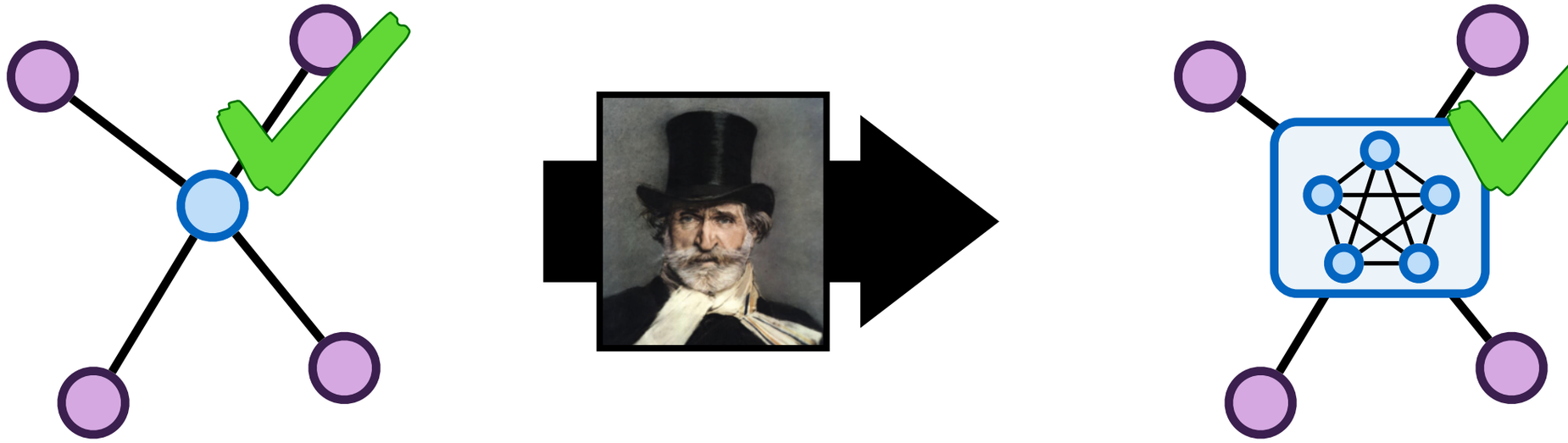
Verifying Distributed Systems



Zachary Tatlock

Verdi Lecture 2 at DeepSpec Summer School 2018

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

PLSE

Distributed Infrastructure

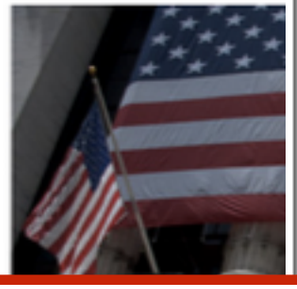
One summer day...

The New York Times

The Stock Market Bell Rings, Compu

By NATHANIEL POPPER JULY 8, 2015

Problems with technology have at times roiled global financial markets, but the 223-year-old New York Stock Exchange has held itself up as an oasis of humans ready to step in when the computers go haywire.



On Wedne
working o
helpless w
the exchar
four hours
bringing a
ceaseless

The excha
action sho

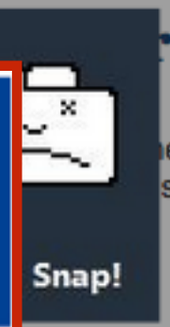
THE WALL STREET JOURNAL.
Digital Network
WSJ.com MarketWatch BARRON'S

THE WALL STREET JOURNAL.

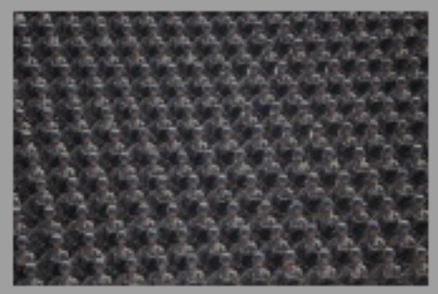
Home

WSJ.com is having technical difficulties. The full site will return shortly.

Cyber Sleuths Trac r to



ent in
state-



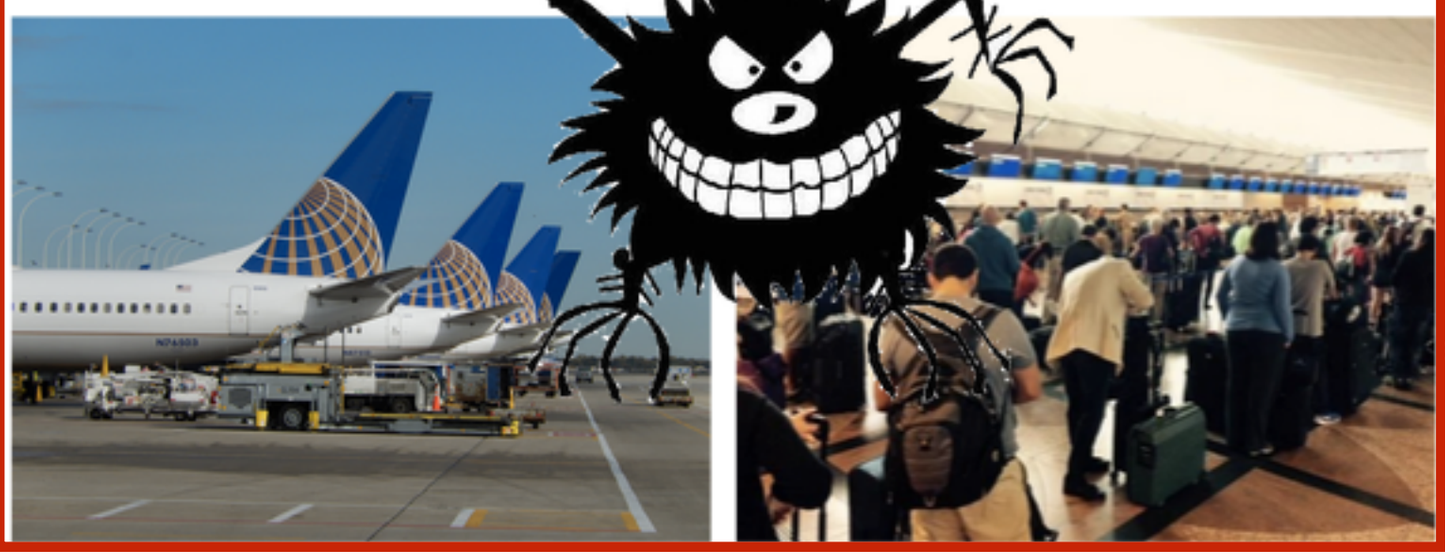
arket for Their Loans

n of college loans are causing snarls
anks may soon ratchet back lending.

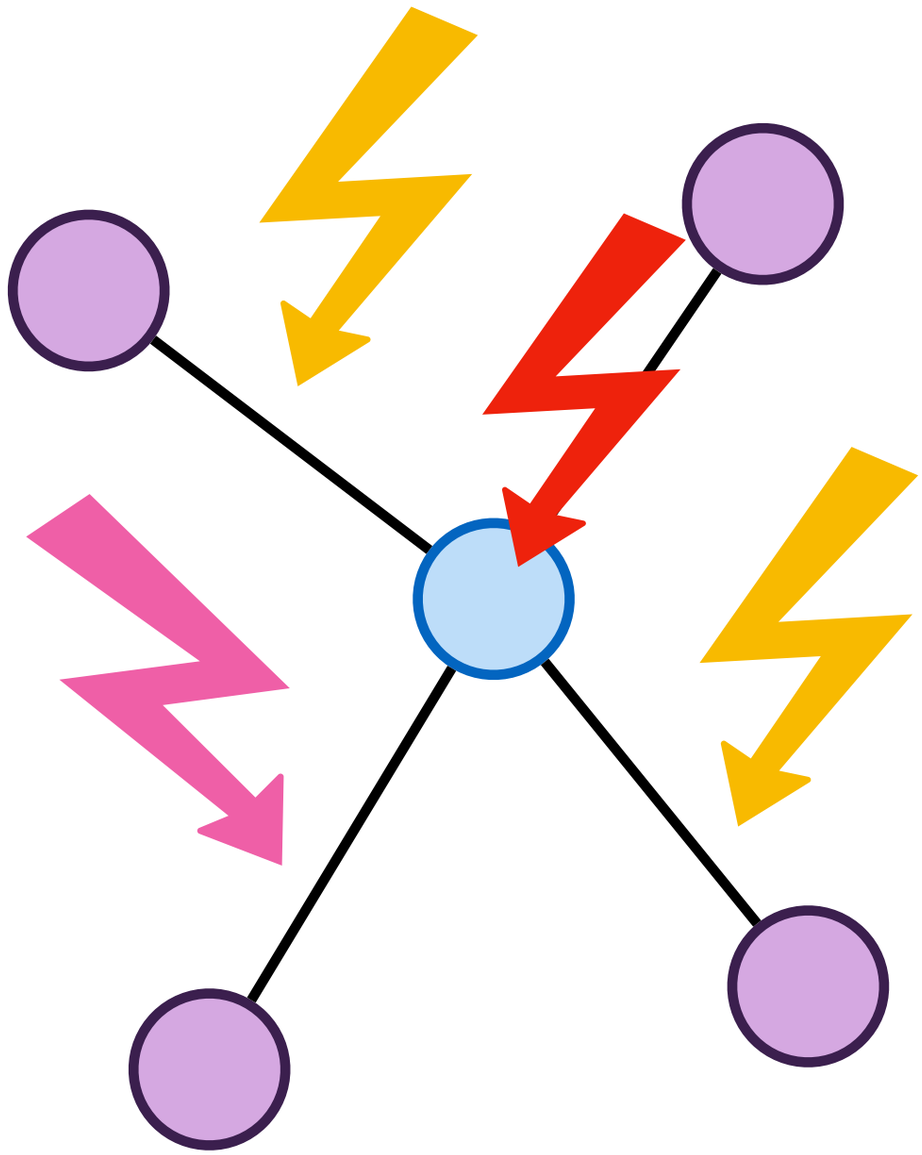
s Trump Humans

, are remaking the $12.7 trillion
s in stock and currency trading.

UNITED



How distributed systems fail



Challenges

concurrency

message drops

message dups

message reorder

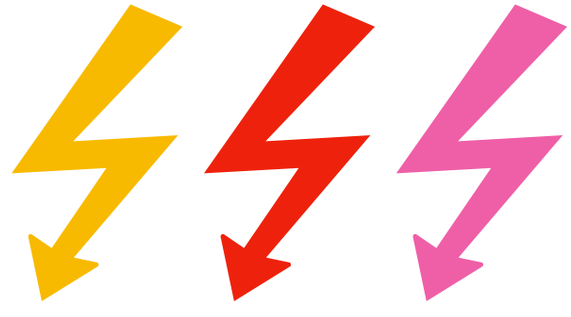
machine crash

machine reboot

Toward verified distributed systems

Formalize *network semantics*

capture how faults can occur

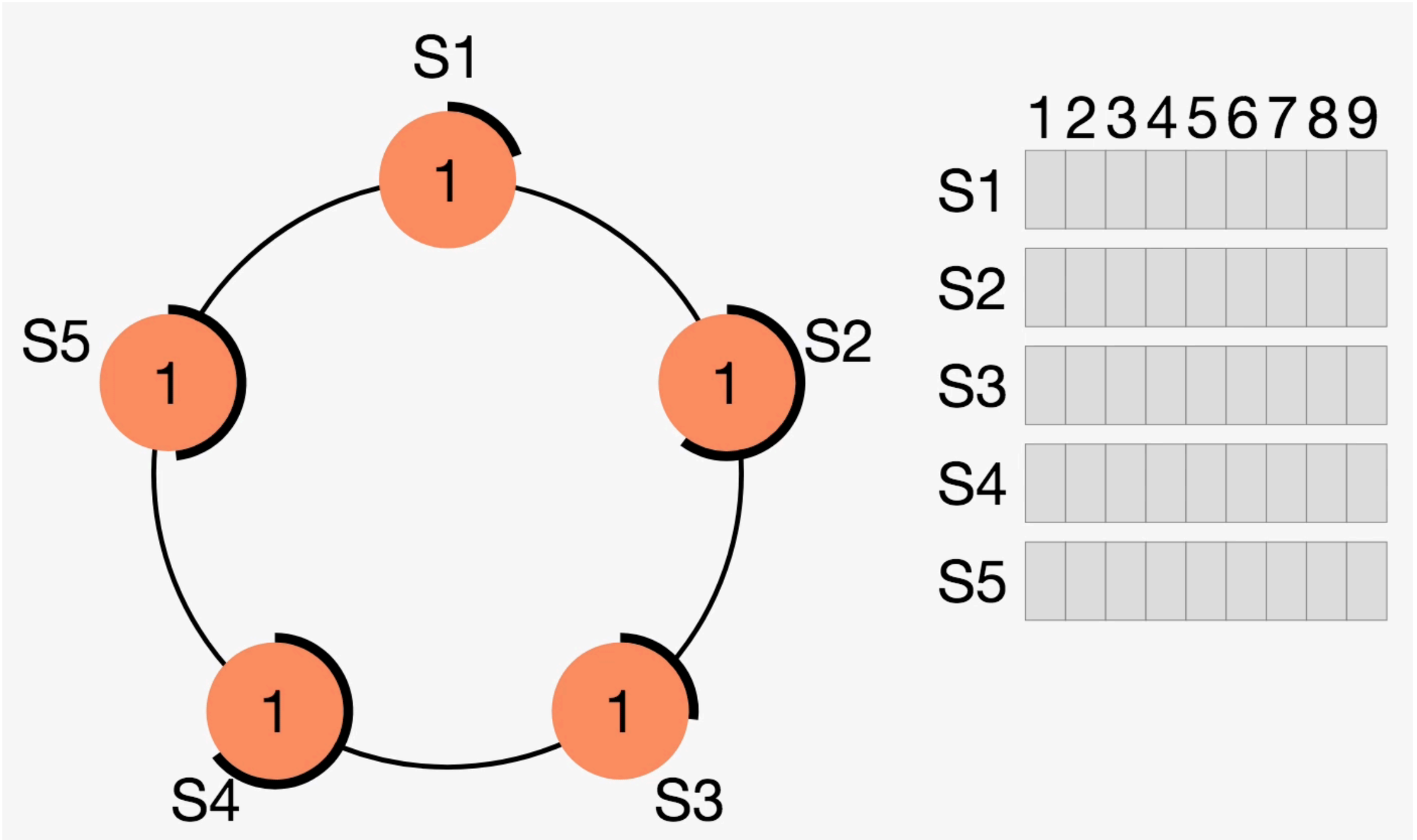


Separate app / fault reasoning

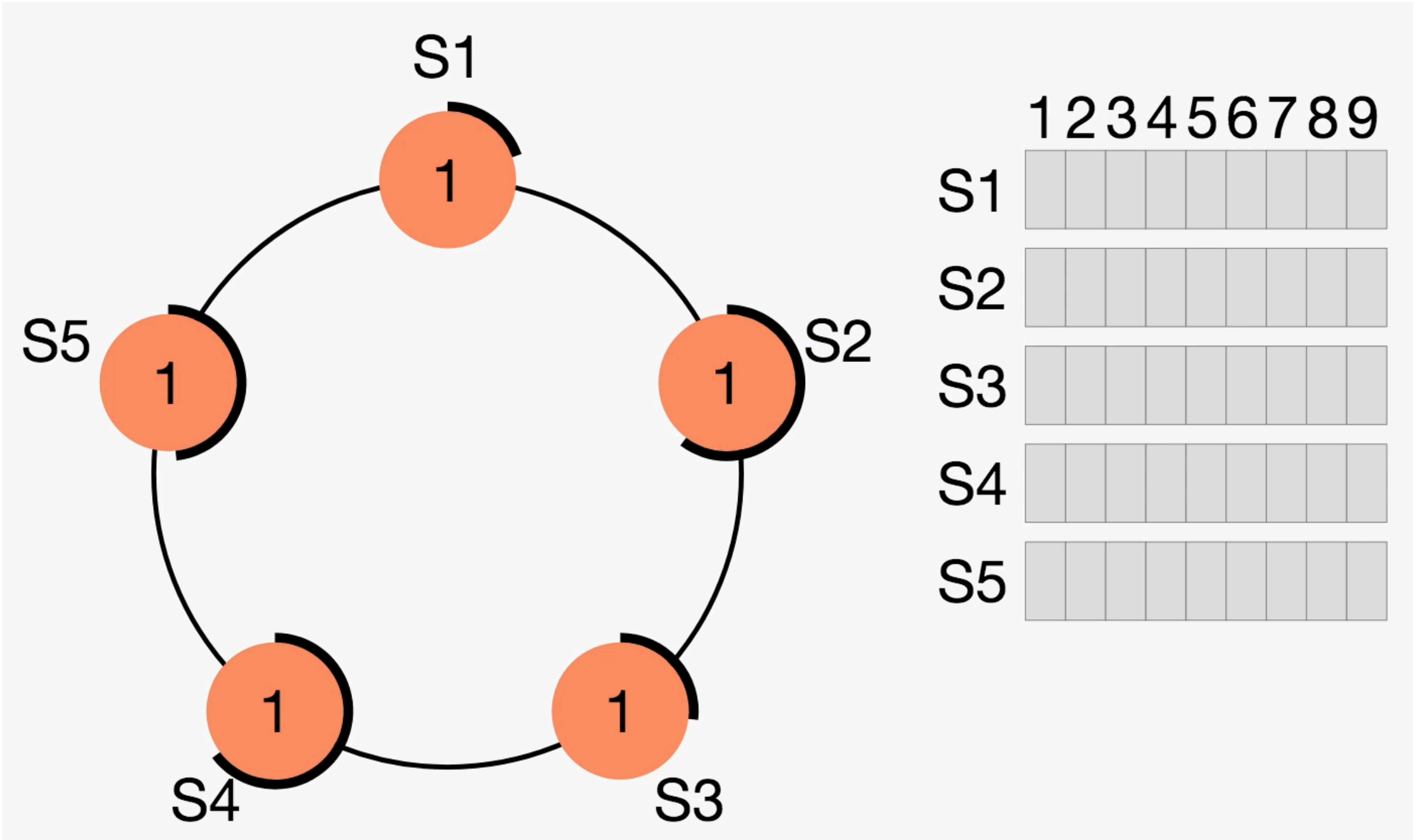
develop and prove in simple fault model

apply generic verified fault handling

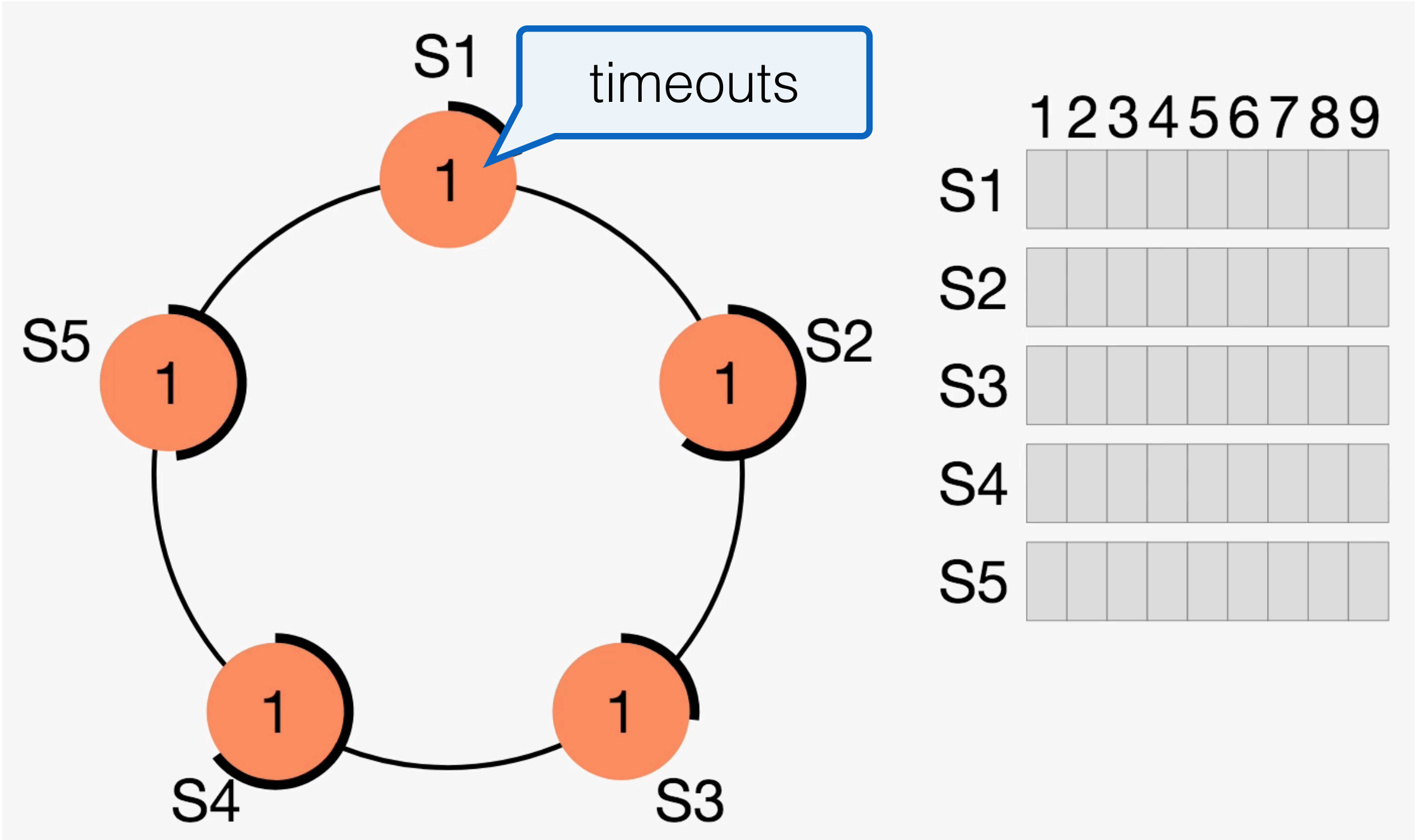
Formalizing distributed systems



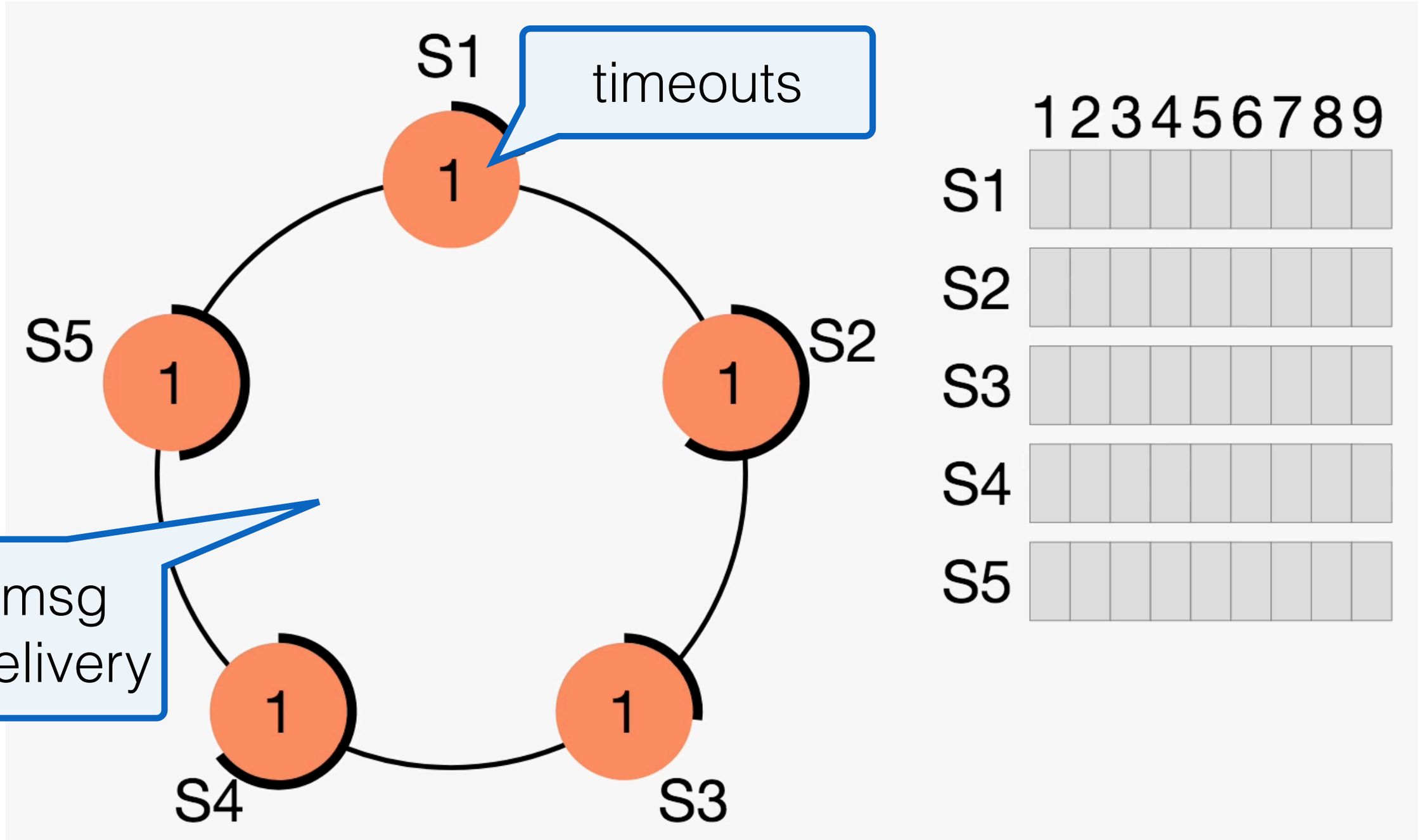
Formalizing distributed systems



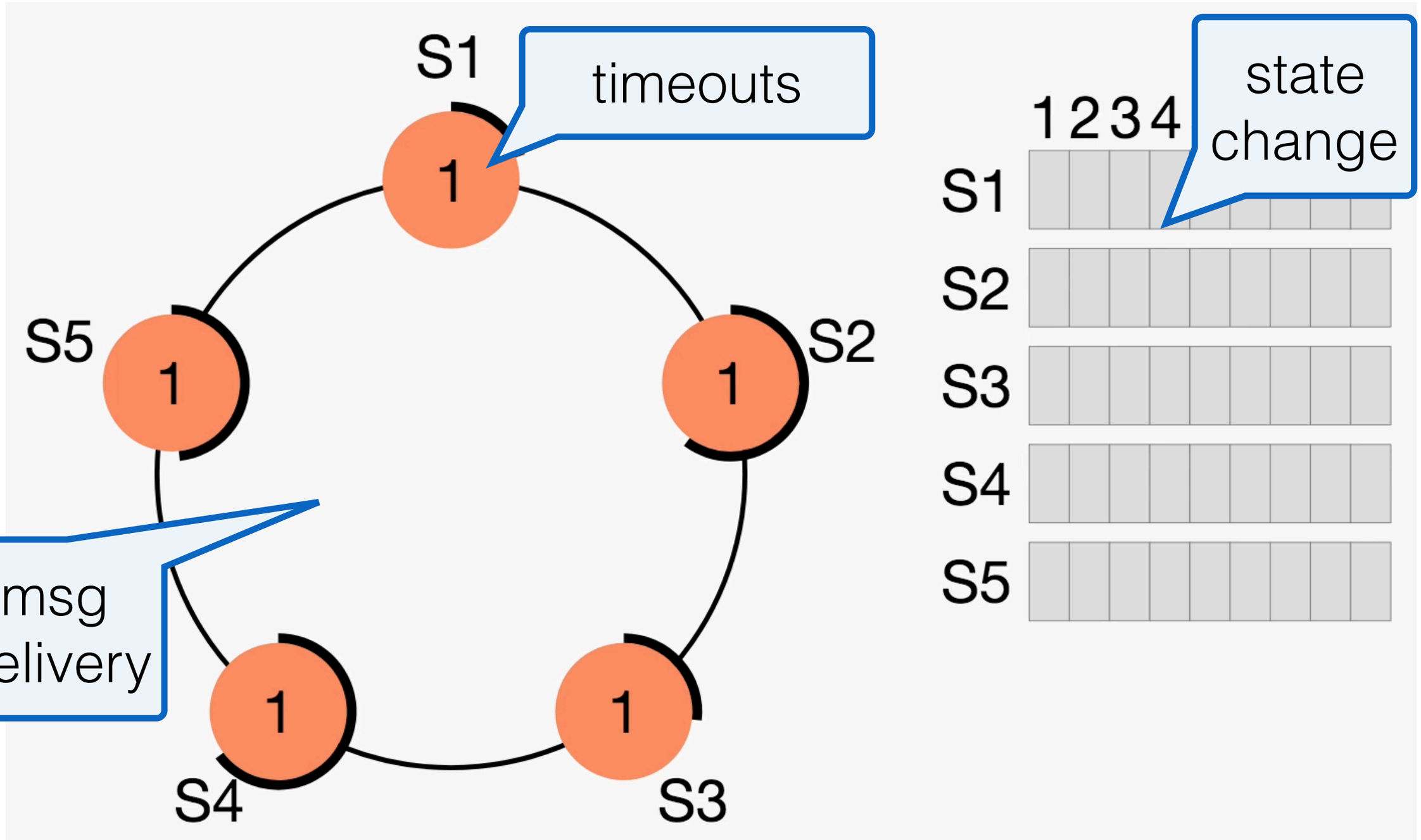
Formalizing distributed systems



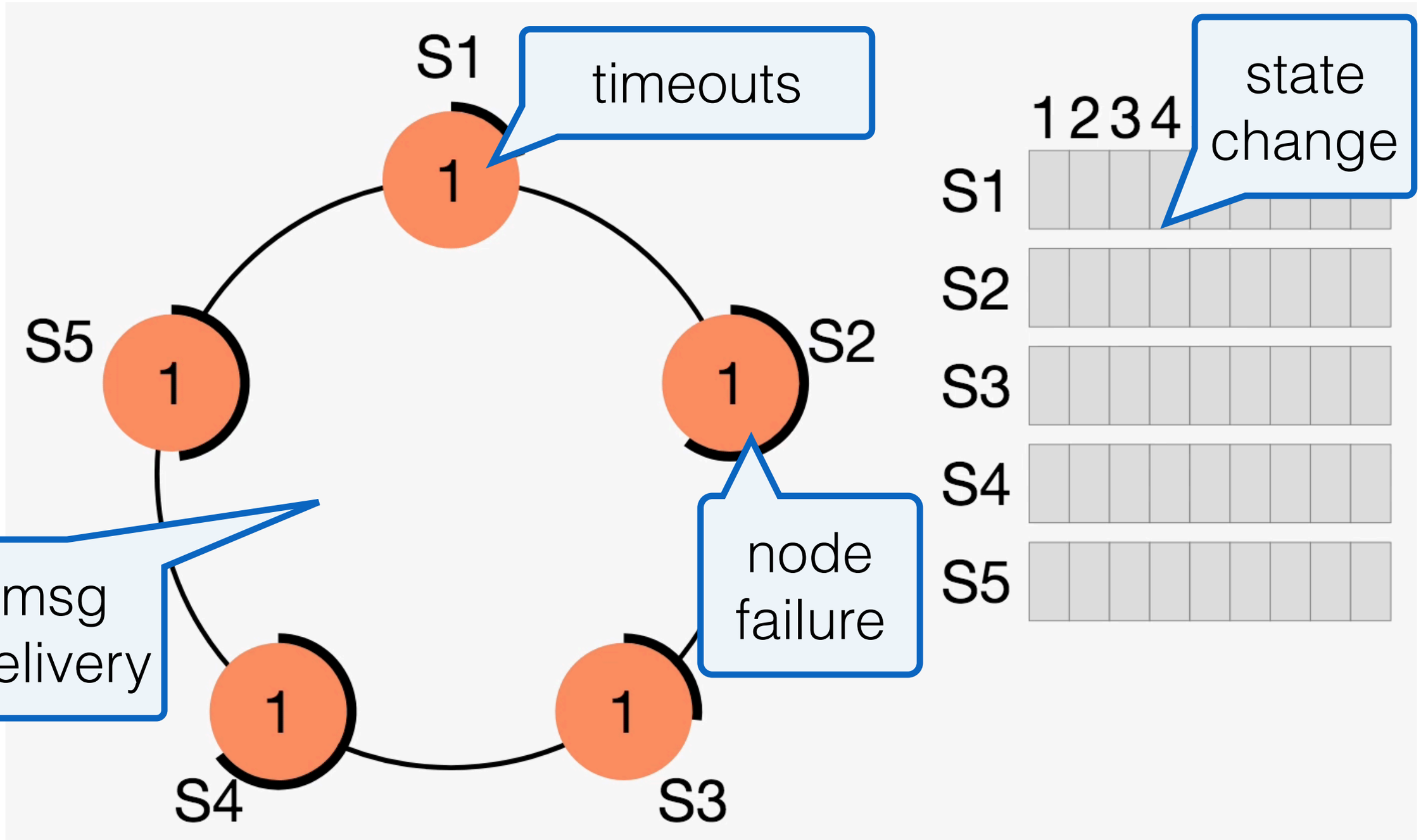
Formalizing distributed systems



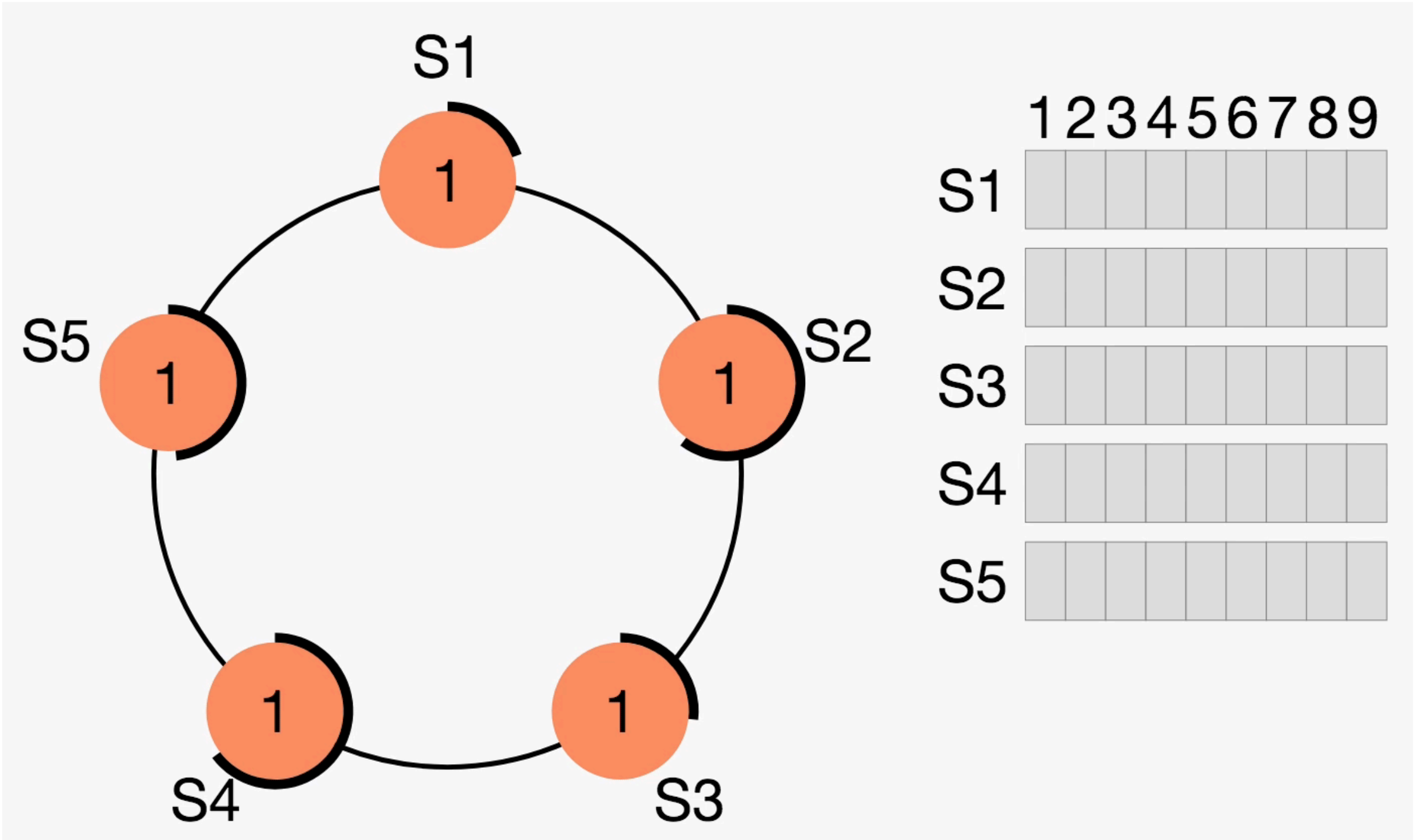
Formalizing distributed systems



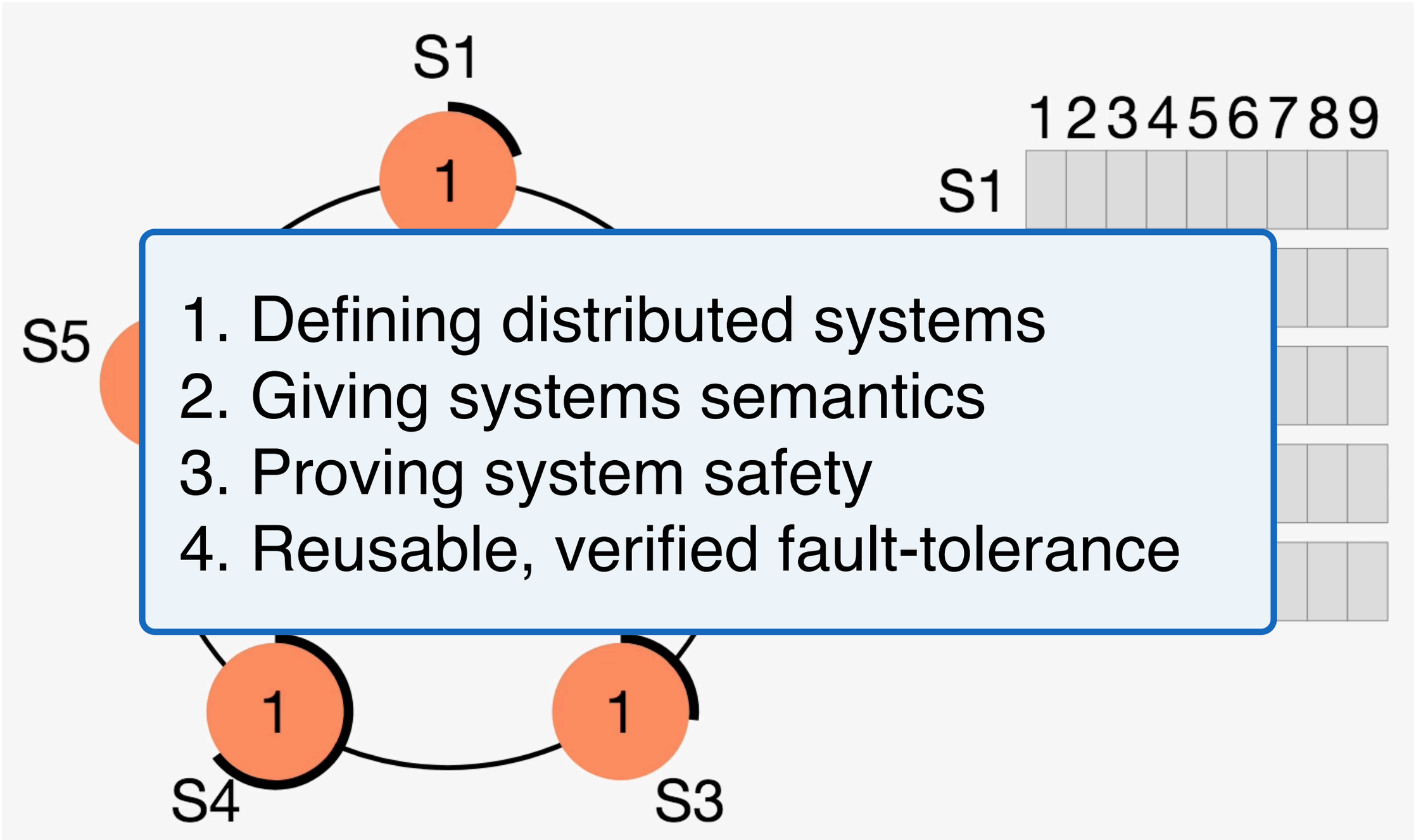
Formalizing distributed systems



Formalizing distributed systems



Formalizing distributed systems



1. Distributed sys as event handlers

```
Def mySys (P : params) : system :=
  // types for state and I/O
  Type in  := (* from external          *)
  Type out := (* to external            *)
  Type msg := (* to/from internal node *)
  Type st  := (* node-local state       *)

  Type handler := /* use monads! */
    st -> st* list msg * list out

  // event handlers
  Def onIn    : in   -> handler
  Def onMsg   : msg  -> handler
  Def onTmOut : unit -> handler
```

2. Network semantics

2. Network semantics

$$(P, \Sigma, T)$$

state of the world

2. Network semantics

$$(P, \Sigma, T)$$

state of the world

packets in flight

2. Network semantics

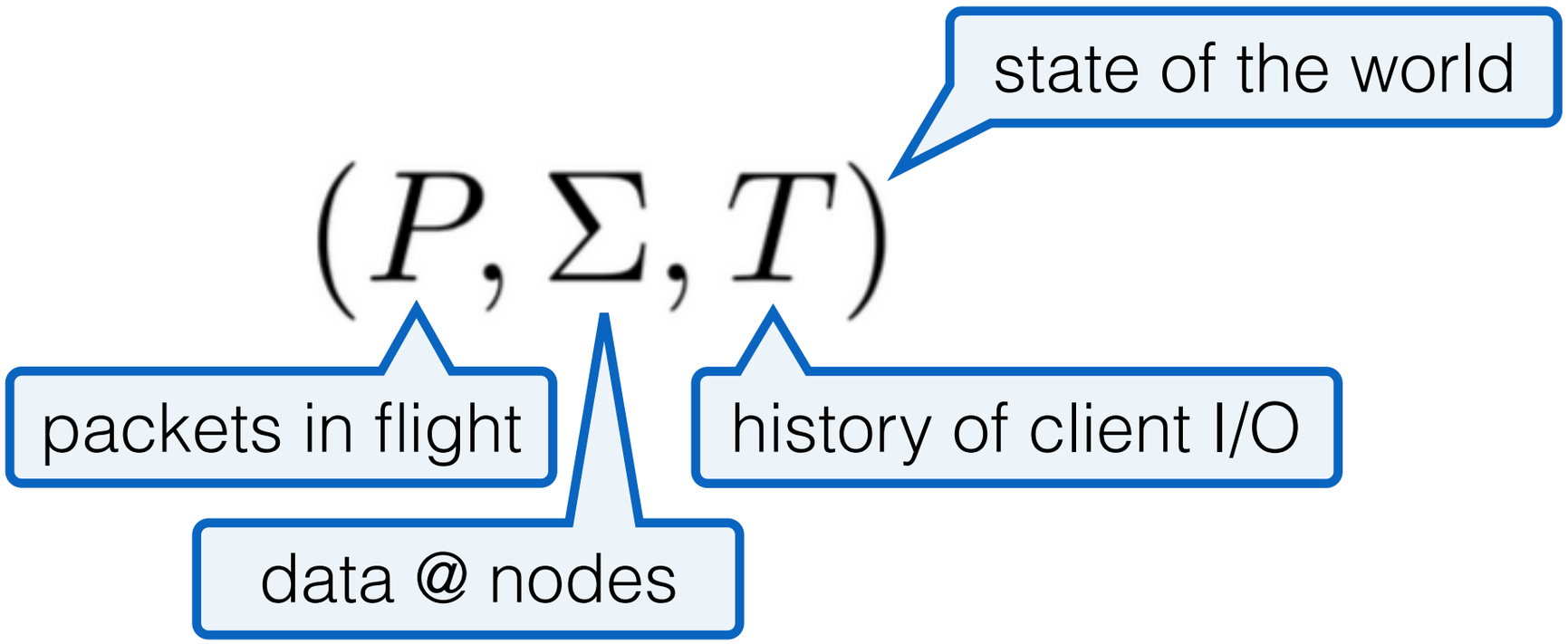
$$(P, \Sigma, T)$$

state of the world

packets in flight

data @ nodes

2. Network semantics



2. Network semantics

$$(P, \Sigma, T)$$

2. Network semantics

$$(P, \Sigma, T) \rightsquigarrow (P', \Sigma', T')$$

Good old small step operational semantics.

Example rule: message delivery

$$\frac{H_{\text{net}}(dst,\ \Sigma[dst],\ src,\ m) = (\sigma',\ o,\ P') \qquad \Sigma' = \Sigma[dst \mapsto \sigma']}{(\{(src,\ dst,\ m)\} \uplus P,\ \Sigma,\ T) \rightsquigarrow (P \uplus P',\ \Sigma',\ T +\!\!+ \langle o \rangle)}$$

Example rule: message delivery

$$\frac{H_{\text{net}}(dst,\ \Sigma[dst],\ src,\ m) = (\sigma',\ o,\ P') \qquad \Sigma' = \Sigma[dst \mapsto \sigma']}{(\{(src,\ dst,\ m)\} \uplus P,\ \Sigma,\ T) \rightsquigarrow (P \uplus P',\ \Sigma',\ T +\!\!+ \langle o \rangle)}$$

if this message is in the network

Example rule: message delivery

run handler on message

$$\frac{H_{\text{net}}(dst,\ \Sigma[dst],\ src,\ m) = (\sigma',\ o,\ P') \qquad \Sigma' = \Sigma[dst \mapsto \sigma']}{(\{(src,\ dst,\ m)\} \uplus P,\ \Sigma,\ T) \rightsquigarrow (P \uplus P',\ \Sigma',\ T +\!\!+ \langle o \rangle)}$$

if this message is in the network

Example rule: message delivery

run handler on message

get response

$$\frac{H_{\text{net}}(dst,\ \Sigma[dst],\ src,\ m) = (\sigma',\ o,\ P') \qquad \Sigma' = \Sigma[dst \mapsto \sigma']}{(\{(src,\ dst,\ m)\} \uplus P,\ \Sigma,\ T) \rightsquigarrow (P \uplus P',\ \Sigma',\ T +\!\!+ \langle o \rangle)}$$

if this message is in the network

Example rule: message delivery

run handler on message

get response

$$\frac{H_{\text{net}}(dst,\ \Sigma[dst],\ src,\ m) = (\sigma',\ o,\ P') \qquad \Sigma' = \Sigma[dst \mapsto \sigma']}{(\{(src,\ dst,\ m)\} \uplus P,\ \Sigma,\ T) \rightsquigarrow (P \uplus P',\ \Sigma',\ T +\!\!+ \langle o \rangle)}$$

if this message is in the network

resulting new global state

2. Network semantics

$$\frac{H_{\text{net}}(dst,\ \Sigma[dst],\ src,\ m) = (\sigma',\ o,\ P') \qquad \Sigma' = \Sigma[dst \mapsto \sigma']}{(\{(src,\ dst,\ m)\} \uplus P,\ \Sigma,\ T) \rightsquigarrow (P \uplus P',\ \Sigma',\ T +\!\!+ \langle o \rangle)}\ \textsc{Deliver}$$

$$\frac{p \in P}{(P,\ \Sigma,\ T) \rightsquigarrow (P \uplus \{p\},\ \Sigma,\ T)}\ \textsc{Duplicate}$$

$$\frac{}{(\{p\} \uplus P,\ \Sigma,\ T) \rightsquigarrow (P,\ \Sigma,\ T)}\ \textsc{Drop}$$

$$\frac{H_{\text{tmt}}(n,\ \Sigma[n]) = (\sigma',\ o,\ P') \qquad \Sigma' =\ \Sigma[n \mapsto \sigma']}{(P,\ \Sigma,\ T) \rightsquigarrow (P \uplus P',\ \Sigma',\ T +\!\!+ \langle \text{tmt},\ o \rangle)}\ \textsc{Timeout}$$

2. Network semantics

$$\frac{H_{\text{net}}(dst,\ \Sigma[dst],\ src,\ m) = (\sigma',\ o,\ P') \qquad \Sigma' = \Sigma[dst \mapsto \sigma']}{(\{(src,\ dst,\ m)\} \uplus P,\ \Sigma,\ T) \rightsquigarrow (P \uplus P',\ \Sigma',\ T +\!\!+ \langle o \rangle)}\ \textsc{Deliver}$$

semantics parameterized by handlers

$$\frac{}{(\{p\} \uplus P,\ \Sigma,\ T) \rightsquigarrow (P,\ \Sigma,\ T)}\ \textsc{Drop}$$

$$\frac{H_{\text{tmt}}(n,\ \Sigma[n]) = (\sigma',\ o,\ P') \qquad \Sigma' = \Sigma[n \mapsto \sigma']}{(P,\ \Sigma,\ T) \rightsquigarrow (P \uplus P',\ \Sigma',\ T +\!\!+ \langle \text{tmt},\ o \rangle)}\ \textsc{Timeout}$$

2. Network semantics

$$\frac{H_{\text{net}}(dst,\ \Sigma[dst],\ src,\ m) = (\sigma',\ o,\ P') \qquad \Sigma' = \Sigma[dst \mapsto \sigma']}{(\{(src,\ dst,\ m)\} \uplus P,\ \Sigma,\ T) \rightsquigarrow (P \uplus P',\ \Sigma',\ T +\!\!+ \langle o \rangle)}\ \textsc{Deliver}$$

$$\frac{p \in P}{(P,\ \Sigma,\ T) \rightsquigarrow (P \uplus \{p\},\ \Sigma,\ T)}\ \textsc{Duplicate}$$

$$\frac{}{(\{p\} \uplus P,\ \Sigma,\ T) \rightsquigarrow (P,\ \Sigma,\ T)}\ \textsc{Drop}$$

$$\frac{H_{\text{tmt}}(n,\ \Sigma[n]) = (\sigma',\ o,\ P') \qquad \Sigma' =\ \Sigma[n \mapsto \sigma']}{(P,\ \Sigma,\ T) \rightsquigarrow (P \uplus P',\ \Sigma',\ T +\!\!+ \langle \text{tmt},\ o \rangle)}\ \textsc{Timeout}$$

Library of network semantics

```
Type sem := state -> state -> Prop

Def sync_sem  := (* in-order delivery *)

Def async_sem := (* + reordering      *)

Def flaky_sem := (* + drops, timeouts *)

Def busy_sem  := (* + duplicates     *)

Def crash_sem := (* + crash, reboot  *)
```

Library of network semantics

```
Type sem := state -> state -> Prop

Def sync_sem  := (* in-order delivery *)

Def async_sem := (* + reordering      *)

Def flaky_sem := (* + drops, timeouts *)

Def busy_sem  := (* + duplicates     *)

Def crash_sem := (* + crash, reboot  *)
```

Precisely characterize fault model for sys.

Library of network semantics

```
Type sem := state -> state -> Prop

Def sync_sem  := (* in-order delivery *)

Def async_sem := (* + reordering      *)

Def flaky_sem := (* + drops, timeouts *)

Def busy_sem  := (* + duplicates     *)

Def crash_sem := (* + crash, reboot  *)
```

Precisely characterize fault model for sys.

more behaviors --> harder proof

3. Verifying system safety

```
Def ok : state -> Prop
```

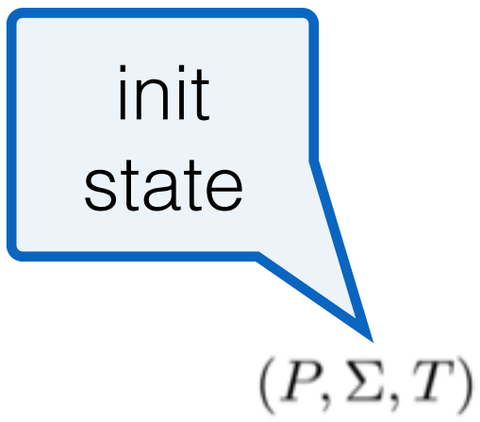
3. Verifying system safety

```
Def ok : state -> Prop
```

(P, Σ, T)

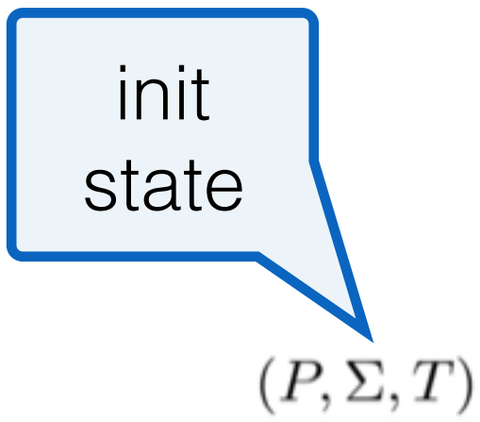
3. Verifying system safety

```
Def ok : state -> Prop
```



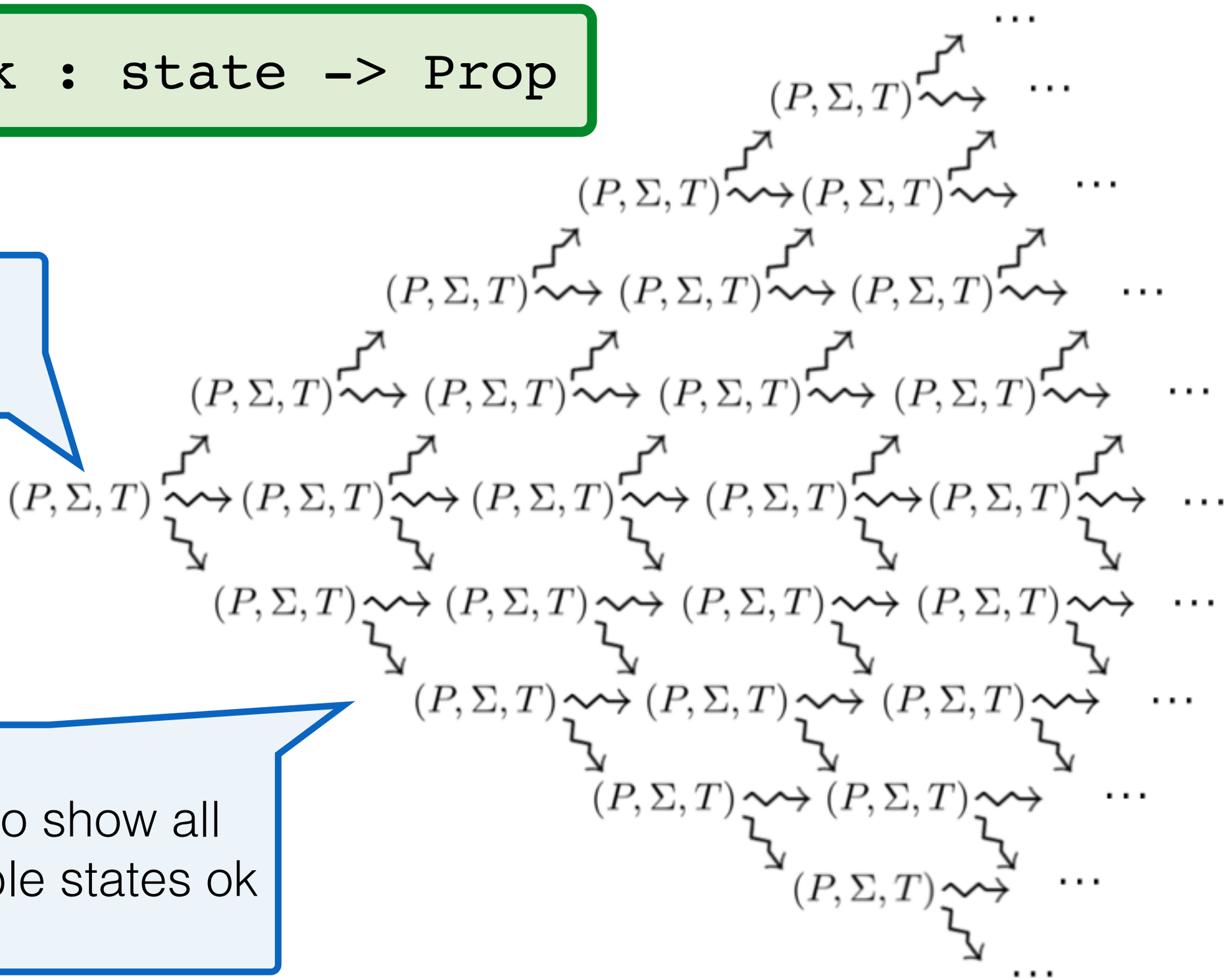
3. Verifying system safety

```
Def ok : state -> Prop
```



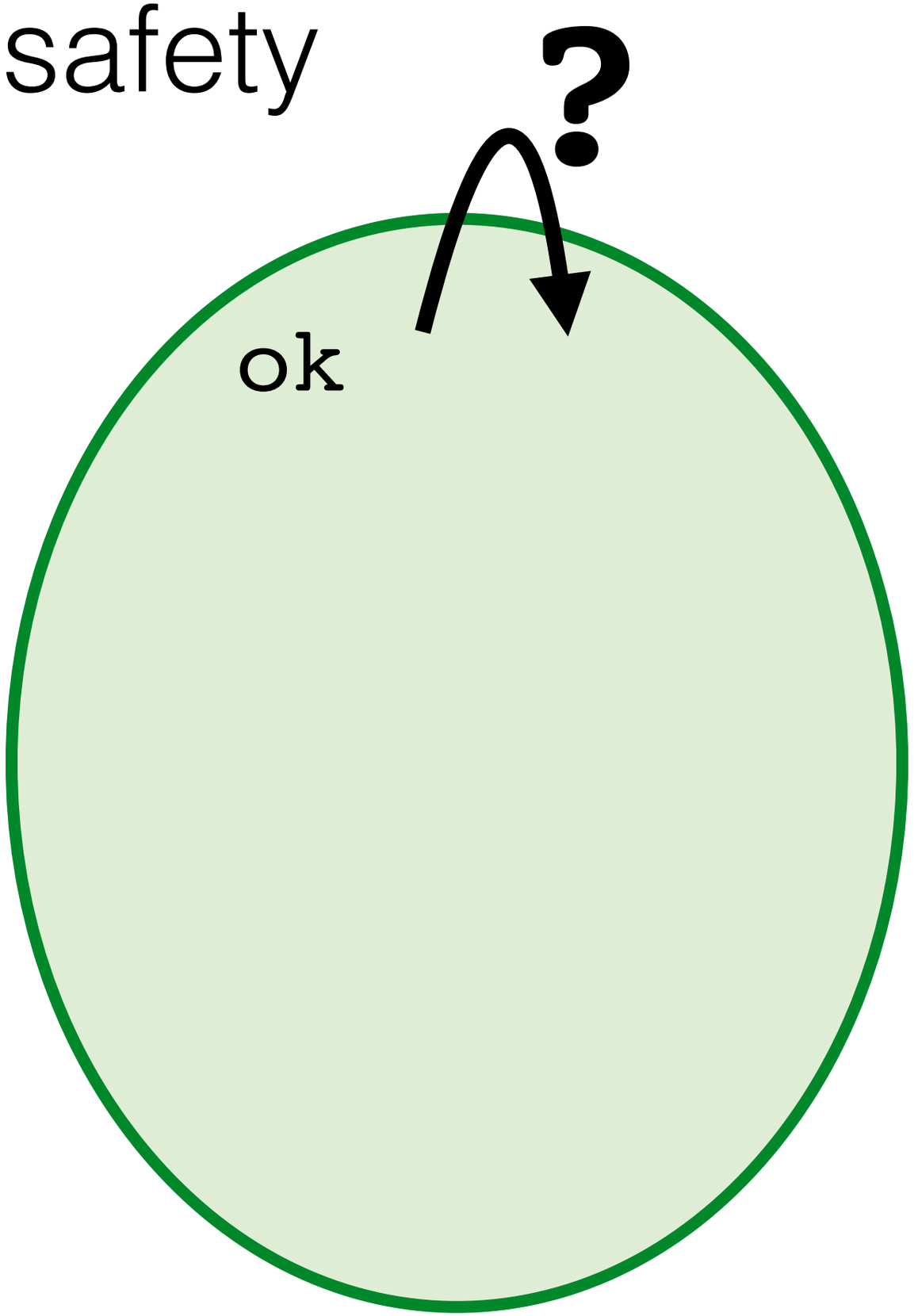
3. Verifying system safety

```
Def ok : state -> Prop
```



3. Verifying system safety

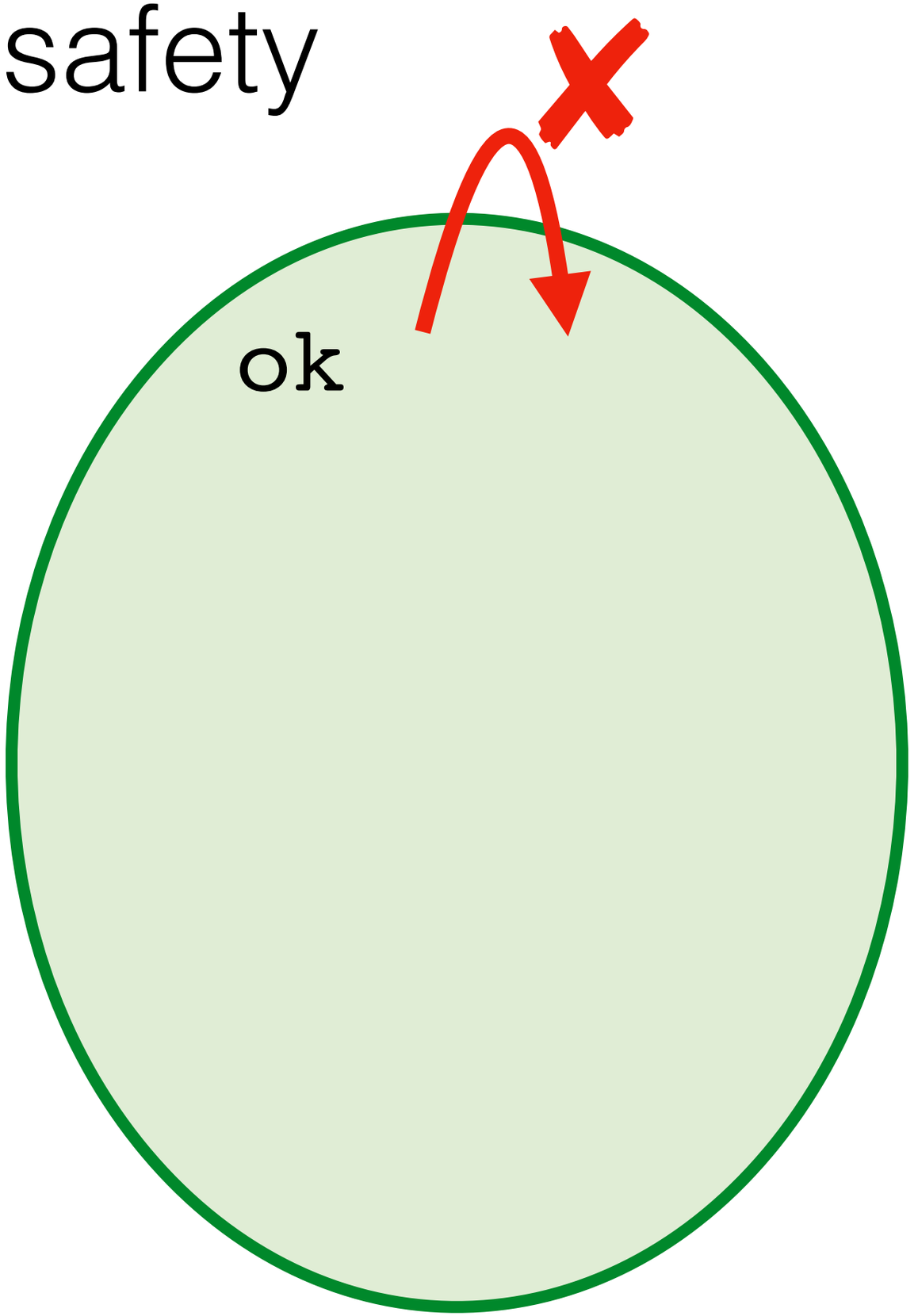
```
Def ok : state -> Prop
```



3. Verifying system safety

```
Def ok : state -> Prop
```

As usual, problem is specs not inductive.

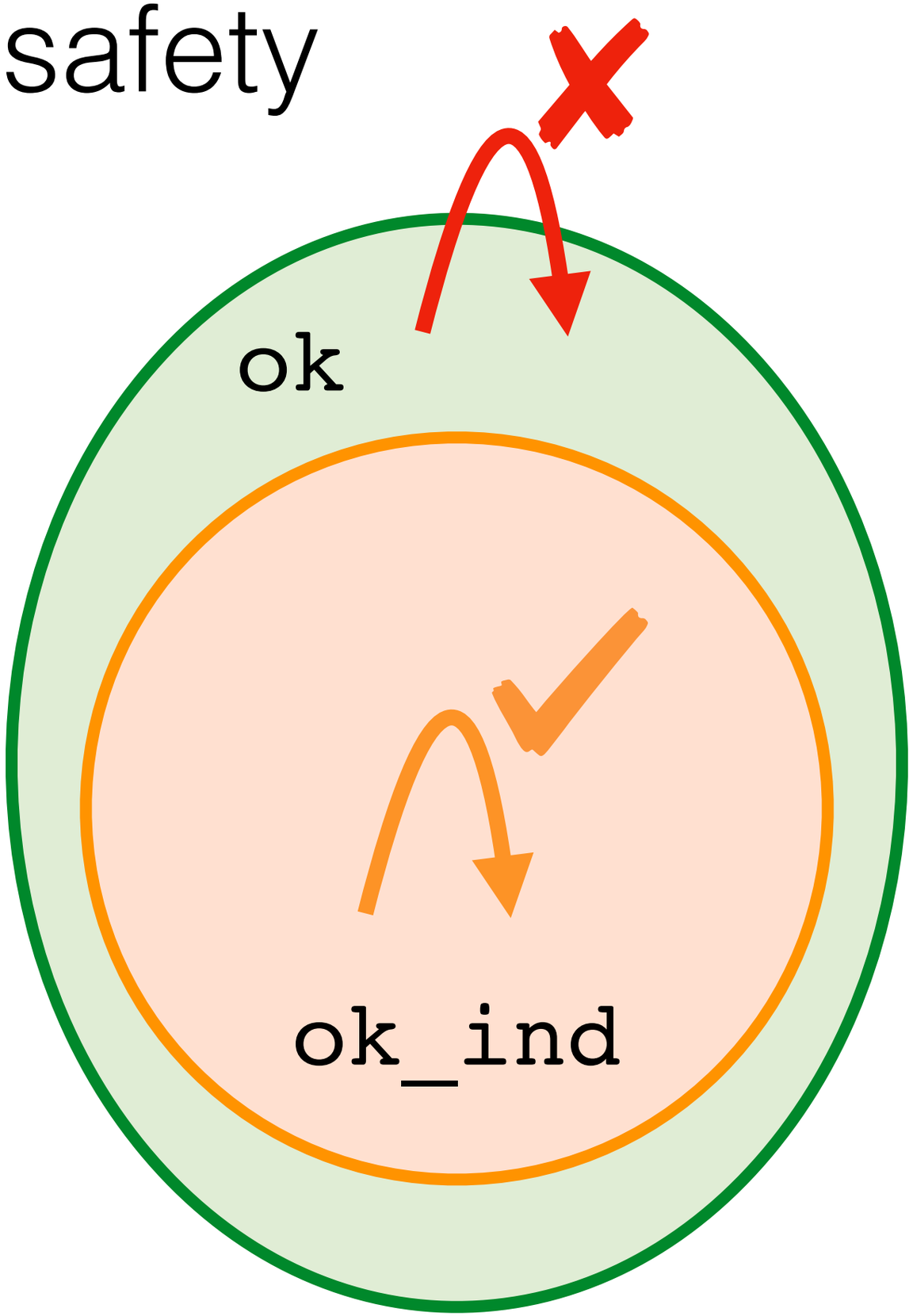


3. Verifying system safety

```
Def ok : state -> Prop
```

As usual, problem is specs not inductive.

Strengthen “ok” to inductive “ok_ind”.



3. Verifying system safety

```
Def ok : state -> Prop
```

When verifying systems in a particular semantics, need to repeat similar fault tolerance reasoning for every system.

4. Verifying system *transformers*

Implement fault tolerance as wrapper

```
Def tcp : system -> system
```

Transfer proofs across semantics

```
Theorem tcp_ok : forall s P,
  P s -> lift_tcp P (tcp s)
```

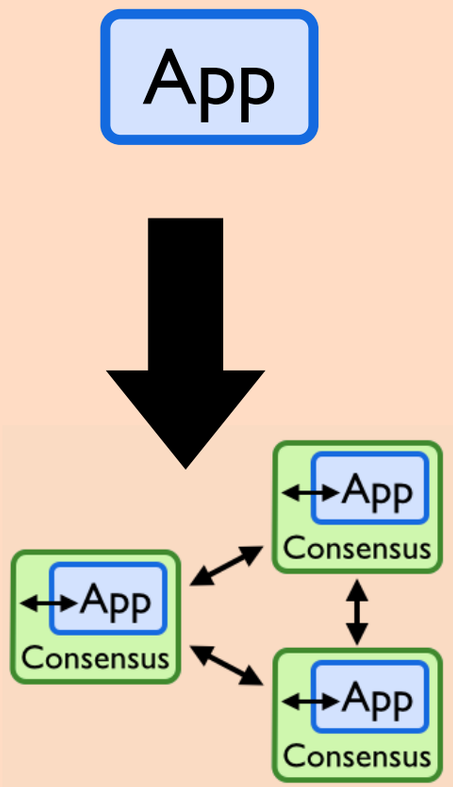
Separate app proof / fault tolerance

handles class of faults once and for all

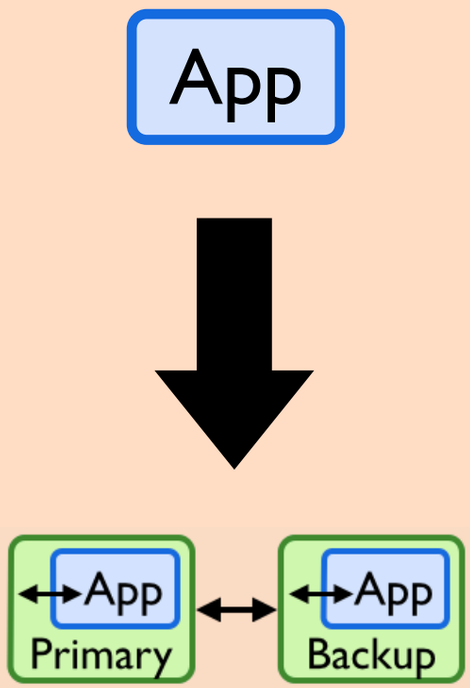
can compose transformers, proofs

4. Verifying system *transformers*

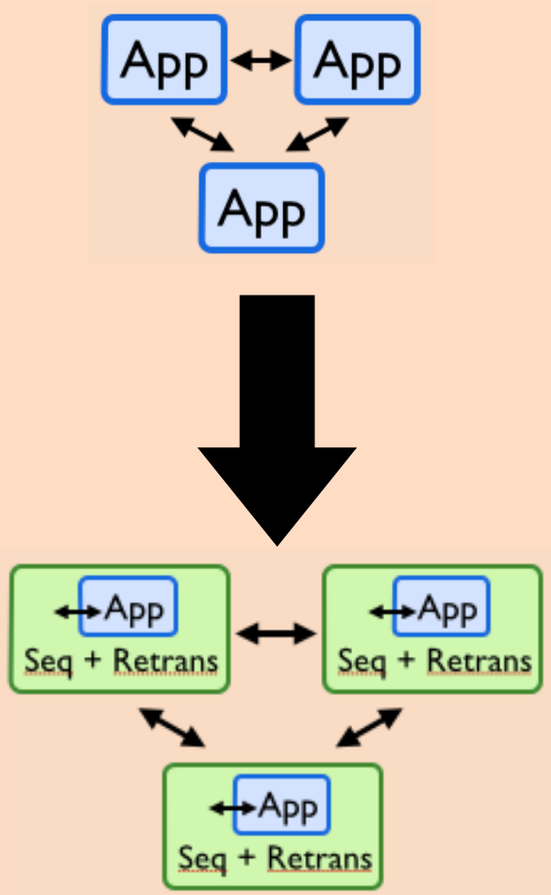
Raft Consensus



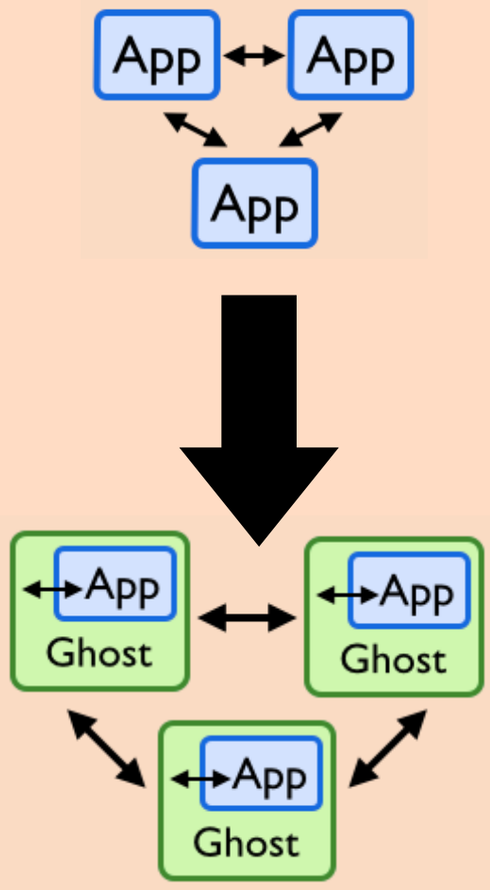
Primary Backup



Seq # and Retrans



Ghost Variables



Verdi Team

James Wilcox

Doug Woos

Pavel Panchekha

Ryan Doenges

Justin Adsuara

Keith Simmons

Steve Anton

Miranda Edwards

Karl Palmskog

Ilya Sergey

Xi Wang

Mike Ernst

Tom Anderson