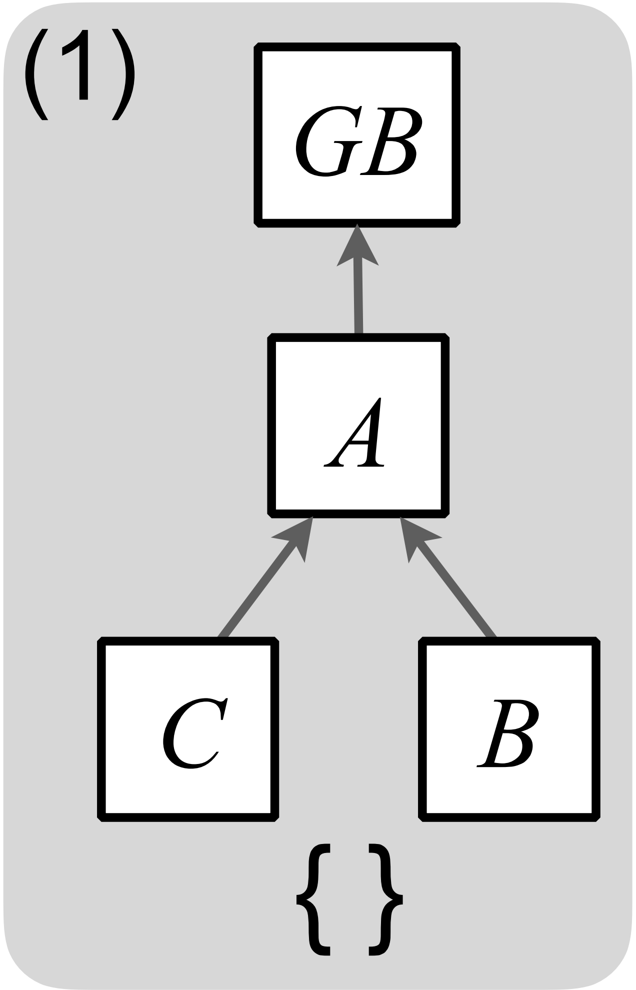
- as block messages propagate, nodes become aware of the <u>fork</u>

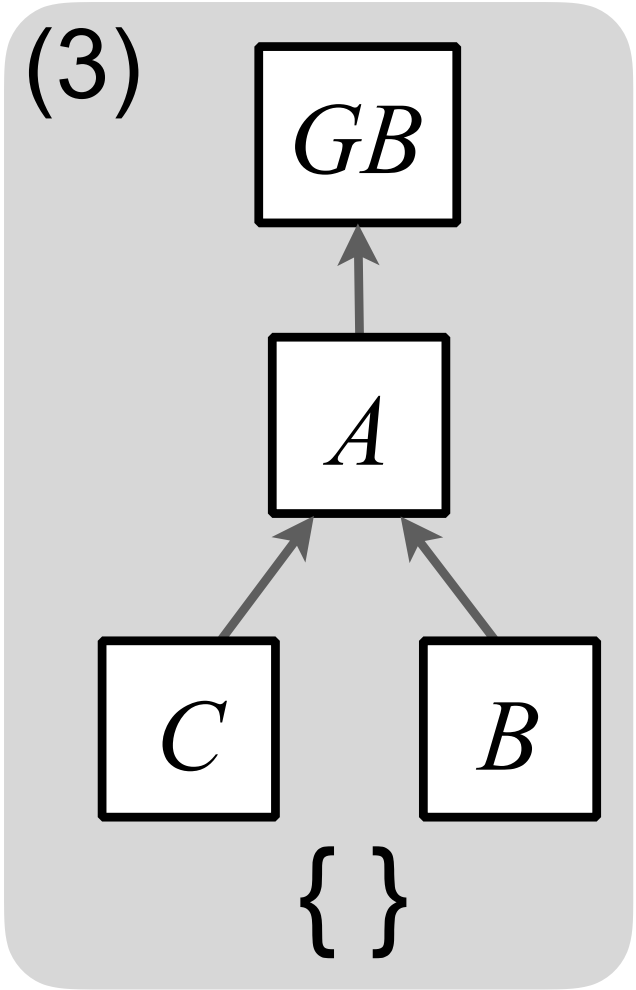


Problem: need to choose

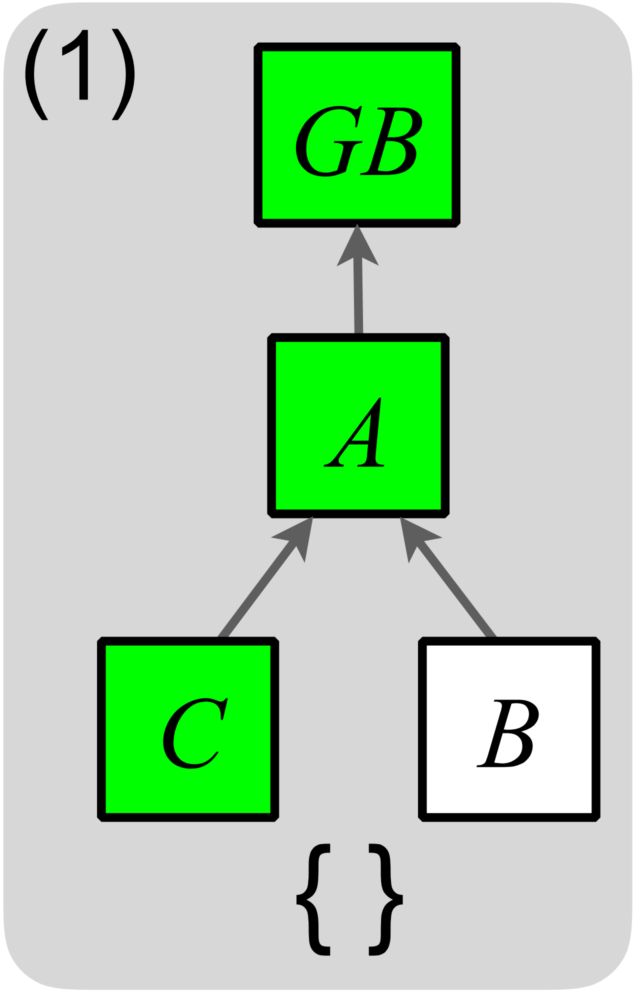
- blockchain "promise" =
  one globally-agreed chain

  - each node must choose <u>one</u> chain
  - nodes with the same information
    must choose <u>the same</u> chain

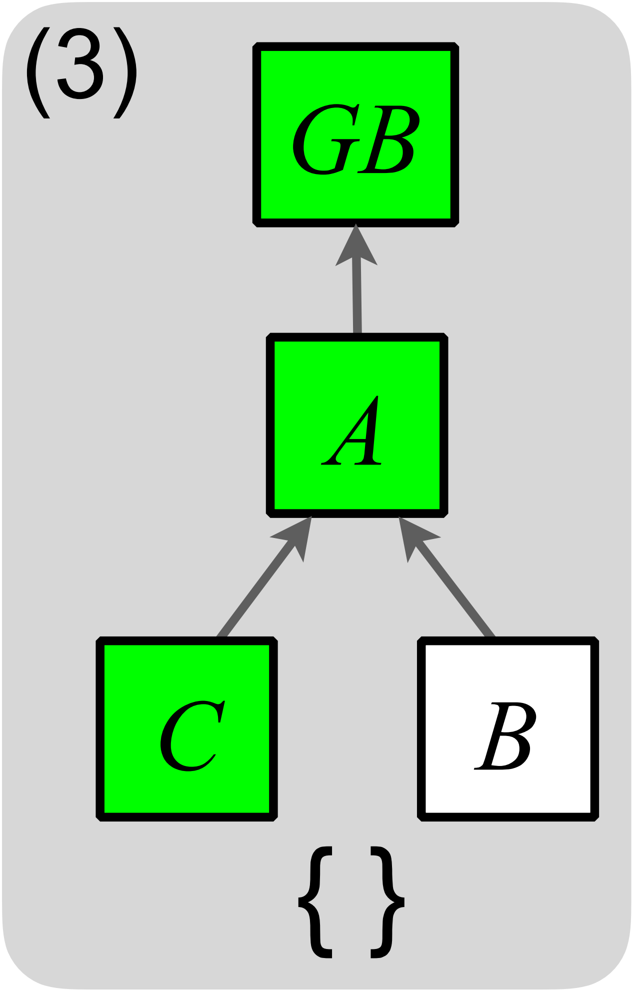




Problem: need to choose

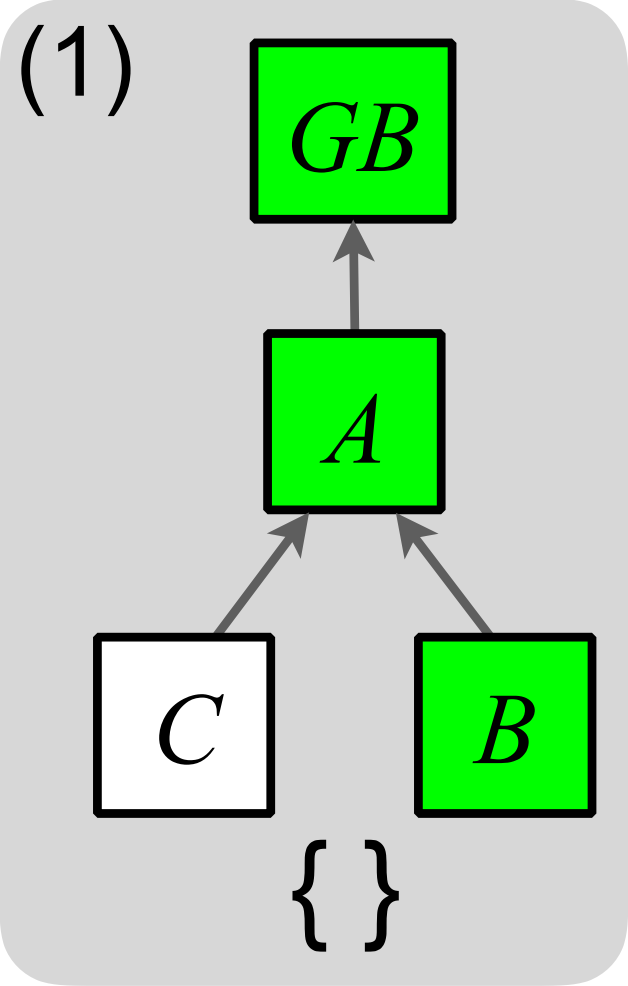


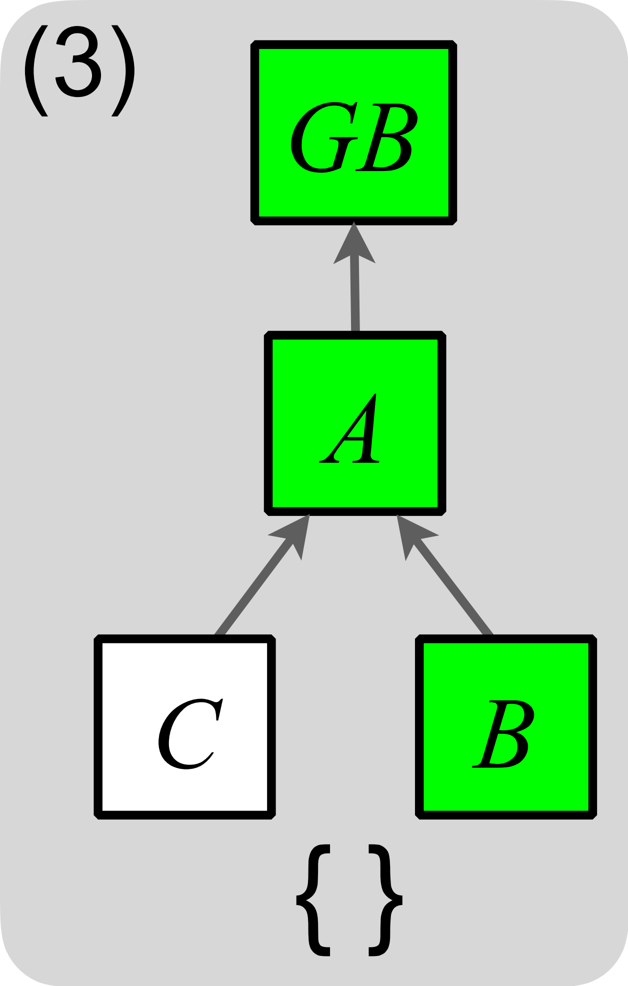
- blockchain "promise" =
  one globally-agreed chain
  - each node must choose <u>one</u> chain
  - nodes with the same information must choose <u>the same</u> chain



Problem: need to choose

- blockchain "promise" = *one globally-agreed chain*
  - each node must choose one chain
  - nodes with the same information must choose the same chain

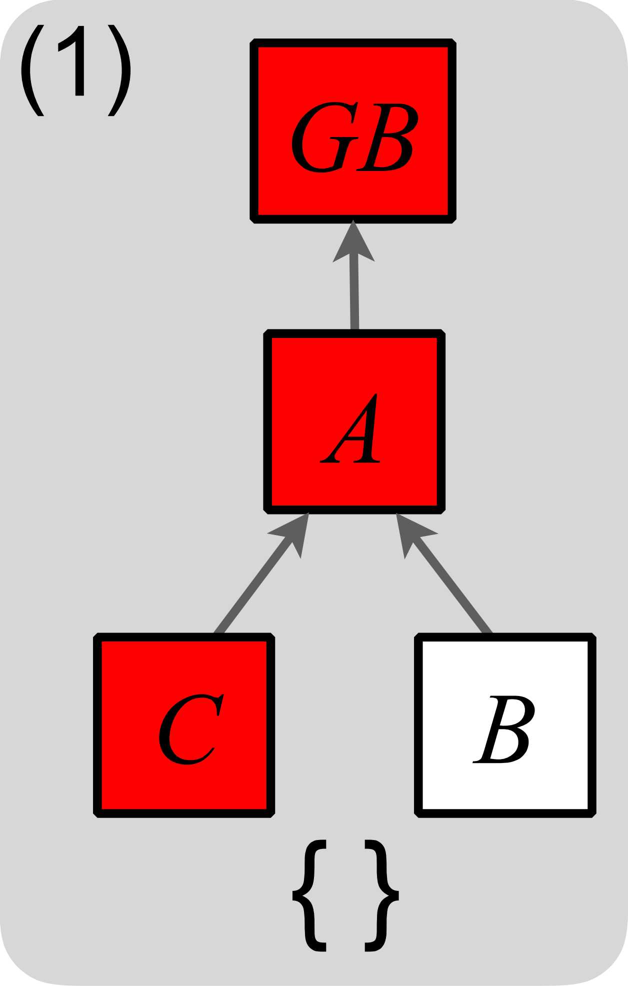


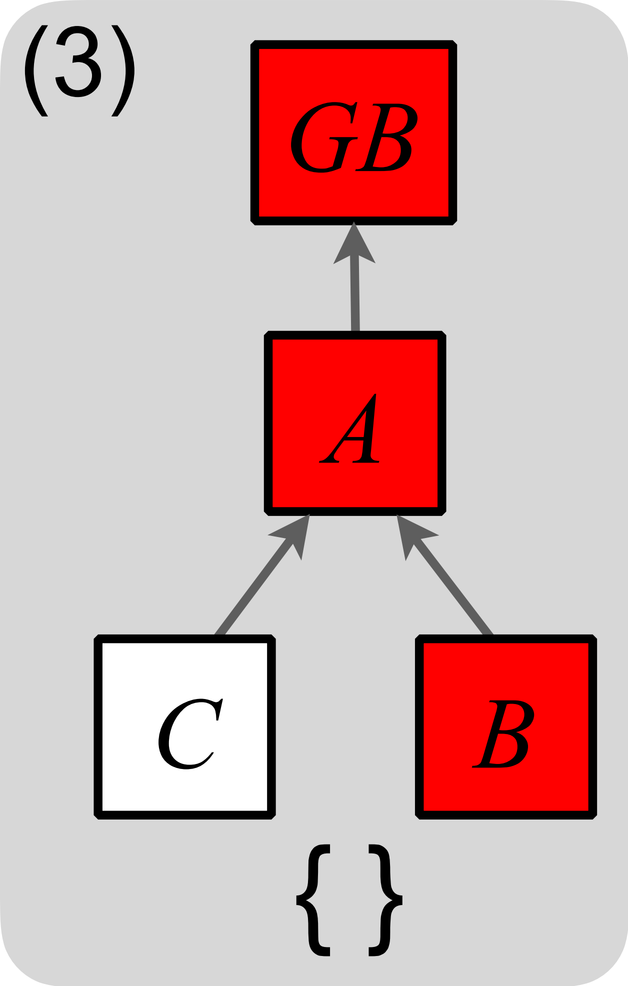


Problem: need to choose

- blockchain "promise" = *one globally-agreed chain*

  - each node must choose <u>one</u> chain
  - nodes with the same information must choose <u>the same</u> chain





Solution: fork choice rule

- Fork choice rule (FCR, >):
    - given two blockchains, says which one is “heavier”
    - imposes a *strict total order* on all possible blockchains
    - same FCR shared by all nodes

- Nodes adopt “heaviest” chain they know

FCR (>)

… > [GB, A, C] > … > [GB, A, B] > … > [GB, A] > … > [GB] > …

Bitcoin: FCR based on “most cumulative work”

A (very basic) Blockchain Protocol Specification:

Eventual Consistency

Specification:

For *any sequence* **sc** of message exchanges in a *closed* network **N**, once *all messages are delivered* with the *corresponding effects*, any two *non-byzantine* nodes n_1 and n_2 share *the same chain*.

or, equivalently

Eventual Consistency Theorem:

$$\forall\, sc\ \forall\, n_1, n_2 \in \mathbf{N},$$
$$run_{BC}(\mathbf{N}, sc).n_1.chain = run_{BC}(\mathbf{N}, sc).n_2.chain$$

Eventual Consistency Theorem:

$$\forall\ sc\ \forall\ n_1, n_2 \in \mathbf{N},$$

$$run_{BC}(\mathbf{N}, sc).n_1.chain = run_{BC}(\mathbf{N}, sc).n_2.chain$$

Proof: ???

Assumptions:

- Rigorous definition of per-node *protocol state machine*
- Rigorous definition of the *network semantics*
- Precise *fault model* (e.g., nodes are non-byzantine)
- The implementation is *correctly compiled*
- The *network/OS software is reliable*

must be trusted
(*i.e.*, better be "sane")

Eventual Consistency Theorem:

$\forall\, sc\ \forall\, n_1, n_2 \in \mathbf{N},$

$\quad run_{BC}(\mathbf{N}, sc).n_1.chain = run_{BC}(\mathbf{N}, sc).n_2.chain$

Proof: ???

once proven,
does not have
to be trusted

Formal Verification

Proving correctness of algorithms or software artefacts

with respect to a given rigorous specification

using mathematical reasoning.

Formal Verification

Proving correctness of algorithms or software artefacts

with respect to a given rigorous specification

using mathematical reasoning.

What is a Proof?

A proof is sufficient evidence
or an argument for the truth of a proposition.

Better Definition

A proof is a *sequence of logical statements*,

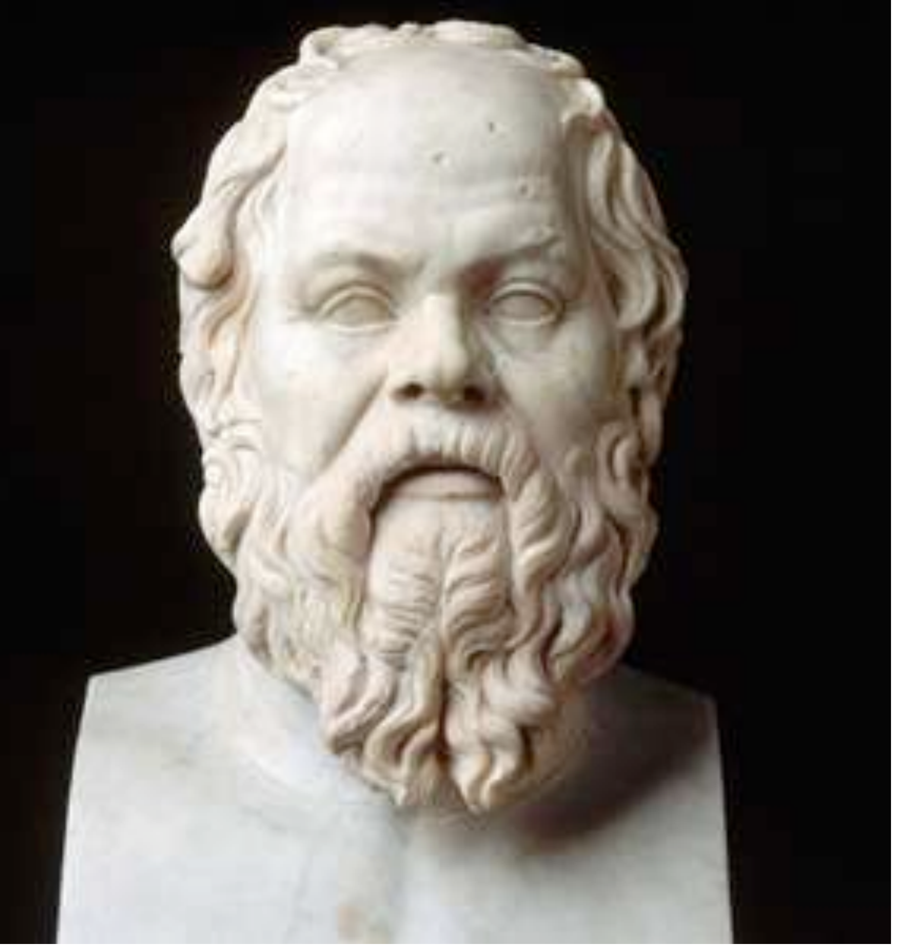
each of which is either *validly derived from those preceding* it
or is an *assumption*,

and the final member of which,
the conclusion, is the statement
of which the truth is thereby established.

Deriving Valid Proofs

The proposition A is true, and, moreover, A being true implies that B is true; then we can derive that B is true.

$$\frac{\vdash A \qquad \vdash A \Rightarrow B}{\vdash B}$$



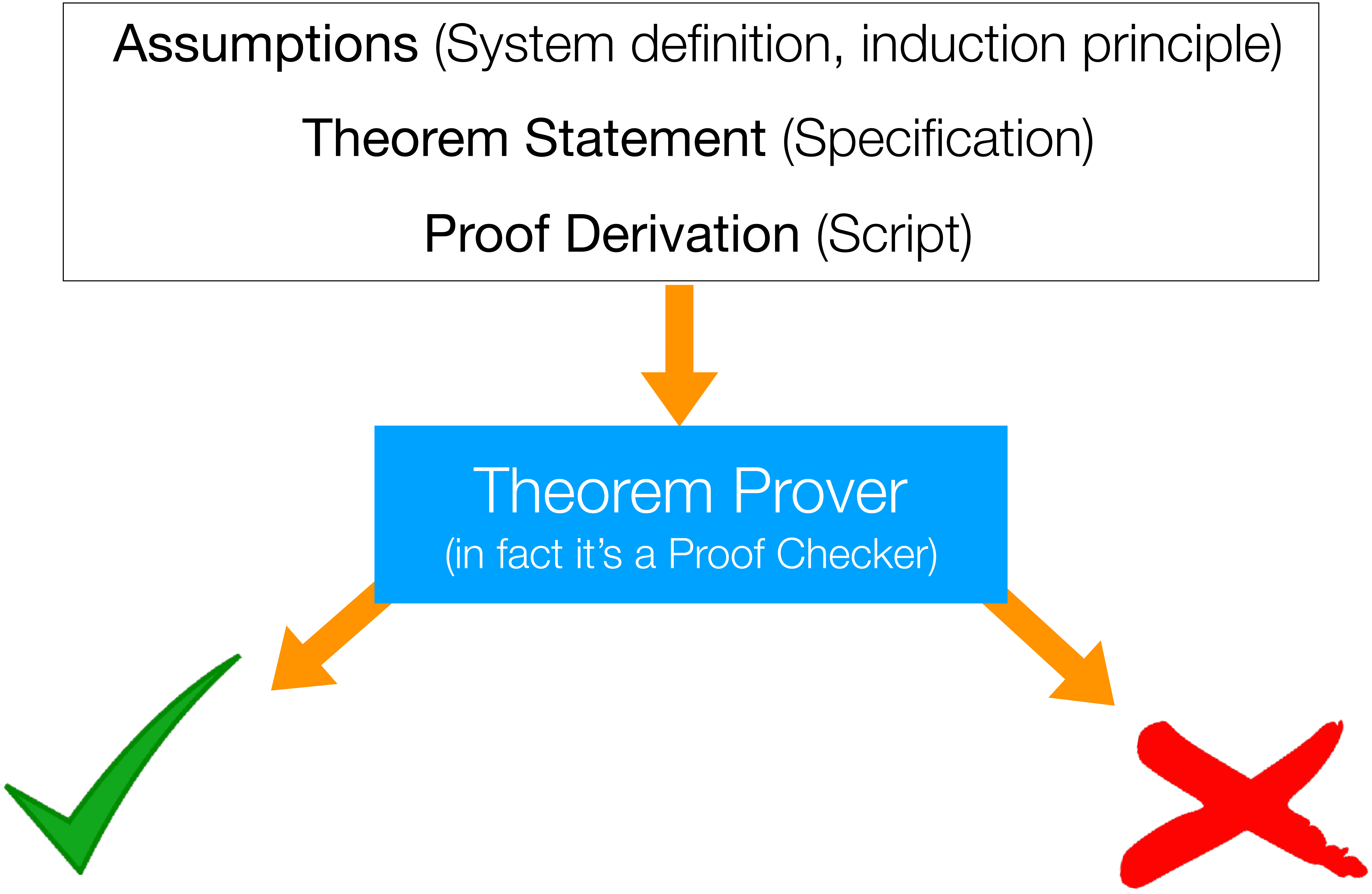
$$\frac{\vdash A \qquad \vdash A \Rightarrow B}{\vdash B}$$

reasonable assumptions

$$\frac{\boxed{\text{Socrates is a man}} \qquad \boxed{\text{is a man} \Rightarrow \text{is mortal}}}{\text{Socrates is mortal}}$$

Even large proofs, when rigorously written, can be ***checked automatically***!

Proofs don’t have to be trusted!



Formal Verification

Proving correctness of algorithms or software artefacts

with respect to a given rigorous specification

using mathematical reasoning.

Mechanised Formal Verification

Proving correctness of algorithms or software artefacts

with respect to a given rigorous specification

using mathematical reasoning,

whose validity is *machine-checked*.

(assuming that you trust the checker)

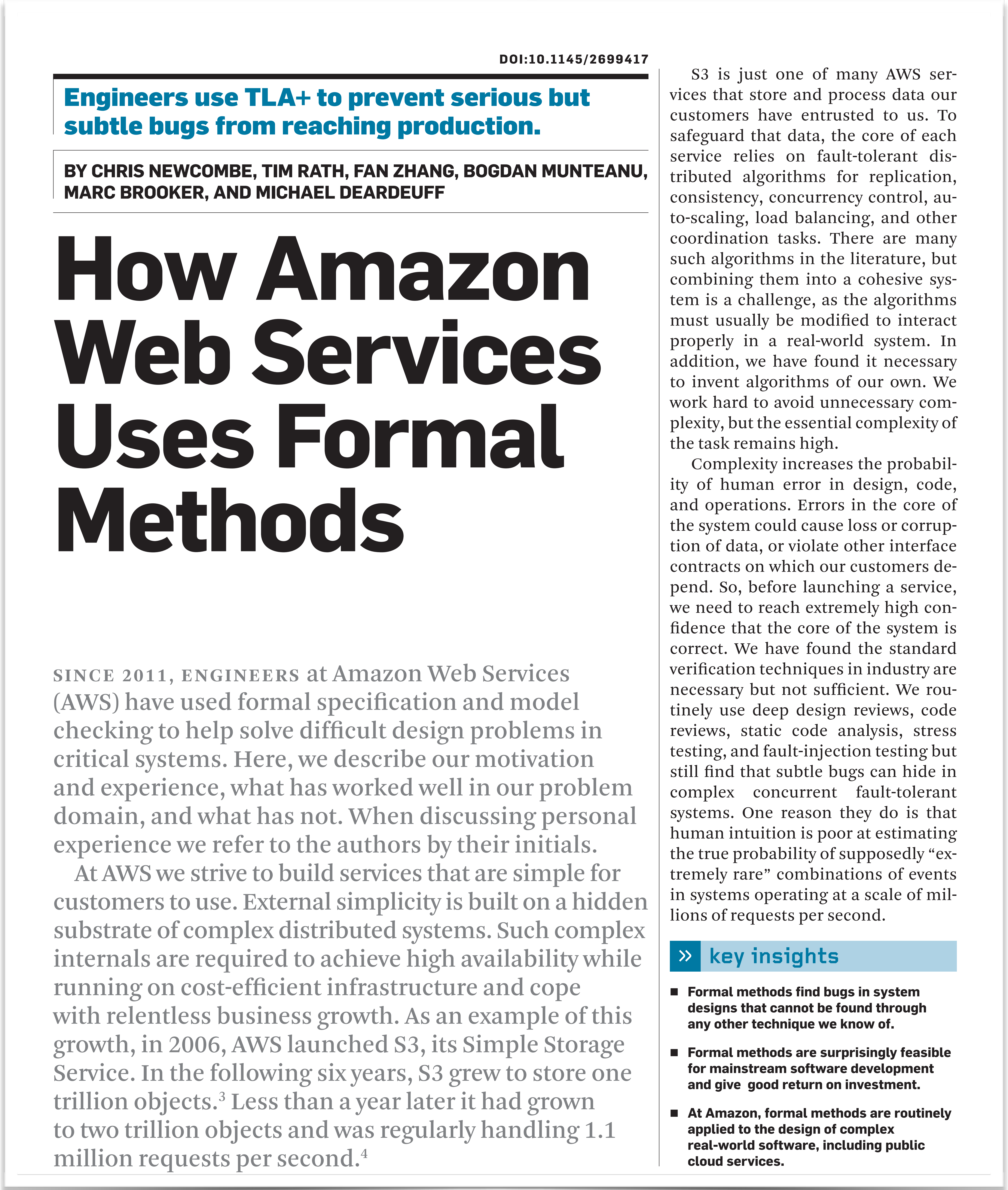
Checkpoint

- For a fully specified system, correctness is a *mathematical theorem*
- It can be proven using rules of *mathematical logic*
- The proofs rest on reasonable assumptions, which must be *trusted*
- *Mechanised Proof Checking* ensures validity of the proof,
  but requires to *trust the checker implementation.*

Mechanised Proof Checking for Distributed Systems

```
Inv1 == \A p \in P : \A c \in C : LastBal(c, Max(Ballot), p) \preceq ballot[p][c]

Inv3 == \A p \in P : \A c \in C :
    LET b == LastBal(c, Max(Ballot), p)
    IN  <<-1,-1>> \prec b => b \in DOMAIN vote[p][c]

Inv2 == \A prop \in Image(propose) : prop.strong \subseteq prop.weak

\* in sub-round 1, the set of weak dependencies is always empty.
Inv4 == \A x \in DOMAIN propose : x[2][2] = 1 => propose[x].weak = {}
```

https://github.com/tlaplus/Examples



DOI:10.1145/2699417

Engineers use TLA+ to prevent serious but subtle bugs from reaching production.

BY CHRIS NEWCOMBE, TIM RATH, FAN ZHANG, BOGDAN MUNTEANU, MARC BROOKER, AND MICHAEL DEARDEUFF

How Amazon Web Services Uses Formal Methods

SINCE 2011, ENGINEERS at Amazon Web Services (AWS) have used formal specification and model checking to help solve difficult design problems in critical systems. Here, we describe our motivation and experience, what has worked well in our problem domain, and what has not. When discussing personal experience we refer to the authors by their initials.

At AWS we strive to build services that are simple for customers to use. External simplicity is built on a hidden substrate of complex distributed systems. Such complex internals are required to achieve high availability while running on cost-efficient infrastructure and cope with relentless business growth. As an example of this growth, in 2006, AWS launched S3, its Simple Storage Service. In the following six years, S3 grew to store one trillion objects.[3] Less than a year later it had grown to two trillion objects and was regularly handling 1.1 million requests per second.[4]

S3 is just one of many AWS services that store and process data our customers have entrusted to us. To safeguard that data, the core of each service relies on fault-tolerant distributed algorithms for replication, consistency, concurrency control, auto-scaling, load balancing, and other coordination tasks. There are many such algorithms in the literature, but combining them into a cohesive system is a challenge, as the algorithms must usually be modified to interact properly in a real-world system. In addition, we have found it necessary to invent algorithms of our own. We work hard to avoid unnecessary complexity, but the essential complexity of the task remains high.

Complexity increases the probability of human error in design, code, and operations. Errors in the core of the system could cause loss or corruption of data, or violate other interface contracts on which our customers depend. So, before launching a service, we need to reach extremely high confidence that the core of the system is correct. We have found the standard verification techniques in industry are necessary but not sufficient. We routinely use deep design reviews, code reviews, static code analysis, stress testing, and fault-injection testing but still find that subtle bugs can hide in complex concurrent fault-tolerant systems. One reason they do is that human intuition is poor at estimating the true probability of supposedly "extremely rare" combinations of events in systems operating at a scale of millions of requests per second.

» key insights

- Formal methods find bugs in system designs that cannot be found through any other technique we know of.
- Formal methods are surprisingly feasible for mainstream software development and give good return on investment.
- At Amazon, formal methods are routinely applied to the design of complex real-world software, including public cloud services.

Proving Blockchain Specification

Assumptions:

- Rigorous definition of per-node *protocol state machine*
- Rigorous definition of the *network semantics*
- Precise *fault model* (e.g., nodes are non-byzantine)
- The implementation is *correctly compiled*
- The *network/OS software is reliable*

Eventual Consistency Theorem:

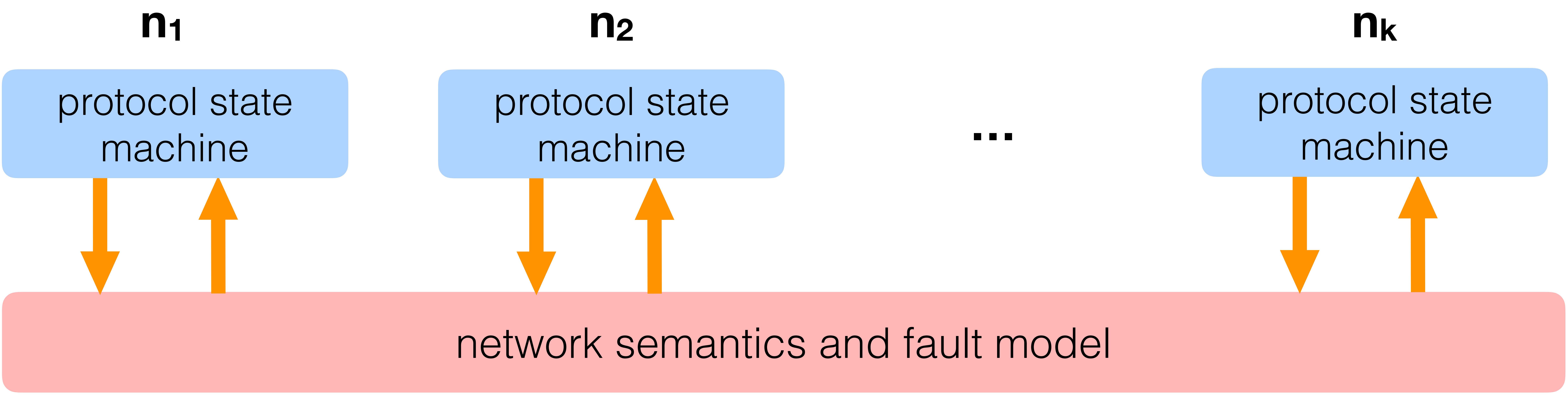
$$\forall\, sc\ \forall\, n_1, n_2 \in \mathbf{N},$$
$$\mathit{run}_{BC}(\mathbf{N}, sc).n_1.\mathit{chain} = \mathit{run}_{BC}(\mathbf{N}, sc).n_2.\mathit{chain}$$

Assumptions:

- Rigorous definition of per-node *protocol state machine*
- Rigorous definition of the *network semantics*
- Precise *fault model* (e.g., nodes are non-byzantine)
- The implementation is *correctly compiled*
- The *network/OS software is reliable*

Eventual Consistency Theorem:

$$\forall\, sc\ \forall\, n_1, n_2 \in \mathbf{N},$$
$$run_{BC}(\mathbf{N}, sc).n_1.chain = run_{BC}(\mathbf{N}, sc).n_2.chain$$



n_1
n_2
n_k
protocol state machine
protocol state machine
...
protocol state machine
network semantics and fault model

protocol state machine

Internal step transitions: $\delta \xrightarrow{\langle i, \tau \rangle}_{\iota} (\delta', ps)$

$$\textsc{IntTx}\quad \frac{ps = \{\langle \mathsf{this}, a, \mathsf{TxMsg}\ tx\rangle \mid a \in as\}}{\langle \mathsf{this}, as, bf, tp\rangle \xrightarrow{\langle \mathsf{DoTx}\ tx, \tau\rangle}_{\iota} (\langle \mathsf{this}, as, bf, tp\rangle, ps)}$$

$$\textsc{IntMint}\quad \frac{\begin{array}{c} \mathit{mkProof}\ \mathsf{this}\ \lceil bf \rceil = \mathsf{Some}\ pf \qquad \mathit{VAF}\ pf\ \tau\ \lceil bf \rceil = \mathsf{true} \\ b = \left\{ \begin{array}{lcl} \mathtt{prev} & := & \#(\mathit{last}\ \lceil bf \rceil); \\ \mathtt{txs} & := & [\ tx \mid tx \in tp \land \mathit{txValid}\ t\ \lceil bf \rceil\ ]; \\ \mathtt{pf} & := & pf \end{array} \right\} \\ bf' = bf \triangleleft b \qquad ps = \{\langle \mathsf{this}, a, \mathsf{BlockMsg}\ b\rangle \mid a \in as\} \\ tp' = \left\{ tx \mid tx \in tp \land \mathit{txValid}\ tx\ \lceil bf' \rceil \right\} \setminus (\mathtt{txs}\ b) \end{array}}{\langle \mathsf{this}, as, bf, tp\rangle \xrightarrow{\langle \mathsf{DoMint}, \tau\rangle}_{\iota} (\langle \mathsf{this}, as, bf', tp'\rangle, ps)}$$

Receive-step transitions: $\delta \xrightarrow{p}_{\rho} (\delta', ps)$

$$\textsc{RcvNull}\qquad \delta \xrightarrow{\langle \mathsf{from}, \mathsf{this}, \mathsf{NullMsg}\rangle}_{\rho} (\delta, \emptyset)$$

$$\textsc{RcvConnect}\quad \frac{\begin{array}{c} as' = as \cup \{\mathsf{from}\} \qquad hs = \mathsf{dom}(bf) \cup \{\#tx \mid tx \in tp\} \\ ps = \{\langle \mathsf{this}, \mathsf{from}, \mathsf{InvMsg}\ hs\rangle\} \end{array}}{\langle \mathsf{this}, as, bf, tp\rangle \xrightarrow{\langle \mathsf{from}, \mathsf{this}, \mathsf{ConnectMsg}\rangle}_{\rho} (\langle \mathsf{this}, as', bf, tp\rangle, ps)}$$

$$\textsc{RcvAddr}\quad \frac{\begin{array}{c} as_1 = \left\{a \mid a \in as' \land a \notin as\right\} \qquad as_2 = as \cup as_1 \\ ps_1 = \{\langle \mathsf{this}, a, \mathsf{ConnectMsg}\rangle \mid a \in as_1\} \\ ps_2 = \{\langle \mathsf{this}, a, \mathsf{AddrMsg}\ as_2\rangle \mid a \in as\} \qquad ps = ps_1 \cup ps_2 \end{array}}{\langle \mathsf{this}, as, bf, tp\rangle \xrightarrow{\langle \mathsf{from}, \mathsf{this}, \mathsf{AddrMsg}\ as'\rangle}_{\rho} (\langle \mathsf{this}, as_2, bf, tp\rangle, ps)}$$

$$\textsc{RcvTx}\quad \frac{\begin{array}{c} tp' = \mathit{txExtend}\ tp\ tx \qquad hs = \mathsf{dom}(bf) \cup \left\{\#tx' \mid tx' \in tp'\right\} \\ ps = \{\langle \mathsf{this}, a, \mathsf{InvMsg}\ hs\rangle \mid a \in as\} \end{array}}{\langle \mathsf{this}, as, bf, tp\rangle \xrightarrow{\langle \mathsf{from}, \mathsf{this}, \mathsf{TxMsg}\ tx\rangle}_{\rho} (\langle \mathsf{this}, as, bf, tp'\rangle, ps)}$$

$$\textsc{RcvBlock}\quad \frac{\begin{array}{c} bf' = bf \triangleleft b \qquad tp' = \left\{tx \mid tx \in tp \land \mathit{txValid}\ tx\ \lceil bf' \rceil\right\} \\ hs = \mathsf{dom}(bf') \cup \left\{\#tx \mid tx \in tp'\right\} \\ ps = \{\langle \mathsf{this}, a, \mathsf{InvMsg}\ hs\rangle \mid a \in as\} \end{array}}{\langle \mathsf{this}, as, bf, tp\rangle \xrightarrow{\langle \mathsf{from}, \mathsf{this}, \mathsf{BlockMsg}\ b\rangle}_{\rho} (\langle \mathsf{this}, as, bf', tp'\rangle, ps)}$$

+

network semantics and fault model

Network transitions: $\langle \Delta, P\rangle \overset{s}{\Longrightarrow} \langle \Delta', P'\rangle$

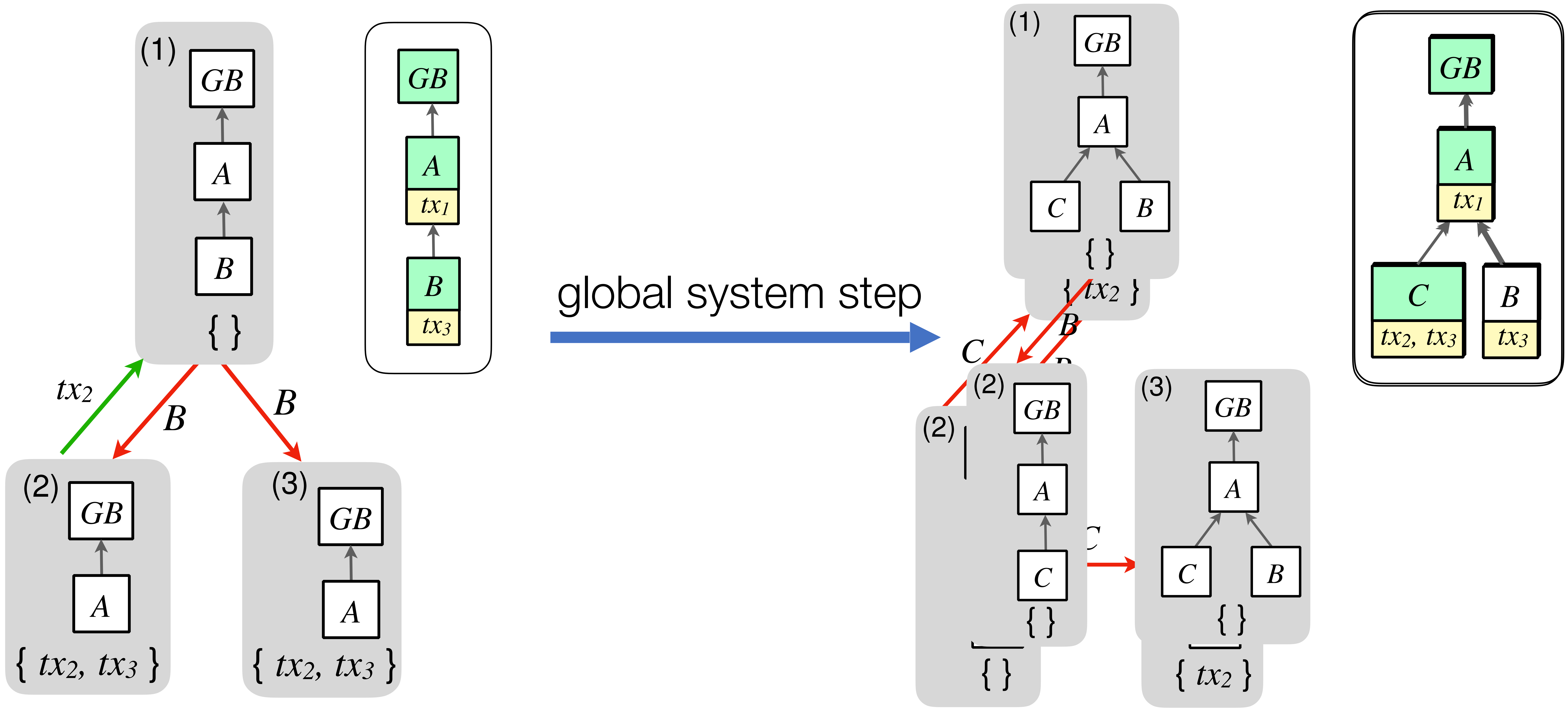
$$\textsc{NetDeliver}\quad \frac{p \in P \qquad \mathit{dest}\ p = a \qquad \Delta(a) = \delta \qquad \delta \xrightarrow{p}_{\rho} (\delta', ps)}{\langle \Delta, P\rangle \overset{\mathsf{SelRcv}\ a}{\Longrightarrow} \langle \Delta[a \mapsto \delta'], P \setminus \{p\} \cup ps\rangle}$$

$$\textsc{NetInternal}\quad \frac{\Delta(a) = \delta \qquad \delta \xrightarrow{\langle i, \tau\rangle}_{\iota} (\delta', ps)}{\langle \Delta, P\rangle \overset{\mathsf{SelInt}\ a\ \tau\ i}{\Longrightarrow} \langle \Delta[a \mapsto \delta'], P \cup ps\rangle}$$

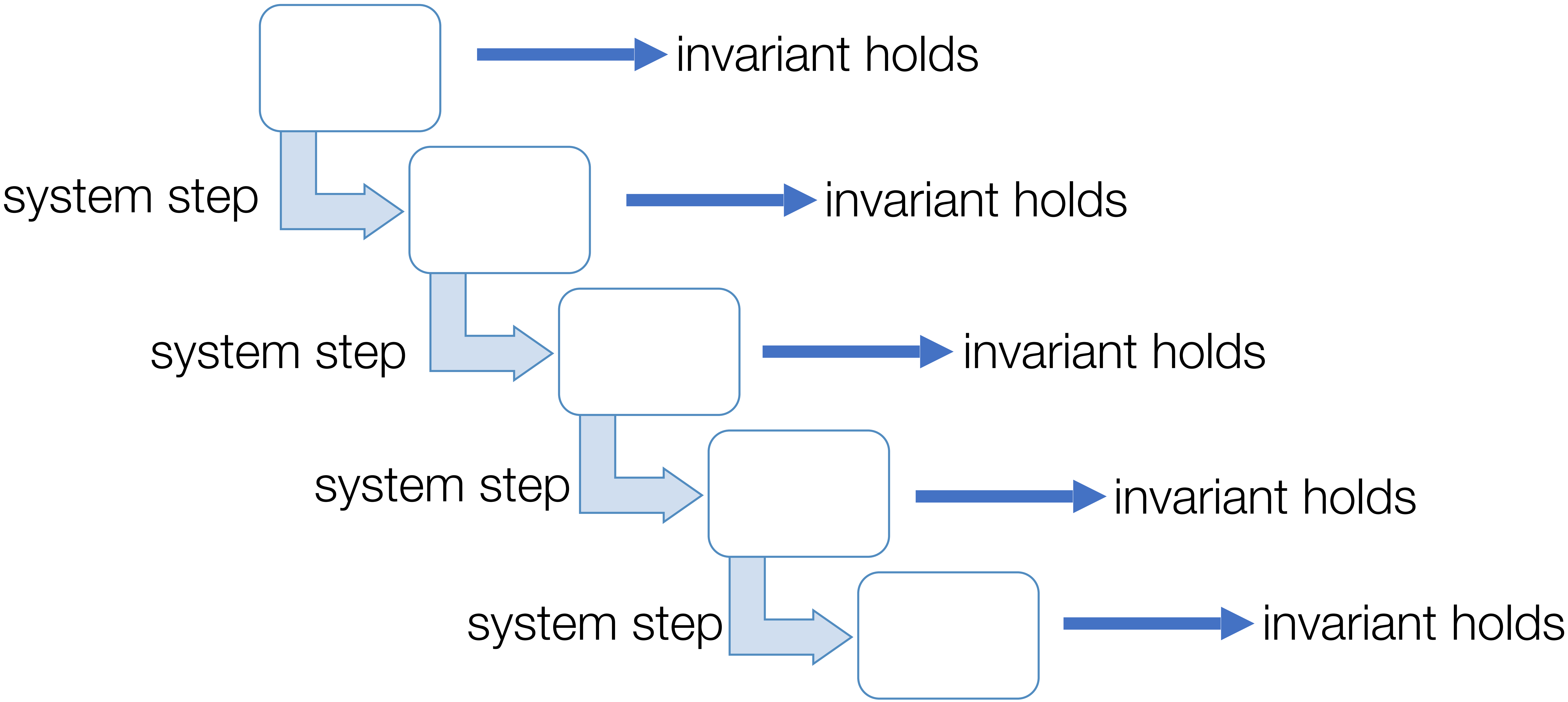
$$\textsc{NetIdle}\qquad \langle \Delta, P\rangle \overset{\mathsf{SelIdl}}{\Longrightarrow} \langle \Delta, P\rangle$$

= global system step

Invariant: local state + “in-flight” = global



Invariant is inductive



Invariant implies Eventual Consistency

- EC: when all blocks delivered, everyone agrees

How:

- local state + “~~in-flight~~” = global
- use FCR to extract “heaviest” chain out of local state

- since everyone has same state & same FCR
  - ➢consensus

Assumptions:

- Rigorous definition of per-node *protocol state machine*
- Rigorous definition of the *network semantics*
- Precise *fault model* (e.g., nodes are non-byzantine)
- The implementation is *correctly compiled*
- The *network/OS software is reliable*

Eventual Consistency Theorem:

$$\forall\, sc\ \forall\, n_1, n_2 \in \mathbf{N},\ \mathit{run}_{BC}(\mathbf{N}, sc).n_1.\mathit{chain} = \mathit{run}_{BC}(\mathbf{N}, sc).n_2.\mathit{chain}$$

Proof: By *induction on the length* of *sc* (a number of system steps).

Temporal Logic of Actions

- *Very effective* for proving invariants via finite-state model checking
- Extensively used for verifying *protocol design*

- TLA+ is a *specification language*, **not** a programming language
- It is a *first-order*, **not** a *higher-order* logic

  for instance, one cannot *quantify over protocol specifications* in TLA+

Theorem Provers that are also programming languages

Isabelle

Coq

Dafny

LEAN
THEOREM PROVER

Frameworks for Verified Distributed Systems Implemented in Proof Assistants

- IronFleet (Paxos), *Hawblitzel et al.*

  IronFleet: Proving Practical Distributed Systems Correct, SOSP'15

- Verdi (Raft), *Wilcox et al.*

  Verdi: A Framework for Implementing and Formally Verifying Distributed Systems, PLDI'15

- Disel (2PC, Paxos), *Sergey et al.*

  Programming and Proving with Distributed Protocols, POPL'18

- Everest (HTTPS, TLS), *Swamy et al.*

  Recalling a Witness: Foundations and Applications of Monotonic State, POPL'19

- Velisarios (PBFT), *Rahli et al.*

  Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq, ESOP'18

Executing Verified Blockchain

protocol state machine

Receive-step transitions: $\delta \xrightarrow{p}_{\rho} (\delta', ps)$

$$\textsc{RcvNull} \qquad \delta \xrightarrow{\langle \text{from, this, NullMsg} \rangle}_{\rho} (\delta, \emptyset)$$

$$\textsc{RcvConnect}\;\frac{\begin{array}{c} as' = as \cup \{\text{from}\} \qquad hs = \text{dom}(bf) \cup \{\#tx \mid tx \in tp\} \\ ps = \{\langle \text{this, from, InvMsg}\ hs \rangle\} \end{array}}{\langle \text{this}, as, bf, tp \rangle \xrightarrow{\langle \text{from, this, ConnectMsg} \rangle}_{\rho} (\langle \text{this}, as', bf, tp \rangle, ps)}$$

$$\textsc{RcvAddr}\;\frac{\begin{array}{c} as_1 = \left\{a \mid a \in as' \land a \notin as\right\} \qquad as_2 = as \cup as_1 \\ ps_1 = \{\langle \text{this}, a, \text{ConnectMsg} \rangle \mid a \in as_1\} \\ ps_2 = \{\langle \text{this}, a, \text{AddrMsg}\ as_2 \rangle \mid a \in as\} \qquad ps = ps_1 \cup ps_2 \end{array}}{\langle \text{this}, as, bf, tp \rangle \xrightarrow{\langle \text{from, this, AddrMsg}\ as' \rangle}_{\rho} (\langle \text{this}, as_2, bf, tp \rangle, ps)}$$

$$\textsc{RcvTx}\;\frac{\begin{array}{c} tp' = txExtend\ tp\ tx \qquad hs = \text{dom}(bf) \cup \left\{\#tx' \mid tx' \in tp'\right\} \\ ps = \{\langle \text{this}, a, \text{InvMsg}\ hs \rangle \mid a \in as\} \end{array}}{\langle \text{this}, as, bf, tp \rangle \xrightarrow{\langle \text{from, this, TxMsg}\ tx \rangle}_{\rho} (\langle \text{this}, as, bf, tp' \rangle, ps)}$$

$$\textsc{RcvBlock}\;\frac{\begin{array}{c} bf' = bf \triangleleft b \qquad tp' = \left\{tx \mid tx \in tp \land txValid\ tx\ \lceil bf' \rceil\right\} \\ hs = \text{dom}(bf') \cup \left\{\#tx \mid tx \in tp'\right\} \\ ps = \{\langle \text{this}, a, \text{InvMsg}\ hs \rangle \mid a \in as\} \end{array}}{\langle \text{this}, as, bf, tp \rangle \xrightarrow{\langle \text{from, this, BlockMsg}\ b \rangle}_{\rho} (\langle \text{this}, as, bf', tp' \rangle, ps)}$$

Receive-step transitions: $\delta \xrightarrow{p}_{\rho} (\delta', ps)$

RcvNull $\quad \delta \xrightarrow{\langle \text{from, this, NullMsg} \rangle}_{\rho} (\delta, \emptyset)$

RcvConnect

$$\frac{as' = as \cup \{\text{from}\} \qquad hs = \mathrm{dom}(bf) \cup \{\#tx \mid tx \in tp\} \qquad ps = \{\langle \text{this, from, InvMsg } hs \rangle\}}{\langle \text{this}, as, bf, tp \rangle \xrightarrow{\langle \text{from, this, ConnectMsg} \rangle}_{\rho} (\langle \text{this}, as', bf, tp \rangle, ps)}$$

RcvAddr

$$\frac{as_1 = \{a \mid a \in as' \wedge a \notin as\} \qquad as_2 = as \cup as_1 \qquad ps_1 = \{\langle \text{this}, a, \text{ConnectMsg} \rangle \mid a \in as_1\} \qquad ps_2 = \{\langle \text{this}, a, \text{AddrMsg } as_2 \rangle \mid a \in as\} \qquad ps = ps_1 \cup ps_2}{\langle \text{this}, as, bf, tp \rangle \xrightarrow{\langle \text{from, this, AddrMsg } as' \rangle}_{\rho} (\langle \text{this}, as_2, bf, tp \rangle, ps)}$$

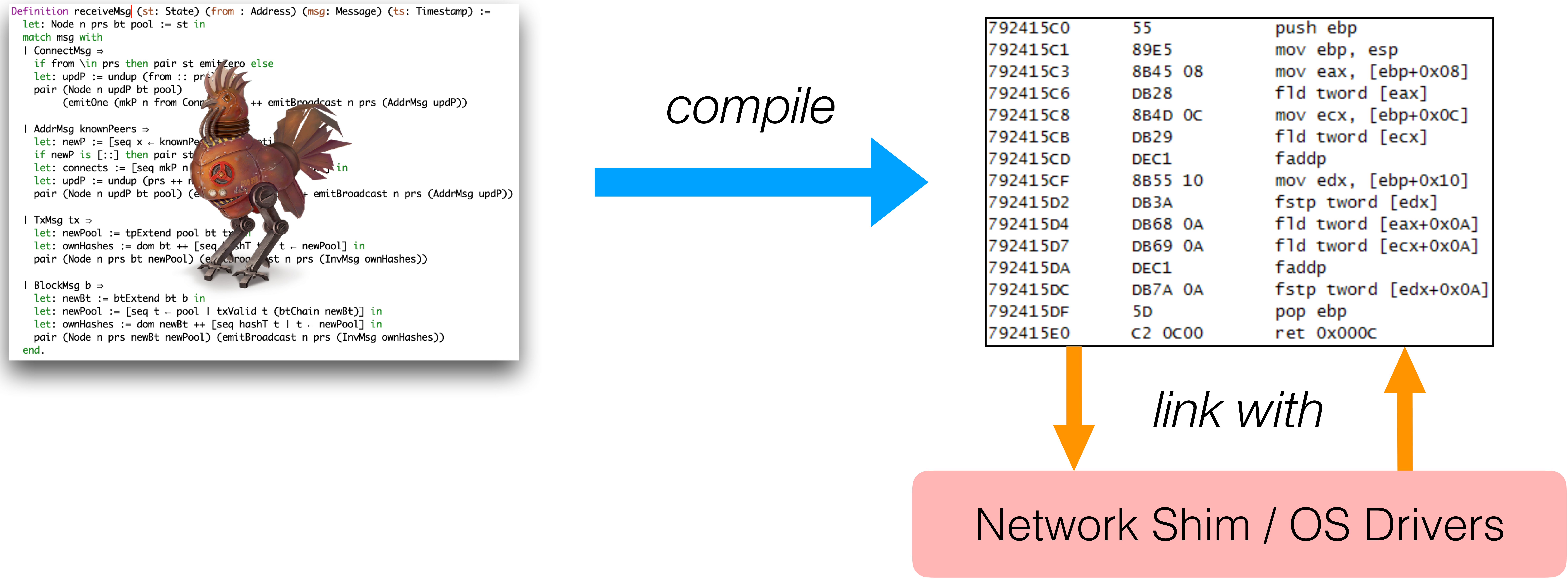
RcvTx

$$\frac{tp' = txExtend\ tp\ tx \qquad hs = \mathrm{dom}(bf) \cup \{\#tx' \mid tx' \in tp'\} \qquad ps = \{\langle \text{this}, a, \text{InvMsg } hs \rangle \mid a \in as\}}{\langle \text{this}, as, bf, tp \rangle \xrightarrow{\langle \text{from, this, TxMsg } tx \rangle}_{\rho} (\langle \text{this}, as, bf, tp' \rangle, ps)}$$

RcvBlock

$$\frac{bf' = bf \triangleleft b \qquad tp' = \{tx \mid tx \in tp \wedge txValid\ tx\ \lceil bf' \rceil\} \qquad hs = \mathrm{dom}(bf') \cup \{\#tx \mid tx \in tp'\} \qquad ps = \{\langle \text{this}, a, \text{InvMsg } hs \rangle \mid a \in as\}}{\langle \text{this}, as, bf, tp \rangle \xrightarrow{\langle \text{from, this, BlockMsg } b \rangle}_{\rho} (\langle \text{this}, as, bf', tp' \rangle, ps)}$$

```
Definition receiveMsg (st: State) (from : Address) (msg: Message) (ts: Timestamp) :=
  let: Node n prs bt pool := st in
  match msg with
  | ConnectMsg ⇒
    if from \in prs then pair st emitZero else
    let: updP := undup (from :: prs) in
    pair (Node n updP bt pool)
         (emitOne (mkP n from ConnectMsg) ++ emitBroadcast n prs (AddrMsg updP))

  | AddrMsg knownPeers ⇒
    let: newP := [seq x ← knownPeers | x \notin prs] in
    if newP is [::] then pair st emitZero else
    let: connects := [seq mkP n p ConnectMsg | p ← newP] in
    let: updP := undup (prs ++ newP) in
    pair (Node n updP bt pool) (emitMany connects ++ emitBroadcast n prs (AddrMsg updP))

  | TxMsg tx ⇒
    let: newPool := tpExtend pool bt tx in
    let: ownHashes := dom bt ++ [seq hashT t | t ← newPool] in
    pair (Node n prs bt newPool) (emitBroadcast n prs (InvMsg ownHashes))

  | BlockMsg b ⇒
    let: newBt := btExtend bt b in
    let: newPool := [seq t ← pool | txValid t (btChain newBt)] in
    let: ownHashes := dom newBt ++ [seq hashT t | t ← newPool] in
    pair (Node n prs newBt newPool) (emitBroadcast n prs (InvMsg ownHashes))
  end.
```



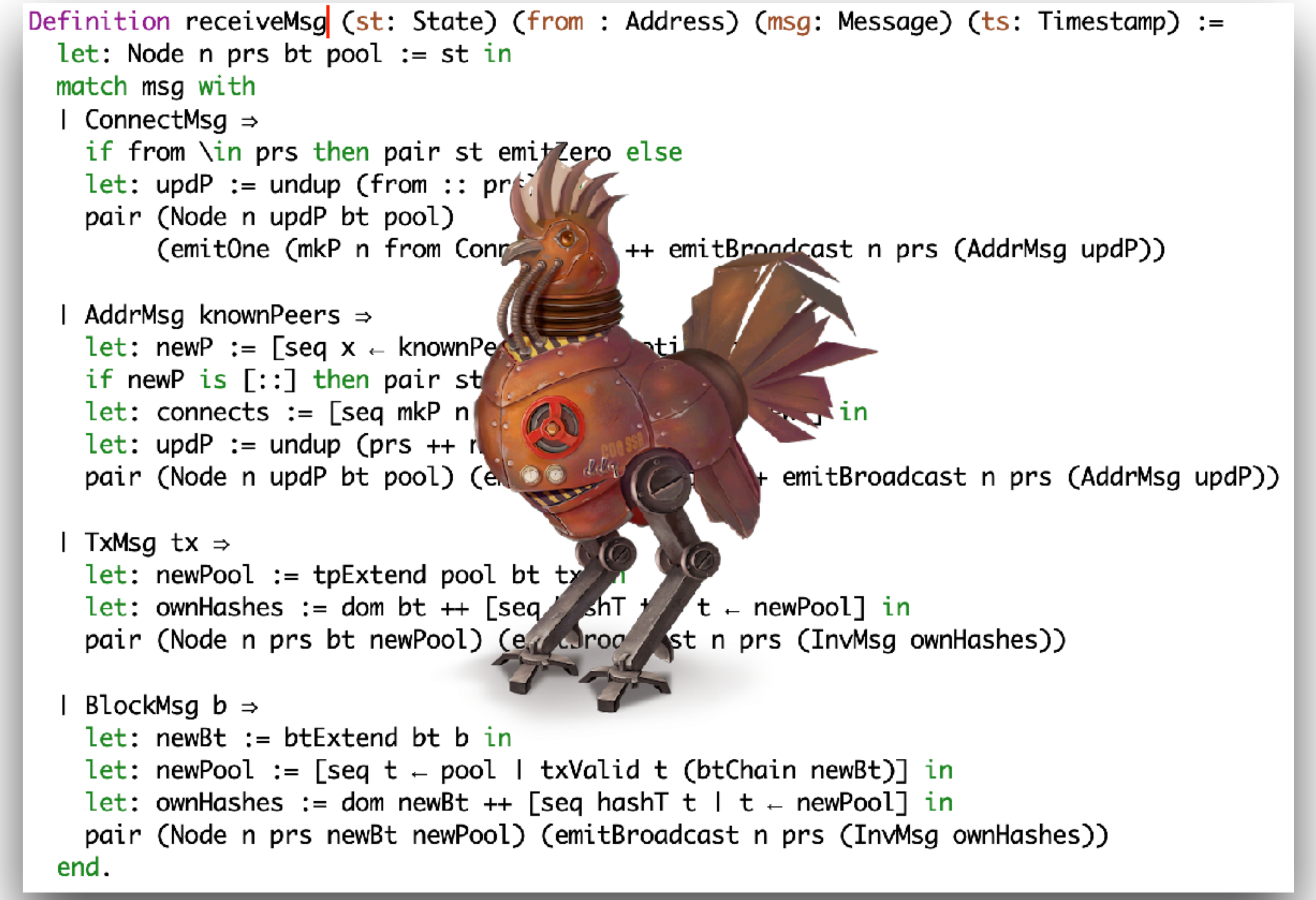
Definition receiveMsg (st: State) (from : Address) (msg: Message) (ts: Timestamp) :=
let: Node n prs bt pool := st in
match msg with
| ConnectMsg ⇒
pair (Node n updP bt pool)
| AddrMsg knownPeers ⇒
emitBroadcast n prs (AddrMsg updP))
| TxMsg tx ⇒
let: newPool := tpExtend pool bt
st n prs (InvMsg ownHashes))
| BlockMsg b ⇒
let: newBt := btExtend bt b in
let: newPool := [seq t ← pool | txValid t (btChain newBt)] in
let: ownHashes := dom newBt ++ [seq hashT t | t ← newPool] in
pair (Node n prs newBt newPool) (emitBroadcast n prs (InvMsg ownHashes))
end.
compile
792415C0 55 push ebp
792415C1 89E5 mov ebp, esp
792415C3 8B45 08 mov eax, [ebp+0x08]
792415C6 DB28 fld tword [eax]
792415C8 8B4D 0C mov ecx, [ebp+0x0C]
792415CB DB29 fld tword [ecx]
792415CD DEC1 faddp
792415CF 8B55 10 mov edx, [ebp+0x10]
792415D2 DB3A fstp tword [edx]
792415D4 DB68 0A fld tword [eax+0x0A]
792415D7 DB69 0A fld tword [ecx+0x0A]
792415DA DEC1 faddp
792415DC DB7A 0A fstp tword [edx+0x0A]
792415DF 5D pop ebp
792415E0 C2 0C00 ret 0x000C
link with
Network Shim / OS Drivers

```
1   |||||||                         18.1%  5   ||||||||||||||||||||||||||||||98.1%
2   ||||||||||||||||||||||||||||||98.1%    6   ||||||                         12.9%
3   ||||||||||||||||||||||||||||||96.2%    7   ||||||||||||||||||||||||||||||95.7%
4   |||||                           13.0%  8   |||||||                        13.6%
Mem ||||||||||||||||||||||||   2.87G/7.65G  Tasks: 136, 771 thr, 162 kthr; 1 running
Swp                               0K/7.85G  Load average:      1.27 0.76
                                            Uptime: 04:00:55
  PID CPU% MEM% Command
 7939 91.6  0.1 ./node.native -me 127.0.0.1 9001 -cluster 127.0.0.1 9000 127.0.0.1 90
 7962 91.6  0.1 ./node.native -me 127.0.0.1 9003 -cluster 127.0.0.1 9001
 7937 89.7  0.1 ./node.native -me 127.0.0.1 9000 -cluster 127.0.0.1 9000 127.0.0.1 90
 7941 89.7  0.1 ./node.native -me 127.0.0.1 9002 -cluster 127.0.0.1 9000 127.0.0.1 90

F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

```
36.00 hashes per second


---------
Chain
0x91dd9087 = {prev = 0x6150cb35, txs = , nonce = 0}
0x00008507 = {prev = 0x91dd9087, txs = TX 2011, nonce = 915684349}
0x0004cddf = {prev = 0x00008507, txs = TX 10041 TX 6718, nonce = 245321381}
0x00041372 = {prev = 0x0004cddf, txs = , nonce = 206618863}
0x0006a3f1 = {prev = 0x00041372, txs = , nonce = 912732568}
0x000602d7 = {prev = 0x0006a3f1, txs = , nonce = 637842253}
0x0001a002 = {prev = 0x000602d7, txs = , nonce = 388297155}
0x000302ce = {prev = 0x0001a002, txs = , nonce = 452593757}
0x0005064b = {prev = 0x000302ce, txs = , nonce = 873241682}
0x0001af4b = {prev = 0x0005064b, txs = , nonce = 970161653}
0x000526b9 = {prev = 0x0001af4b, txs = , nonce = 92284887}
---------
40.40 hashes per second
```

```
dranov@dranov-laptop:~/Developer/toychain$ tail -n20 node-00.log
Received packet (127.0.0.1:9000, 127.0.0.1:9000, InvMsg [0x00008507; 0x0001a002; 0x0
001af4b; 0x000302ce; 0x00041372; 0x0004cddf; 0x0004eeaa; 0x0005064b; 0x000526b9; 0x0
00602d7; 0x0006a3f1; 0x91dd9087; 0xa8320b01])
Received packet (127.0.0.1:9003, 127.0.0.1:9000, InvMsg [0x00008507; 0x0001a002; 0x0
001af4b; 0x000302ce; 0x00041372; 0x0004cddf; 0x0004eeaa; 0x0005064b; 0x000526b9; 0x0
00602d7; 0x0006a3f1; 0x91dd9087; 0xa8320b01])
Received packet (127.0.0.1:9002, 127.0.0.1:9000, InvMsg [0x00008507; 0x0001a002; 0x0
001af4b; 0x000302ce; 0x00041372; 0x0004cddf; 0x0004eeaa; 0x0005064b; 0x000526b9; 0x0
00602d7; 0x0006a3f1; 0x91dd9087; 0xa8320b01])

---------
Chain
0x91dd9087 = {prev = 0x6150cb35, txs = , nonce = 0}
0x00008507 = {prev = 0x91dd9087, txs = TX 2011, nonce = 915684349}
0x0004cddf = {prev = 0x00008507, txs = TX 10041 TX 6718, nonce = 245321381}
0x00041372 = {prev = 0x0004cddf, txs = , nonce = 206618863}
0x0006a3f1 = {prev = 0x00041372, txs = , nonce = 912732568}
0x000602d7 = {prev = 0x0006a3f1, txs = , nonce = 637842253}
0x0001a002 = {prev = 0x000602d7, txs = , nonce = 388297155}
0x000302ce = {prev = 0x0001a002, txs = , nonce = 452593757}
0x0005064b = {prev = 0x000302ce, txs = , nonce = 873241682}
0x0001af4b = {prev = 0x0005064b, txs = , nonce = 970161653}
0x000526b9 = {prev = 0x0001af4b, txs = , nonce = 92284887}
---------
44.10 hashes per second

dranov@dranov-laptop:~/Developer/toychain$
```

```
[0] 0:bash*                                          "dranov-laptop" 22:16 03-Apr-19
```



compile

```
792415C0    55        push ebp
792415C1    89E5      mov ebp, esp
792415C3    8B45 08   mov eax, [ebp+0x08]
792415C6    DB28      fld tword [eax]
792415C8    8B4D 0C   mov ecx, [ebp+0x0C]
792415CB    DB29      fld tword [ecx]
792415CD    DEC1      faddp
792415CF    8B55 10   mov edx, [ebp+0x10]
792415D2    DB3A      fstp tword [edx]
792415D4    DB68 0A   fld tword [eax+0x0A]
792415D7    DB69 0A   fld tword [ecx+0x0A]
792415DA    DEC1      faddp
792415DC    DB7A 0A   fstp tword [edx+0x0A]
792415DF    5D        pop ebp
792415E0    C2 0C00   ret 0x000C
```

link with

Network Shim / OS Drivers

Have to be trusted!

Assumptions:

- Meaningful definition of per-node *protocol state machine*
- Meaningful definition of the *network semantics*
- Precise *fault model* (e.g., nodes are non-byzantine)
- The implementation is *correctly compiled*
- The *network/OS software is reliable*

Eventual Consistency Theorem:

$$\forall\, sc\ \forall\, n_1, n_2 \in \mathbf{N},\ run_{BC}(\mathbf{N}, sc).n_1.chain = run_{BC}(\mathbf{N}, sc).n_2.chain$$

Proof: By *induction on the length* of *sc* (a number of system steps).

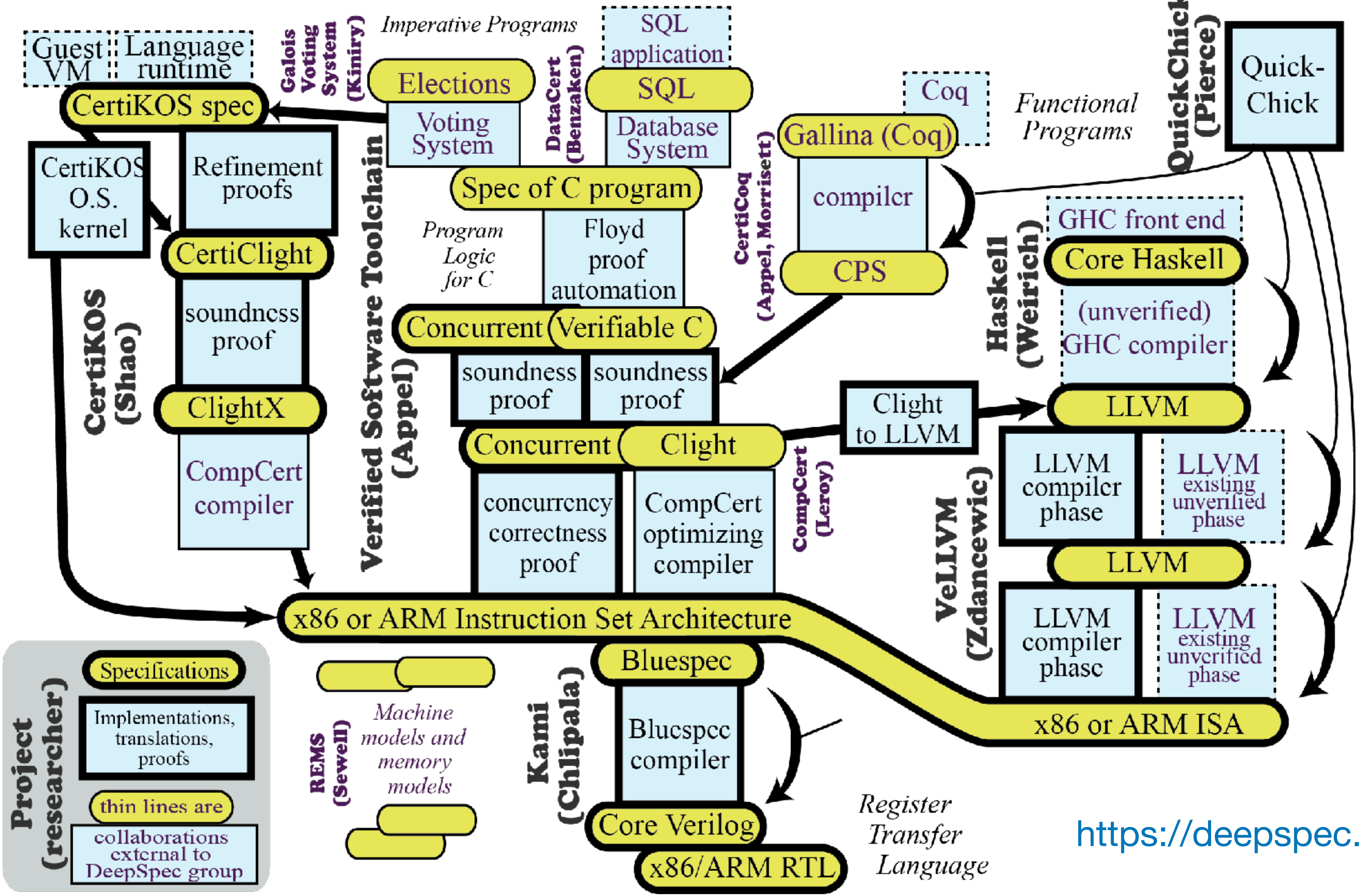
Assumptions:

- Meaningful definition of per-node *protocol state machine*
- Meaningful definition of the *network semantics*
- Precise *fault model* (e.g., nodes are non-byzantine)
- The implementation is *correctly compiled*
- The *network/OS software is reliable*

Eventual Consistency Theorem:

$$\forall\, sc\ \forall\, n_1, n_2 \in \mathbf{N},\ run_{BC}(\mathbf{N}, sc).n_1.chain = run_{BC}(\mathbf{N}, sc).n_2.chain$$

Proof: By *induction on the length* of *sc* (a number of system steps).



https://deepspec.org

State of the Art
in Formally Verified Systems

CompCert (2006-now)

a mechanically verified C compiler

Formal Certification of a Compiler Back-end

***or*: Programming a Compiler with a Proof Assistant**

Xavier Leroy
INRIA Rocquencourt
Xavier.Leroy@inria.fr

- Specification: source and target programs are equivalent
- Assumptions: underlying hardware semantics, unverified parser
- Proof effort: 146 kLOC of specifications and proofs

FSCQ (2015)

a crash-tolerant file system

Using Crash Hoare Logic for Certifying the FSCQ File System

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich
MIT CSAIL

- **Specification:** asynchronous disk writes are not affected by crashes
- **Assumptions** about semantics of extraction and linking with other drivers
- **Proof effort:** 81 kLOC of specifications and proofs

Verdi (2015)

a formally verified Raft consensus implementation

Verdi: A Framework for Implementing and Formally Verifying Distributed Systems

James R. Wilcox   Doug Woos   Pavel Panchekha
Zachary Tatlock   Xi Wang   Michael D. Ernst   Thomas Anderson
University of Washington, USA
{jrw12, dwoos, pavpan, ztatlock, xi, mernst, tom}@cs.washington.edu

- **Specification:** Raft provides *transparent replication* (linearisability)
- **Assumptions:** unlimited memory, TCP works atomically, …
- **Proof effort:** 50 kLOC of specifications and proofs

Does it really work?

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr
University of Utah, School of Computing
{jxyang, chenyang, eeide, regehr}@cs.utah.edu

(in PLDI 2011)

Compilers should be correct.

To improve the quality of C compilers, we created Csmith, a **randomized test-case generation tool**, and spent **three years** using it to find compiler bugs.

During this period we reported **more than 325 previously unknown bugs** to compiler developers.

The striking thing about our **CompCert** results is that the middle-end bugs we found in all other compilers are **absent**.

As of early 2011, the under-development version of **CompCert** is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task.

The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

Bye-bye testing?

Formal Verification is Expensive

- CompCert
  146 kLOC
- Verdi
  50 kLOC
- FSCQ
  81 kLOC

Formal Verification is Expensive

- CompCert
  146 kLOC, 10+ person-years
- Verdi
  50 kLOC, 3+ person-years
- FSCQ
  81 kLOC, 5+ person-years

Formal Verification is Expensive

- CompCert
  146 kLOC, 10+ person-**years**
- Verdi
  50 kLOC, 3+ person-**years**
- FSCQ
  81 kLOC, 5+ person-**years**

Assumptions Matter

Story 1: CompCert

Finding and Understanding Bugs in C Compilers

Xuejun Yang Yang Chen Eric Eide John Regehr

University of Utah, School of Computing

{ jxyang, chenyang, eeide, regehr }@cs.utah.edu

The second CompCert problem we found was illustrated by two bugs that resulted in generation of code like this:

```
stwu r1, -44432(r1)
```

Here, a large PowerPC stack frame is being allocated. The problem is that the 16-bit displacement field is overflowed. CompCert's PPC semantics failed to specify a constraint on the width of this immediate value, on the assumption that the assembler would catch out-of-range values. In fact, this is what happened. We also found a

Wrong assumption about compiled assembly execution!

Story 2: FSCQ

We found a bug in a verified file system! We ran Crashmonkey's suite of tests on MIT's FSCQ and found that it does not persist data on fdatasync properly. We emailed the authors, they have acked and fixed the bug.

Come see our paper at #osdi18!

Details: github.com/utsaslab/crash…

Vijay Chidambaram @vj_chidambaram
Excited to share our #osdi18 paper on finding crash-consistency bugs in Linux file systems! I will explain the intuition behind our system in this thread....
Show this thread

Story 2: FSCQ

We found a bug in a verified file system! We ran Crashmonkey's suite of tests on MIT's FSCQ and found that it does not persist data on fdatasync properly. We emailed the authors, they have acked and fixed the bug.

Come see our paper at #osdi18!

Details: github.com/utsaslab/crash

Vijay Chidambaram @vj_chidambaram
Excited to share our #osdi18 paper on finding crash-consistency b
Linux file systems! I will explain the intuition behind our system in t
thread....
Show this thread

John Regehr @johnregehr · Oct 3

Replying to @vj_chidambaram

what was the root cause of their failure to find this bug during verification?

1 3

Vijay Chidambaram @vj_chidambaram · Oct 3

Even verified file systems have unverified parts :) it was due to a buggy optimization in the Haskell-c bindings.

1 5 4

Linking with
unverified component

Story 3: Verdi

An Empirical Study on the Correctness of Formally Verified Distributed Systems

Pedro Fonseca Kaiyuan Zhang Xi Wang Arvind Krishnamurthy
University of Washington

Overall, 7 bugs are found

4.3 Resource Limits

This section describes three bugs that involve exceeding resource limits.

Bug V6: *Large packets cause server crashes.*

The server code that handled incoming packets had a bug that could cause the server to crash under certain conditions. The bug, due to an insufficiently small buffer in the OCaml code, caused incoming packets to truncate large packets and subsequently prevented the server from correctly unmarshaling the message.

Wrong assumption about the crash model!

Checkpoint

- A distributed system engineered in a *proof assistant* can be run and rely on *independently verified components*.
- *Costs* of formal verification *are high*, but so are the provided *correctness guarantees*
- ***Mind the gap***: assumptions *might be broken* in the real world, thus invalidating the claims of theorems
- *Testing* helps to validate the assumptions.

Checkpoint

- A distributed system engineered in a *proof assistant* can be run and rely on *independently verified components*.
- *Costs* of formal verification *are high*, but so are the provided *correctness guarantees*
- ***Mind the gap***: assumptions *might be broken* in the real world, thus invalidating the claims of theorems
- *Testing* helps to validate the assumptions.

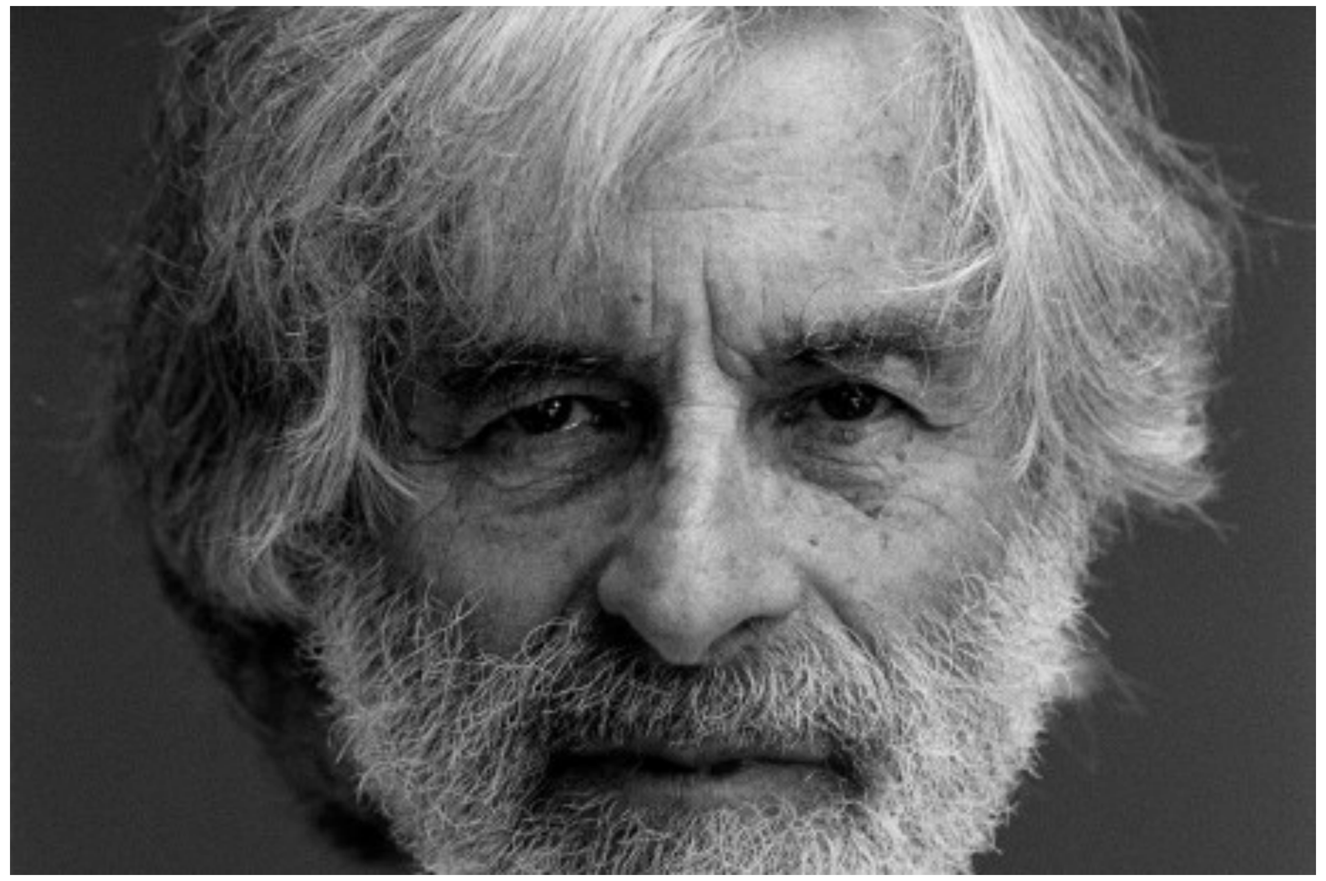
Checkpoint

- A distributed system engineered in a *proof assistant* can be run and rely on *independently verified components*
- *Costs* of formal verification *are high*, but so are the provided *correctness guarantees*
- ***Mind the gap***: assumptions *might be broken* in the real world, thus invalidating the claims of theorems
- *Testing* helps to validate the assumptions.

Composition

Composition:
A Way to Make Proofs Harder

Lamport, 1997

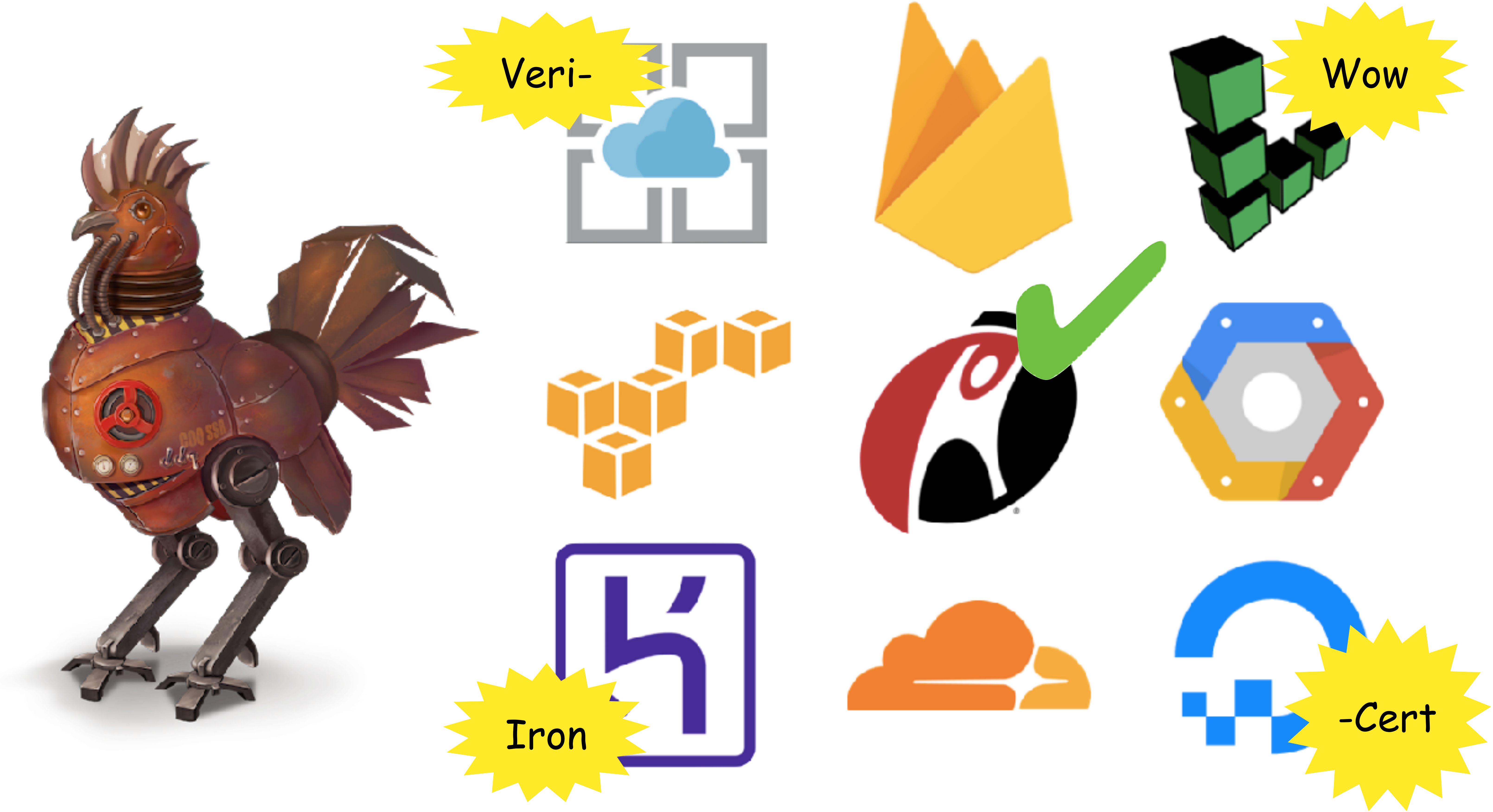


In 1997, the unfortunate reality is that engineers rarely specify and reason formally about the systems they build.

It seems unlikely that reasoning about the composition of open-system specifications will be a practical concern within the next 15 years.

Distributed *Infrastructure*

Verified Distributed *Infrastructure*



Distributed *Applications*

Verified Distributed *Applications*

Veri-

Wow

Iron

-Cert

Frameworks for Compositional Systems Validation

Compositional Programming and Testing of Dynamic Distributed Systems

ANKUSH DESAI, University of California, Berkeley, USA
AMAR PHANISHAYEE, Microsoft Research, USA
SHAZ QADEER, Microsoft Research, USA
SANJIT A. SESHIA, University of California, Berkeley, USA

Modularity for Decidability of Deductive Verification with Applications to Distributed Systems

Marcelo Taube
Tel Aviv University, Israel
mail.marcelo.taube@gmail.com

Giuliano Losa
UCLA, USA
giuliano@cs.ucla.edu

Kenneth L. McMillan
Microsoft Research, USA
kenmcmil@microsoft.com

Oded Padon
Tel Aviv University, Israel
odedp@mail.tau.ac.il

Mooly Sagiv
Tel Aviv University, Israel
msagiv@post.tau.ac.il

Sharon Shoham
Tel Aviv University, Israel
sharon.shoham@gmail.com

James R. Wilcox
University of Washington, USA
jrw12@cs.washington.edu

Doug Woos
University of Washington, USA
dwoos@cs.washington.edu

Programming and Proving with Distributed Protocols

ILYA SERGEY, University College London, UK
JAMES R. WILCOX, University of Washington, USA
ZACHARY TATLOCK, University of Washington, USA

Composition of Verified Systems

- **Horizontal:** *independently* verified implementations interact in the *same system*

- **Vertical:** A system can be run on top of a *different back-ends*

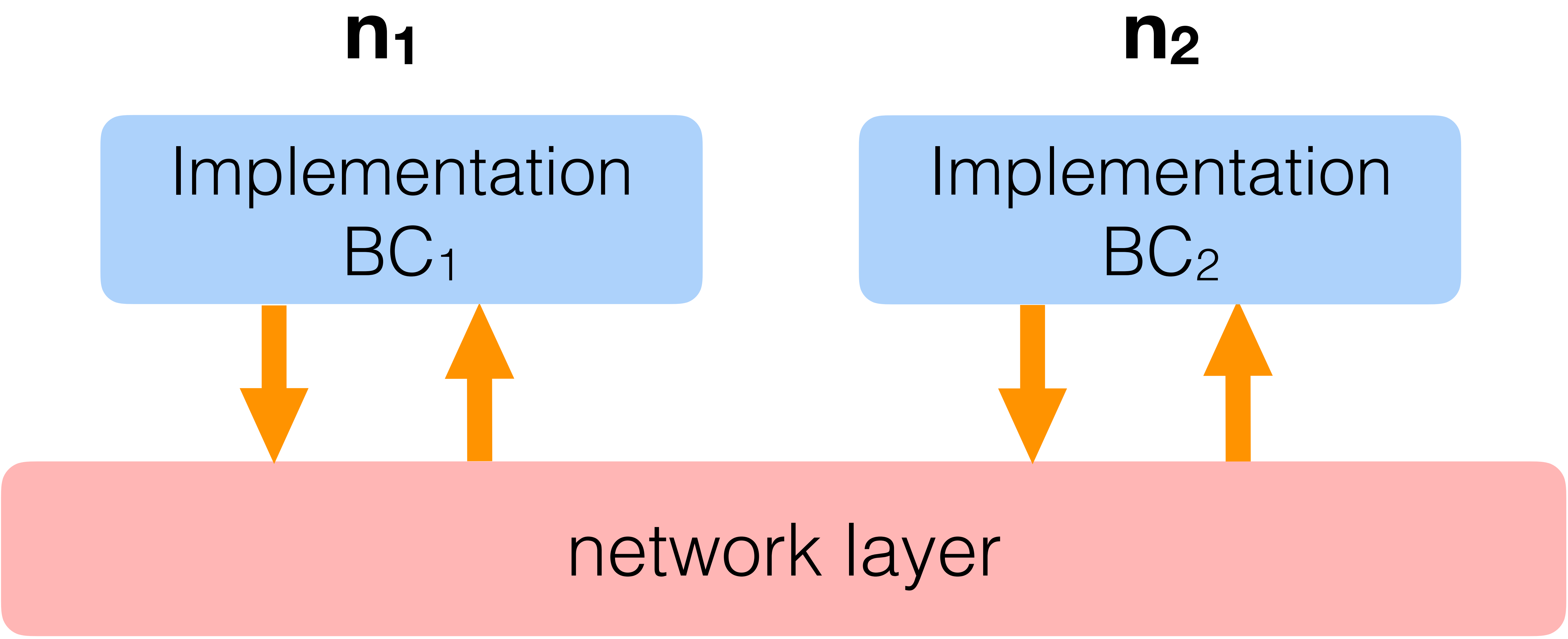
Towards Reusable Verification

Blockchain Eventual Consistency:

$\forall\ sc\ \forall\ \mathrm{n}_1, \mathrm{n}_2 \in \mathbf{N},\ \mathit{run}_{\mathit{BC}}(\mathbf{N}, sc).\mathrm{n}_1.\mathit{chain} = \mathit{run}_{\mathit{BC}}(\mathbf{N}, sc).\mathrm{n}_2.\mathit{chain}$

HO Logics for Horizontal Composition

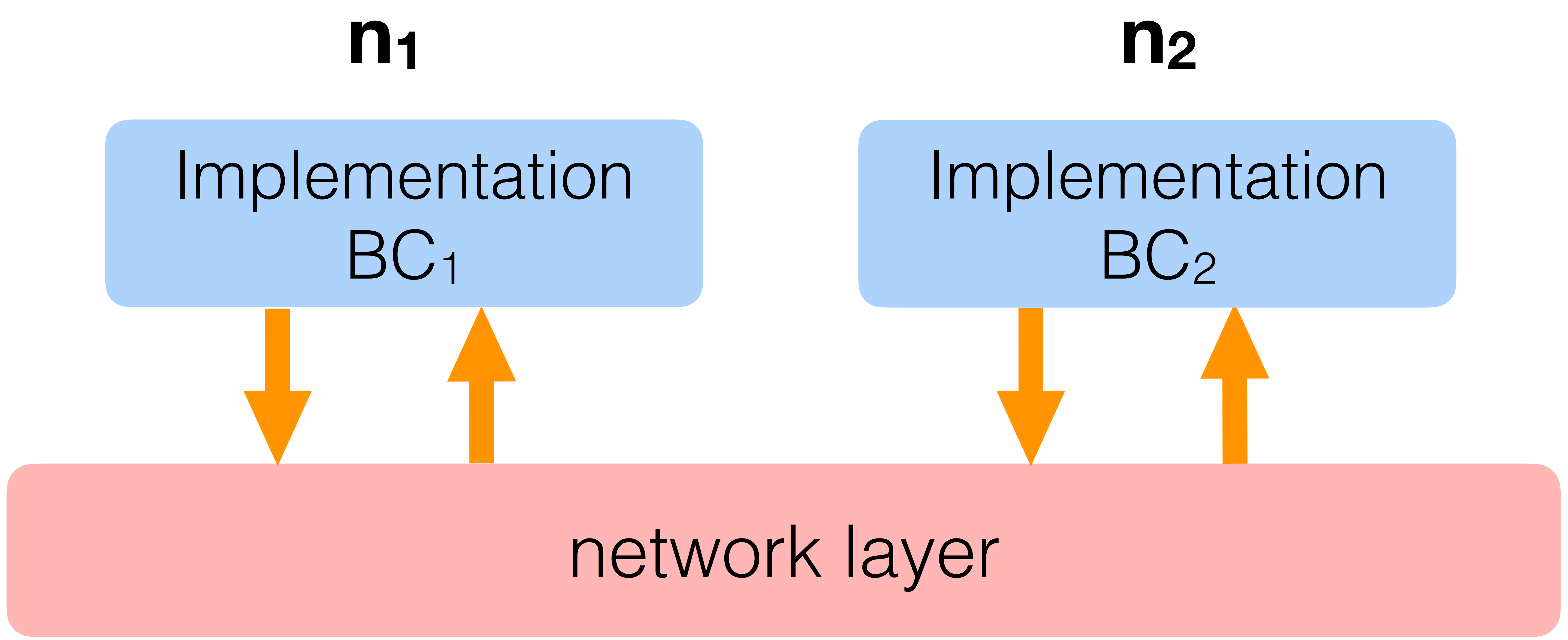
$$\forall\, sc\ \forall\, n_1, n_2 \in \mathbf{N},\ run_{BC}(\mathbf{N}, sc).n_1.chain = run_{BC}(\mathbf{N}, sc).n_2.chain$$



HO Logics for Horizontal Composition

$\forall$ BC_1, BC_2, BC_1 *refines* BC and BC_2 *refines* BC $\Rightarrow$

$\forall$ *sc* $\forall$ n_1, $n_2 \in \mathbf{N}$, $run_{BC_1}(\mathbf{N}, sc).n_1.chain = run_{BC_2}(\mathbf{N}, sc).n_2.chain$



HO Logics for Horizontal Composition

$\forall\ \boxed{BC_1, BC_2},\ BC_1\ \textit{refines}\ BC \text{ and } BC_2\ \textit{refines}\ BC \Rightarrow$

$\forall\ sc\ \forall\ n_1, n_2 \in \mathbf{N},\ \textit{run}_{BC_1}(\mathbf{N}, sc).n_1.\textit{chain} = \textit{run}_{BC_2}(\mathbf{N}, sc).n_2.\textit{chain}$

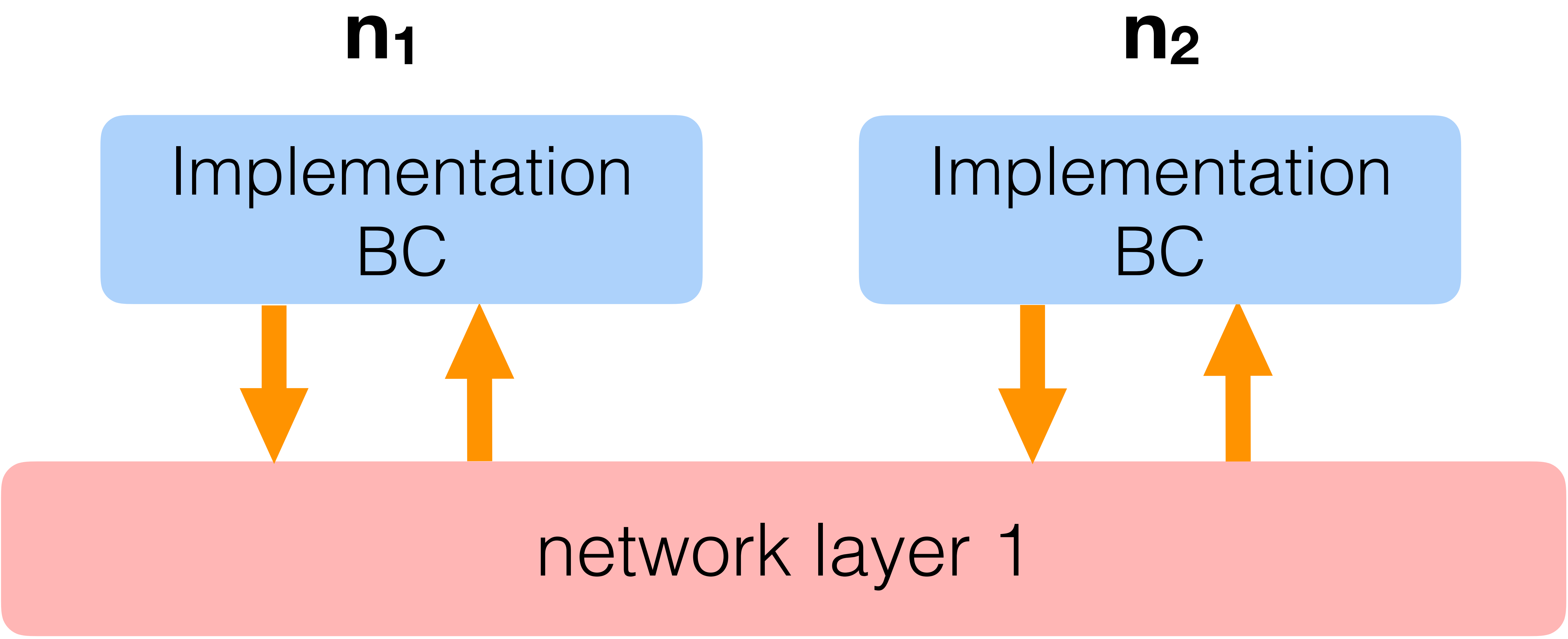
Programming and Proving with Distributed Protocols

ILYA SERGEY, University College London, UK
JAMES R. WILCOX, University of Washington, USA
ZACHARY TATLOCK, University of Washington, USA

POPL'18

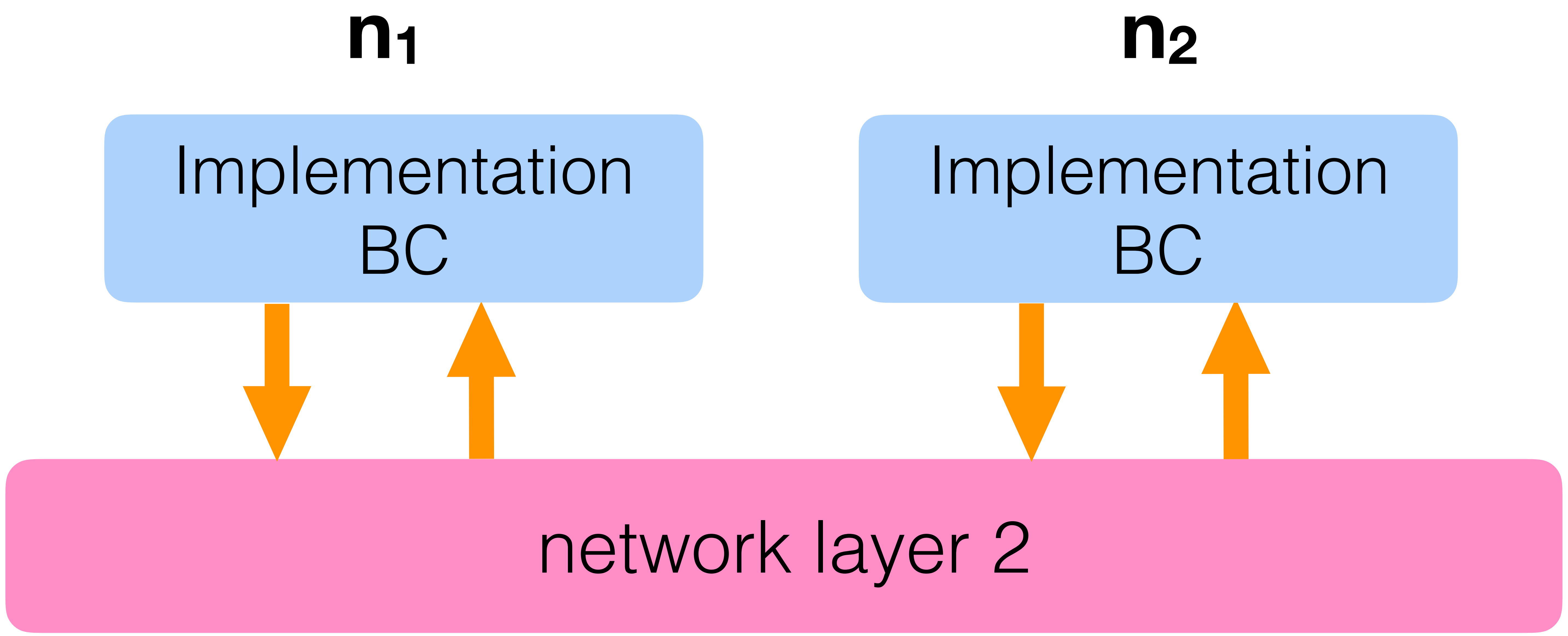
HO Logics for Vertical Composition

$\forall\, sc\ \forall\, n_1, n_2 \in \mathbf{N},\ run_{BC}(\mathbf{N}, sc).n_1.chain = run_{BC}(\mathbf{N}, sc).n_2.chain$



HO Logics for Vertical Composition

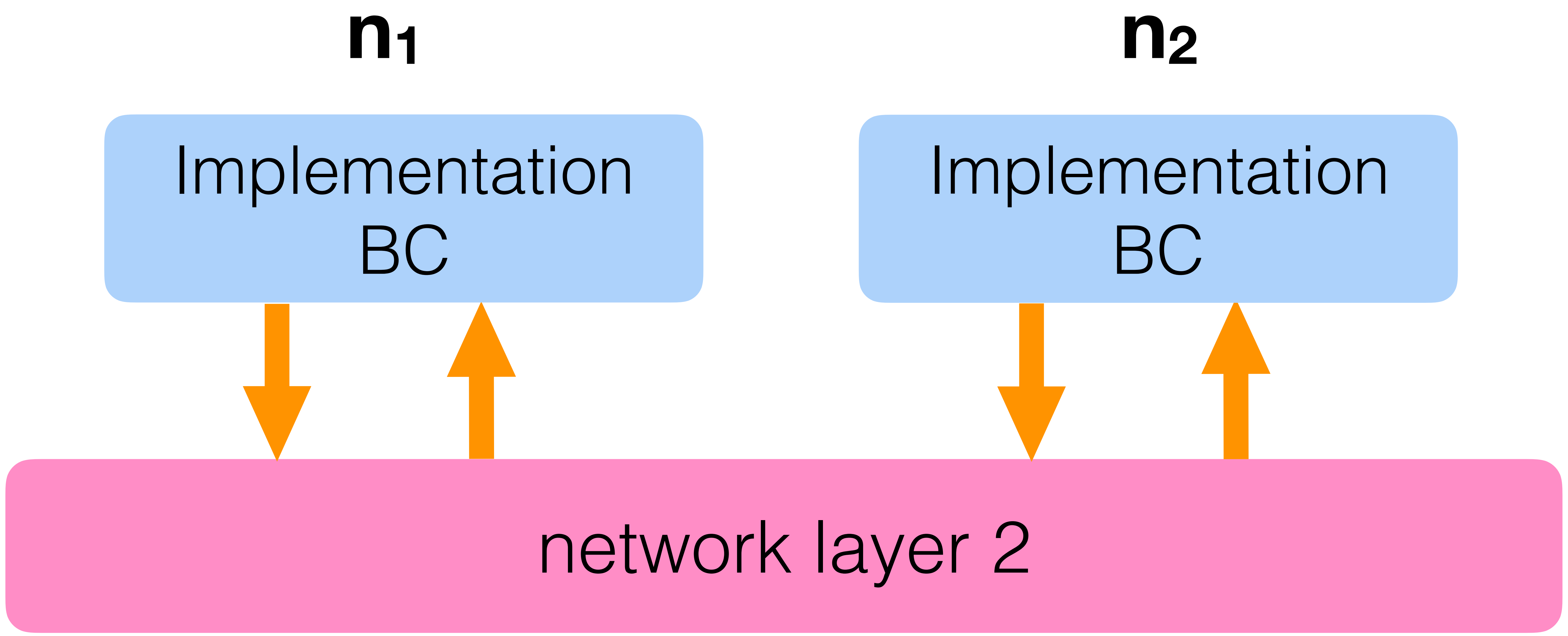
$\forall\, sc\ \forall\, n_1, n_2 \in \mathbf{N},\ \mathit{run}_{\mathit{BC}}(\mathbf{N}, sc).n_1.\mathit{chain} = \mathit{run}_{\mathit{BC}}(\mathbf{N}, sc).n_2.\mathit{chain}$



HO Logics for Vertical Composition

$\forall$ *run* such that *BC* tolerates *run*'s faults $\Rightarrow$

$\forall\, sc\ \forall\, n_1, n_2 \in \mathbf{N},\ run_{BC}(\mathbf{N}, sc).n_1.chain = run_{BC}(\mathbf{N}, sc).n_2.chain$



HO Logics for Vertical Composition

$\forall$ *run* such that *BC* tolerates *run*'s faults $\Rightarrow$

$\forall\, sc\ \forall\, n_1, n_2 \in \mathbf{N},\ run_{BC}(\mathbf{N}, sc).n_1.chain = run_{BC}(\mathbf{N}, sc).n_2.chain$

Verdi: A Framework for Implementing and Formally Verifying Distributed Systems

James R. Wilcox   Doug Woos   Pavel Panchekha
Zachary Tatlock   Xi Wang   Michael D. Ernst   Thomas Anderson
University of Washington, USA
{jrw12, dwoos, pavpan, ztatlock, xi, mernst, tom}@cs.washington.edu

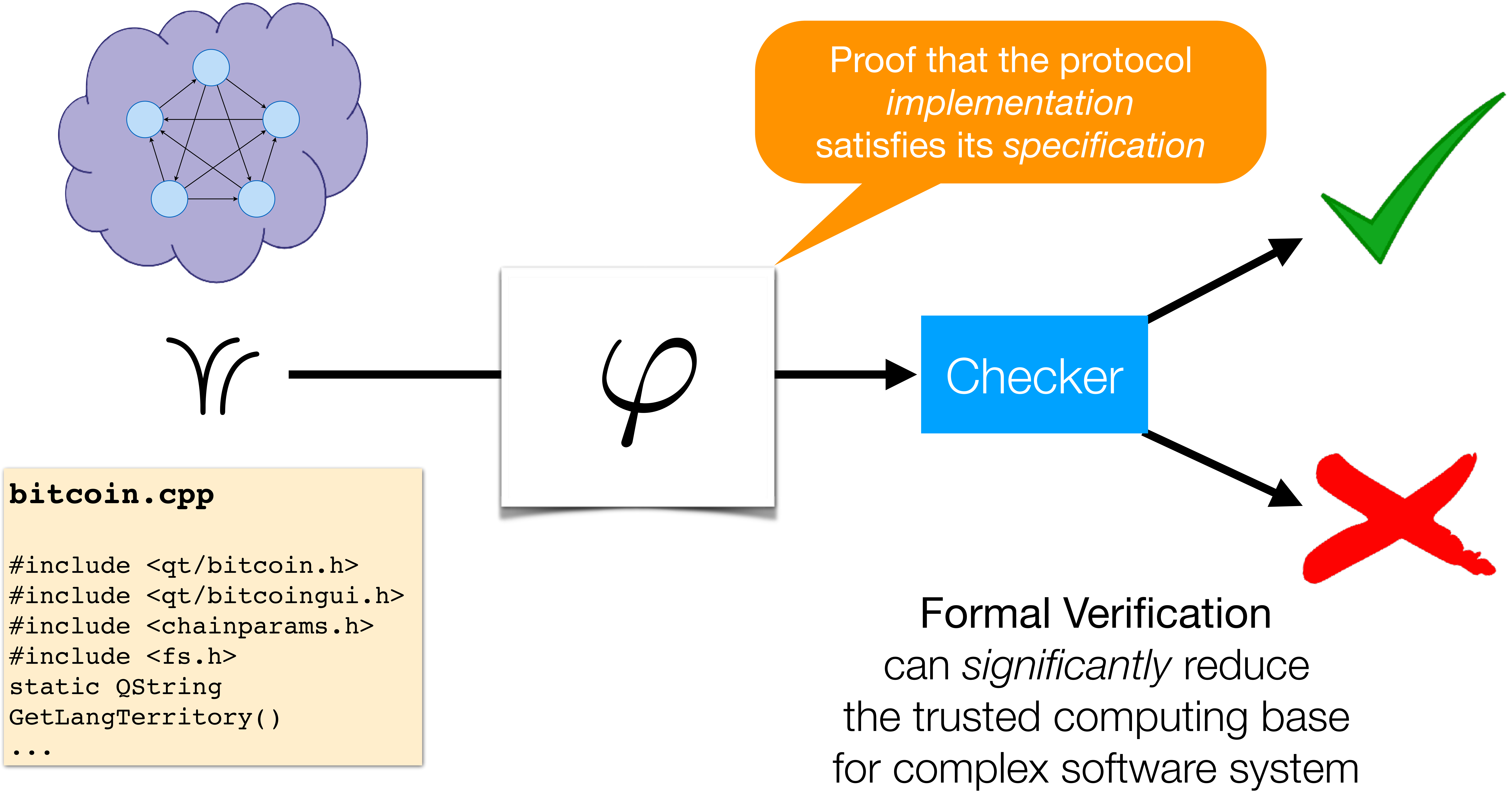
PLDI'15

Conclusion

To Take Away

- We still don't have *fully verified* distributed systems (and probably never will).
- But now we know how to engineer them so we *can remove many assumptions* about crucial implementation components, thanks to *machine-checked proofs*.
- Proof assistants that are also programming languages (e.g., **Coq**) allow one to verify *runnable implementations*.
- Higher-order specifications are crucial for *compositionality and proof reuse*.

Thanks!


Proof that the protocol *implementation* satisfies its *specification*
φ
Checker
bitcoin.cpp
#include <qt/bitcoin.h>
#include <qt/bitcoingui.h>
#include <chainparams.h>
#include <fs.h>
static QString
GetLangTerritory()
...
Formal Verification
can *significantly* reduce
the trusted computing base
for complex software system