Compositional Verification of Composite Byzantine Protocols

Qiyuan Zhao, George Pîrlea, Karolina Grzeszkiewicz
Seth Gilbert and Ilya Sergey

Distributed Protocols

- Distributed systems are important!
  - Scalability, reliability, performance, …
  - Theoretical foundation: distributed protocols
    - Defining how a node collaborates with other nodes

Byzantine Fault Tolerance

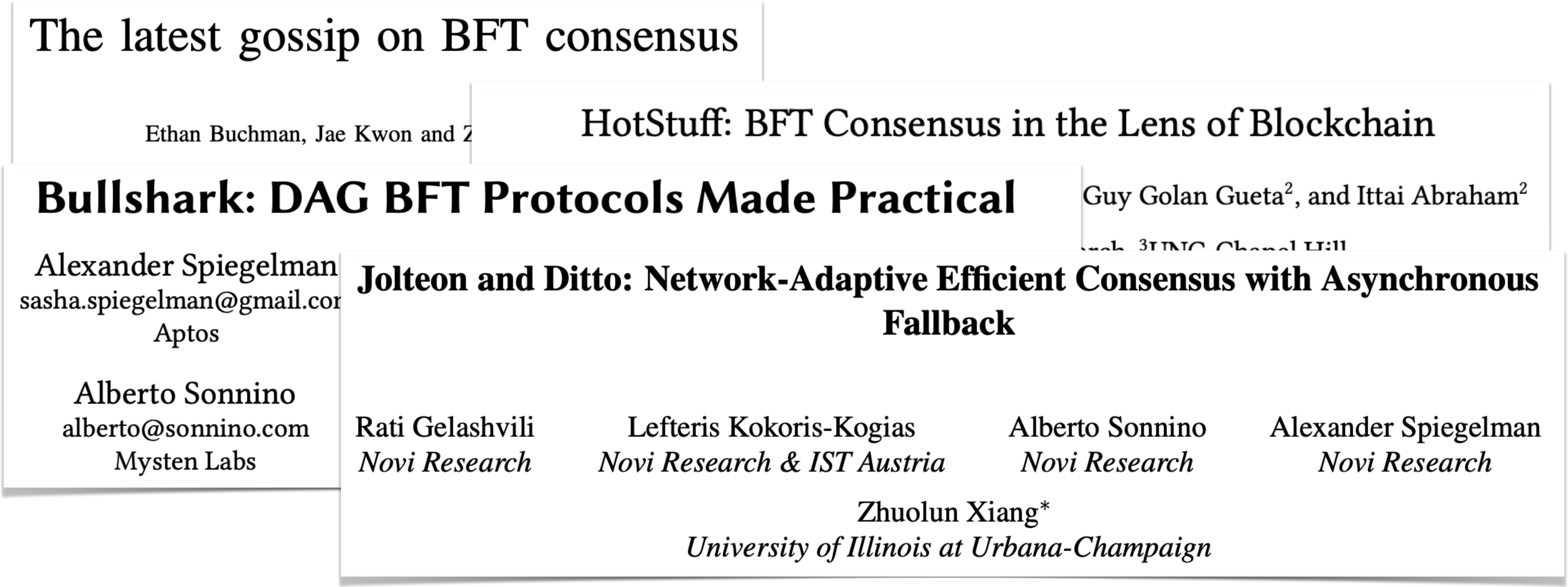
- Fault tolerance: a key goal in protocol design
- Byzantine fault:
  - Faulty nodes that can deviate from the protocol arbitrarily

The Byzantine Generals Problem

LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE
SRI International

Byzantine Fault Tolerance Protocols

- Key in ensuring the reliability and integrity of various Internet services



The latest gossip on BFT consensus

Ethan Buchman, Jae Kwon and Z

HotStuff: BFT Consensus in the Lens of Blockchain

Guy Golan Gueta[2], and Ittai Abraham[2]

Bullshark: DAG BFT Protocols Made Practical

Alexander Spiegelman
sasha.spiegelman@gmail.com
Aptos

Alberto Sonnino
alberto@sonnino.com
Mysten Labs

Jolteon and Ditto: Network-Adaptive Efficient Consensus with Asynchronous Fallback

Rati Gelashvili, *Novi Research*
Lefteris Kokoris-Kogias, *Novi Research & IST Austria*
Alberto Sonnino, *Novi Research*
Alexander Spiegelman, *Novi Research*
Zhuolun Xiang*, *University of Illinois at Urbana-Champaign*

BFT Protocols Are Hard to Get Right

BFT Protocols Are Hard to Get Right

dranov / **protocol-bugs-list**

Errors found in distributed protocols

Protocol	Reference	Violation	Counter-example	#Year(s) taken to discover the bug
Sync HotStuff	[Abraham et al. 2019]	safety & liveness	[Momose and Cruz 2019]	≤ 1
Tendermint	[Buchman 2016]	liveness	[Cachin and Vukolić 2017]	≈ 1
hBFT	[Duan et al. 2015]	safety	[Shrestha et al. 2019]	≈ 4
Zyzzyva	[Kotla et al. 2007; Kotla et al. 2010]	safety	[Abraham et al. 2017]	≈ 7
FaB Paxos	[Martin and Alvisi 2005; Martin and Alvisi 2006]	liveness	[Abraham et al. 2017]	≈ 12
PBFT[1]	[Castro and Liskov 1999]	liveness	[Berger et al. 2021]	≈ 22

Source: https://github.com/dranov/protocol-bugs-list

BFT Protocols Are Hard to Get Right

jepsen QUINT

TLA+

APALACHE

- Testing or model checking BFT protocols may not be effective
  - Byzantine behavior ⇒ large search space
  - Precisely capturing Byzantine behavior is difficult

Verification Builds Trust

- Reducing the risk of having bugs by formal verification
  - Proving properties rigorously with proofs aided/checked by machine

Verification is Also Laborious

IronFleet: Proving Practical Distributed Systems Correct

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch,
Bryan Parno, Michael L. Roberts, Srinath Setty, Brian Zill

Microsoft Research

“Proofs take 39253 LoC in total”

Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq ⋆

Vincent Rahli ✉, Ivana Vukotic, Marcus Völp, Paulo Esteves-Verissimo

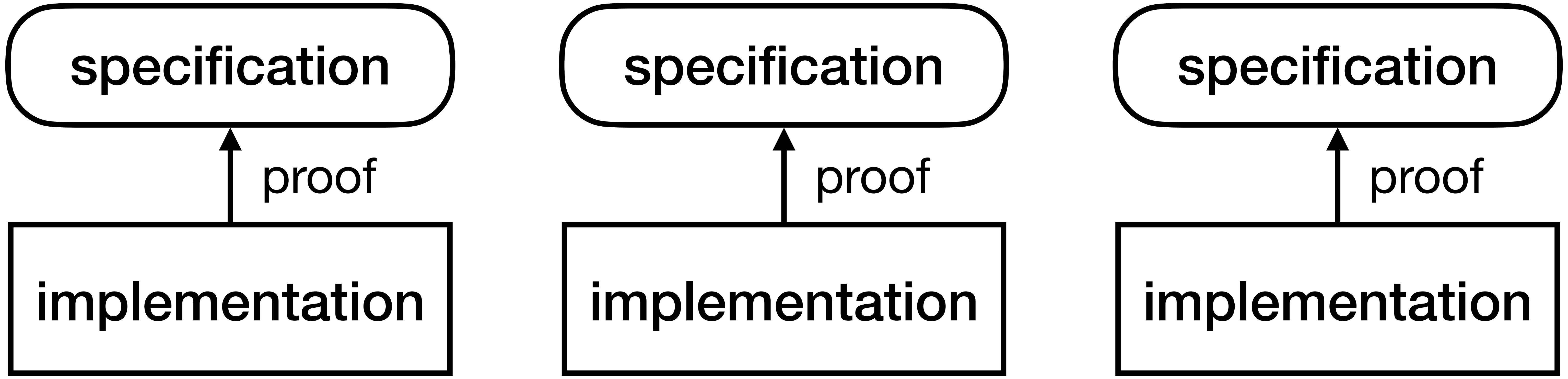
SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg
firstname.lastname@uni.lu

“Verifying PBFT takes around 20000 lines of specs and around 20000 lines of proofs”

- Such great efforts are difficult to reuse!

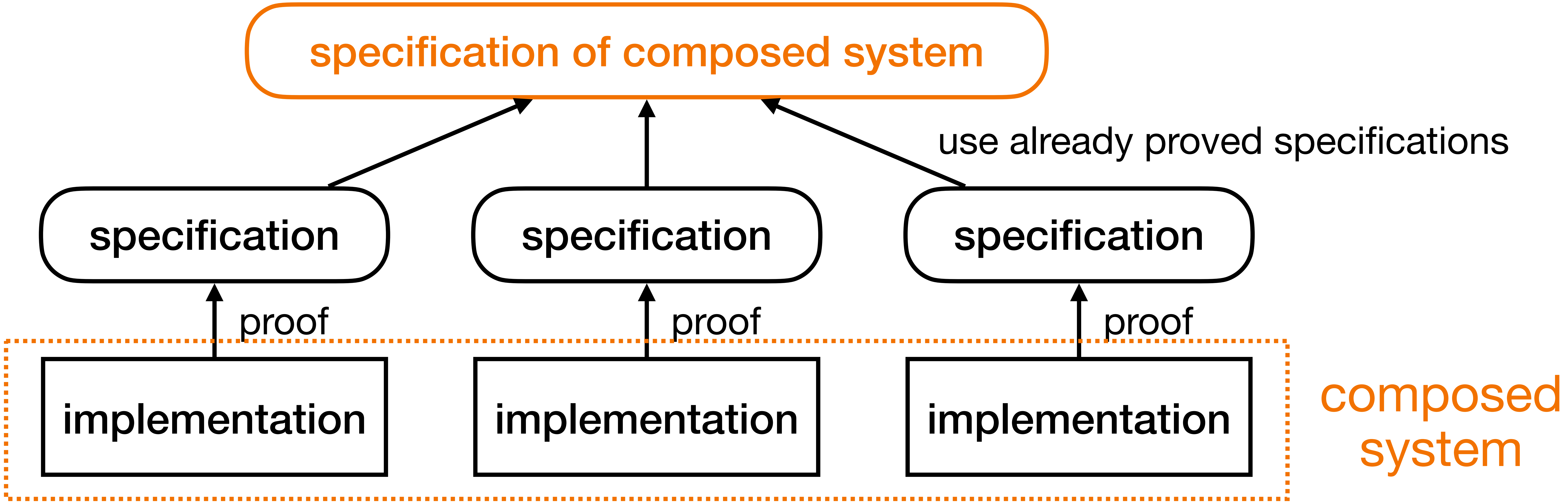
Compositionality For The Win

- Compositionality: the conventional wisdom in doing verification
  - Separation of specification and implementation
  - Modularity & proof reuse



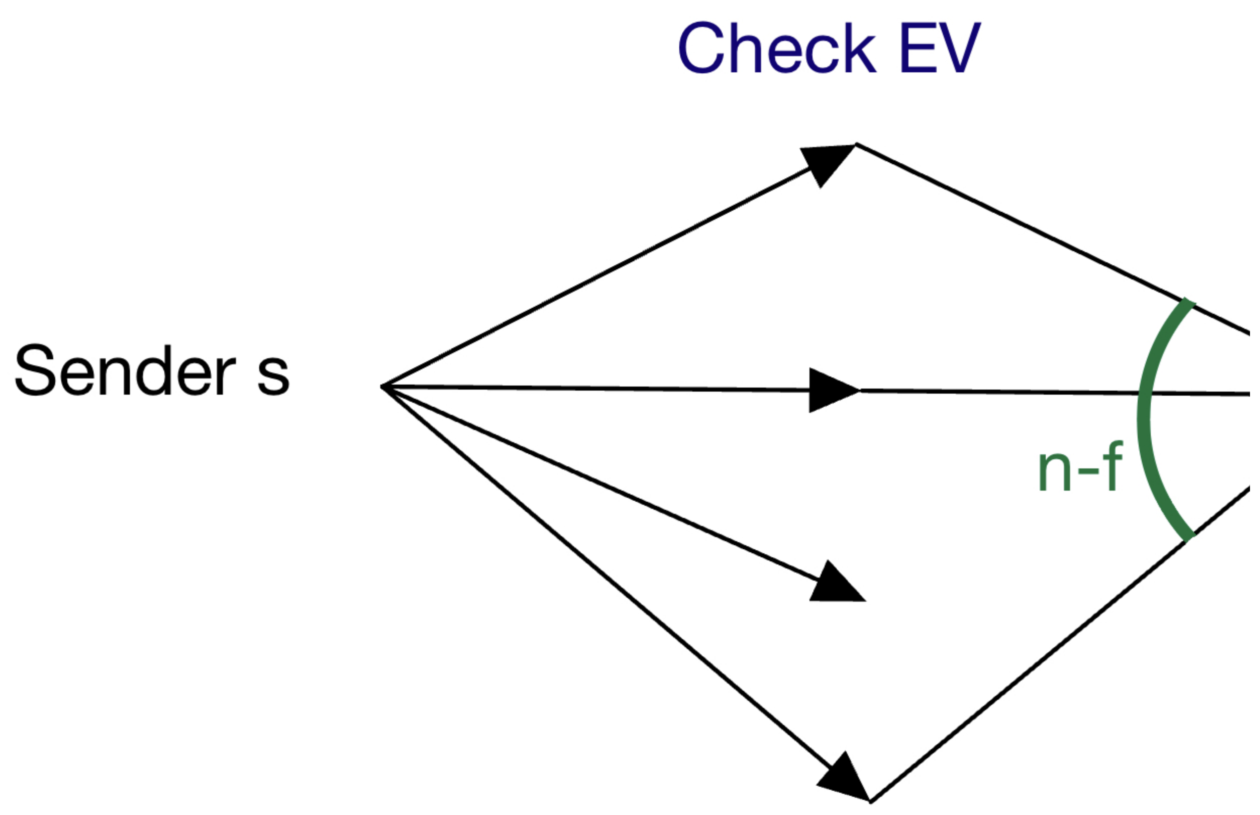
Compositionality For The Win

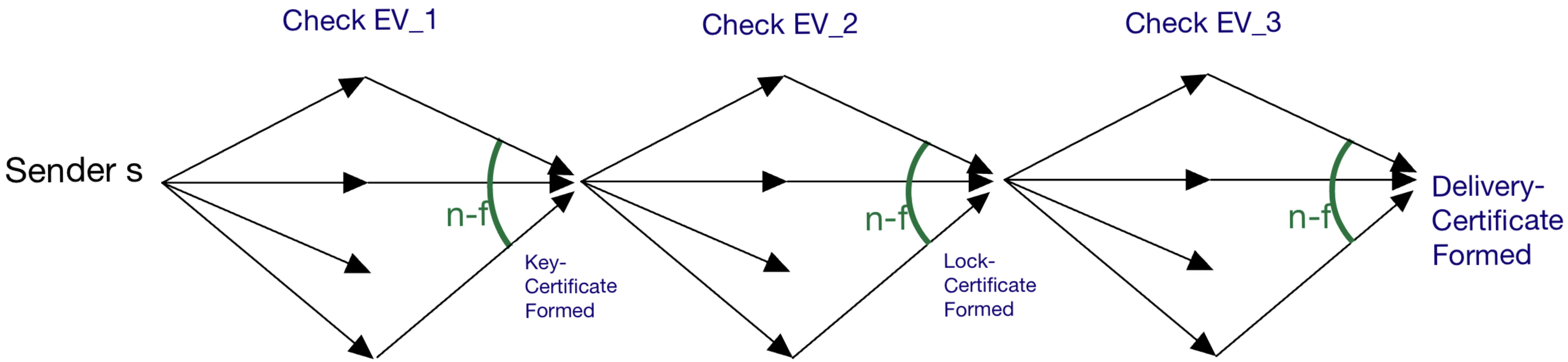
- Compositionality: the conventional wisdom in doing verification
  - Separation of specification and implementation
  - Modularity & proof reuse



Compositionality For The Win

- Composition: strategy for reducing conceptual complexity in BFT protocol design





Blog source: https://decentralizedthoughts.github.io/
Image credit: https://decentralizedthoughts.github.io/2022-09-10-provable-broadcast/

We want to make verification compositional for (potentially composite) BFT protocols.

Our Contribution

- BYTHOS: streamlining the verification of BFT protocols and their compositions
  - Embedded in the Coq proof assistant ⇒ foundational
  - The first framework that supports:

☑ Reasoning about Byzantine faults

☑ Modular safety & liveness proofs of BFT protocols

☑ Proof reuse for verifying composite BFT protocols

☑ Executable reference implementation extracted to OCaml

Specifying Systems in BYTHOS

Workflow

Encoding the protocol

Proving safety properties

Reasoning about liveness

Composing protocols

Verifying composite protocols

Techniques

Specifying Systems in BYTHOS

Workflow

Encoding the protocol

Proving safety properties

Reasoning about liveness

Composing protocols

Verifying composite protocols

Techniques

Running Example: Provable Broadcast (PB)

Provable Broadcast

Written by Ittai Abraham, Alexander Spiegelman
Posted on September 10, 2022

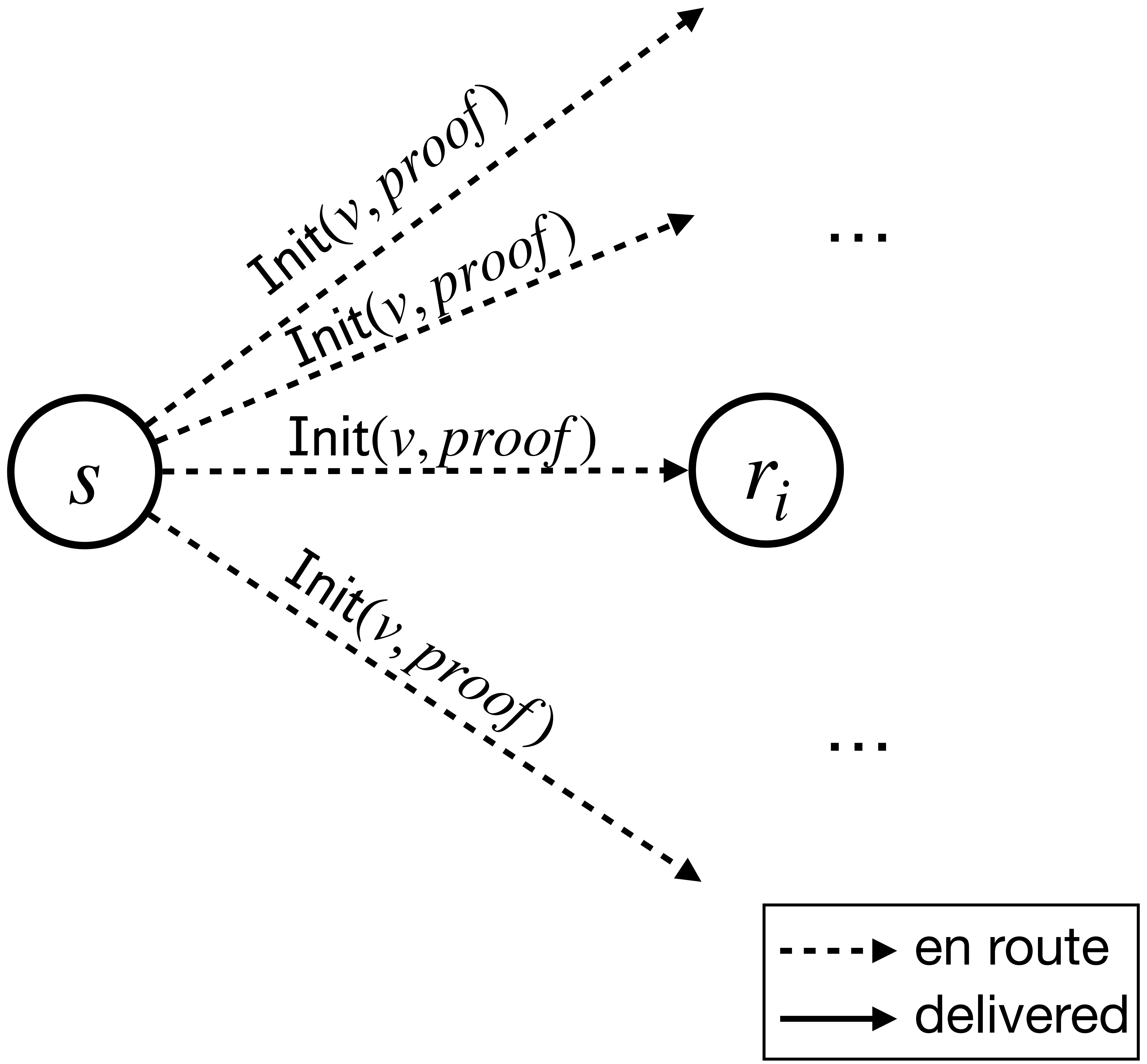
*“PB based protocols are the backbone of many authenticated consensus protocols.”**

- Intuitively, it is for ensuring that more than f non-faulty nodes accept some value that satisfies a notion of *external validity*
    - Assume $n > 3f$ (n: the number of nodes, at most f nodes are Byzantine)

* https://decentralizedthoughts.github.io/2022-09-10-provable-broadcast/

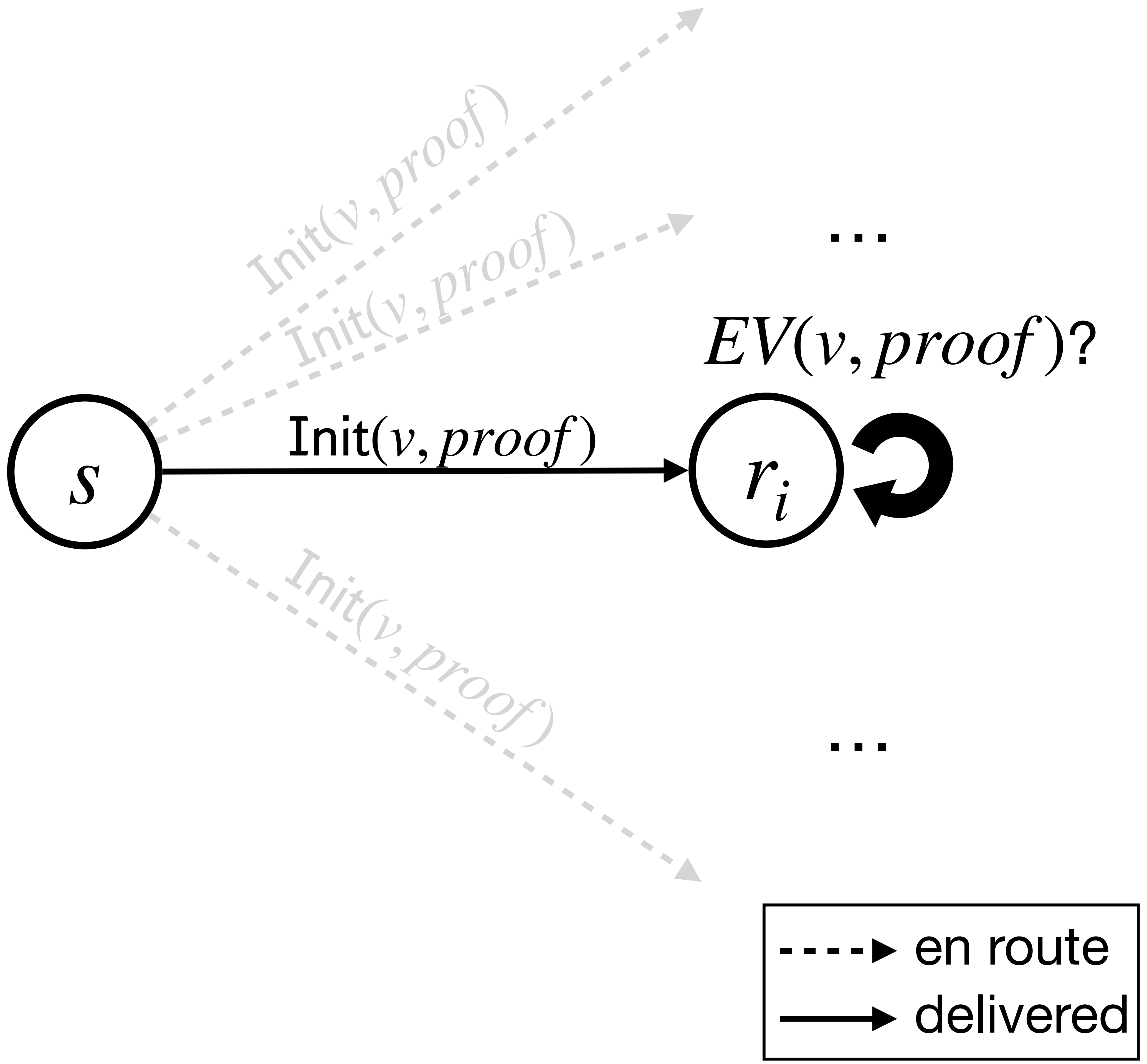
Running Example: Provable Broadcast

- A sender broadcasts a value v and an associated proof



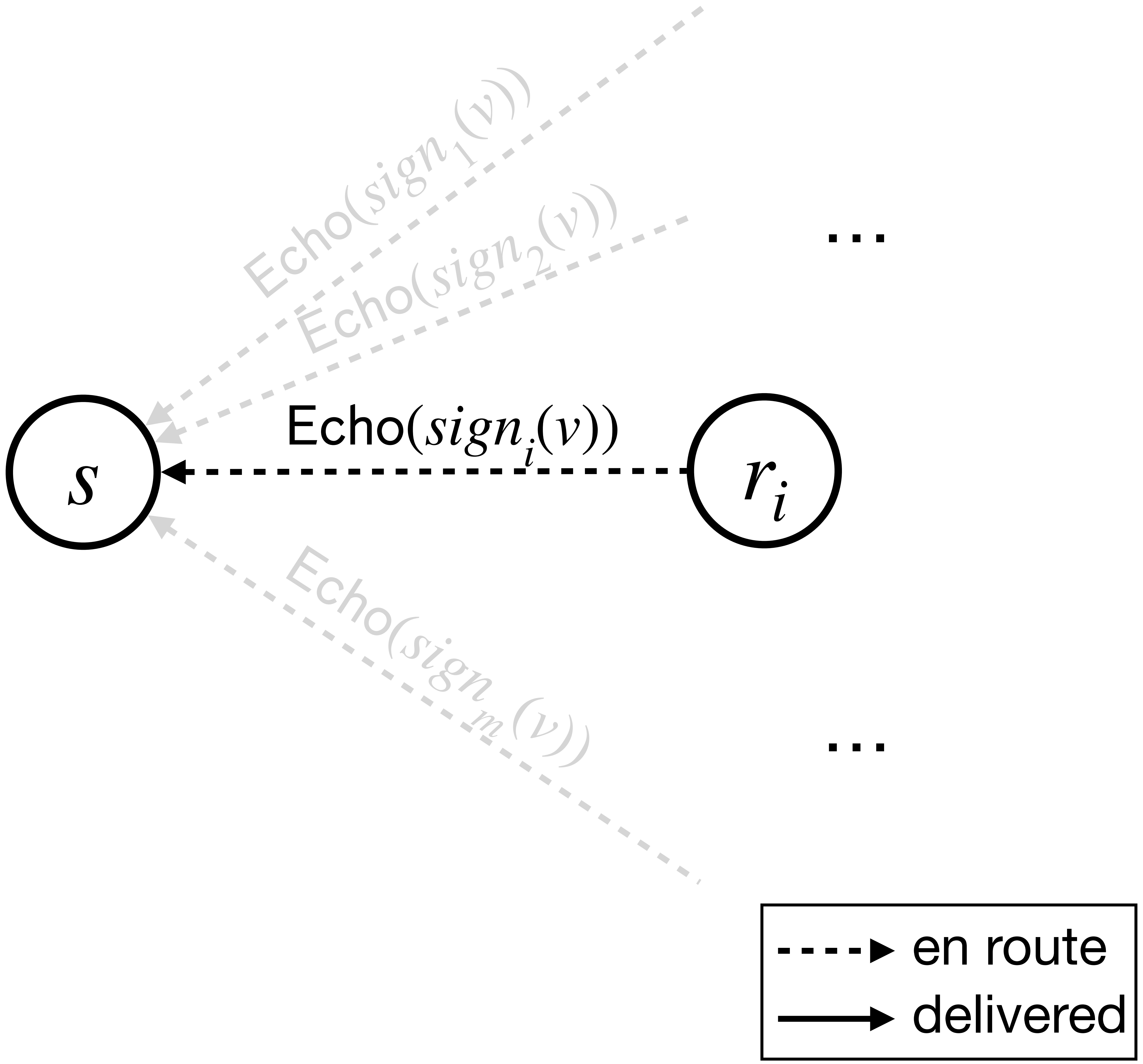
Running Example: Provable Broadcast

- A sender broadcasts a value v and an associated proof
- Each receiver validates the value using an external validity function EV



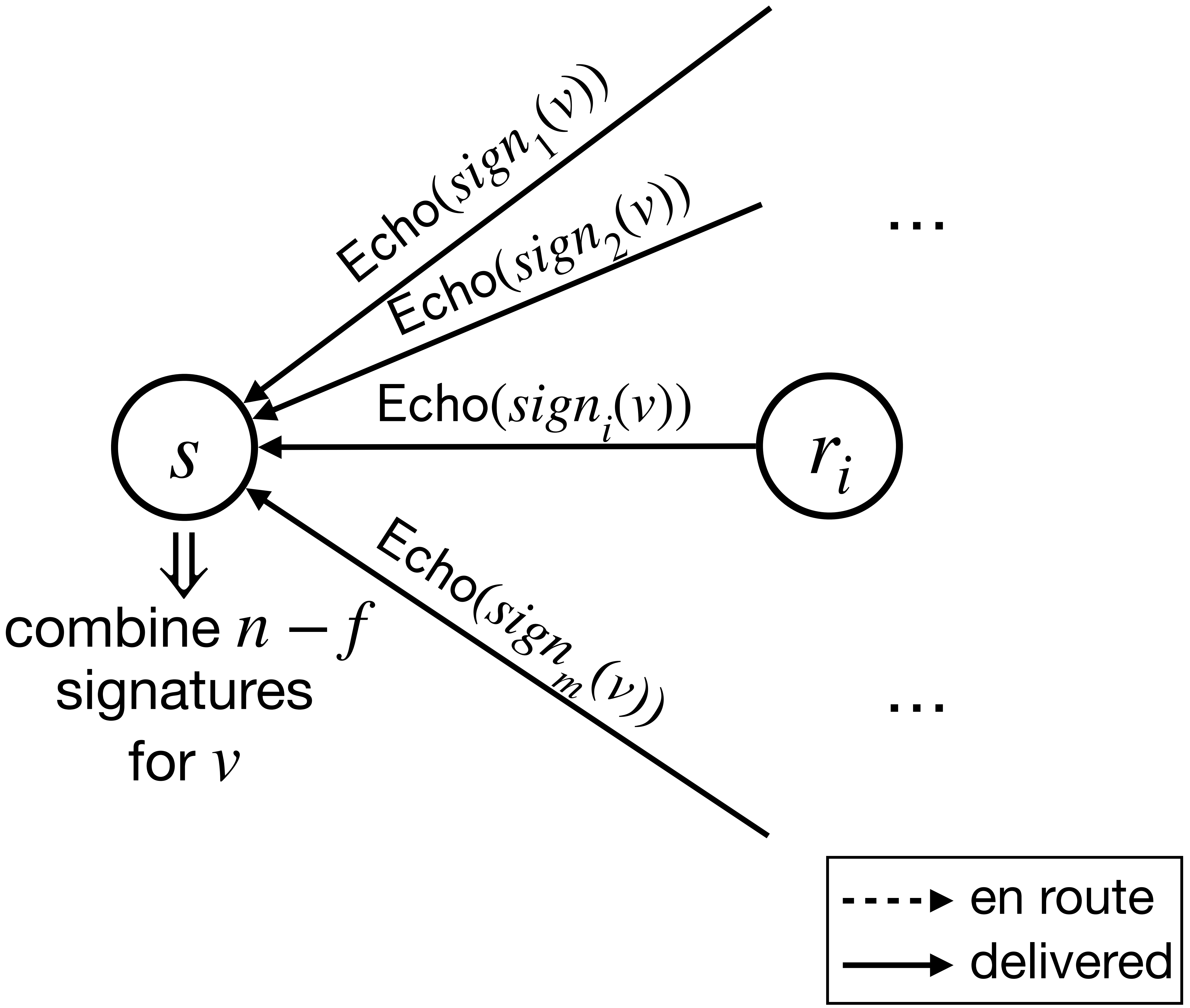
Running Example: Provable Broadcast

- A sender broadcasts a value v and an associated proof
- Each receiver validates the value using an external validity function EV
- For the first externally valid value v, the receiver signs v and echoes the signature to the sender



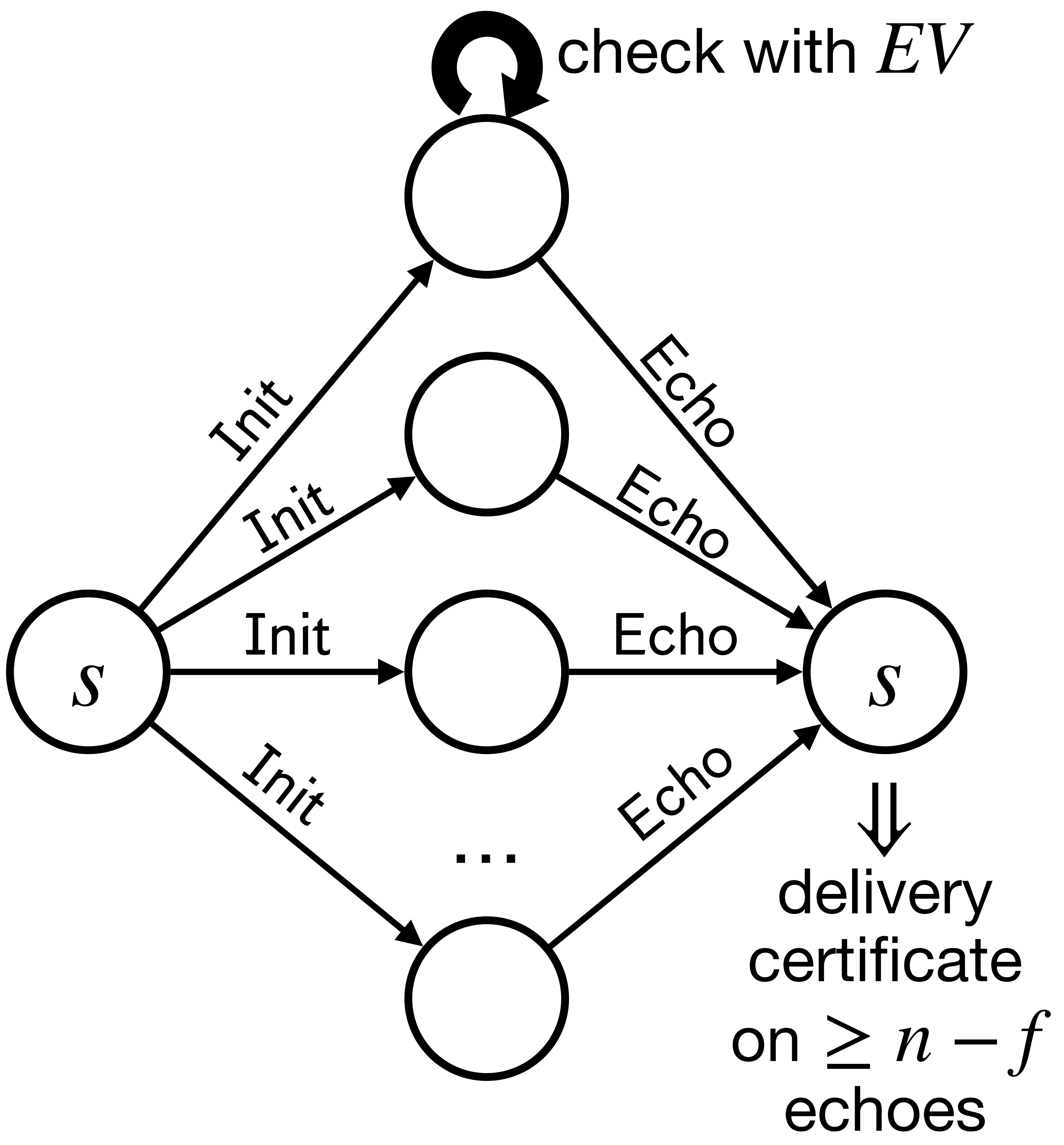
Running Example: Provable Broadcast

- A sender broadcasts a value v and an associated proof
- Each receiver validates the value using an external validity function EV
- For the first externally valid value v, the receiver signs v and echoes the signature to the sender
- The sender waits for $n - f$ echoes to combine into a *delivery certificate*



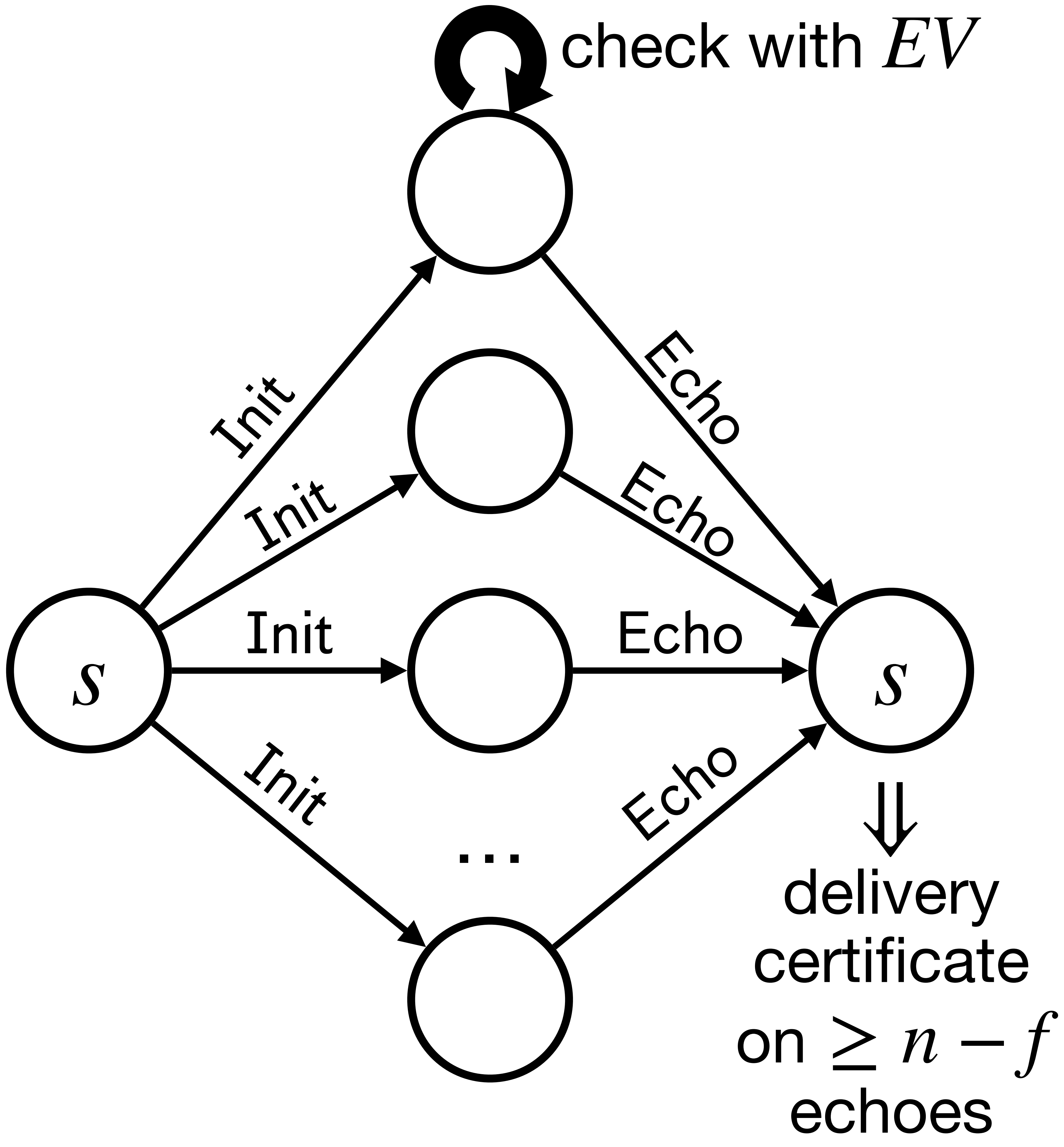
Specification of Provable Broadcast

- Safety ("bad thing never happens"):
  - If a delivery certificate exists for v, then
    - at least $n - 2f$ non-faulty nodes know and echoed v
    - v is externally valid
    - no any other value can have delivery certificate



Specification of Provable Broadcast

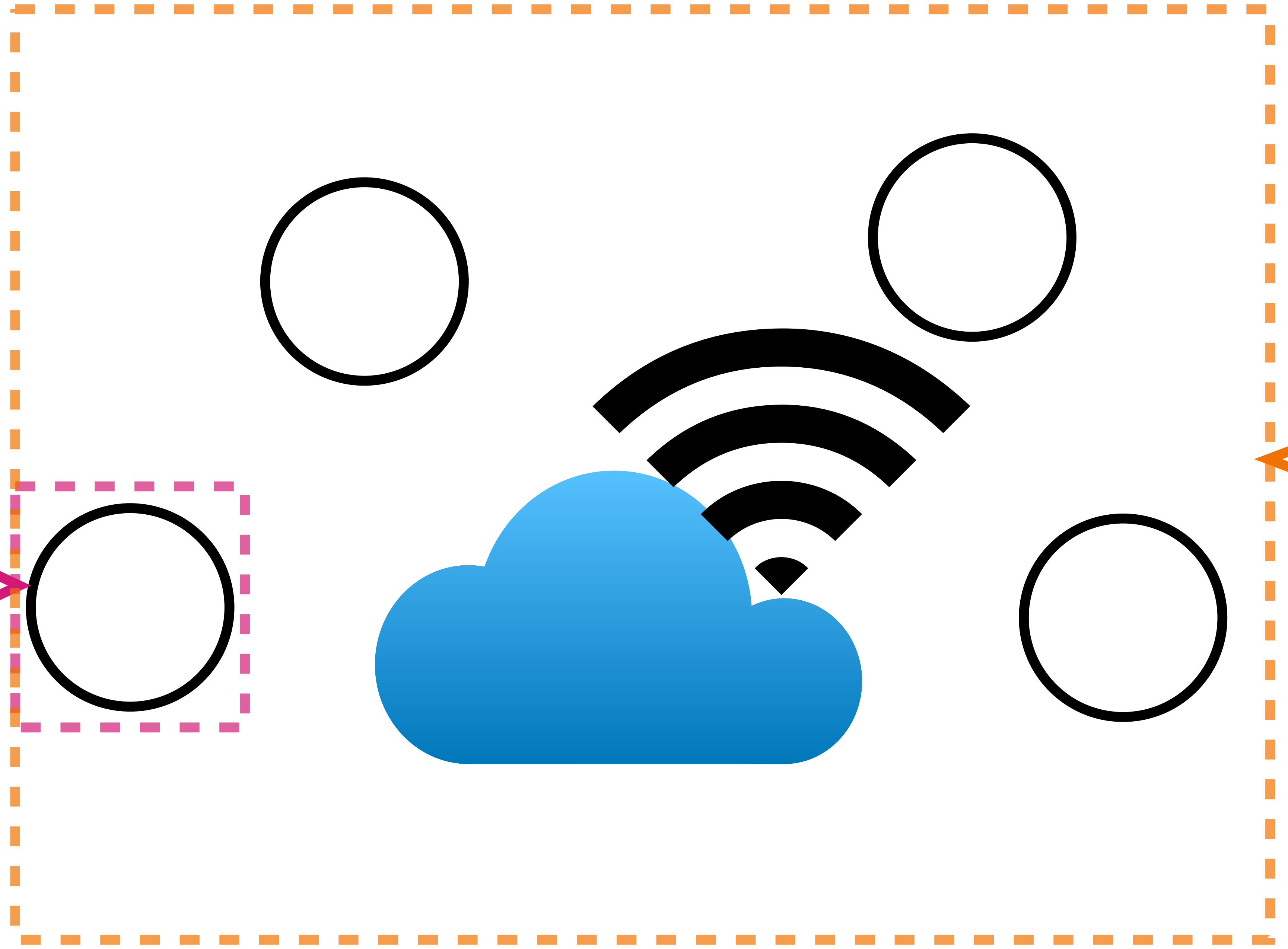
- Liveness ("good thing eventually happens"):
  - Given that s is non-faulty and v is externally valid, if s broadcast v, then s will <u>eventually</u> obtain a delivery certificate for v



Encoding the Protocol

- System in BYTHOS: includes the set of nodes and a network

Protocol logic:
- User-provided
- Determines node behavior



System semantics:
- Parametric over protocol logic
- Works as a state machine

Ingredients of a Protocol

- The kinds of internal events
- The kinds of messages

```
Inductive InternalEvent :=
  | Start.
Inductive Message :=
  | Init (v : Value) (pf : Proof)
  | Echo (sig : Signature).
```

Each bar represents one kind

Ingredients of a Protocol

- The kinds of internal events
- The kinds of messages

```
Inductive InternalEvent :=
  | Start.
Inductive Message :=
  | Init (v : Value) (pf : Proof)
  | Echo (sig : Signature).
```

payload

Ingredients of a Protocol

- The kinds of internal events
- The kinds of messages
- The local state of a non-faulty node
  - Keeps track of what the node has done

```
Record State := {
  id : Address;
  started : option (Value × Proof);
  delivery_certificate : option CombinedSignature;
  echo_counter : set (Address × Signature);

  echoed : option (Value × Proof) }.
```

self address

Ingredients of a Protocol

- The kinds of internal events
- The kinds of messages
- The local state of a non-faulty node
  - Keeps track of what the node has done

```
Record State := {
  id : Address;
                                    sender state
  started : option (Value × Proof);
  delivery_certificate : option CombinedSignature;
  echo_counter : set (Address × Signature);

  echoed : option (Value × Proof) }.
```

records from whom the Echo messages comes from and the attached signatures

Ingredients of a Protocol

- The kinds of internal events
- The kinds of messages
- The local state of a non-faulty node
    - Keeps track of what the node has done

```
Record State := {
  id : Address;

  started : option (Value × Proof);
  delivery_certificate : option CombinedSignature;
  echo_counter : set (Address × Signature);

  echoed : option (Value × Proof) }.
```

receiver state
(records to which value
and proof the node has echoed)

Ingredients of a Protocol

- The kinds of internal events
- The kinds of messages
- The local state of a non-faulty node
- The handler for internal events

Handler: given the original state,
returns the updated state
and the messages to send out

```
Definition procInt (st : State) (ev : InternalEvent)
  : State × list Packet := (* ... *).
```

Ingredients of a Protocol

- The kinds of internal events
- The kinds of messages
- The local state of a non-faulty node
- The handler for internal events
- The handler for incoming messages

Handler: given the original state,
returns the updated state
and the messages to send out

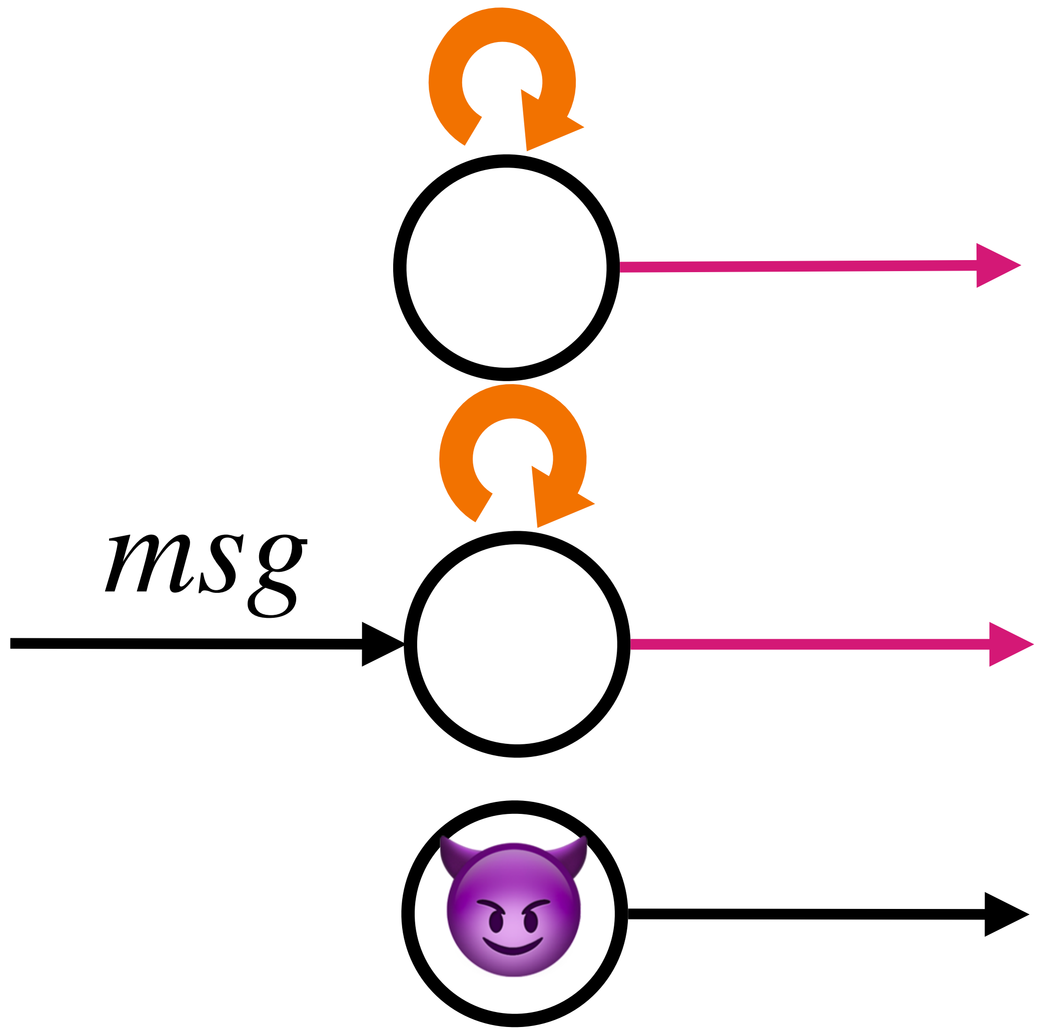
```
Definition procMsg (st : State) (sender : Address)
  (msg : Message) : State × list Packet :=
  match msg with
  | Init v pf =>
    if (st.echoed == None) && (EV v pf)
    then
      (st <| echoed := Some (v, pf) |>,
       (* the packet containing Echo (sign v) to sender *))
    else (st, empty_list)
  | Echo sig => (* ... *)
  end.
```

Ingredients of a Protocol

- The kinds of internal events
- The kinds of messages
- The local state of a non-faulty node
- The handler for internal events
- The handler for incoming messages
- The constraint over Byzantine nodes

System Semantics

- System in BYTHOS: state machine
  - System state = local states of nodes + state of network (all sent messages)
- At most one node performs an atomic step in one transition



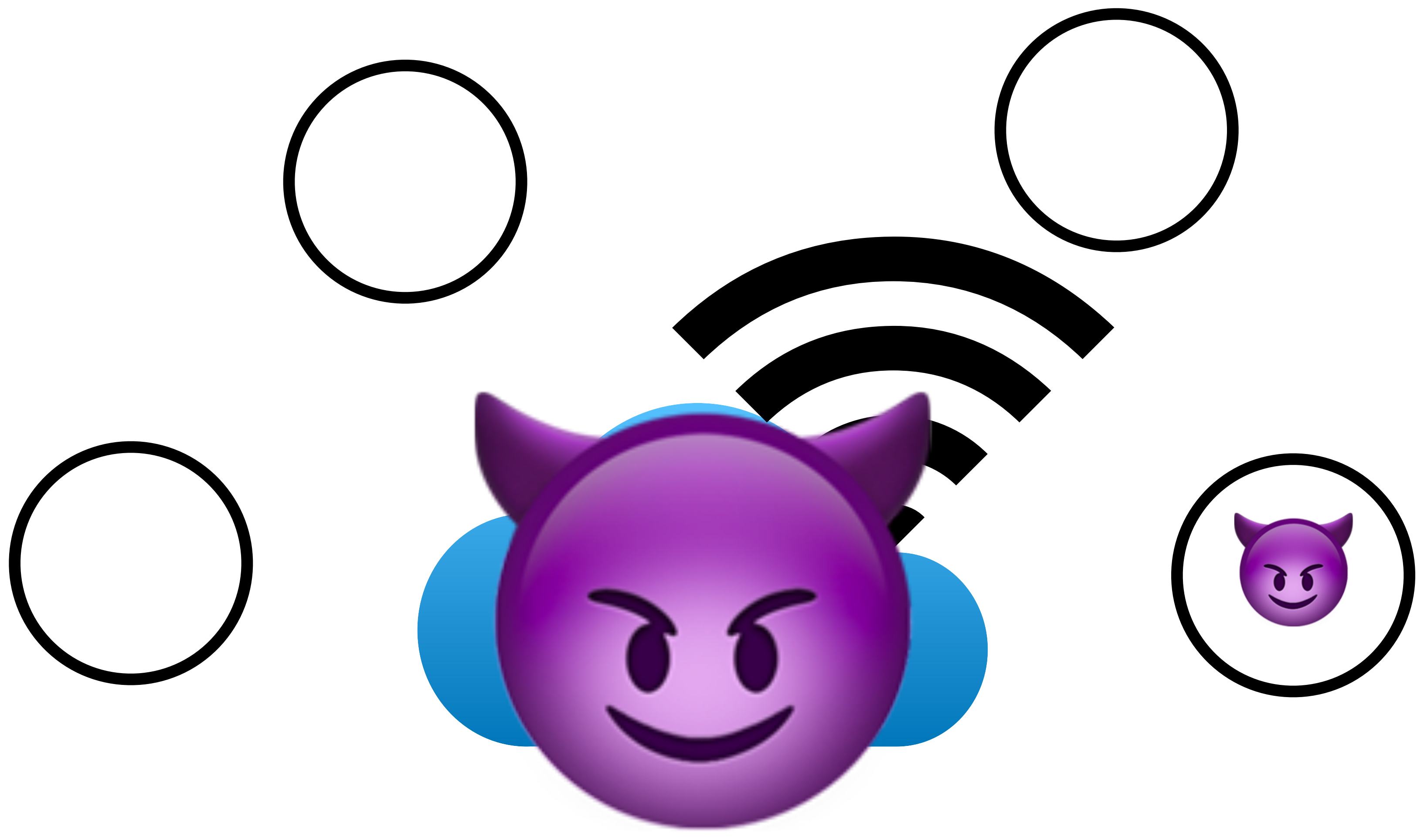
Handling an internal event with `procInt`

Handling an incoming message with `procMsg`

Byzantine node sending out message

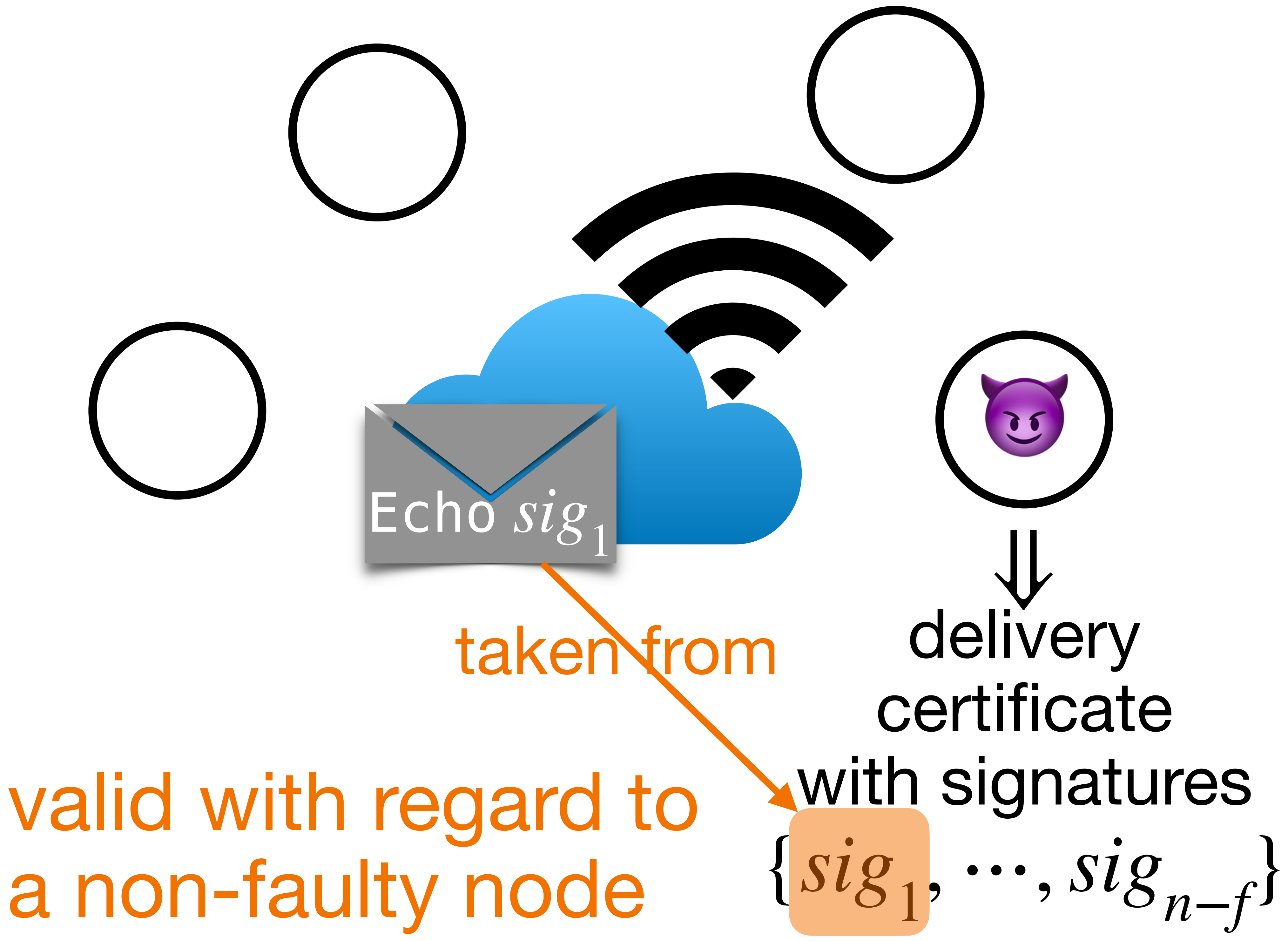
Modeling Byzantine Adversary

- Assume an adversary controlling both the network and Byzantine nodes
    - Network is asynchronous
    - Byzantine nodes can intercept messages
- Byzantine nodes affect the system only by sending out messages
    - No modeling of their local states



Modeling Byzantine Adversary

- Using Dolev-Yao model for constraining Byzantine behavior
  - E.g., Byzantine nodes can take signatures from existing messages but cannot forge signatures
- Byzantine messages are under such constraints



Specifying Systems in BYTHOS

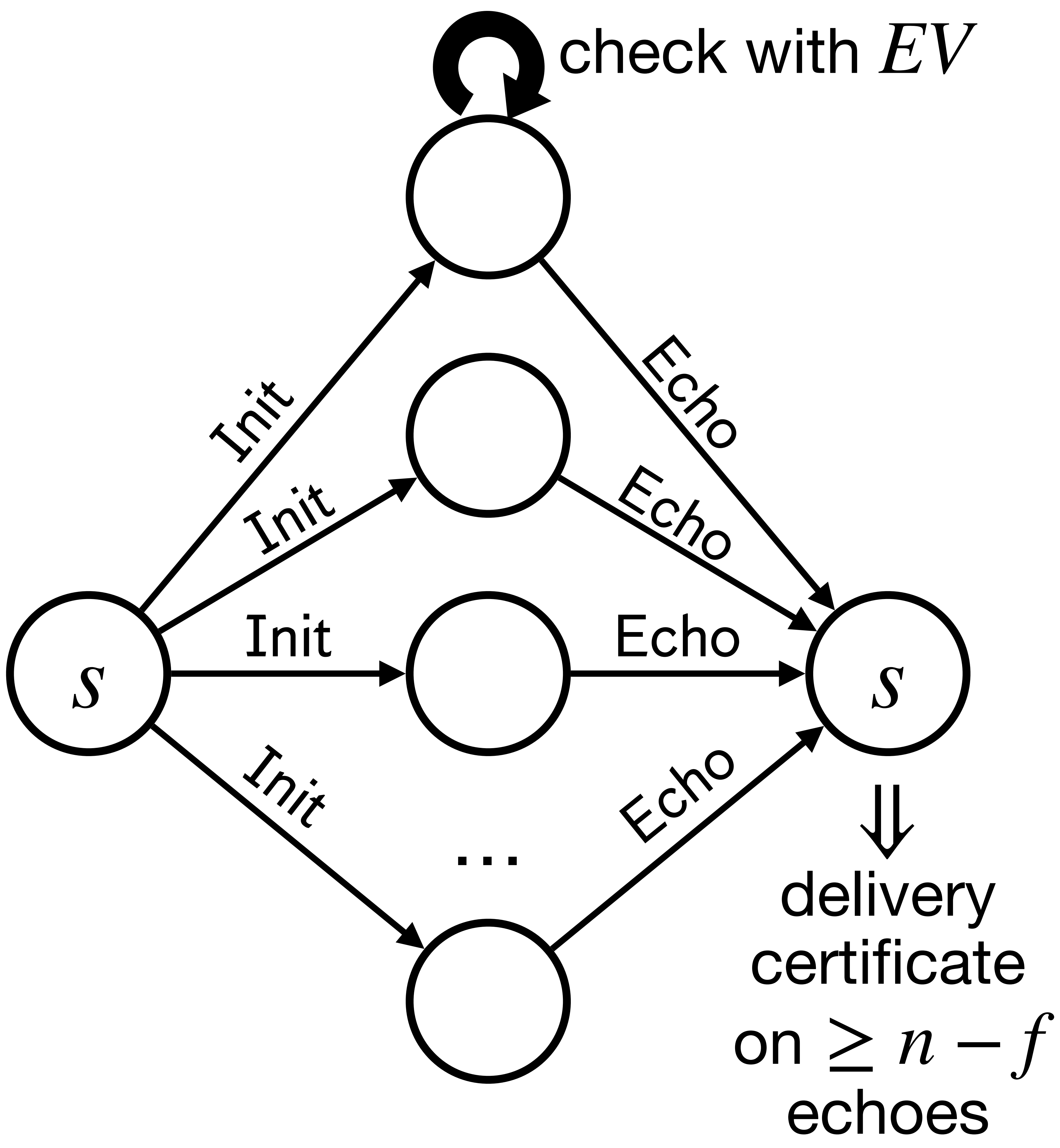
Workflow

- Encoding the protocol
- Proving safety properties
- Reasoning about liveness
- Composing protocols
- Verifying composite protocols

Techniques

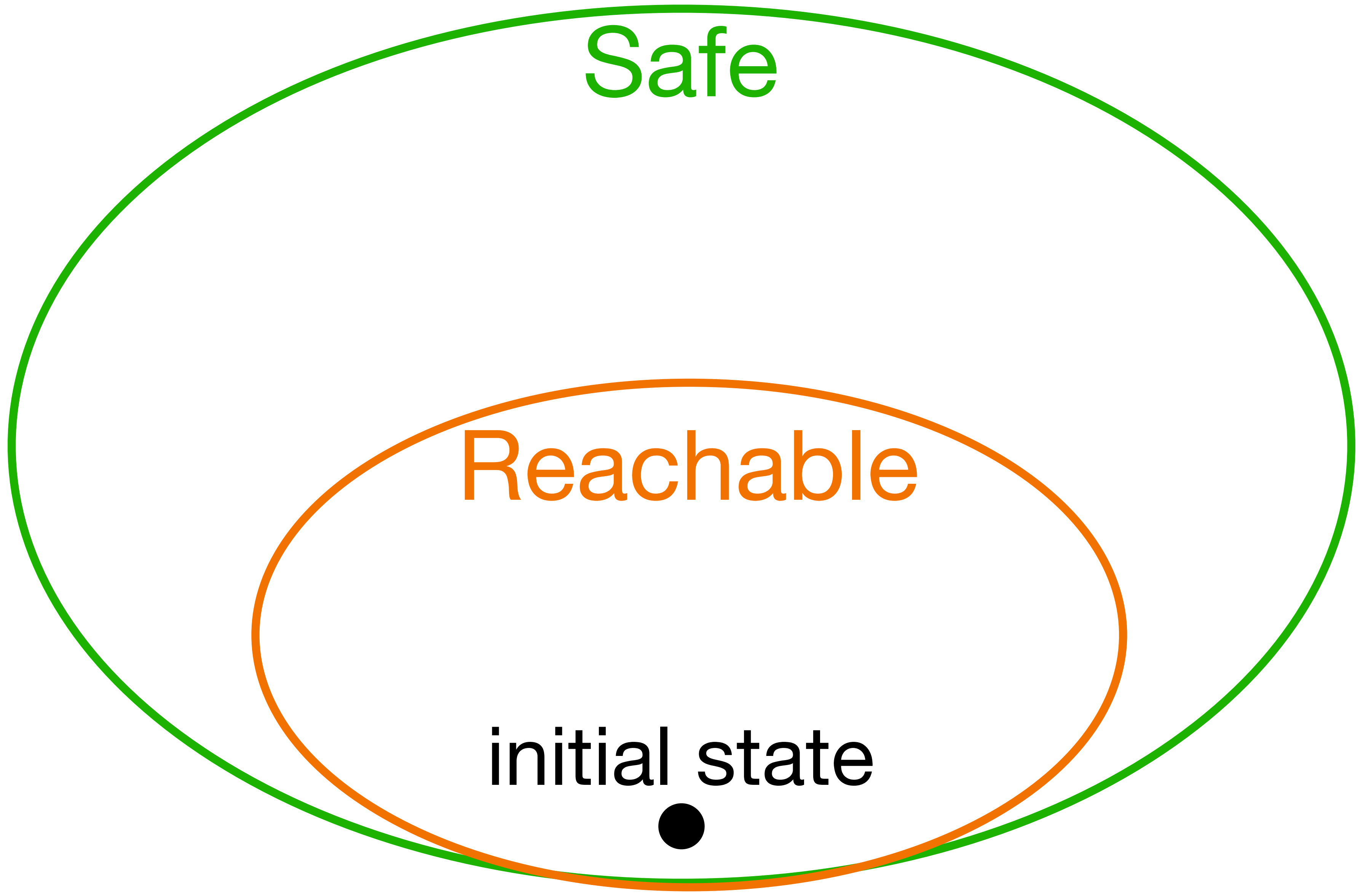
Safety Properties of Provable Broadcast

- Safety ("bad thing never happens"):
  - If a delivery certificate exists for v, then
    - at least $n-2f$ non-faulty nodes know and echoed v
    - v is externally valid
    - no any other value can have delivery certificate



Safety Properties, Formalized

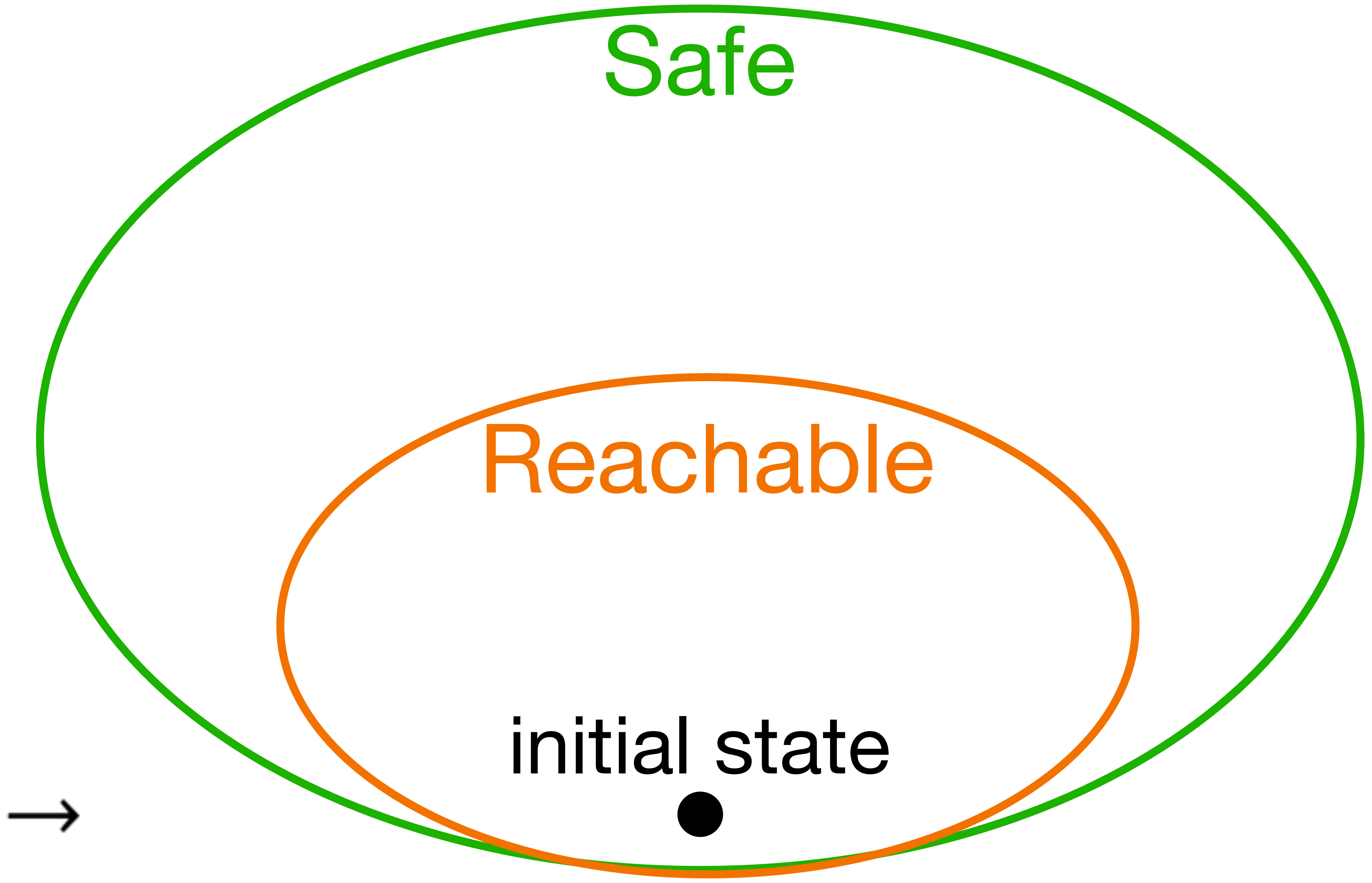
- Proving safety amounts to establishing it as an invariant
    - Invariant: a predicate that holds on all system states reachable via transitions from an initial system state



Safety Properties, Formalized

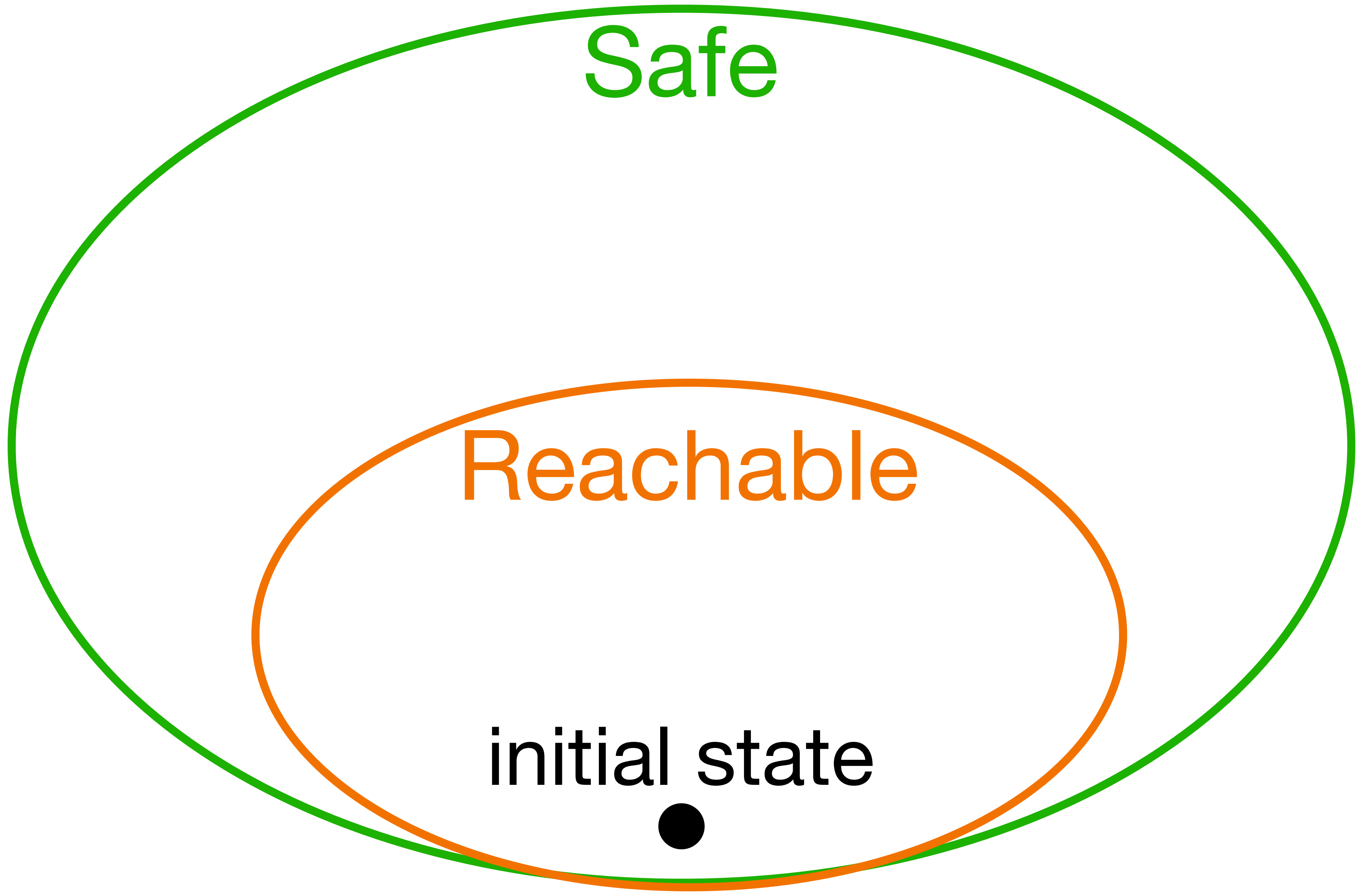
```
Definition safety (σ : SystemState) : Prop :=
  forall (v : Value),
    (* a delivery certificate exists for v in σ *) →
      (exists S, |S| ≥ n − 2f ∧
        (forall q, q ∈ S →
          isByzantine q = false ∧
          (* q's local state in σ records that
             q has echoed to v *)))
      ∧
      externally_valid v
      ∧
      (forall (v′ : Value),
        (* a delivery certificate exists for v′ in σ *) →
        v = v′).

Goal forall σ, reachable σ → safety σ.
```



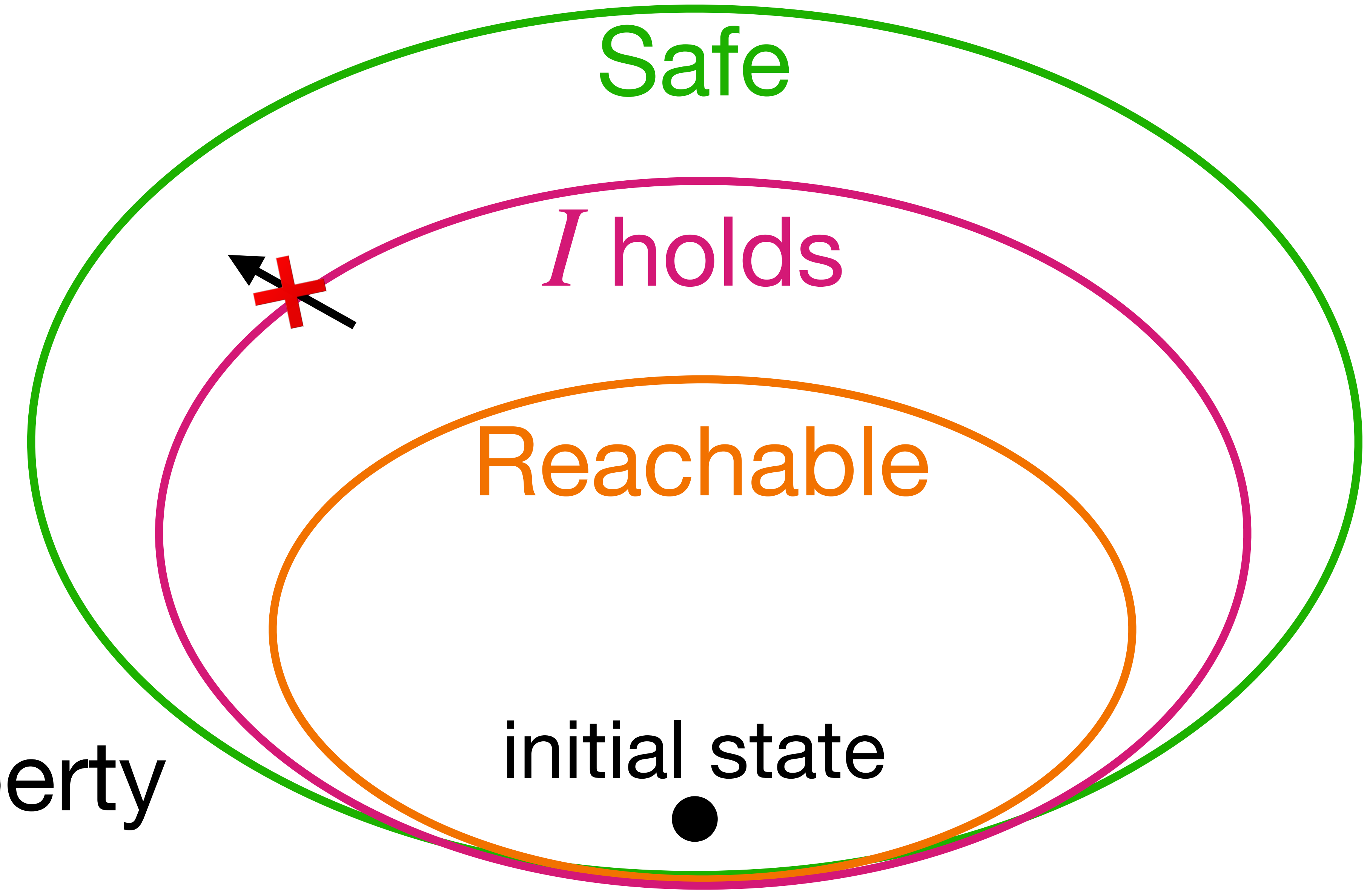
Proving Safety Properties

- Reachability is inductively defined, but proving safety directly by induction may be infeasible
    - Since safety is weak when used as the induction hypothesis



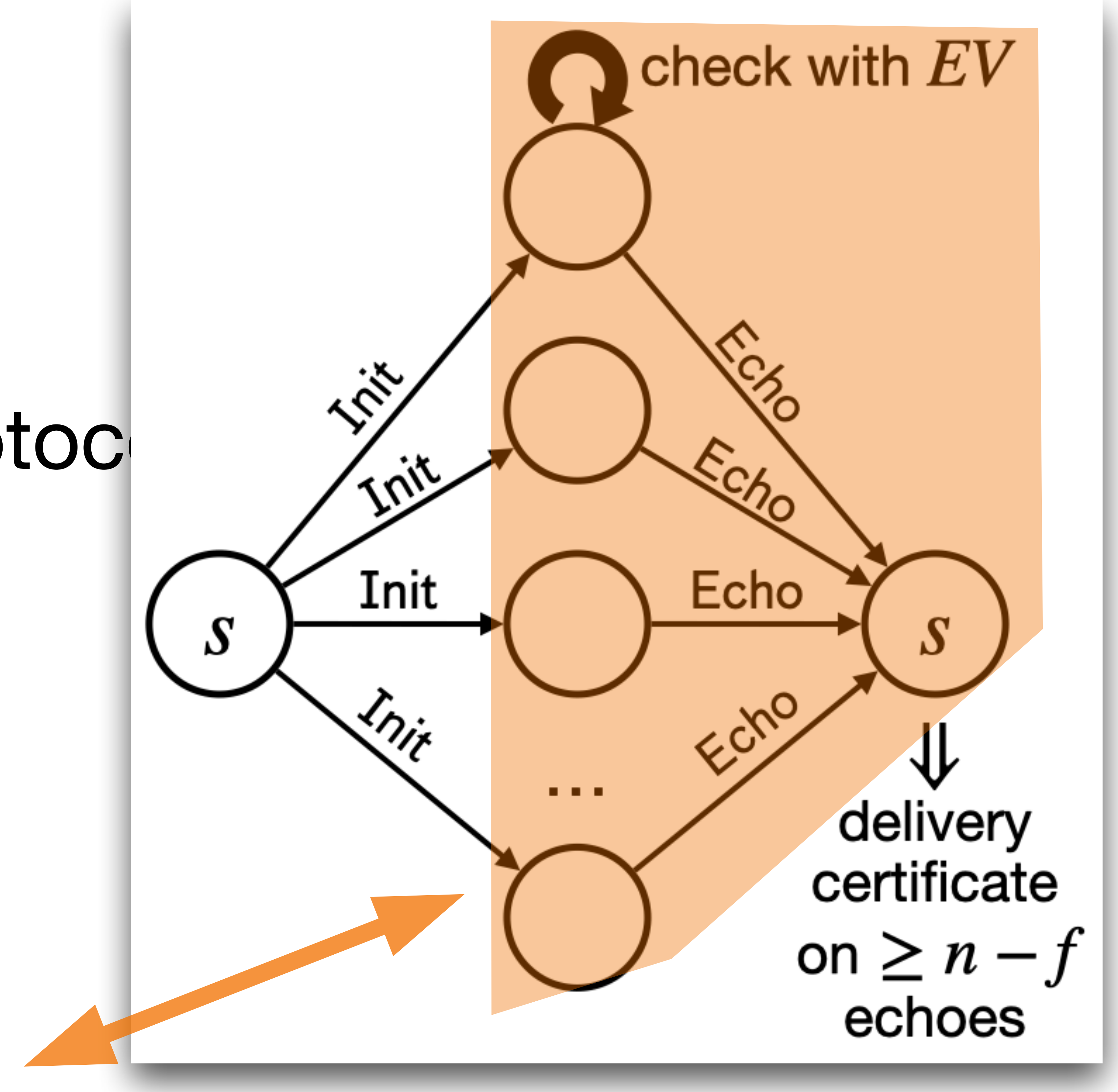
Proving Safety Properties

- The standard approach to proving safety:
  - Finding an inductive invariant I
    - Inductive: I is preserved after any transition
  - Showing that I implies the desired safety property



Inductive Invariants

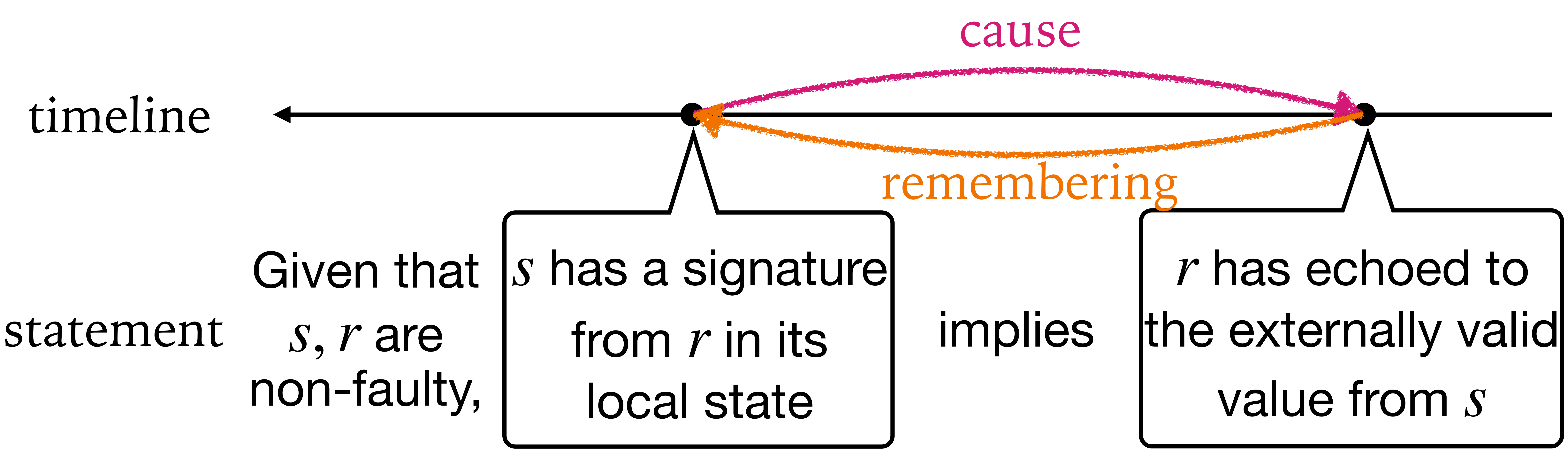
- Summarize the *knowledge* (or, causality) about protoc system state



statement Given that s, r are non-faulty, $\boxed{s \text{ has a signature from } r \text{ in its local state}}$ implies $\boxed{r \text{ has echoed to the externally valid value from } s}$

Inductive Invariants

- Summarize the *knowledge* (or, causality) about protocol execution “within” a system state



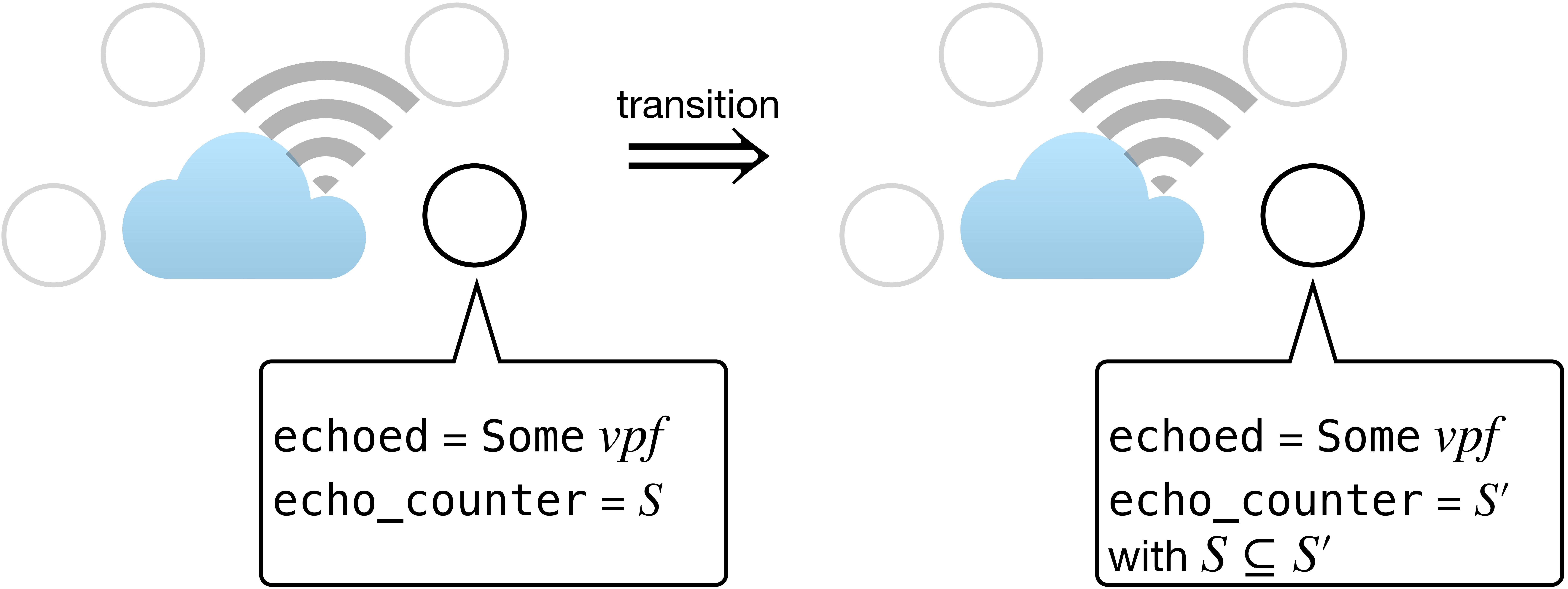
Knowledge Lemmas

- Coming up with all such knowledge that helps prove safety all at once is hard

- **Knowledge lemmas**:
  - Systematically capturing low-level properties of the protocol that ***directly*** follow from the protocol design
  - Higher-level knowledge can be obtained by composing knowledge lemmas

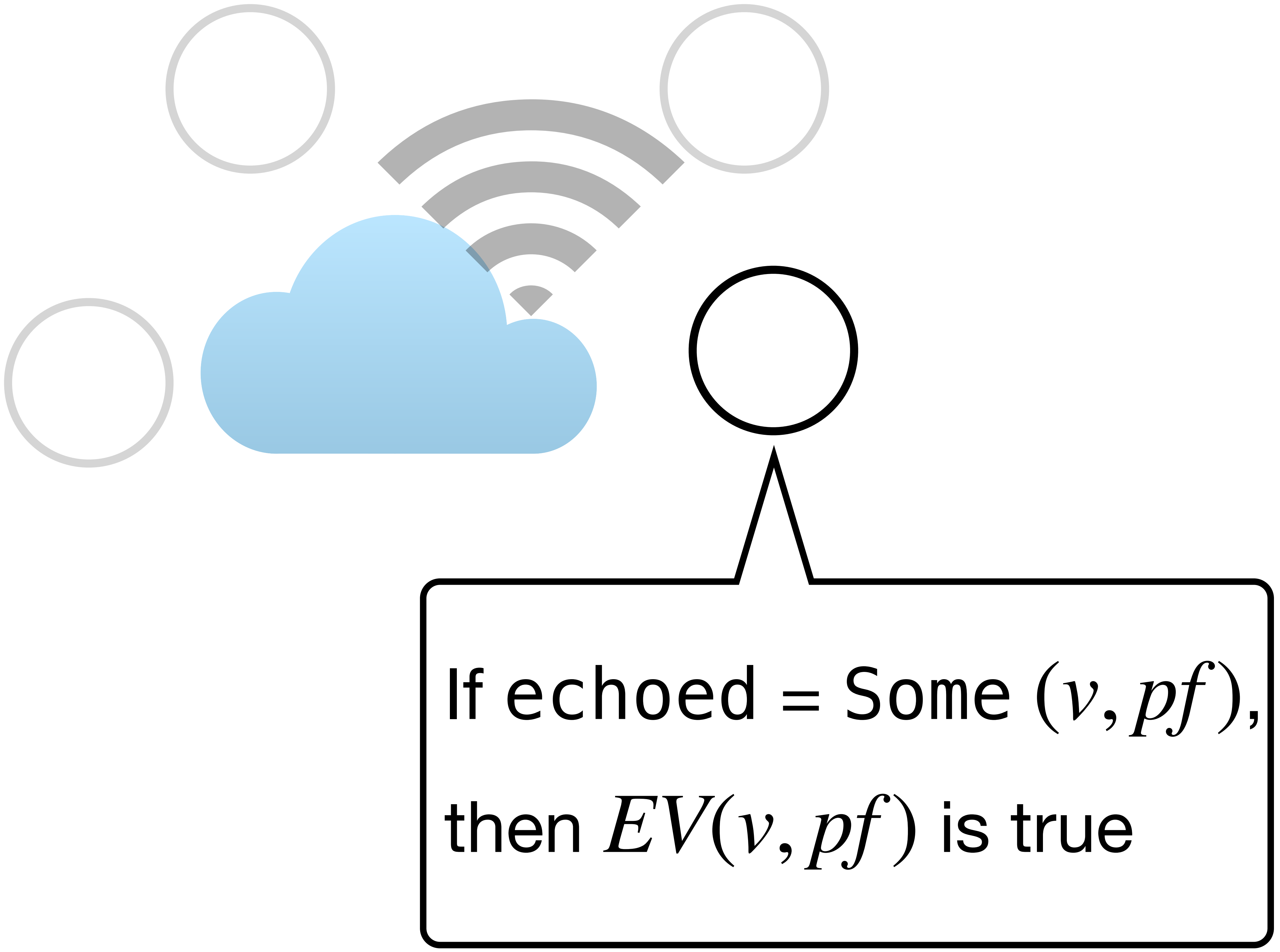
Knowledge Lemmas, Classified

- Data persistence: “a field only grows or never gets overwritten”



Knowledge Lemmas, Classified

- Data representation: “local invariants” maintained inside the local state



Knowledge Lemmas, Classified

- Knowledge propagation within a node: ***direct*** causal relationship within ***multiple*** fields of the local state

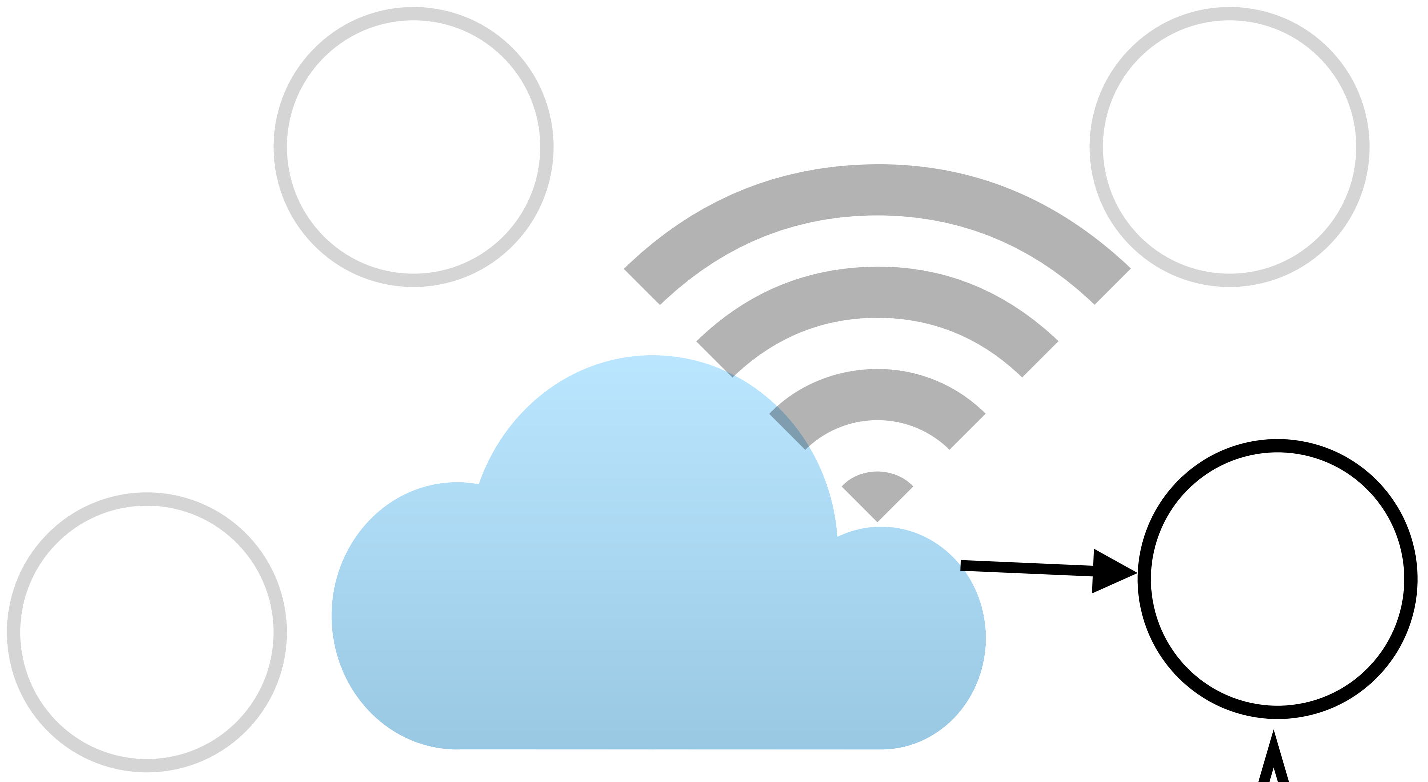
If `delivery_certificate` = `Some` d,

then d is made of signatures in `echo_counter`

and $|\texttt{echo\_counter}| \geq n - f$

Knowledge Lemmas, Classified

- Knowledge propagation through messages: ***direct***, mutual effect between non-faulty nodes and messages sent from or to them



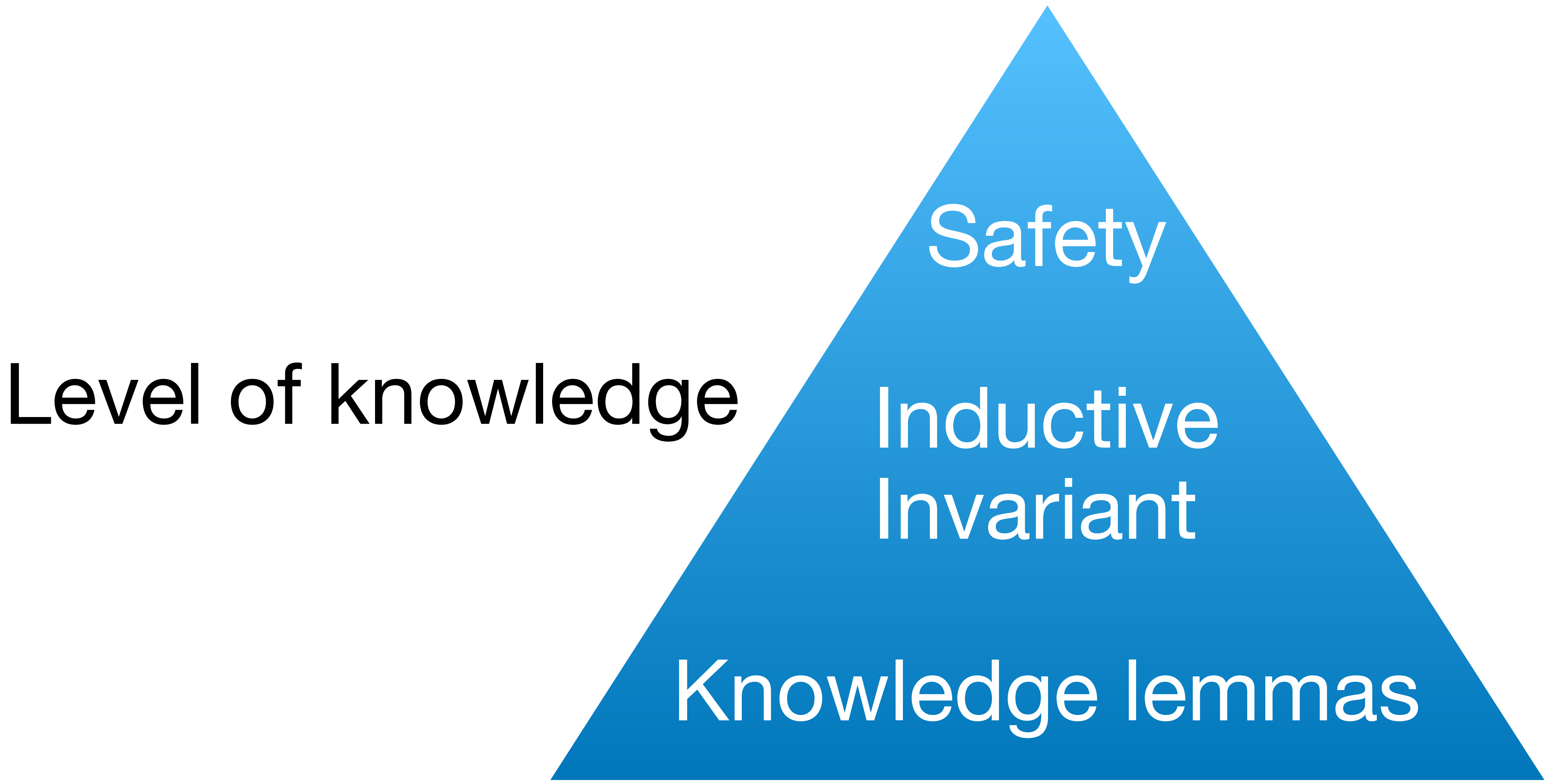
If $(r, sig) \in$ `echo_counter`, then the node must have received `Echo` sig from r

Devising Knowledge Incrementally

- Knowledge lemmas facilitates incremental construction of inductive invariants
  - Devising knowledge lemmas does not require much intellectual burden
  - More knowledge can be devised by composing existing knowledge

Devising Knowledge Incrementally

- Knowledge lemmas facilitates incremental construction of inductive invariants
  - Devising knowledge lemmas does not require much intellectual burden
  - More knowledge can be devised by composing existing knowledge



Knowledge-Driven Proof of Safety

Given that s, r are non-faulty,

s has a signature from r in its local state

implies

r has echoed to the externally valid value from s

knowledge propagation through message

data persistence & data representation

s has received an Echo message from r; or

r has sent an Echo message to s

- Safety is then just the knowledge derived from existing knowledge!

Specifying Systems in BYTHOS

Workflow

Encoding the protocol

Proving safety properties

Reasoning about liveness

Composing protocols

Verifying composite protocols

Techniques

Knowledge lemmas

Specifying Systems in BYTHOS

Workflow

Encoding the protocol

Proving safety properties

Reasoning about liveness

Composing protocols

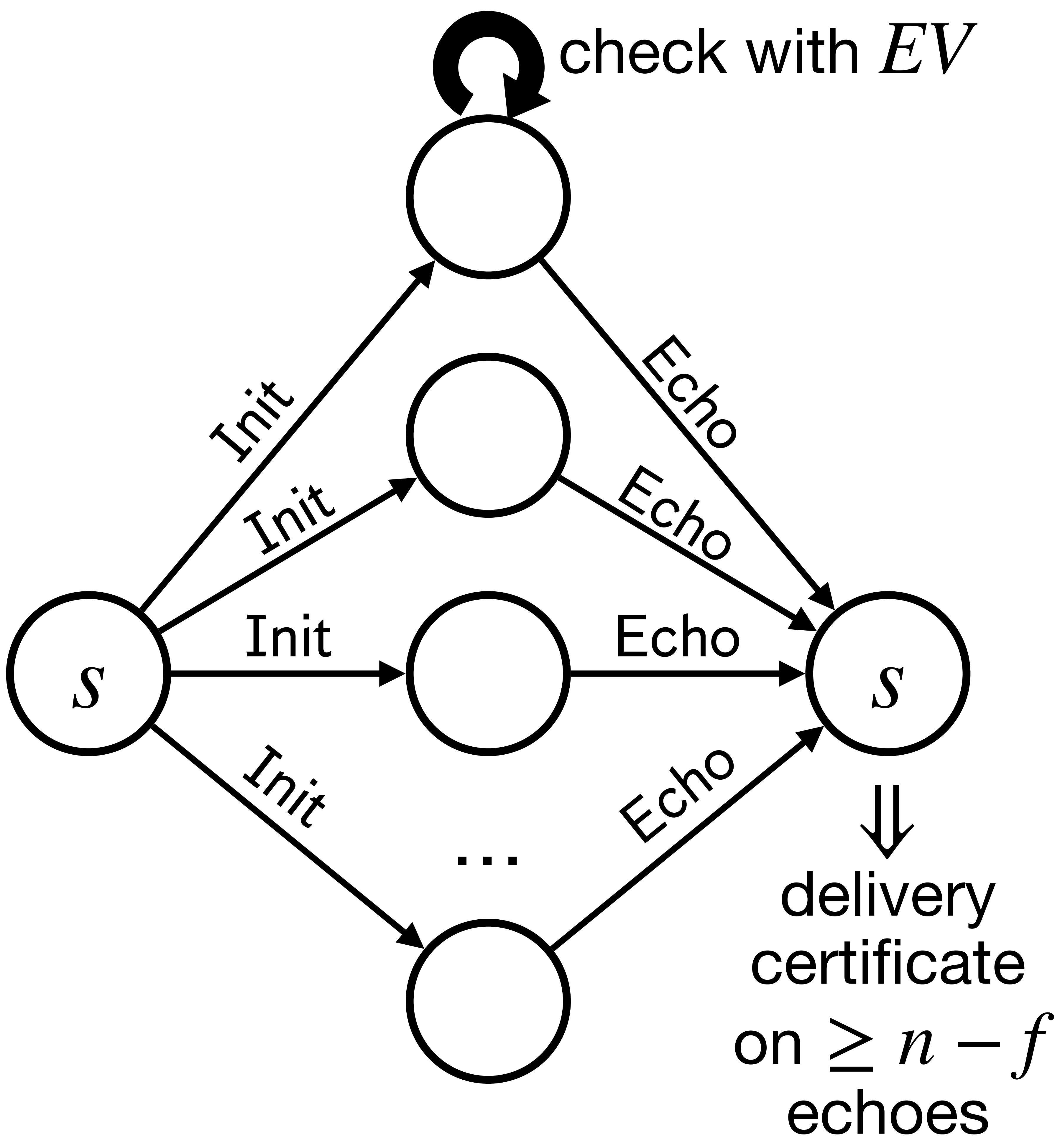
Verifying composite protocols

Techniques

Knowledge lemmas

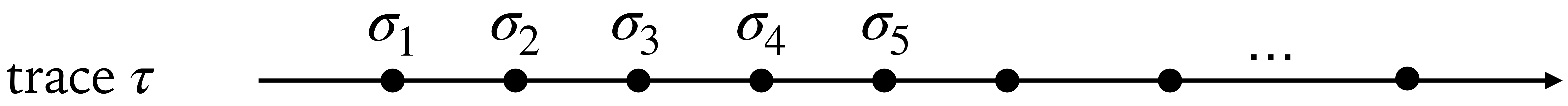
Liveness Property of Provable Broadcast

- Liveness ("good thing eventually happens"):
  - Given that s is non-faulty and v is externally valid, if s broadcast v, then s will eventually obtain a delivery certificate for v



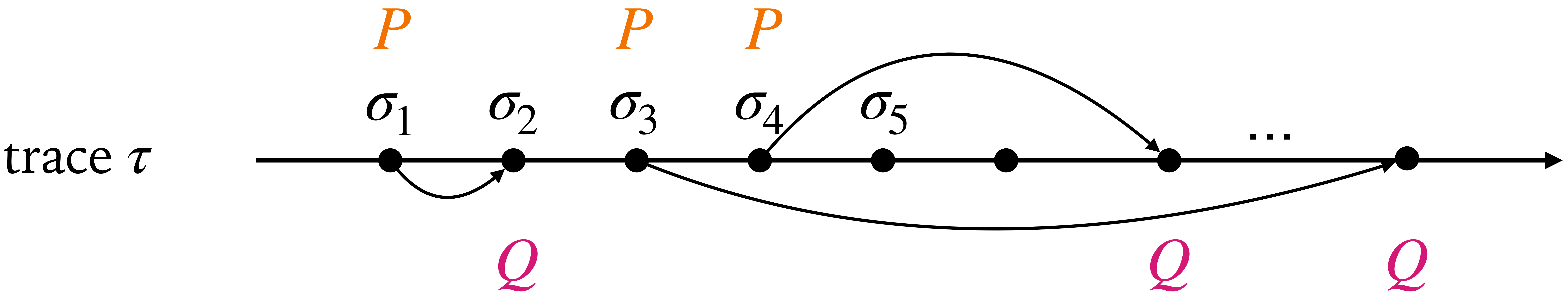
Liveness Property, Formalized

- A liveness property is a predicate on the infinite-length traces of system states
  - Can be expressed in the language of Linear Temporal Logic (LTL)



Liveness Property, Formalized

- A liveness property is a predicate on the infinite-length traces of system states
  - Can be expressed in the language of Linear Temporal Logic (LTL)



- "If P happens, then eventually Q will happen": formalized as "leads-to" $\leadsto$
  - τ satisfies $\ulcorner P \urcorner \leadsto \ulcorner Q \urcorner$ if at any moment when P holds, then there exists a subsequent moment when Q will hold

Liveness Property, Formalized

- Liveness properties would only hold on “reasonable” traces
  - E.g., a trace with only Byzantine nodes moving is not reasonable to consider
- Fairness condition: “reasonableness” in the form of LTL formula
  - The fairness condition in BYTHOS: every message between non-faulty nodes will be eventually received
    - Unrelated to clock or Byzantine nodes, due to the presence of asynchrony and adversary

Liveness Property, Formalized

- Enable temporal logic reasoning by using the CoqTLA library

tchajed / coq-tla

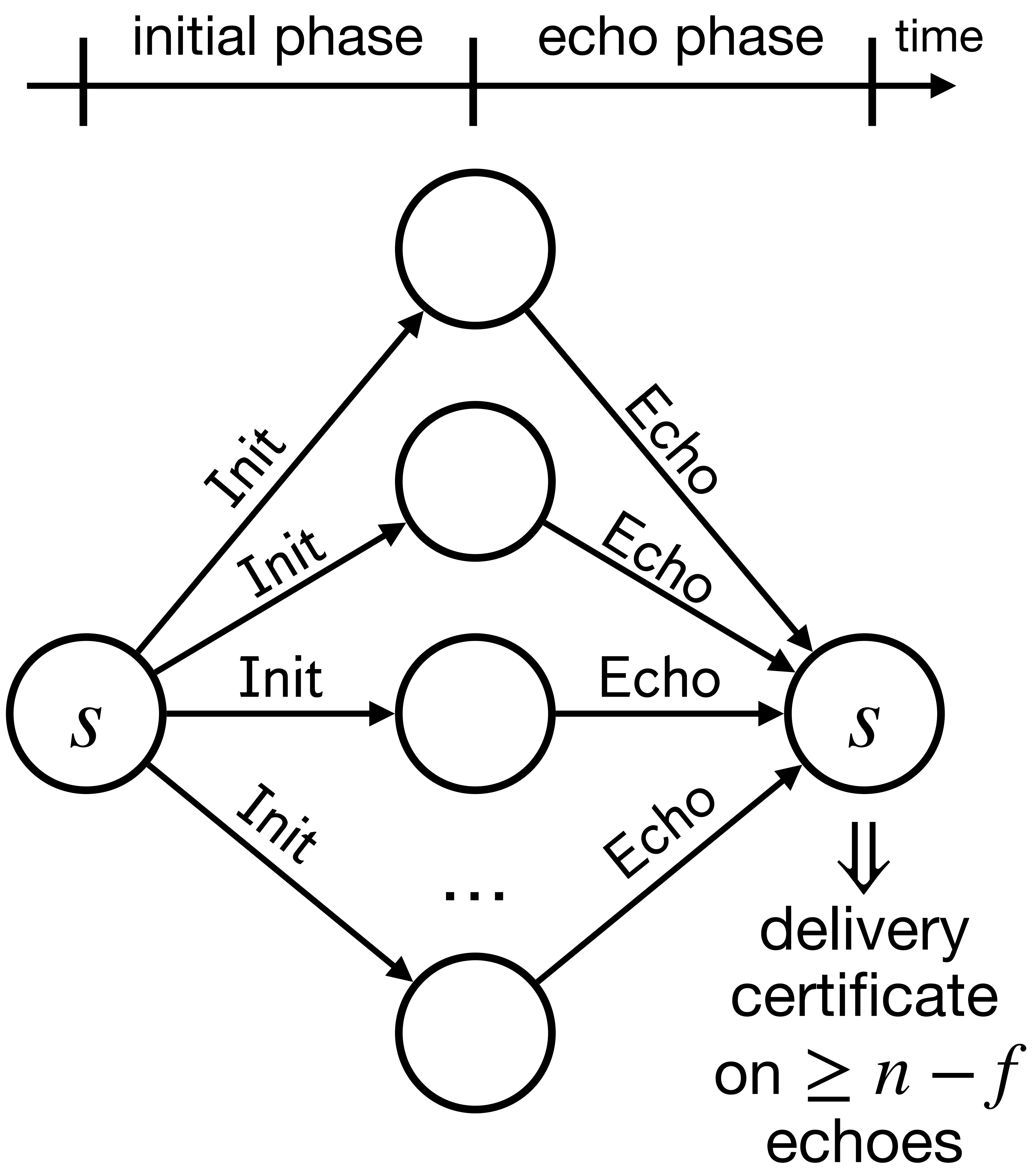
```
Definition liveness : Prop :=
  forall (s : Address) (v : Value),
    isByzantine s = false ∧
    externally_valid v →
    ⌜ init ⌝ ∧ □ ⟨ next ⟩ ∧ fairness ⊢
    ⌜ (* s broadcast v *) ⌝ ~>
      ⌜ (* s has delivery certificate for v *) ⌝.
```

wellformedness condition

"any trace satisfying conditions before ⊢ would satisfy those after"

Reasoning about Liveness

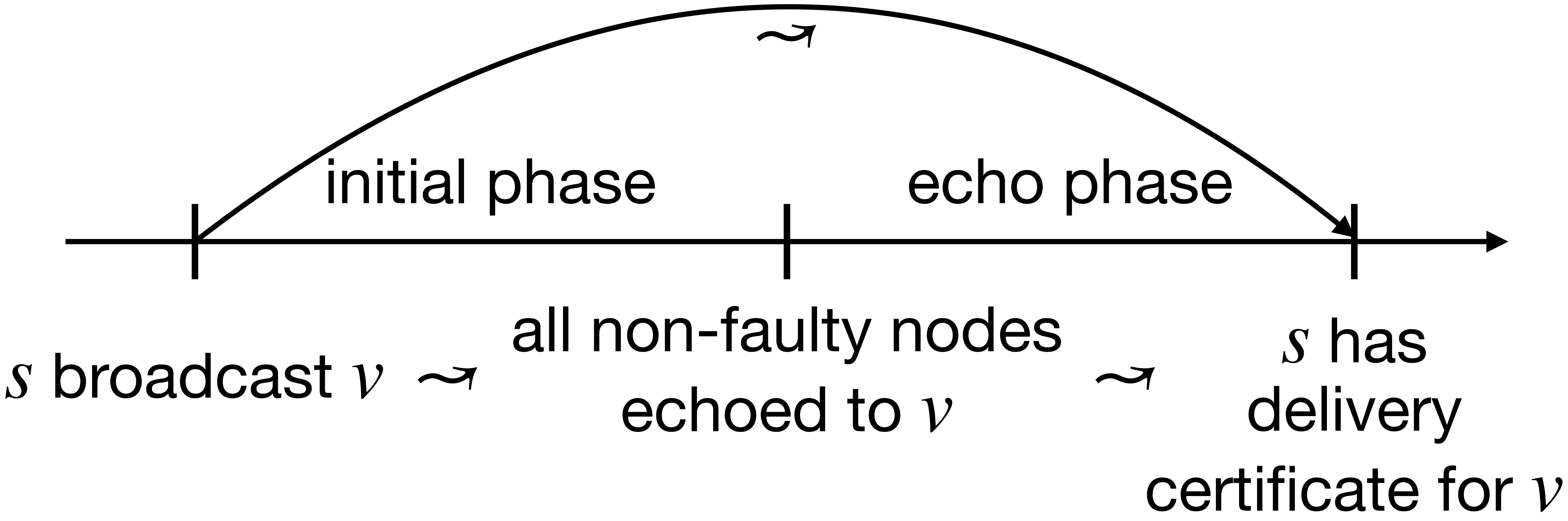
- Several “**phases**” can be identified in the protocol execution
- Proving liveness amounts to showing that these phases are guaranteed to happen consecutively, assuming fairness



Phase Decomposition

- Phases can be proved separately and be composed using the transitivity of $\leadsto$

$$\forall P, Q, R,\ P \leadsto Q \land Q \leadsto R \vdash P \leadsto R$$



Specifying Systems in BYTHOS

Workflow

- Encoding the protocol
- Proving safety properties
- Reasoning about liveness
- Composing protocols
- Verifying composite protocols

Techniques

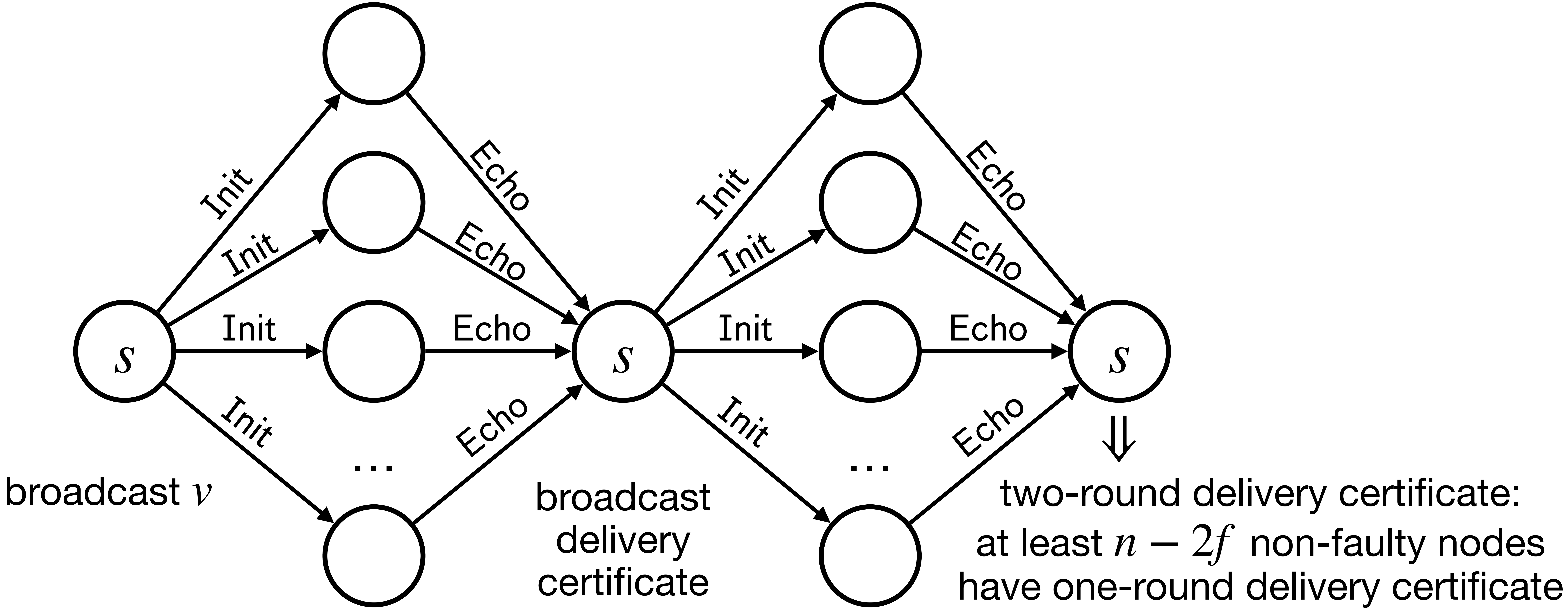
- Knowledge lemmas
- Phase decomposition

Specifying Systems in BYTHOS

Workflow	Techniques
Encoding the protocol	
Proving safety properties	Knowledge lemmas
Reasoning about liveness	Phase decomposition
Composing protocols	
Verifying composite protocols	

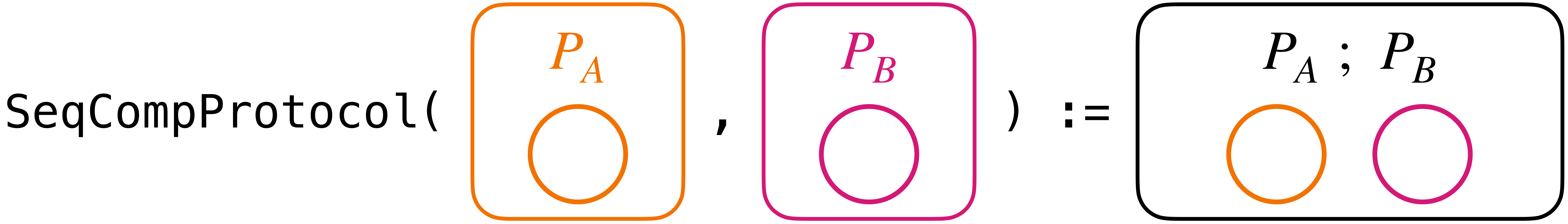
Sequential Composition of Protocols

- Sequencing protocols help achieve stronger guarantees



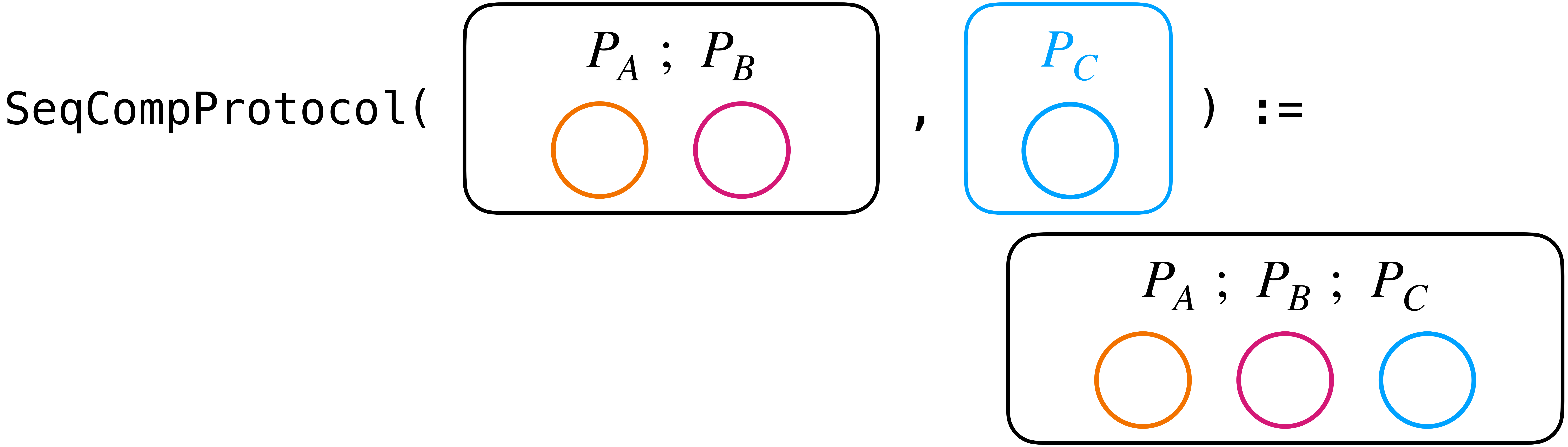
Functor for Protocol Composition

- The protocol logic of a protocol is encapsulated as a Coq module
- **Composition functor**: given two protocol modules, constructs a new one



Functor for Protocol Composition

- The protocol logic of a protocol is encapsulated as a Coq module
- **Composition functor**: given two protocol modules, constructs a new one
  - Allows for multiple composition



Composite Protocol Construction

- The composite protocol reuses definitions from sub-protocols
  - The local state of $P_A \ ; \ P_B$ = the pair of local states of P_A and P_B
  - The kinds of messages of $P_A \ ; \ P_B$ = the union of messages of P_A and P_B
- A node running $P_A \ ; \ P_B \approx$ two threads running P_A and P_B separately
  - Exception: P_B is instructed to start by the user-provided **triggers**

Triggers

- Firing internal events of P_B based on the execution of P_A

```
Parameter trigger_procMsg :
  P_A.State (* local state before executing P_A.procMsg *) ->
  P_A.State (* local state after executing P_A.procMsg *) ->
  option P_B.InternalEvent.
Parameter trigger_procInt : (* the same type as above *)
```

Triggers

- The logic of `procMsg` of $P_A \; ; \; P_B$:
  - If the incoming message is for P_B, then handle it using the `procMsg` of P_B
  - Otherwise:
    - Handle it using the `procMsg` of P_A
    - Check whether the trigger for `procMsg` is fired
    - If the trigger gives the internal event ev of P_B, then handle it using the `procInt` of P_B

Specifying Systems in BYTHOS

Workflow

- Encoding the protocol
- Proving safety properties
- Reasoning about liveness
- Composing protocols
- Verifying composite protocols

Techniques

- Knowledge lemmas
- Phase decomposition
- Composition functor

Specifying Systems in BYTHOS

Workflow

Encoding the protocol

Proving safety properties

Reasoning about liveness

Composing protocols

Verifying composite protocols

Techniques

Knowledge lemmas

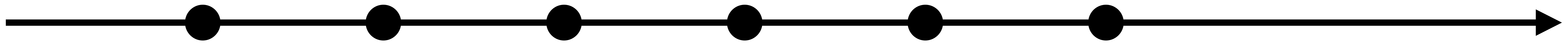
Phase decomposition

Composition functor

Composing Proofs

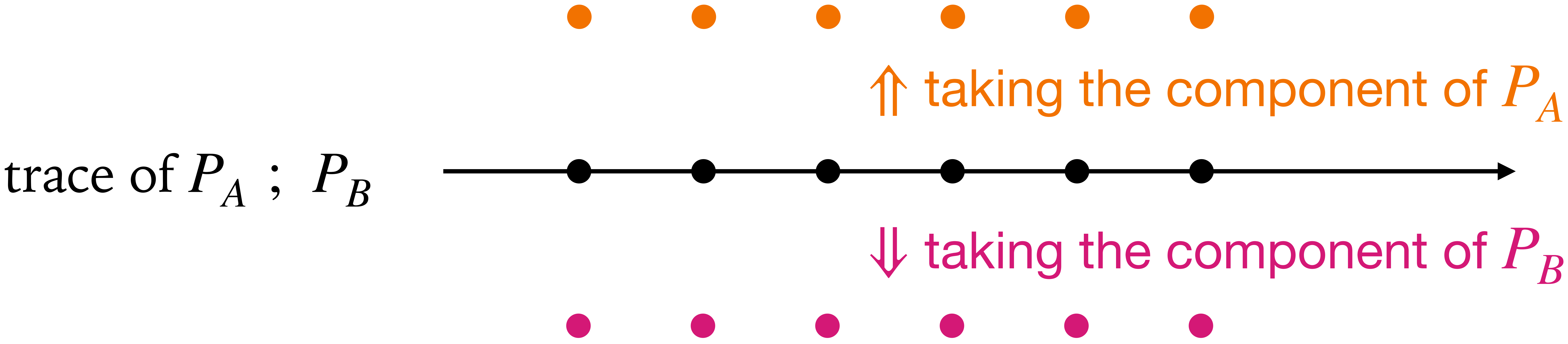
- The execution of a composite protocol can be projected into the executions of sub-protocols

trace of $P_A \; ; \; P_B$



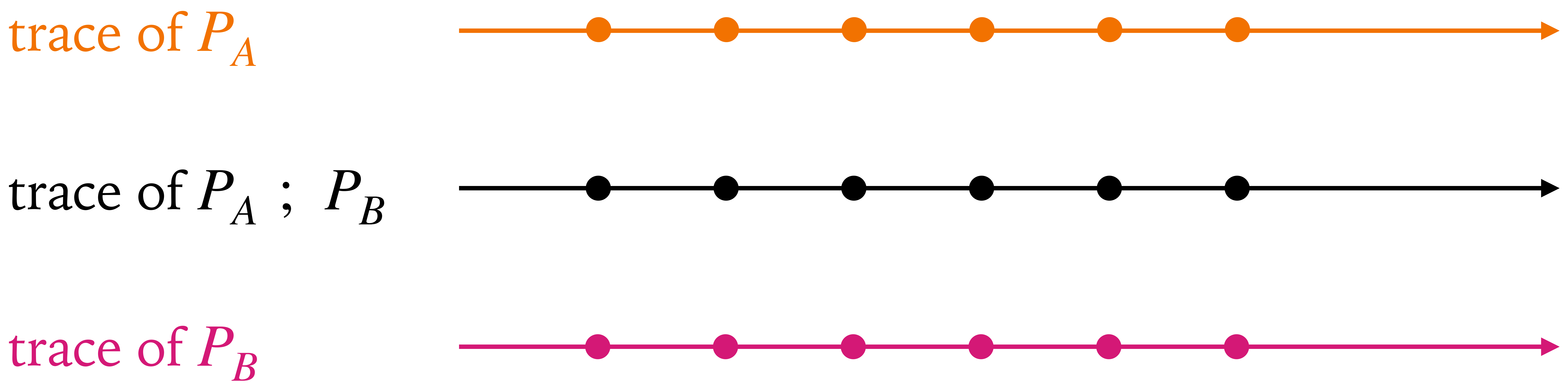
Composing Proofs

- The execution of a composite protocol can be projected into the executions of sub-protocols



Composing Proofs

- The execution of a composite protocol can be projected into the executions of sub-protocols
  - Allows for composing proofs of sub-protocols by **lifting**

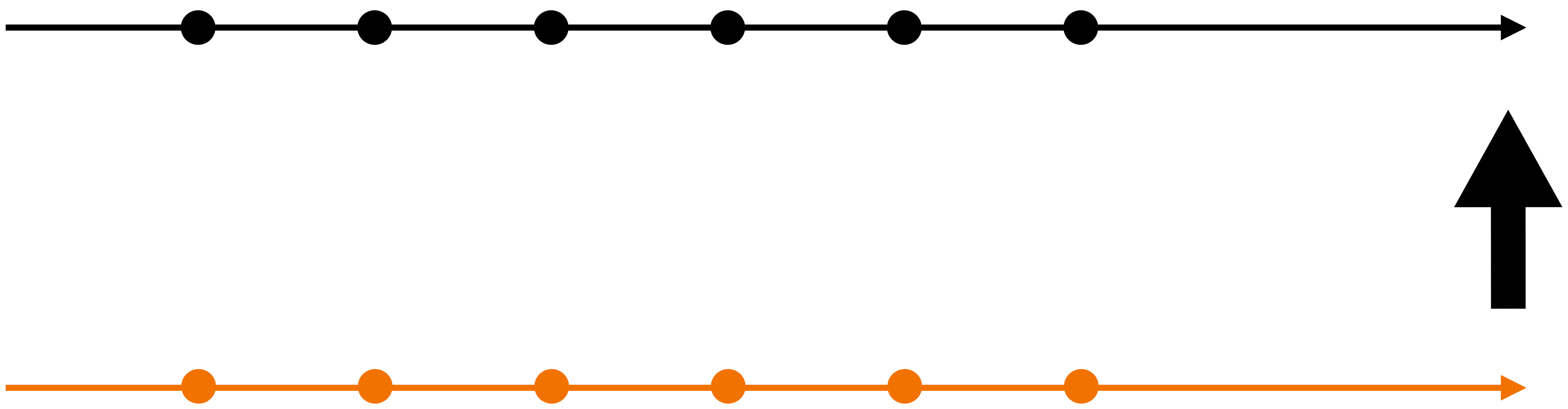


Lifting Safety

- “If every reachable system state of P_A satisfies some safety, then the P_A components of $P_A\ ;\ P_B$ would also satisfy it”

holds on $proj_A(\sigma)$ for every reachable σ

trace of $P_A\ ;\ P_B$



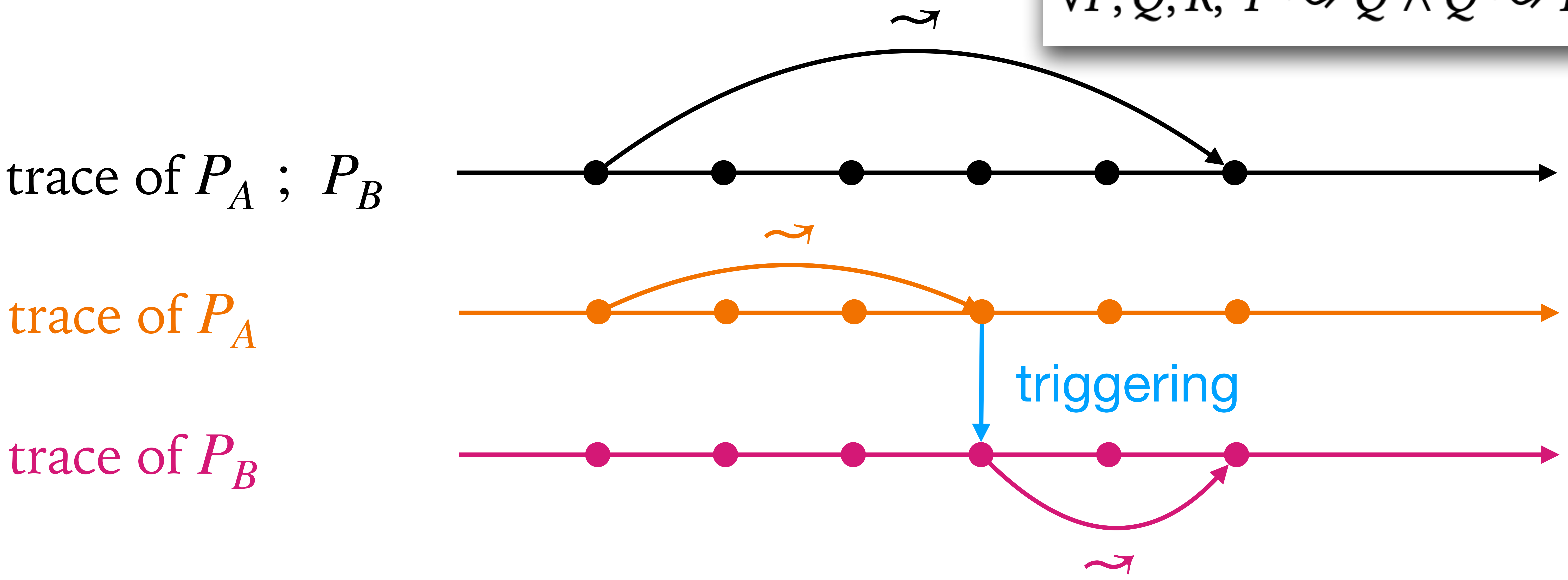
trace of P_A

holds on every reachable σ

Lifting and Composing Liveness

- Liveness properties of sub-protocols can be lifted and composed
  - Requires triggers to be fired properly

$$\forall P, Q, R,\ P \leadsto Q \land Q \leadsto R \vdash P \leadsto R$$



Specifying Systems in BYTHOS

Workflow	Techniques
Encoding the protocol	
Proving safety properties	Knowledge lemmas
Reasoning about liveness	Phase decomposition
Composing protocols	Composition functor
Verifying composite protocols	Proof lifting

Verified Case Studies

- Provable Broadcast
- Reliable Broadcast
- Accountable Confirmer
- Accountable Reliable Broadcast

Case Study: Reliable Broadcast

INFORMATION AND COMPUTATION 75, 130–143 (1987)

Asynchronous Byzantine Agreement Protocols

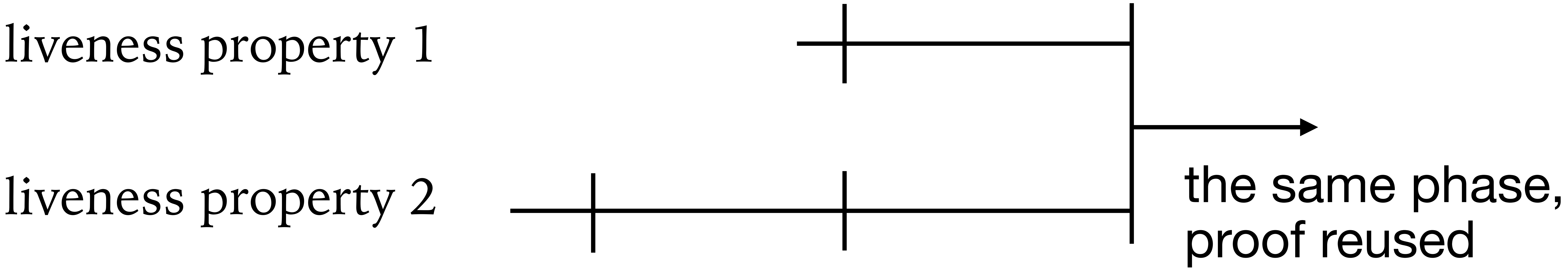
GABRIEL BRACHA

13Bart Street, Tel-Aviv 69104, Israel

- A classic BFT protocol for broadcasting values with several guarantees
  - Used as sub-protocol in some BFT consensus protocols (e.g., Bullshark)

Proof Reuse in Liveness Proofs

- The proof of one phase can be used in proving different liveness properties
  - 5 phases in total, but only need to prove 4 phases



Case Study: Accountable Confirmer

As easy as ABC: Optimal (A)ccountable (B)yzantine (C)onsensus is easy!

Pierre Civit[1], Seth Gilbert[2], Vincent Gramoli[3,4], Rachid Guerraoui[4] and Jovan Komatovic[4]

[1]Sorbonne University, CNRS, LIP6
[2]NUS Singapore
[3]University of Sydney
[4]EPFL

- A generic “plug-in” providing BFT protocols with *accountability*
  - Allows non-faulty nodes to detect Byzantine culprits when the safety is compromised due to too many Byzantine nodes

Uncovering Implicit Assumptions

- The protocol implicitly assumes the existence of a message buffer, while the pseudo-code does not mention it
    - Without the buffer the protocol may not be live
- Evidence that formal verification can uncover subtle issues!

Case Study: Accountable Reliable Broadcast

- Sequential composition of Accountable Confirmer and Reliable Broadcast
  - Providing Reliable Broadcast with accountability
- It only takes 7 lines of proof to show the composite liveness property!

Proof Efforts

- In total: around 7100 lines of Coq code

Library	Component	Spec	Proof	Total
Bythos (Sec. 3)	System (Sec. 3.1)	729	465	1194
	Liveness (Sec. 3.2)	160	181	341
	Composition (Sec. 3.3)	329	255	584
	Utilities	184	157	341
	Total	1402	1058	2460
Provable Broadcast (Sec. 2)	Implementation (Sec. 2.1)	121	6	127
	Safety (Sec. 2.2)	404	320	724
	Liveness (Sec. 2.3)	92	67	159
	Composition (Sec. 2.4)	85	10†	95
	Total	702	403	1105
Reliable Broadcast (Sec. 4.1)	Implementation	130	6	136
	Safety (Sec. 4.1.1)	448	432	880
	Liveness (Sec. 4.1.2)	144	161	305
	Total	722	599	1321
Accountable Confirmer (Sec. 4.2)	Implementation	237	109	346
	Safety	619	709	1328
	Liveness (Sec. 4.2.2)	172	200	372
	Total	1028	1018	2046
Accountable Reliable Broadcast (Sec. 4.3)	Implementation	33	0	33
	Connector (Sec. 4.3.1)	48	92	140
	Liveness (Sec. 4.3.1)	3	7	10
	Total	84	99	183

Summary

- Bythos: streamlining the verification of BFT protocols and their compositions
- Supporting standard toolsets:
  inductive ***invariant based safety*** reasoning and ***LTL-based liveness*** reasoning
- Further facilitating proofs with ***knowledge lemmas*** and phase decomposition
- Allowing verifying ***composite BFT protocols*** by reusing proofs of components

Code

Paper

Thanks!