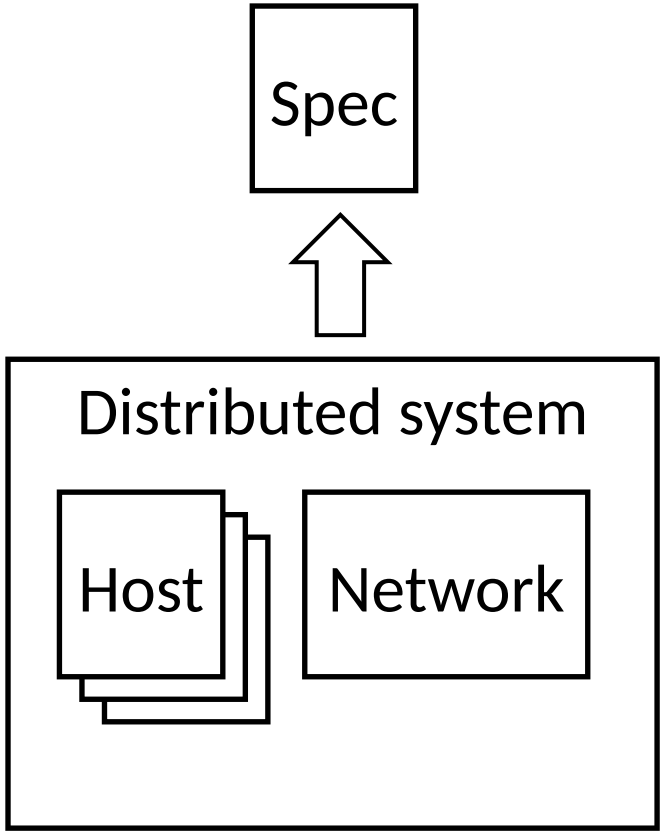
EECS498-003
Formal Verification of Systems Software

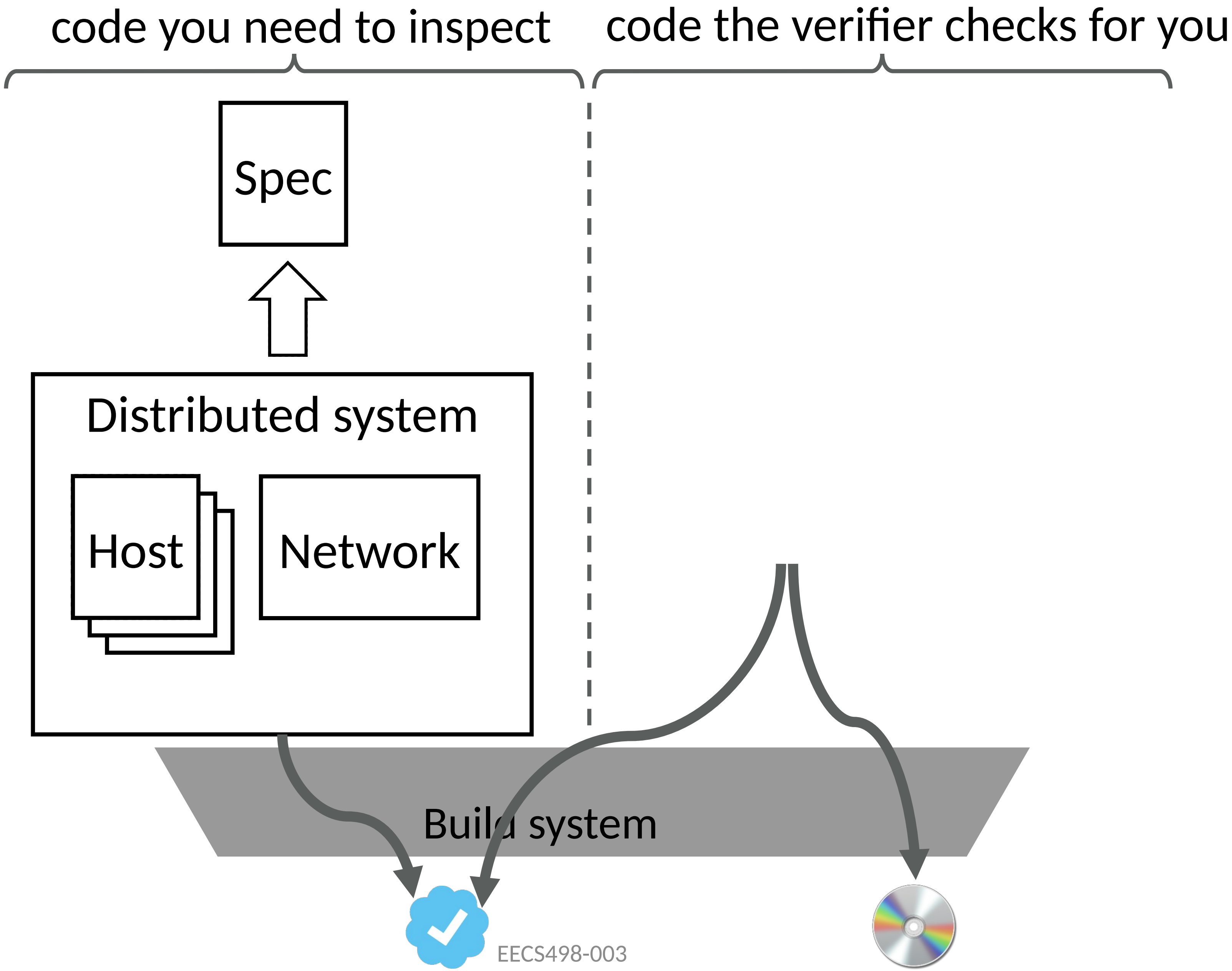
Material and slides created by

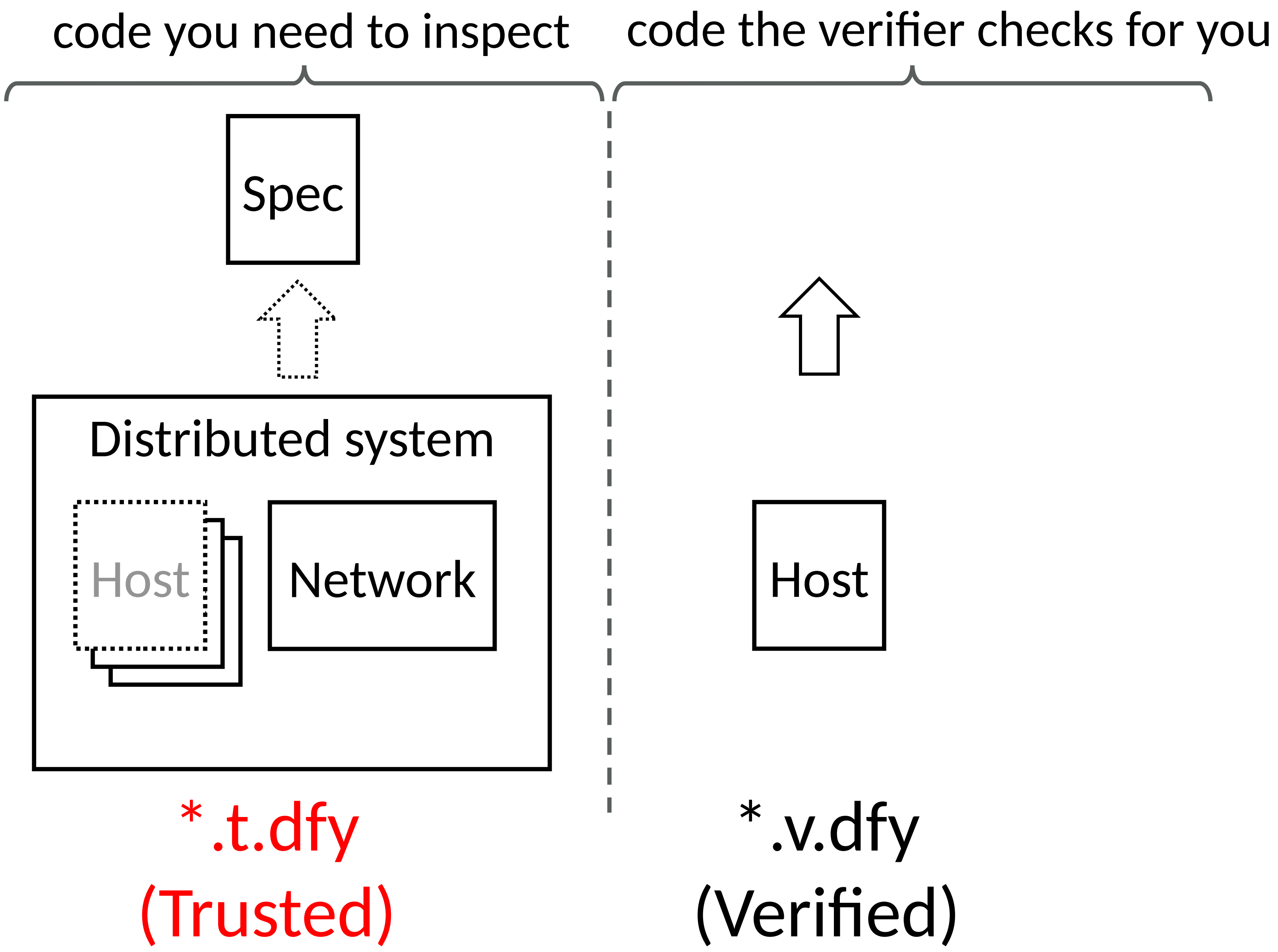
Jon Howell and Manos Kapritsos

Refinement recap




```
ghost function Abstraction(v:Variables) : Spec.Variables
predicate Inv(v:Variables)

lemma RefinementInit(v:Variables)
    requires Init(v)
    ensures Inv(v) // Inv base case
    ensures Spec.Init(Abstraction(v))  // Refinement base case

lemma RefinementNext(v:Variables, v':Variables)
    requires Next(v, v’, evt)
    requires Inv(v)
    ensures Inv(v')  // Inv inductive step
    ensures Spec.Next(Abstraction(v), Abstraction(v’), evt)
          || Abstraction(v) == Abstraction(v’) && evt == NoOp
```


code you need to inspect
code the verifier checks for you
Spec
Distributed system
Host
Network
Build system


code you need to inspect
code the verifier checks for you
Spec
Distributed system
Host
Network
Host
*.t.dfy
(Trusted)
*.v.dfy
(Verified)

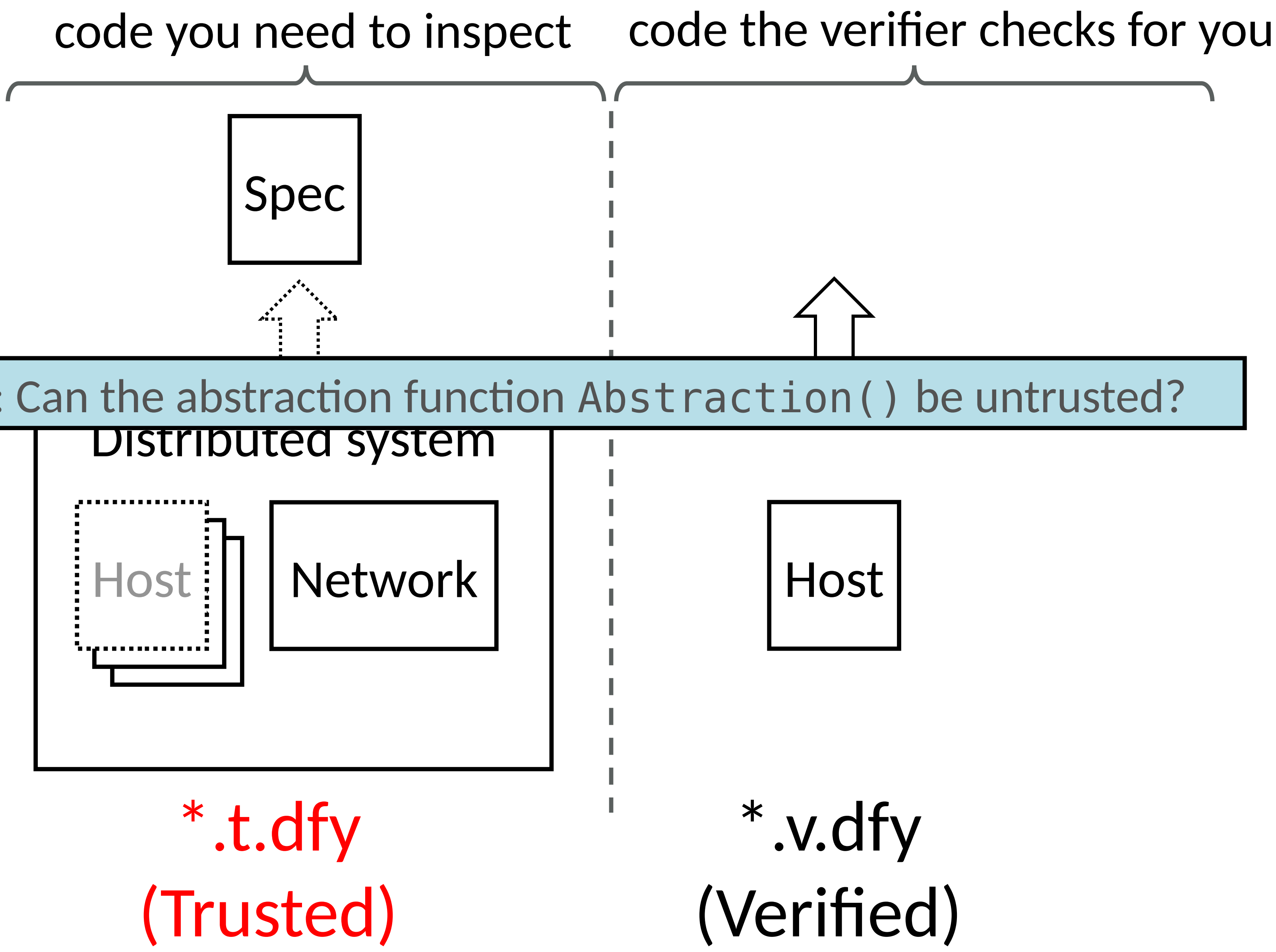
The verification game

- Player 1: the benign verification expert 
- Player 2: the malicious engineer 

Player 1 sets up the trusted environment
(i.e. all `.t.dfy` files)

Player 2 writes the implementation and proof
(i.e. all `.v.dfy` files)

Player 1 runs the build system



Q: Can the abstraction function `Abstraction()` be untrusted?

What if the abstraction function pretended nothing ever happened?

Always returns the initial state

```
function Abstraction(v:Variables) :
                Spec.Variables {
  var a0 :| SpecInit(a0);
  a0
}

predicate Inv(v:Variables) { true }
```

...or just made up a fake story?

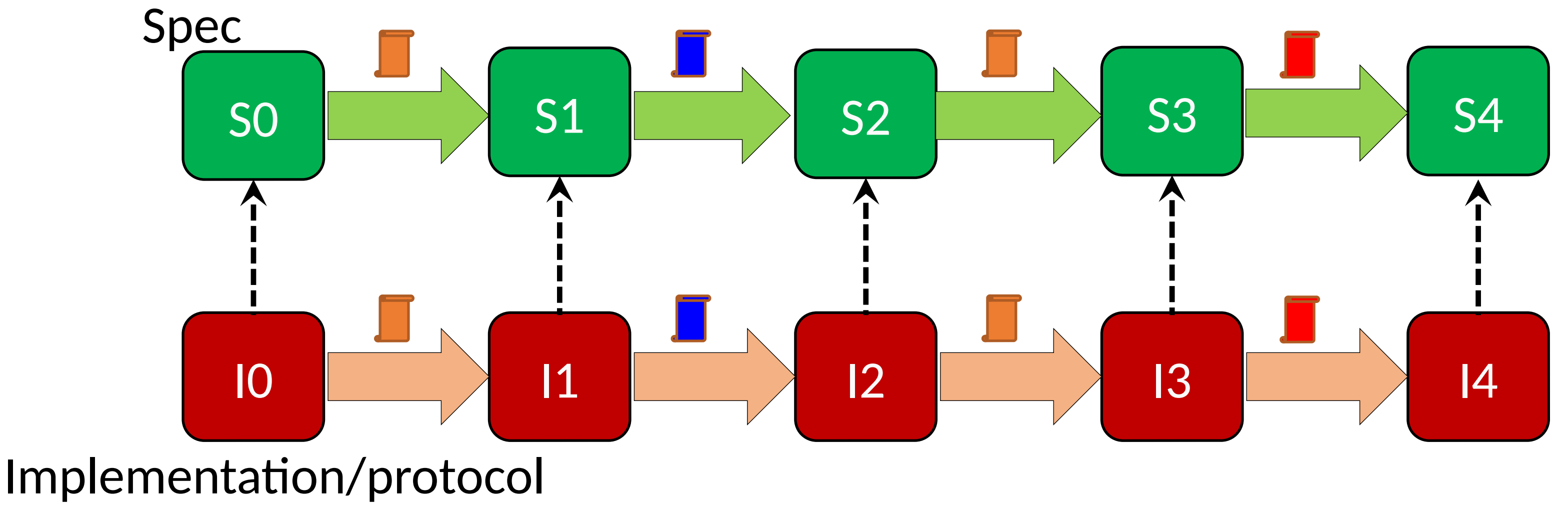
```
datatype Variables =
Variables(actualState: Stuff, fakeState:
HostState)

function Abstraction(v:Variables) :
Spec.Variables {
      v.fakeState
}
```

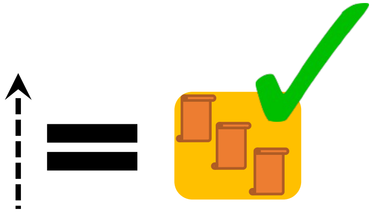
Returns fake state

Events to the rescue

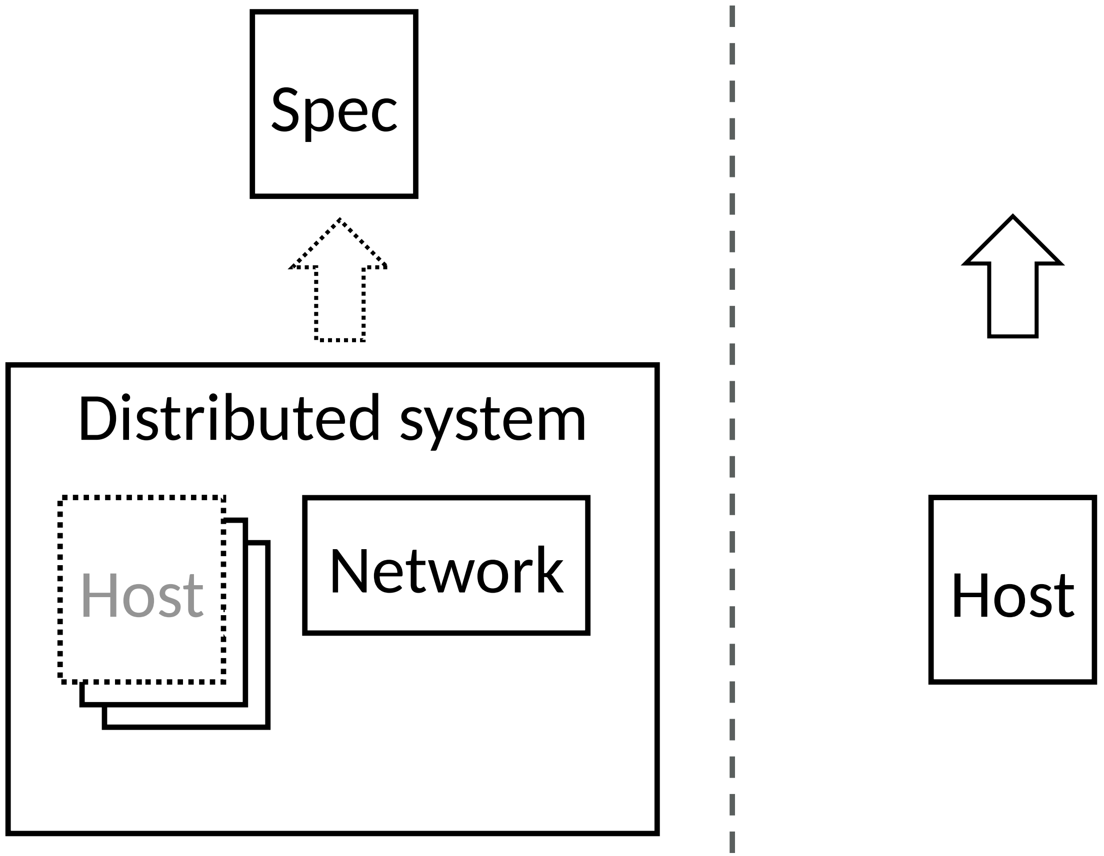
```
ghost function Abstraction(v:Variables) : Spec.Variables
predicate Inv(v:Variables)

lemma RefinementInit(v:Variables)
    requires Init(v)
    ensures Inv(v) // Inv base case
    ensures Spec.Init(Abstraction(v))  // Refinement base case

lemma RefinementNext(v:Variables, v':Variables)
    requires Next(v, v’, evt)
    requires Inv(v)
    ensures Inv(v')  // Inv inductive step
    ensures Spec.Next(Abstraction(v), Abstraction(v’), evt) // Refinement
inductive step
        || Abstraction(v) == Abstraction(v’) && evt == NoOp // OR stutter step
```

Application correspondence





The Abstraction function is untrusted



*.t.dfy
(Trusted)

*.v.dfy
(Untrusted)

The Abstraction function *must* be untrusted

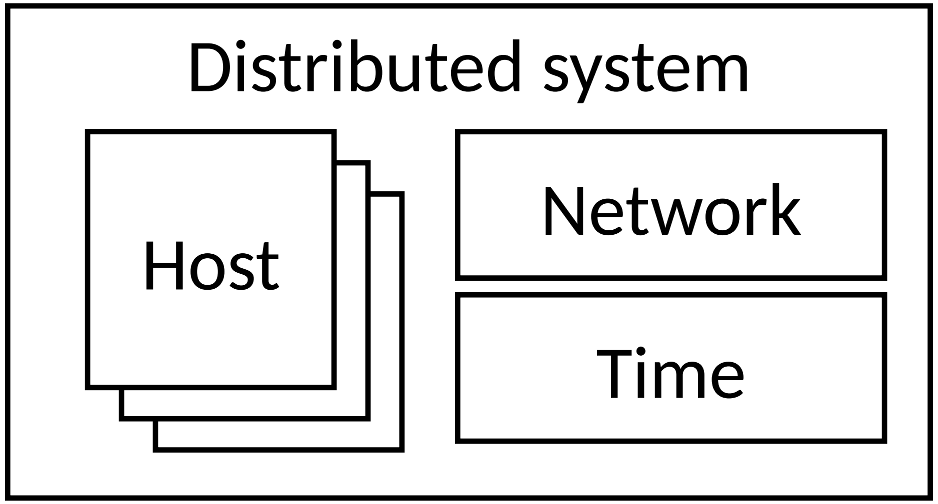
- If it were trusted, we would have to inspect it
- To fully understand it, we would also have to inspect the entire low-level state
- The entire edifice of verification would collapse!

Administrivia

- Project 1 due today
- PS4 released tomorrow
- No class next Tuesday 11/12
  - Manos out of town

Revisiting the distributed system model

- Composite state machine
  - Hosts
  - Network
  - Time



In each step of this state machine:

- at most one Host takes a step, together with the Network
- or Time advances

Are the steps *really* atomic?

Model: S0 → S1 → S2 → S3 → S4

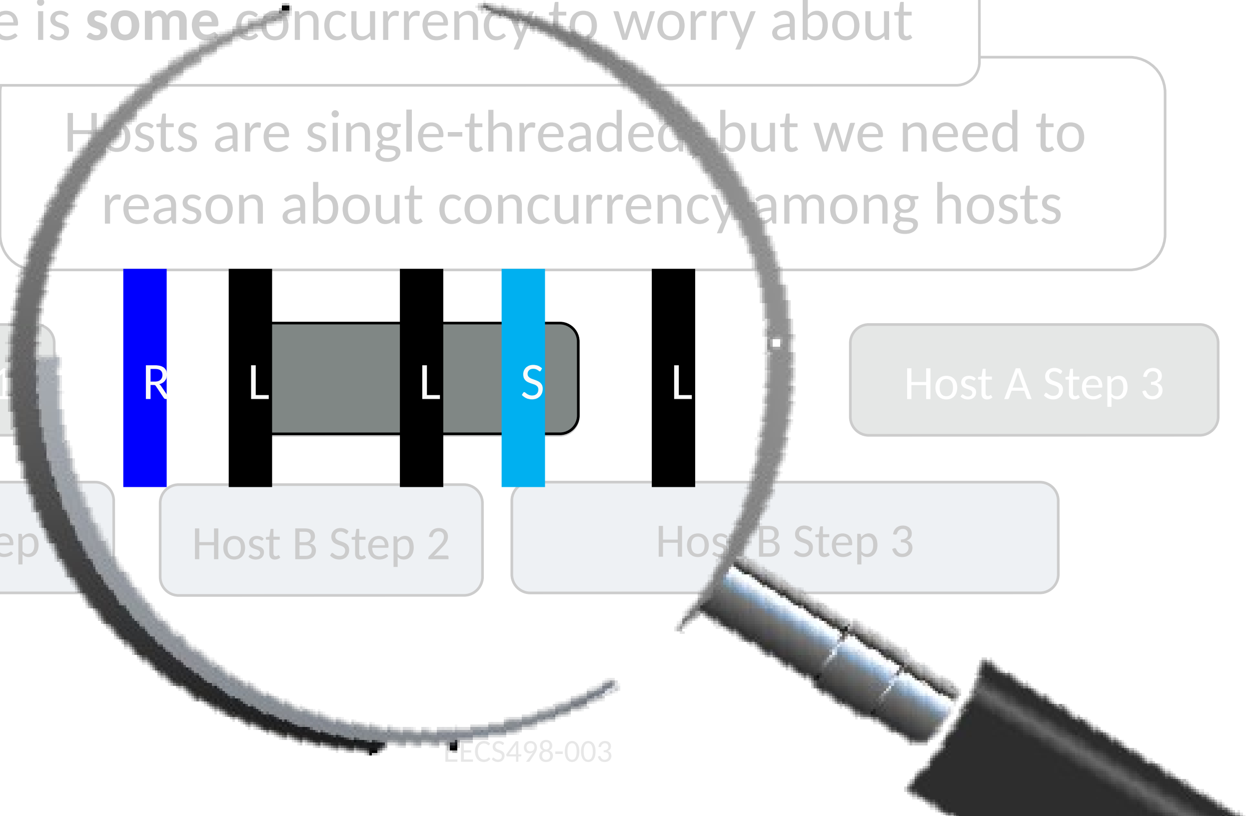
There is **some** concurrency to worry about

Hosts are single-threaded, but we need to reason about concurrency among hosts

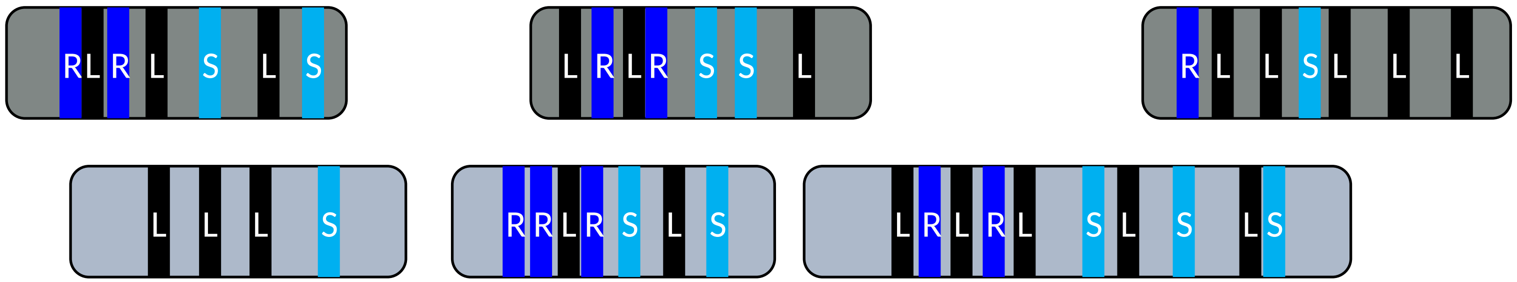
Reality:

Host A Step 1 | Host A Step 2 | Host A Step 3

Host B Step 1 | Host B Step 2 | Host B Step 3



A distributed execution in real life

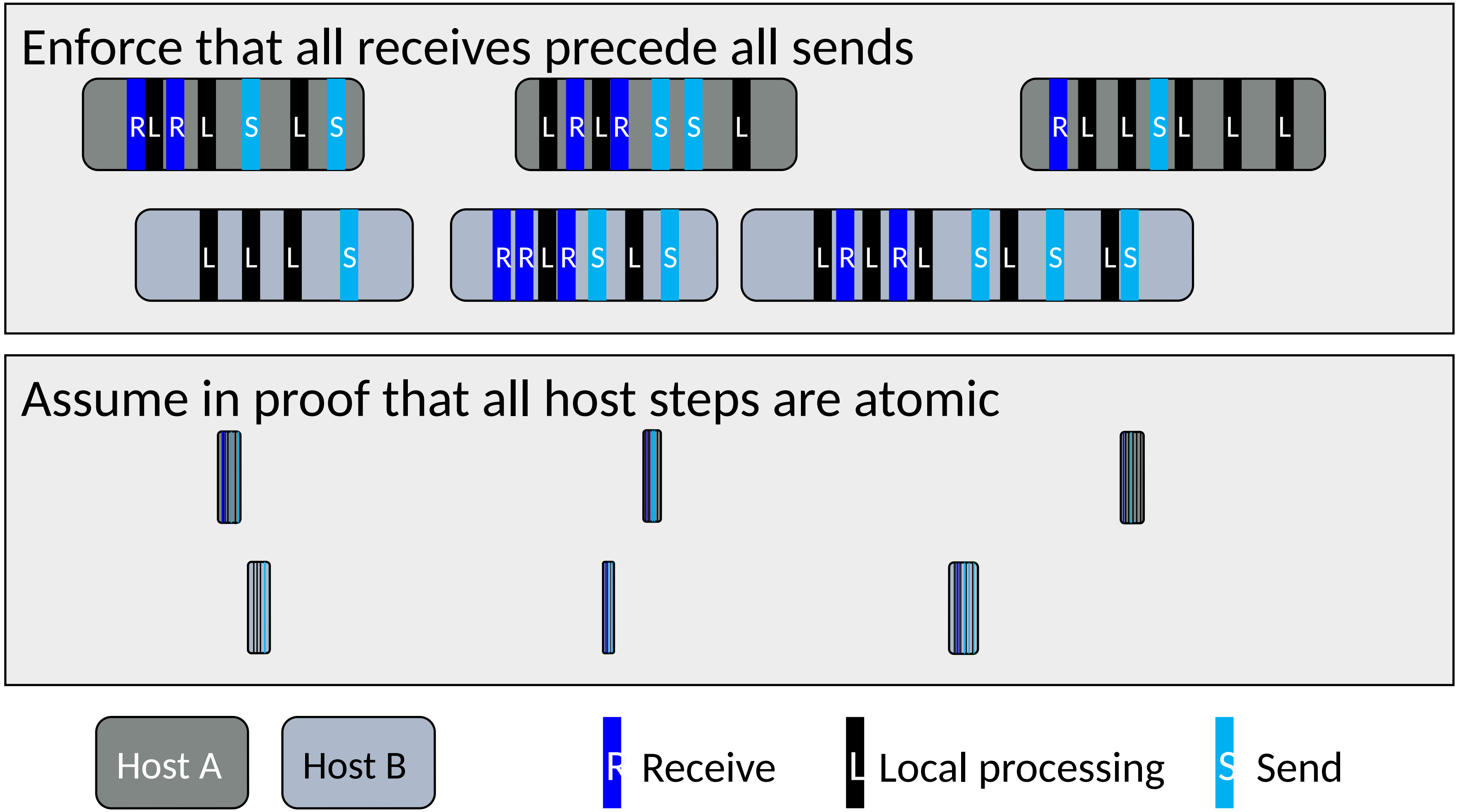


Reason about all possible interleavings of the substeps?

Host A Host B

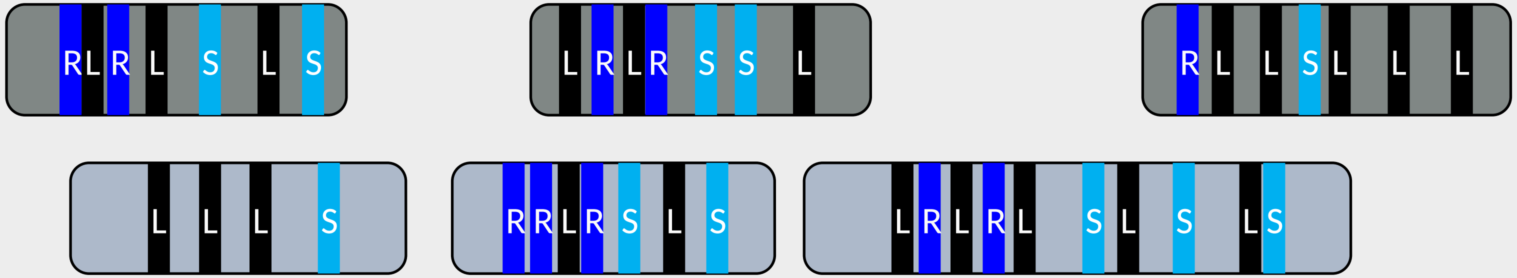
R Receive L Local processing S Send

Concurrency containment



Concurrency containment

Reduction argument: for every real trace…



Host A

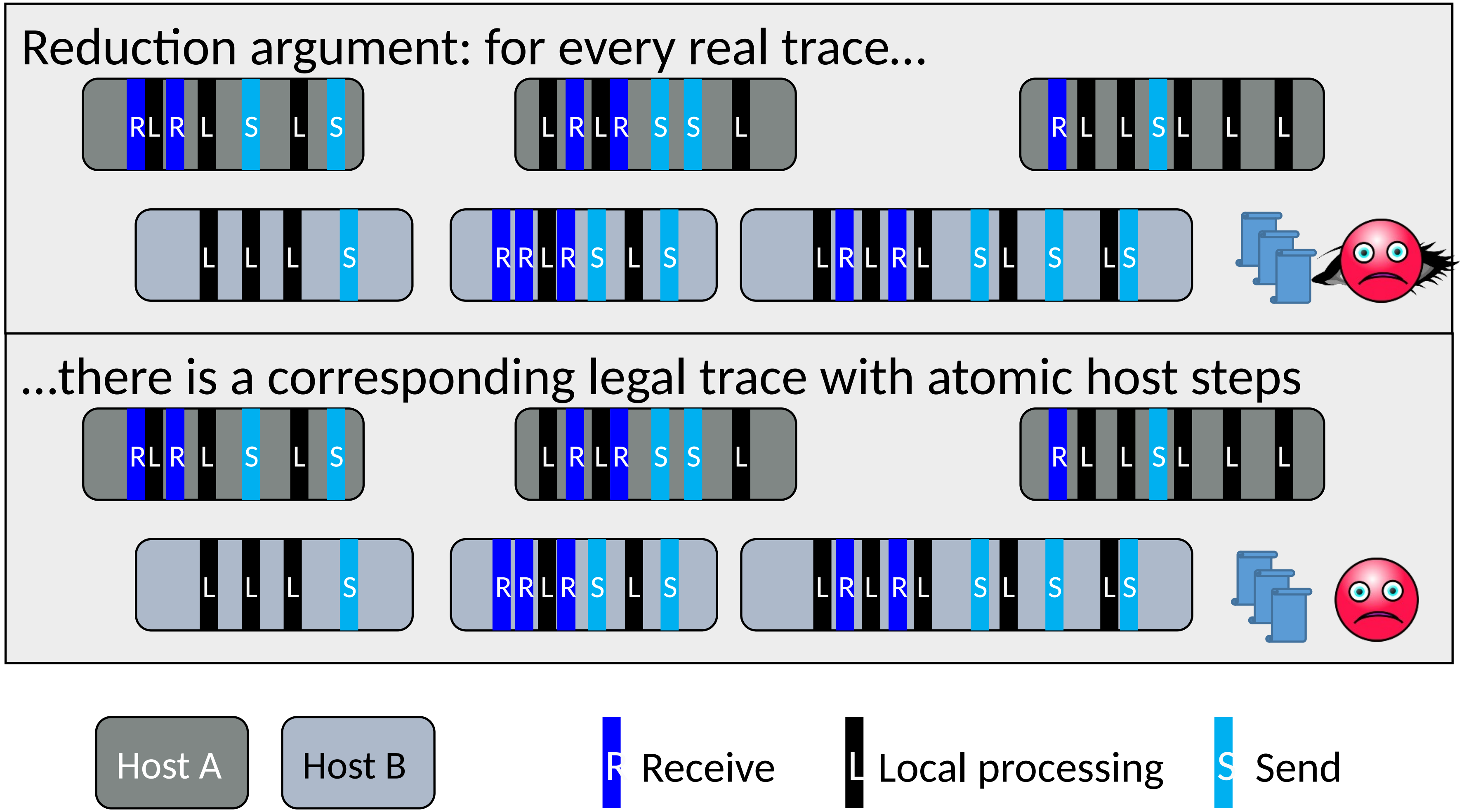
Host B

R Receive

L Local processing

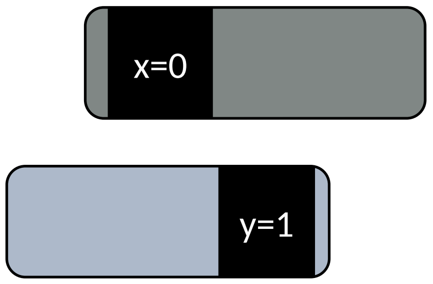
S Send

Concurrency containment

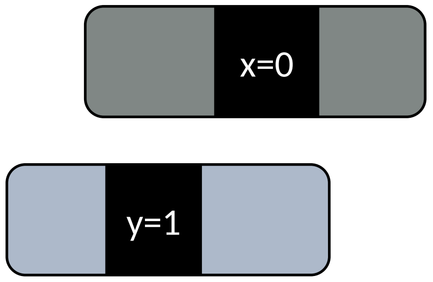


The concept of “movers”

Actual execution



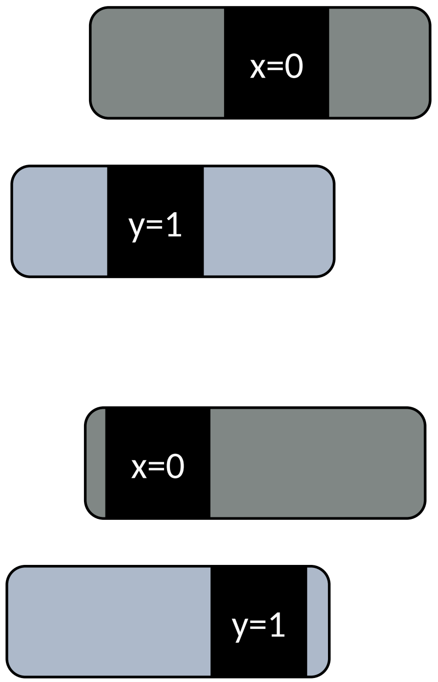
Indistinguishable execution



Local computations can move either way

Actual execution

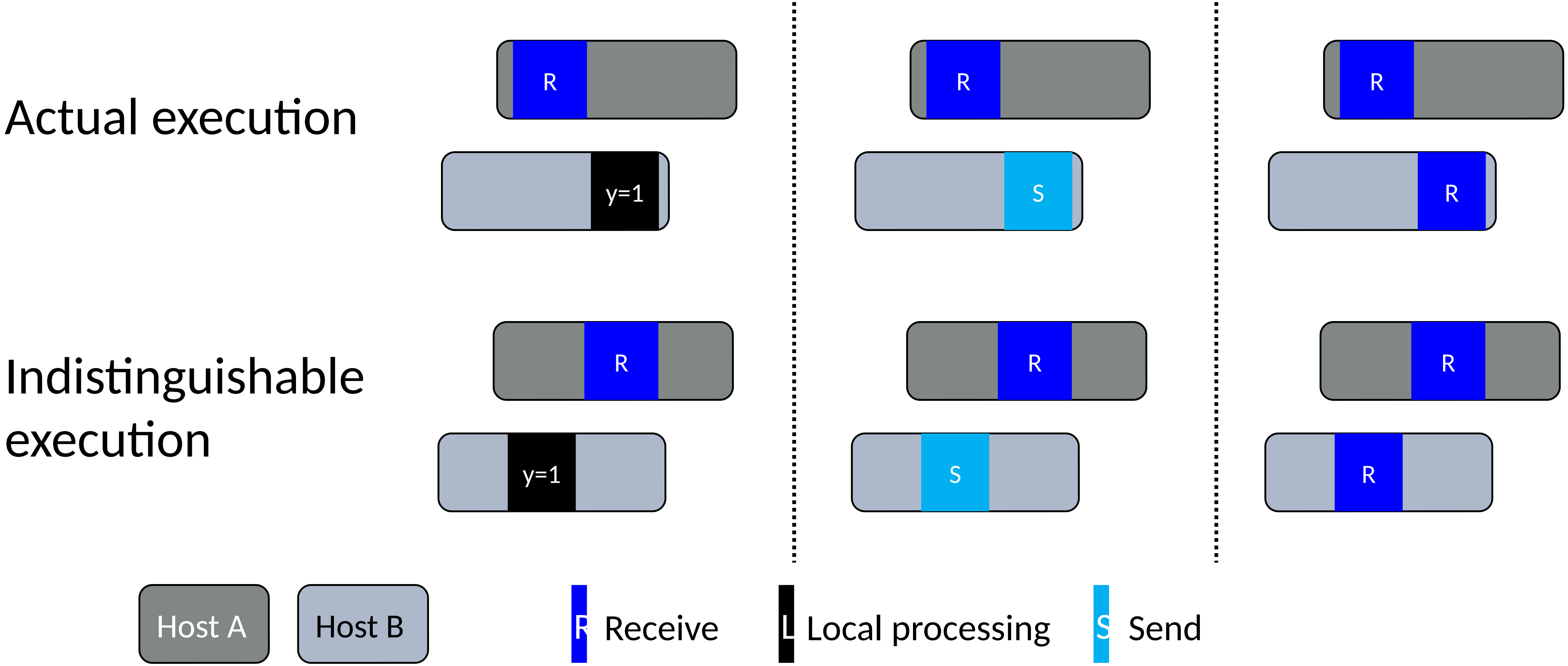
Indistinguishable execution



Host A Host B

R Receive L Local processing S Send

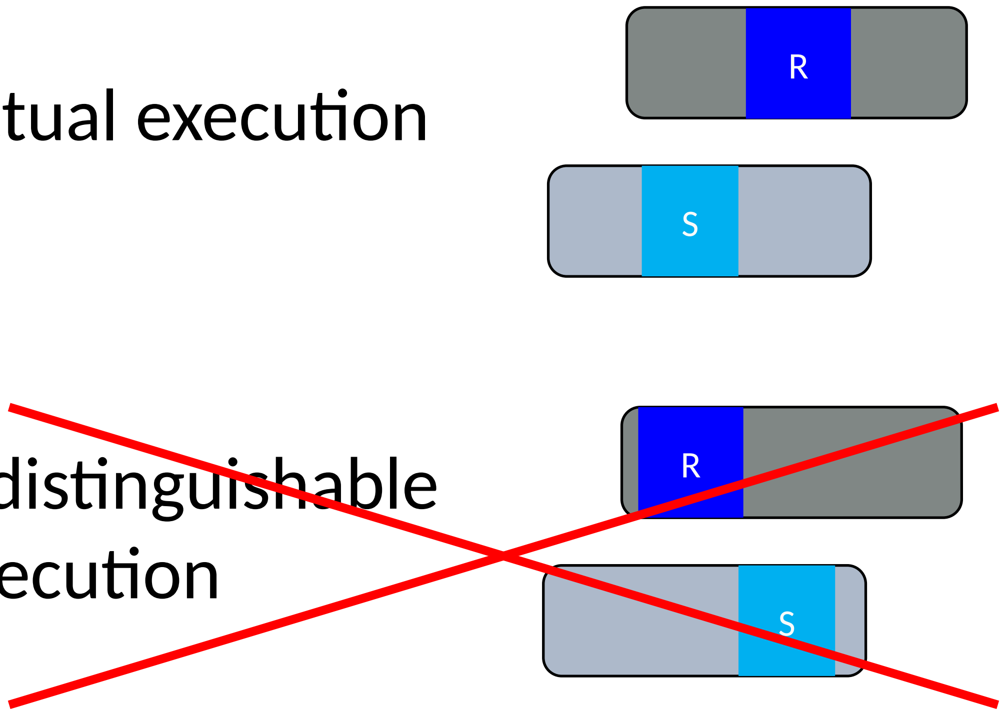
Receives are right movers



Receives are not left movers

Actual execution

Indistinguishable execution



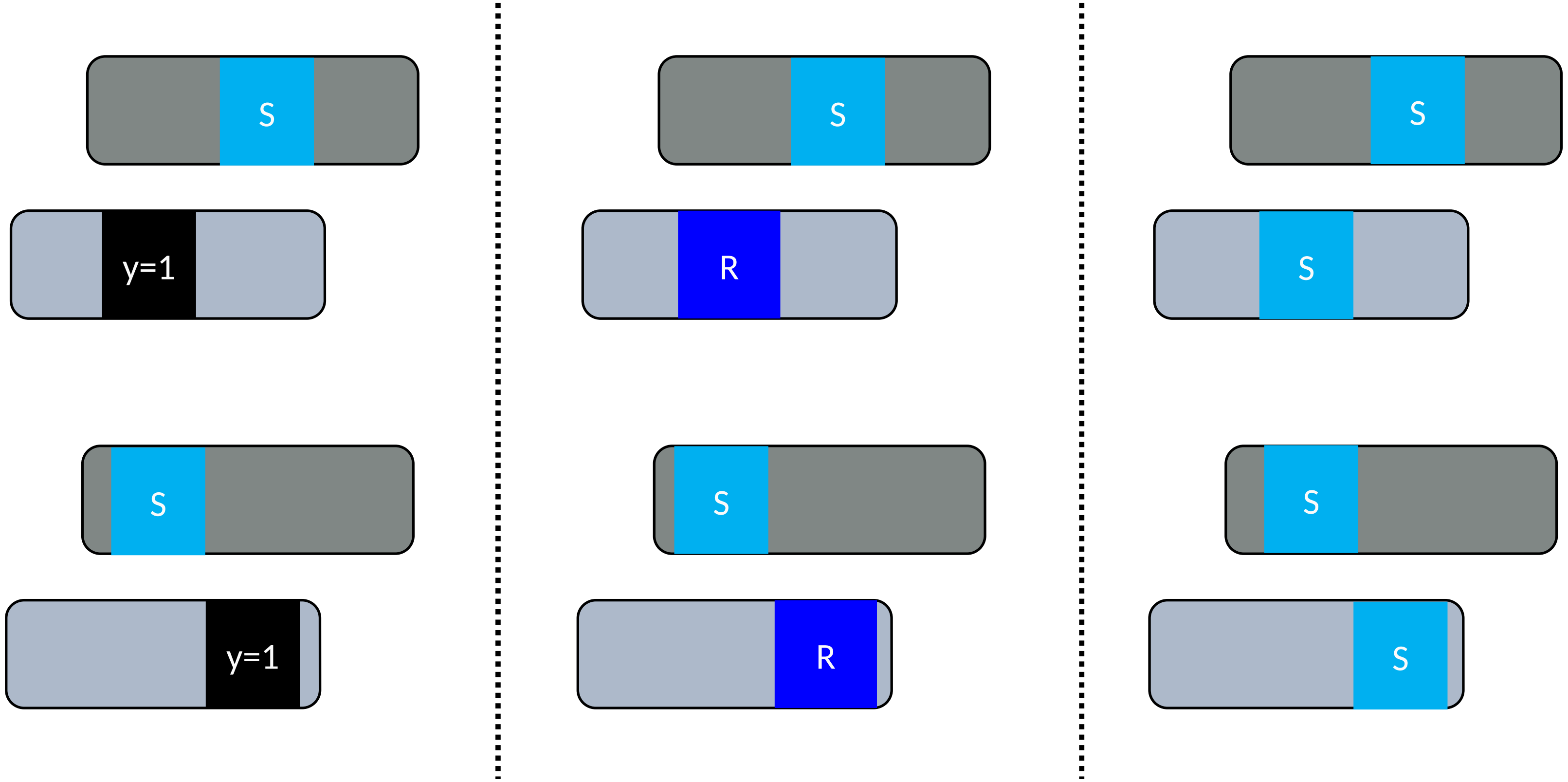
Host A
Host B

R Receive
L Local processing
S Send

Sends are left movers

Actual execution

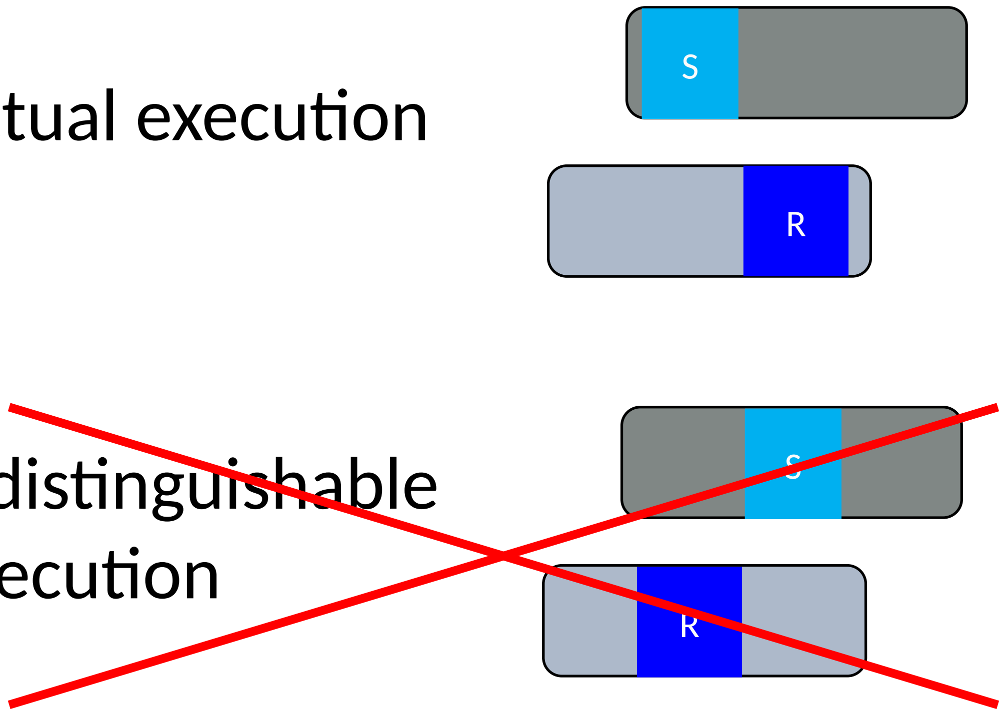
Indistinguishable execution



Sends are not right movers

Actual execution

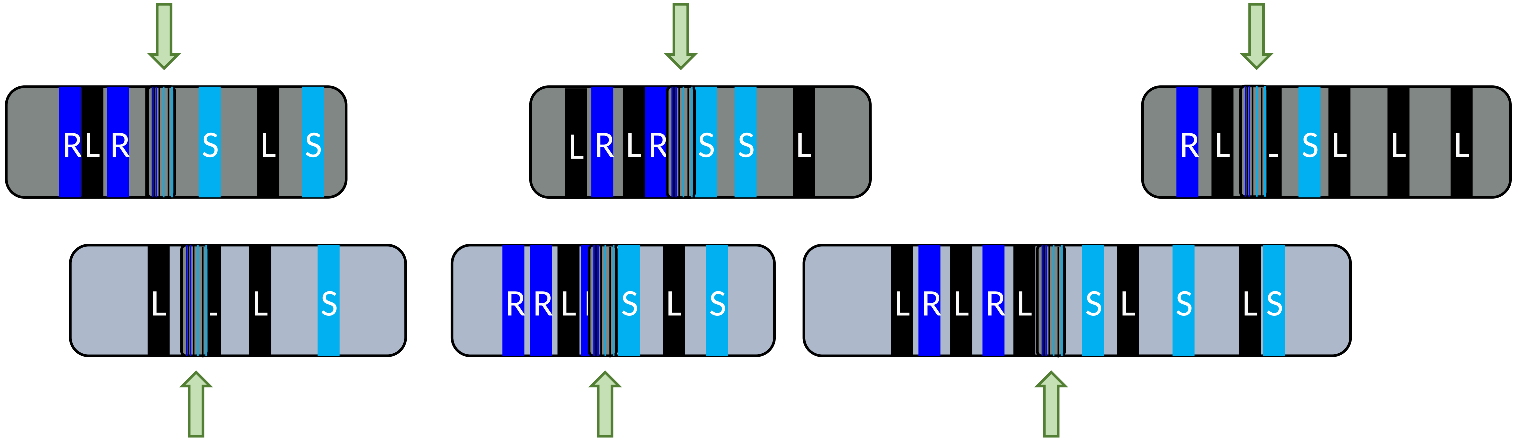
Indistinguishable execution



Summary of movers

- Local computation moves both ways
- Sends move to the left
- Receives move to the right

Creating the atomic trace



Host A

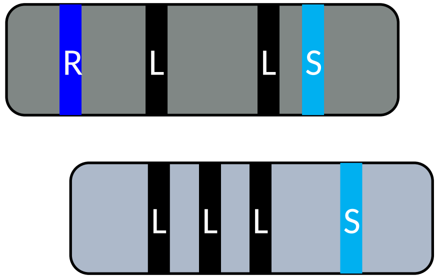
Host B

R Receive

L Local processing

S Send

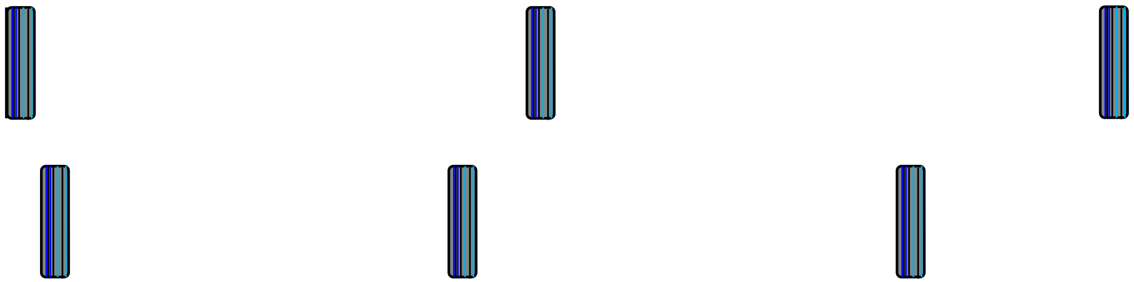
Creating the atomic trace



We can keep moving individual instructions to the left/right, until the entire action is atomic (i.e. does not interleave with other actions)

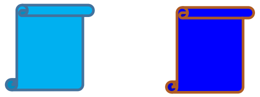
Host A   Host B   R Receive   L Local processing   S Send

The atomic trace is legal



Local event ordering is preserved

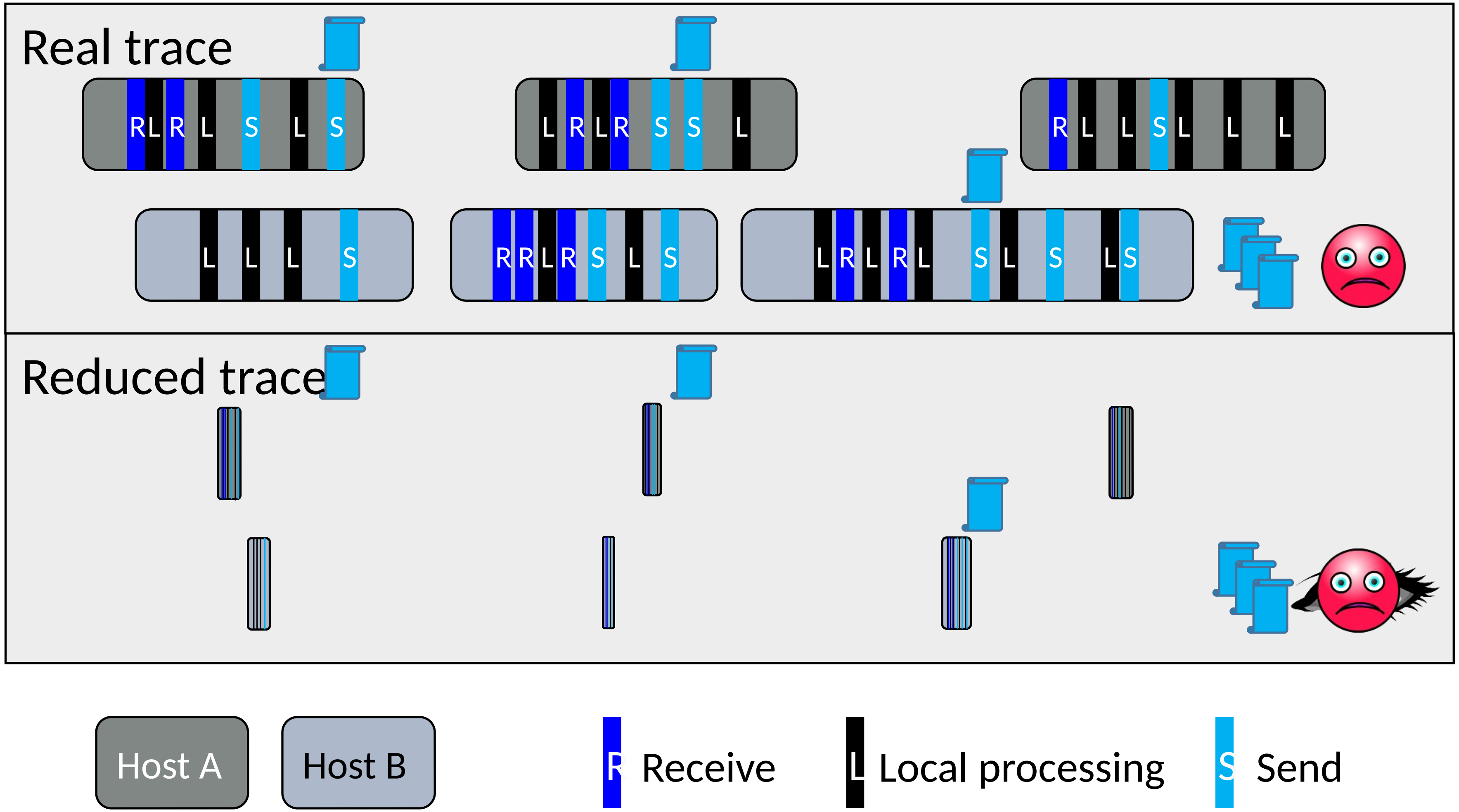
Packets are never received before they're sent



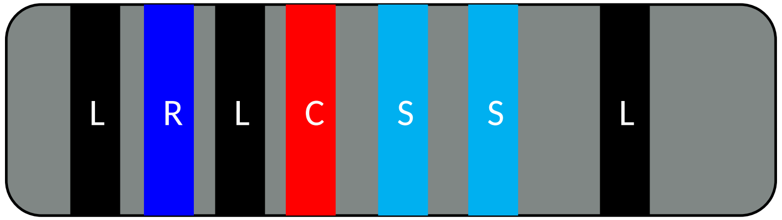
Host A Host B

R Receive L Local processing S Send

The atomic trace preserves failures



Reading the clock is a “non-mover”



You can only have one of these, and it must be the “atomic point”