COMPUTER SCIENCE & ENGINEERING

EECS498-003 Formal Verification of Systems Software

Material and slides created by

Jon Howell and Manos Kapritsos

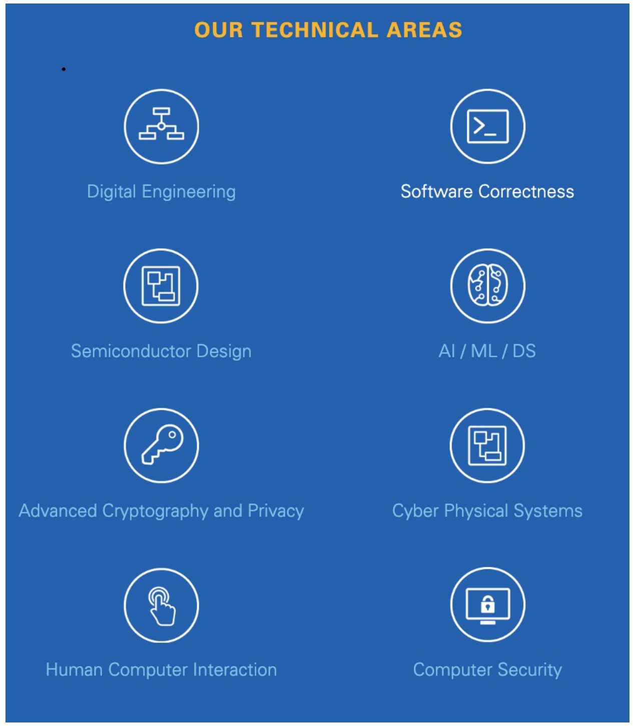
Formal Methods in the Field

Amazon

Cedar: A New Language for Expressive, Fast, Safe, and Analyzable Authorization

"Cedar is used at scale in Amazon Verified Permissions and Amazon Verified Access"

Galois



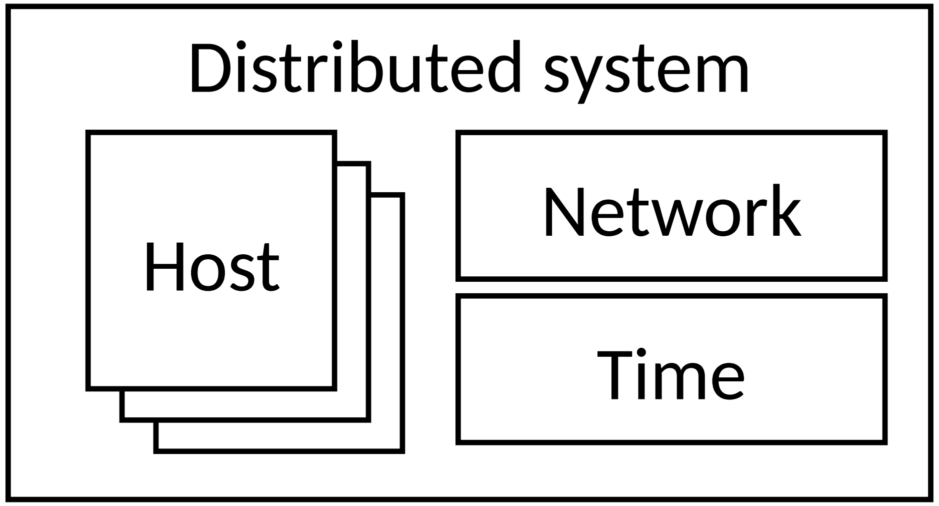
Imandra

Formal Verification of Financial Infrastructure

"Firms like Goldman Sachs, Itiviti and OneChronos rely upon Imandra's algorithm governance tools for the design, regulation and calibration of many of their most complex algorithms."

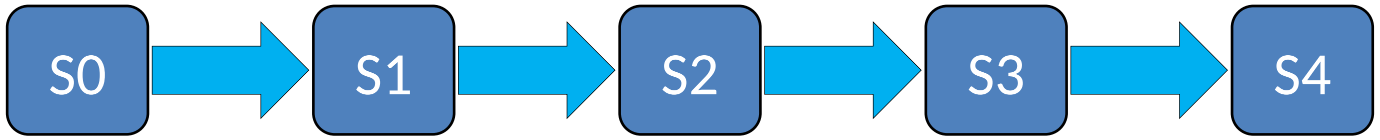
Revisiting the distributed system model

- Composite state machine
  - Hosts
  - Network
  - Time



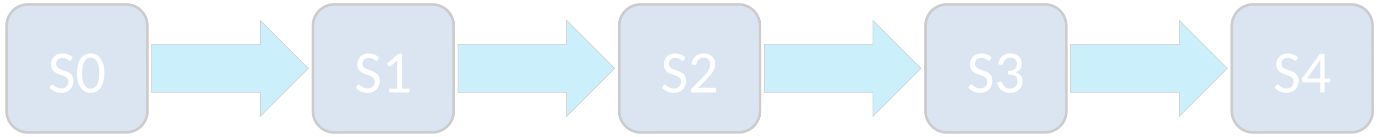
In each step of this state machine:

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



Are the steps *really* atomic?

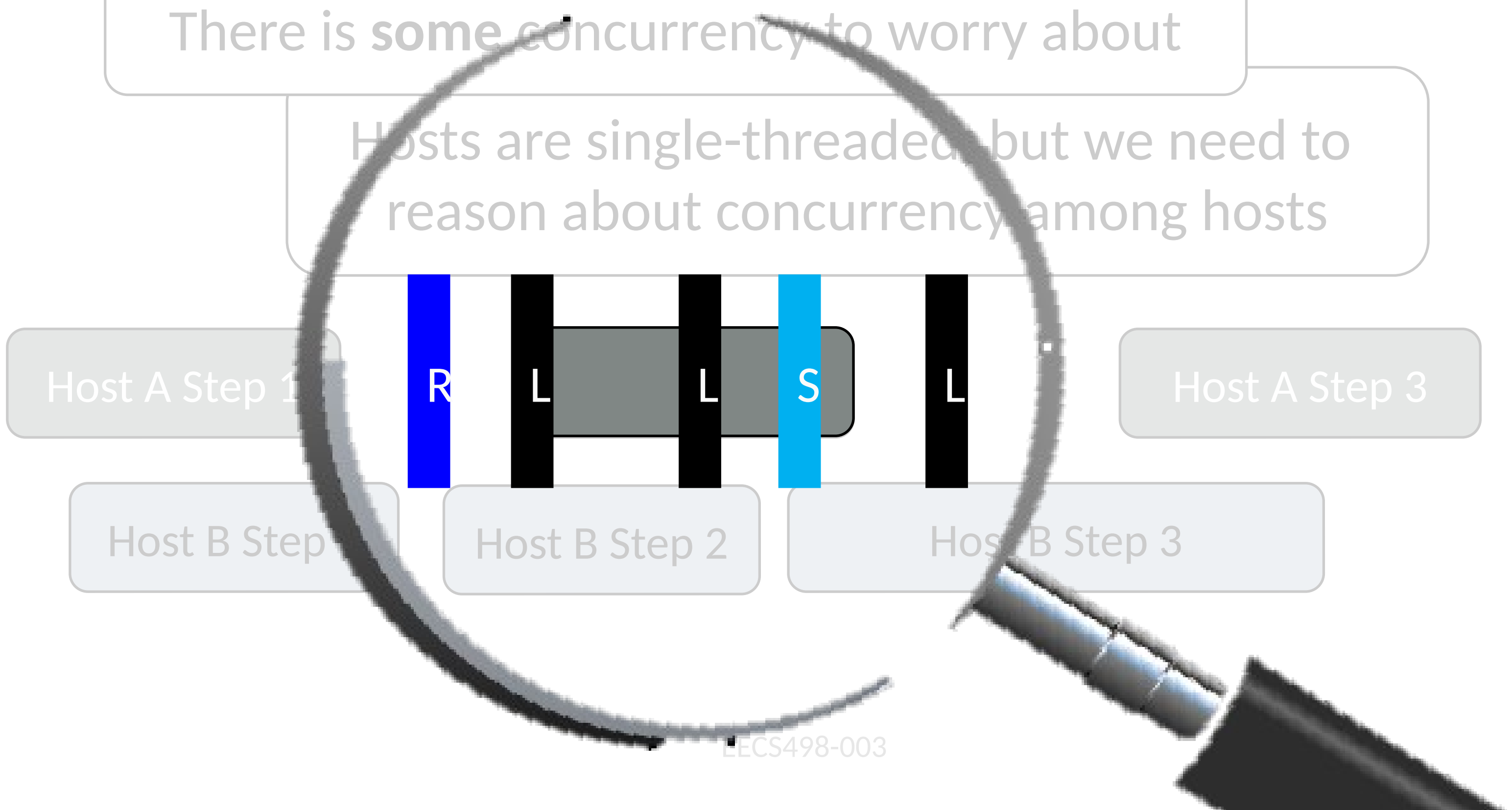
Model:



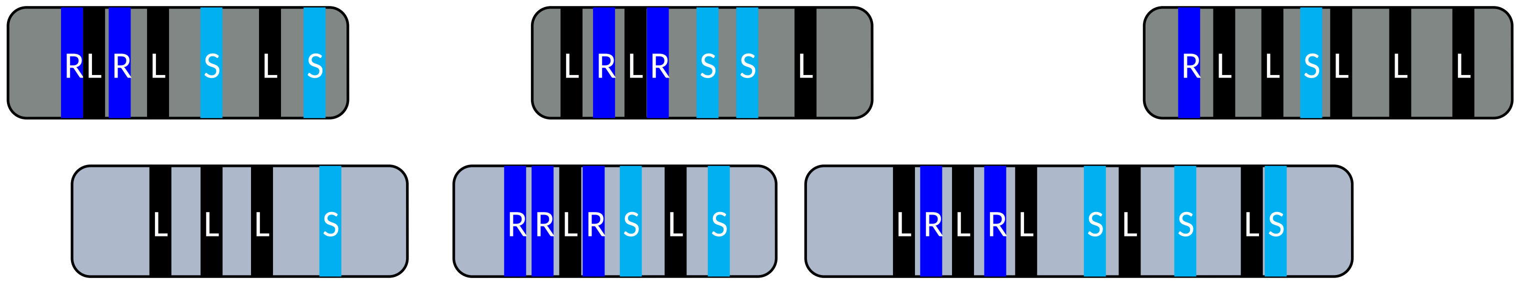
There is **some** concurrency to worry about

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

Reality:



A distributed execution in real life

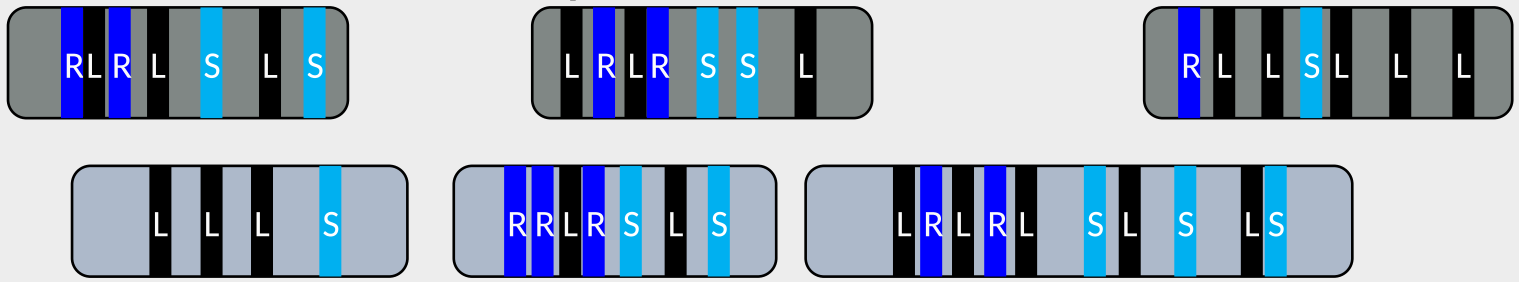


Reason about all possible interleavings of the substeps?

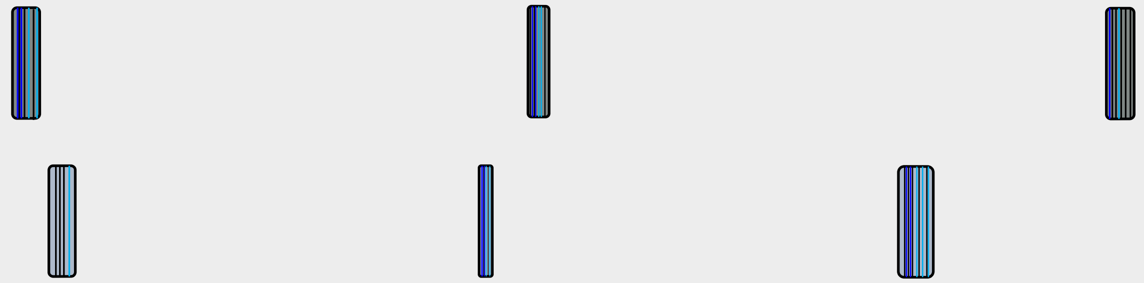
Host A
Host B
R Receive
L Local processing
S Send

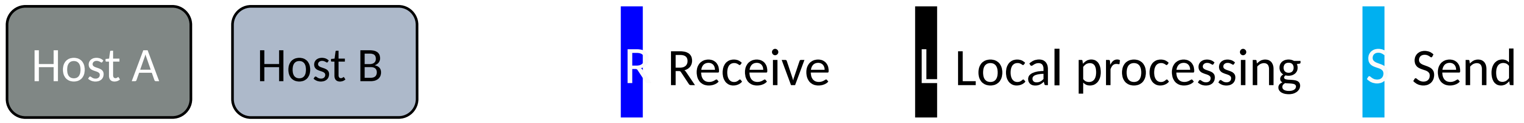
Concurrency containment

Enforce that all receives precede all sends



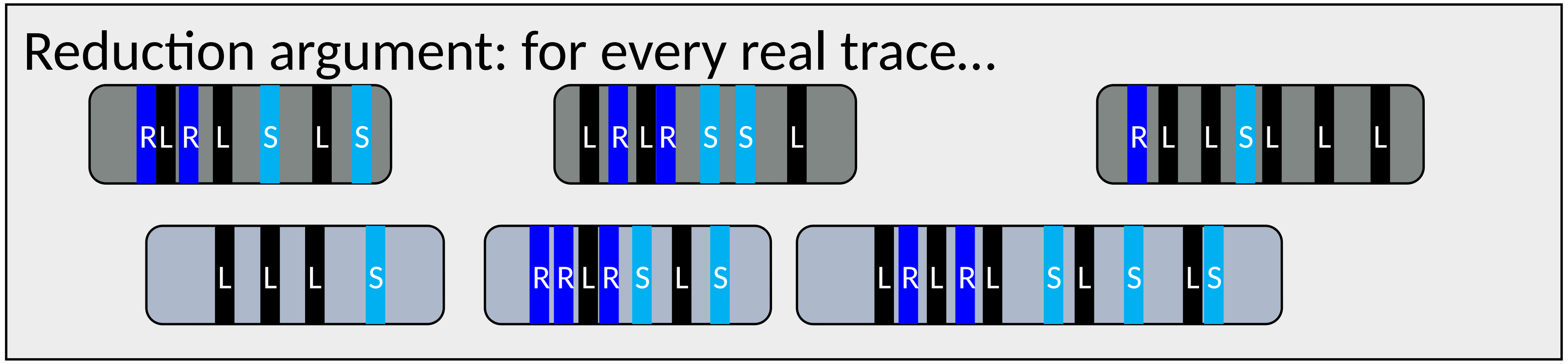
Assume in proof that all host steps are atomic





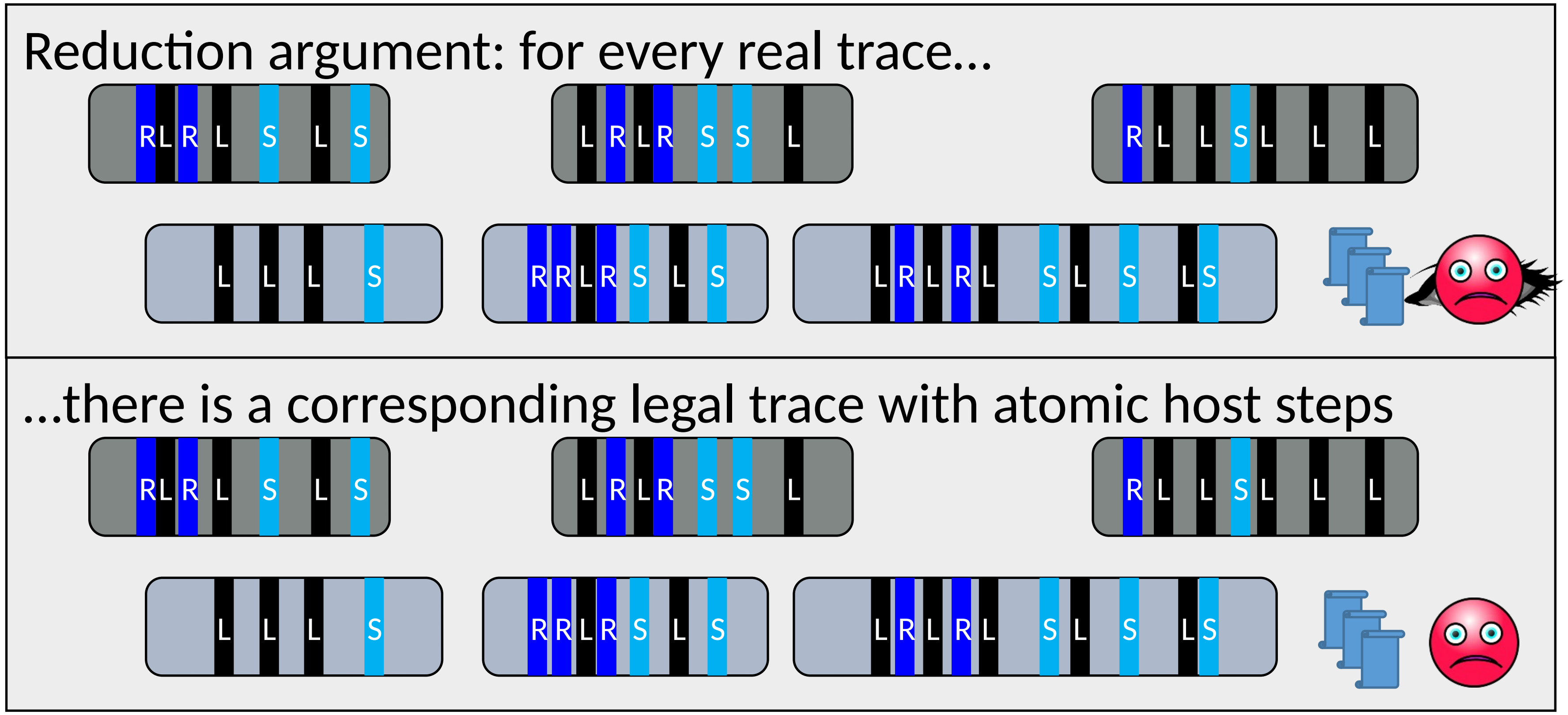
Concurrency containment

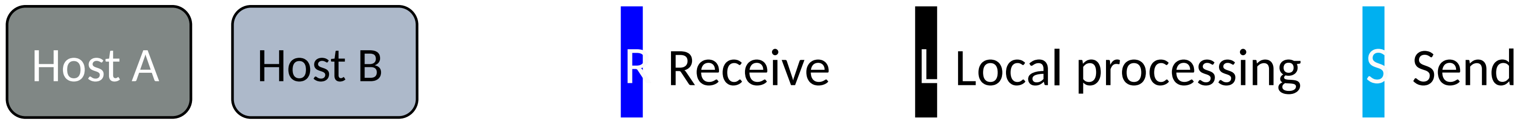
Reduction argument: for every real trace…



Host A
Host B
R Receive
L Local processing
S Send

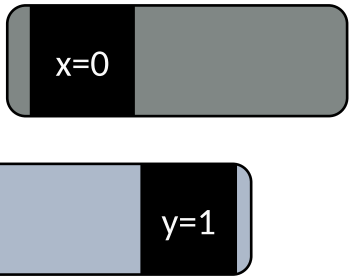
Concurrency containment



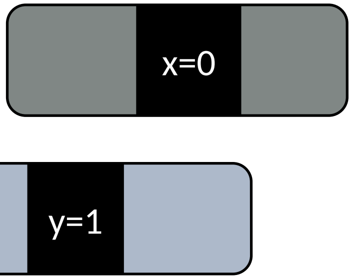


The concept of "movers"

Actual execution



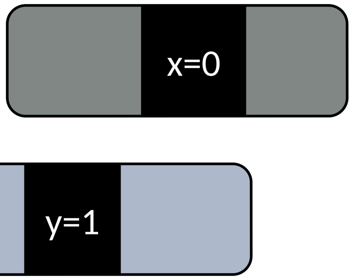
Indistinguishable execution



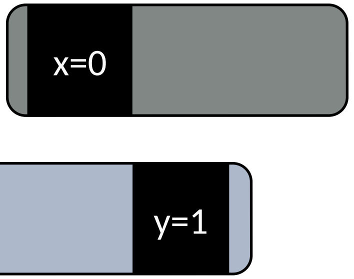
Host A
Host B
R Receive
L Local processing
S Send

Local computations can move either way

Actual execution

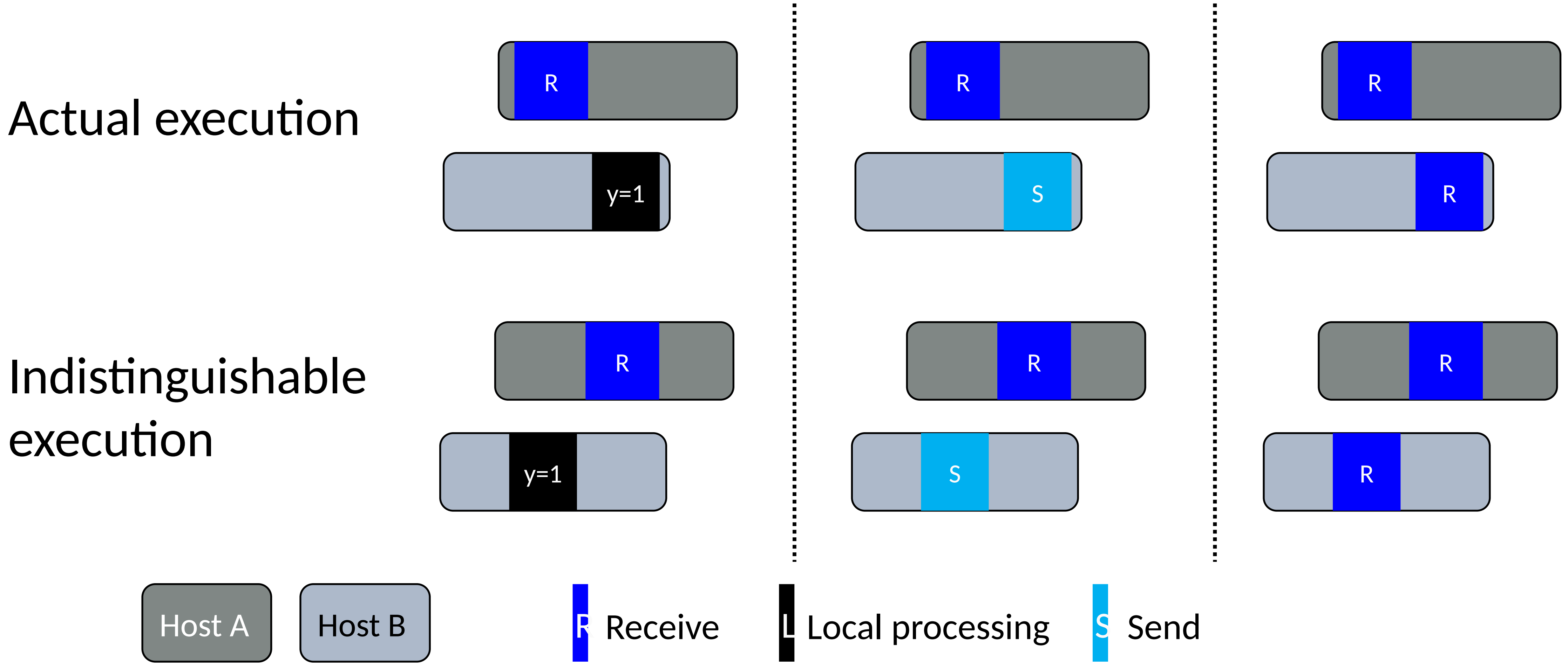


Indistinguishable execution

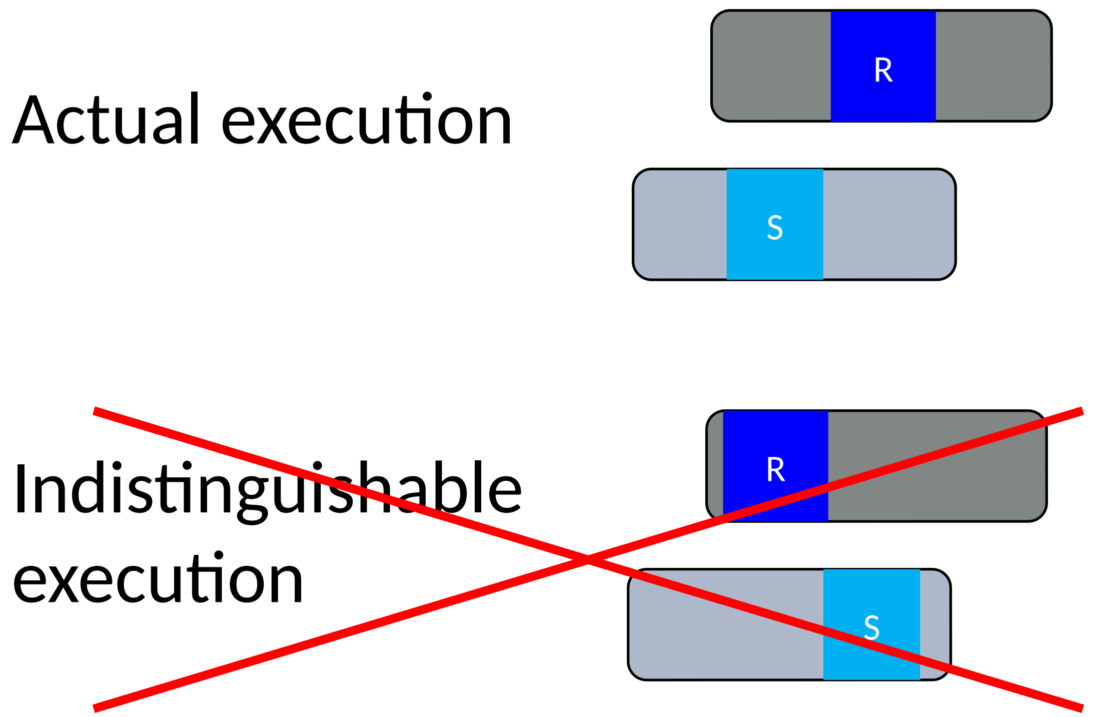


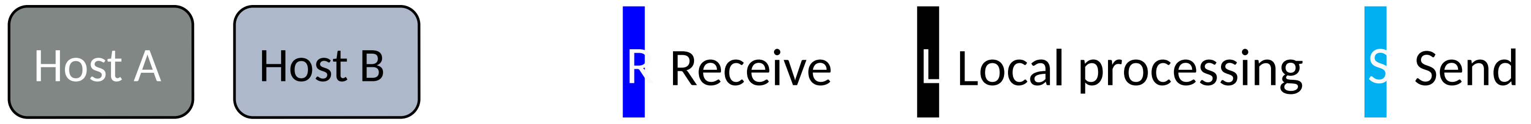
Host A
Host B
R Receive
L Local processing
S Send

Receives are right movers



Receives are not left movers

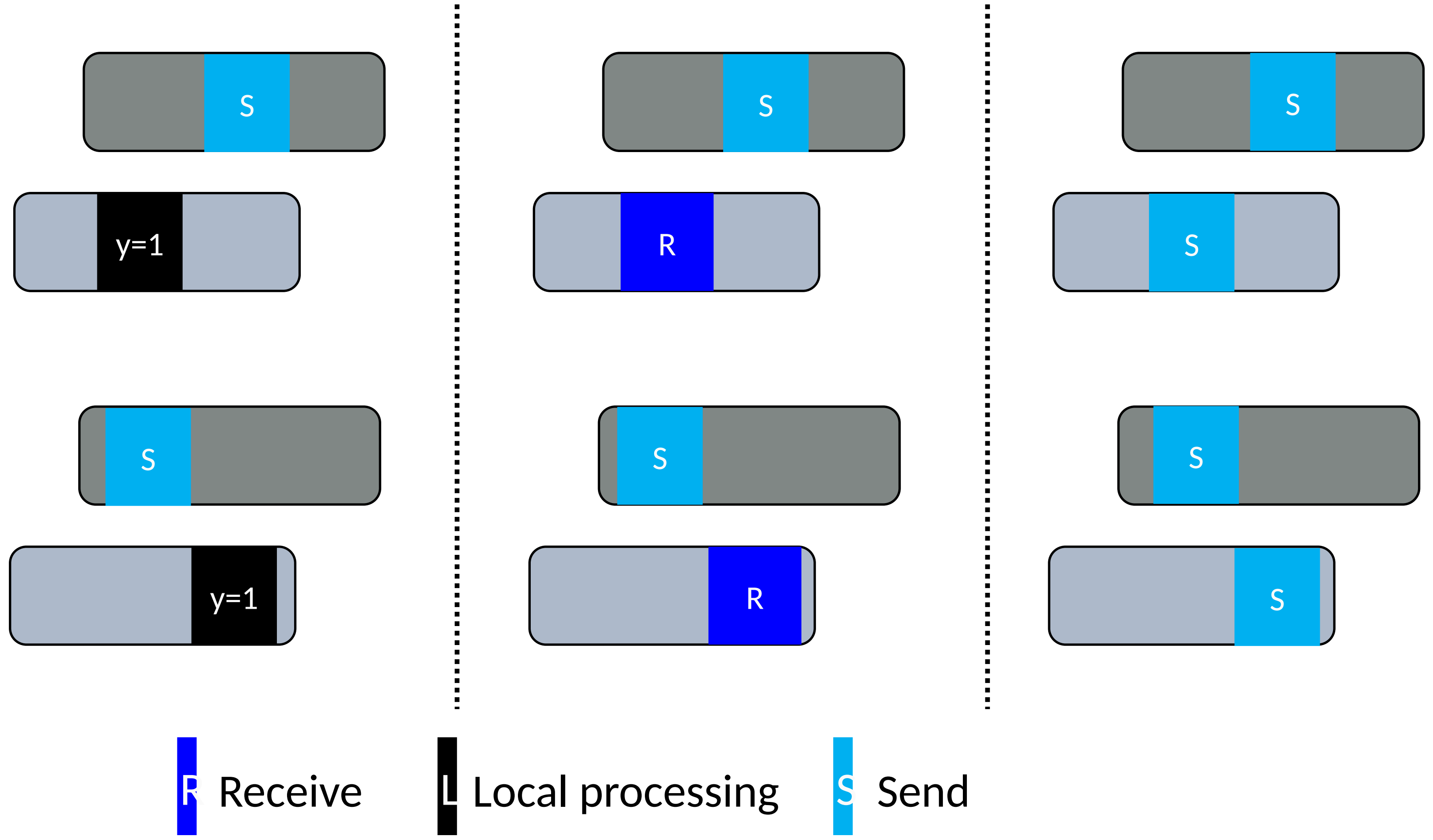




Sends are left movers

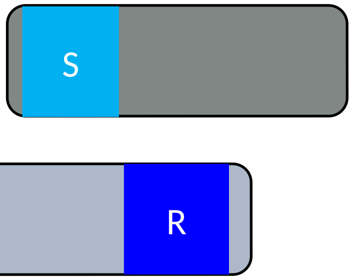
Actual execution

Indistinguishable execution

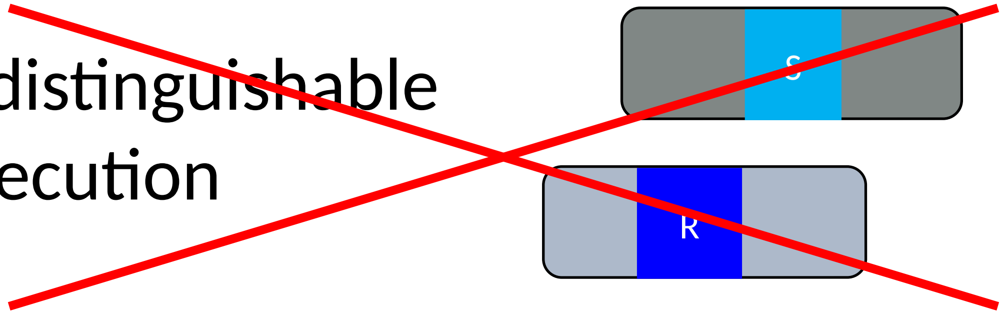


Sends are not right movers

Actual execution



~~Indistinguishable execution~~

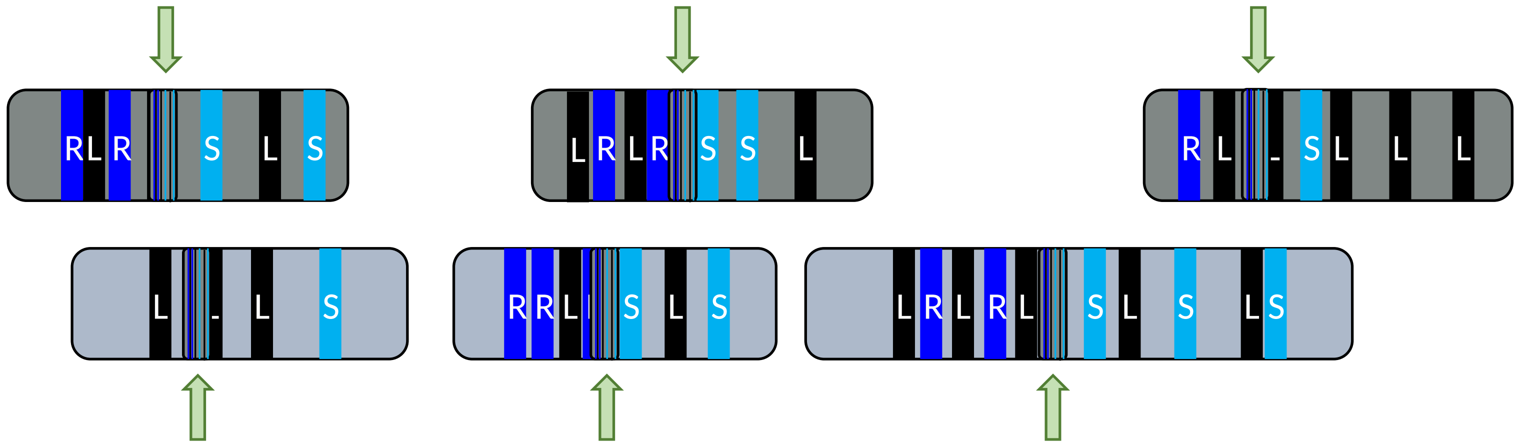


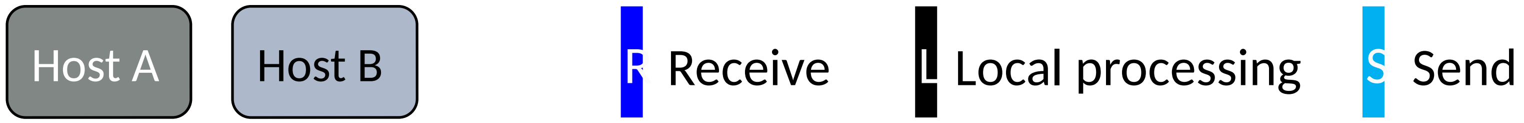
Host A
Host B
R Receive
L Local processing
S Send

Summary of movers

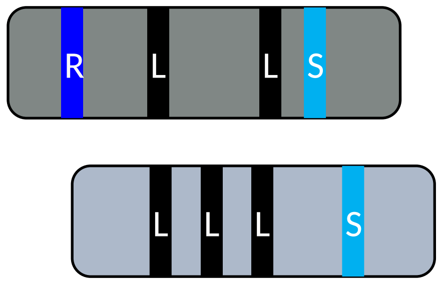
- Local computation moves both ways
- Sends move to the left
- Receives move to the right

Creating the atomic trace

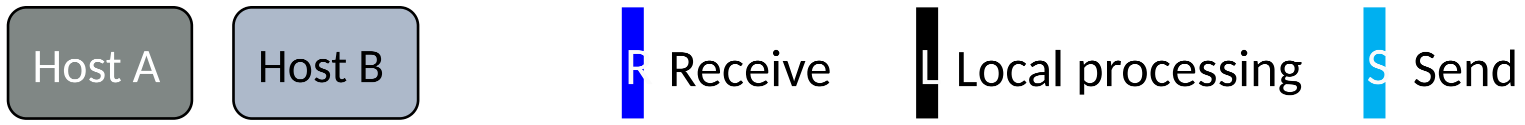




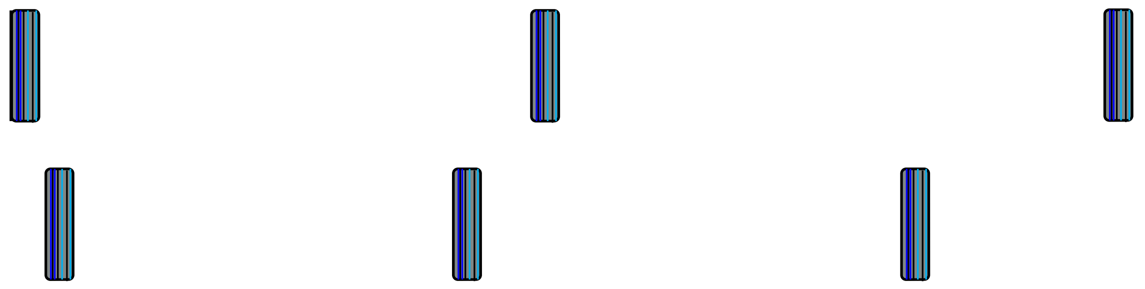
Creating the atomic trace



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

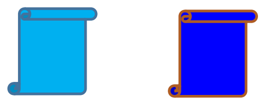


The atomic trace is legal



Local event ordering is preserved

Packets are never received before they're sent



Host A

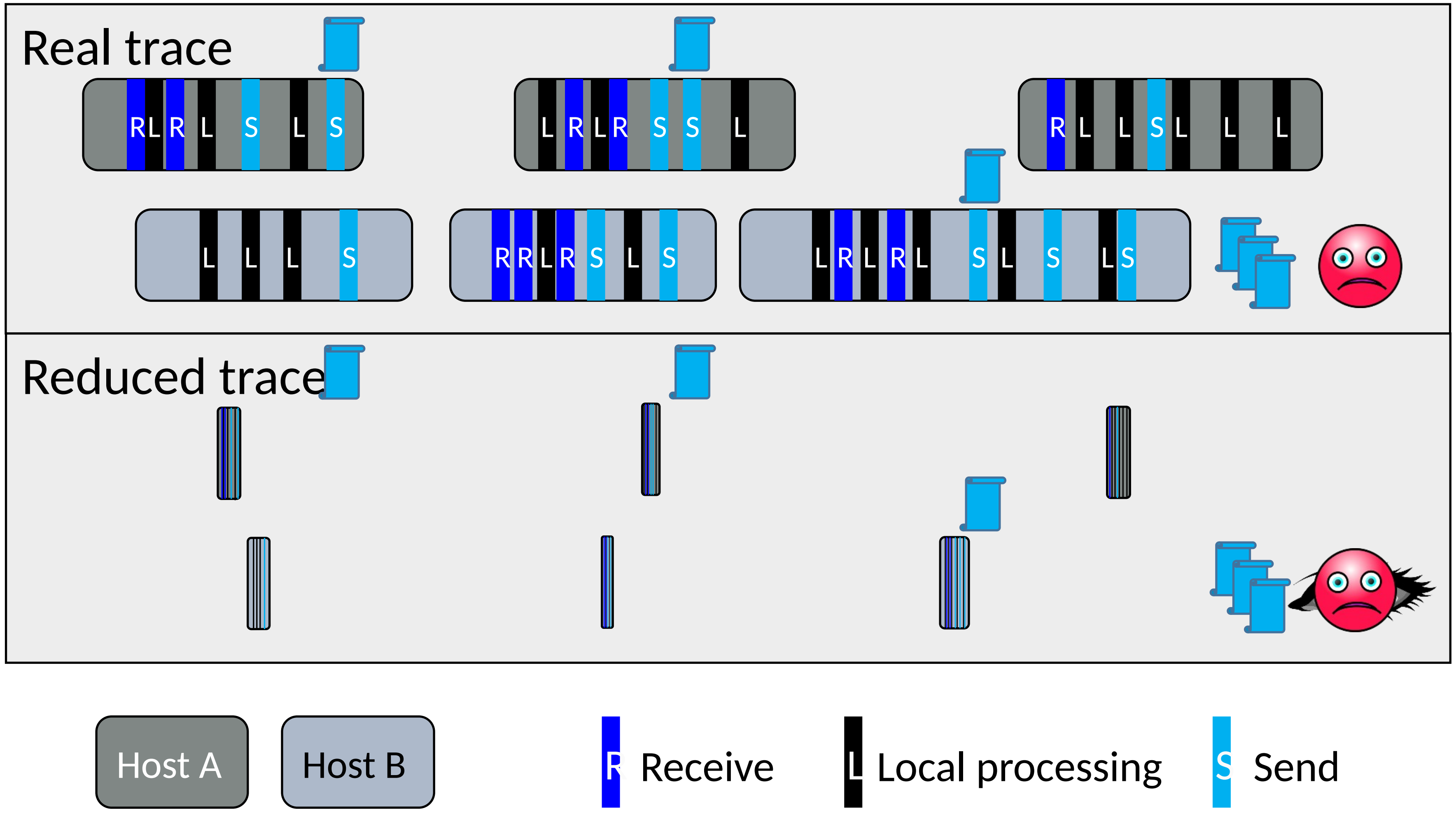
Host B

R Receive

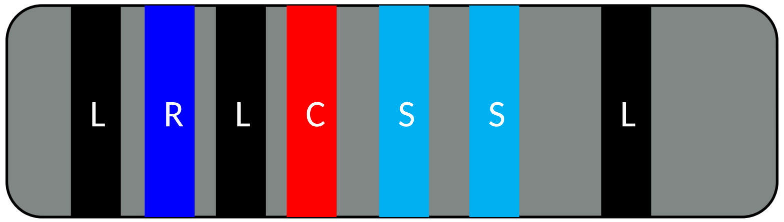
L Local processing

S Send

The atomic trace preserves failures



Reading the clock is a “non-mover”

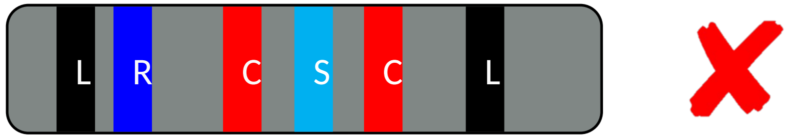


You can only have one of these, and it must be the “atomic point”

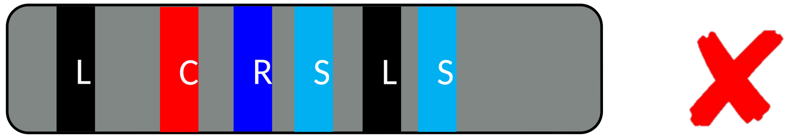
Reduction quiz

Which of the following actions are amenable to reduction?

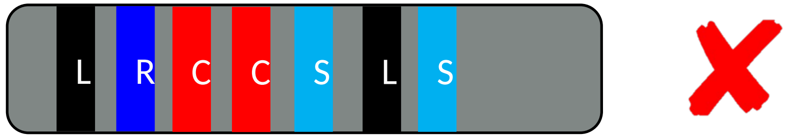
A.



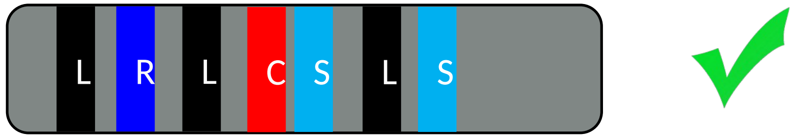
B.



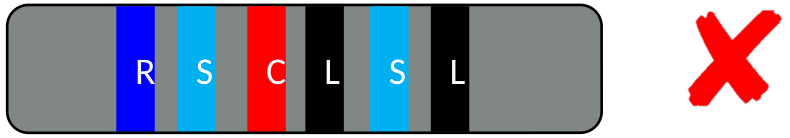
C.



D.



E.

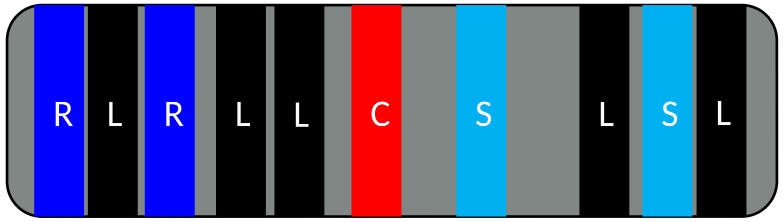


You can only have one clock read, and it must be the "atomic point"

Receives before Clock,
Sends after Clock

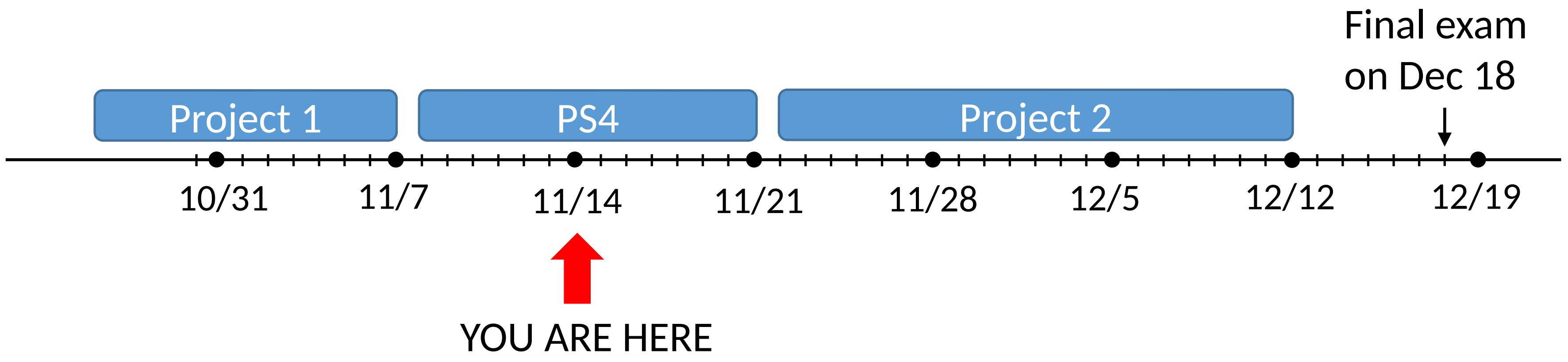
Reduction-enabling obligation

- Each action should be of the form:
  - R* C? S*
  - i.e., Receives then Clock then Sends
    - with local actions interspersed between them



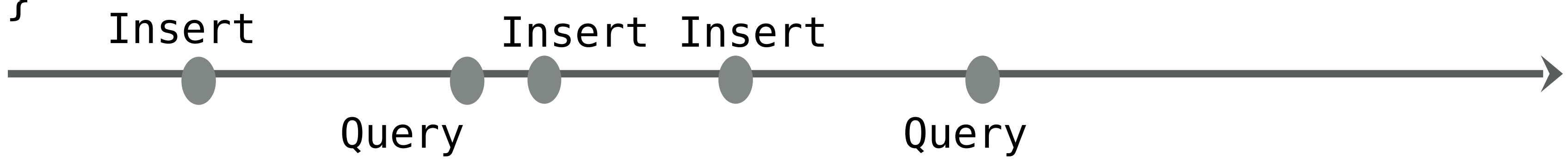
Administrivia

- PS4 (Chapter 6 – Refinement) is due next week

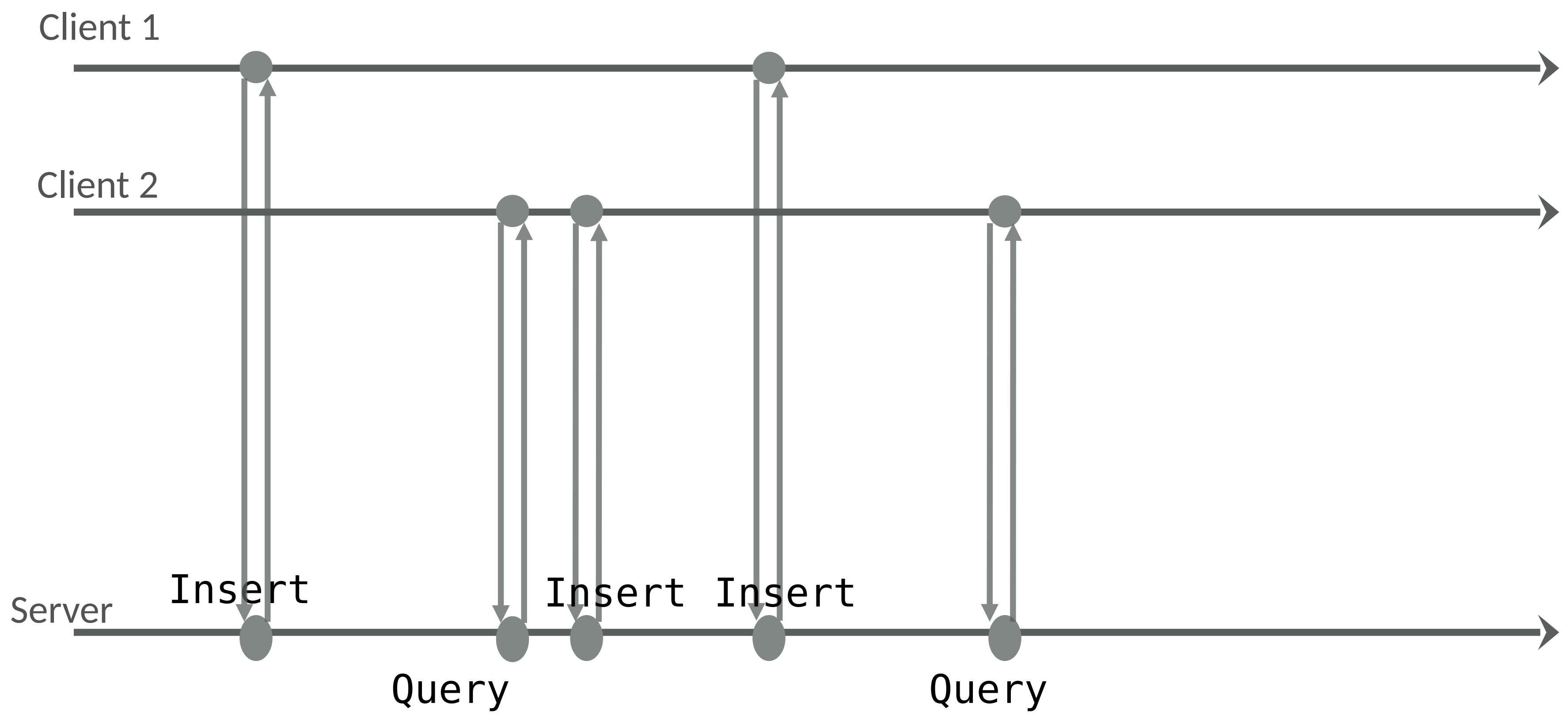


Synchronous specs

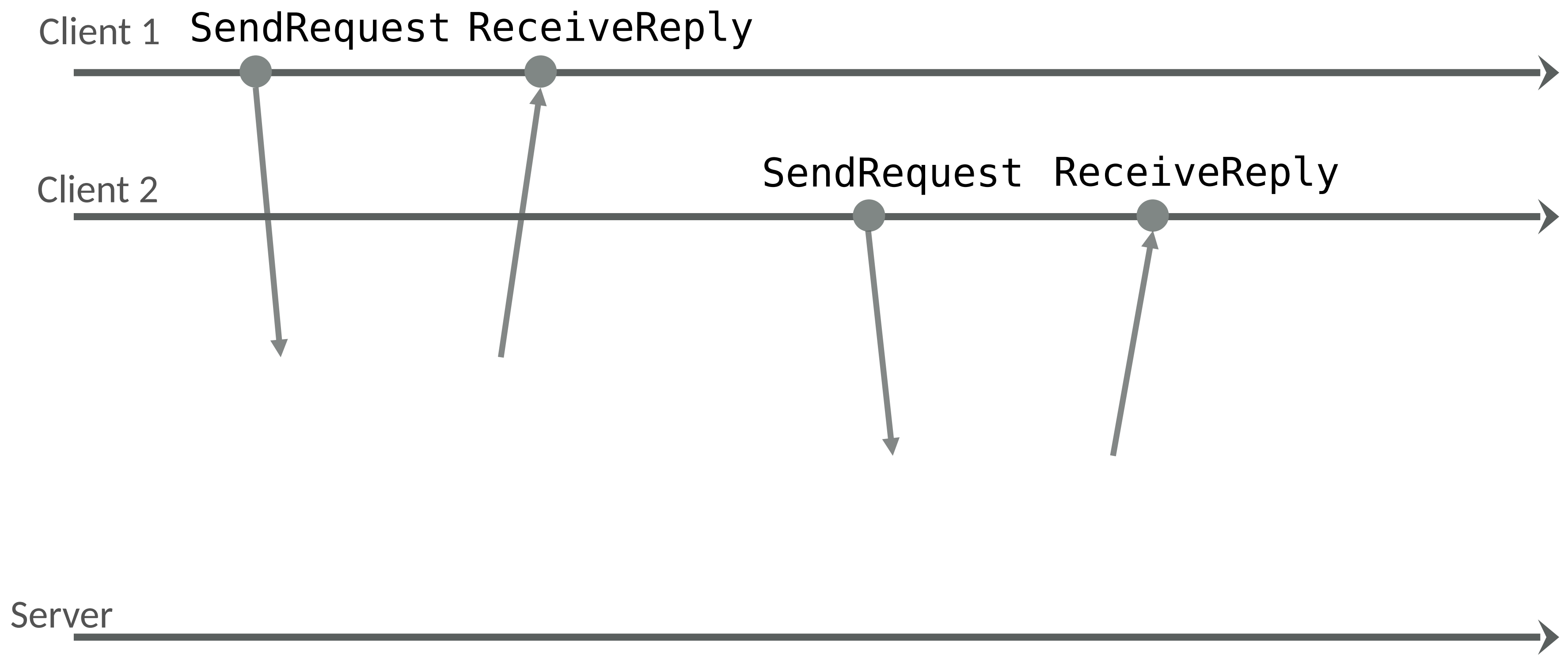
```
module MapSpec {
  datatype Variables = Variables(mapp:map<Key, Value>)

  predicate InsertOp(v:Variables, v':Variables, key:Key,
value:Value) {
    ...
  }

  predicate QueryOp(v:Variables, v':Variables, key:Key,
output:Value) {
    ...
  }
}
```



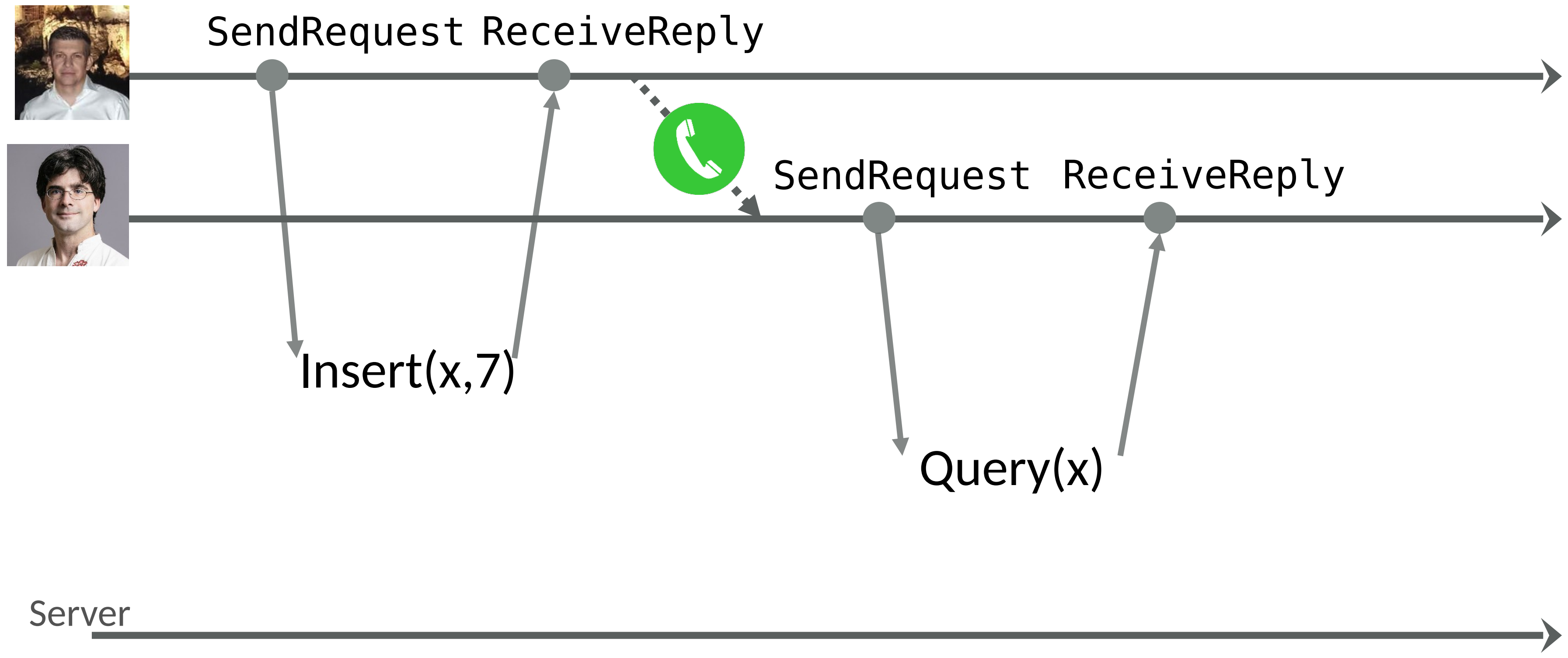
Synchronous specs



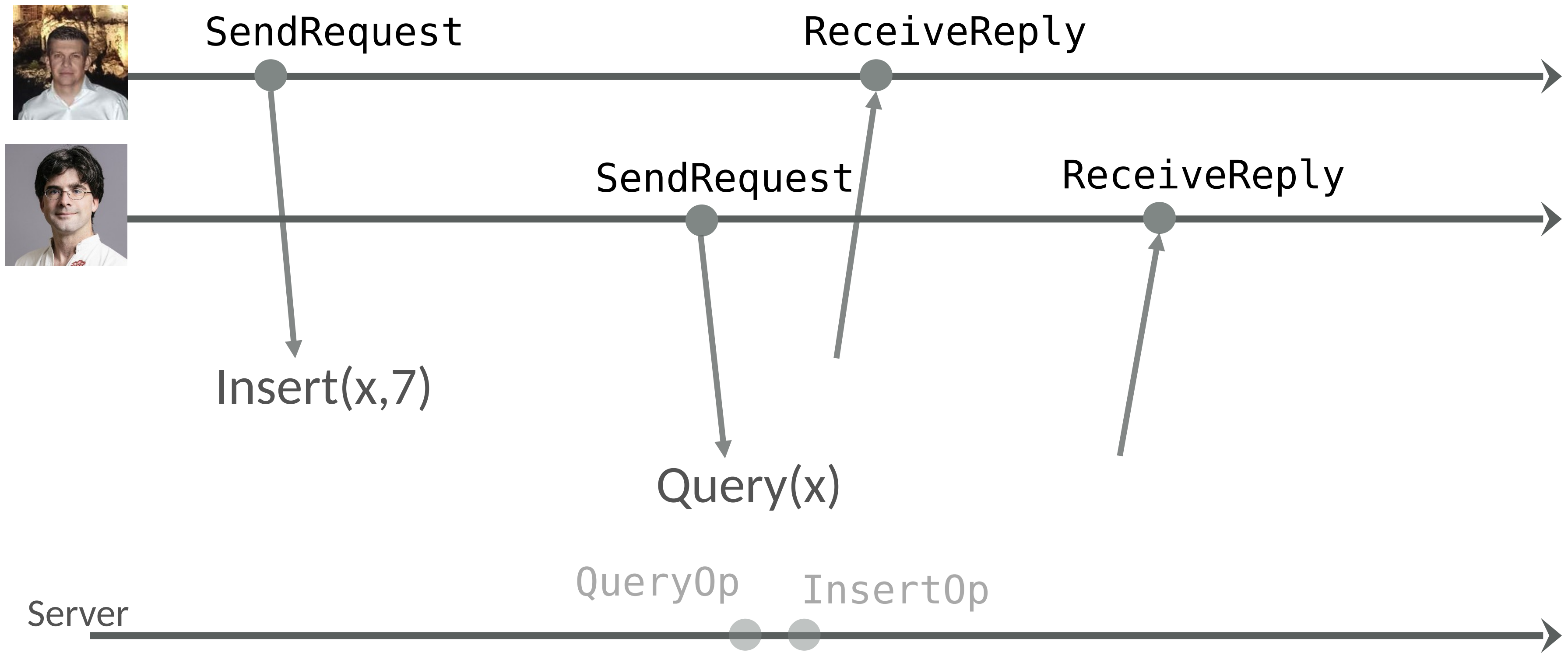
Asynchrony in real life



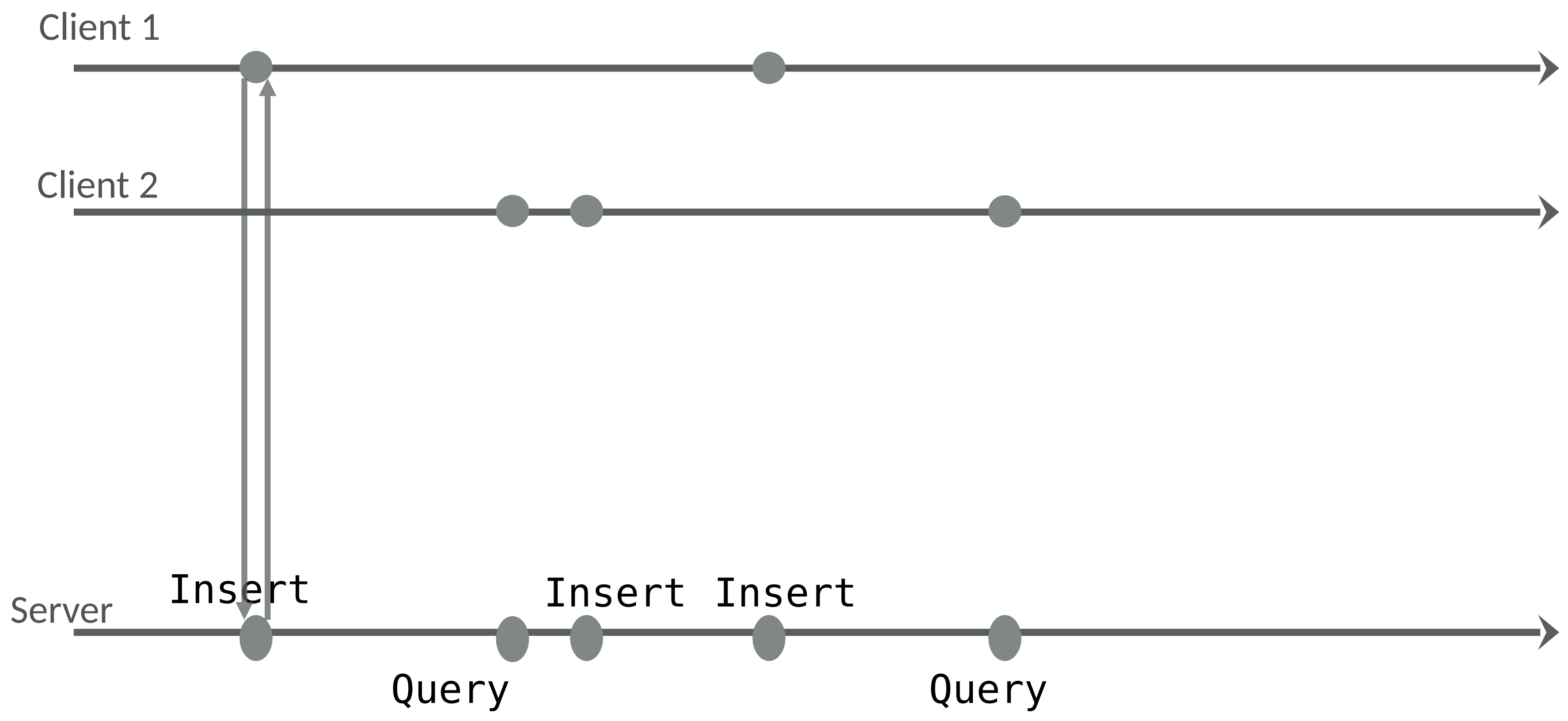
Linearizability



Linearizability

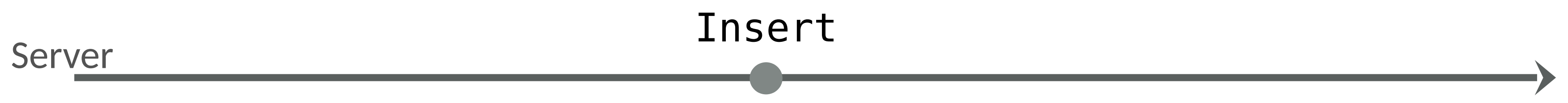


The limitation of Synchronous specs

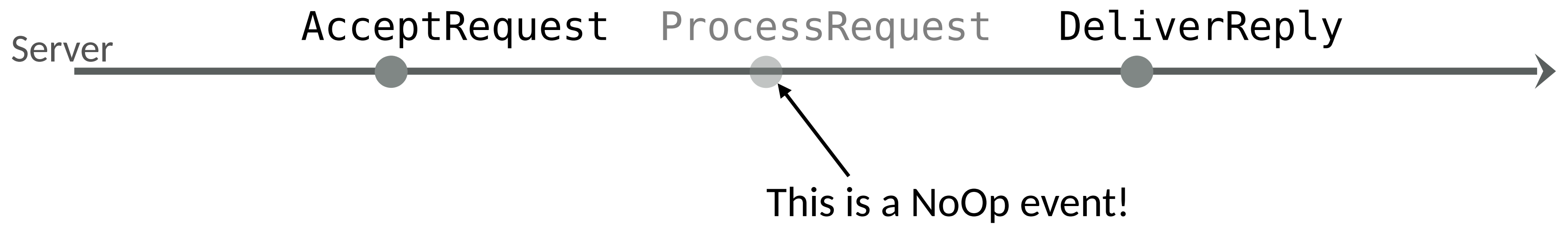


The answer: more events!

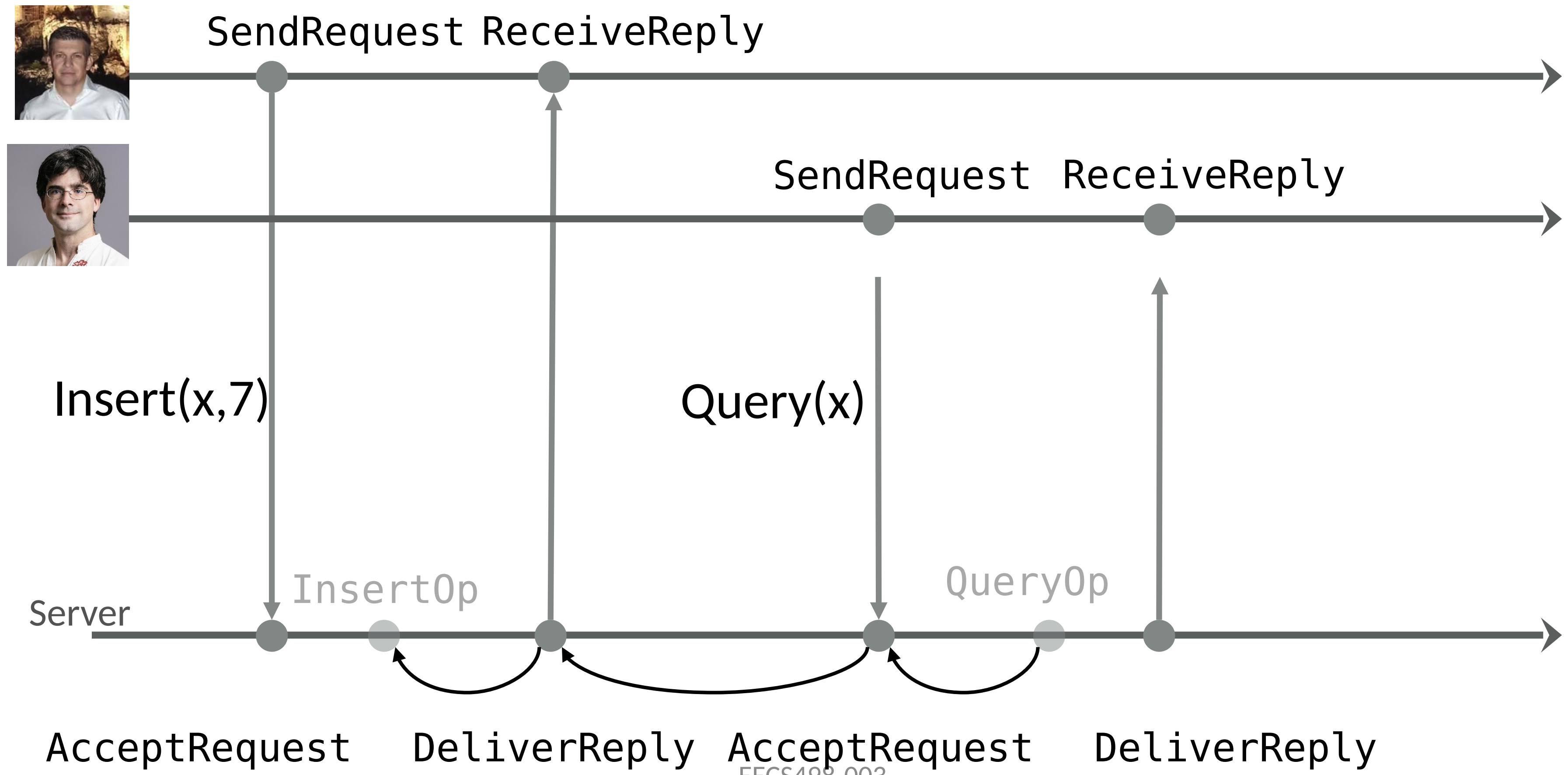
Instead of:



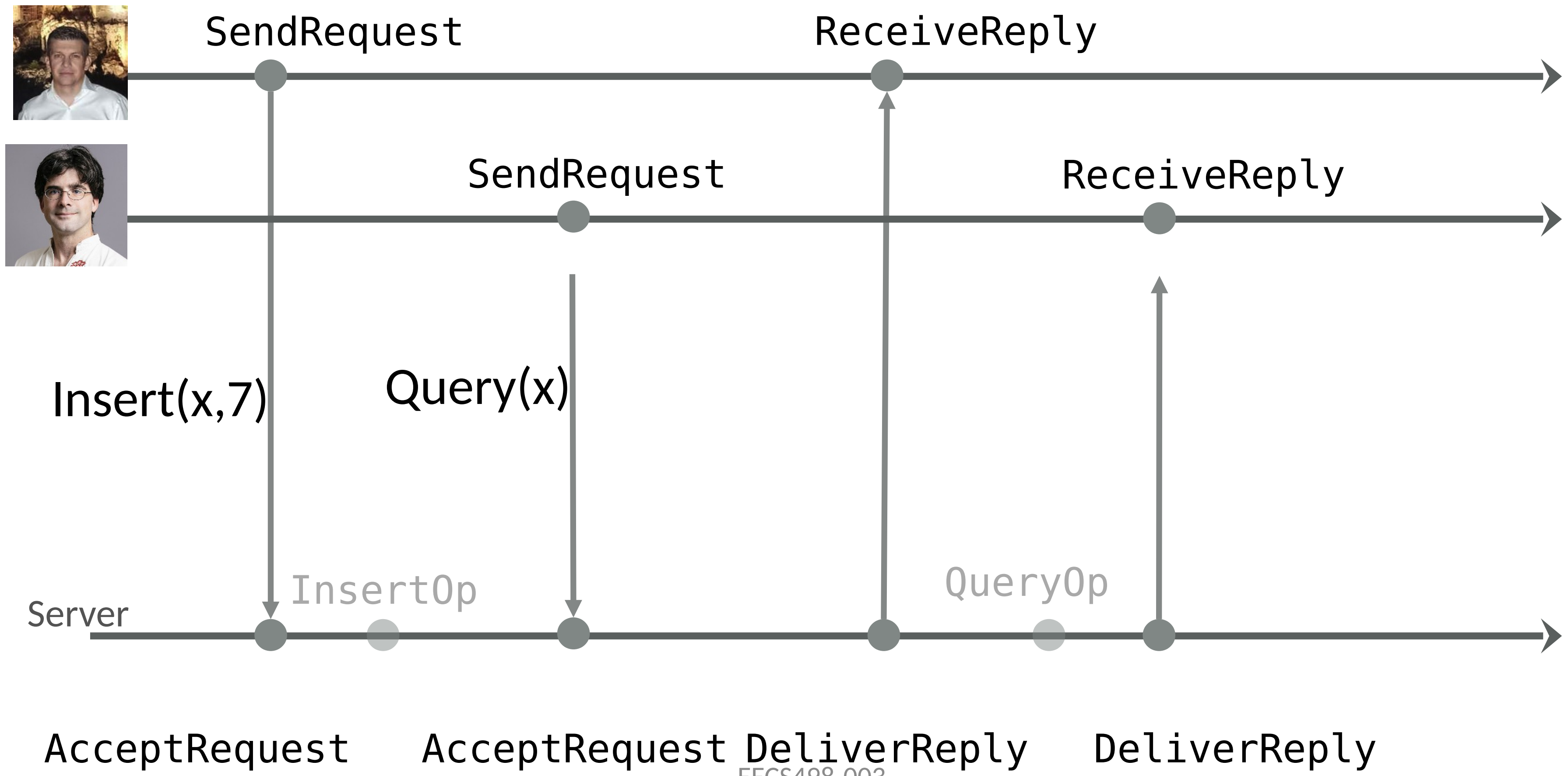
Use this:



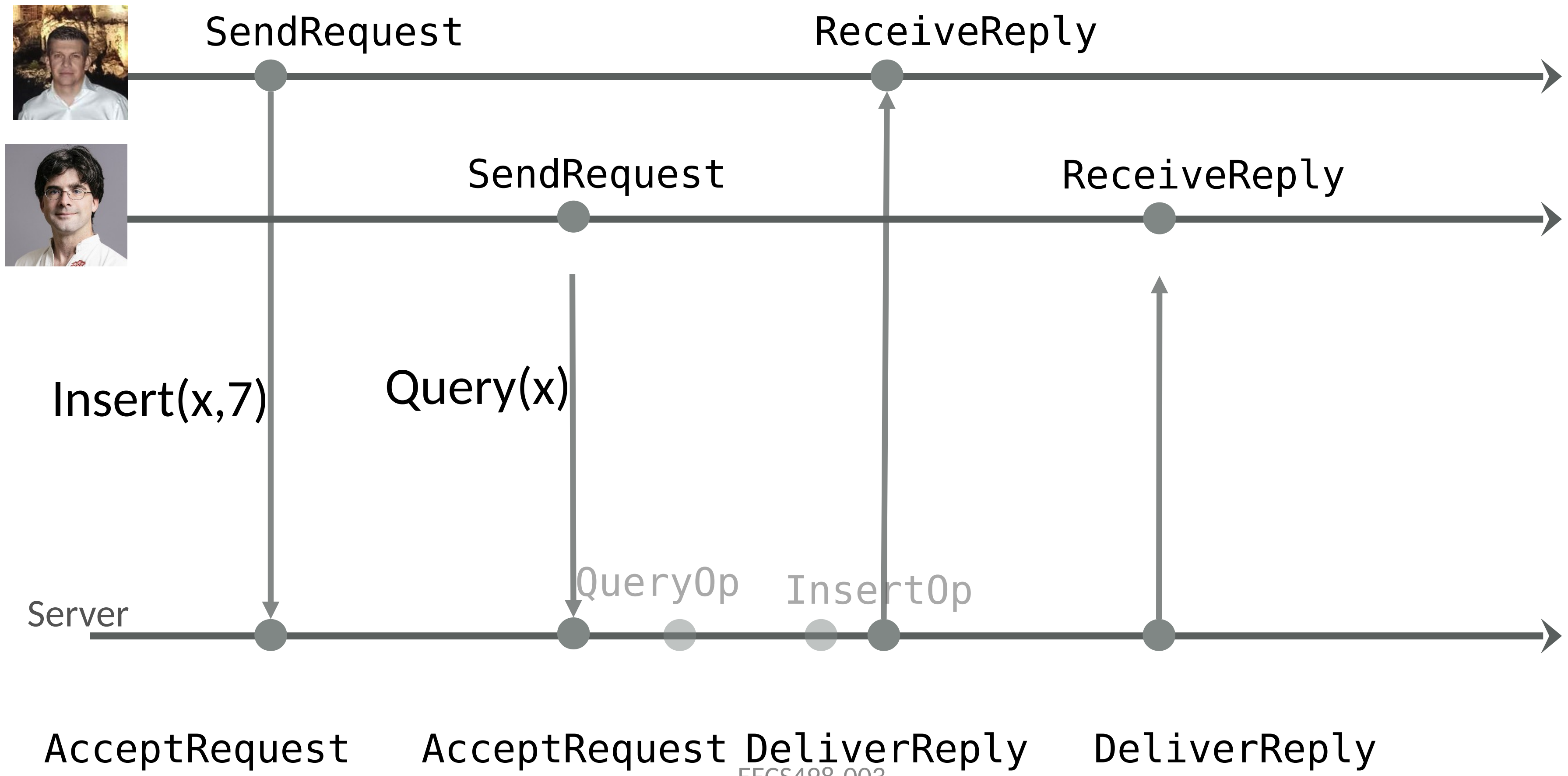
Example run


SendRequest
ReceiveReply
SendRequest
ReceiveReply
Insert(x,7)
Query(x)
InsertOp
QueryOp
Server
AcceptRequest
DeliverReply
AcceptRequest
DeliverReply

Example run #2



Example run #2



Administrivia

- No class this Monday, 04/08
- No lab this Friday, extra OH instead
  - Keshav will make an announcement with the exact time on Piazza
- Final exam logistics
  - Time: May 2, 8-10am
  - Location: This classroom, COOL G906
  - If you have special accommodations, I will email you about the time/place

Dafny: finite set heuristics

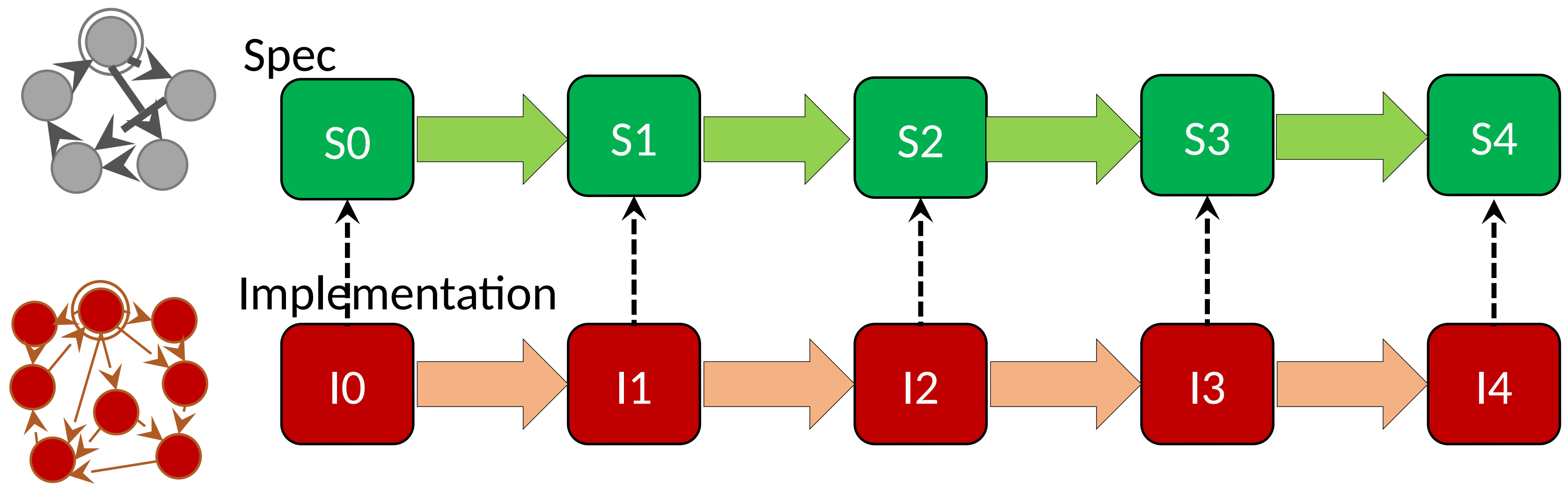
```
predicate IsEven(x:int) {
  x/2*2==x
}

predicate IsModest(x:int) {
 0 <= x < 10
}

lemma IsThisSetFinite() {
  var modestEvens := set x | IsModest(x) &&
IsEven(x);
  assert modestEvens == {0,2,4,6,8};
}
```

Error: the result of a set comprehension must be finite, but Dafny's heuristics can't figure out how to produce a bounded set of values for 'x'

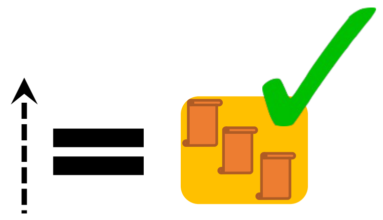
Dafny: finite set heuristics

```
predicate IsEven(x:int) {
  x/2*2==x
}

predicate IsModest(x:int) {
  0 <= x < 10
}

function ModestNumbers() : set<int> {
  set x | 0 <= x < 10
}

lemma IsThisSetFinite() {
  var modestEvens := set x | x in ModestNumbers() &&
IsEven(x);
  assert modestEvens == {0,2,4,6,8};
}
```

Refinement (down to an implementation)



method Main()

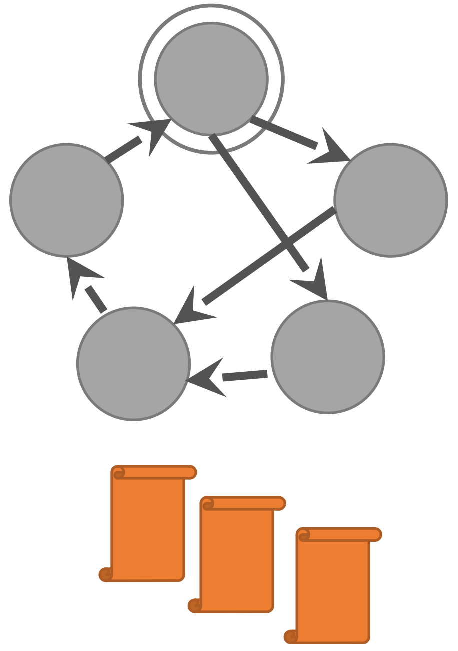


Example: Map spec

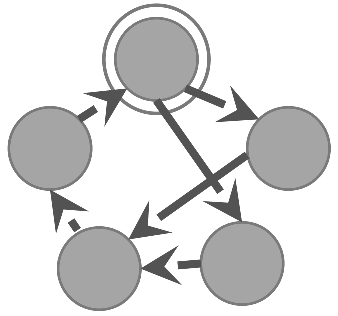
```
datatype Variables = Variables(mapp:map<Key, Value>)
```

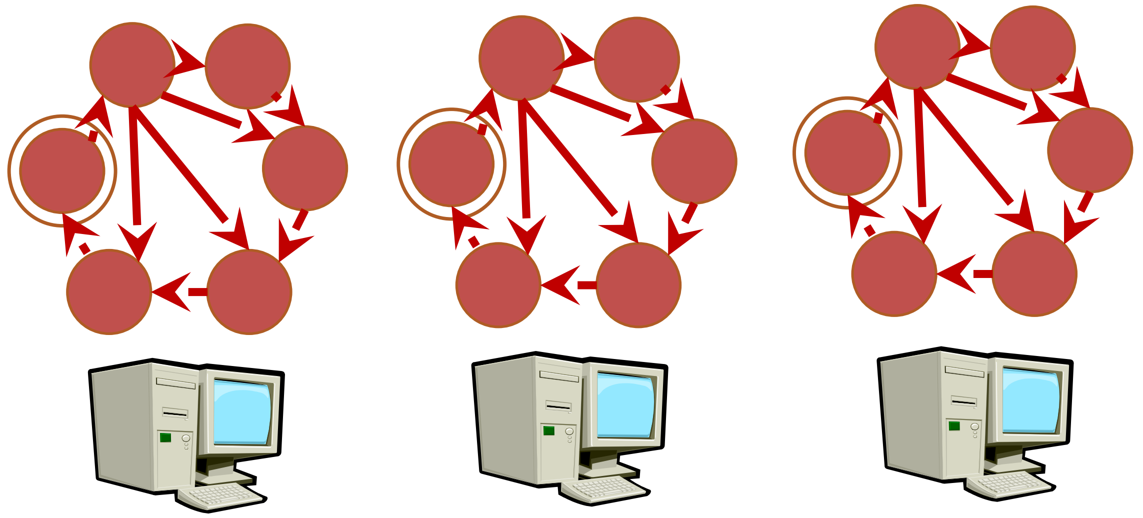
```
predicate SpecInit(v:Variables)
{
    v == map[]
}
```

```
predicate SpecNext(v:Variables,
                   v':Variables)
{
    || InsertOp()
    || QueryOp()
}
```



Implementation

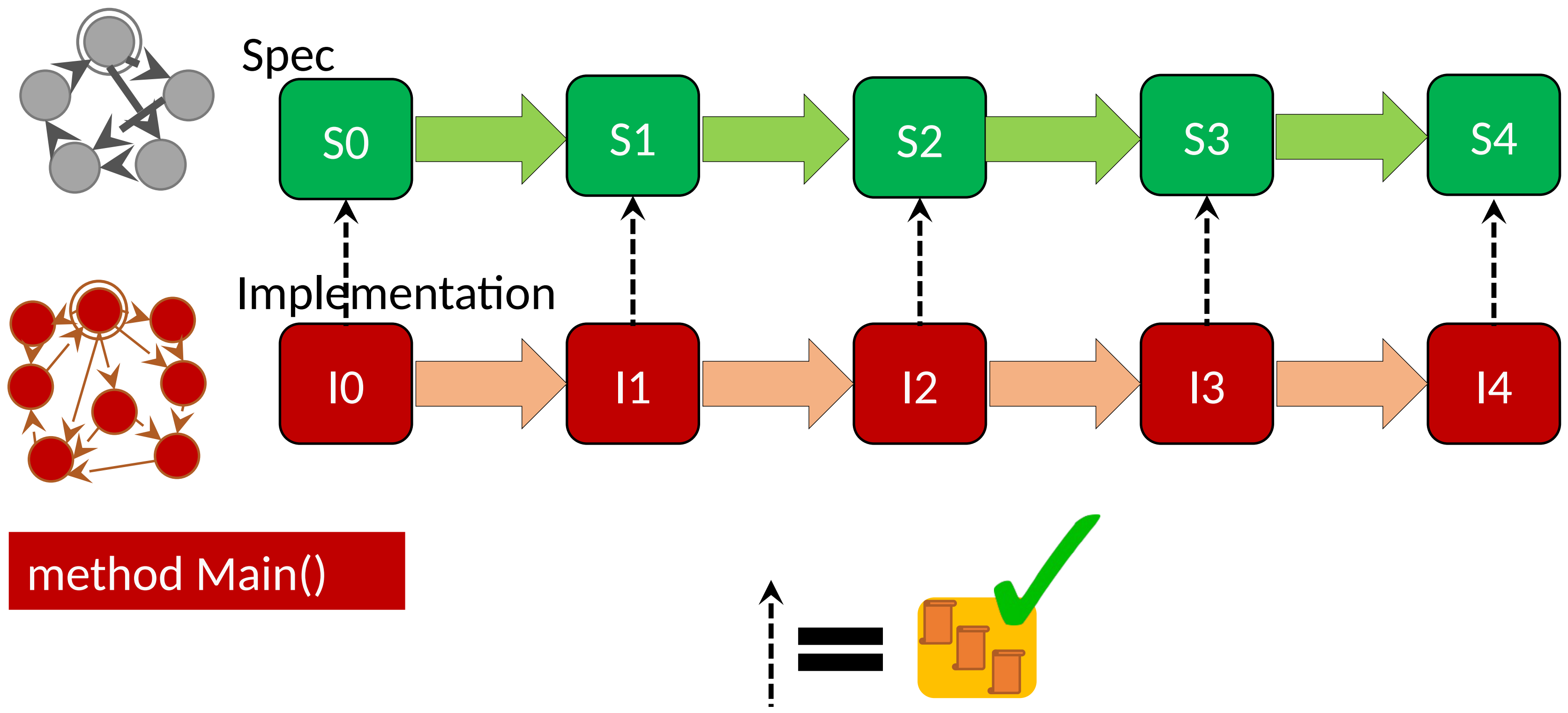




```
method Main()
{
    var v:ImplVariables;
    v := ImplInit();
    while (true) {
        v := EventHandler(v);
    }
}
```

Host implementation is a single-threaded event-handler loop

We could do direct refinement, but…



Complexit
implemen

rotocols

Using effic
structures

global

Memory management

Dealing with hosts
acting concurrently

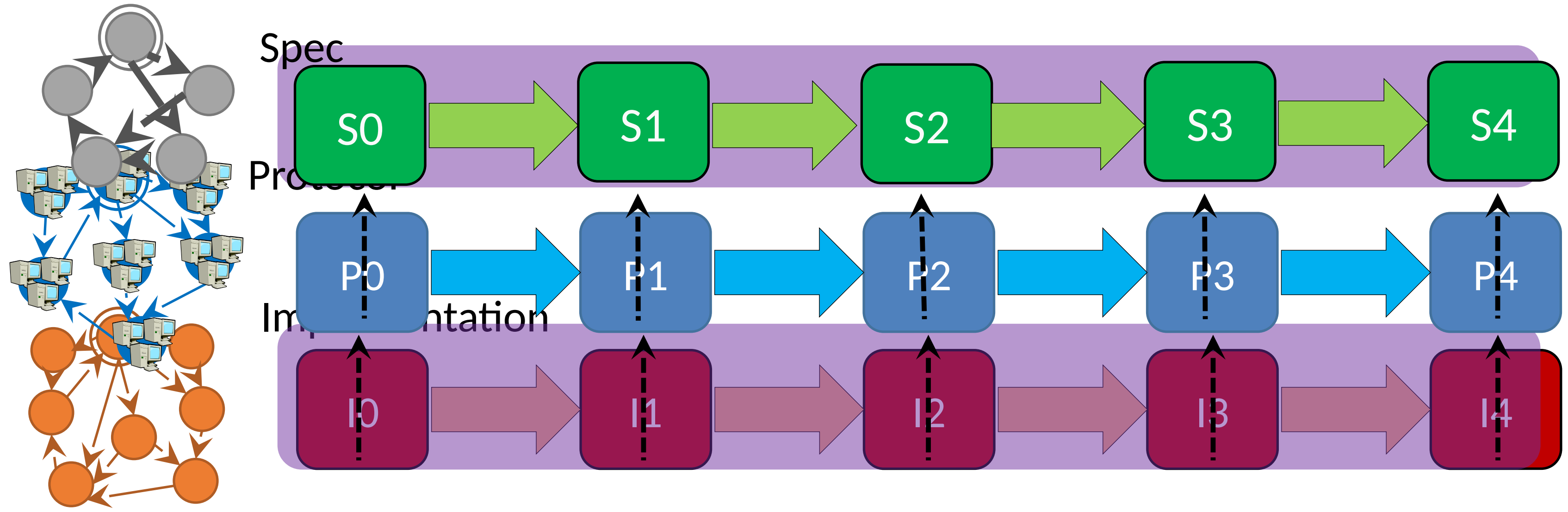
Avoiding integer
overflow

Ensuring progress

Separation of concerns

Complexities of implementation	Subtleties of distributed protocols
Using efficient data structures	Maintaining global invariants
Memory management	Dealing with hosts acting concurrently
Avoiding integer overflow	Ensuring progress

Two-level refinement

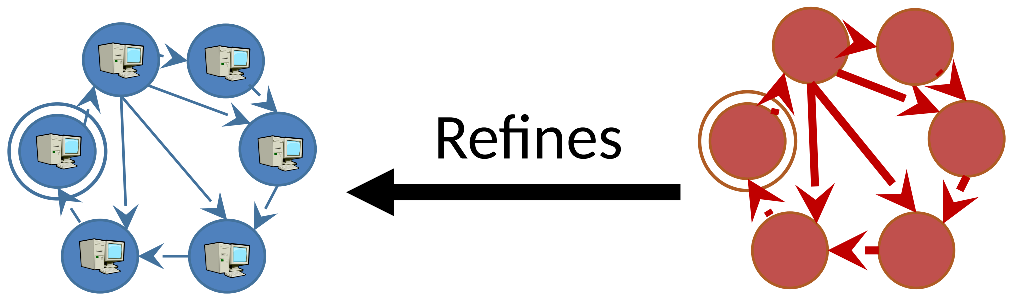


Protocol Layer

```
seq<int>
array<uint64>
```

```
predicate ProtocolNext(v:HostState, v':HostState)
method EventHandler(v:HostState) returns (v':HostState)
```

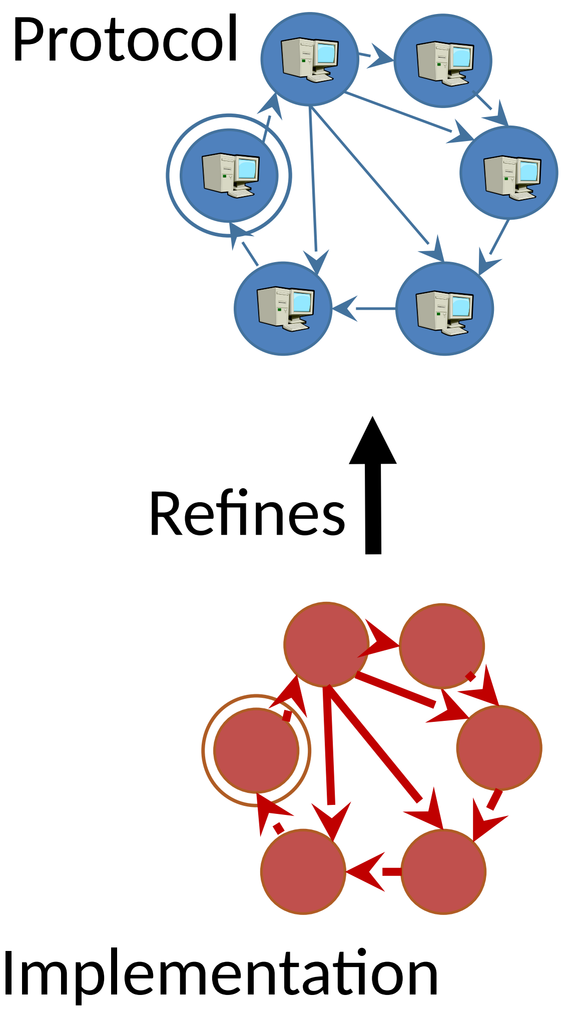
```
type Message = MessageRequest() | MessageReply() | ...
type Packet = array<byte>
```



Protocol steps (predicates)

Implementation (methods)

From Implementation to Protocol

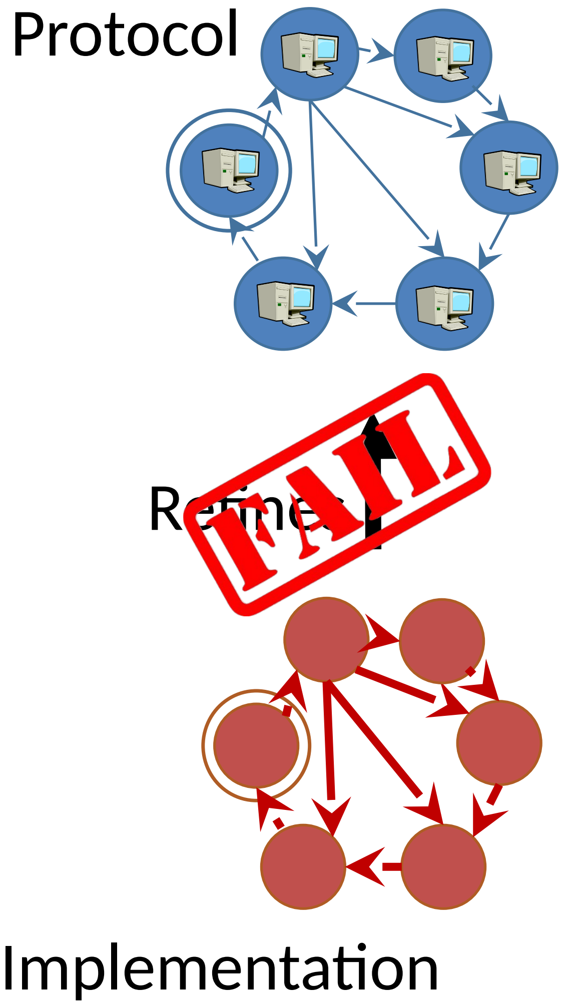


```
datatype Variables = Variables(x:int, y:int)

predicate Init(v:Variables)
{
        v.x == 0;
        v.y == 5;
}
```

```
function Abstraction(impl:ImplVariables) : Variables
{
        Variables(int(impl.x), int(impl.y))
}
```

```
datatype ImplVariables = ImplVariables(x:uint64, y:uint64)

method InitImpl(v:ImplVariables)
        ensures Init(Abstraction(v))
{
        v.x := 0;
        v.y := 5;
}
```

From Implementation to Protocol



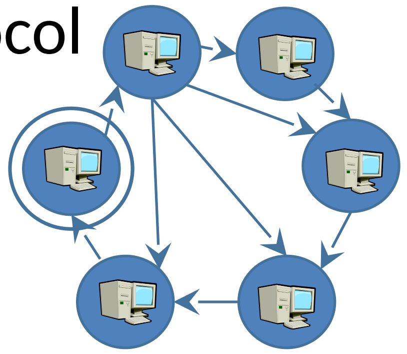
```
datatype Variables = Variables(x:int, y:int)

predicate MoveNorth(v:Variables, v':Variables)
{
          v'.x == v.x;
          v'.y == v.y + 1;
}
```

```
function Abstraction(impl:ImplVariables) : Variables
{
          Variables(int(impl.x), int(impl.y))
}
```

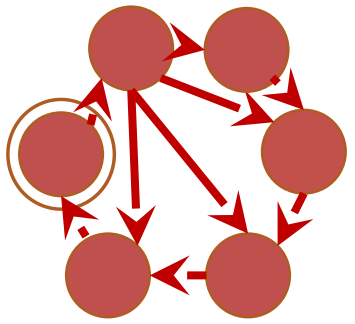
```
datatype ImplVariables = ImplVariables(x:uint64, y:uint64)

method MoveNorthImpl(v:ImplVariables) returns
(v’:ImplVariables)
          ensures MoveNorth(Abstraction(v), Abstraction(v’))
{
          v'.x := v.x;
          v'.y := v.y + 1;
}
```

From Implementation to Protocol

Protocol



Refines



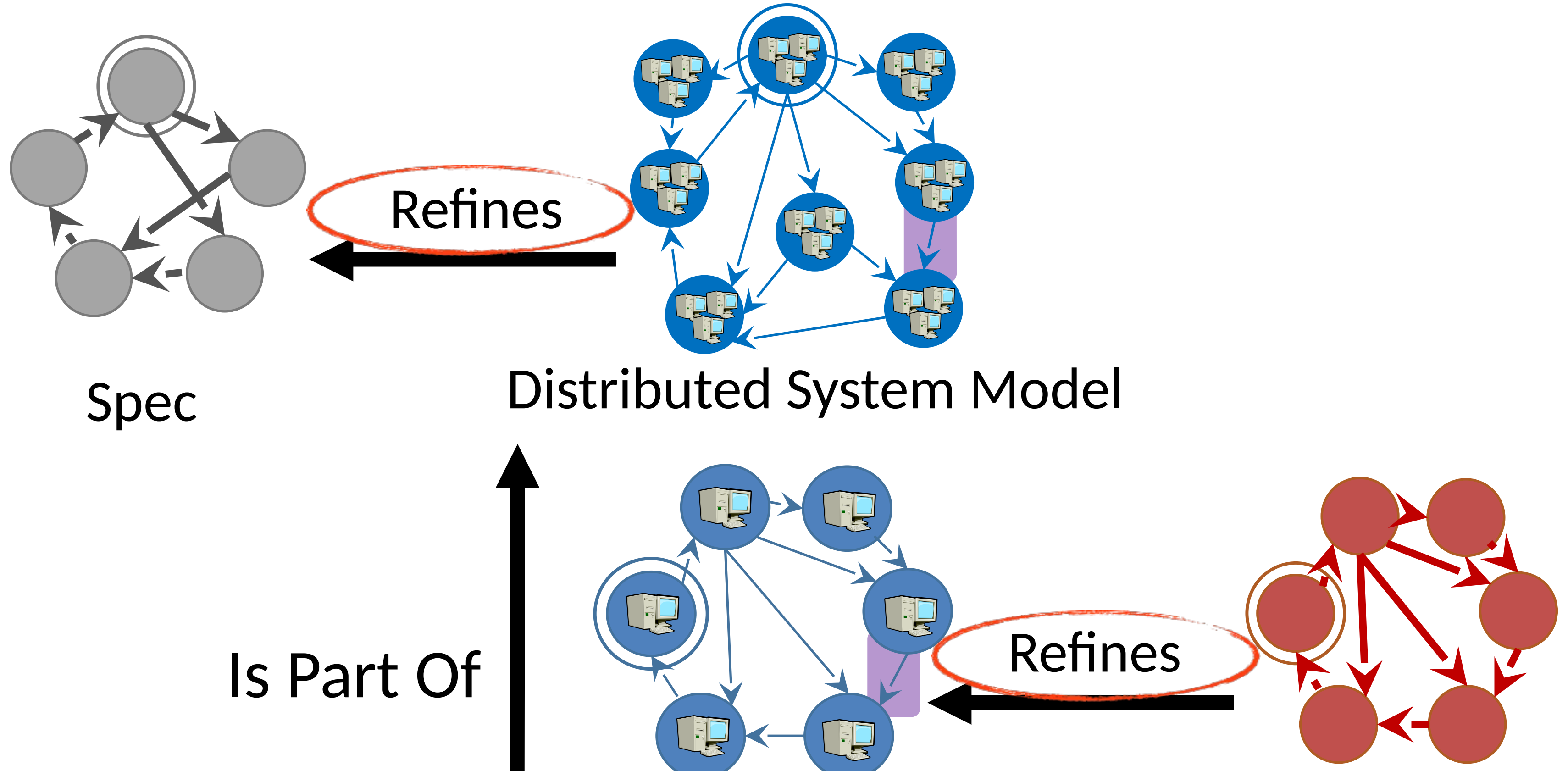
Implementation

```
datatype Variables = Variables(x:int, y:int)

predicate MoveNorth(v:Variables, v':Variables)
{
        v'.x == v.x;
        v'.y == v.y + 1;
}
```

```
function Abstraction(impl:ImplVariables) : Variables
{
        Variables(int(impl.x), int(impl.y))
}
```

```
datatype ImplVariables = ImplVariables(x:uint64, y:uint64)

method MoveNorthImpl(v:ImplVariables) returns
(v’:ImplVariables)
        ensures MoveNorth(Abstraction(v), Abstraction(v’))
                        // or stutter
{
        if(v.y < 0xFFFF_FFFF_FFFF_FFFF) {
                v'.x := v.x;
                v'.y := v.y + 1;
        } else {
                v’ := v;
        }
```

The big picture


Refines
Spec
Distributed System Model
Is Part Of
Refines
Protocol steps (predicates)
Implementation (methods)