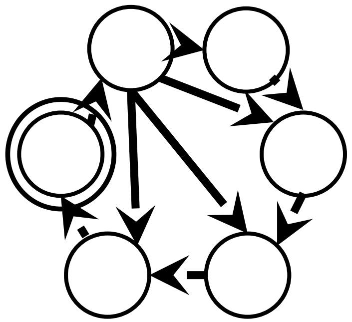
EECS498-003
Formal Verification of Systems Software

Material and slides created by

Jon Howell and Manos Kapritsos

Chapter 6: Refinement

State machines: a versatile tool

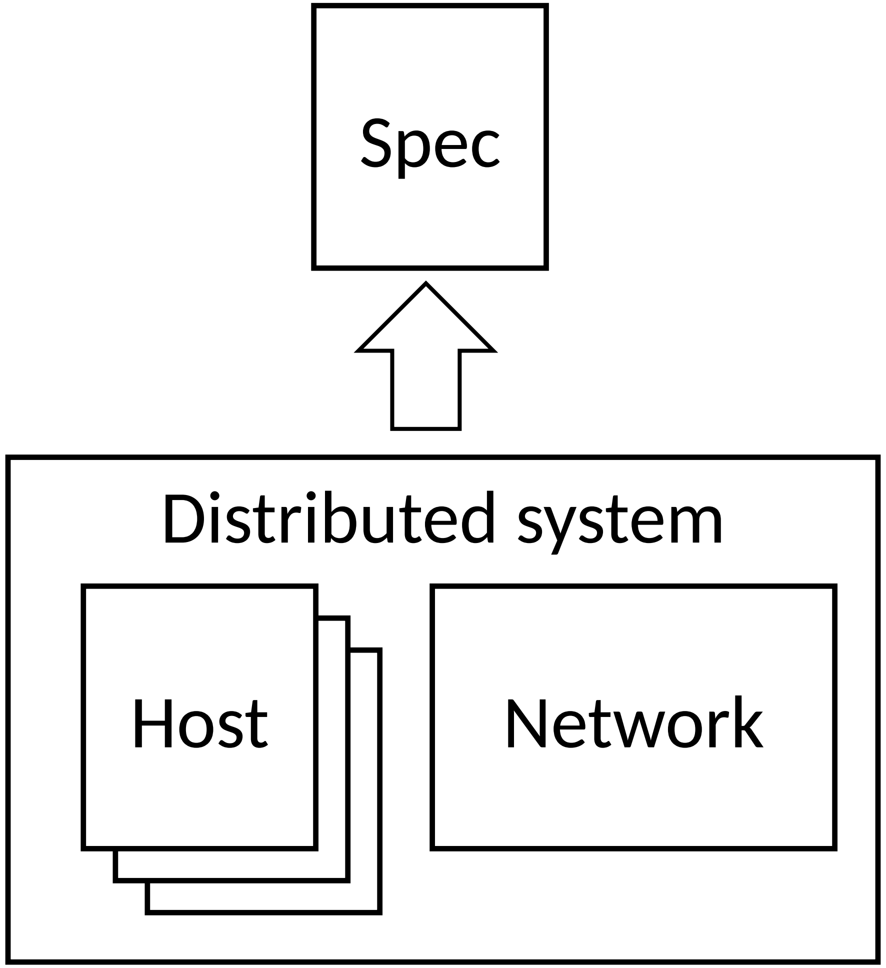


State machines can be used to

- Model the program
- Model environment components
- Model how the system (program+environment) fits together
- Specify the system behavior

Different ways to specify behavior

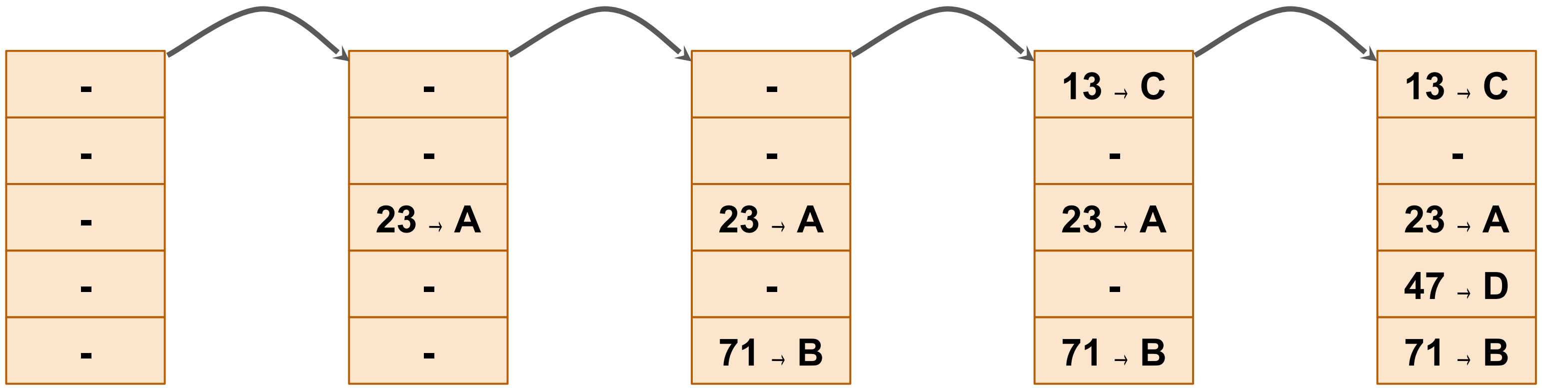
- C-style assertions
- Postconditions
- Properties/invariants
- Refinement to a state machine



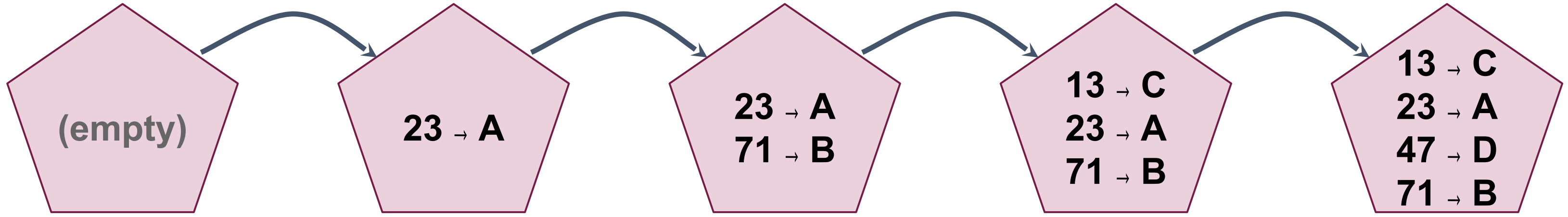
Example: hashtable

```
module HashTable {
  datatype Variables = Variables(tbl:seq<Pair<int, string>>)

  predicate Insert(v:Variables, v':Variables, key:int,
val:string) {
    var free := Probe(v.tbl, key);
    && free.Some?
    && v'.tbl == v.tbl[free.value := Pair(key, val)]
  }
}
```



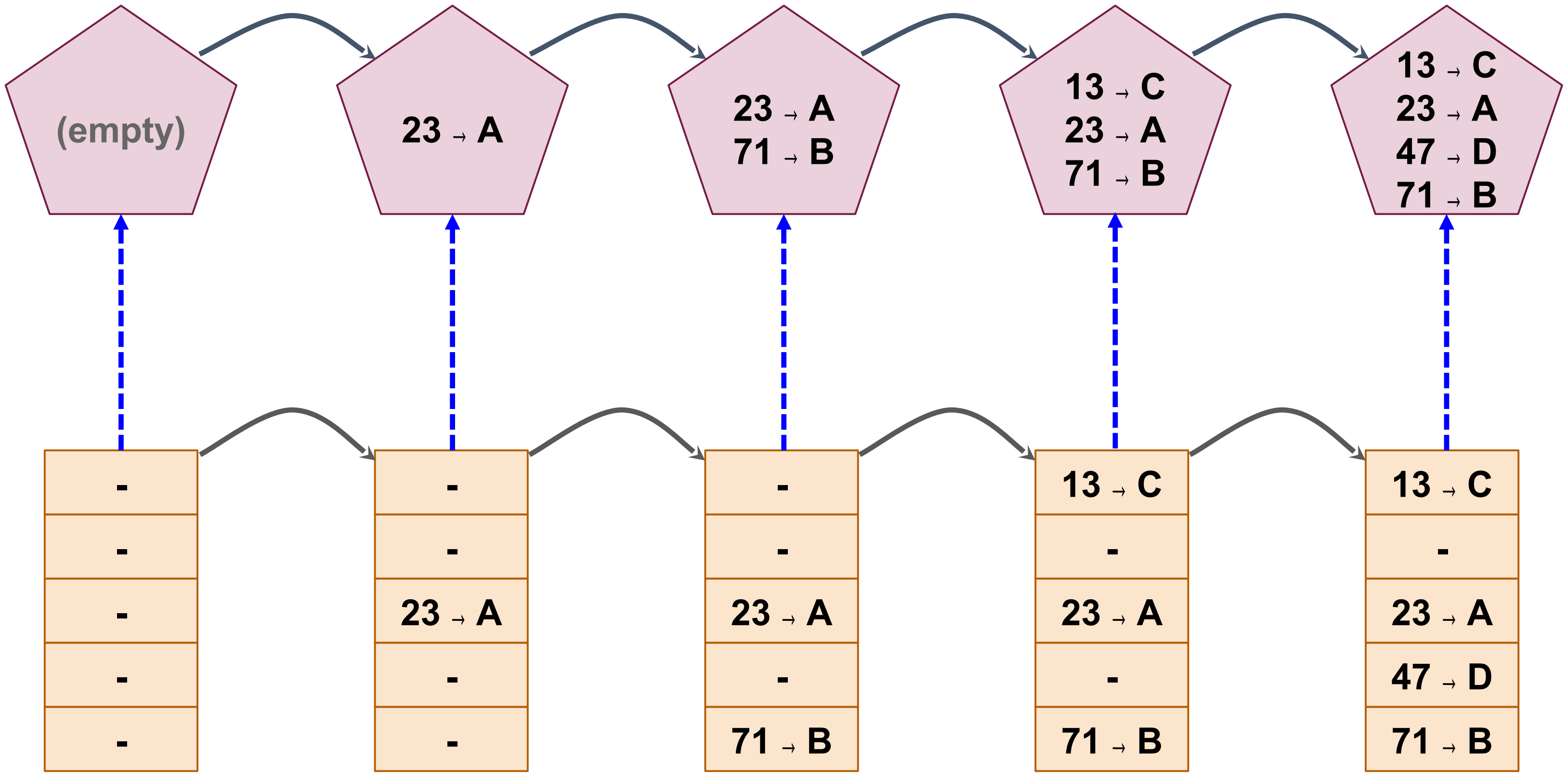
The spec: a simple map




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

  predicate InsertOp(v:Variables, v':Variables, key:Key,
value:Value) {
    && v'.mapp == v.mapp[key := value]
  }
}
```

Refinement



(empty)
23 → A
23 → A
71 → B
13 → C
23 → A
71 → B
13 → C
23 → A
47 → D
71 → B
-
-
-
-
-
-
-
23 → A
-
-
-
-
23 → A
-
71 → B
13 → C
-
23 → A
-
71 → B
13 → C
-
23 → A
47 → D
71 → B

The benefits of refinement

Refinement allows for good specs

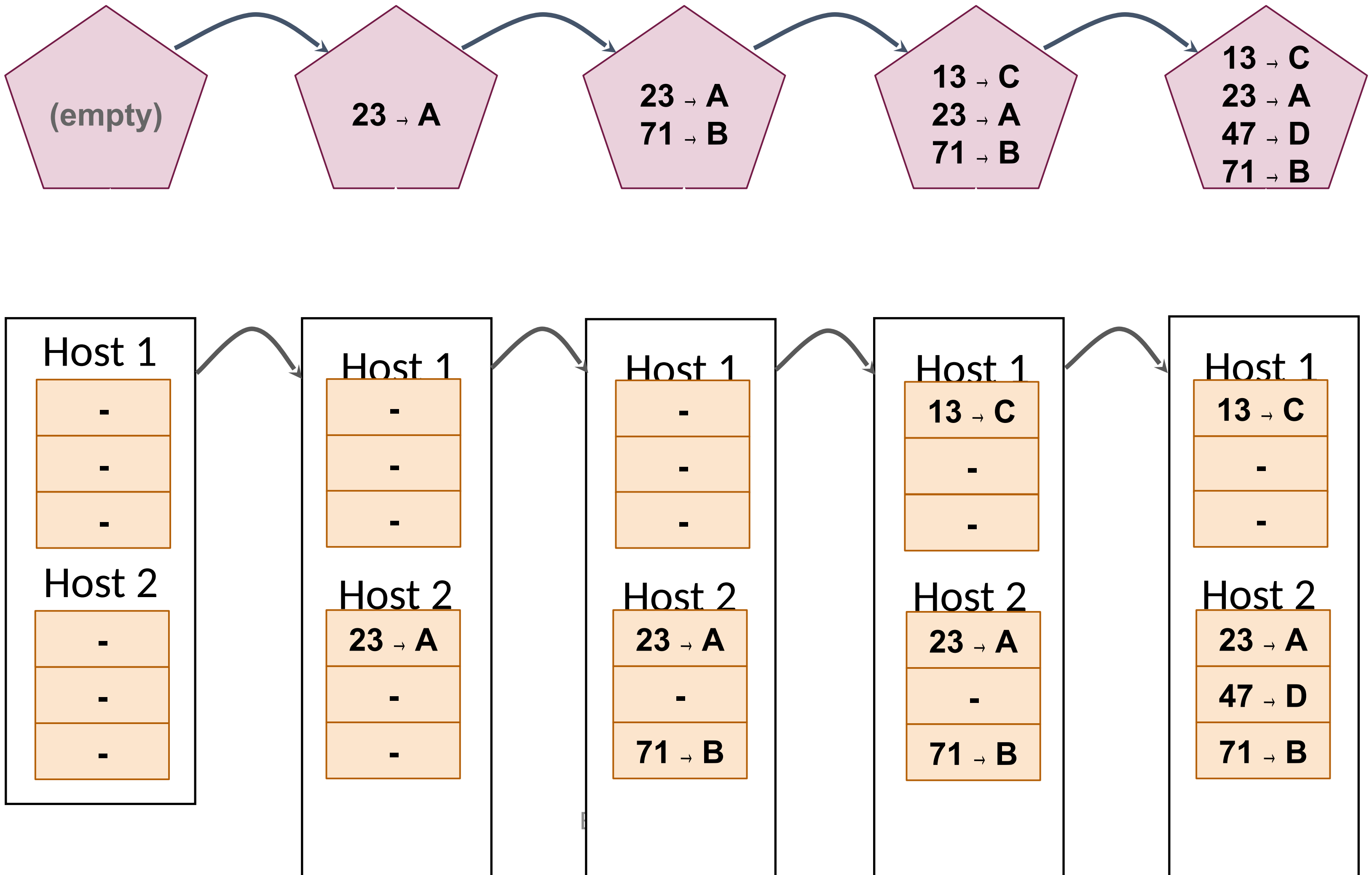
- Abstract: elide implementation details
- Concise: simple state machine
- Complete: better than a “bag of properties”
  - But if you want, you can prove properties about the spec

Refinement is very powerful

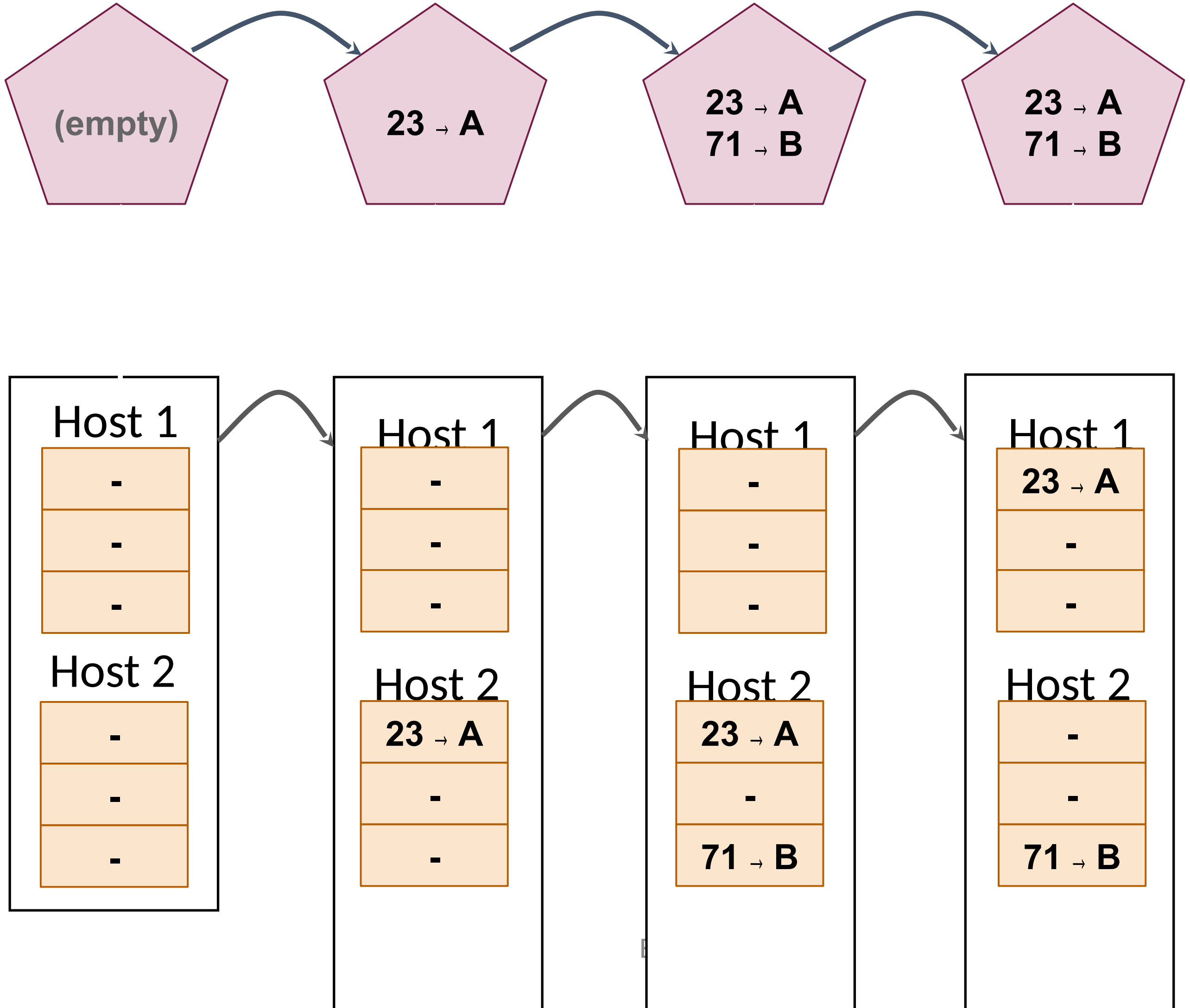
- Can specify systems that are hard to specify otherwise
  - E.g. linearizability

A sharded key-value store

Logically centralized, physically distributed



Stutter steps



One “stutter” step for the spec

One normal step for the implementation

Midterm exam

- Well done! Midterm stats:
  - Median: 72
  - Std dev: 23.5
  - Passing grade: 36.75
    - Your *average* exam score must be above the *average* passing grade
- Review session will be held this week during this week's lab
  - Last chance to close gaps in your understandings

Regrade requests

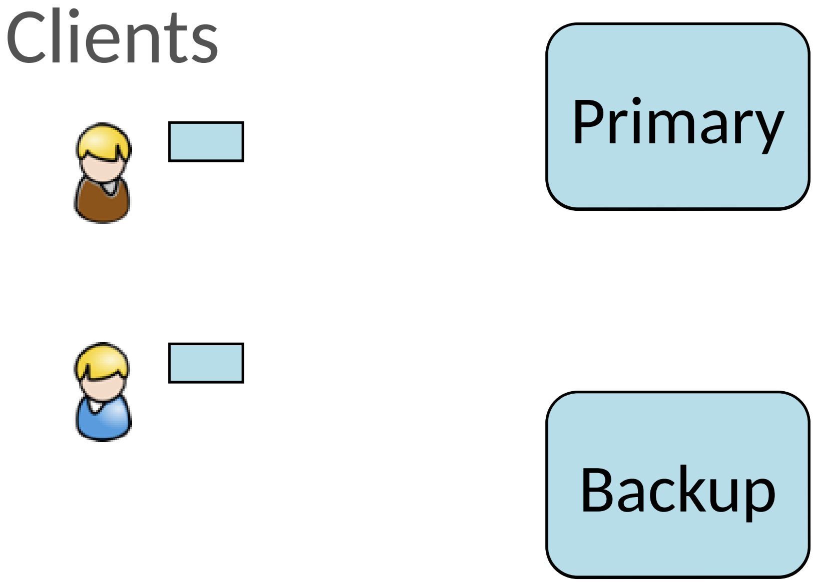
- Regrade requests will open after the review session
  - They will stay open for a week
- Submit **clear** reasoning for why you think your answer is correct
- We will optionally re-grade the entire question or exam
  - Your grade may go up or down as a result

Administrivia

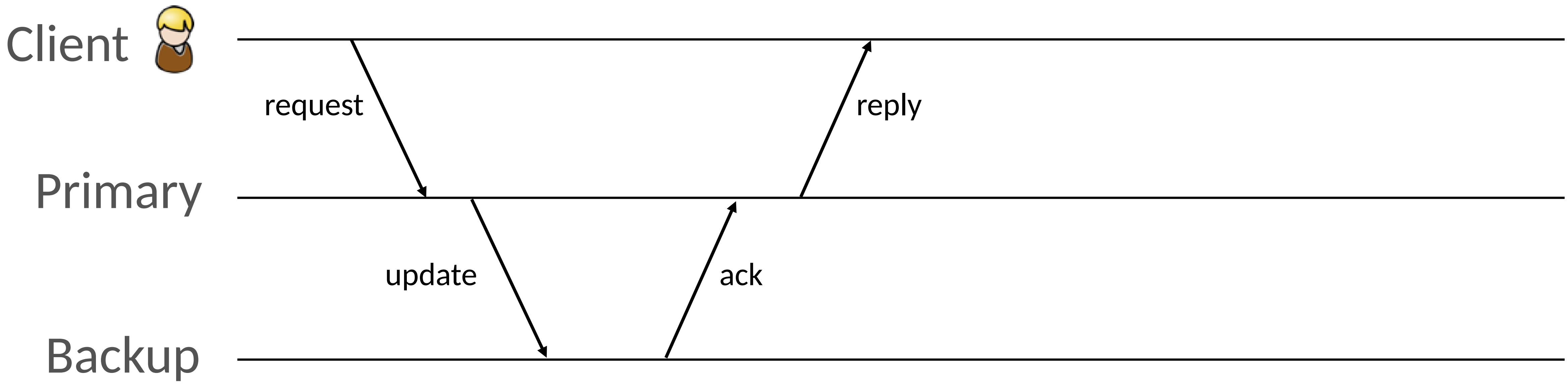
- No class on Tuesday, Nov 5
  - Travel for me, vote for you
- No class on Tuesday, Nov 12
  - Just travel for me

- PS3 due this Thursday, Oct 24
- Project 1 released Friday, Oct 25

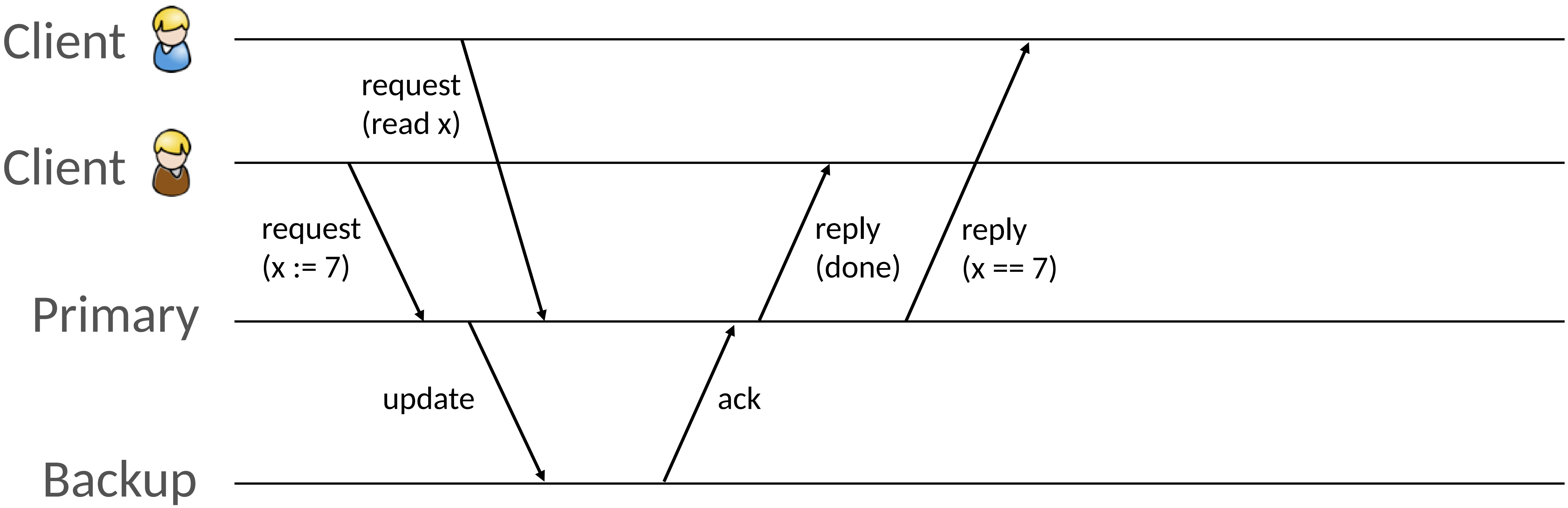
A primary-backup protocol



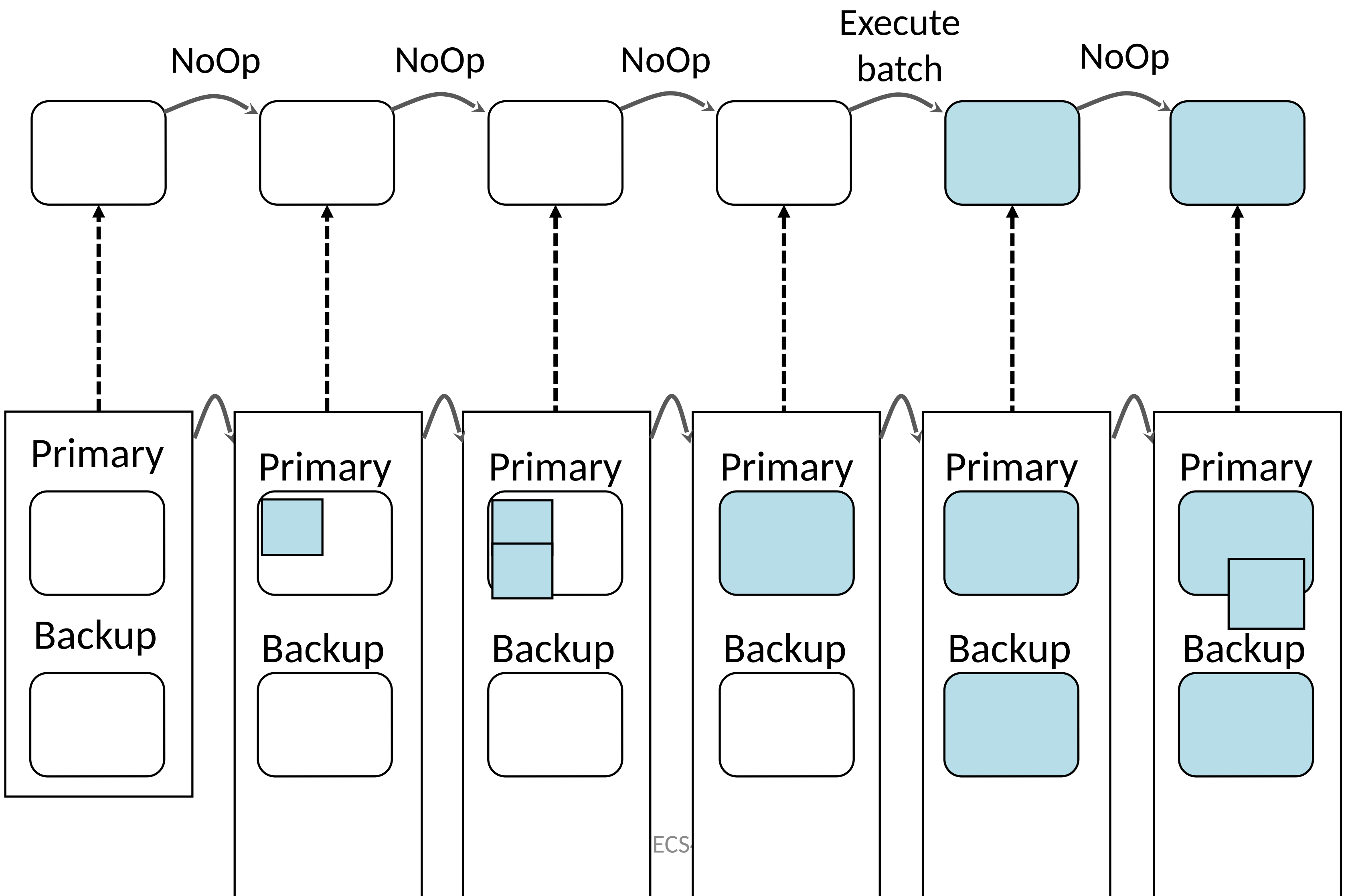
A primary-backup protocol



A primary-backup protocol



A primary-backup protocol

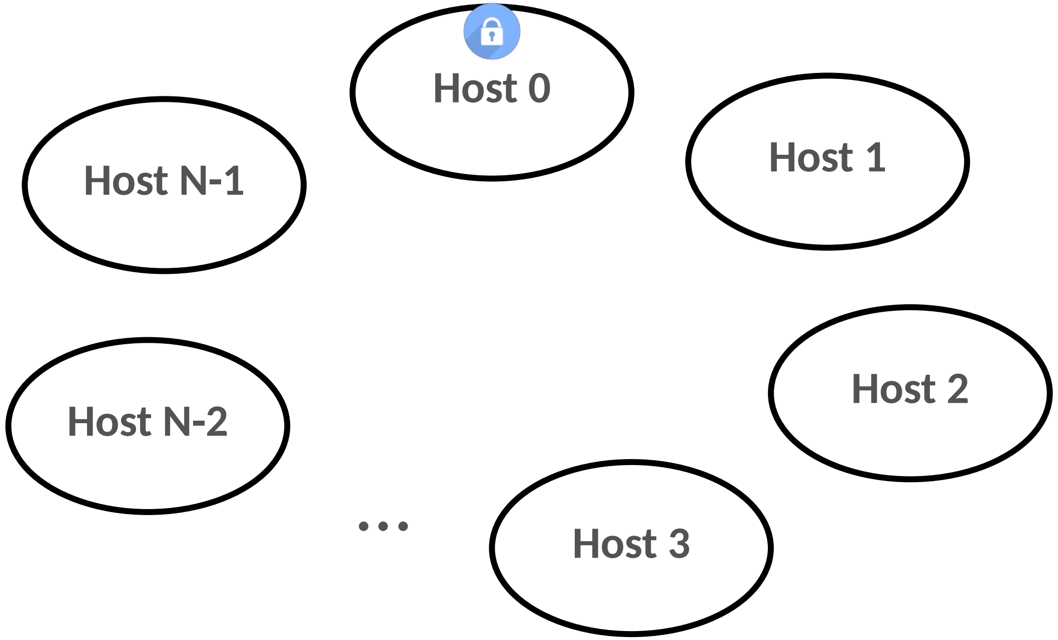


Project 1: Distributed lock service

Differences from centralized lock server

- **No centralized server** that coordinates who holds the lock
  - The hosts pass the lock amongst themselves
- The hosts communicate via **asynchronous messages**
  - A single state machine transition **cannot** read/update the state of two hosts

Distributed lock server



- N = numHosts, defined in network.t.dfy
- Messages are asynchronous (i.e. sending and receiving are two separate steps)

Distributed lock server

The lock is associated with a monotonically increasing epoch number

Accept an incoming message only if it has a higher epoch number than your current epoch

Distributed lock server

Safety property:

The desirable property is the same as the centralized lock server: at most one node holds the lock at any given time

Project files

Framework files
(trusted/immutable)

network.t.dfy

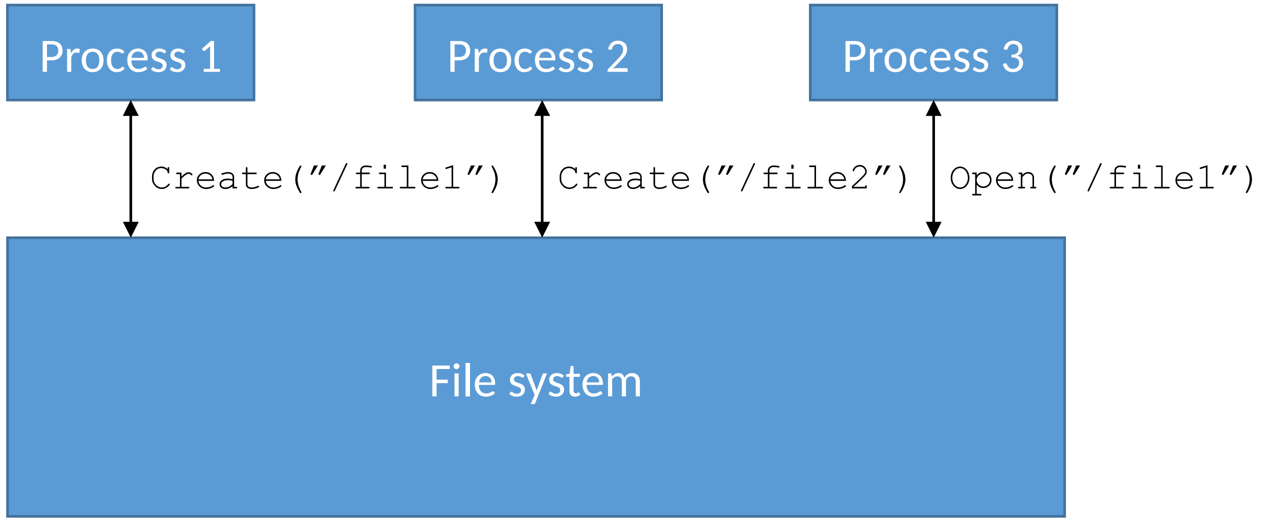
distributed_system.t.dfy

Host and proof files
(for you to complete)

host.v.dfy

exercise01.dfy

World-visible events



Which of these behaviors are correct?
(assuming an initially empty file system)

Behavior #1

```
Create(f, "/file1")     (returns OK)
Create(f, "/file2")     (returns OK)
Create(d, "/dir")       (returns OK)
Create(f, "/dir/file1") (returns OK)
```

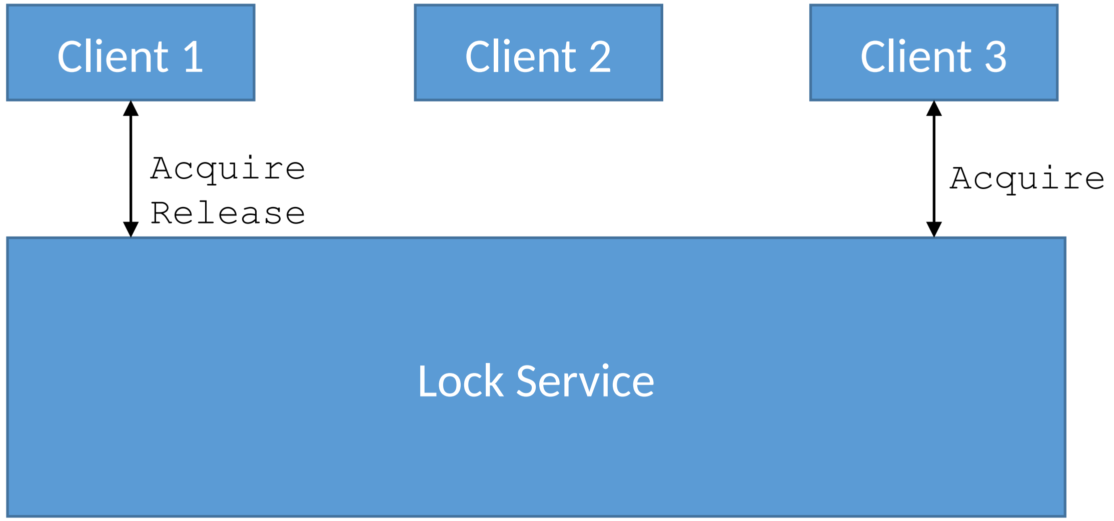
Behavior #2

```
Create(f, "/file1")     (returns OK)
Create(f, "/file2")     (returns OK)
Create(f, "/dir/file1") (returns Err)
```

Behavior #3

```
Create(f, "/file1")     (returns OK)
Write(f, "/file2")      (returns OK)
Create(d, "/dir")       (returns OK)
Create(f, "/dir/file1") (returns OK)
```

World-visible events



Which of these behaviors are correct?
(assuming no one holds the lock initially)

Behavior #1

```
Acquire(client1)
Acquire(client1)
Release(client1)
Release(client1)
```

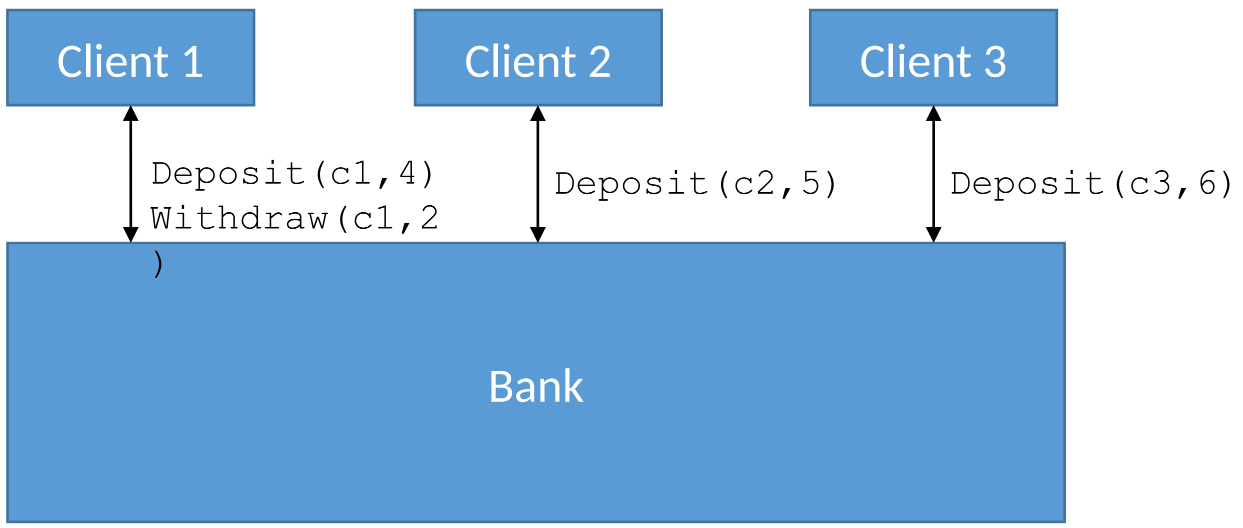
Behavior #2

```
Release(client2)
Acquire(client1)
Release(client1)
```

Behavior #3

```
Acquire(client1)
Release(client1)
Acquire(client2)
```

World-visible events



Which of these behaviors are correct?

(assuming all account are initially empty)

Behavior #1

```
Deposit(client1, 6)     (returns OK)
Withdraw(client1, 3)    (returns OK)
Withdraw(client1, 2)    (returns OK)
Deposit(client1, 3)     (returns Err)
```

Behavior #2

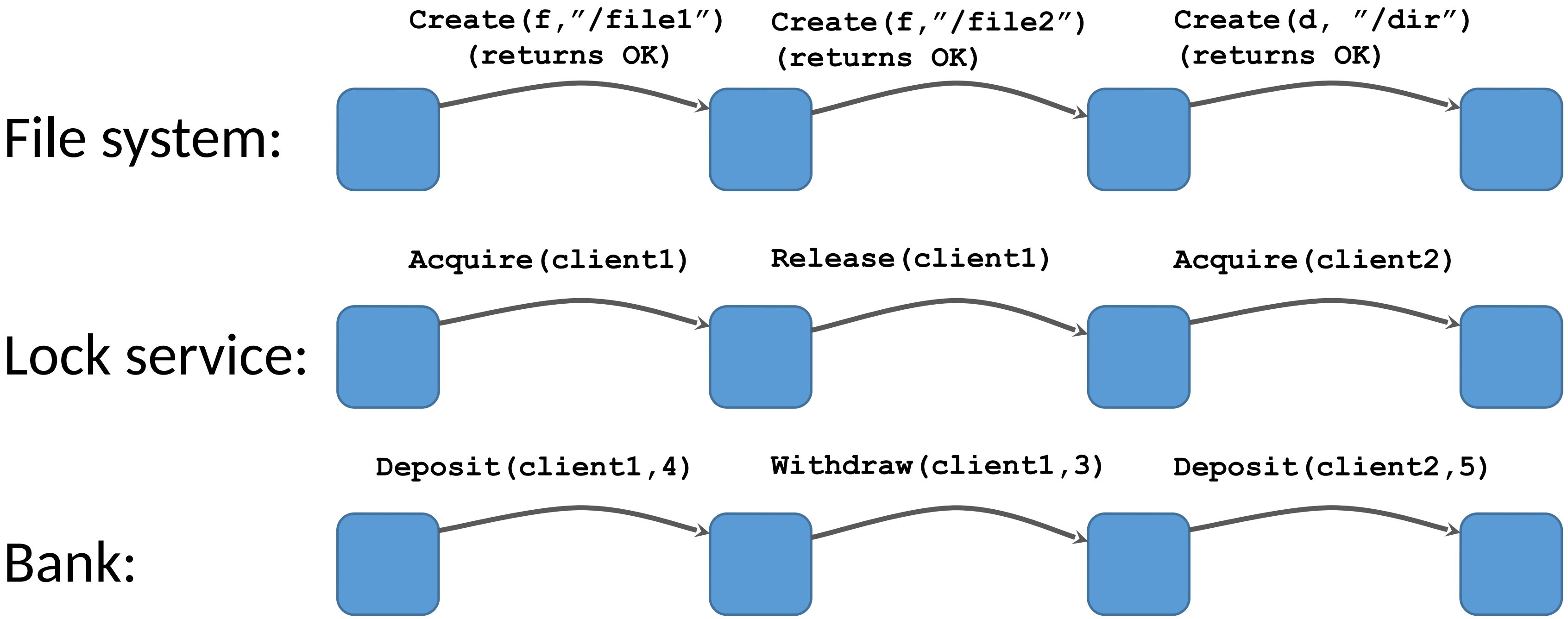
```
Deposit(client1, 6)     (returns OK)
Withdraw(client1, 3)    (returns OK)
Withdraw(client2, 2)    (returns OK)
```

Behavior #3

```
Deposit(client1, 6)     (returns OK)
Withdraw(client1, 3)    (returns OK)
Withdraw(client1, 2)    (returns OK)
Withdraw(client1, 3)     (returns Err)
```

Events define correctness

One should be able to evaluate the correctness of the system by inspecting a behavior (sequence) consisting of world-visible events



Event-enriched state machines

We will be adding events to our spec state machines

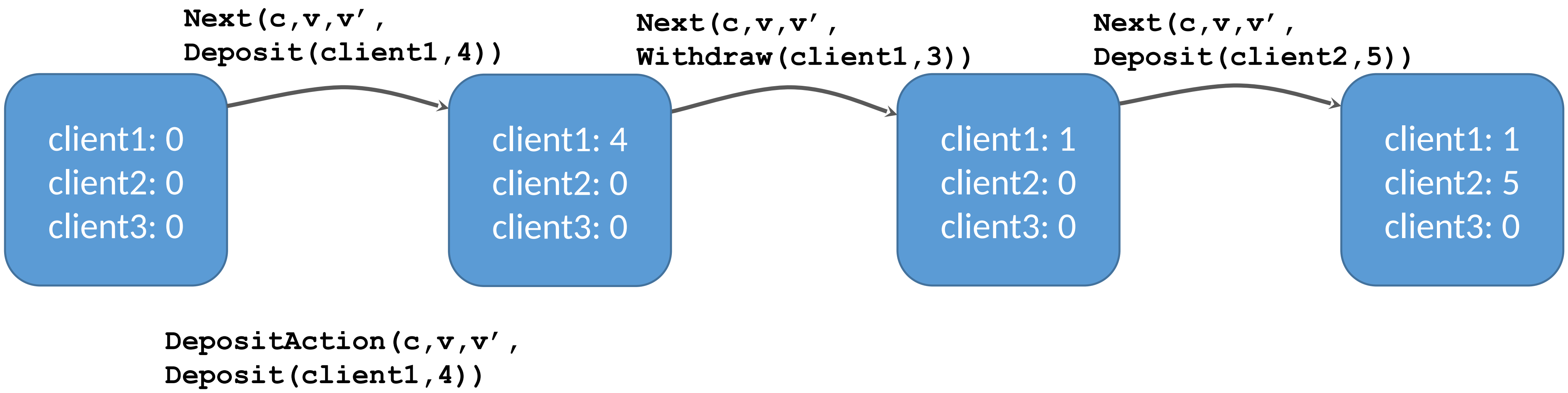
For example, the lock service would use this Event datatype:

```
datatype Event = Acquire(clientId:nat) | Release(clientId:nat) | NoOp
```

The Next() transition will now be parameterized by an Event:

```
ghost predicate Next(c: Constants, v: Variables, v': Variables, evt: Event)
```

Example: Bank spec state machine



Event-enriched state machines

We will **also** be adding events to our protocol state machines

Using the exact same type as the spec state machine uses

E.g. for lock service

```
datatype Event = Acquire(clientId:nat | Release(clientId:nat) | NoOp
```

The Next() transition of both Host and DistributedSystem will now be parameterized by an Event:

```
ghost predicate Next(c: Constants, v: Variables, v': Variables, evt: Event)
```

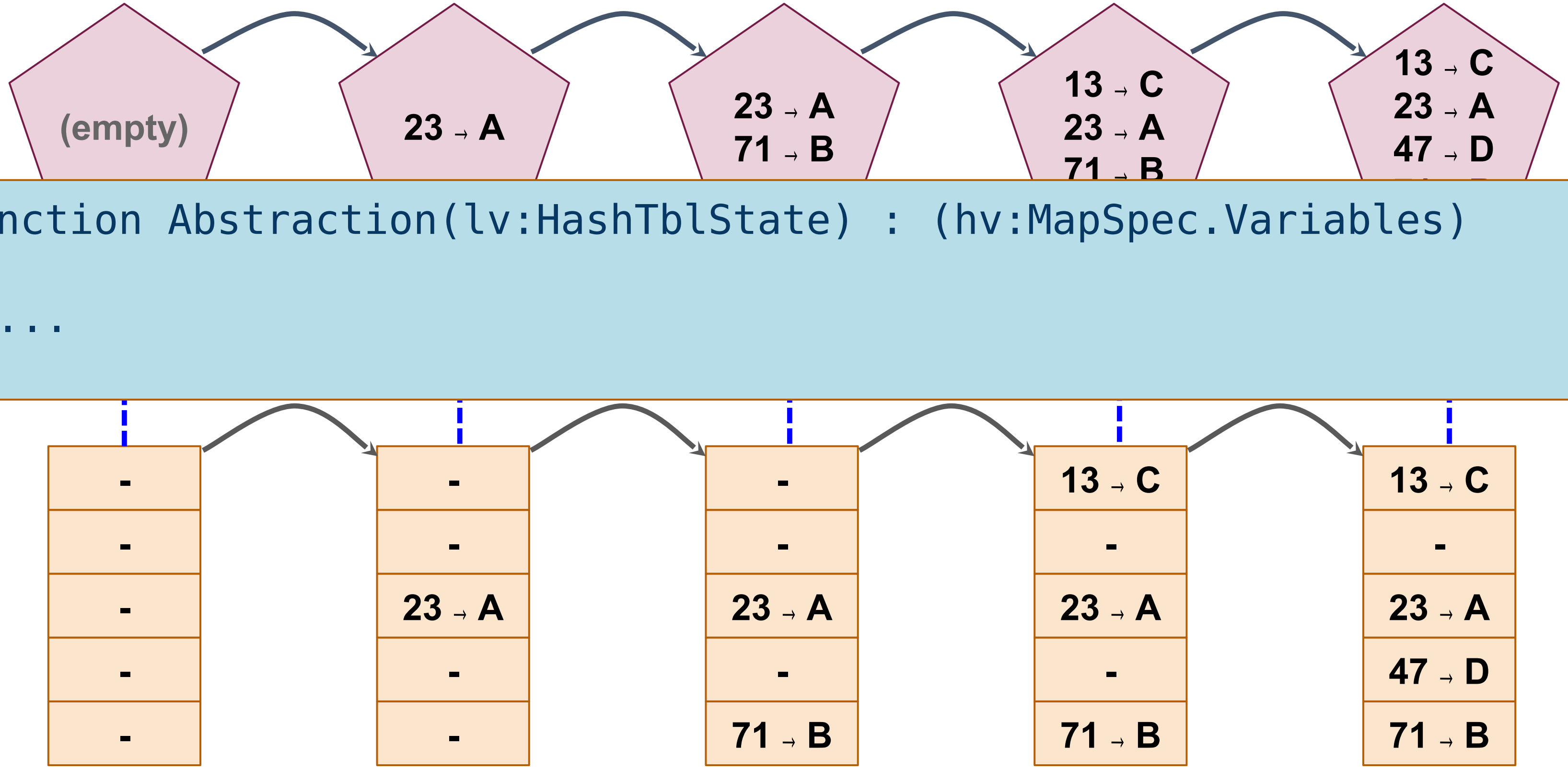
Event-enriched state machines

…and bound together using the Event as a binding variable

```
module DistributedSystem {
...
ghost predicate NextStep(c: Constants, v: Variables, v': Variables, evt: Event,
step: Step)
{
  // HostAction calls Host.Next with evt
  && HostAction(c, v, v', evt, step.hostid, step.msgOps)
  && Network.Next(c.network, v.network, v'.network, step.msgOps)
}

ghost predicate Next(c: Constants, v: Variables, v': Variables, evt: Event)
{
  exists step :: NextStep(c, v, v', evt, step)
}
```

The Abstraction function




```
function Abstraction(lv:HashTblState) : (hv:MapSpec.Variables)
{
  ...
}
```

COMPUTER SCIENCE & ENGINEERING

A refinement proof

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

lemma RefinementInit(v:Variables)
    requires Init(v)

    ensures Spec.Init(Abstraction(v))  // Refinement base case

lemma RefinementNext(v:Variables, v':Variables)
    requires Next(v, v', evt)


    ensures Spec.Next(Abstraction(v), Abstraction(v'), evt) // Refinement inductive step
        || Abstraction(v) == Abstraction(v') && evt == NoOp // OR stutter step
```