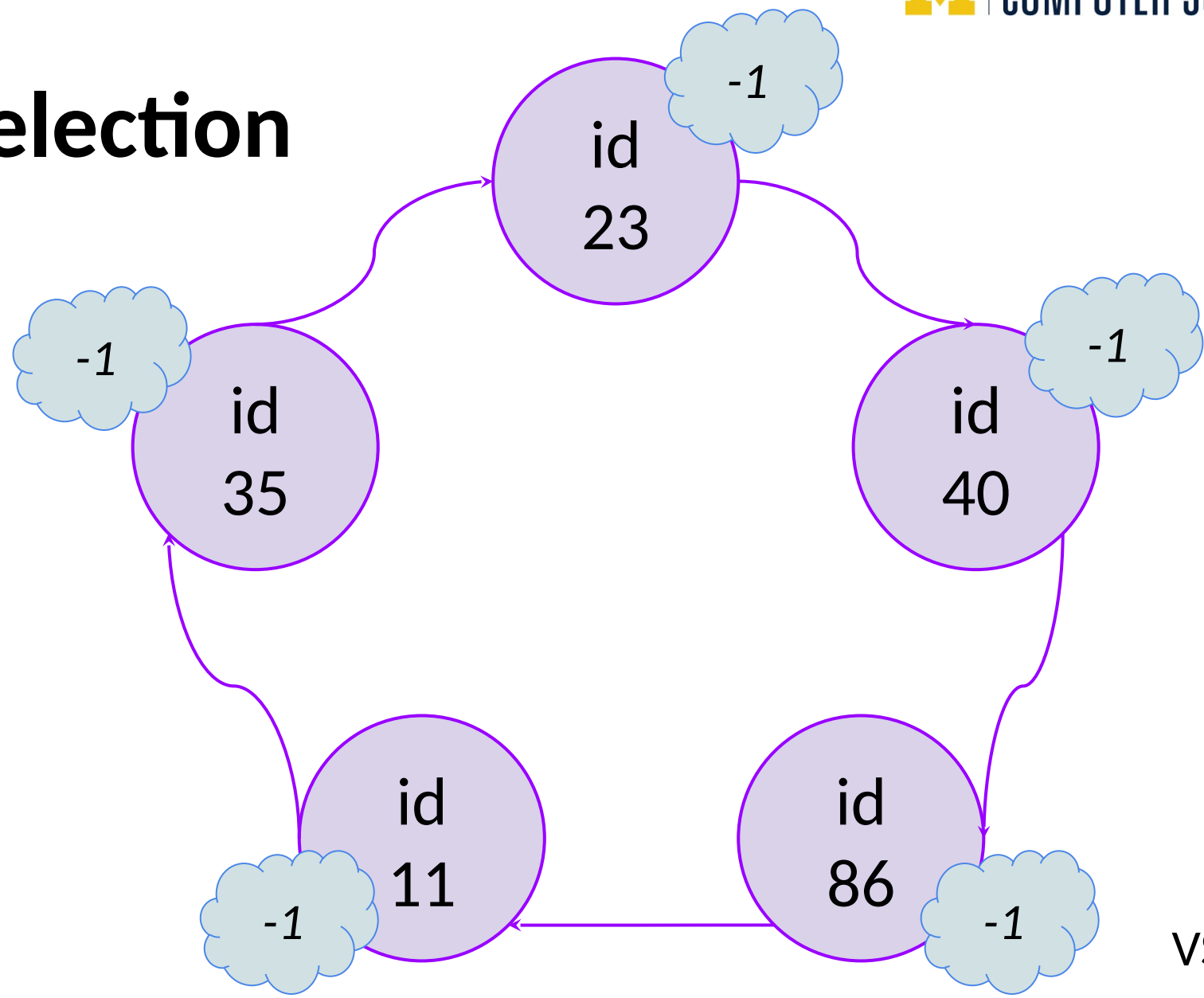


# EECS498-003

# Formal Verification of Systems Software

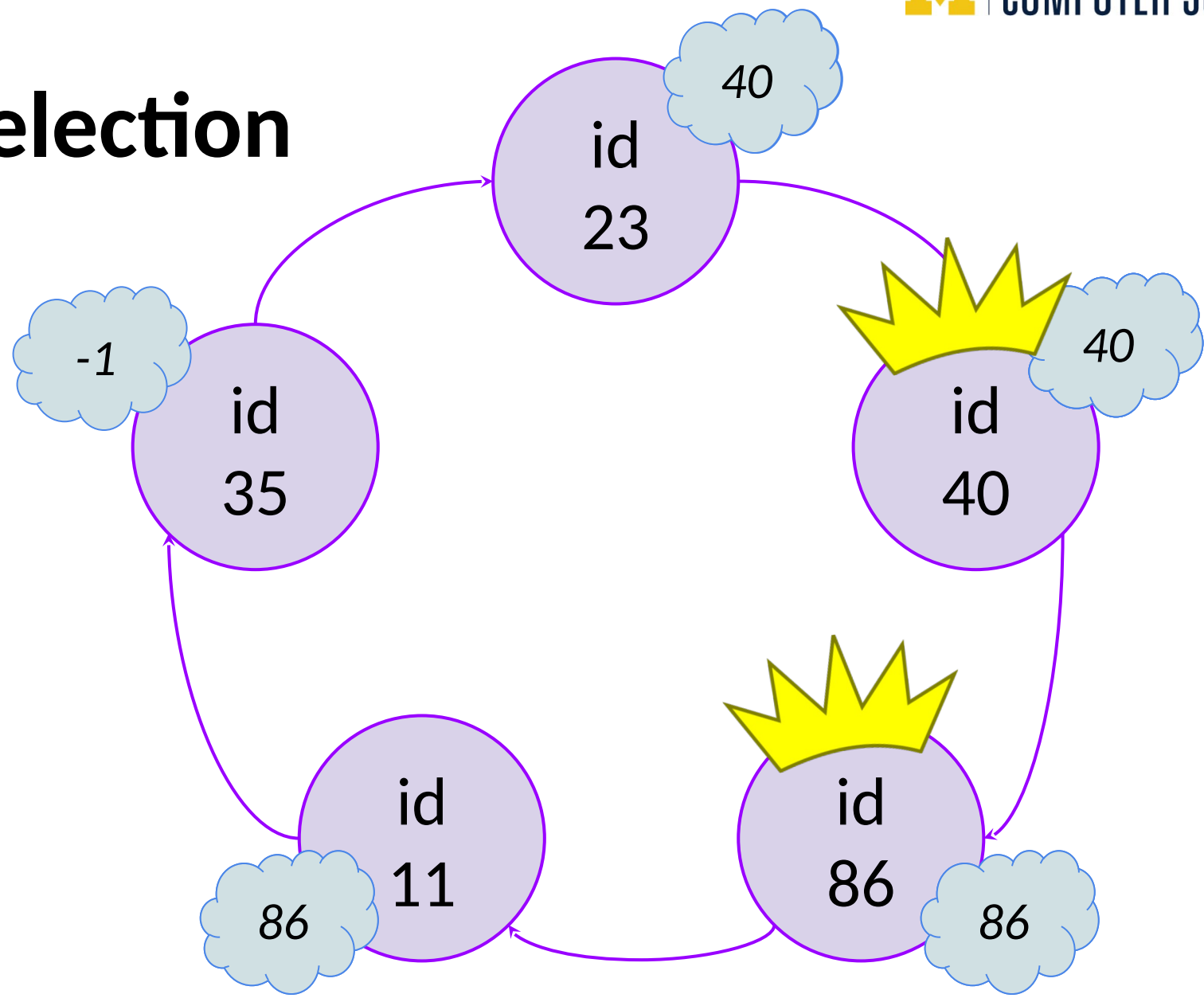
Material and slides created by  
Jon Howell and Manos Kapritsos

# Leader election

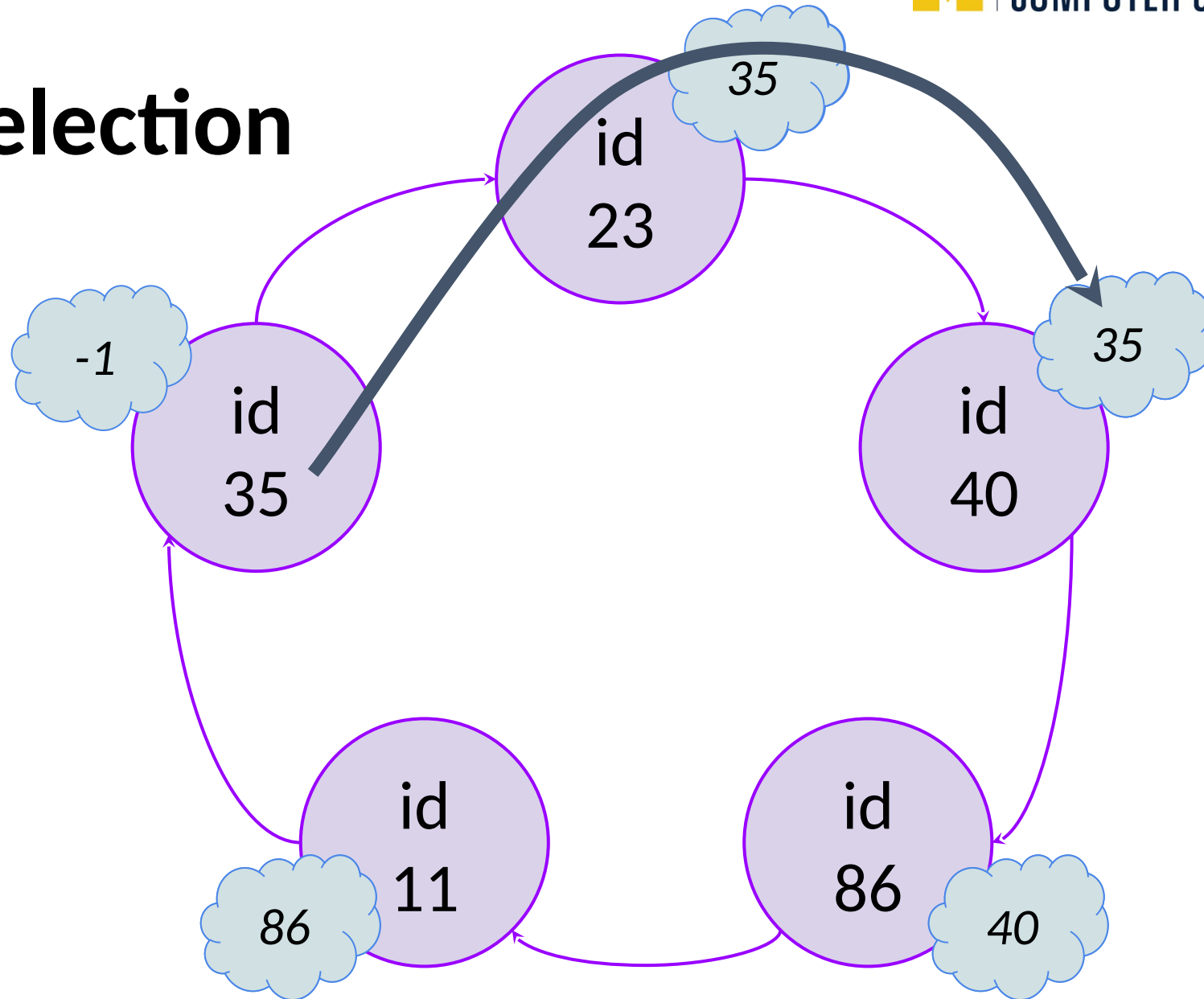


VSCode transition

# Leader election



# Leader election



# Administrivia

- Midterm exam on Thursday, 10/17
  - 6-8pm
  - No lecture that day
  - Sample exams coming soon™
- Closed books
  - Allowed one double-sided cheat-sheet, 10pt minimum
- Covers everything up to Chapter 4 (i.e. excluding distributed systems)
  
- Problem set 2 (Chapters 3 and 4) due **today!**
- Problem set 3 (Chapter 5) will be released tomorrow

# Introduction to distributed systems

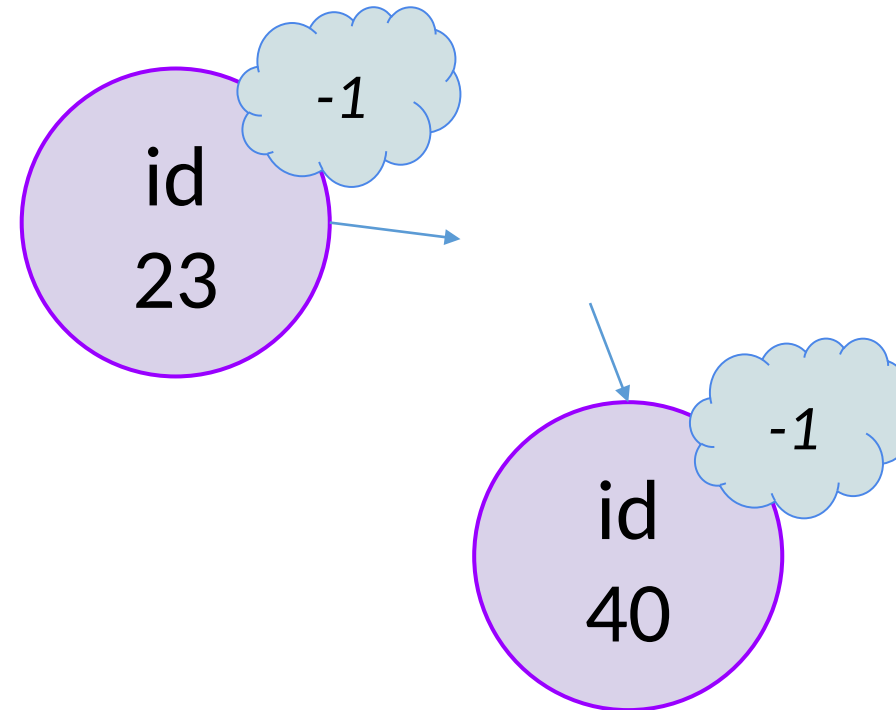
What is a distributed system?

A collection of distinct processes that:

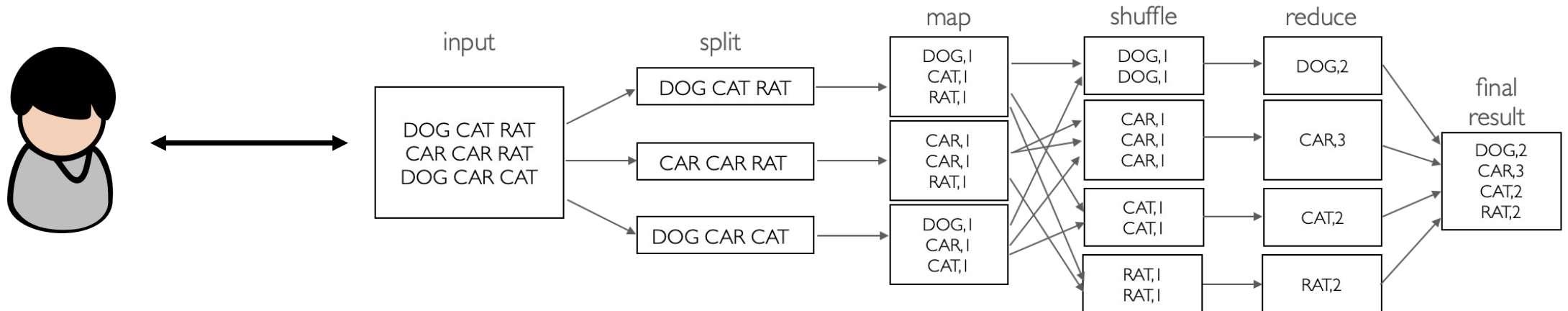
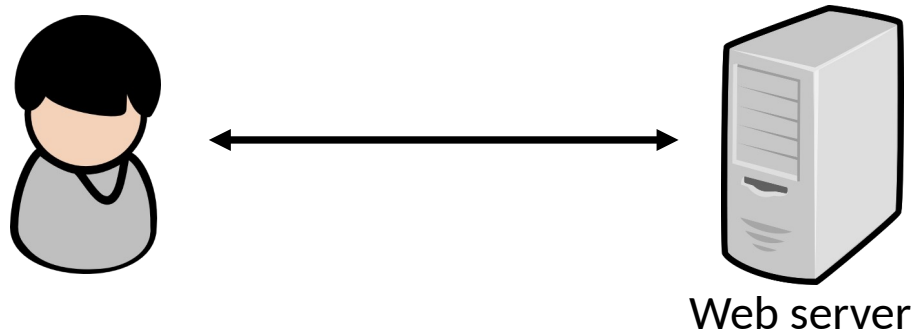
- are spatially separated
- communicate with one another by exchanging messages
- have non-negligible communication delay
- do not share fate
- have separate, imperfect, unsynchronized physical clocks

# Leader election

...as a distributed, **asynchronous** system

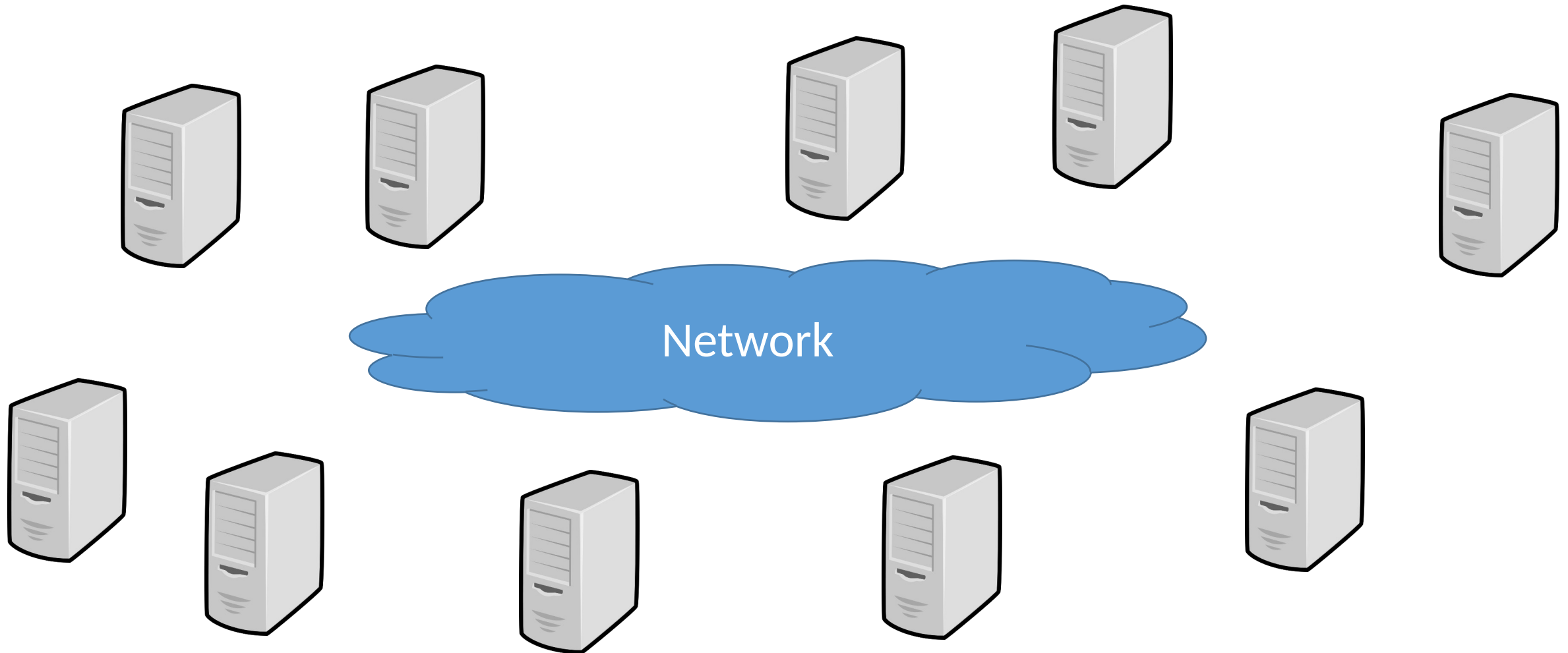


# Other examples of distributed systems



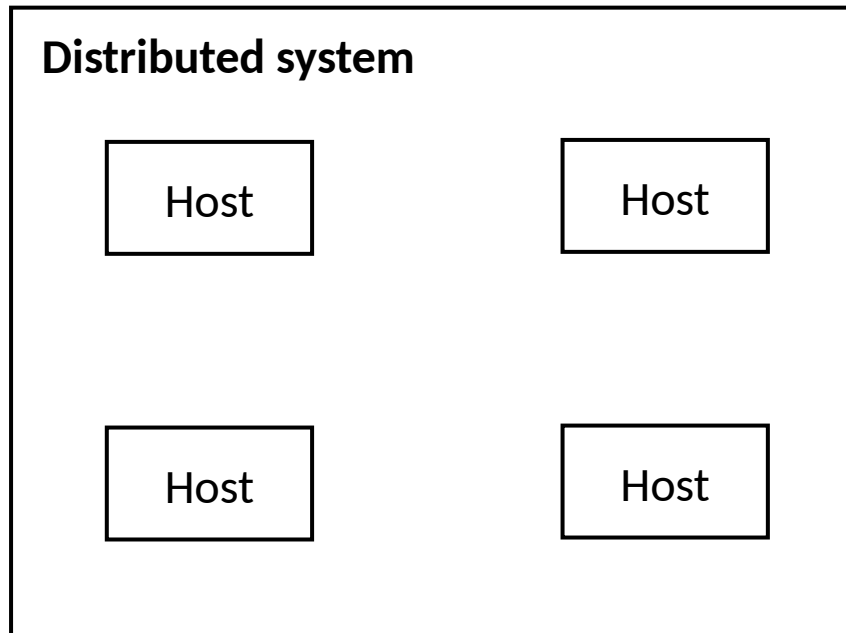


# A distributed system



# Modeling distributed systems

A distributed system is composed of multiple hosts



## Distributed System: attempt #1

```
module DistributedSystem {
  datatype Variables =
    Variables(hosts:seq<Host.Variables>)

  predicate Next (v:Variables, v':Variables, hostid: nat)
  {
    && Host.Next(v.hosts[hostid],v'.hosts[hostid])
    && forall otherHost:nat | otherHost != hostid ::
      v'.hosts[otherHost] == v.hosts[otherHost]
  }
}
```

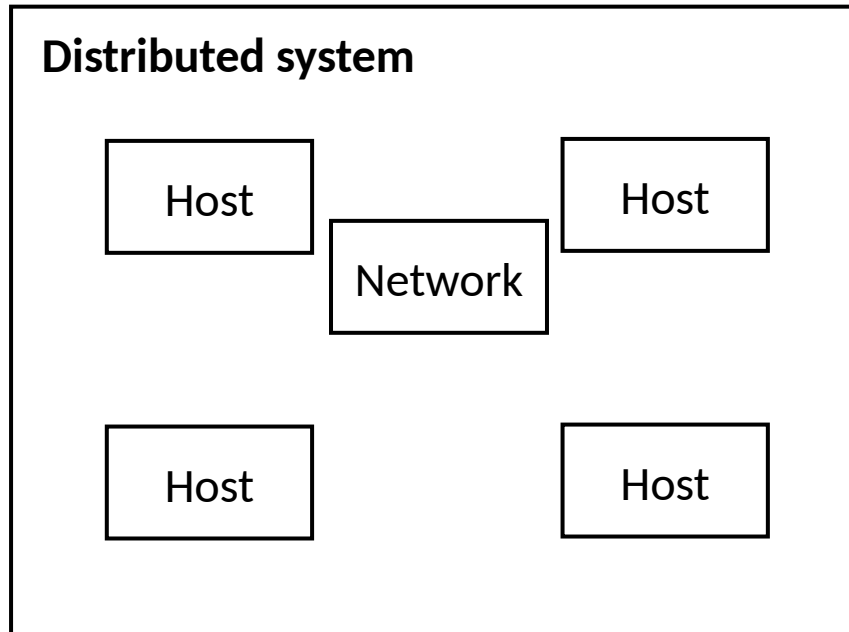
# New Dafny syntax: modules

Modules allow us to break up our code into multiple parts

```
module A {  
    predicate MyPredicate() { ... }  
}
```

```
module B {  
    import A  
    predicate MySecondPredicate() { A.MyPredicate() }  
}
```

# Defining the network



```
datatype Option<T> = Some(value:T) | None
datatype MessageOps = MessageOps(
    recv:Option<Message>,
    send:Option<Message>)
```

## Network module

```
module Network {
    datatype Variables =
        Variables(sentMsgs: set<Message>)

    predicate Next(v, v', msgOps:MessageOps) {
        // can only receive messages that have been sent
        && (msgOps.recv.Some? ==> msgOps.recv.value in
            v.sentMsgs)
        // Record the sent message, if there was one
        && v'.sentMsgs ==
            v.sentMsgs + if msgOps.send.None? then {}
                        else {msgOps.send.value}
    }
}
```

# A distributed system is composed of multiple hosts and a network

Distributed system: attempt #2

```

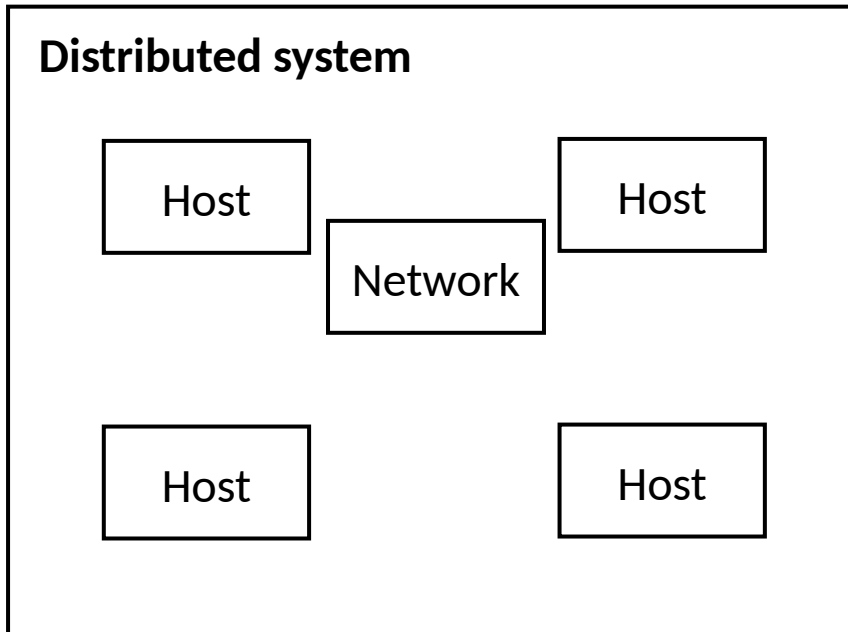
module DistributedSystem {
  datatype Variables =
    Variables(hosts:seq<Host.Variables>,
              network: Network.Variables)

  predicate HostAction(v, v', hostid, msgOps) {
    && Host.Next(v.hosts[hostid],v'.hosts[hostid],msgOps)
    && forall otherHost:nat | otherHost != hostid ::
      v'.hosts[otherHost] == v.hosts[otherHost]
  }

  predicate Next(v, v', hostid, msgOps: MessageOps) {
    && HostAction(v, v', hostid, msgOps)
    && Network.Next(v, v', msgOps)
  }
}

```

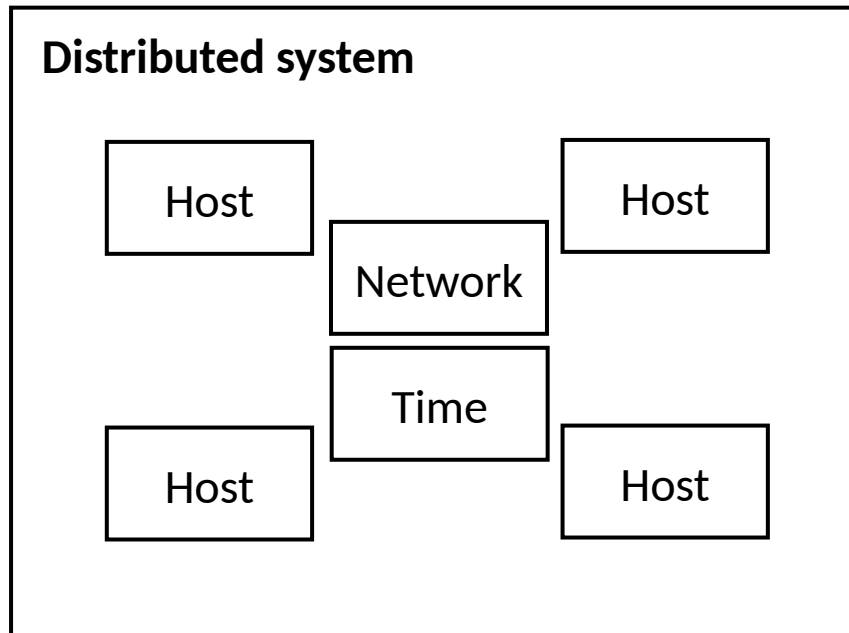
Binding variable



# A distributed system is composed of multiple hosts, a network and clocks

Distributed system: attempt #3

```
module DistributedSystem {
  datatype Variables =
    Variables(hosts:seq<Host.Variables>,
              network: Network.Variables,
              time: Time.Variables)
```



```
predicate Next(v, v', hostid, msgOps: MessageOps,
              clk:Time) {
  || (&& HostAction(v, v', hostid, msgOps, clk)
      && Network.Next(v, v', msgOps)
      && Time.Read(v.time, clk))
  || (&& Time.Advance(v.time, v'.time)
      && v'.hosts == v.hosts
      && v'.network == v.network)
}
```

Binding variable

# Administrivia

- Problem set 3 (Chapter 5) will be released later today
- Start looking for partners for Project 1 (released after PS3)
- Midterm evaluations are up
  - Please provide feedback!
  - Note the additional questions

# Atomic Commit (Problem Set 3)



-Do you take each other?  
-I do.  
-I do.  
-I now pronounce you  
atomically committed.



# Atomic Commit: the objective

Preserve data consistency for distributed transactions

Example: book a hotel and flight on Expedia

# Atomic Commit: the setup

- One coordinator
- A set of participants
  - Allowed to be empty in our model
- Every participant has an “input” value, called **vote/preference**  
 $vote_i \in \{Yes, No\}$
- Every participant/coordinator has an “output” value, called **decision**  
 $decision_i \in \{Commit, Abort\}$
- We are ignoring the possibility of failures

# Atomic Commit: the spec (simplified to ignore failures)

- AC-1: All processes that reach a decision reach the same one
- AC-3: The **Commit** decision can only be reached if all processes vote **Yes**
- AC-4: If ~~there are no failures and~~ all processes vote **Yes**, then the decision must be **Commit**

AC-2 and AC-5 ignored

# Two Phase Commit (2PC)

Coordinator  $c$

Participant  $p_i$

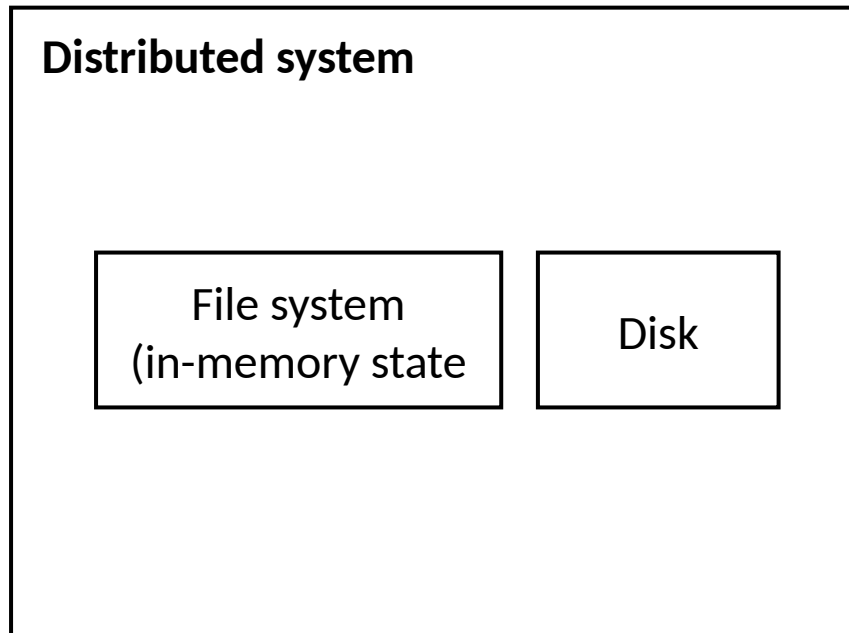
1. sends **VOTE-REQ** message to all participants

2. sends  $vote_i$  to coordinator  
if  $vote_i == \text{No}$   
then  $decision_i := \text{Abort}$

3. Wait for all votes to come in  
If all votes are **Yes** then  
 $decision_c := \text{Commit}$   
Send **Commit** message to all  
else  
 $decision_c := \text{Abort}$   
Send **Abort** message to all who voted **Yes**

4. If received **Commit** then  
 $decision_i := \text{Commit}$   
else  
 $decision_i := \text{Abort}$

# A “distributed” system



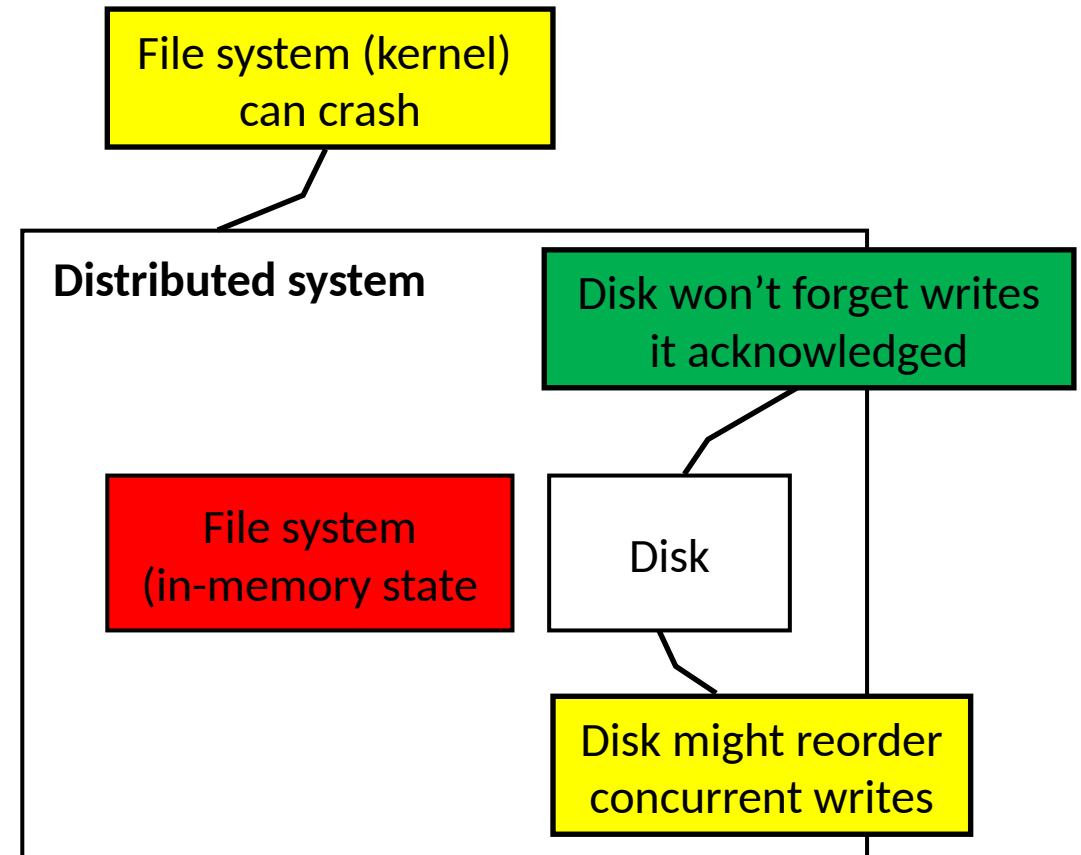
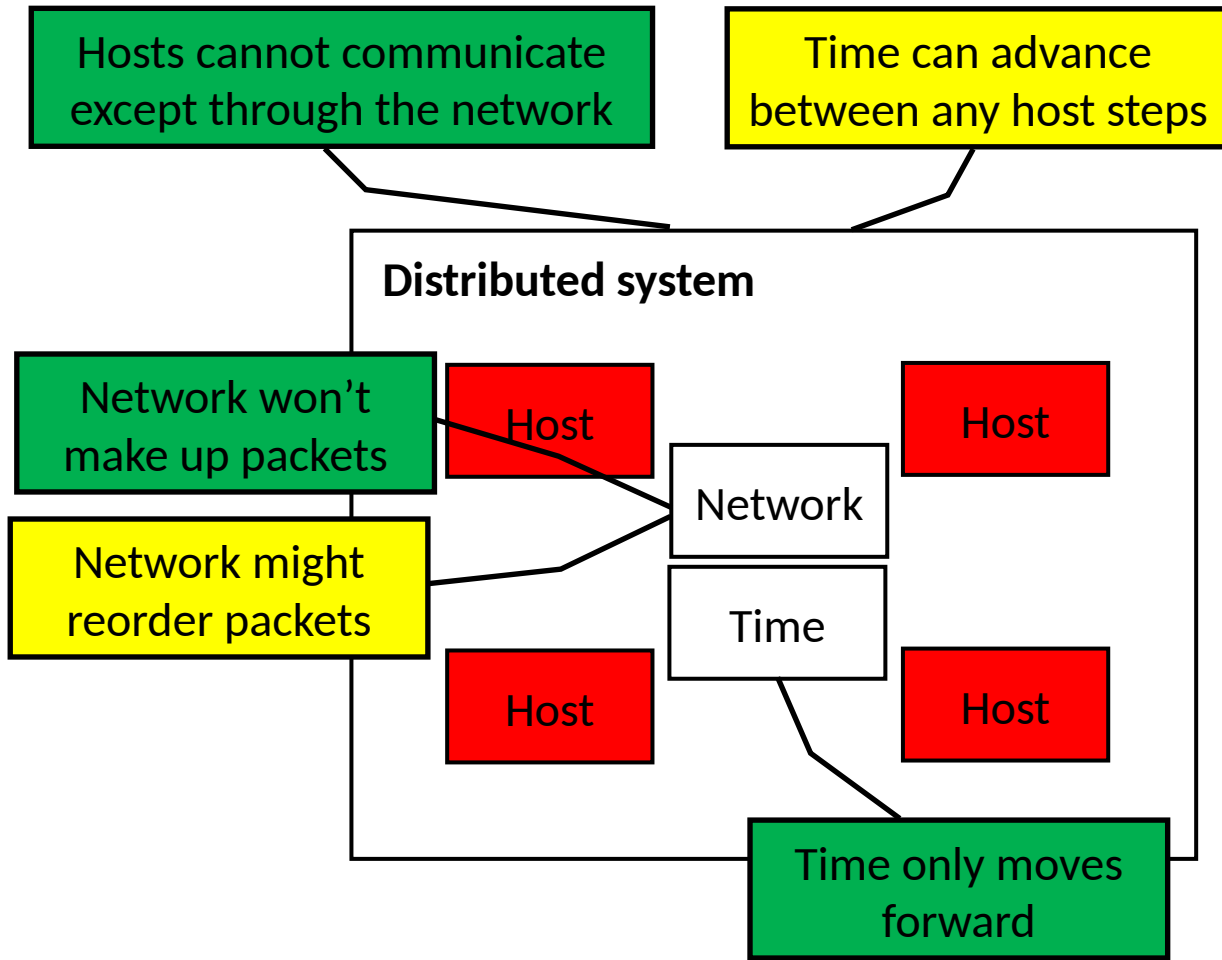
```

module DistributedSystem {
  datatype Variables =
    Variables(fs: FileSystem.Variables,
             disk: Disk.Variables)

  predicate Next(v, v') {
    || (exists io ::
        && FileSystem.Next(v.fs, v'.fs, io)
        && Disk.Next(v.disk, v'.disk, io) Binding variable
    || ( // Crash!
        && FileSystem.Init(v'.fs)
        && v'.disk == v.disk
    )
  }
}

```

# Trusted vs proven



# **SPECIFICATION** →: the systems specification sandwich



image: pixabay

