CSCI 7000-001 Distributed Systems Verification

Lec 1: Introduction

CUPLV
CU Programming Languages
& Verification

Introductions

- About me: Gowtham Kaki

- Assistant Professor, Dept. of Computer Science
- New to CU Boulder - Joined Fall 2020
- PhD from Purdue University, 2019.
  - Thesis: Automatic Reasoning Techniques for Non-Serializable Data-Intensive Applications
- Research: Programming Languages and Formal Methods. Applications in Concurrent and Distributed Systems.
- Best known for Quelea (PLDI 2015) and MRDTs (OOPSLA 2019).
- Enjoy reading pop-science books (recent: Emperor of All Maladies) and biographies/memoirs (recent: Hillbilly Elegy). Amateur cartoonist and racquetball player. Maker of terrible puns.

- About you?
  - Name
  - Academic program
  - Research interests
  - Other interests

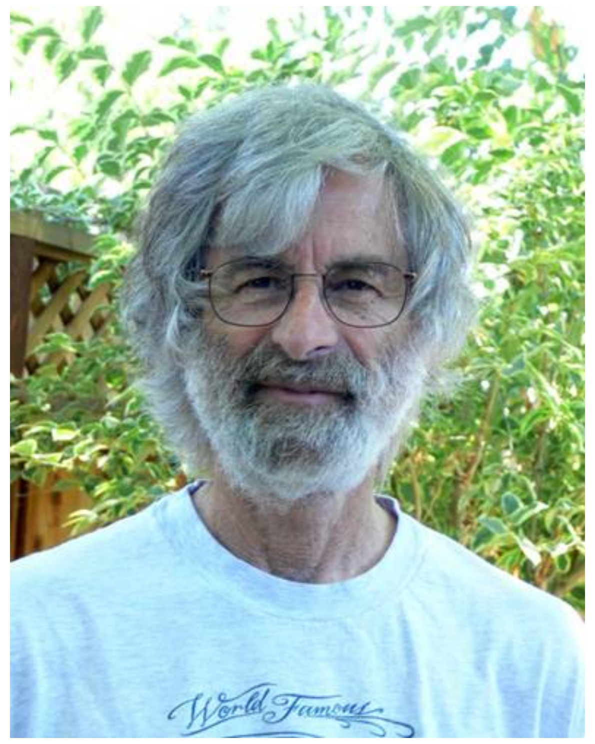
About the course

- Key learning objective is to appreciate and internalize a scientific approach to building and reasoning about distributed systems.
- We shall learn formal mathematics to reason about distributed systems, and apply it to design novel systems.
- Why do we need formal mathematics?

About the course

- Key learning objective is to appreciate and internalize a scientific approach to building and reasoning about distributed systems.
- We shall learn formal mathematics to reason about distributed systems, and apply it to design novel systems.
- Why do we need formal mathematics?

“Writing is nature’s way of letting you know how sloppy your thinking is;
Mathematics is nature’s way of letting you know how sloppy your writing is;
Formal mathematics is nature’s way of letting you know how sloppy your mathematics is.”

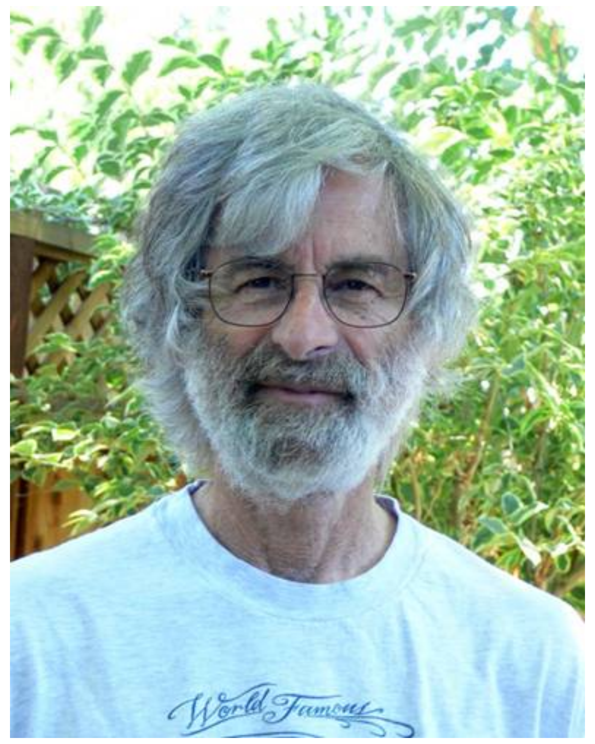


Leslie Lamport

About the course

- Key learning objective is to appreciate and internalize a scientific approach to building and reasoning about distributed systems.
- We shall learn formal mathematics to reason about distributed systems, and apply it to design novel systems.
- Why do we need formal mathematics?

> *“Writing is nature’s way of letting you know how sloppy your thinking is;*
> *Mathematics is nature’s way of letting you know how sloppy your writing is;*
> *Formal mathematics is nature’s way of letting you know how sloppy your mathematics is.”*



Leslie Lamport

- Distributed systems are complex beasts.
  - Sloppy thinking is easy.
  - Sloppy thinking $\Longrightarrow$ terrible systems.

Course structure

- Seminar-style course:
  - Part 1: Instructor-led lectures (10-12).
  - Part 2: Student-led paper presentations and discussions (≥ 2×#students).
- Lectures review the foundations of distributed systems; introduce relevant formal methods & tools.
  - Asynchronicity
  - Logical Time & Vector Clocks
  - Consensus
  - Fault tolerance
  - Safety and Liveness
  - FLP & CAP Impossibilities
  - Paxos, Raft etc
  - Byzantine faults
  - State transition systems
  - Temporal Logic of Actions
  - Inductive reasoning
  - Program Logics
  - Refinement Proof Technique
  - TLA+/PlusCal
  - IVy
- Papers presentations review the state-of-the-art in scientific approach to building distributed systems.
  - Tentative list of papers is posted on the course website. List evolves as the semester progresses.

Grading

Item	Weight
Programming assignment (TLA/PlusCal or IVy)	25%
Research paper presentations	30%
Exploratory project based on a research paper	30%
Contributing to paper discussions	15%

Grading

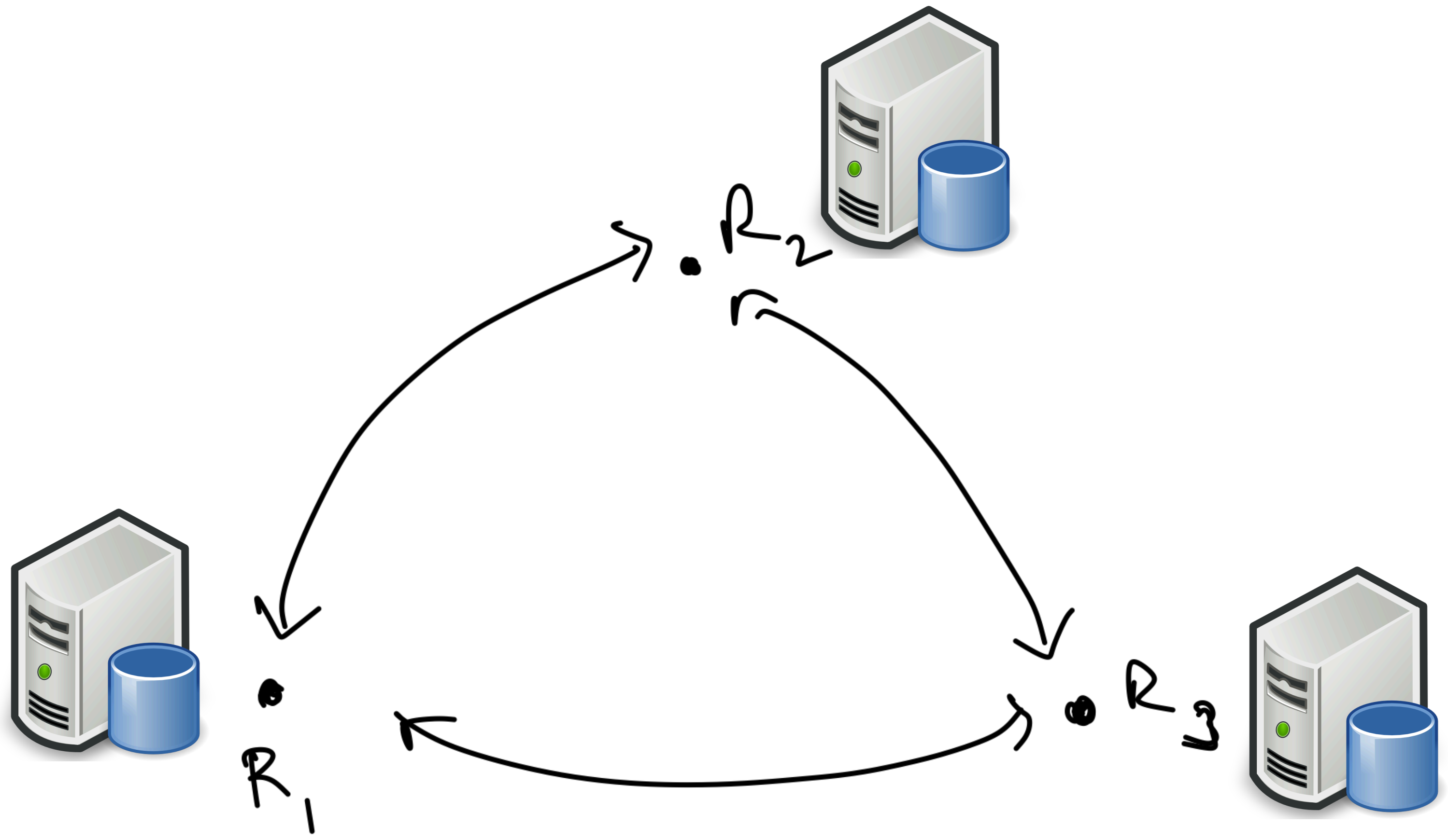
Item	Weight
Programming assignment (TLA/PlusCal or IVy)	25%
Research paper presentations	30%
Exploratory project based on a research paper	30%
Contributing to paper discussions	15%

- Important: the intent of grading is *not* to evaluate you, but to incentivize learning.
- Partial credit shall be awarded wherever possible.
- Efforts to think creatively and try something new shall be rewarded even if the outcome is not a total success.
- Let’s learn together and have fun!

Introduction to distributed systems & formal reasoning

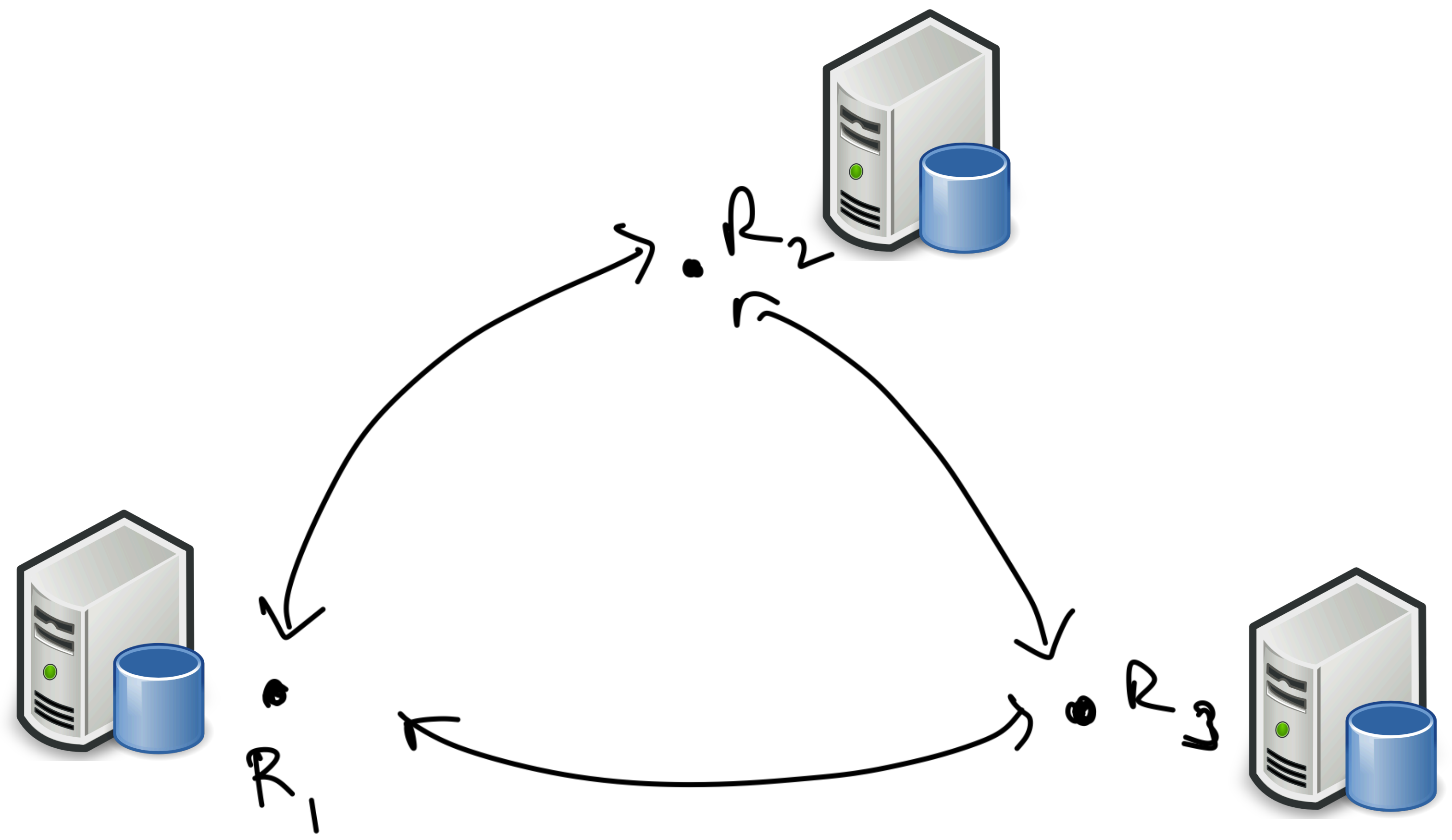
Distributed System

- System of interconnected computers coordinating to execute a computational task.



Distributed System

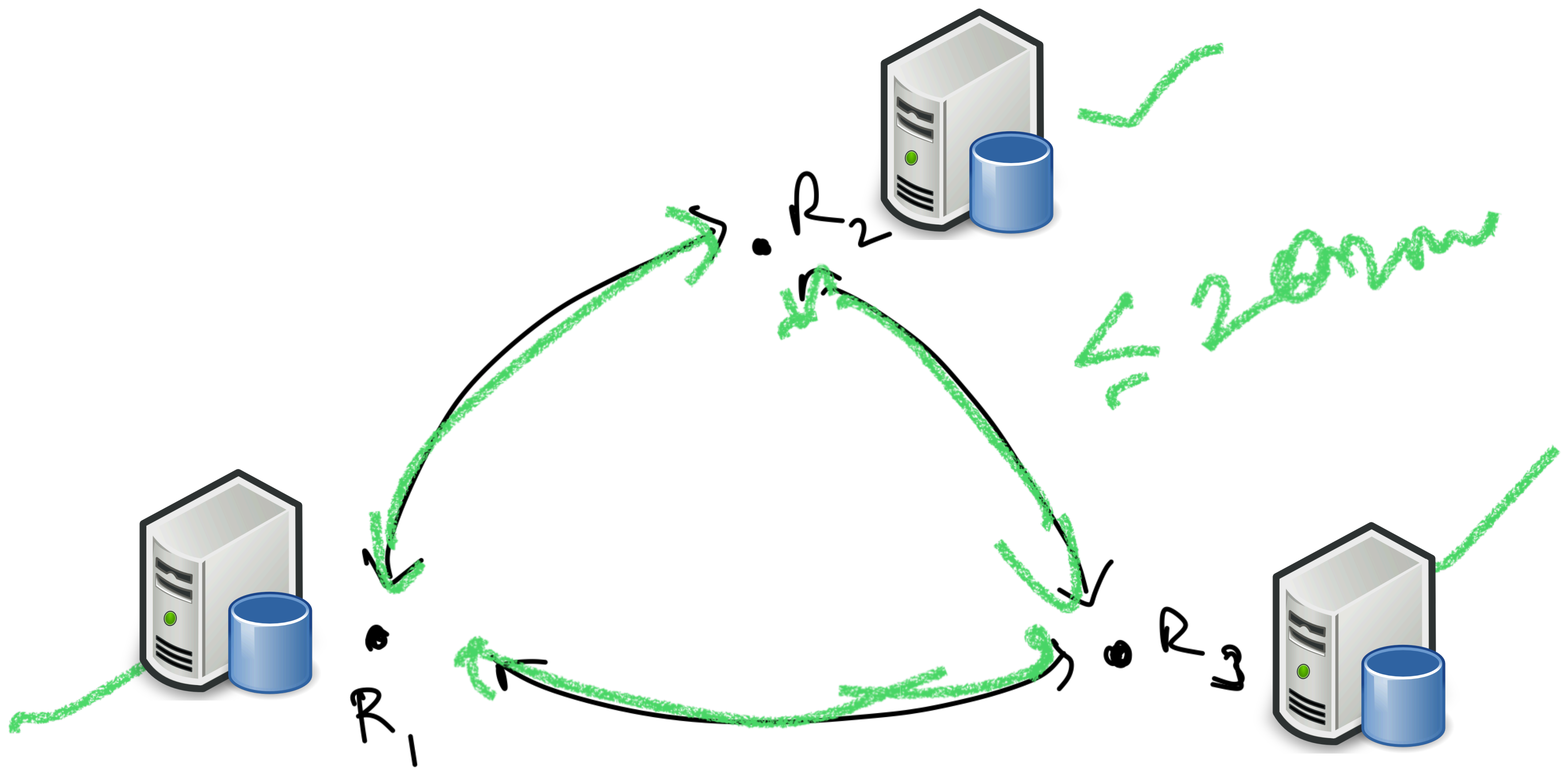
- System of interconnected computers coordinating to execute a computational task.



- In an ideal world:
  - Nodes never crash.
  - Network never fails (latency is finite and known).
  - No message is ever lost or corrupted.

Distributed System

- System of interconnected computers coordinating to execute a computational task.

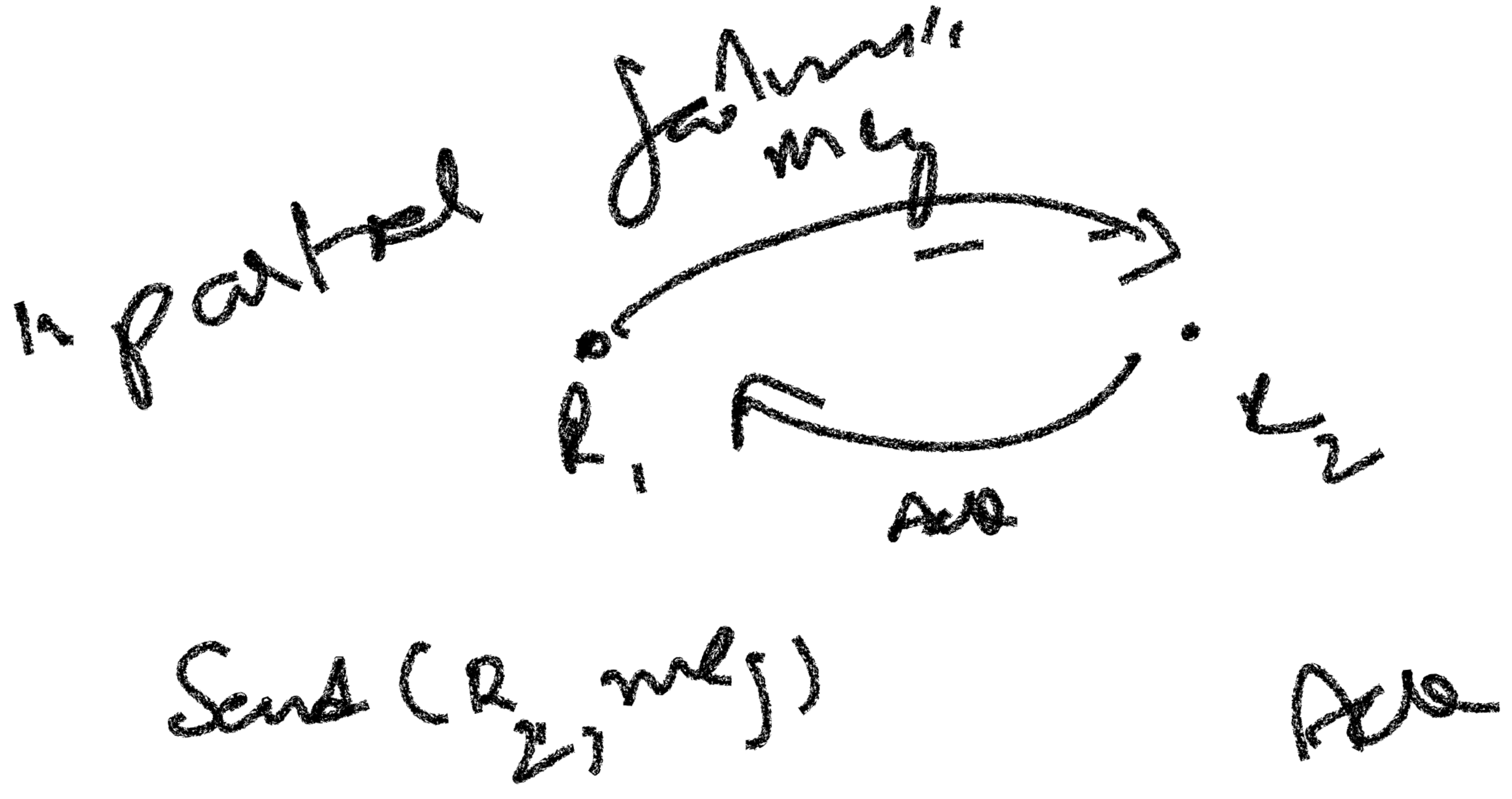


- In an ideal world:
  - Nodes never crash.
  - Network never fails (latency is finite and known).
  - No message is ever lost or corrupted.

- Characterized by partial failures
  - Sub-components can fail independently.
  - Worse: It’s impossible to reliably detect failures!

Distributed System

- System of interconnected computers coordinating to execute a computational task.



* Server can might fail. because internal error

* Network might fail.

* Network may lose the message.

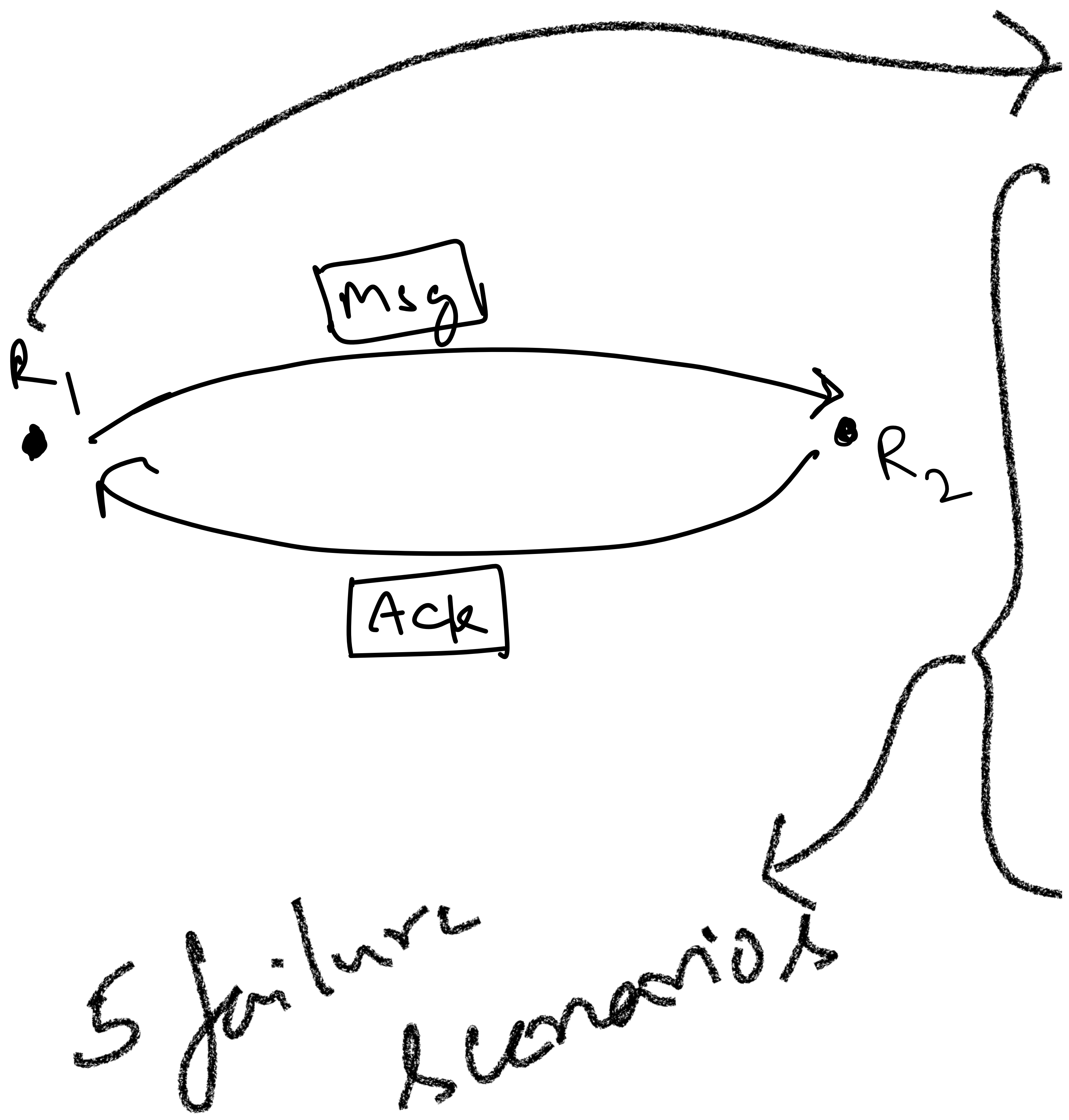
- Characterized by partial failures
  - Sub-components can fail independently.
  - Worse: It's impossible to reliably detect failures!

How to handle failures?

Non answer: terminate the program on every failure.

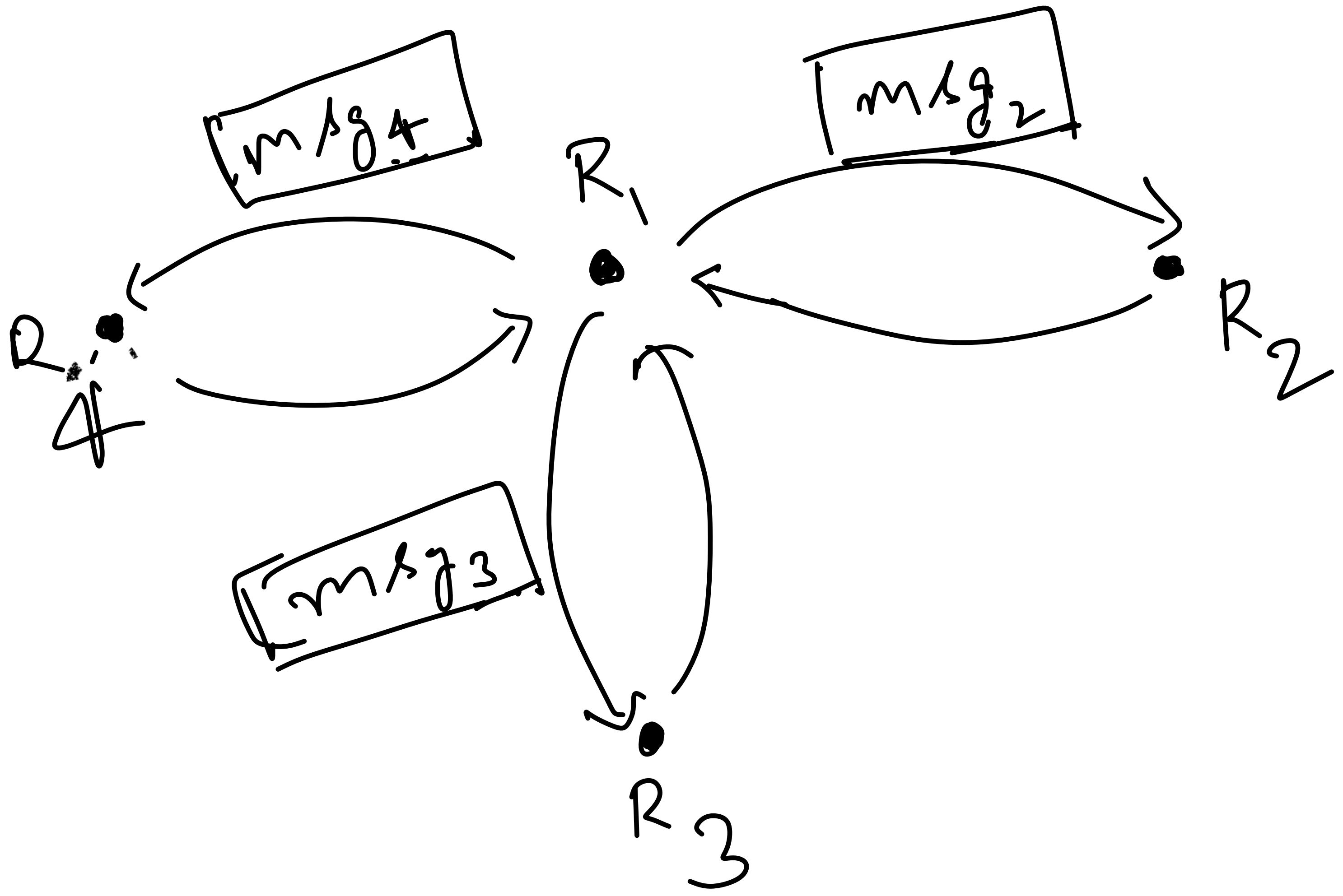
How to handle failures?

Non answer: terminate the program on every failure.



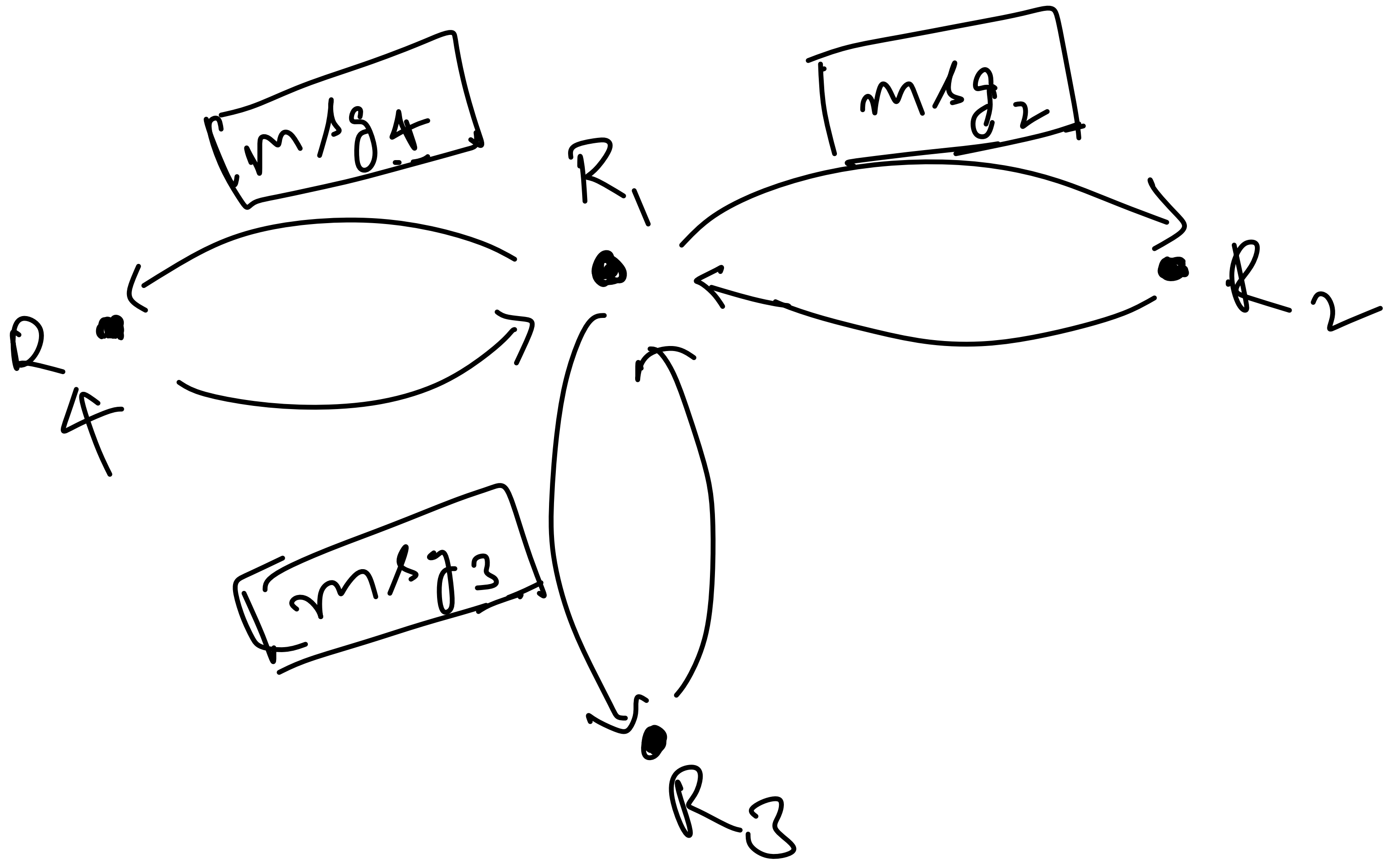
```
resp = send(R[2],msg);
switch (resp->errno) {
    case SEND_FAILED:
    // R2 never got the msg.
    case TIMEOUT:
    // R2 may or may not have received the msg.
    case RETRY:
    // R2 got the msg, but could not respond
    // due to a transient internal error.
    case INVALID_REQ:
    // R2 received corrupt or invalid msg.
    case INVALID_RES:
    // R2's response is corrupt or invalid.
    case SUCCESS:
    // msg sent and response received
}
```

Failure scenarios accumulate



```
for(i=2; i<5; i++) {
    resp[i] = send(R[i], msg[i]);
    //....
}
```

Failure scenarios accumulate



```
for(i=2; i<5; i++) {
    resp[i] = send(R[i], msg[i]);
    //....
}
```

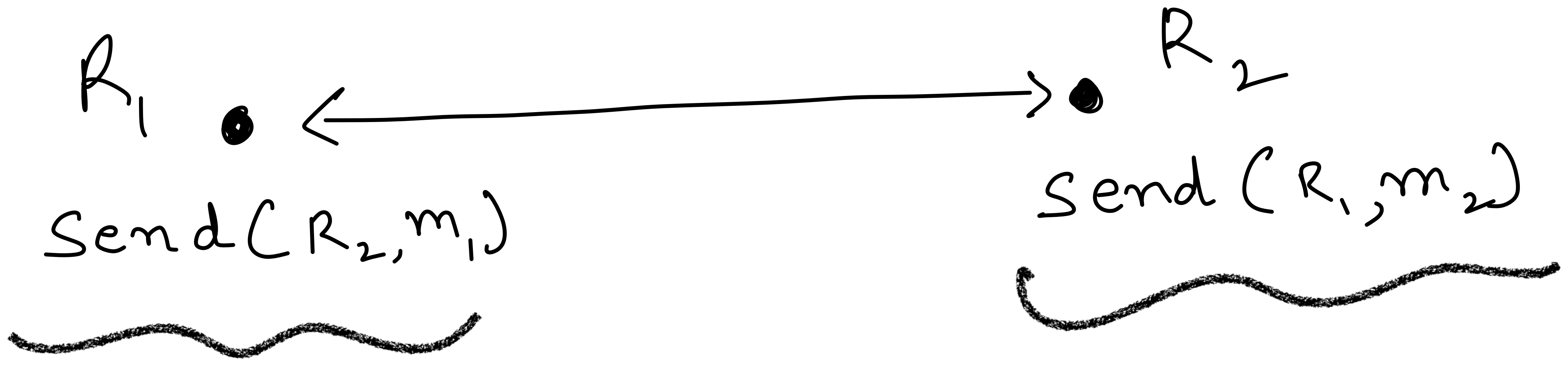
Scenarios:

- `send(R2,..)` fails.
- `send(R2,..)` succeeds, but `send(R3,..)` fails
- `send(R2,..)` and `send(R3,..)` succeed, but `send(R4,..)` fails.

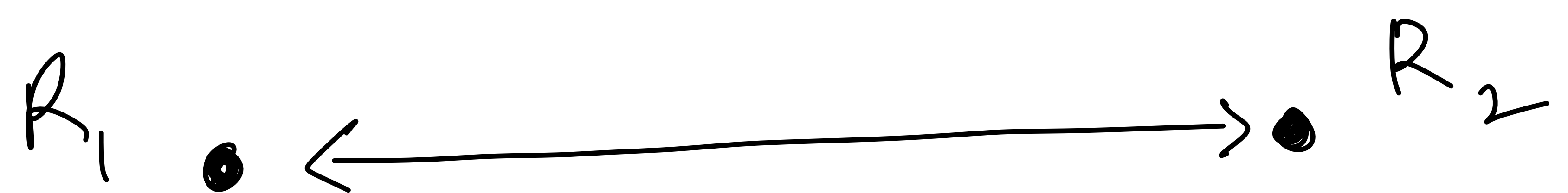
5
+ 5
+ 5

15 failure scenarios

Failure scenarios multiply



Failure scenarios multiply



send(R2,m1)

send(R1,m2)

```
resp = send(R[2],msg);
switch (resp->errno) {
    case SEND_FAILED:
    // R2 never got the msg.
    case TIMEOUT:
    // R2 may or may not have received the msg.
    case RETRY:
    // R2 got the msg, but could not respond
    // due to a transient internal error.
    case INVALID_REQ:
    // R2 received corrupt or invalid msg.
    case INVALID_RES:
    // R2's response is corrupt or invalid.
    case SUCCESS:
    // msg sent and response received
}
```

$\underbrace{5 \times 5}$

25 failure scenarios

```
resp = send(R[2],msg);
switch (resp->errno) {
    case SEND_FAILED:
    // R2 never got the msg.
    case TIMEOUT:
    // R2 may or may not have received the msg.
    case RETRY:
    // R2 got the msg, but could not respond
    // due to a transient internal error.
    case INVALID_REQ:
    // R2 received corrupt or invalid msg.
    case INVALID_RES:
    // R2's response is corrupt or invalid.
    case SUCCESS:
    // msg sent and response received
}
```

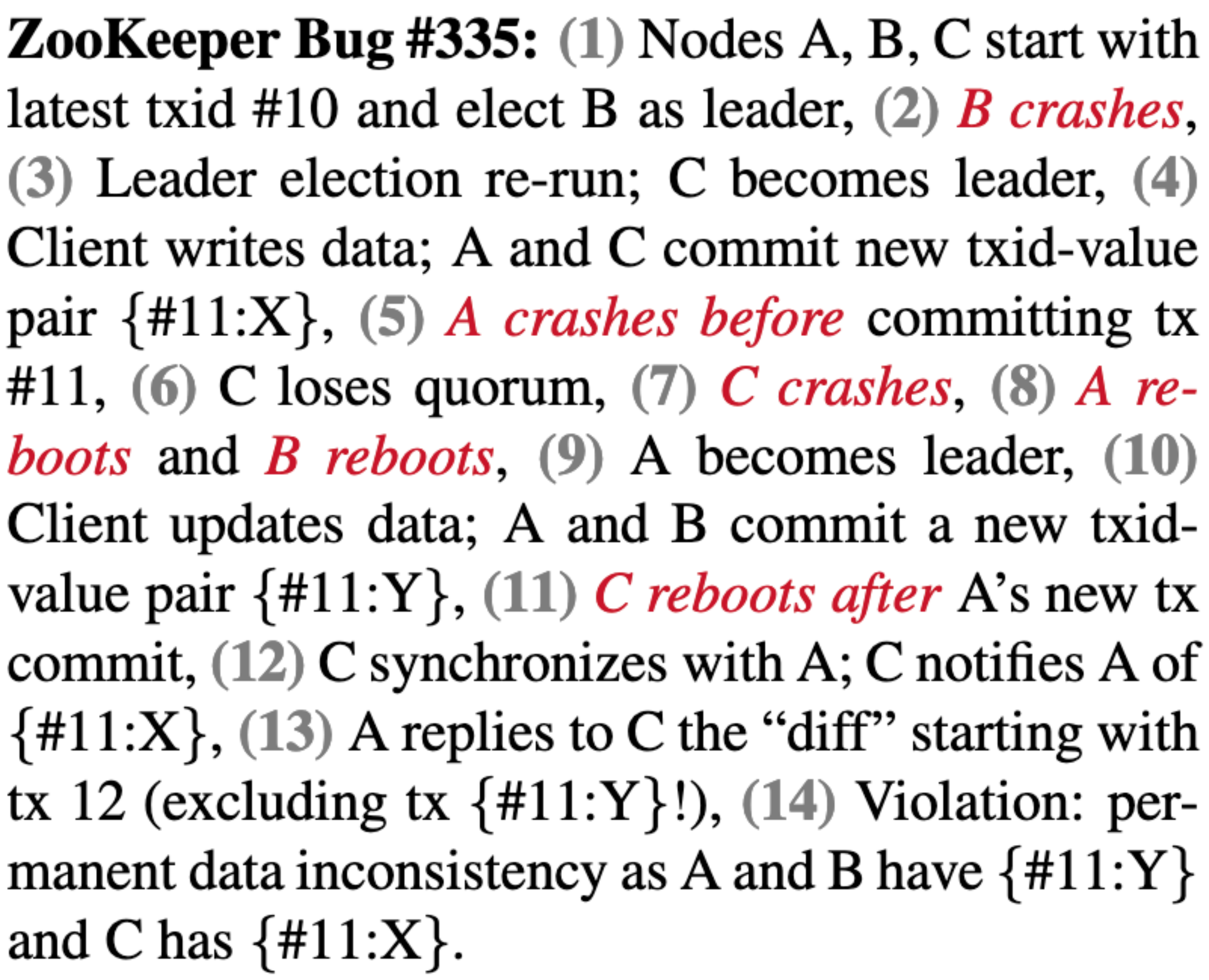
Failure scenarios multiply

- Exhaustively testing a non-trivial distributed system is practically impossible!

Failure scenarios multiply

- Exhaustively testing a non-trivial distributed system is practically impossible!
- Debugging is nightmarish!

From Leesatapornwongsa et al, OSDI’14:



ZooKeeper Bug #335: (1) Nodes A, B, C start with latest txid #10 and elect B as leader, (2) *B crashes*, (3) Leader election re-run; C becomes leader, (4) Client writes data; A and C commit new txid-value pair {#11:X}, (5) *A crashes before* committing tx #11, (6) C loses quorum, (7) *C crashes*, (8) *A reboots* and *B reboots*, (9) A becomes leader, (10) Client updates data; A and B commit a new txid-value pair {#11:Y}, (11) *C reboots after* A’s new tx commit, (12) C synchronizes with A; C notifies A of {#11:X}, (13) A replies to C the “diff” starting with tx 12 (excluding tx {#11:Y}!), (14) Violation: permanent data inconsistency as A and B have {#11:Y} and C has {#11:X}.

14 steps!

Testing vs Formal Verification

- E.g., “slow” multiplication.

```
// Assume m>=0 and n>=0
int slow_multiply(int m, int n) {
    int s = 0, i = 0;
    while(i<n){
        s = s + m;
        i = i + 1;
    }
    return s;
}
```

32 bit int

$2^{32} \times 2^{32} = 2^{64}$

~

18 quintillion.

Combinations of unique test inputs!

- `slow_multiply(m,n) = m×n`

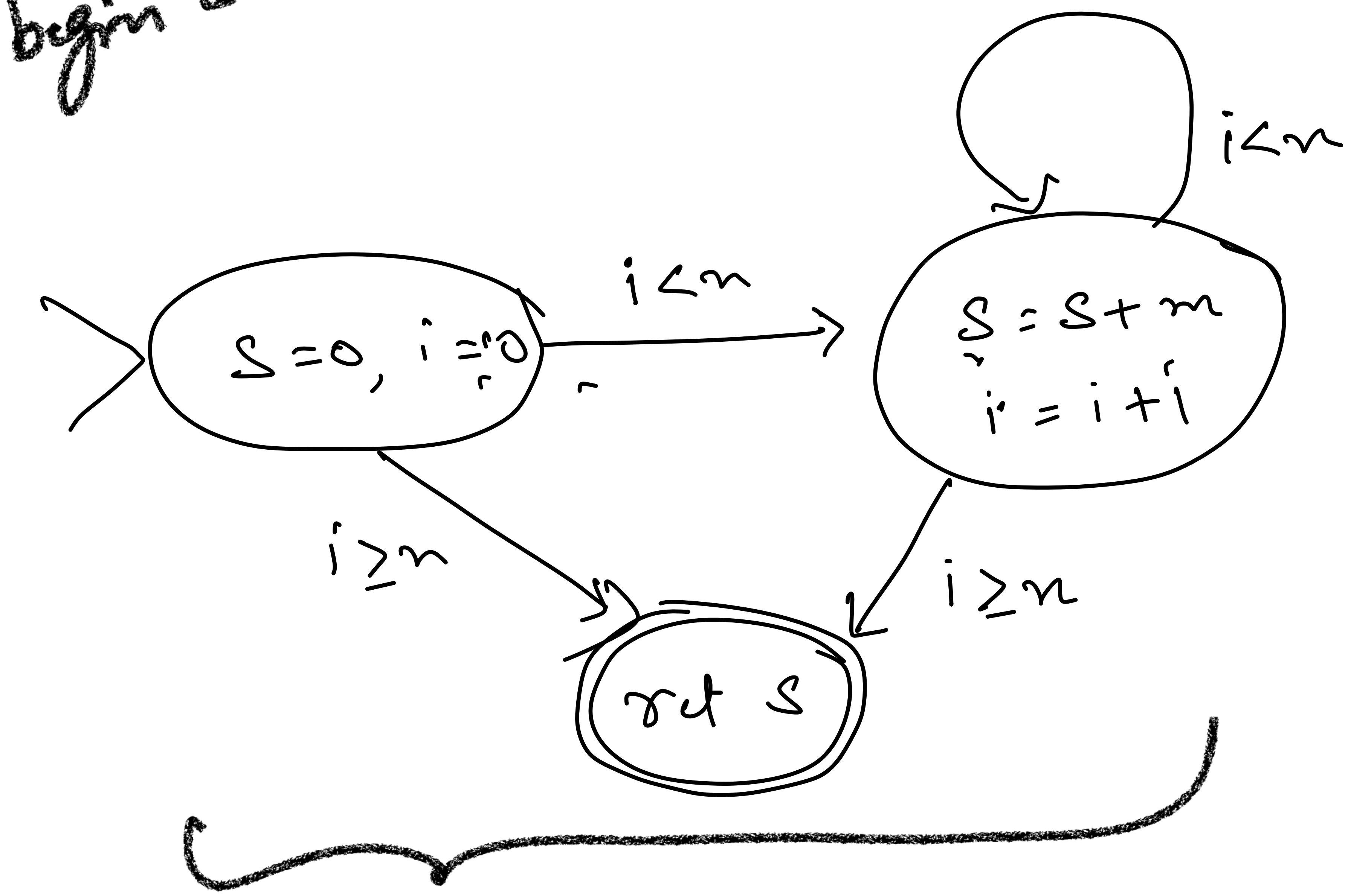
Testing vs Formal Verification

- E.g., “slow” multiplication.

we assume m & n are non negative to begin with

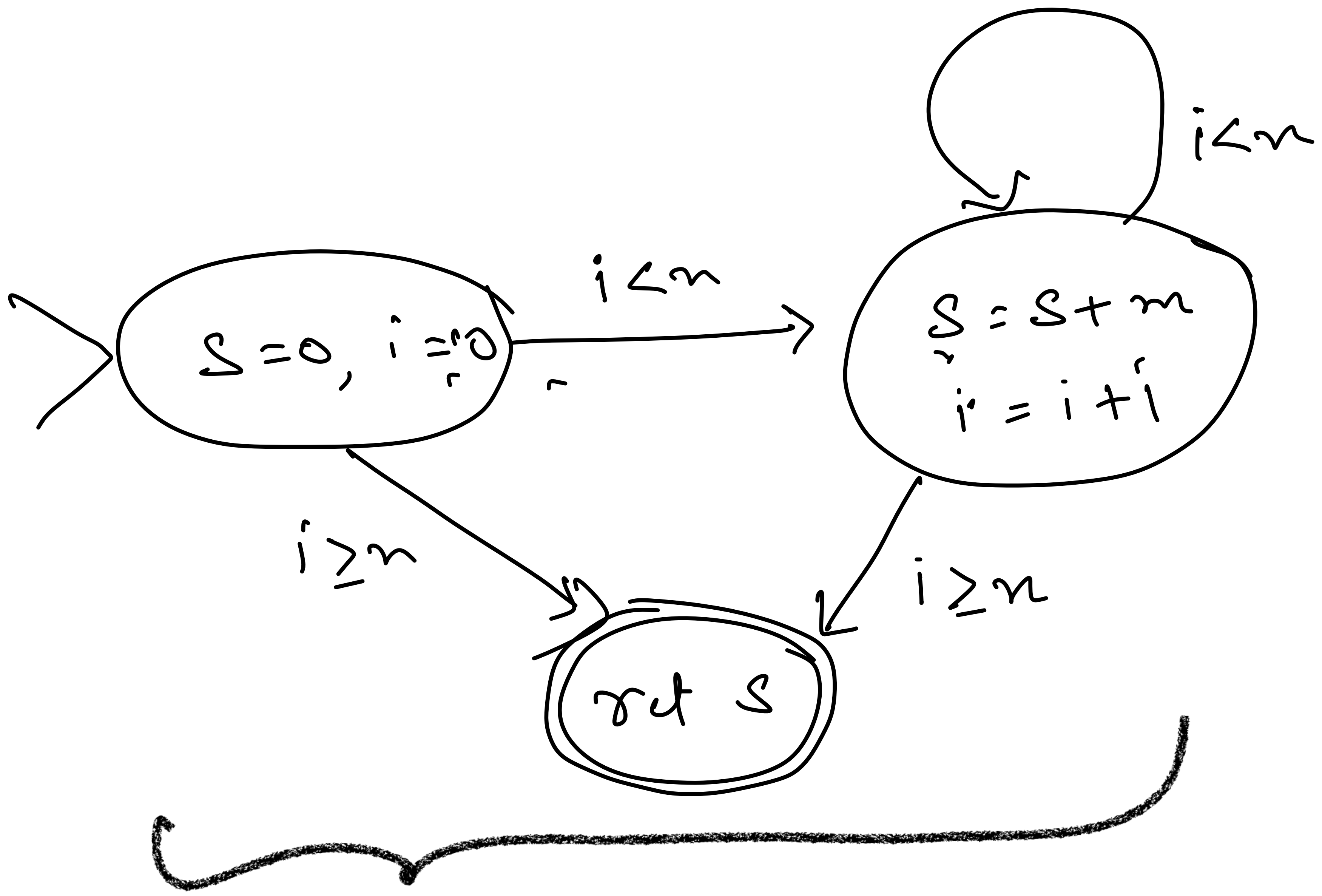
```
// Assume m>=0 and n>=0
int slow_multiply(int m, int n) {
    int s = 0, i = 0;
    while(i<n){
        s = s + m;
        i = i + 1;
    }
    return s;
}
```

- `slow_multiply(m,n) = m×n`



Let's use a Control-Flow Automaton (CFA) as a mathematical model of the program. We would like to prove that in the final state, s=m×n

Testing vs Formal Verification

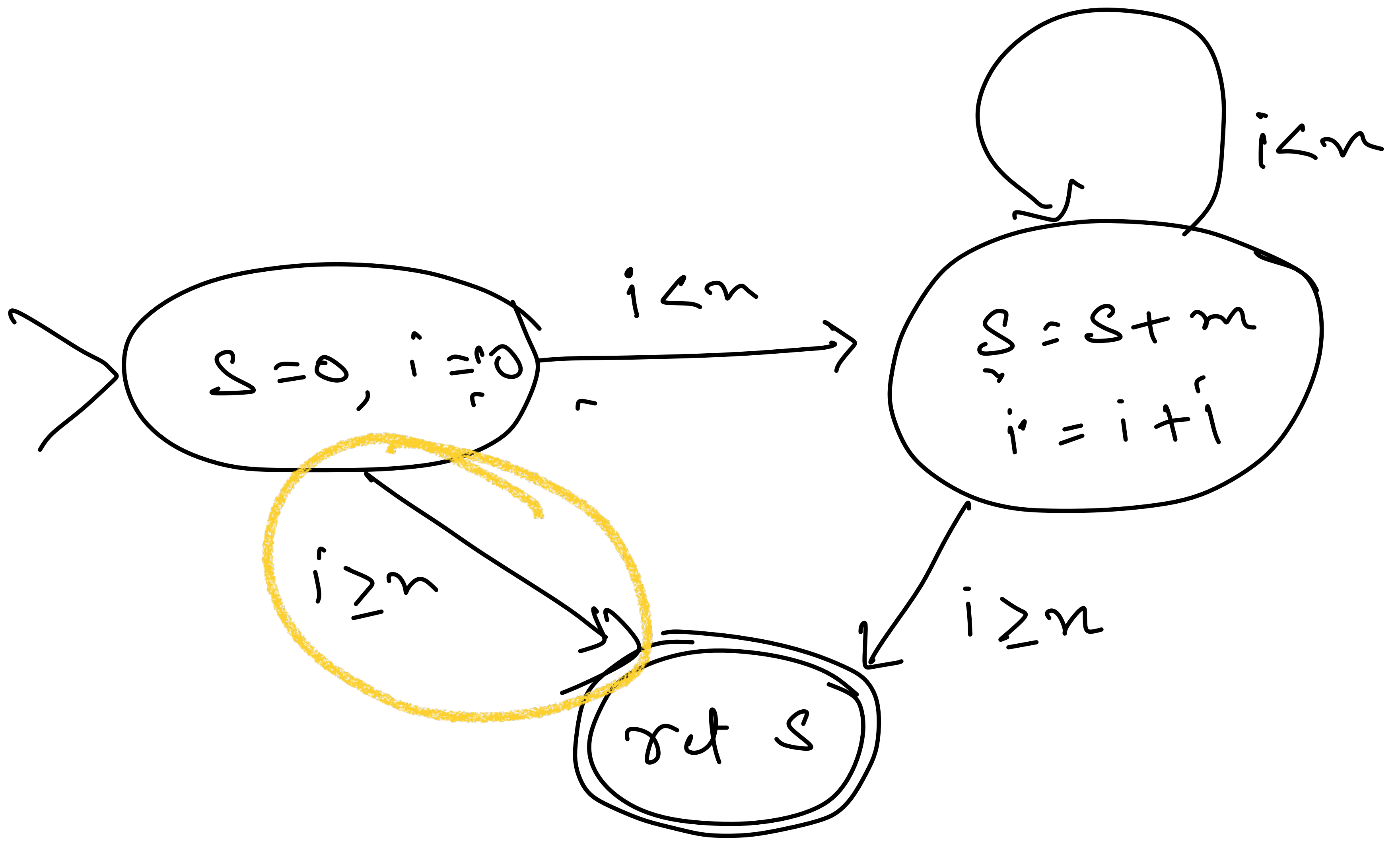


Let's use a Control-Flow Automaton (CFA) as a mathematical model of the program. We would like to prove that in the final state, $\underline{s=m\times n}$

Final state has two incoming transitions. If we know the values of s & i before each transition, we can then assert something about their values after the transition. However, we only know the values at initial state, so we can only do this for 1 transition directly.

Testing vs Formal Verification

Final state has two incoming transitions. If we know the values of s & i before each transition, we can then assert something about their values after the transition. However, we only know the values at initial state, so we can only do this for 1 transition directly.



For this transition;

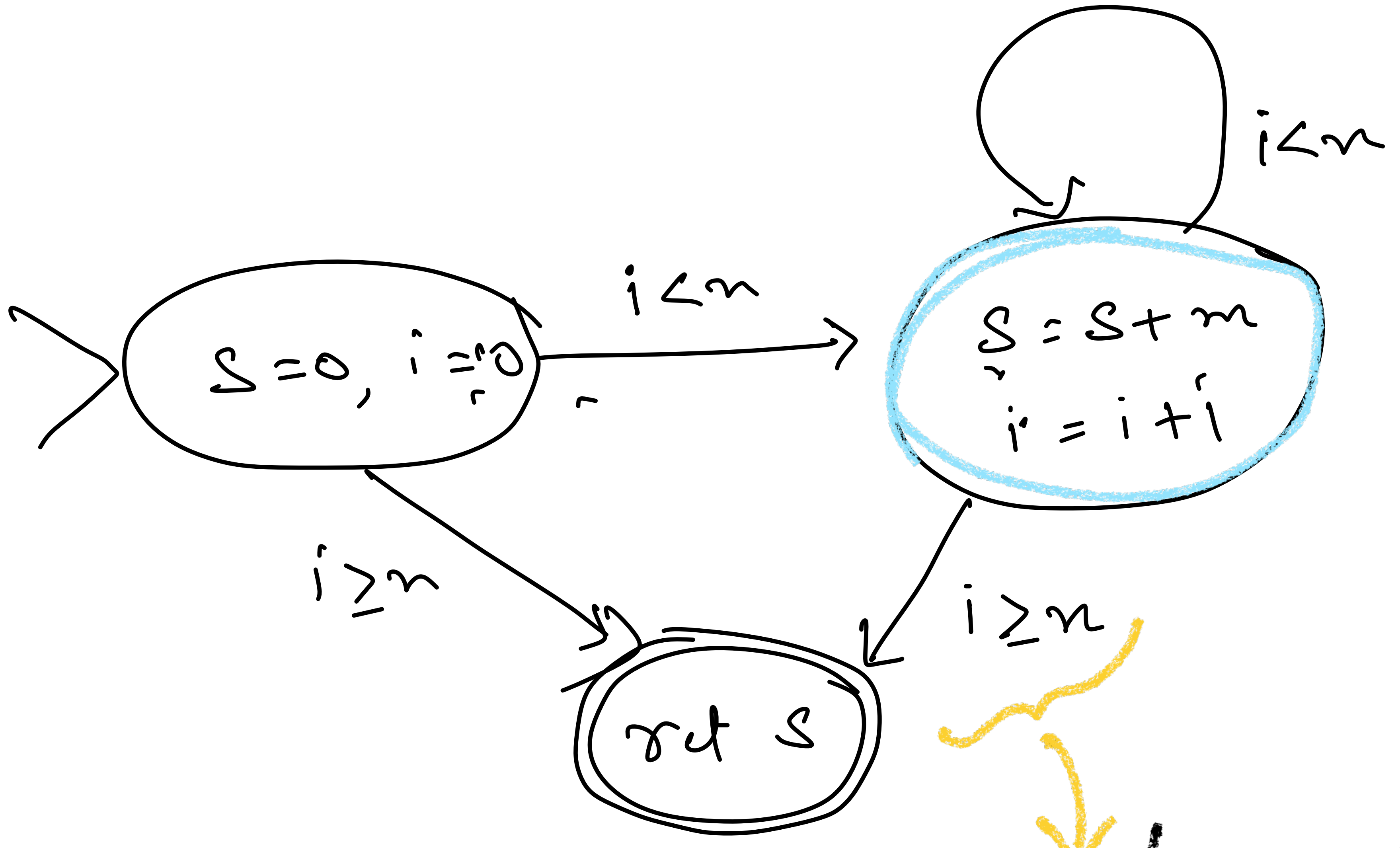
$m \geq 0 \wedge n \geq 0$ (precondition)

$s = 0 \wedge i = 0$ (initial values)

$i \geq n$ (transition condition)

$$\Rightarrow n = 0 \Rightarrow \underline{\underline{s = m \times n}}$$

Testing vs Formal Verification



what about the other transition?

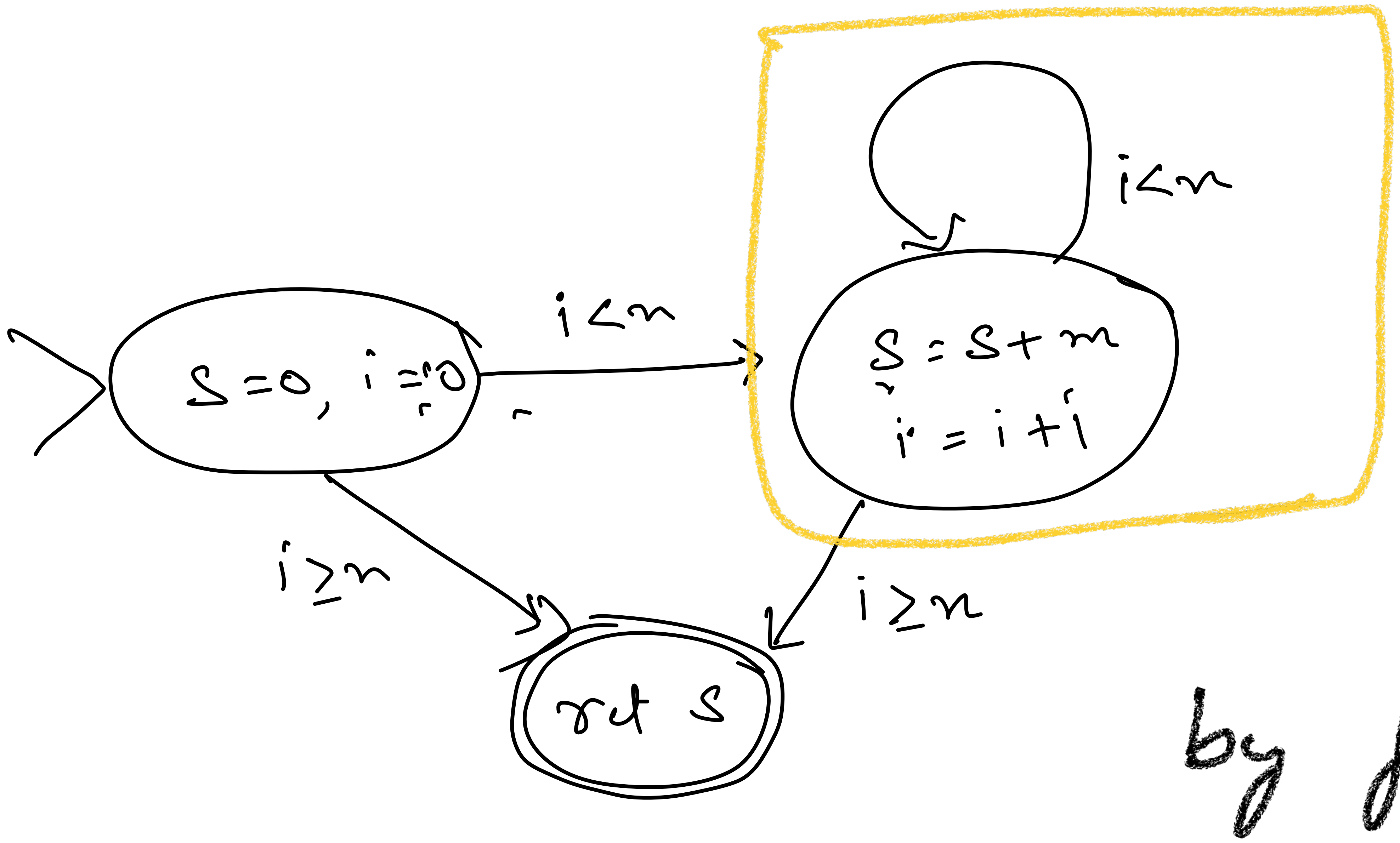
Unfortunately, here we don't know the precise values of s & i in the pre-state. The values are defined "recursively" in terms of themselves

For proof to work, we have to "untangle" this recursion and come up with a formula Φ s.t. Φ is true as long as the control is at the intermediate state (pre-state for final transition; circled in blue)

I claim $\Phi = i \leq n \wedge s = m \times i$

How did I come up with this Φ? purely based on the intuition, but there are automatic methods for this (lookup research on "inductive invariant inference").

Testing vs Formal Verification



Can be replaced by
$i \leq n \wedge s = m \times i$

With the intermediate state replaced by formula $\Phi = i \leq n \wedge s = m \times i$, let's see if we can now reason about second transition.

For the second transition:

$m \geq 0 \wedge n \geq 0$ (Pre condition)

$i \leq n \wedge s = m \times i$ (property of pre-state)

$i \geq n$ (transition condition)

$\Rightarrow i = n \Rightarrow \underline{\underline{s = m \times n}}$

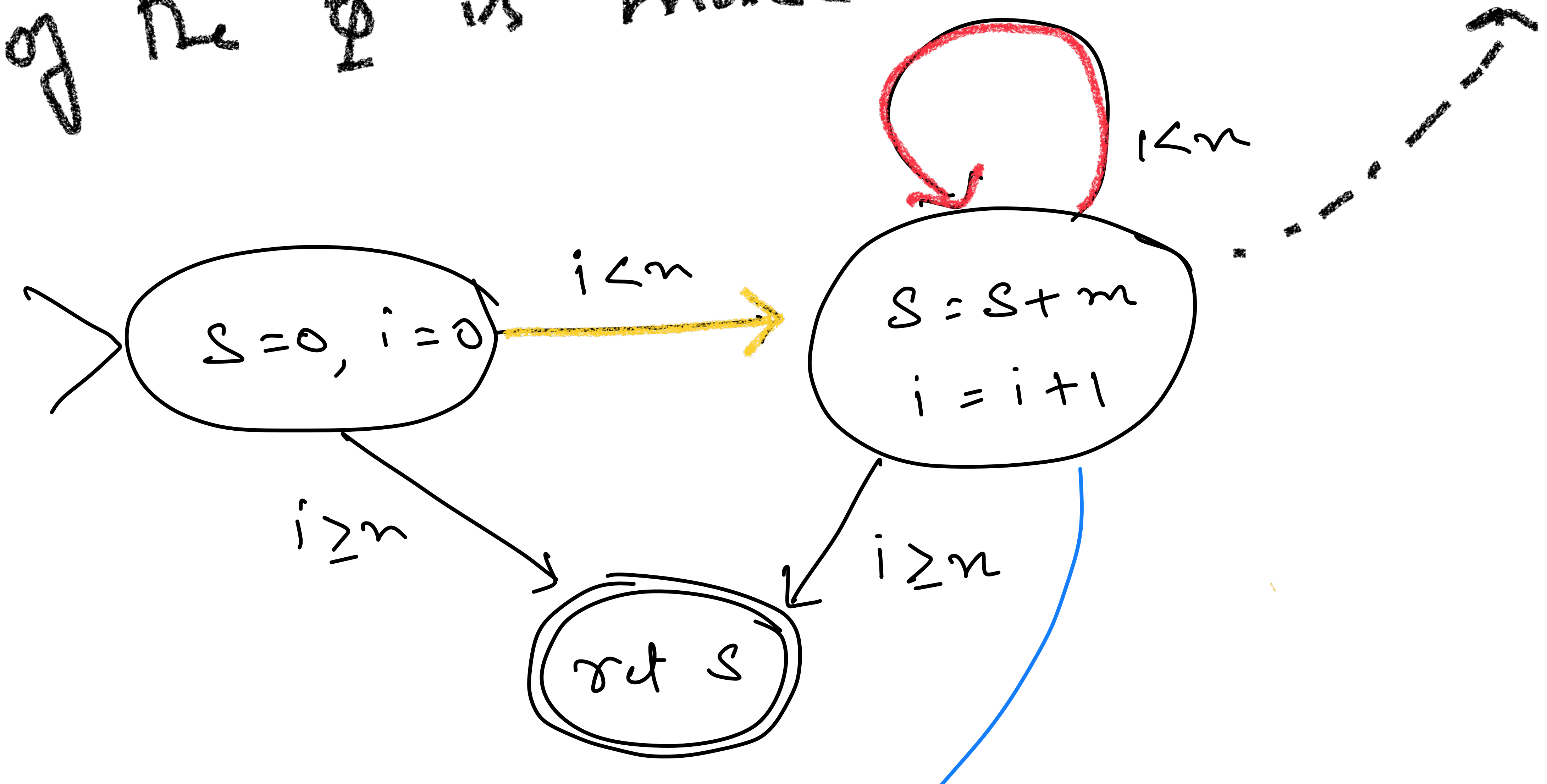
Testing vs Formal Verification

- E.g., "slow" multiplication.

```
// Assume m>=0 and n>=0
int slow_multiply(int m, int n) {
    int s = 0, i = 0;
    while(i<n){
        s = s + m;
        i = i + 1;
    }
    return s;
}
```

- Prove `slow_multiply(m,n)` = m×n

Prove the Φ is indeed true in this state



Claim: As long as we are in this state, $i \leq n \wedge s = m \times i$

(1) Base Case: $m \geq 0 \wedge n \geq 0 \wedge s = 0 \wedge i = 0 \wedge i < n$

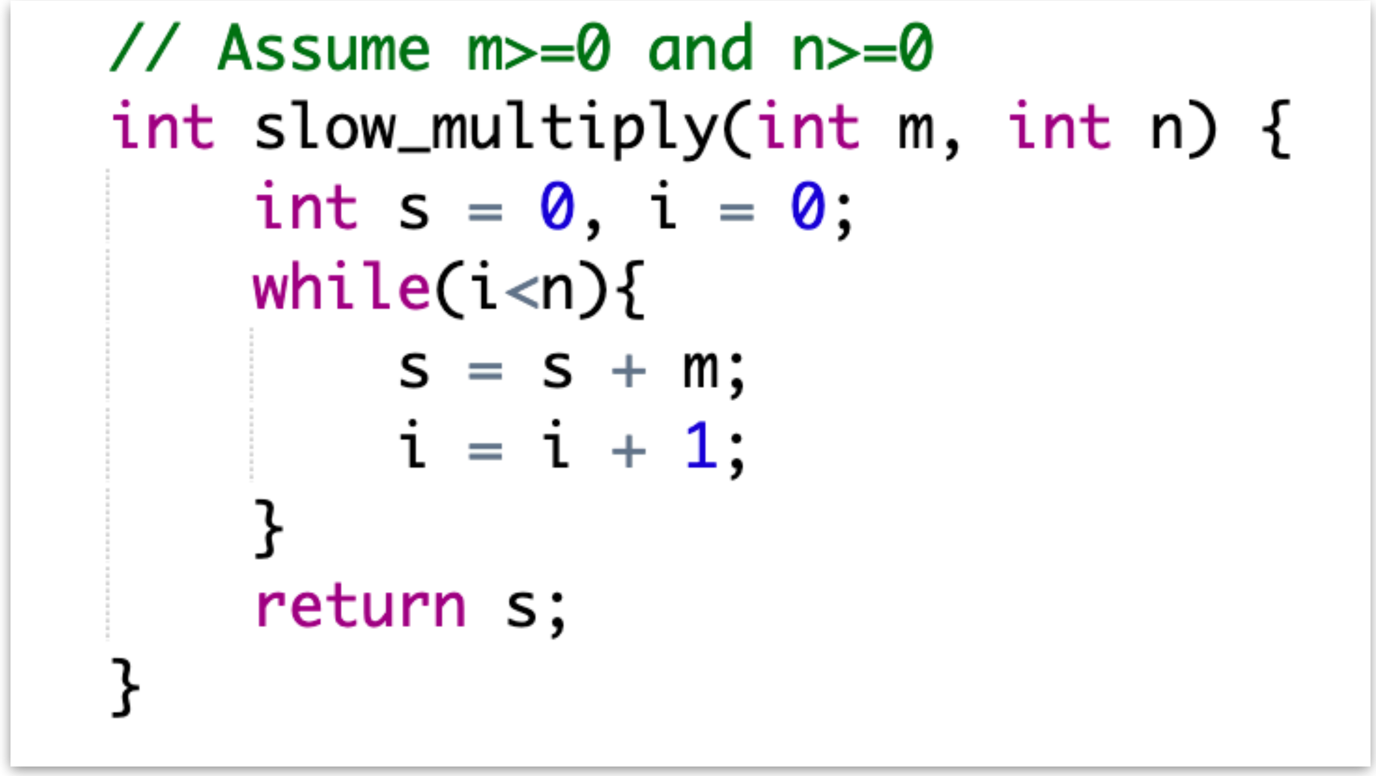
(Yellow transition) $\Rightarrow i \leq n \wedge s = m \times i$

(2) Inductive Case. $i \leq n \wedge s = m \times i \wedge i < n$

(red transition) $\Rightarrow i+1 \leq n \wedge s+m = m \times (i+1)$

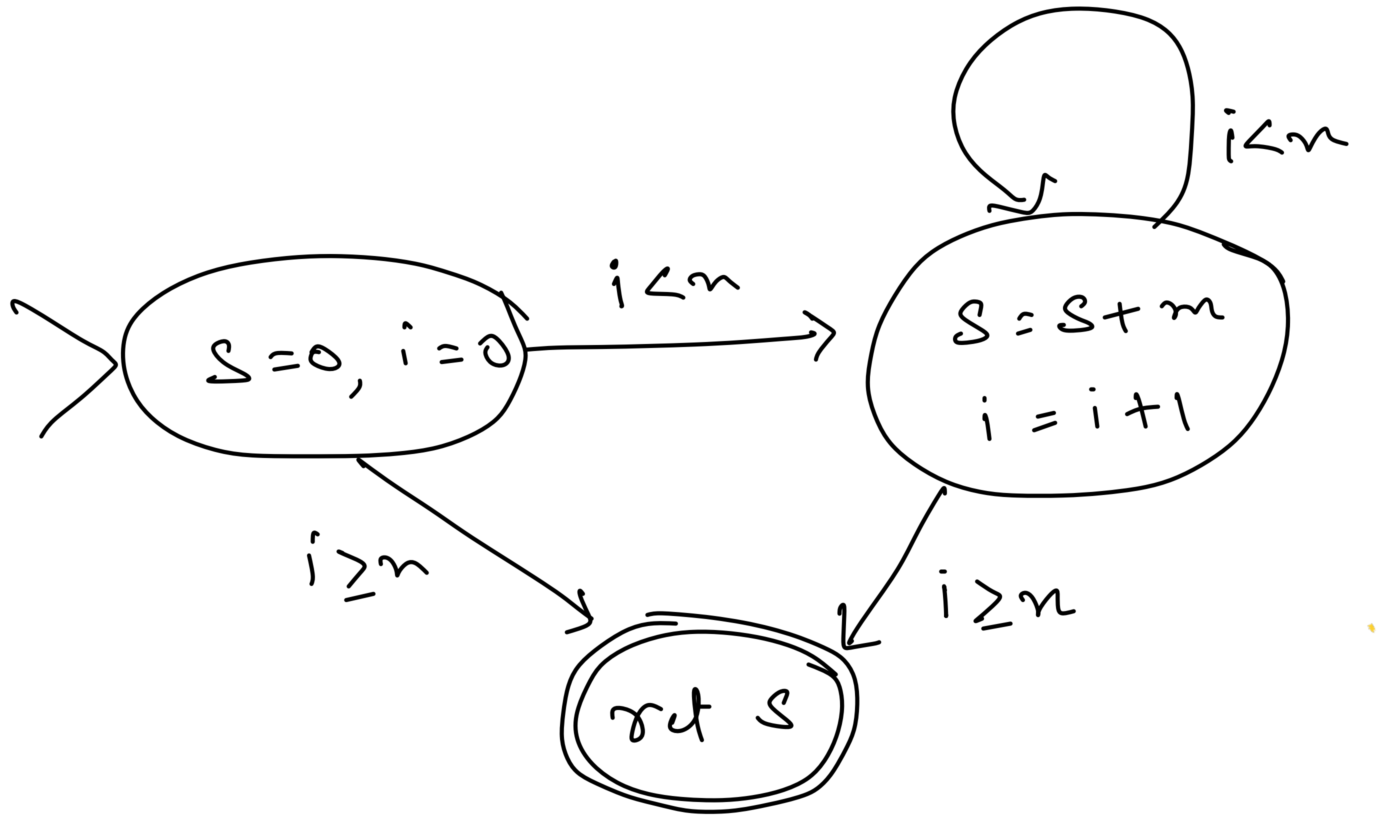
Formal Verification for Distributed Systems

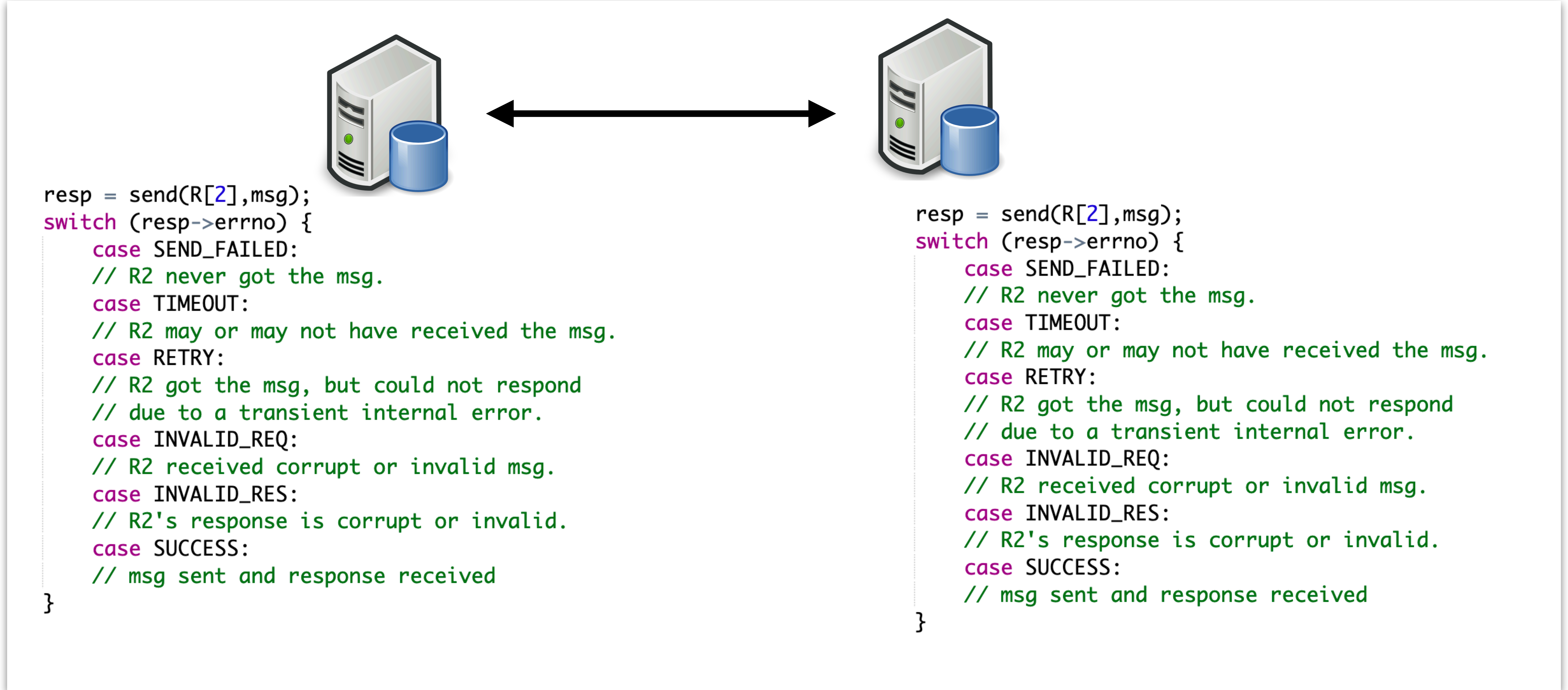
- How do we formalize a distributed system/program?



```
// Assume m>=0 and n>=0
int slow_multiply(int m, int n) {
    int s = 0, i = 0;
    while(i<n){
        s = s + m;
        i = i + 1;
    }
    return s;
}
```

$\cong$





```
resp = send(R[2],msg);
switch (resp->errno) {
    case SEND_FAILED:
    // R2 never got the msg.
    case TIMEOUT:
    // R2 may or may not have received the msg.
    case RETRY:
    // R2 got the msg, but could not respond
    // due to a transient internal error.
    case INVALID_REQ:
    // R2 received corrupt or invalid msg.
    case INVALID_RES:
    // R2's response is corrupt or invalid.
    case SUCCESS:
    // msg sent and response received
}
```

```
resp = send(R[2],msg);
switch (resp->errno) {
    case SEND_FAILED:
    // R2 never got the msg.
    case TIMEOUT:
    // R2 may or may not have received the msg.
    case RETRY:
    // R2 got the msg, but could not respond
    // due to a transient internal error.
    case INVALID_REQ:
    // R2 received corrupt or invalid msg.
    case INVALID_RES:
    // R2's response is corrupt or invalid.
    case SUCCESS:
    // msg sent and response received
}
```

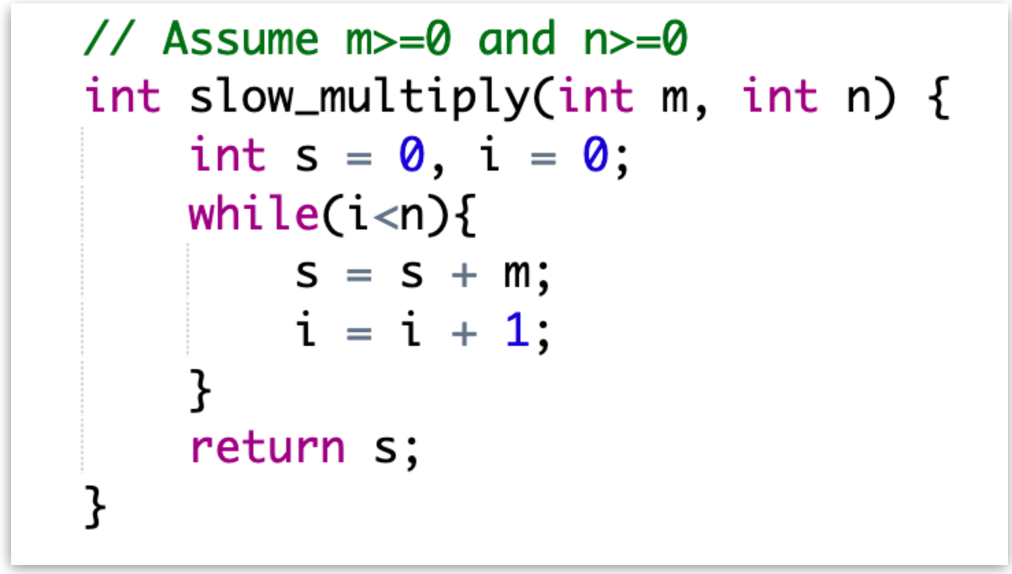
$\cong$

??

Formal Verification for Distributed Systems

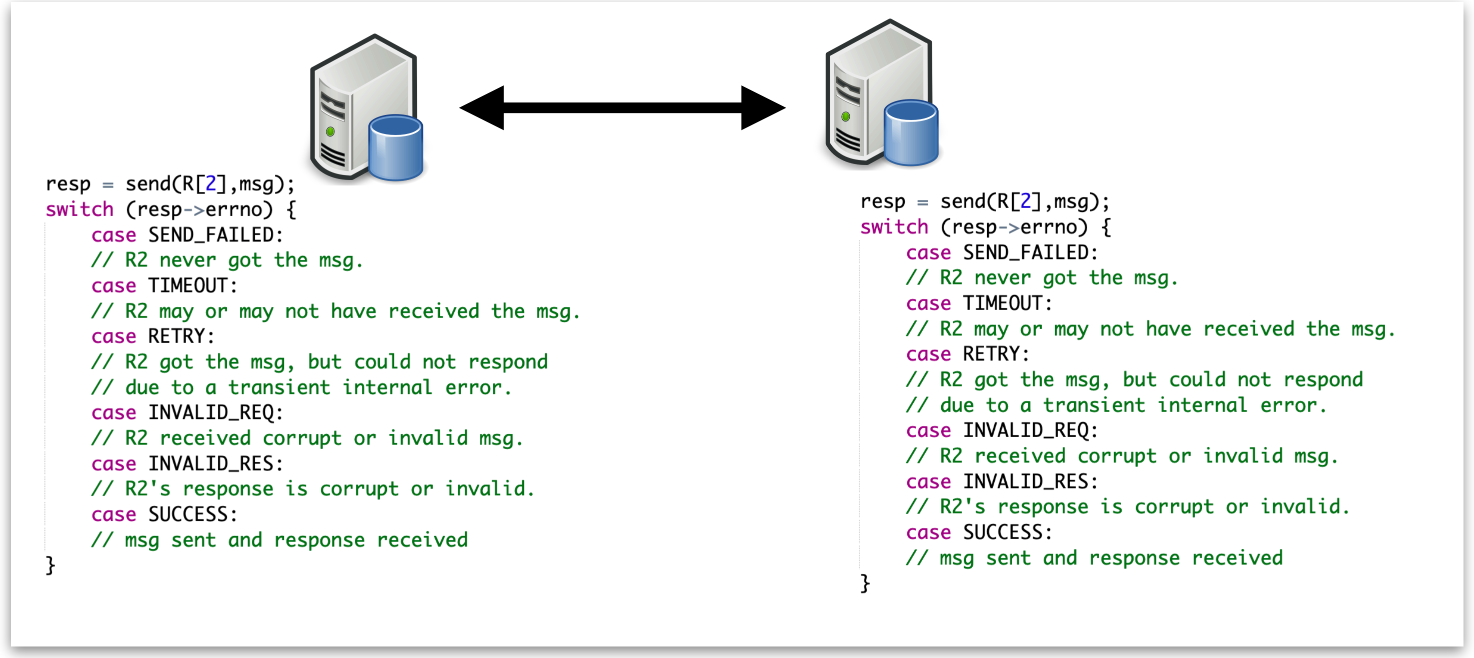
- How do we formalize a distributed system/program?
- What are the properties of interest? How are they specified?

LTL, variants of FOL,



```
// Assume m>=0 and n>=0
int slow_multiply(int m, int n) {
    int s = 0, i = 0;
    while(i<n){
        s = s + m;
        i = i + 1;
    }
    return s;
}
```

$$\underbrace{\texttt{slow\_multiply(m,n) = m×n}}$$



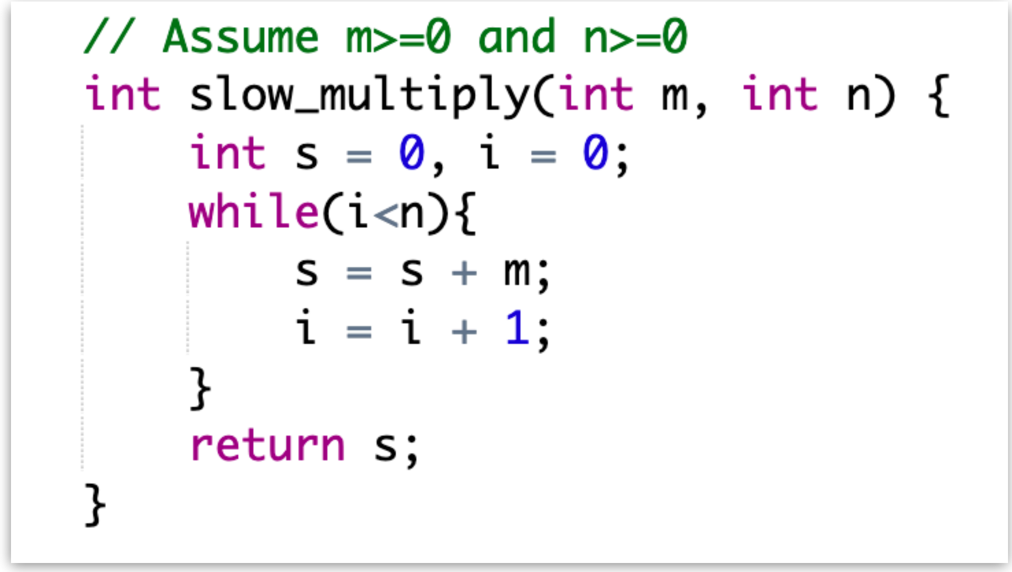
```
resp = send(R[2],msg);
switch (resp->errno) {
    case SEND_FAILED:
    // R2 never got the msg.
    case TIMEOUT:
    // R2 may or may not have received the msg.
    case RETRY:
    // R2 got the msg, but could not respond
    // due to a transient internal error.
    case INVALID_REQ:
    // R2 received corrupt or invalid msg.
    case INVALID_RES:
    // R2's response is corrupt or invalid.
    case SUCCESS:
    // msg sent and response received
}
```

```
resp = send(R[2],msg);
switch (resp->errno) {
    case SEND_FAILED:
    // R2 never got the msg.
    case TIMEOUT:
    // R2 may or may not have received the msg.
    case RETRY:
    // R2 got the msg, but could not respond
    // due to a transient internal error.
    case INVALID_REQ:
    // R2 received corrupt or invalid msg.
    case INVALID_RES:
    // R2's response is corrupt or invalid.
    case SUCCESS:
    // msg sent and response received
}
```

???

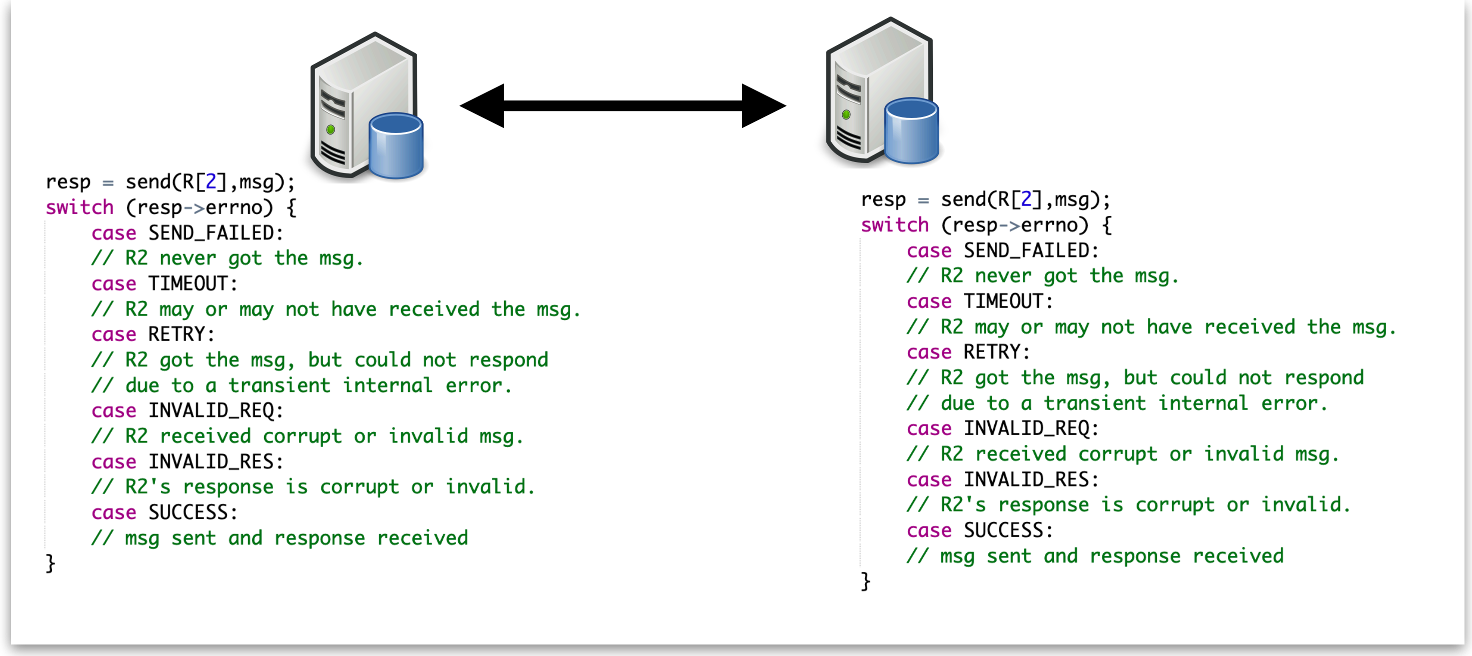
Formal Verification for Distributed Systems

- How do we formalize a distributed system/program?
- What are the properties of interest? How are they specified?
- How do we prove those properties (automatically)?



```
// Assume m>=0 and n>=0
int slow_multiply(int m, int n) {
    int s = 0, i = 0;
    while(i<n){
        s = s + m;
        i = i + 1;
    }
    return s;
}
```

Induction



```
resp = send(R[2],msg);
switch (resp->errno) {
    case SEND_FAILED:
    // R2 never got the msg.
    case TIMEOUT:
    // R2 may or may not have received the msg.
    case RETRY:
    // R2 got the msg, but could not respond
    // due to a transient internal error.
    case INVALID_REQ:
    // R2 received corrupt or invalid msg.
    case INVALID_RES:
    // R2's response is corrupt or invalid.
    case SUCCESS:
    // msg sent and response received
}
```

```
resp = send(R[2],msg);
switch (resp->errno) {
    case SEND_FAILED:
    // R2 never got the msg.
    case TIMEOUT:
    // R2 may or may not have received the msg.
    case RETRY:
    // R2 got the msg, but could not respond
    // due to a transient internal error.
    case INVALID_REQ:
    // R2 received corrupt or invalid msg.
    case INVALID_RES:
    // R2's response is corrupt or invalid.
    case SUCCESS:
    // msg sent and response received
}
```

???

We hand-wrote informal manual proof of correctness for slow_mult. We don't want to do this for complex distributed programs. How do we automate the reasoning?

Formal Verification for Distributed Systems

- How do we formalize a distributed system/program?
- What are the properties of interest? How are they specified?
- How do we prove those properties (automatically)?

+

- Effective testing strategies
- Design principles
- Domain-specific reasoning techniques

This course!