Toychain

Formally Verified Blockchain Consensus

George Pîrlea and Ilya Sergey

Toychain

- Internship project with Ilya in summer 2017
  - mechanised proof of quiescent consistency in Coq
  - published at CPP in Jan 2018

- Continued for my Master's thesis
  - **extraction to Ocaml**
  - *proven-correct implementation* of proof-of-work Nakamoto consensus

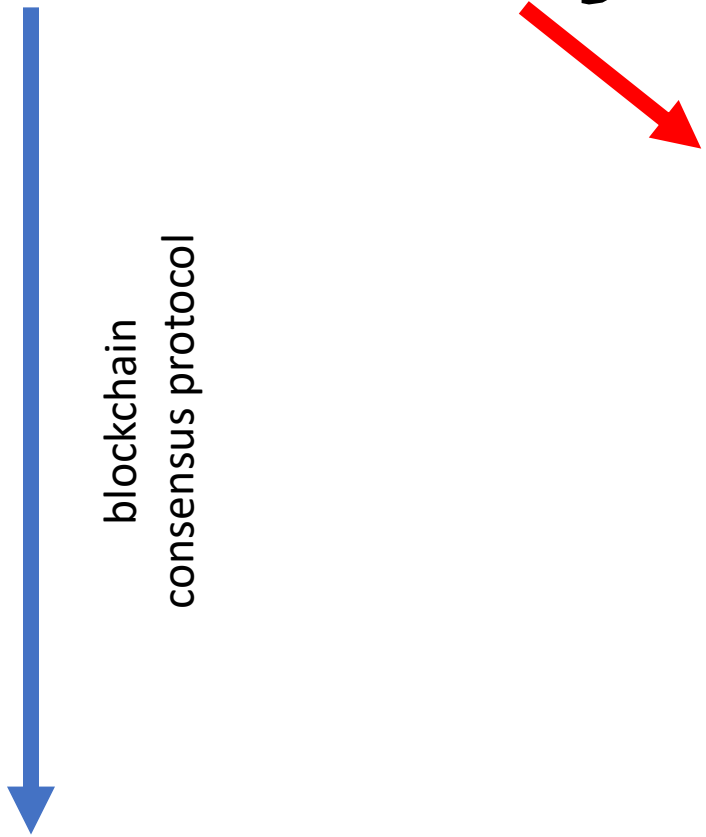
Motivation

1. Understand blockchain consensus
    - **what** it is
    - **how** it works
    - **why** it works: our formalisation

2. Lay foundation for *verified* practical implementation
    - verified Byzantine-tolerant consensus layer **Not there yet!**
    - platform for verified smart contracts

Nakamoto-style vs BFT-style

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

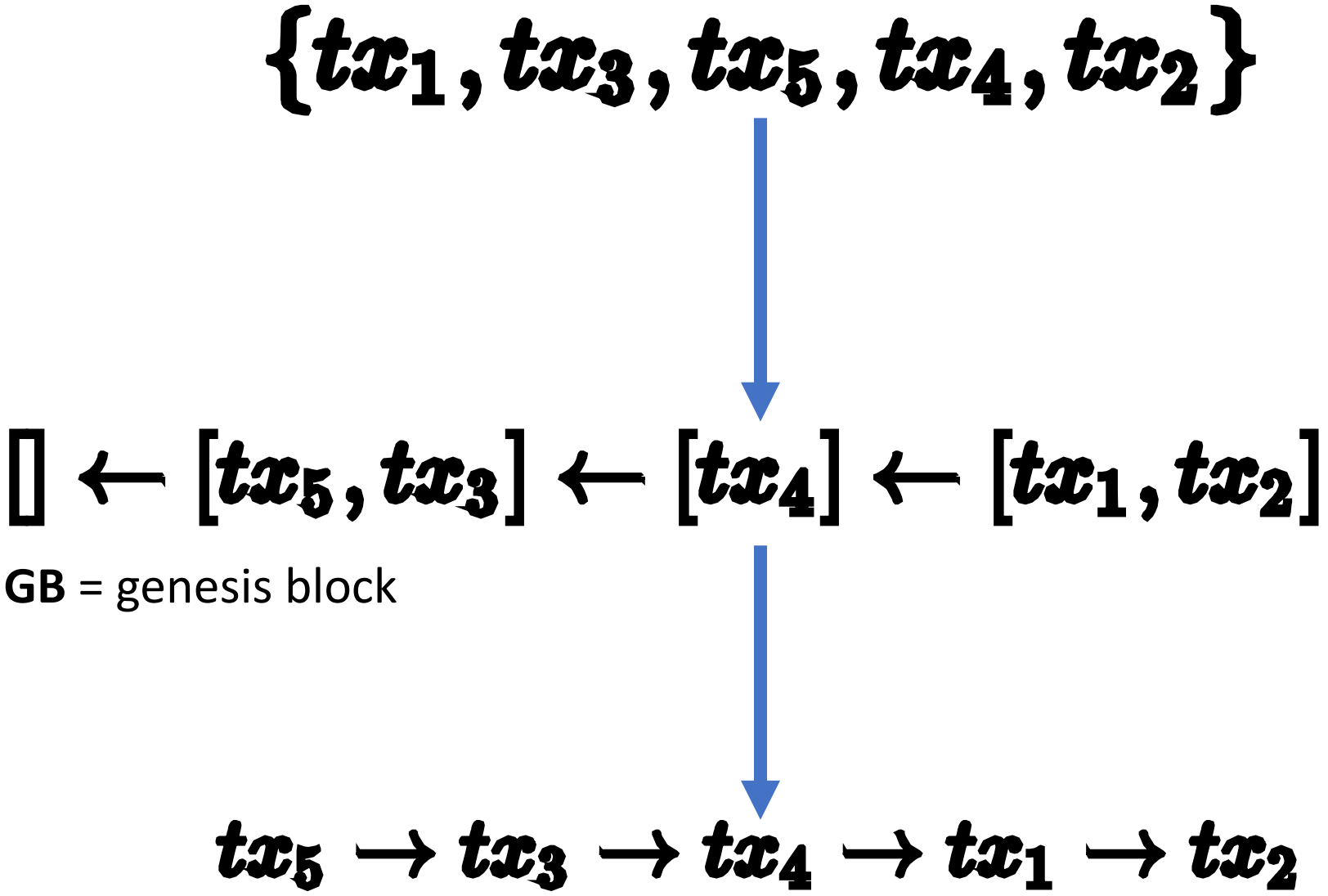


meaning of transactions is not relevant

blockchain consensus protocol

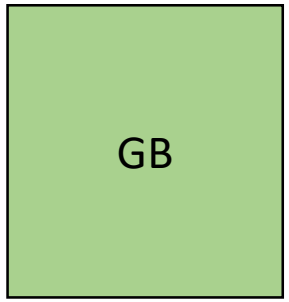
- transforms a **set** of transactions into a *globally-agreed* **sequence**
- "distributed timestamp server" (Nakamoto2008)

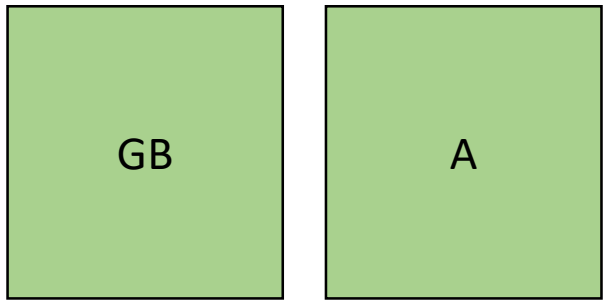
$$tx_1 \rightarrow tx_2 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_5$$

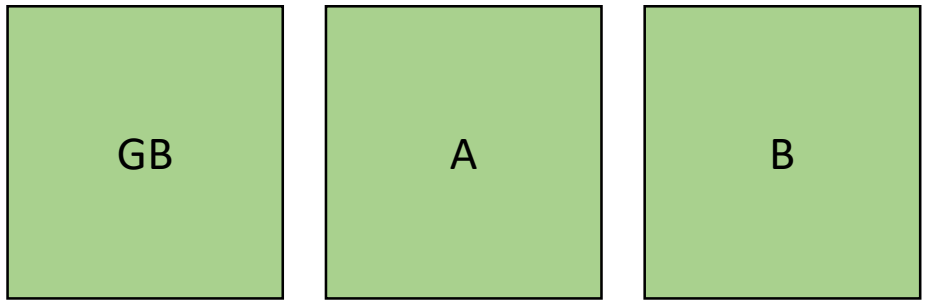

$\{tx_1, tx_3, tx_5, tx_4, tx_2\}$
$[] \leftarrow [tx_5, tx_3] \leftarrow [tx_4] \leftarrow [tx_1, tx_2]$
GB = genesis block
$tx_5 \rightarrow tx_3 \rightarrow tx_4 \rightarrow tx_1 \rightarrow tx_2$

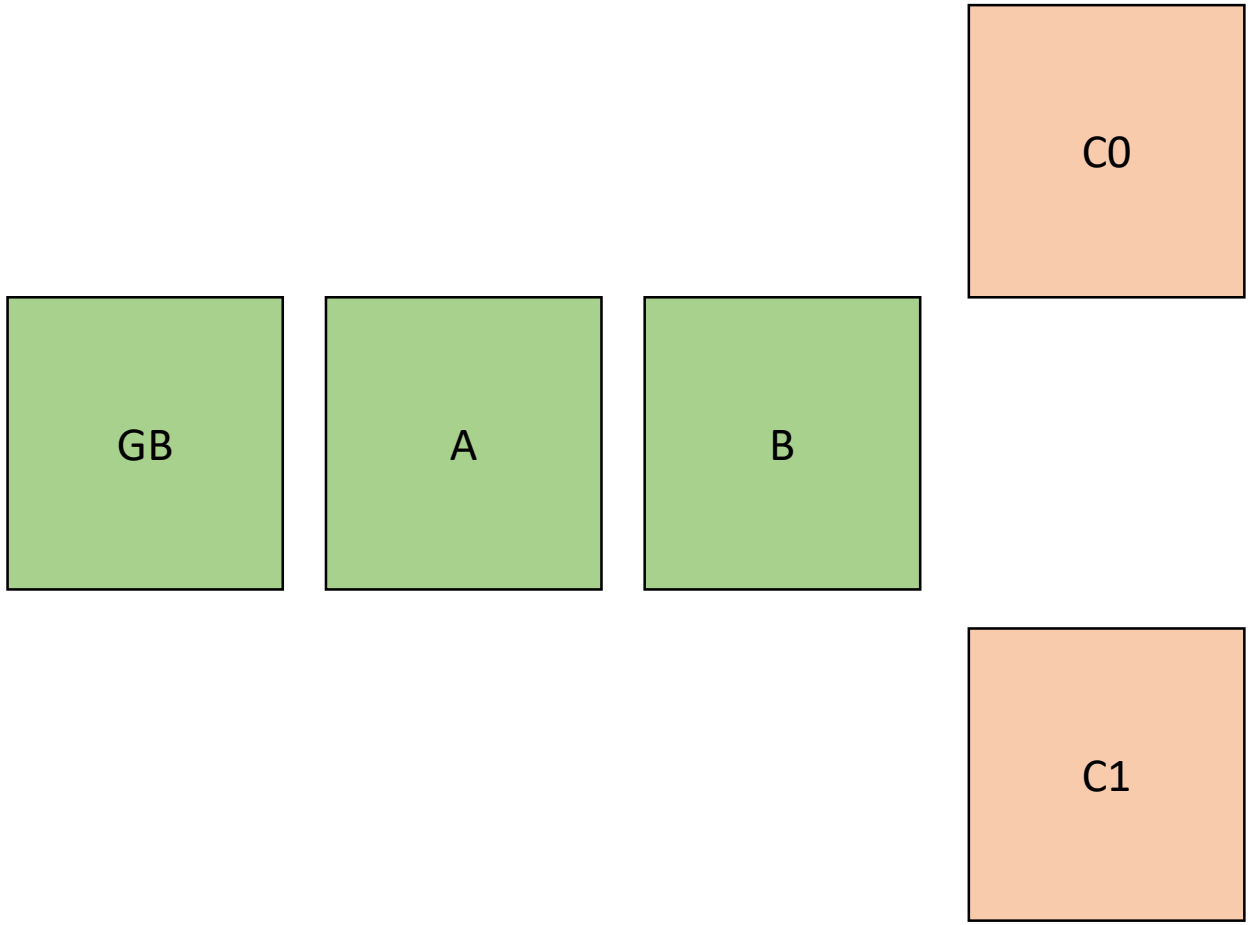
```
//
// The timechain is a tree shaped structure starting with the
// genesis block at the root, with each block potentially having multiple
// candidates to be the next block.  pprev and pnext link a path through the
// main/longest chain.  A blockindex may have multiple pprev pointing back
// to it, but pnext will only point forward to the longest branch, or will
// be null if the block is not part of the longest chain.
//
class CBlockIndex
{
public:
    CBlockIndex* pprev;
    CBlockIndex* pnext;
    unsigned int nFile;
    unsigned int nBlockPos;
    int nHeight;
```

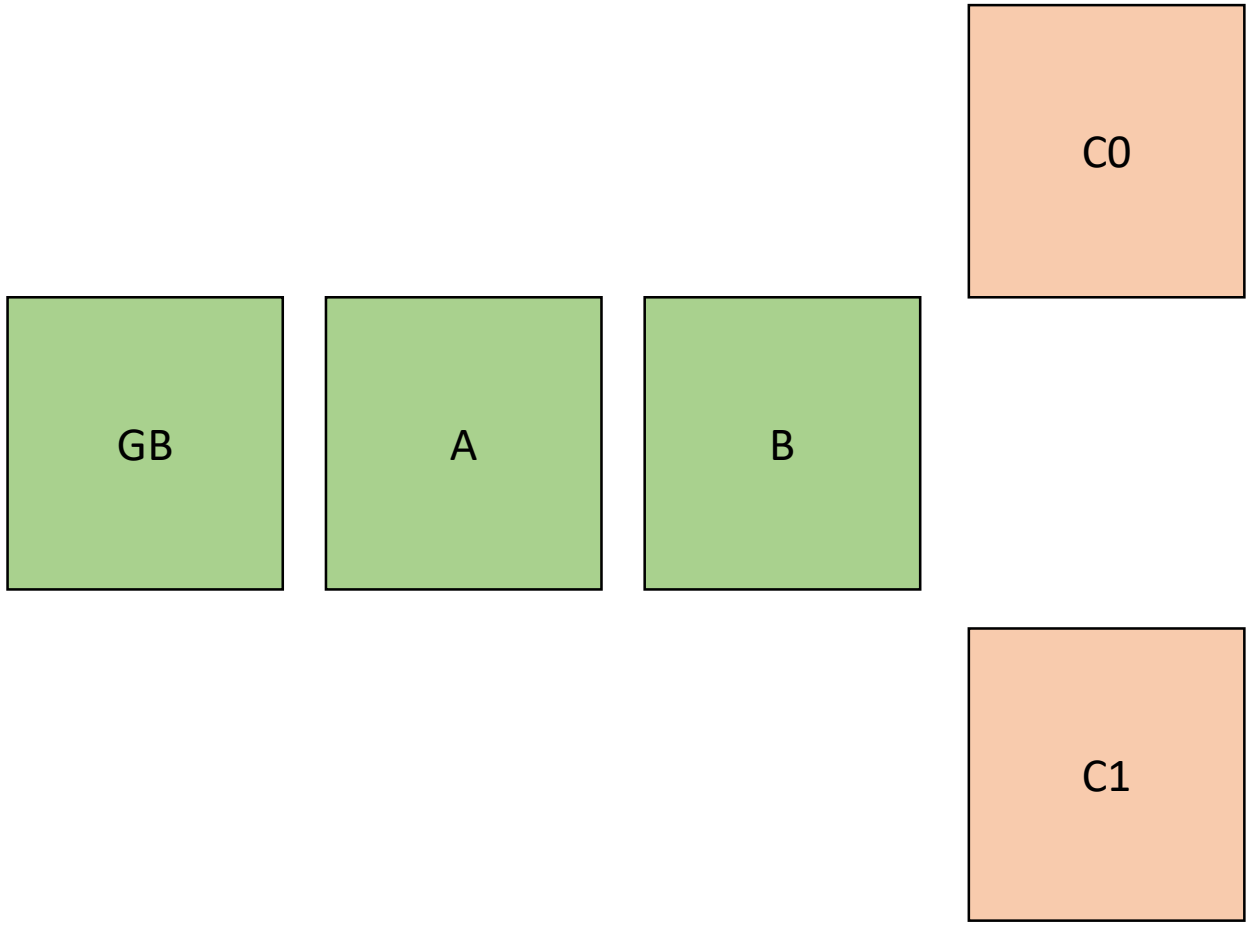
Pre-release Bitcoin source code, Nov. 2008


GB


GB
A


GB
A
B

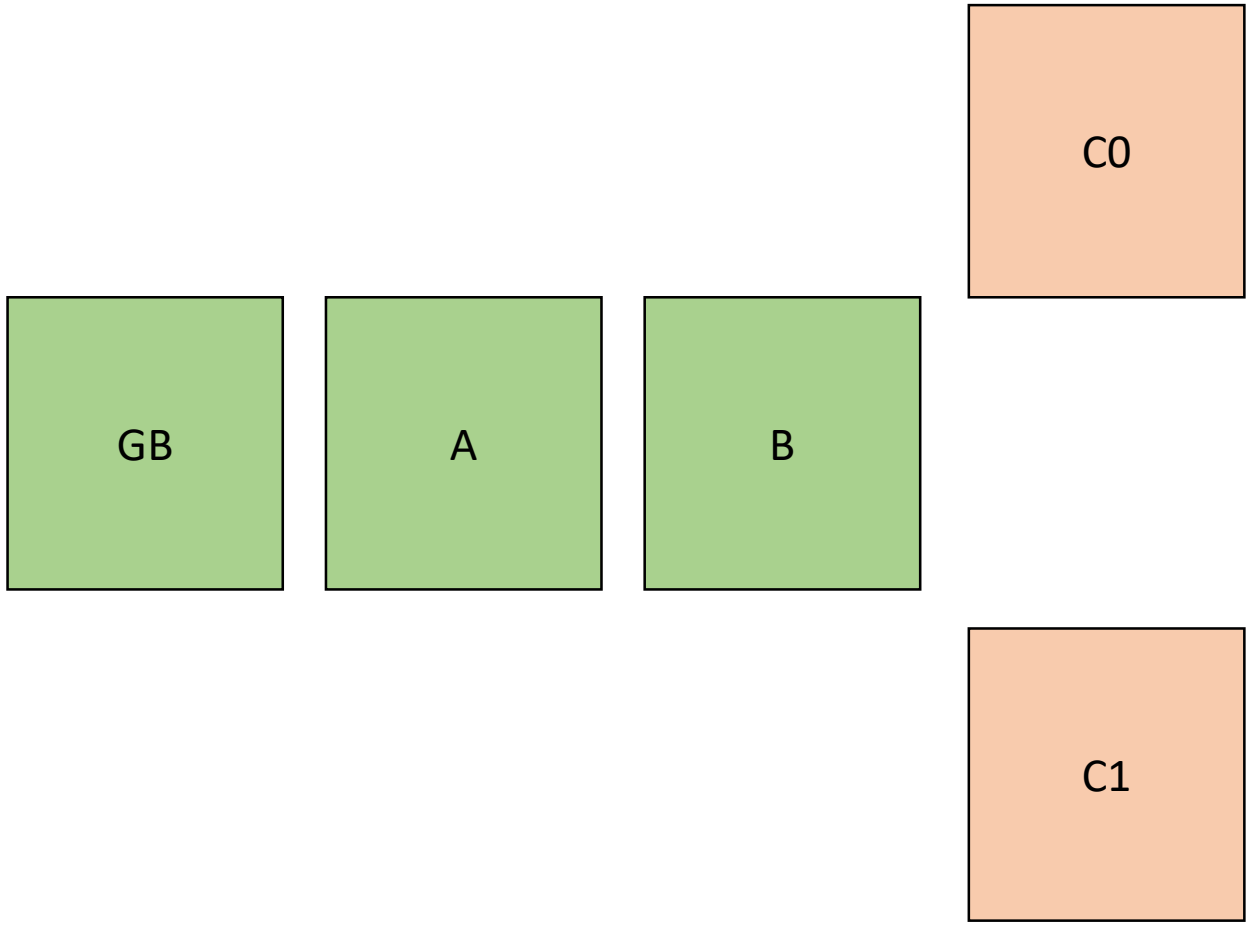

C0
GB
A
B
C1



Nakamoto-style consensus can fork!

Need a way to decide which of the branches is the “main” one

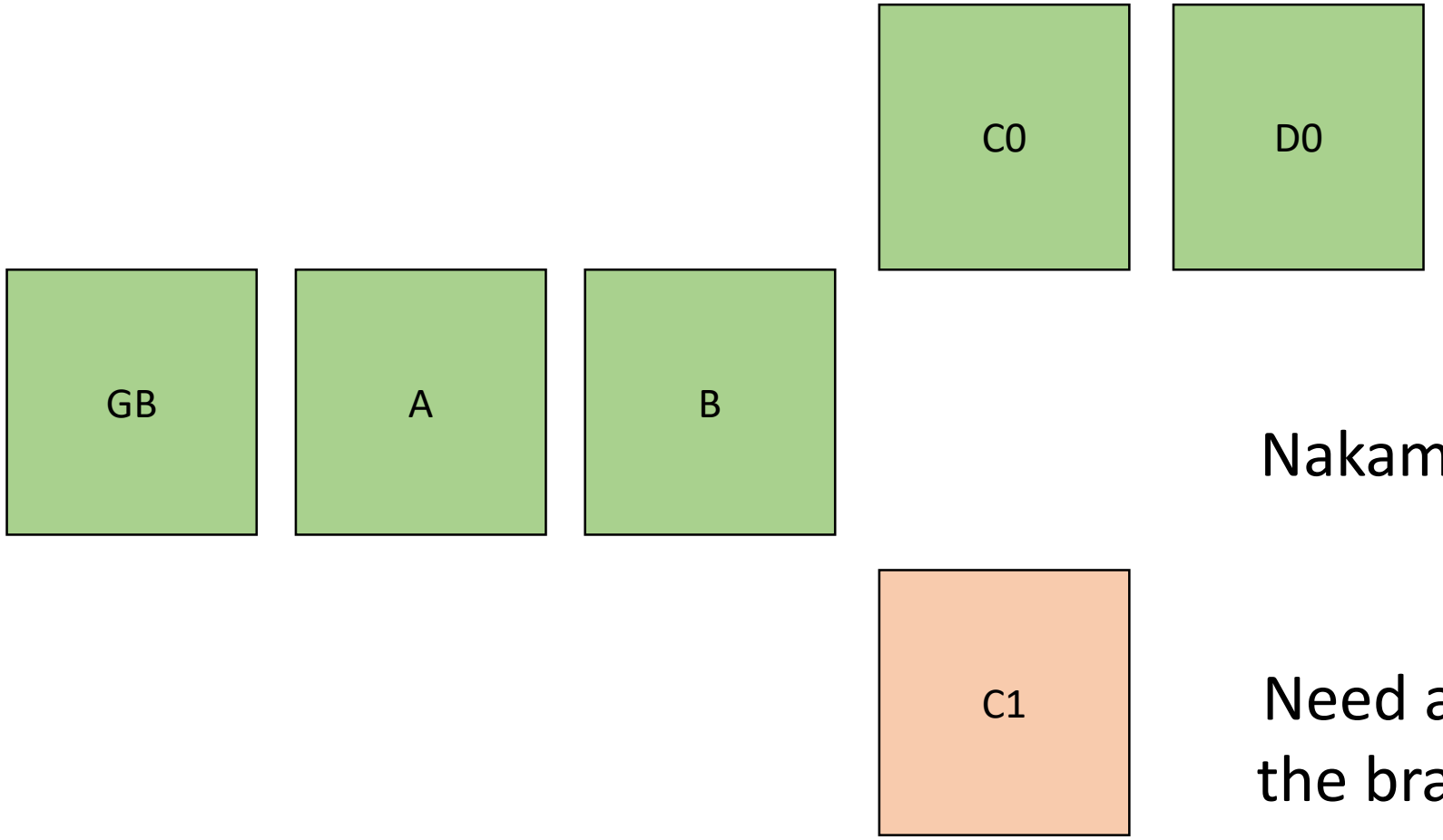
➔ fork choice rule



Nakamoto-style consensus can fork!

Need a way to decide which of the branches is the “main” one

➔ fork choice rule

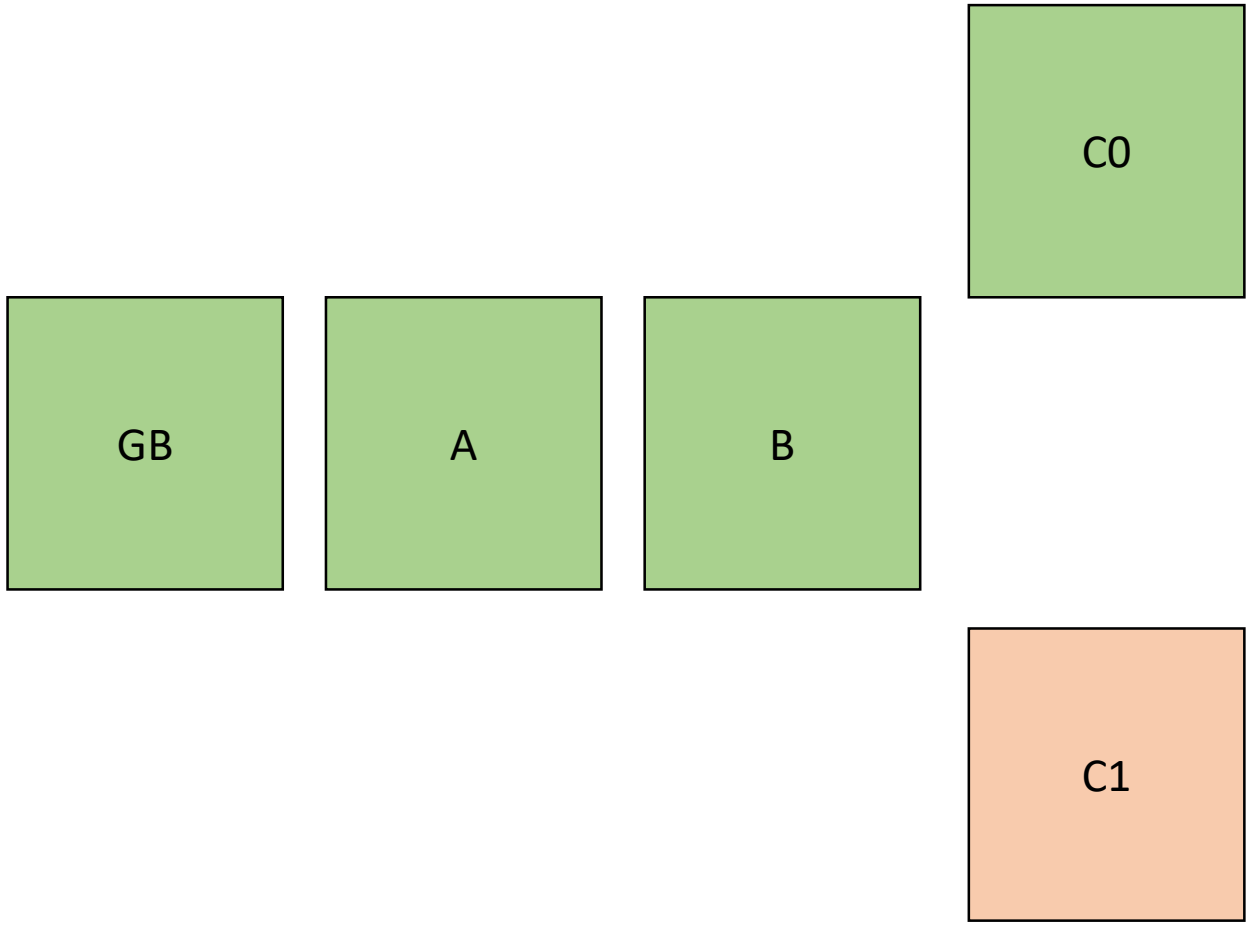


Nakamoto-style consensus can fork!

Need a way to decide which of the branches is the “main” one

➔ fork choice rule

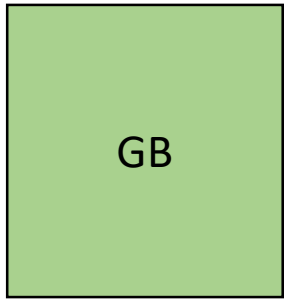
“most accumulated proof-of-work”

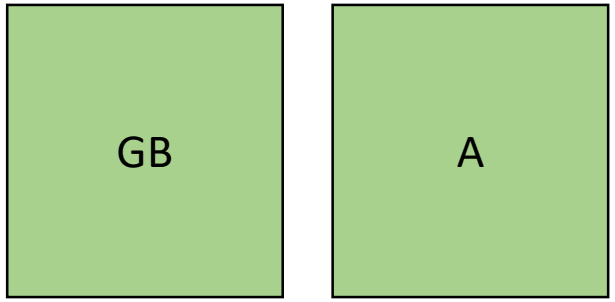


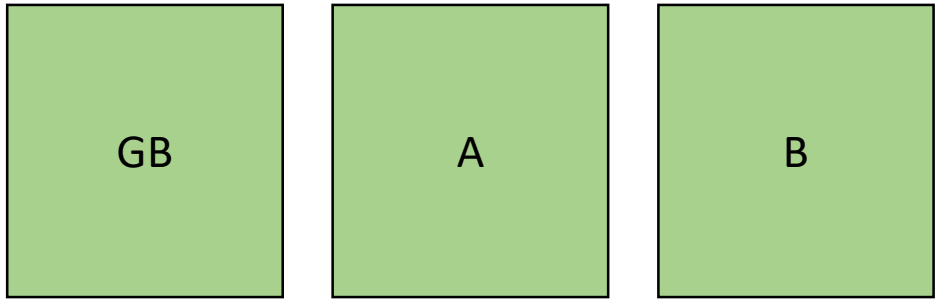
Nakamoto-style consensus can fork!

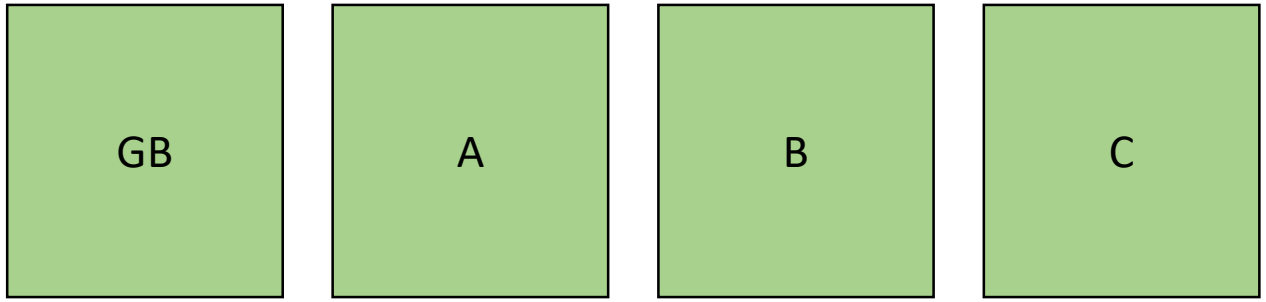
Need a way to decide which of the branches is the “main” one

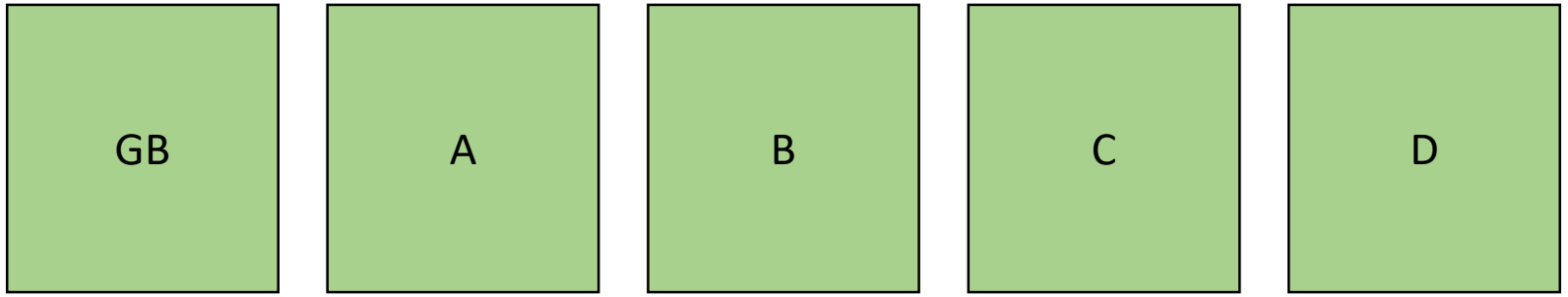
➔ fork choice rule

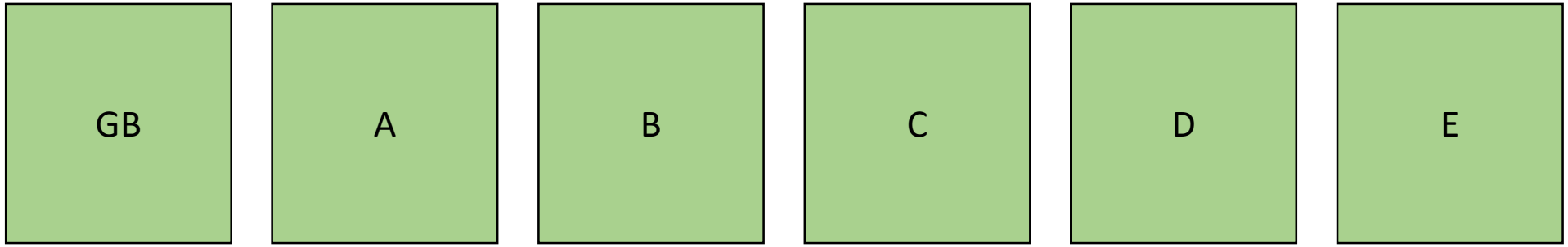

GB

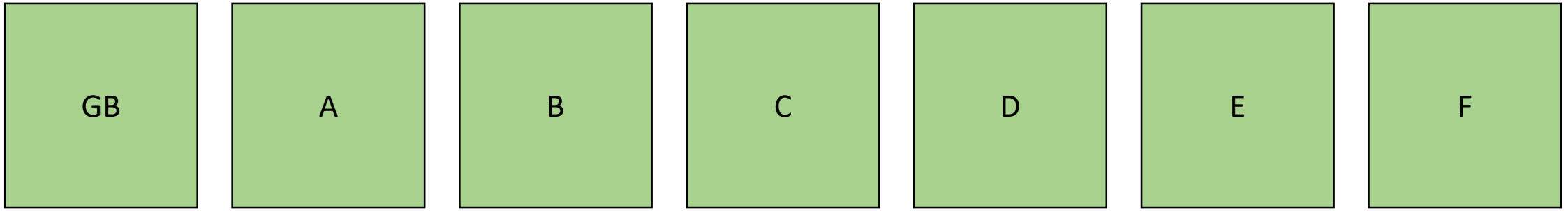

GB
A

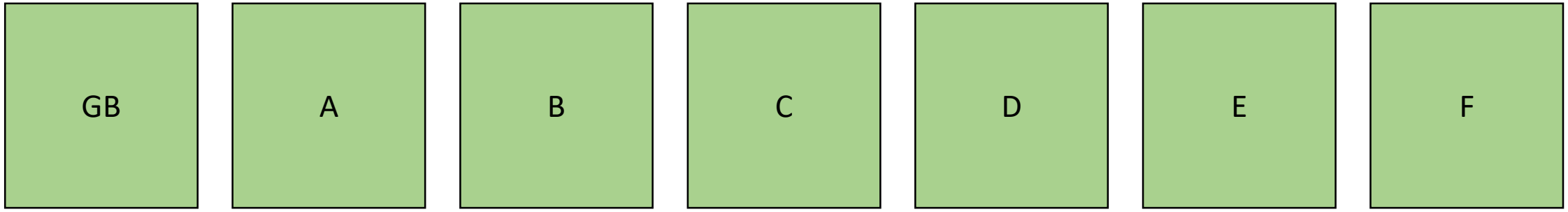

GB
A
B


GB
A
B
C


GB
A
B
C
D

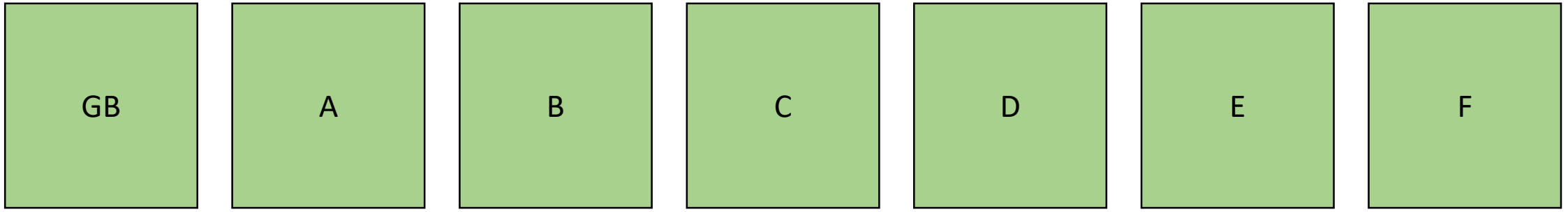

GB
A
B
C
D
E


GB
A
B
C
D
E
F



BFT-style consensus does not fork.
The protocol is inherently synchronized.

Still need a way to choose which participants can create blocks
→ validator acceptance function



BFT-style consensus does not fork.
The protocol is inherently synchronized.

Still need a way to choose which participants can create blocks
→ validator acceptance function

Bitcoin has this too:
it's the proof-of-work!

Toychain formalises Nakamoto-style consensus.

Nakamoto consensus

Blocks and chains

$$b \in \text{Block} ::= \{\, \texttt{prev} : \text{Hash};\ \texttt{txs} : \text{Tx}^*;\ \texttt{pf} : \text{Proof} \,\}$$

$$c \in \text{Chain} \triangleq \text{Block}^*$$

Blocks and chains

links blocks together

$$b \in \text{Block} ::= \{\, \texttt{prev} : \text{Hash};\ \texttt{txs} : \text{Tx}^*;\ \texttt{pf} : \text{Proof} \,\}$$

$$c \in \text{Chain} \triangleq \text{Block}^*$$

Blocks and chains

links blocks together

$hash_b : \text{Block} \to \text{Hash}$

$$b \in \text{Block} ::= \{ \texttt{prev} : \text{Hash};\ \texttt{txs} : \text{Tx}^*;\ \texttt{pf} : \text{Proof} \}$$

$$c \in \text{Chain} \triangleq \text{Block}^*$$

Blocks and chains

$$b \in \text{Block} ::= \{\, \texttt{prev} : \text{Hash};\ \texttt{txs} : \text{Tx}^*;\ \texttt{pf} : \text{Proof} \,\}$$

$$c \in \text{Chain} \triangleq \text{Block}^*$$

Blocks and chains

$$b \in \text{Block} ::= \{\, \texttt{prev} : \text{Hash};\ \texttt{txs} : \text{Tx}^*;\ \texttt{pf} : \text{Proof} \,\}$$

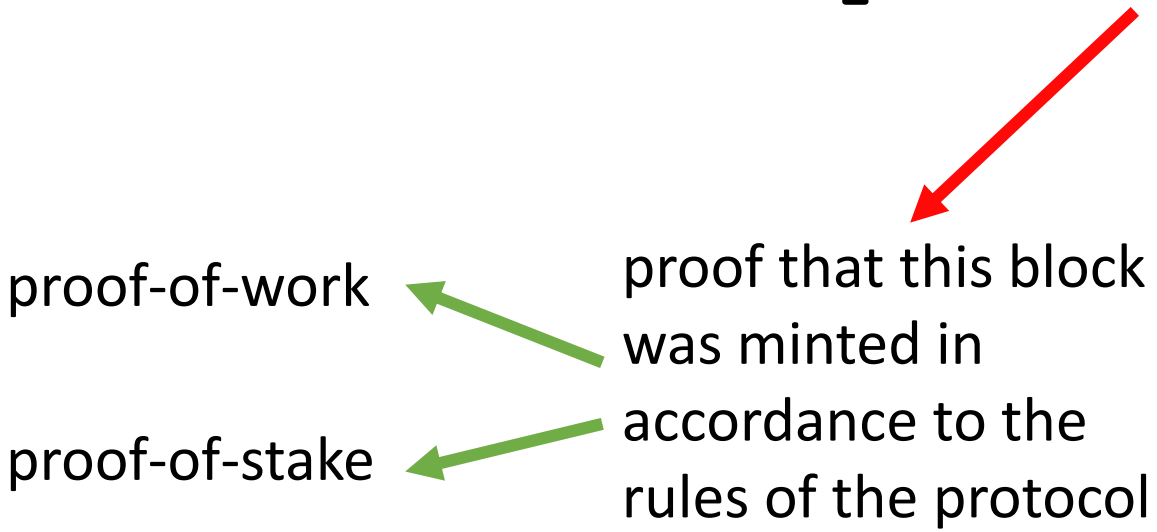
$$c \in \text{Chain} \triangleq \text{Block}^*$$

proof that this block was minted in accordance to the rules of the protocol

Blocks and chains

$$b \in \text{Block} ::= \{\, \texttt{prev} : \text{Hash};\ \texttt{txs} : \text{Tx}^*;\ \texttt{pf} : \text{Proof} \,\}$$

$$c \in \text{Chain} \triangleq \text{Block}^*$$



Blocks and chains

$$b \in \text{Block} ::= \{\, \texttt{prev} : \text{Hash};\ \texttt{txs} : \text{Tx}^*;\ \texttt{pf} : \text{Proof} \,\}$$

$$c \in \text{Chain} \triangleq \text{Block}^*$$

$$GB : \text{Block}$$

Minting and verifying

$$mkProof : \text{Chain} \rightarrow \text{Tx}^* \rightarrow \text{option Proof}$$
$$VAF : \text{Block} \rightarrow \text{Chain} \rightarrow \text{bool}$$

Minting and verifying

try to generate a proof = “ask the protocol for permission” to mint

$$mkProof : \text{Chain} \rightarrow \text{Tx}^* \rightarrow \text{option Proof}$$

$$VAF : \text{Block} \rightarrow \text{Chain} \rightarrow \text{bool}$$

Minting and verifying

try to generate a proof = “ask the protocol for permission” to mint

$$mkProof : \text{Chain} \rightarrow \text{Tx}^* \rightarrow \text{option Proof}$$

$$VAF : \text{Block} \rightarrow \text{Chain} \rightarrow \text{bool}$$

validate a proof = ensure protocol rules were followed

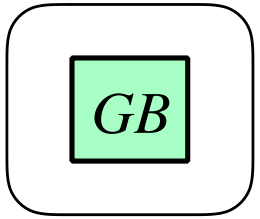
Resolving conflict

$$FCR : \mathrm{Chain} \to \mathrm{Chain} \to \mathrm{bool}$$

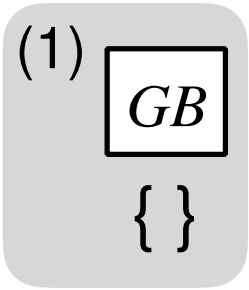
Toychain model

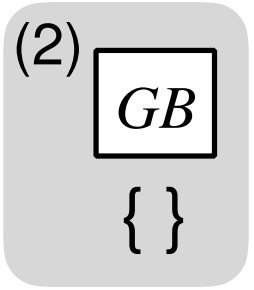
```
//
// The timechain is a tree shaped structure starting with the
// genesis block at the root, with each block potentially having multiple
// candidates to be the next block.  pprev and pnext link a path through the
// main/longest chain.  A blockindex may have multiple pprev pointing back
// to it, but pnext will only point forward to the longest branch, or will
// be null if the block is not part of the longest chain.
//
class CBlockIndex
{
public:
    CBlockIndex* pprev;
    CBlockIndex* pnext;
    unsigned int nFile;
    unsigned int nBlockPos;
    int nHeight;
```

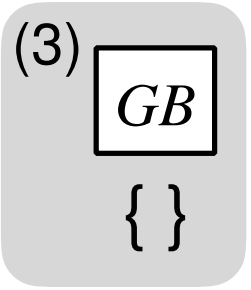
Pre-release Bitcoin source code, Nov. 2008



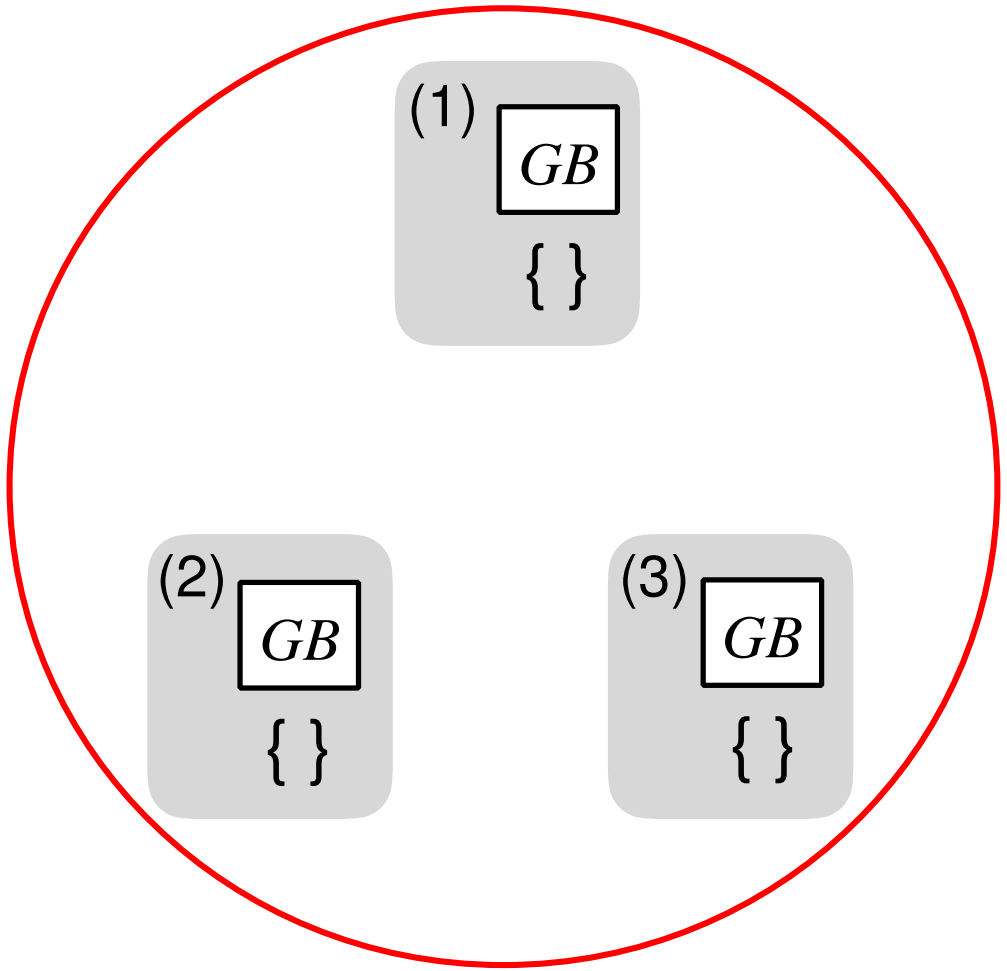
- **distributed**
  - multiple <u>nodes</u>
- all start with same GB

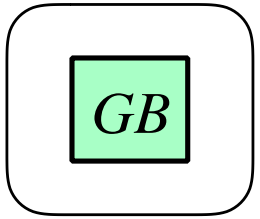






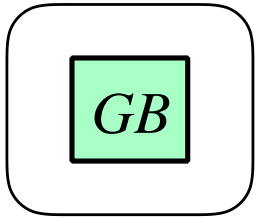
- **distributed**
  - multiple <u>nodes</u>
- all start with same GB

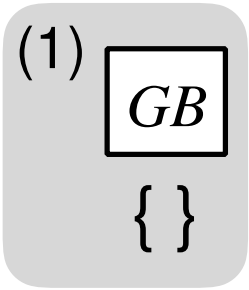




view of all participants' state

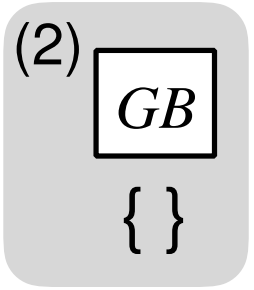
what everyone
eventually *agrees* on

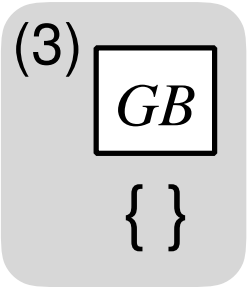


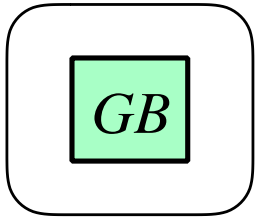


- **distributed**
  - multiple <u>nodes</u>

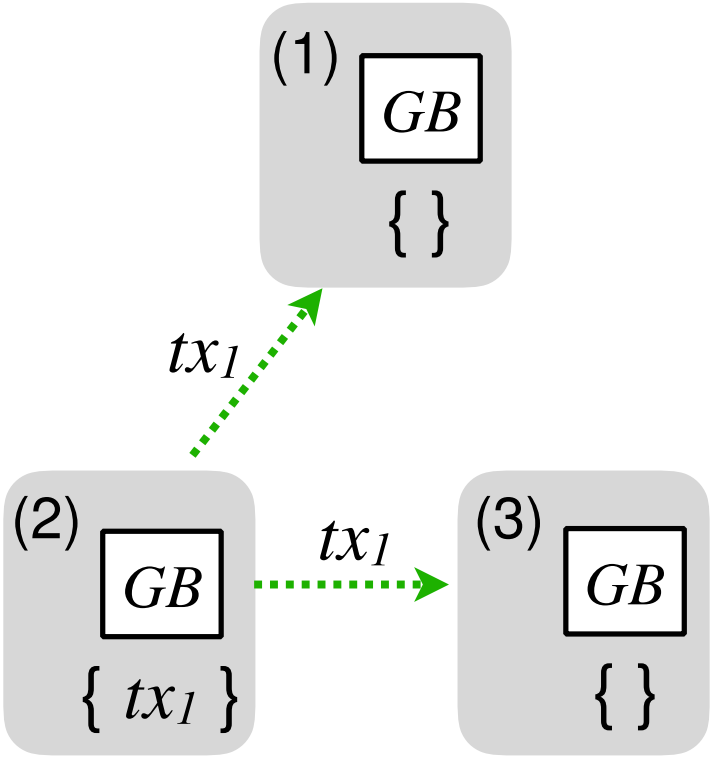
- all start with same GB

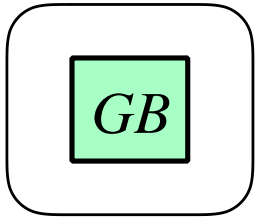




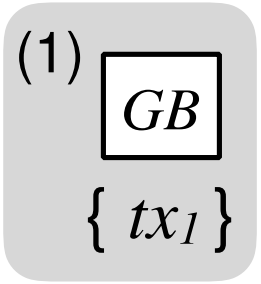


- **distributed**
  - multiple nodes
  - message-passing over a network
- all start with same GB

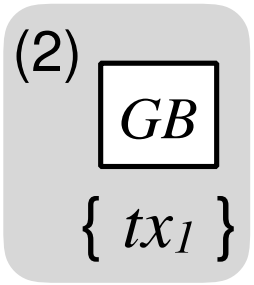


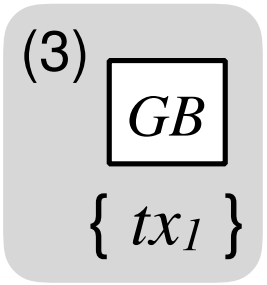


- **distributed**
  - multiple nodes
  - message-passing over a network



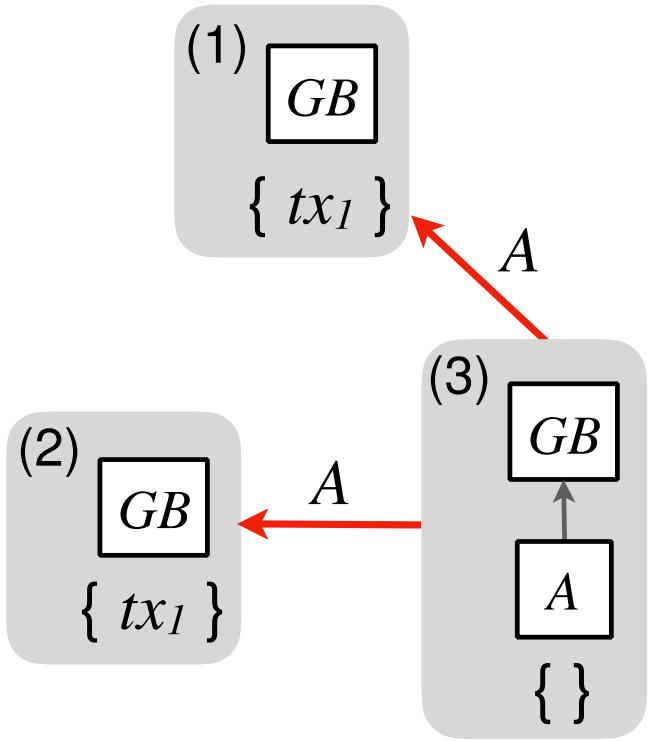
- all start with same GB
- have a transaction pool

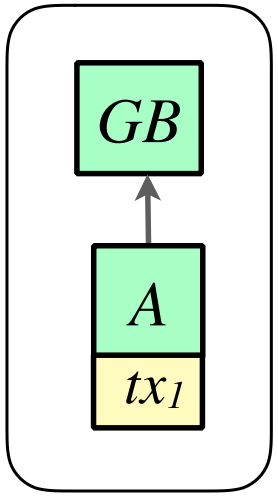




- **distributed**
  - multiple nodes
  - message-passing over a network

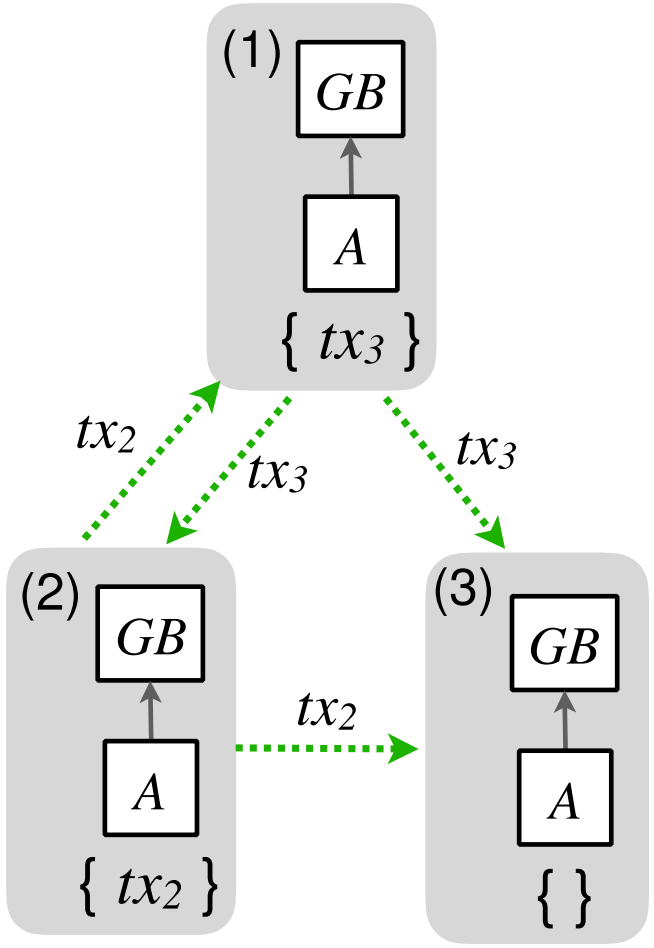
- all start with same GB
- have a transaction pool
- can <u>mint blocks</u>

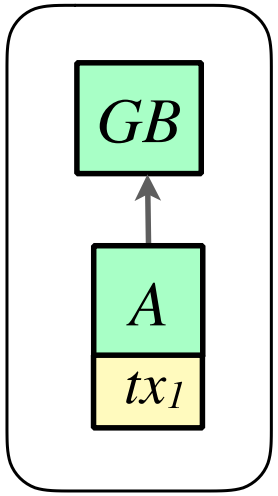




- **distributed** => <u>concurrent</u>
  - multiple nodes
  - message-passing over a network

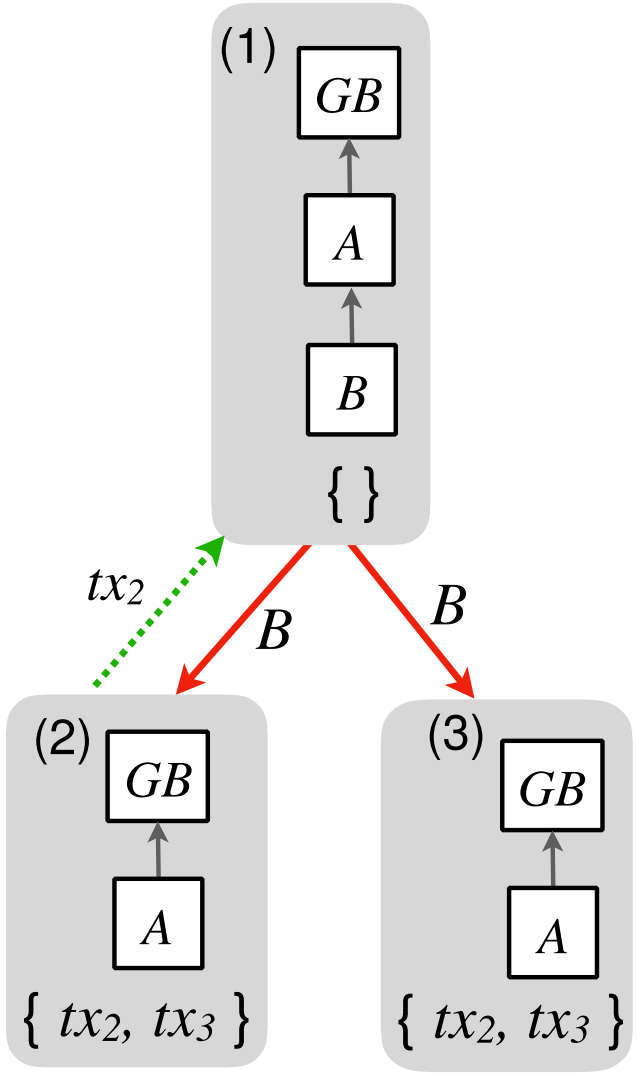
- multiple transactions can be issued and propagated concurrently

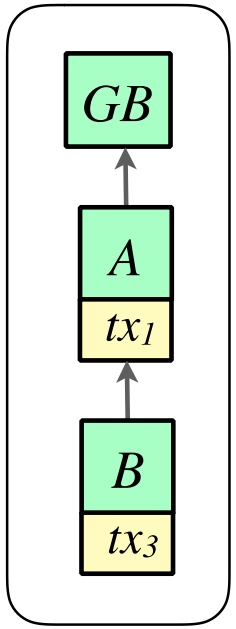




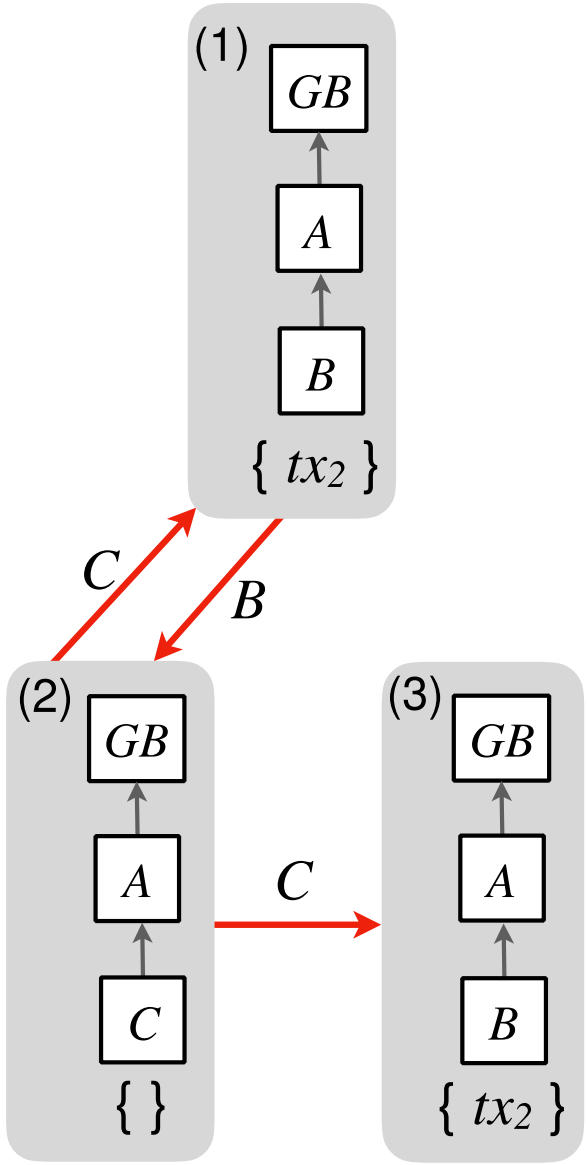
- **distributed** => <u>concurrent</u>
  - multiple nodes
  - message-passing over a network

- blocks can be minted without full knowledge of all transactions

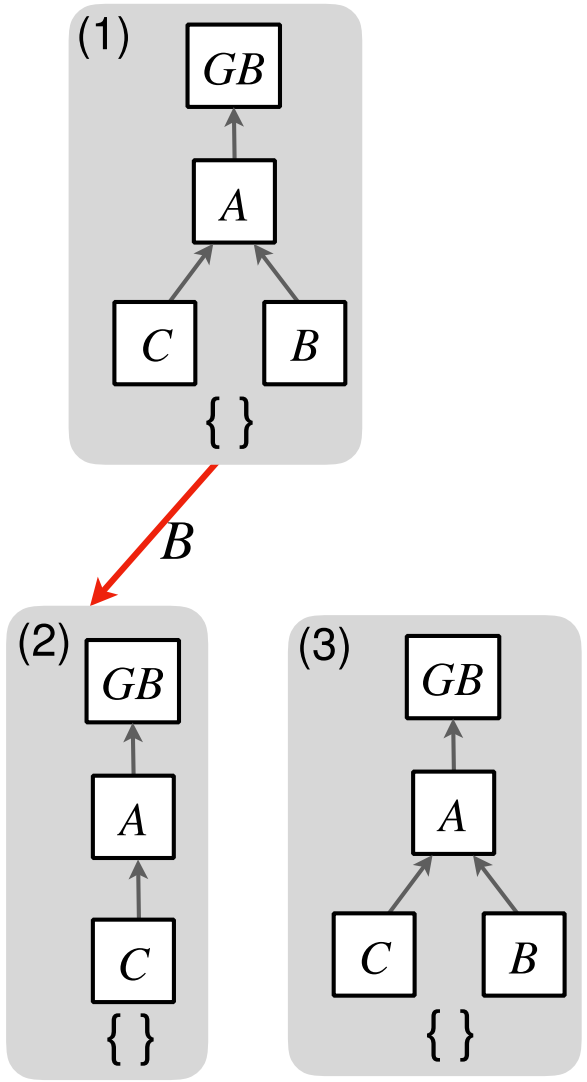




- chain fork has happened, but nodes don't know



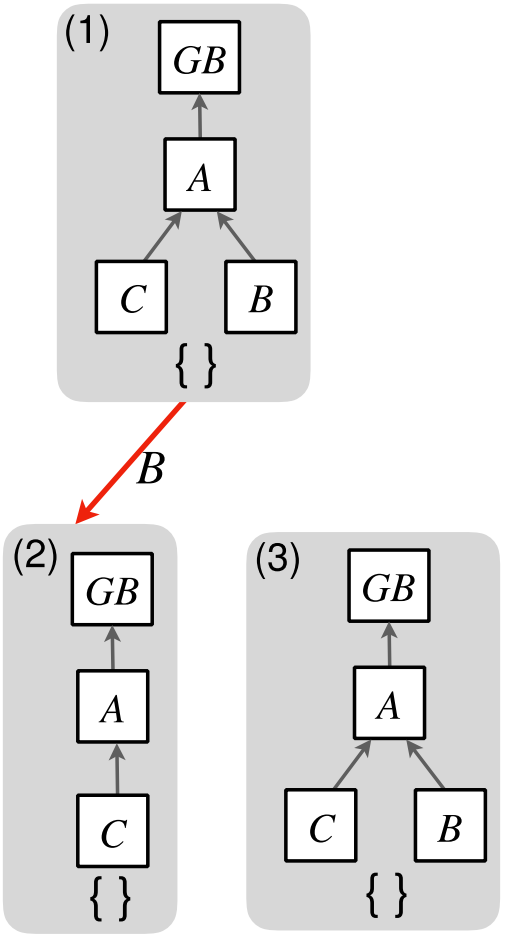
- as block messages propagate, nodes become aware of the fork
- and use the fork choice rule to resolve the conflcit

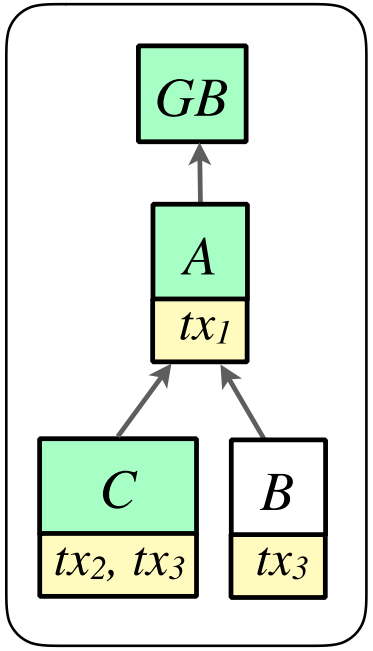


Quiescent consistency

- **distributed**
  - multiple nodes
  - all start with GB
  - message-passing over a network
  - equipped with same FCR

- quiescent consistency: when all block messages have been delivered, everyone agrees

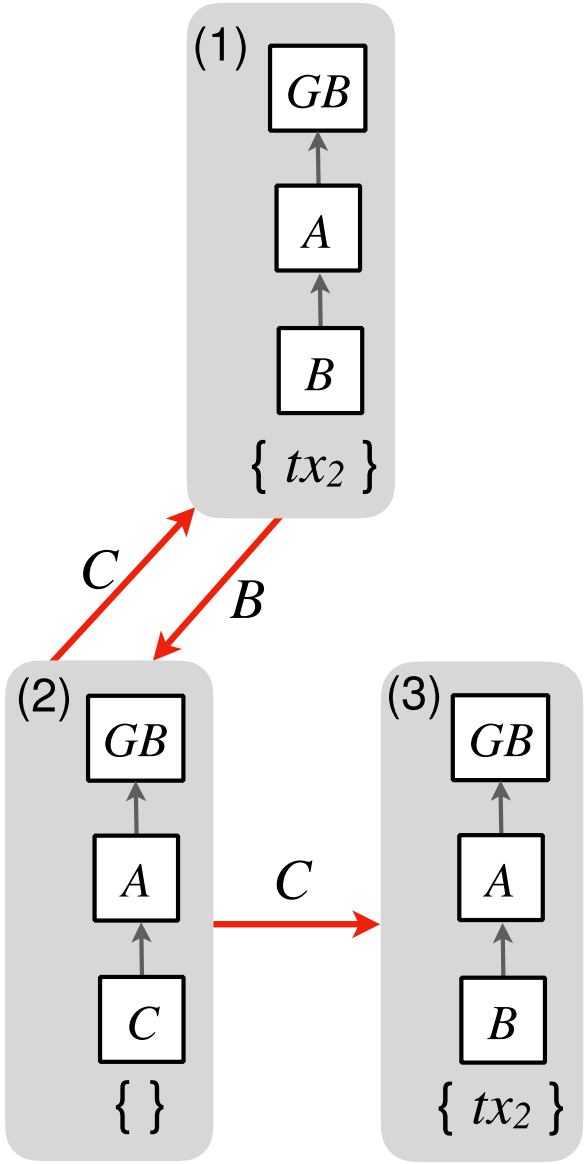




- Every node has same **GB** and same **FCR**

- Adding to block forest is <u>commutative</u>
  - i.e. message delivery order does not matter
  - system invariant: local + “in-flight” = global

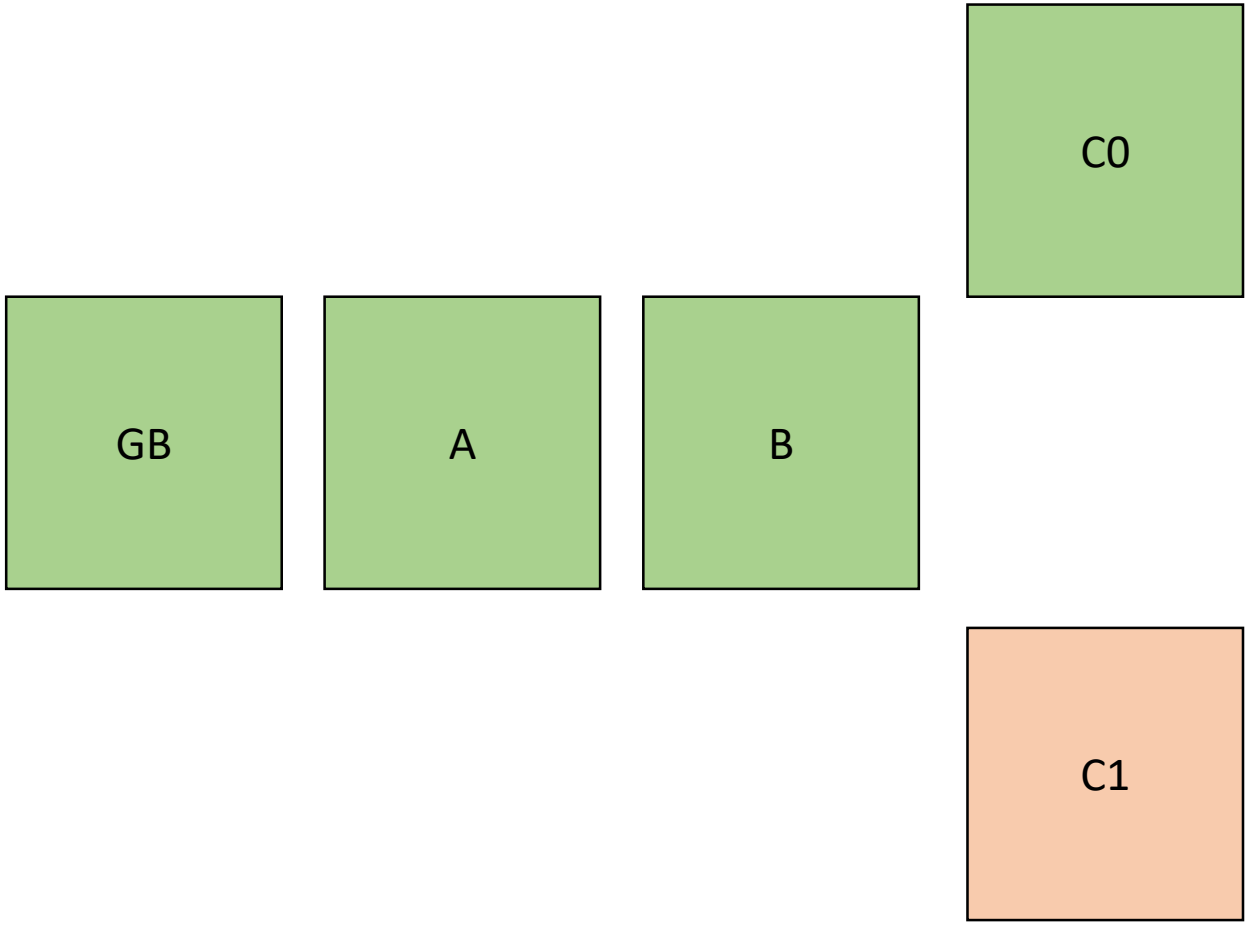
- When all `BlockMsg` delivered, all block forests equal
  - FCR gives same result for all nodes



Assumptions*

- FCR imposes a *strict total order* on all blockchains

$$
\begin{aligned}
FCR\_rel &: \forall c_1\ c_2, c_1 = c_2 \vee c_1 > c_2 \vee c_2 > c_1 \\
FCR\_trans &: \forall c_1\ c_2\ c_3, c_1 > c_2 \wedge c_2 > c_3 \implies c_1 > c_3 \\
FCR\_nrefl &: \forall c, c > c \implies \text{False}
\end{aligned}
$$

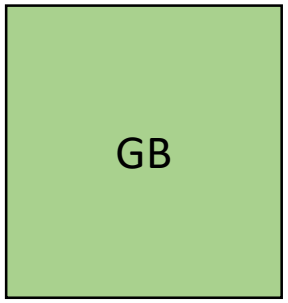

C0
GB
A
B
C1

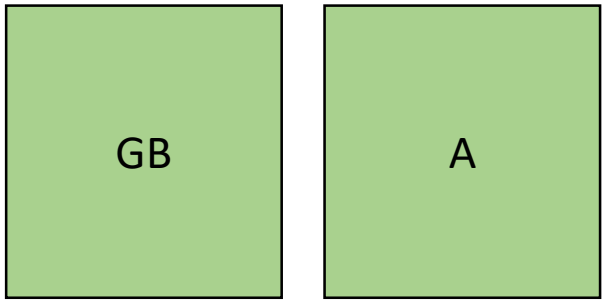
Assumptions*

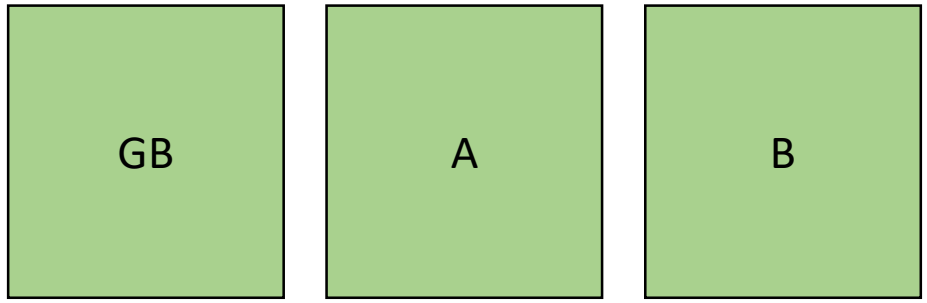
- FCR is *additive*

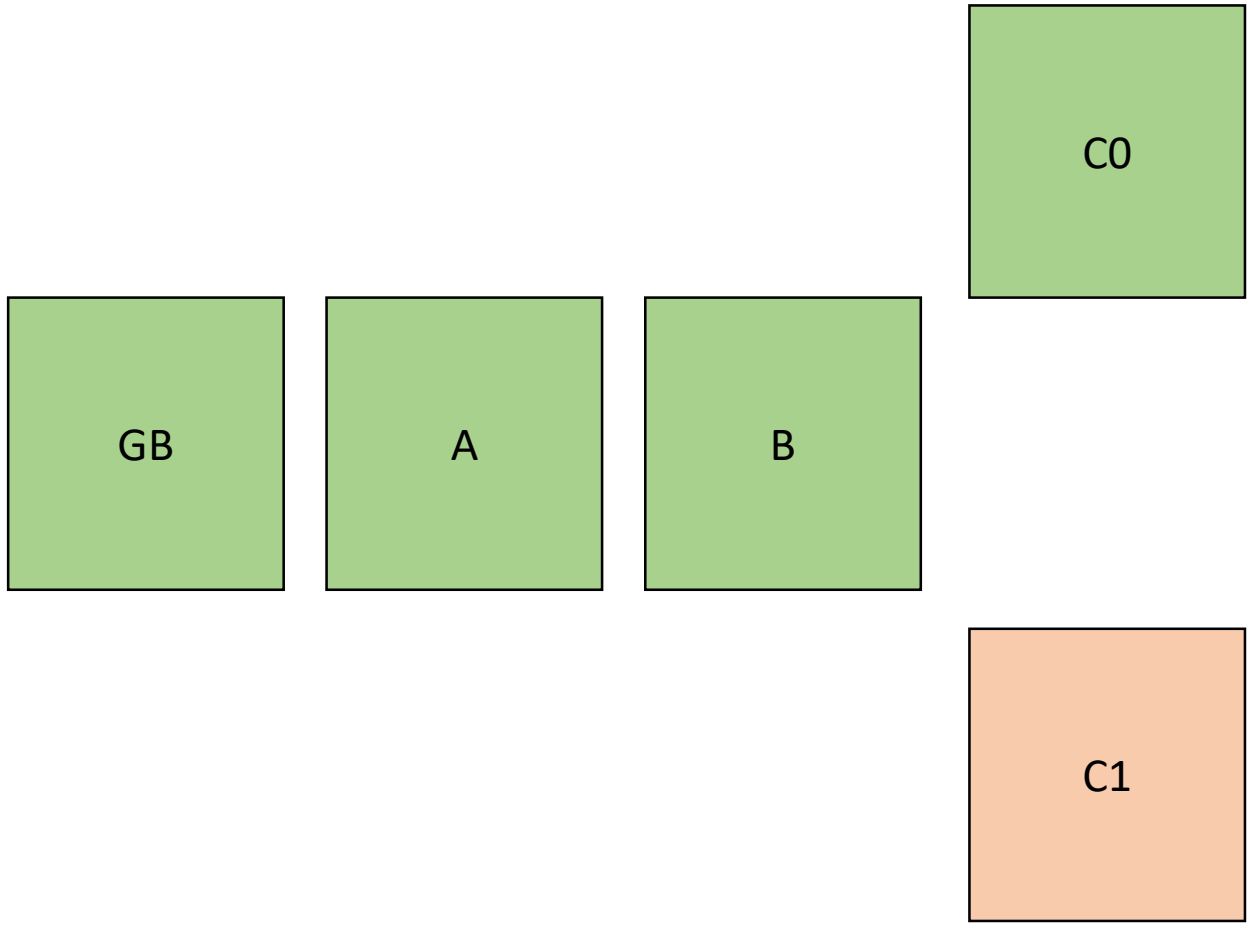
$$FCR\_ext \quad : \ \forall c_1\ c_2\ b, c_1 +\!\!+ (b :: c_2) > c_1$$

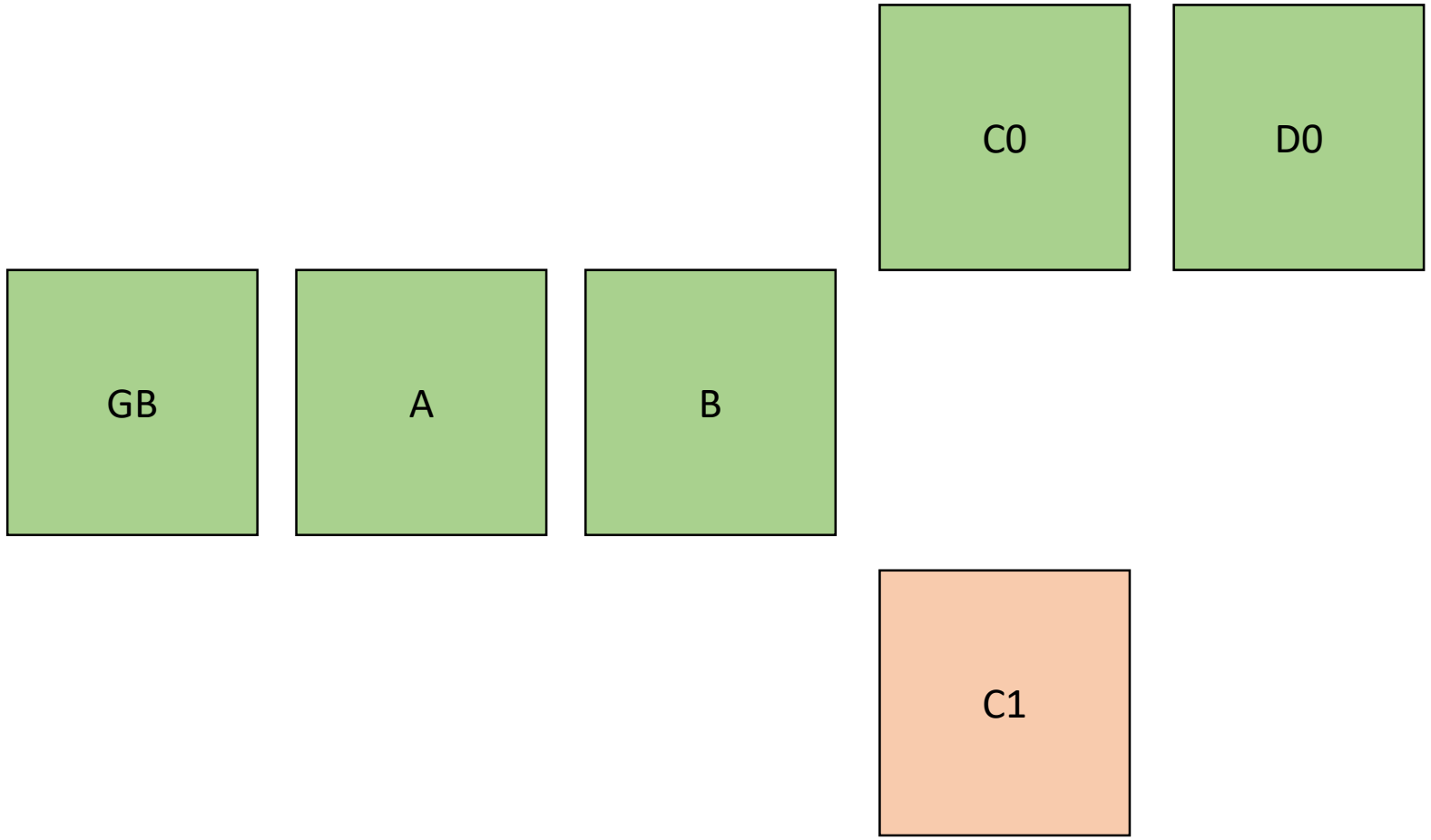
$$FCR\_subch \quad : \ \forall c_1\ c_2, c_1 < c_2 \implies c_2 \geq c_1$$


GB


GB
A

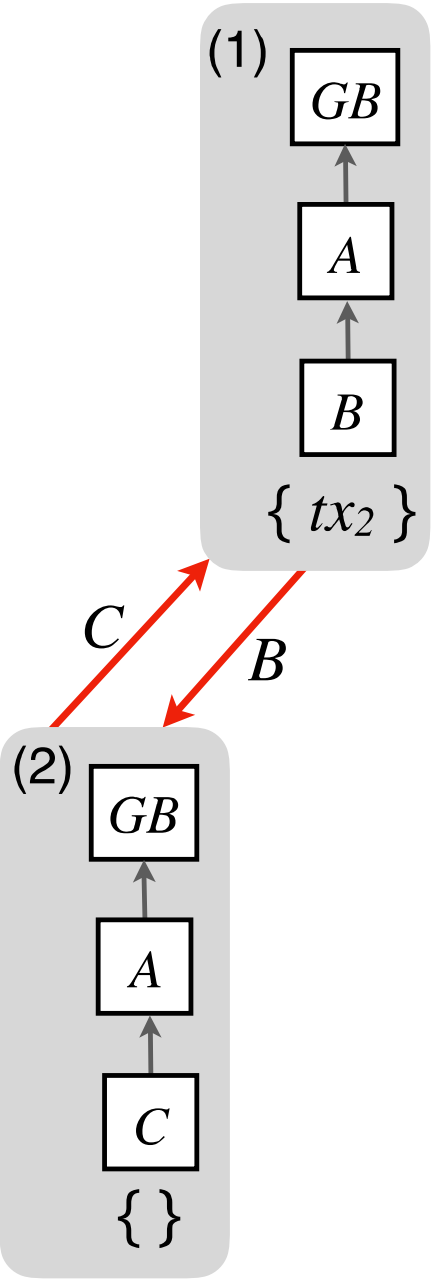

GB
A
B


C0
GB
A
B
C1


C0
D0
GB
A
B
C1

Commutativity under hash collisions?

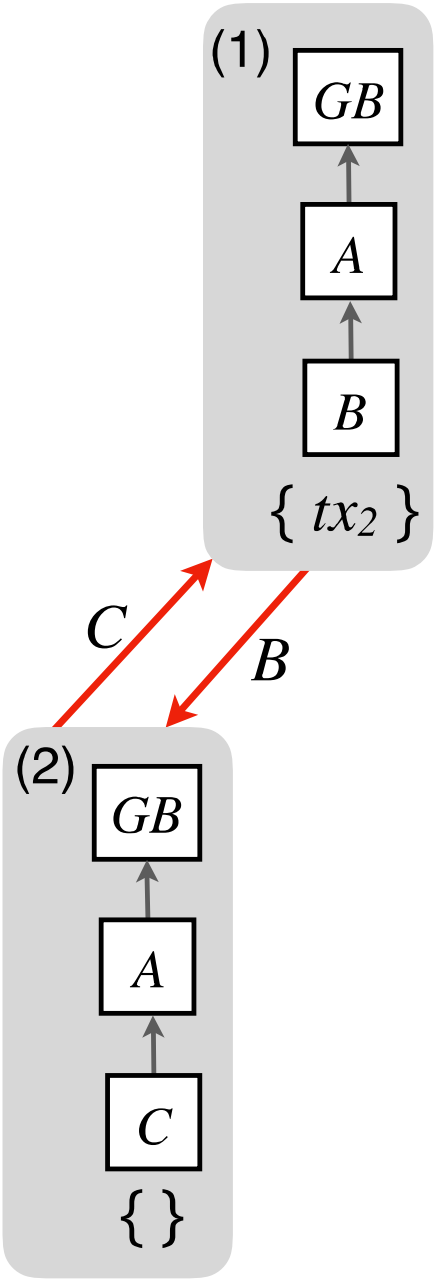
What happens if *hash*(B) = *hash*(C)?



Commutativity under hash collisions?

What happens if *hash*(B) = *hash*(C)?

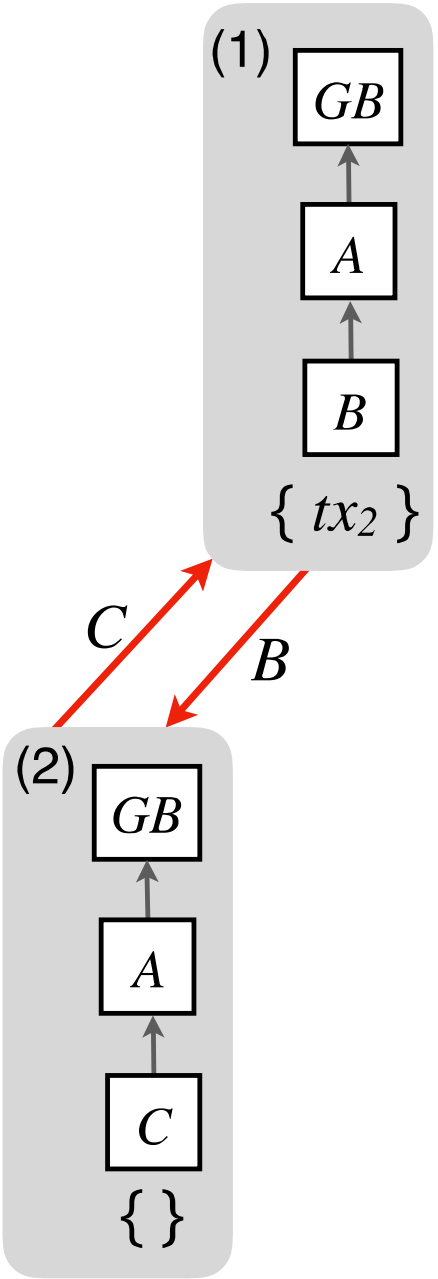
$$hash\_inj \quad : \forall x\ y,\ \#x = \#y \implies x = y$$



Commutativity under hash collisions?

What happens if *hash*(B) = *hash*(C)?

~~$hash\_inj \quad : \forall x\ y,\ \#x = \#y \implies x = y$~~



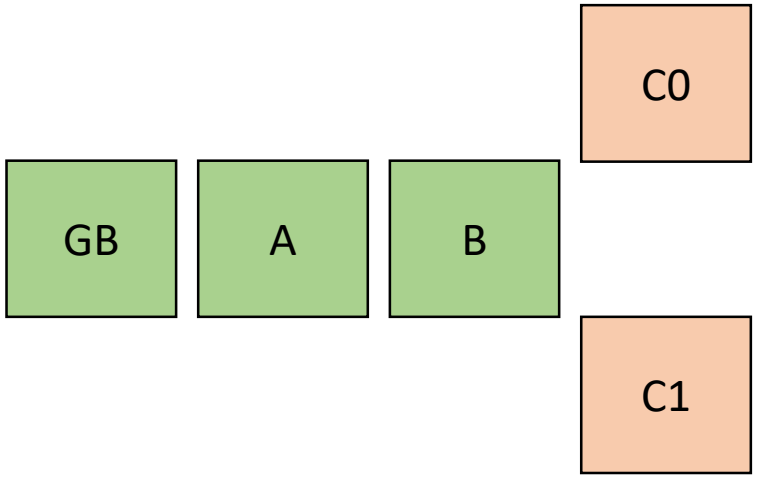
Limitations of the proof

1. Practical **FCR**s are not strict

Bitcoin: two blocks at same height have same weight!

(not true across difficulty-change boundaries)

Ethereum: diff. chains can nonetheless have same <u>total</u> work

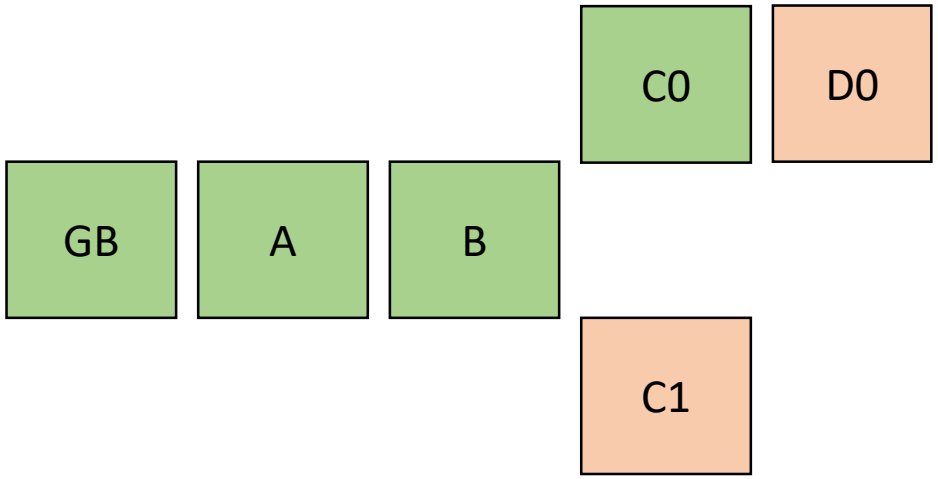


1. Practical **FCR**s are not strict

Bitcoin: two blocks at same height have same weight!

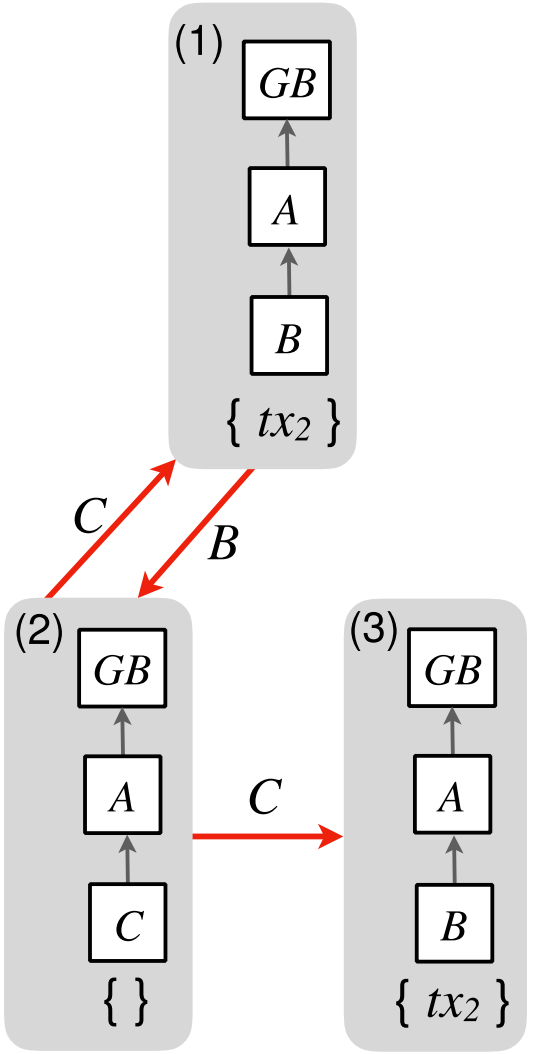
(not true across difficulty-change boundaries)

Ethereum: diff. chains can nonetheless have same <u>total</u> work



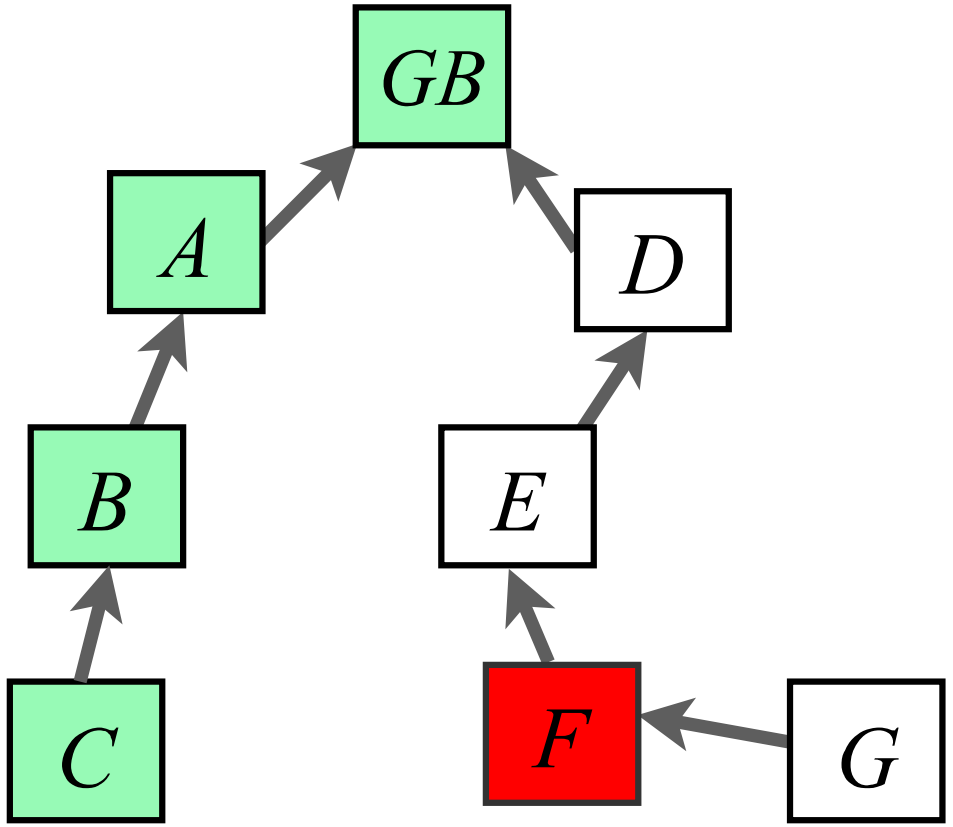
2. `BlockMsg` are delivered over gossip

- need to find a way to abstract gossip mechanism
- difficulty is in finding appropriate abstraction; proof follows trivially



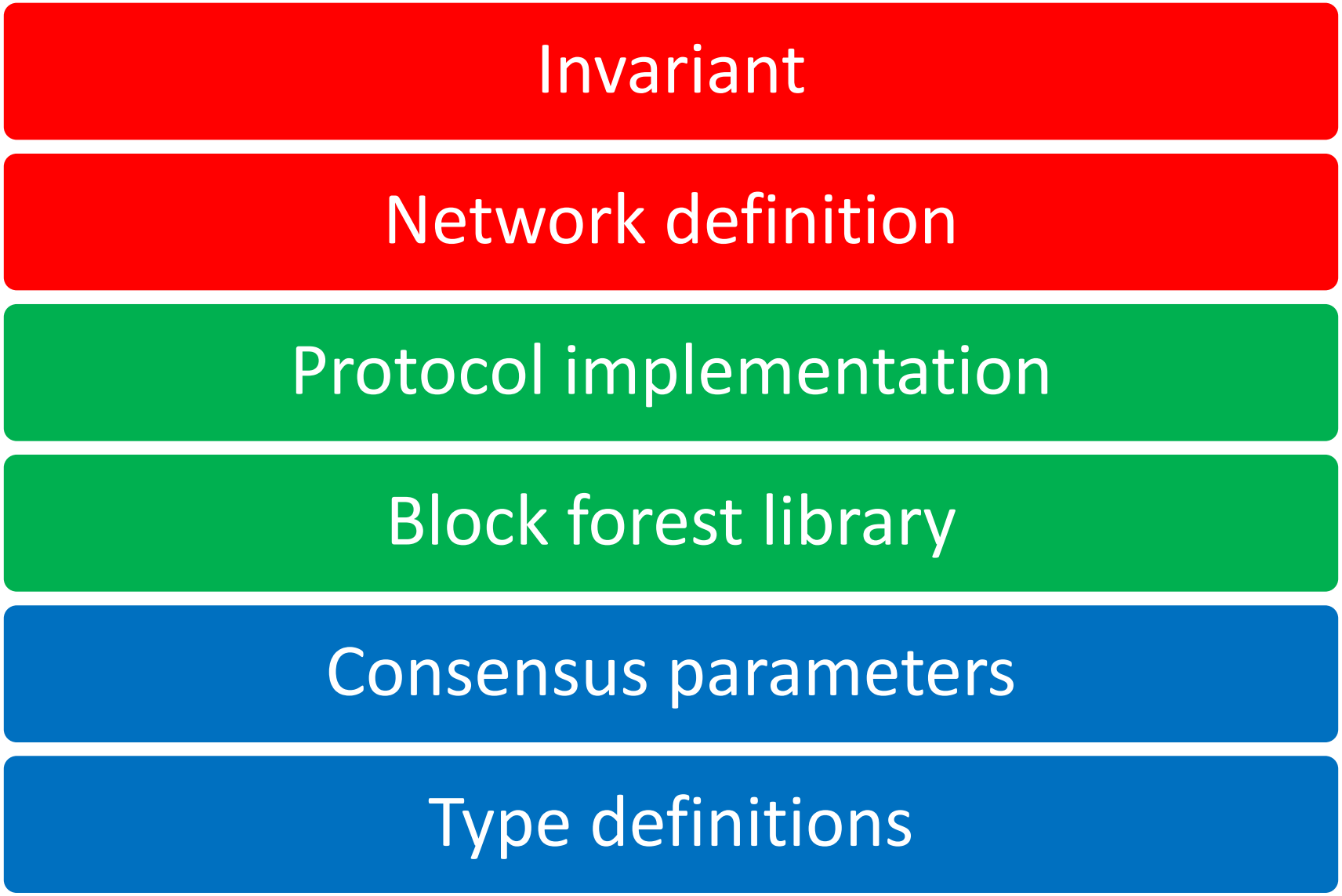
3. Byzantine adversaries can invalidate invariant

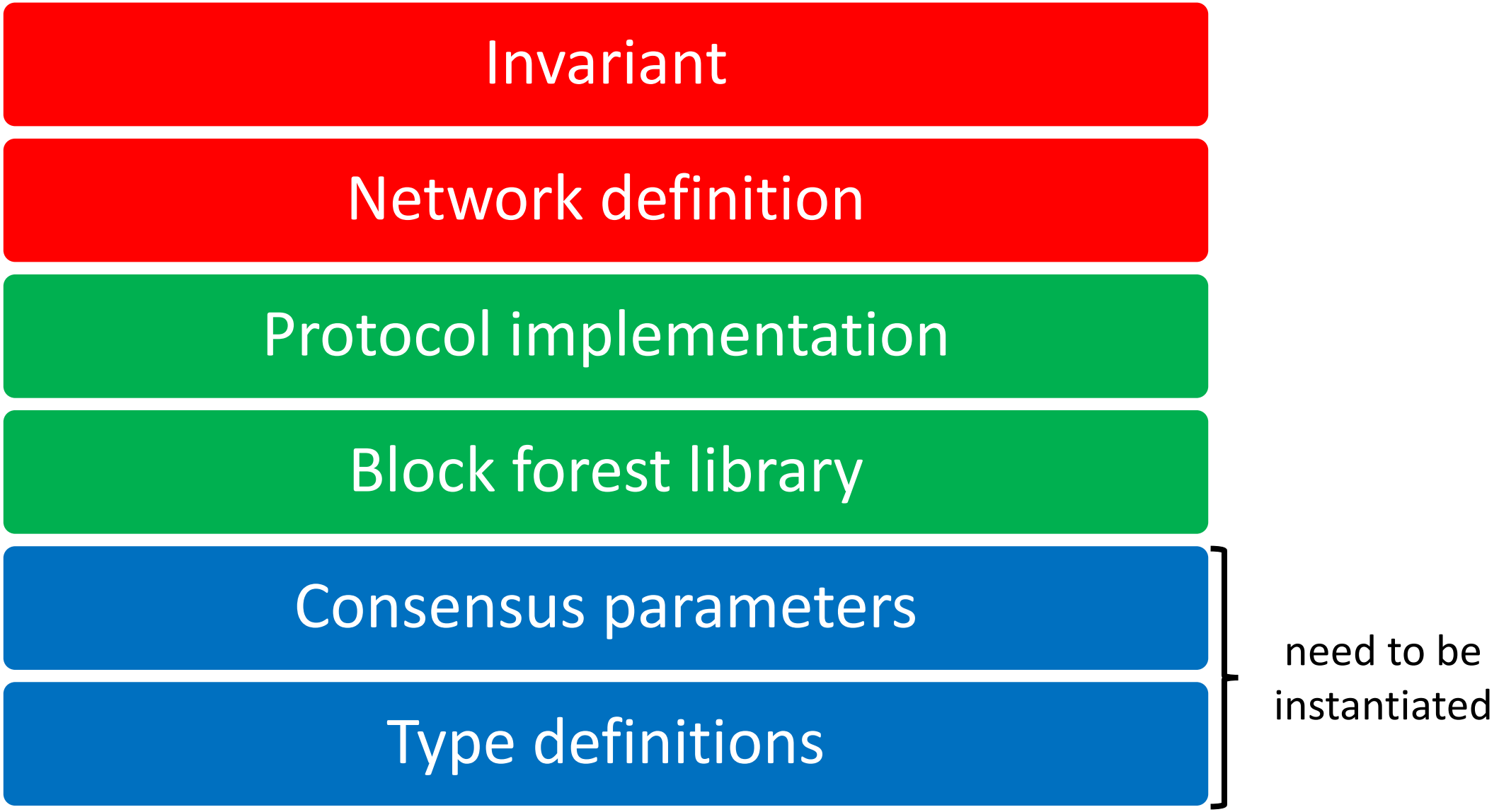
- relies on blocks only being mined at chain tips
  - indistinguishable from honest miners
- true under cryptographic assumptions
  - contrary implies hash prediction

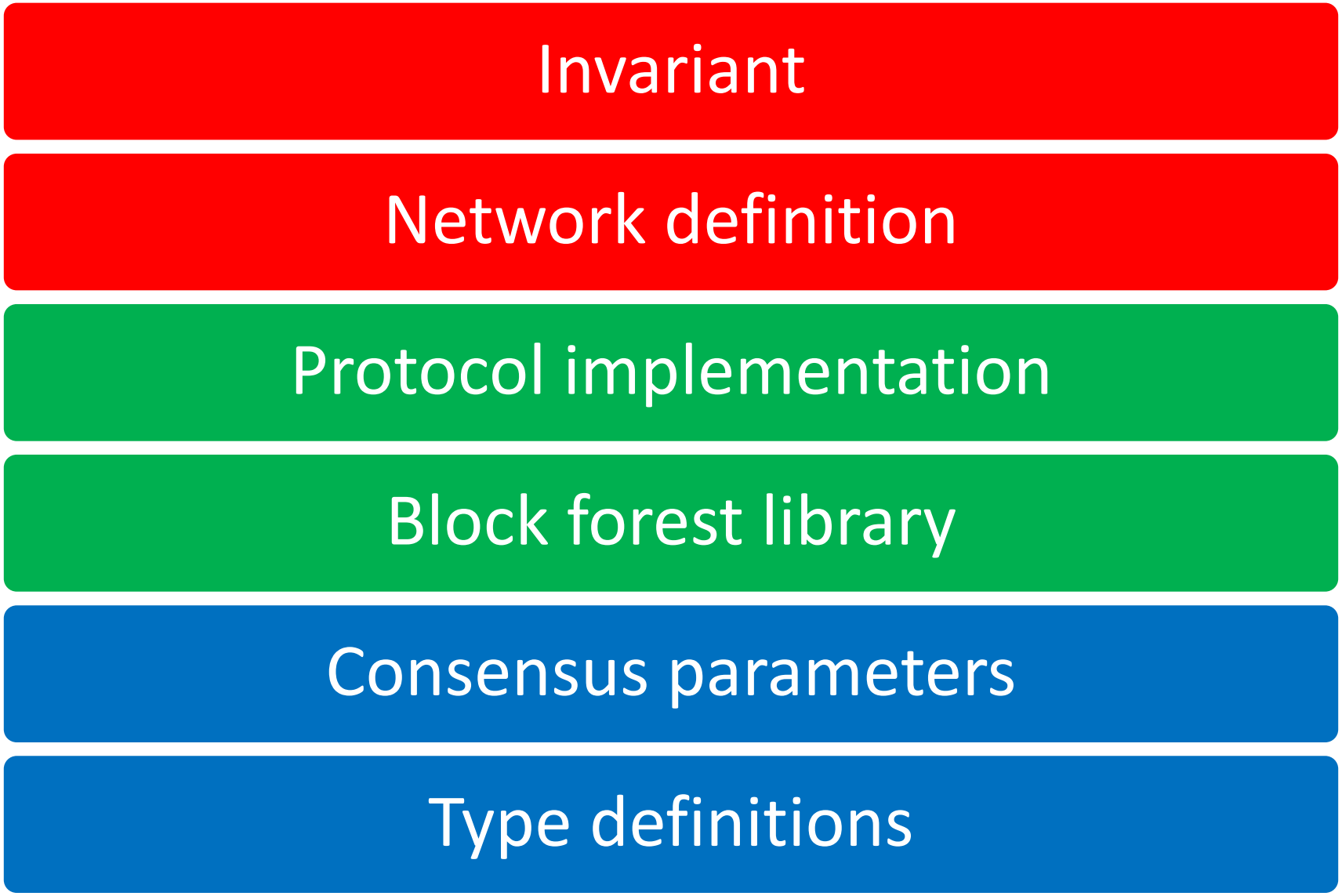


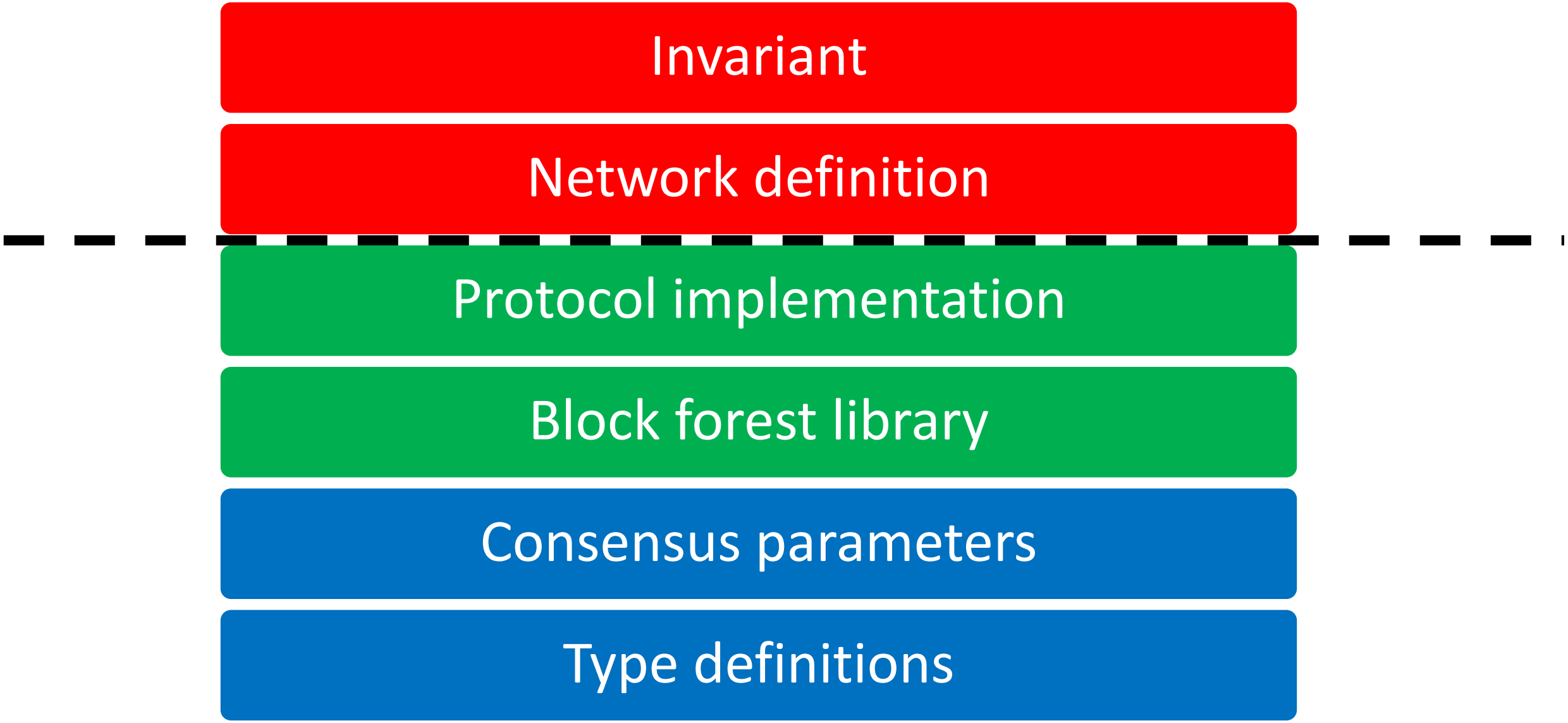
From Proof to Program

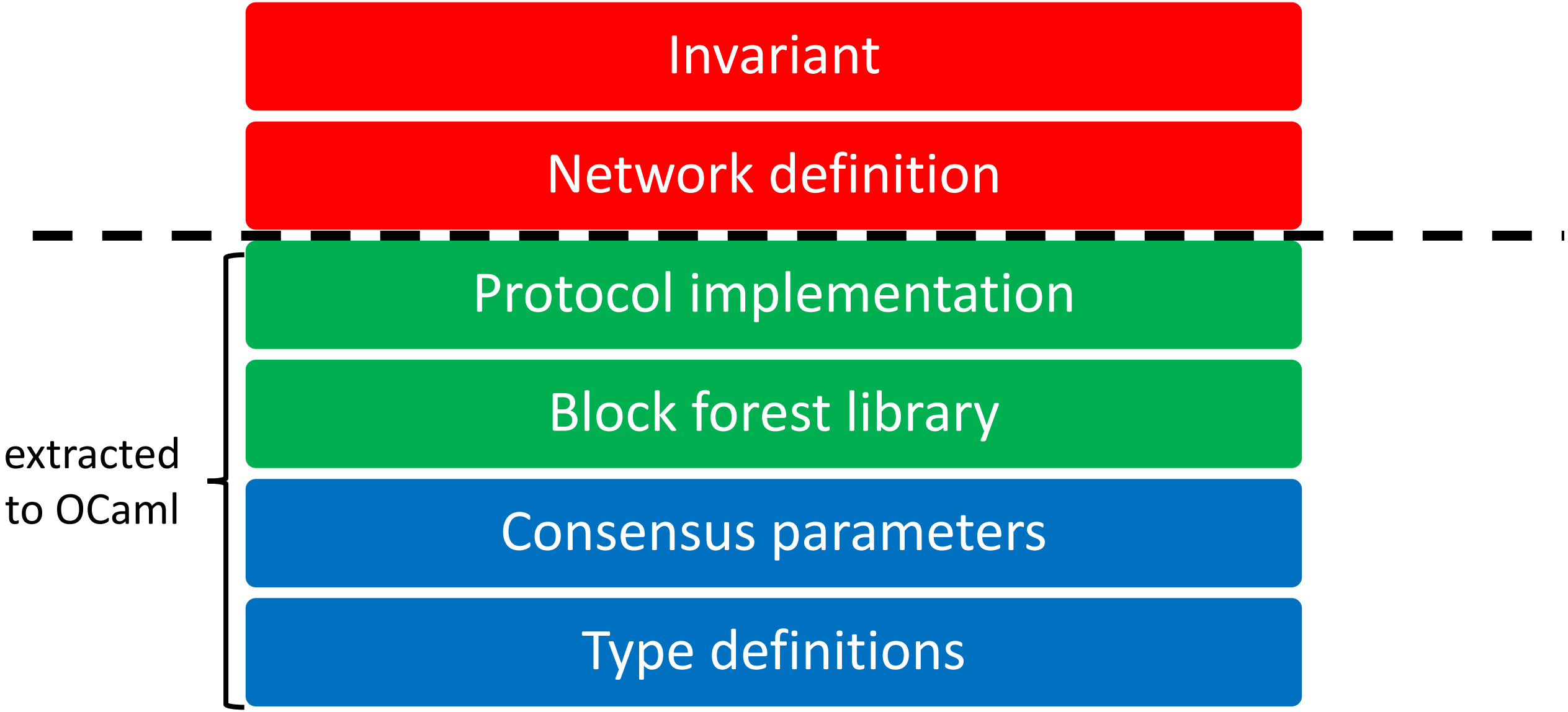
Getting executable code

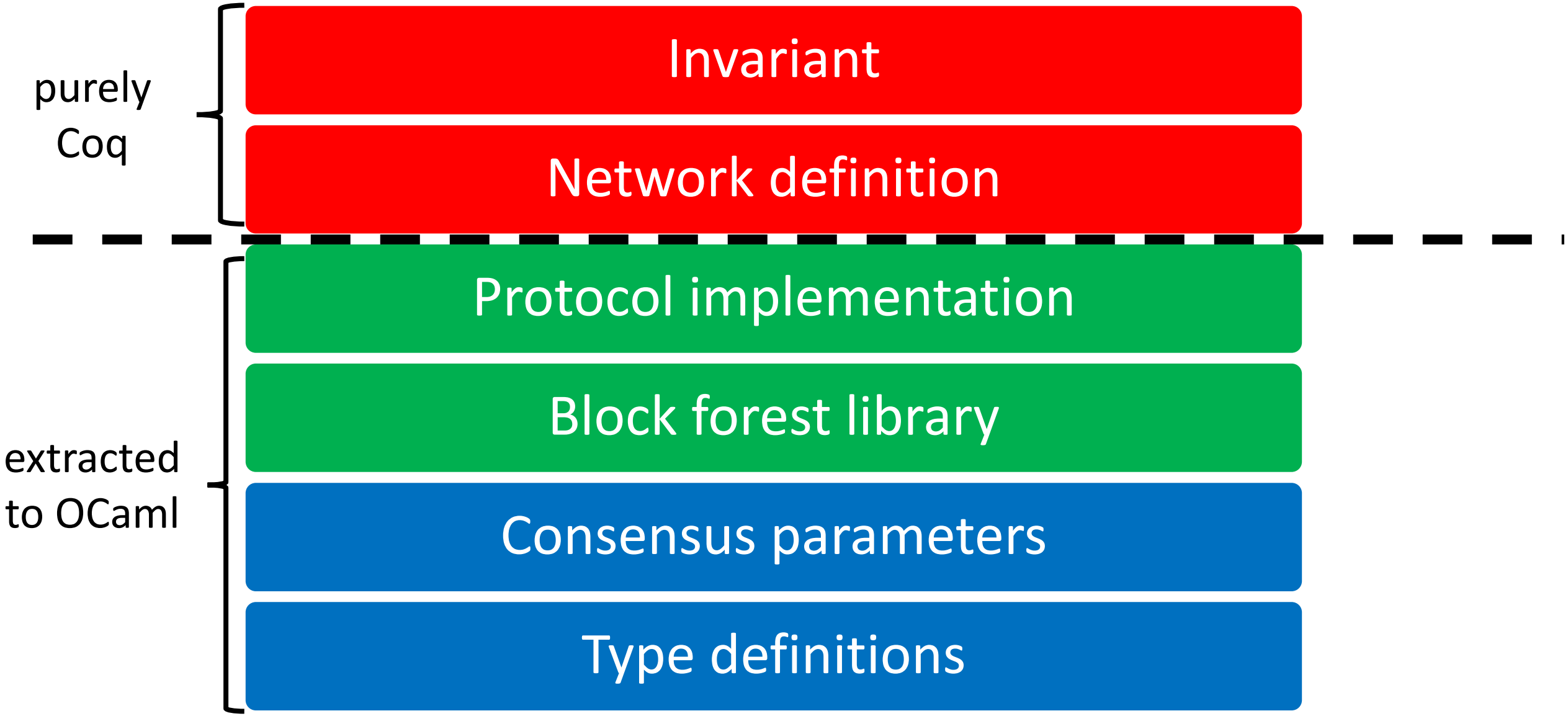

Invariant
Network definition
Protocol implementation
Block forest library
Consensus parameters
Type definitions


Invariant
Network definition
Protocol implementation
Block forest library
Consensus parameters
Type definitions
need to be instantiated


Invariant
Network definition
Protocol implementation
Block forest library
Consensus parameters
Type definitions


Invariant
Network definition
Protocol implementation
Block forest library
Consensus parameters
Type definitions


Invariant
Network definition
Protocol implementation
Block forest library
Consensus parameters
Type definitions
extracted
to OCaml


purely
Coq
Invariant
Network definition
extracted
to OCaml
Protocol implementation
Block forest library
Consensus parameters
Type definitions

```
Record State :=
  Node {
    id : Address;
    peers : peers_t;
    blockTree : BlockTree;
    txPool : TxPool;
  }.
```

```
Definition procInt (st : State) (tr : InternalTransition) :=
    let: Node n prs bt pool := st in
    match tr with
    | TxT tx => pair st (emitBroadcast n prs (TxMsg tx))

    (* Assumption: nodes broadcast to themselves as well! => simplifies logic *)
    | MintT =>
      let: bc := btChain bt in
      let: allowedTxs := [seq t <- pool | txValid t bc] in
      match genProof bc allowedTxs ts with
      | Some (txs, pf) =>
        let: prevBlock := last GenesisBlock bc in
        let: b := mkB (hashB prevBlock) txs pf in
        if valid_chain_block bc b then
          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 (BlockMsg b))
        else
          pair st emitZero
      | None => pair st emitZero
      end
    end.
```

```
Definition procMsg (st: State) (from : Address) (msg: Message) :=
    let: Node n prs bt pool := st in
    match msg with
    | 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))

    | InvMsg peerHashes =>
      let: ownHashes := dom bt ++ [seq hashT t | t <- pool] in
      let: newH := [seq h <- peerHashes | h \notin ownHashes] in
      let: gets := [seq mkP n from (GetDataMsg h) | h <- newH] in
      pair st (emitMany gets)

    | 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))
    end.
```

```
(** Instantiate Toychain with a proof-of-work scheme **)
Module ProofOfWork <: (ConsensusParams TypesImpl).
Import TypesImpl.

Definition GenesisBlock : block :=
  mkB (("0x6150cb353fe365318be1040f4f1d55ba6a6235c7fdee7e94602fed76f112f2de")%string <: Hash)
      [::]
      ((N_of_nat 0) <: VProof).

(* Hash should be HexStrings prefixed with 0x, e.g. '0x1c2139314aab35' *)
Parameter hashT : Transaction -> Hash.
Parameter hashB : block -> Hash.

Definition work (b : block) : WorkAmnt :=
  count_binary_zeroes (hashB b).
```

```
(* You'd normally want some difficulty adjustment.*)
Definition VAF (b : Block) (bc : Blockchain) : bool :=
  (* GenesisBlock doesn't have work requirements *)
  if (b == GenesisBlock) then
    if (bc == [::]) then true else false
  (* All other blocks do *)
  else if (12 <? (work b))%N then true else false.

(* For proof-of-work, this would be more aptly called "getNonce" *)
Parameter genProof : Blockchain -> TxPool -> option VProof.
```

```
(* Behaves as > *)
Definition FCR bc bc' : bool :=
  let w := total_work bc in
  let w' := total_work bc' in
  let l := (List.length bc) in
  let l' := (List.length bc') in
  let eW := w == w' in
  let eL := l == l' in
  let eO := bc == bc' in

  (* Written in this weird fashion to be able to prove both
     transitivity and totality. *)
  match eW, eL, eO with
  | true, true, true => false
  | true, true, false => ords bc bc'
  | true, _, _ => l' > l
  | false, _, _ => w' > w
  end.
```

```
while true do
   procInt_wrapper ();
   procMsg_wrapper ();
done;
```

Demo

Final thoughts

Take away

- Formalisation of a blockchain consensus protocol in Coq:
  - minimal set of required security primitives
  - per-node protocol logic & data structures
  - proof of global eventual consistency

- Extracted proven-correct OCaml implementation

https://github.com/certichain/toychain

Future work

- Abstract gossip mechanism
- Non-strict FCRs
- Probabilistic reasoning for security properties

and a lot more…