

# Teaching Rigorous Distributed Systems With Efficient Model Checking

Ellis Michael

Doug Woos

Thomas Anderson

Michael D. Ernst

Zachary Tatlock

UNIVERSITY *of* WASHINGTON



# UW CSE 452



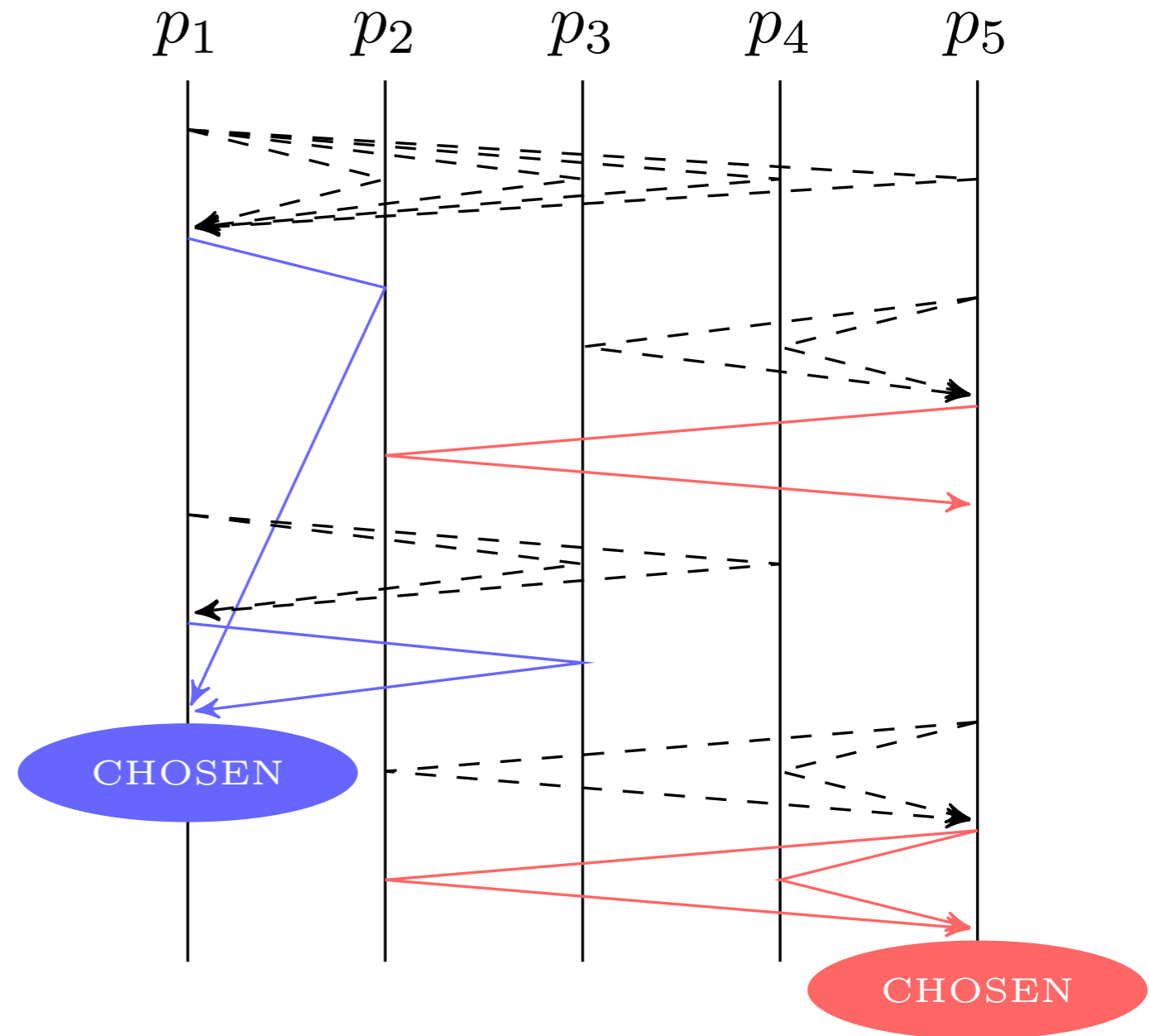
- Course on distributed systems for undergraduates and 5th year Master's students, enrollment grown to approximately 200
- Lab assignments building fault-tolerant, consistent distributed systems, based on assignments developed for MIT 6.824:
  1. Exactly-once RPC
  2. Primary-backup
  3. Paxos-based state machine replication
  4. Sharded key-value store
  5. Distributed transactions using two-phase commit
- Tests used for grading assignments given to students

Goal: Tests which identify common bugs, provide timely feedback, and assist debugging to help students build systems to rigorous standards.

**Systems solution for  
teaching distributed  
systems**

# Testing Distributed Systems is Difficult

- Simple Paxos bug: leader checks for quorum with matching values (rather than proposal numbers).
- Finding such a bug is difficult with current tools.
- This false quorum bug could be caused by a fundamental misunderstanding.



"Just 3 days before the deadline of the project, my partner and I discovered that our Paxos failed 1 of 100,000 tests. ... We realized that the bug comes from our optimization of duplicate request detection before putting request on the Paxos operation log. ... We needed to rewrite fifty percent of the whole project but we did not give up. Finally, after 30 hours of work in 2 days, we fixed the design flaw and eliminated the bug. We were so excited that we started to dance in the lab."

– CSE 452 Student

# Checking Correctness



- Execution-based testing is insufficient; can miss bugs unlikely to occur based on timing.
- Manual review does not scale or provide feedback quickly enough.
- Formal verification is difficult and time-consuming, not approachable for students.

# Checking Correctness: Model Checking



- Researchers and practitioners use model checking to validate protocols and software, systematically searching through possible executions.
- Some specification languages are difficult to learn, do not produce runnable code.
- Naïve methods do not scale well, fail to find rare bugs quickly and reliably.

# DSLabs

A framework for creating distributed systems labs and test suites

- ... capable of finding common bugs in students' implementations quickly and reliably

- ... using a widely-used programming language (Java) and easily-learned tools

- ... that helps students write correct, efficient, runnable code

  - ... and understand errors when they do arise.

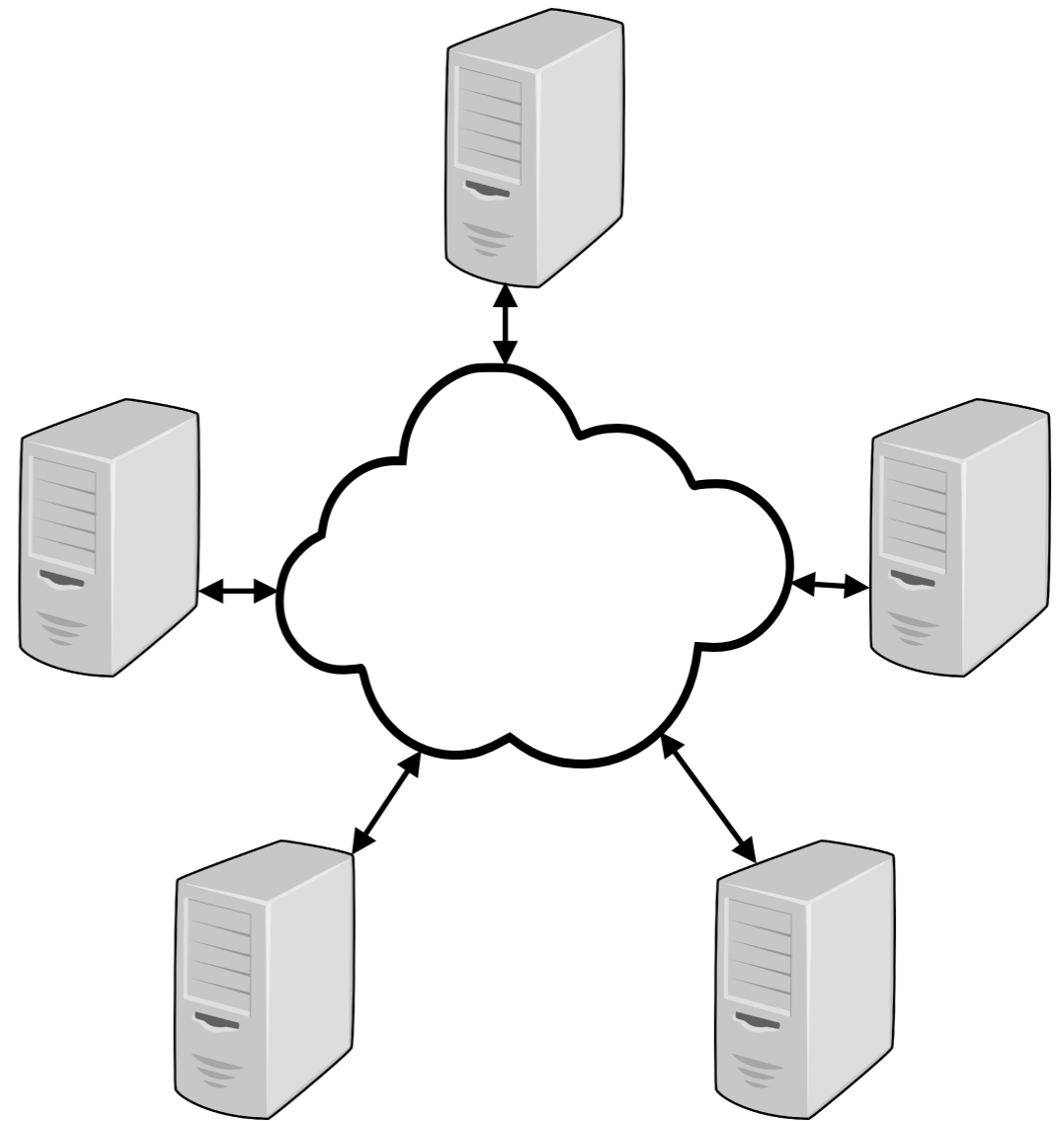


# The Rest of This Talk

1. The DSLabs programming model
2. Model checking strategies and optimizations
3. Understandability and Oddity visual debugger
4. Experiences

# DSLabs Programming Model

- A distributed system consists of a set of nodes which communicate over an asynchronous network, working together to run a protocol.
- Nodes are I/O automata; they run as single-threaded event loops.
- Nodes are split between client and server nodes.

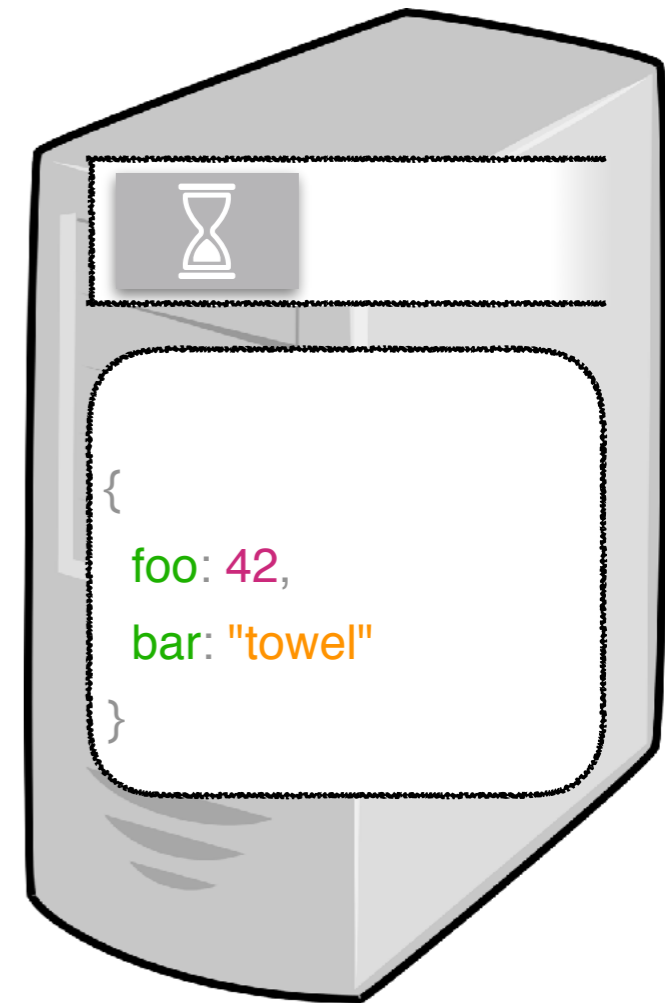


# DSLabs Programming Model

- A distributed system consists of a set of nodes which communicate over an asynchronous network, working together to run a protocol.
- Nodes are I/O automata; they run as single-threaded event loops.
- Nodes are split between client and server nodes.

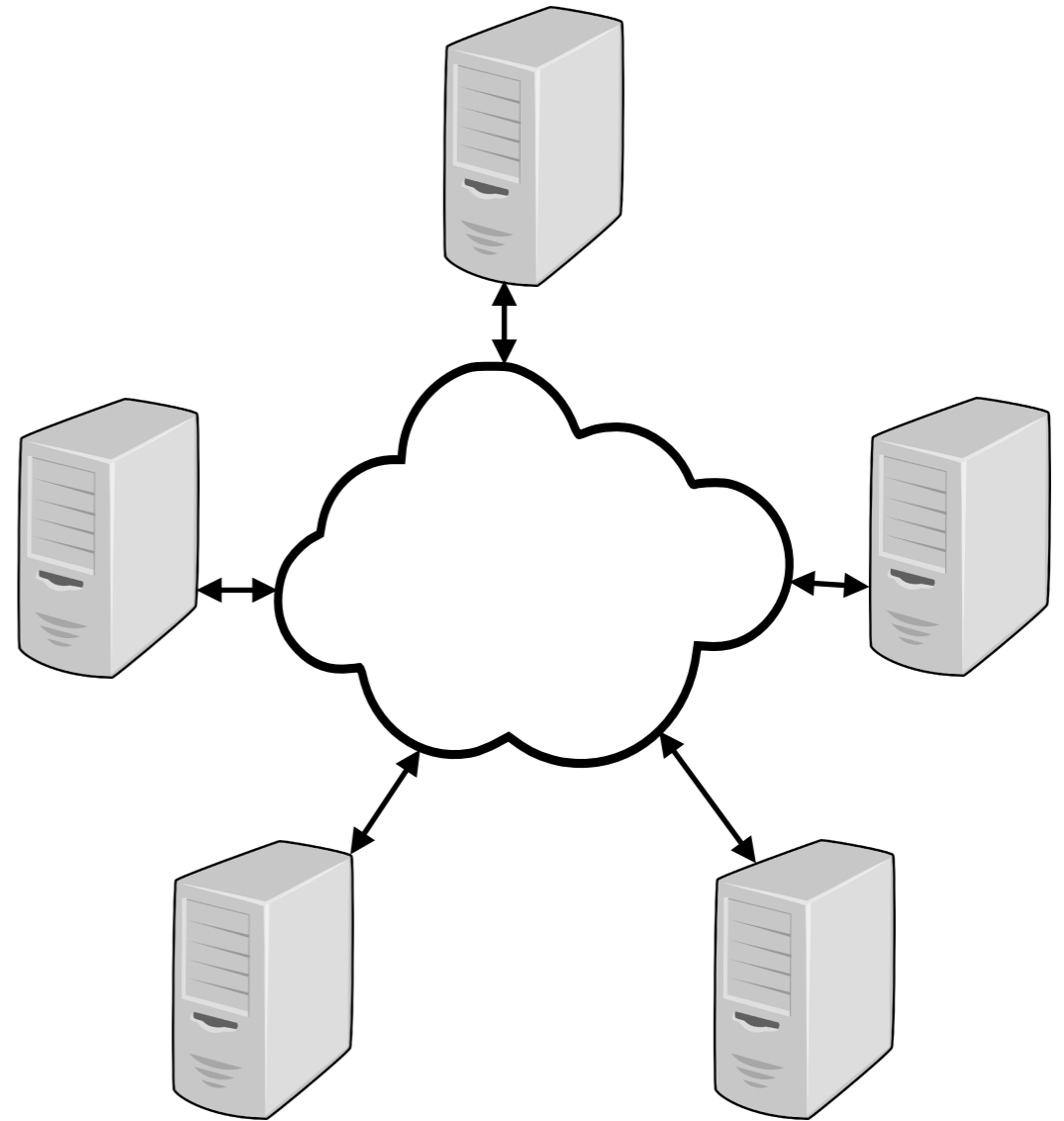


```
1: init()
2: loop
3:   e <- rcv_timer() ||
      rcv_msg()
4:   update_state(e)
5:   send_msgs()
6:   set_timers()
7: endloop
```



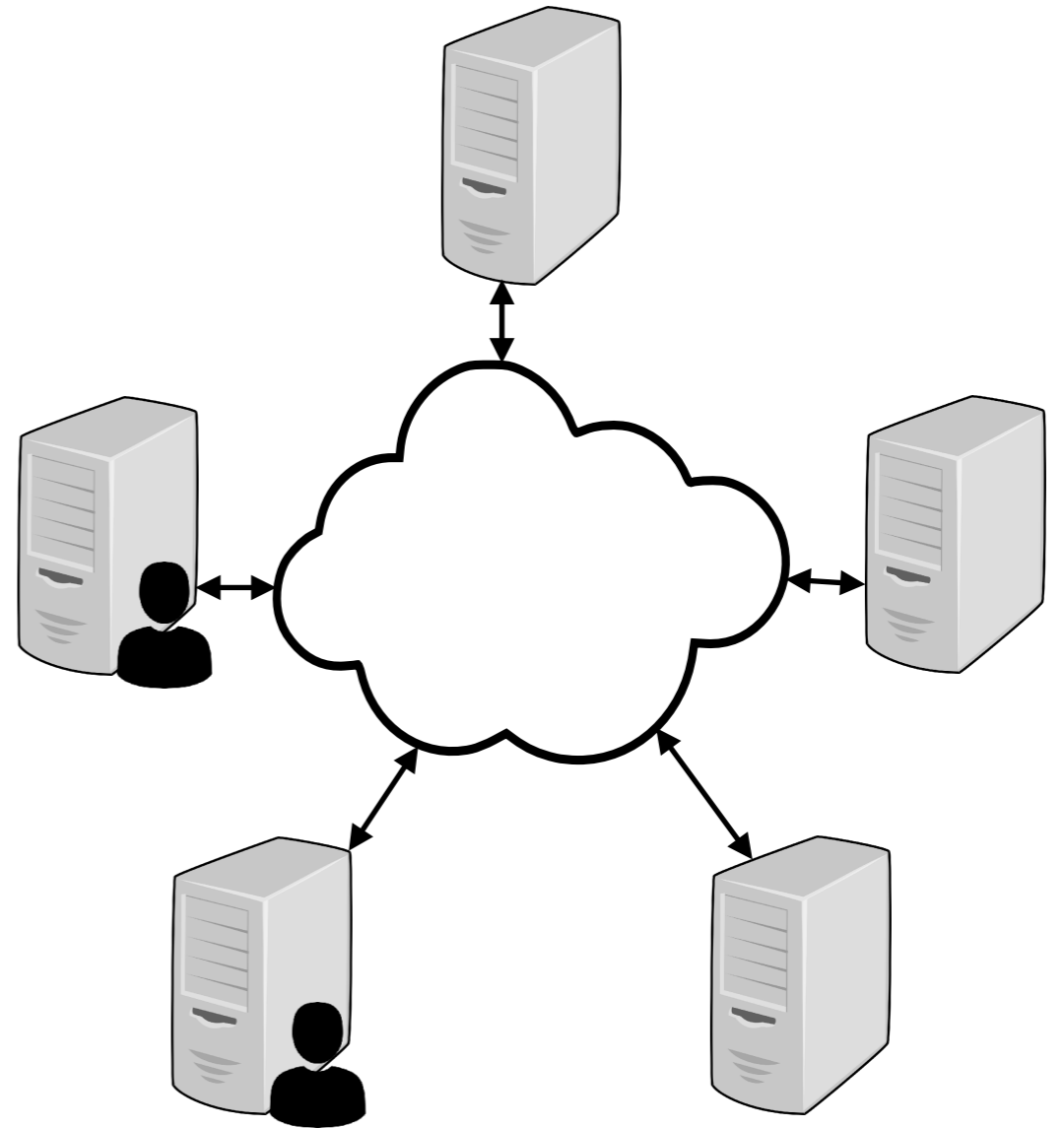
# DSLabs Programming Model

- A distributed system consists of a set of nodes which communicate over an asynchronous network, working together to run a protocol.
- Nodes are I/O automata; they run as single-threaded event loops.
- Nodes are split between client and server nodes.



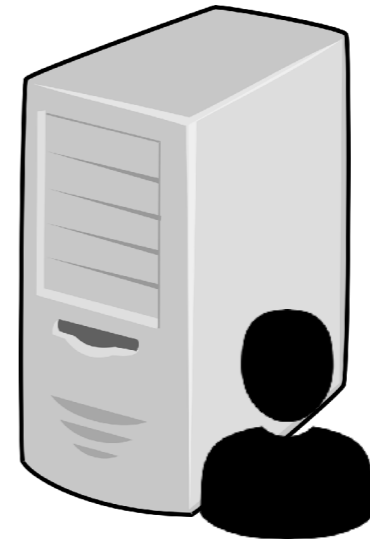
# DSLabs Programming Model

- A distributed system consists of a set of nodes which communicate over an asynchronous network, working together to run a protocol.
- Nodes are I/O automata; they run as single-threaded event loops.
- Nodes are split between client and server nodes.



# DSLabs Programming Model

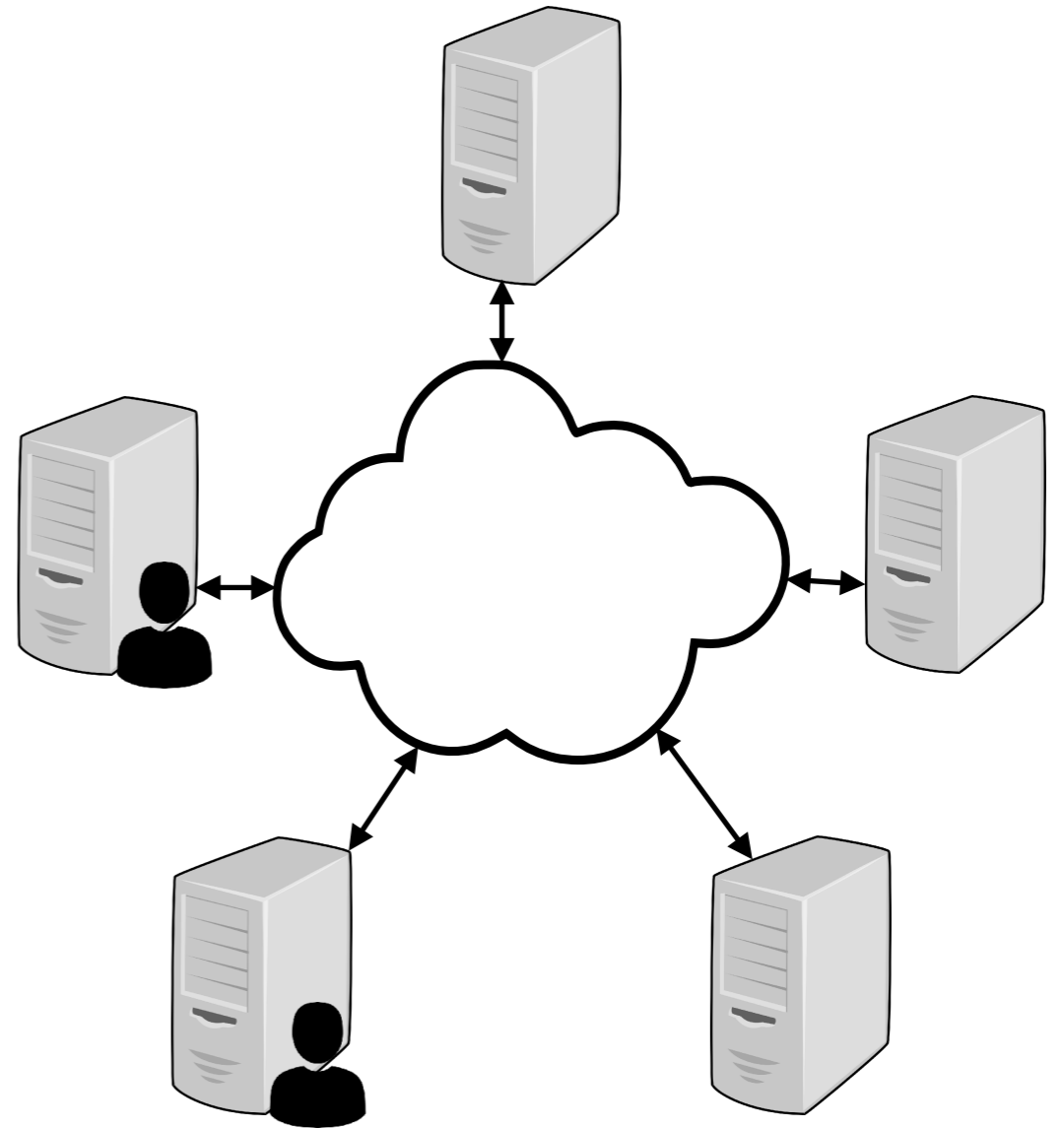
- A distributed system consists of a set of nodes which communicate over an asynchronous network, working together to run a protocol.
- Nodes are I/O automata; they run as single-threaded event loops.
- Nodes are split between client and server nodes.



```
interface Client {  
  
    void sendCommand(Command command);  
  
    boolean hasResult();  
  
    Result getResult();  
  
}
```

# DSLabs Programming Model

- A distributed system consists of a set of nodes which communicate over an asynchronous network, working together to run a protocol.
- Nodes are I/O automata; they run as single-threaded event loops.
- Nodes are split between client and server nodes.



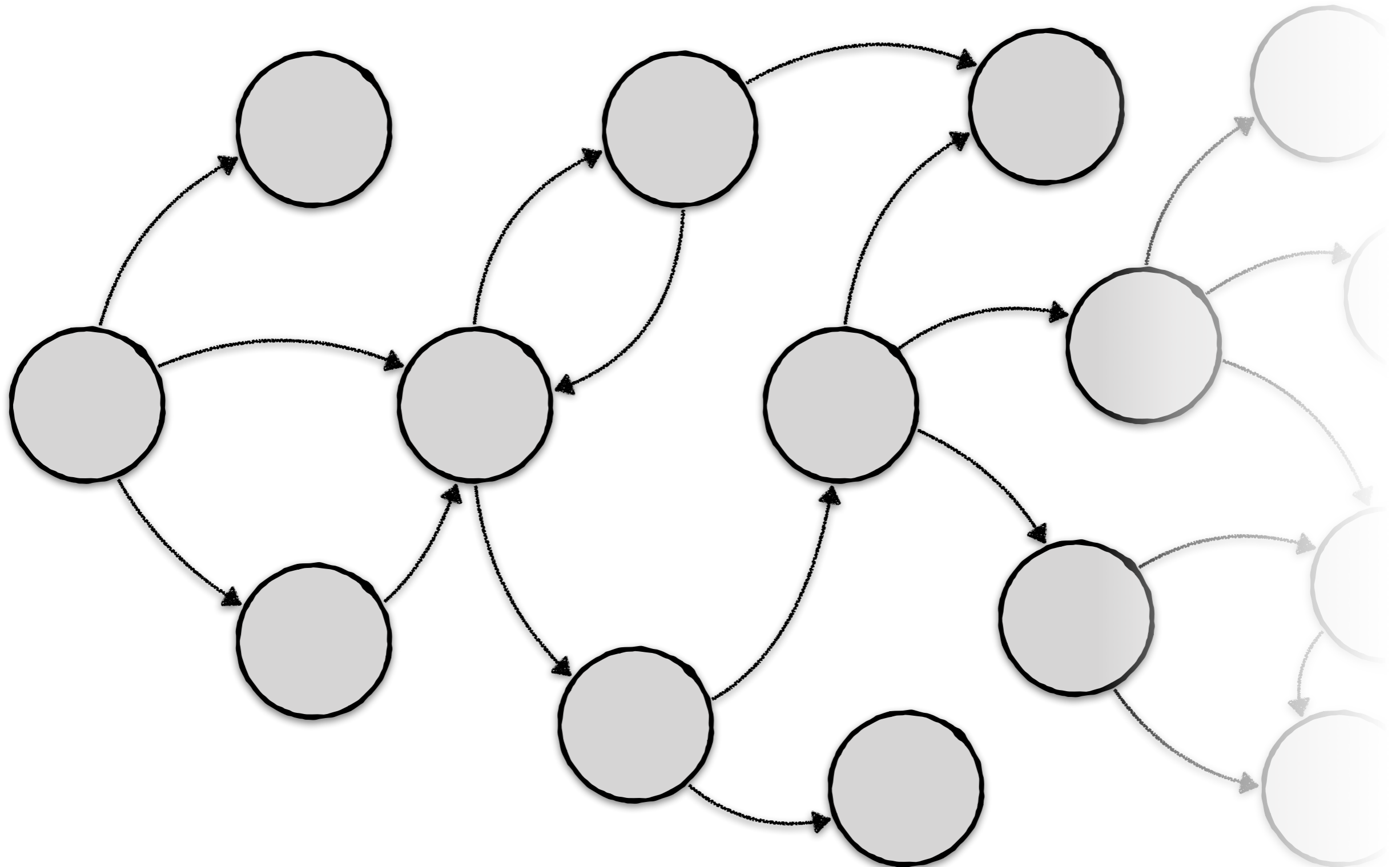
# Programming Model Benefits



- Isolates concurrency to coarsest possible granularity
- Lets students focus on distributed protocols, avoiding issues such as deadlock within a node
- Allows for model checking at the protocol level without significant modification or overhead

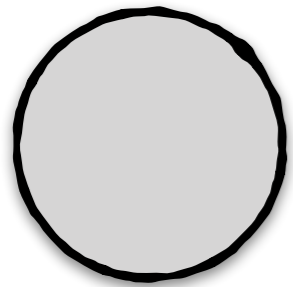


# Model Checking

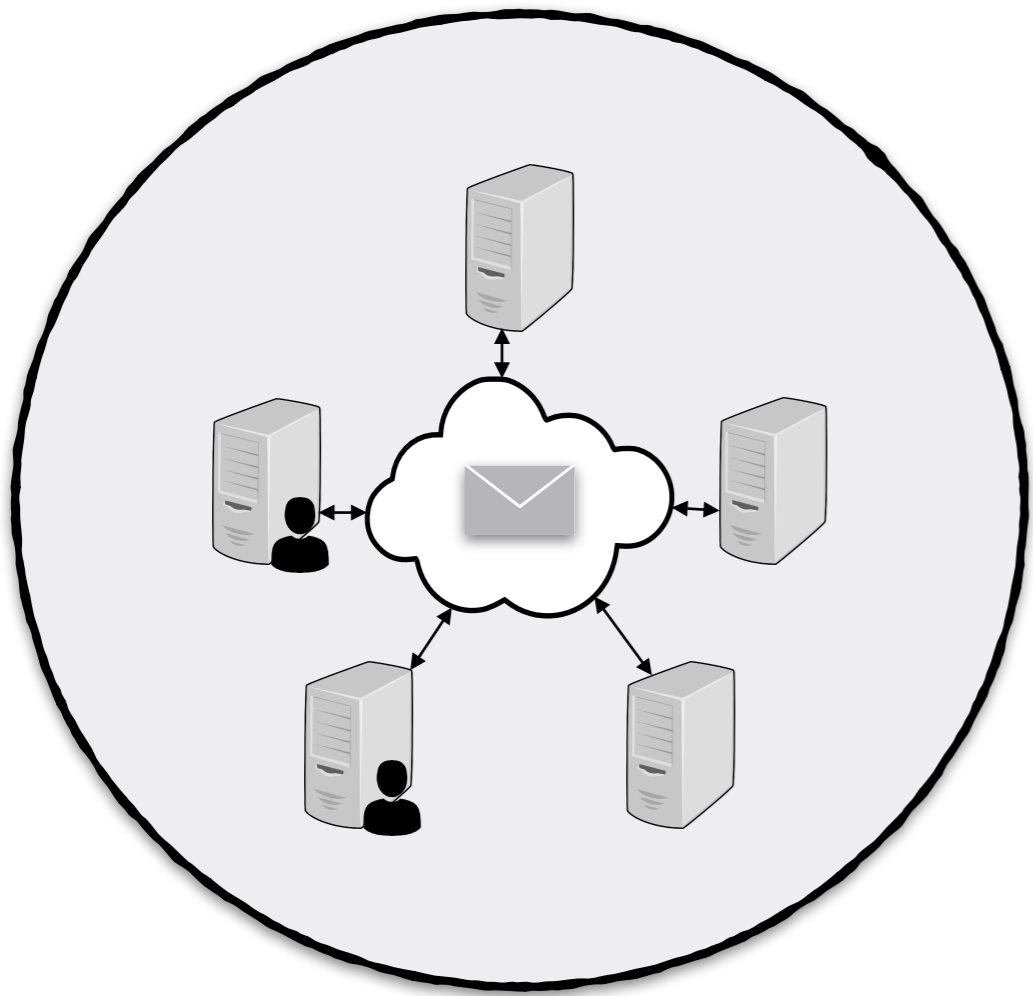


# Model Checking

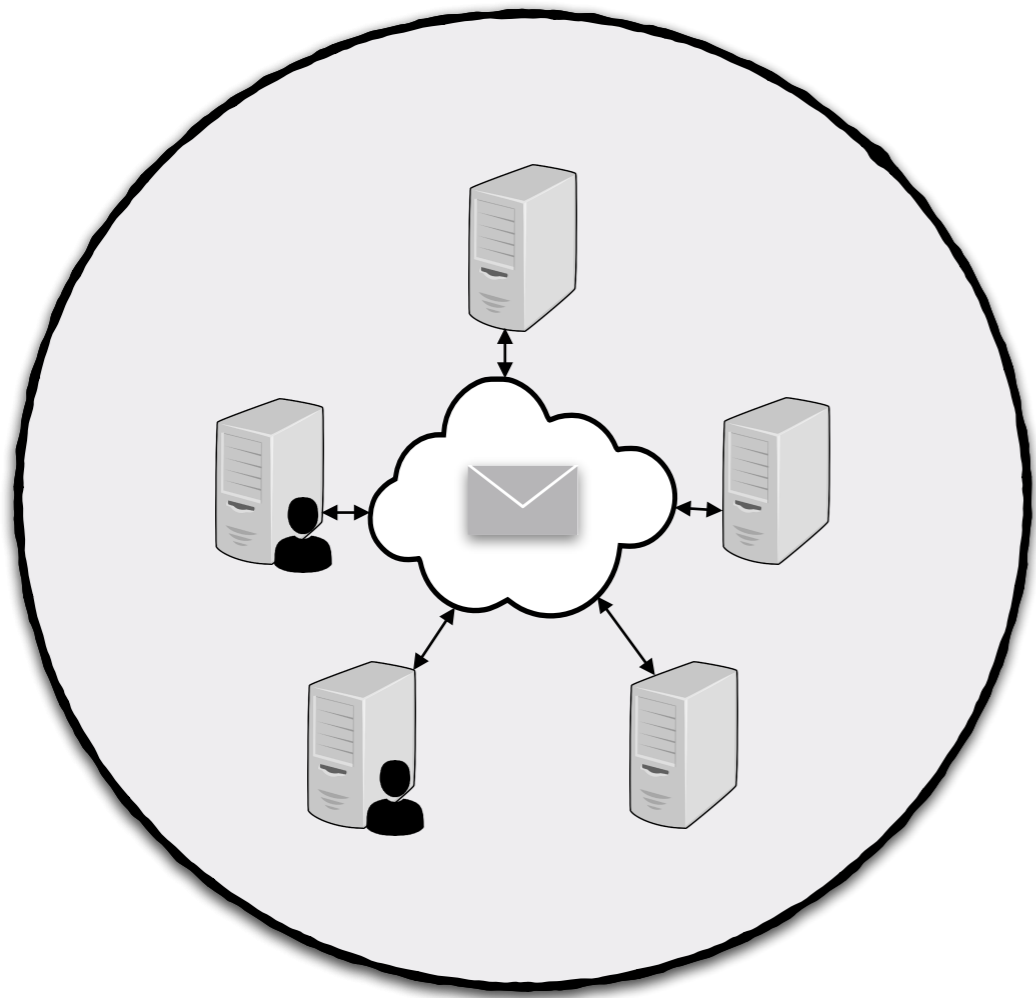
---



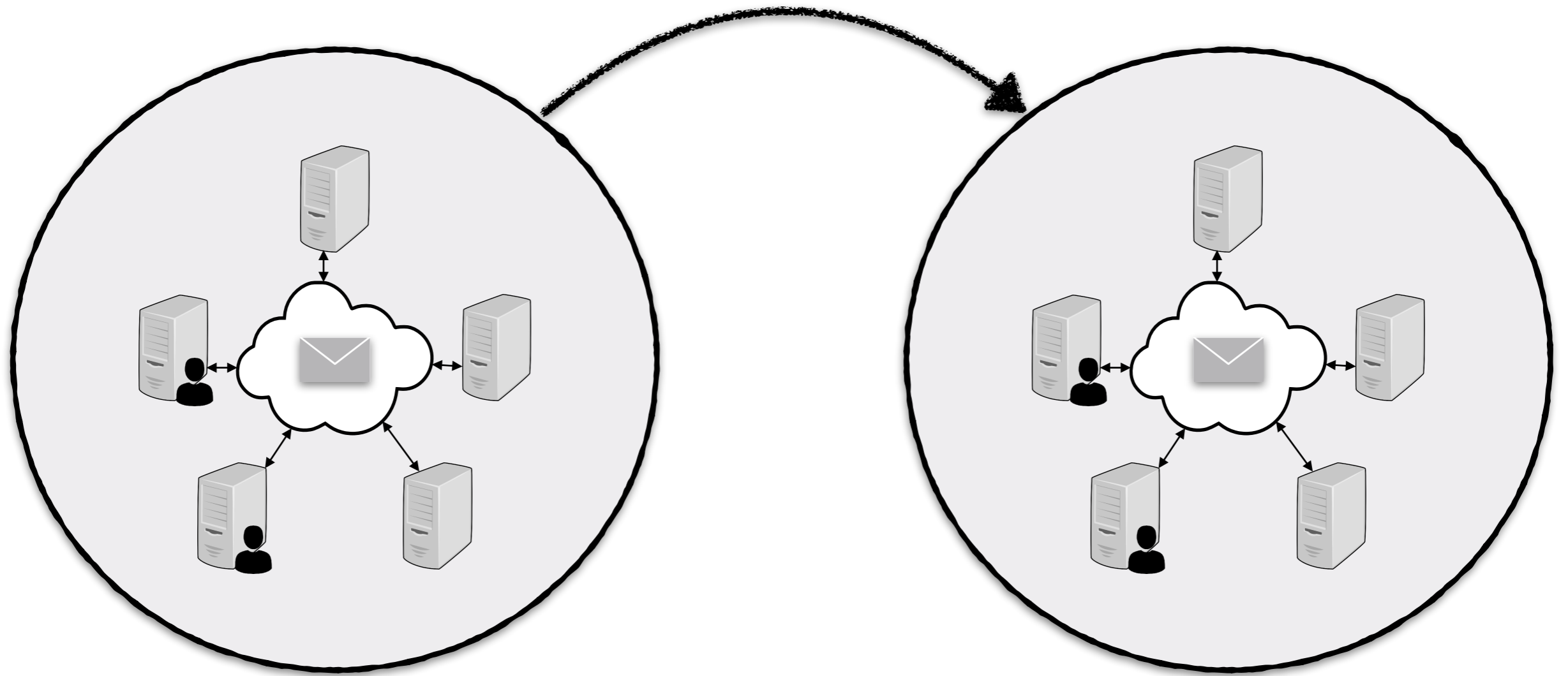
# Model Checking



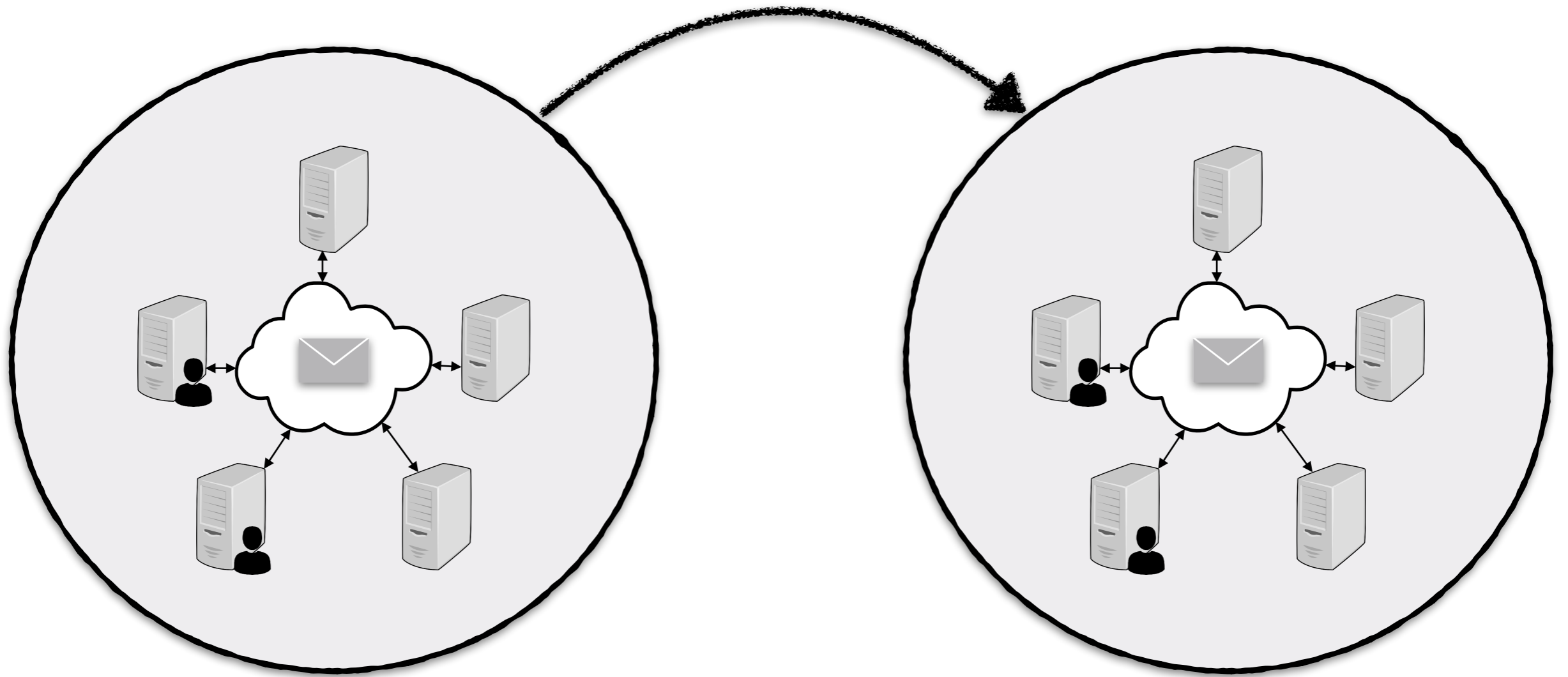
# Model Checking



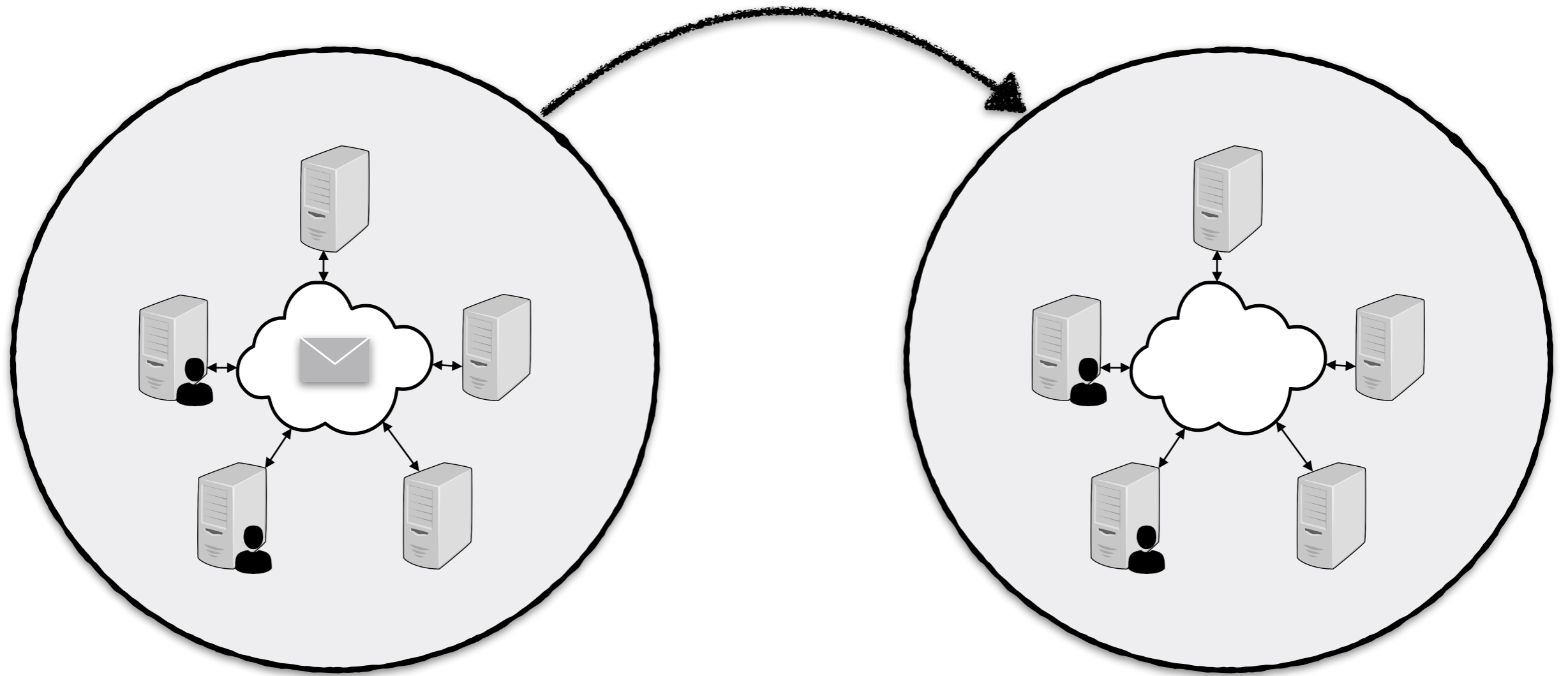
# Model Checking



# Model Checking

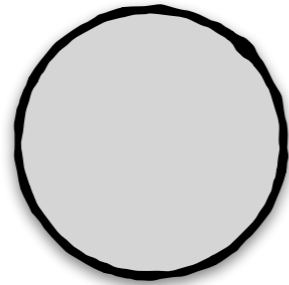
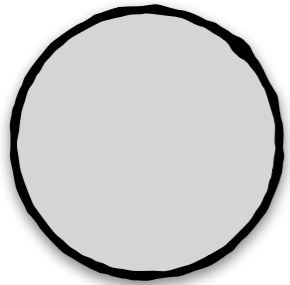


# Model Checking



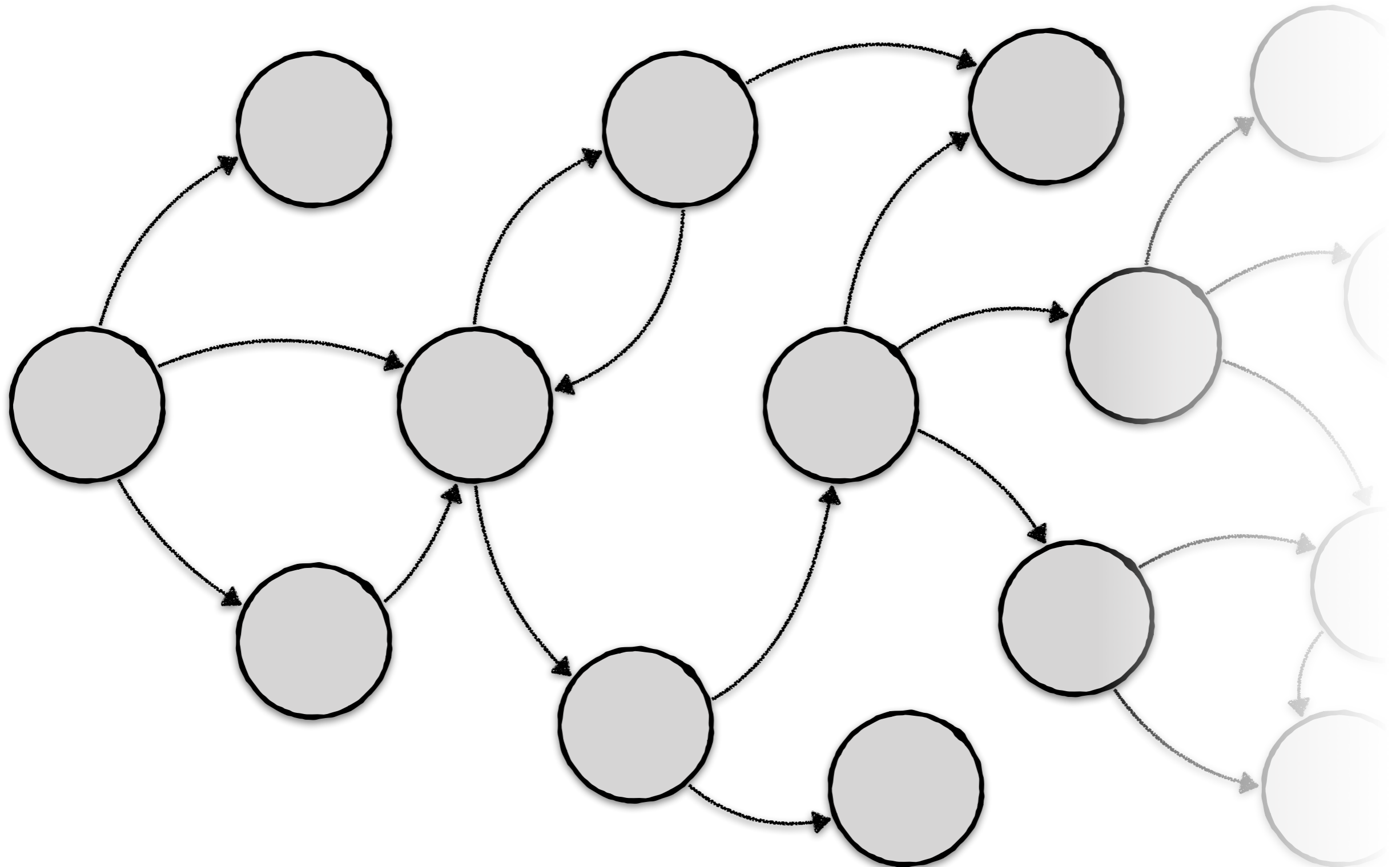
# Model Checking

---

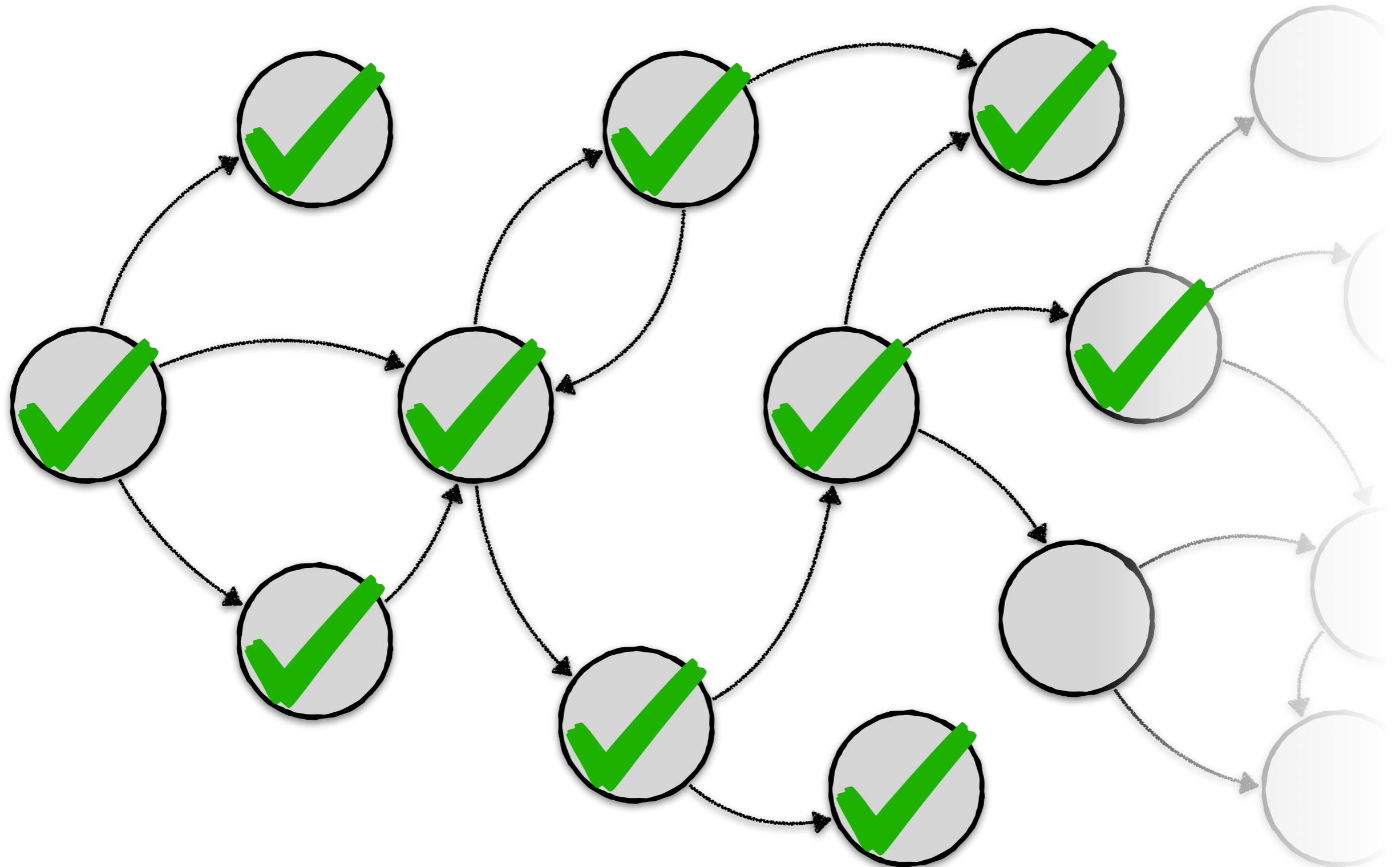




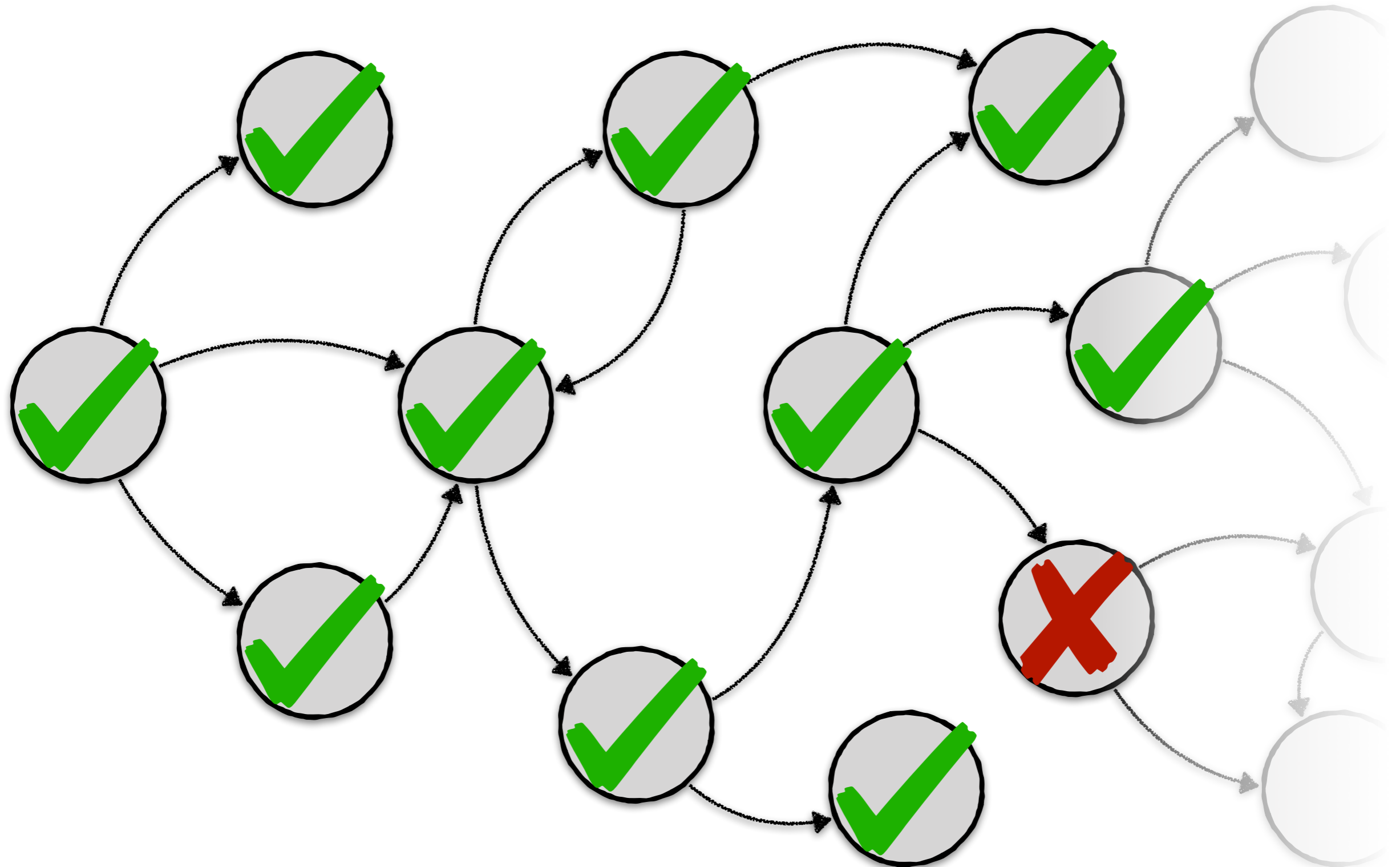
# Model Checking



# Model Checking



# Model Checking



# Outline

1. The DSLabs programming model
2. Model checking strategies and optimizations
3. Understandability and Oddity visual debugger
4. Experiences

How can the model checker evaluate states of student implementations?

What should the interface be between the tests and student implementations?

# Black-Box

- Tests can check end-to-end properties, nothing else
- Allows maximum flexibility during implementation
- Doesn't allow checking more complicated properties, optimizations

## Black-Box

- Tests can check end-to-end properties, nothing else
- Allows maximum flexibility during implementation
- Doesn't allow checking more complicated properties, optimizations

## White-box

- Message formats, and even internal data structures defined for students
- Allows for thorough, incremental checking
- Solves design challenges for students
- Couples tests to implementation

## Black-Box

- Tests can check end-to-end properties, nothing else
- Allows maximum flexibility during implementation
- Doesn't allow checking more complicated properties, optimizations

## White-box

- Message formats, and even internal data structures defined for students
- Allows for thorough, incremental checking
- Solves design challenges for students
- Couples tests to implementation



## Black-Box

- Tests can check end-to-end properties, nothing else
- Allows maximum flexibility during implementation
- Doesn't allow checking more complicated properties, optimizations

## Gray-Box

- Students implement limited, informational interface
- Allows enough insight into state for thorough checking
- Leaves most design decisions to students

## White-box

- Message formats, and even internal data structures defined for students
- Allows for thorough, incremental checking
- Solves design challenges for students
- Couples tests to implementation

## Black-Box

- Tests can check end-to-end properties, nothing else
- Allows maximum flexibility during implementation
- Doesn't allow checking more complicated properties, optimizations

## Gray-Box

- Students implement limited, informational interface
- Allows enough insight into state for thorough checking
- Leaves most design decisions to students

## White-box

- Message formats, and even internal data structures defined for students
- Allows for thorough, incremental checking
- Solves design challenges for students
- Couples tests to implementation

# Improving Model Checking Performance, Reliability



Model checking faces state-space explosion problem.

Strategies:

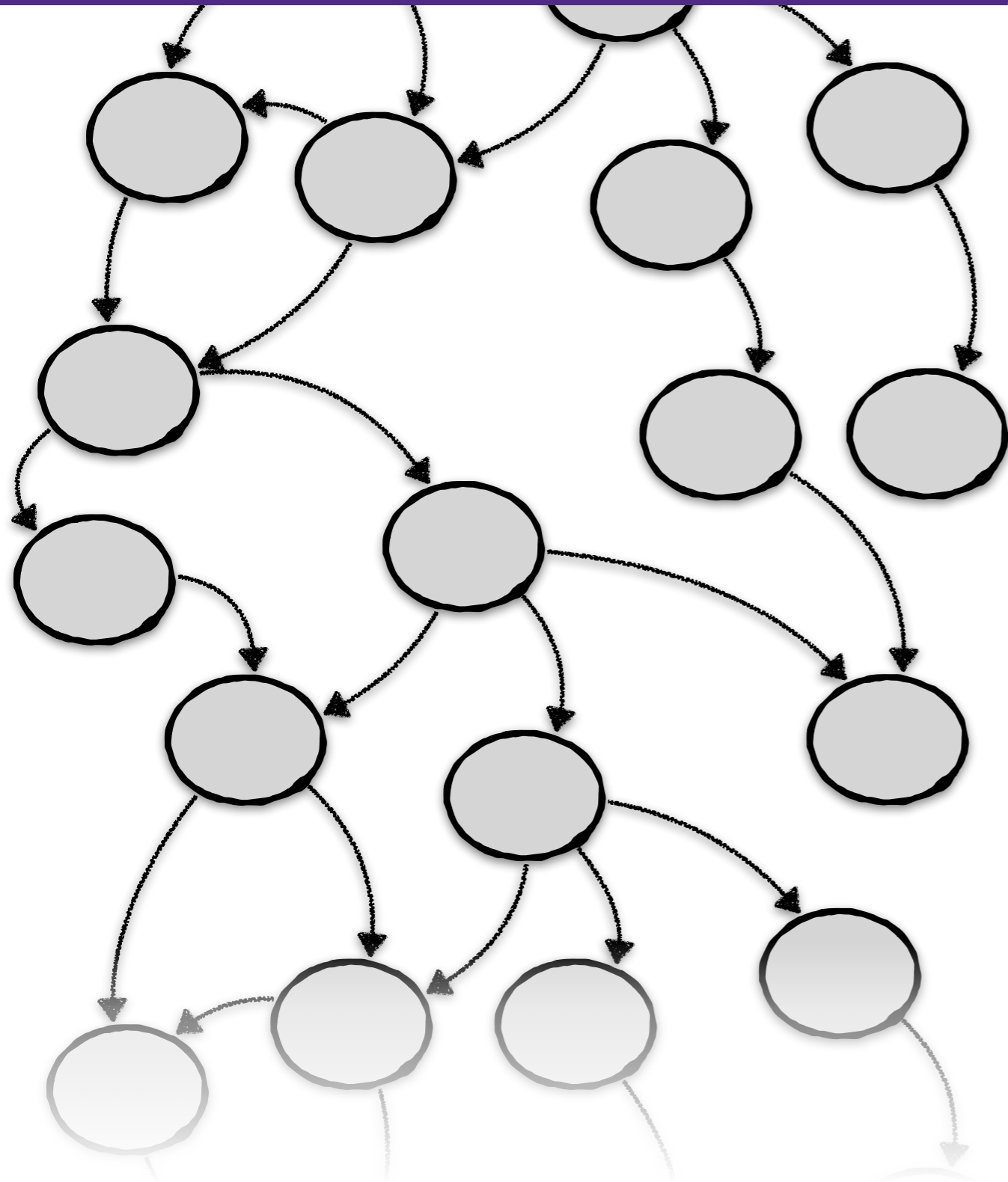
1. Pruning the search space
2. Punctuated search
3. Searching for progress





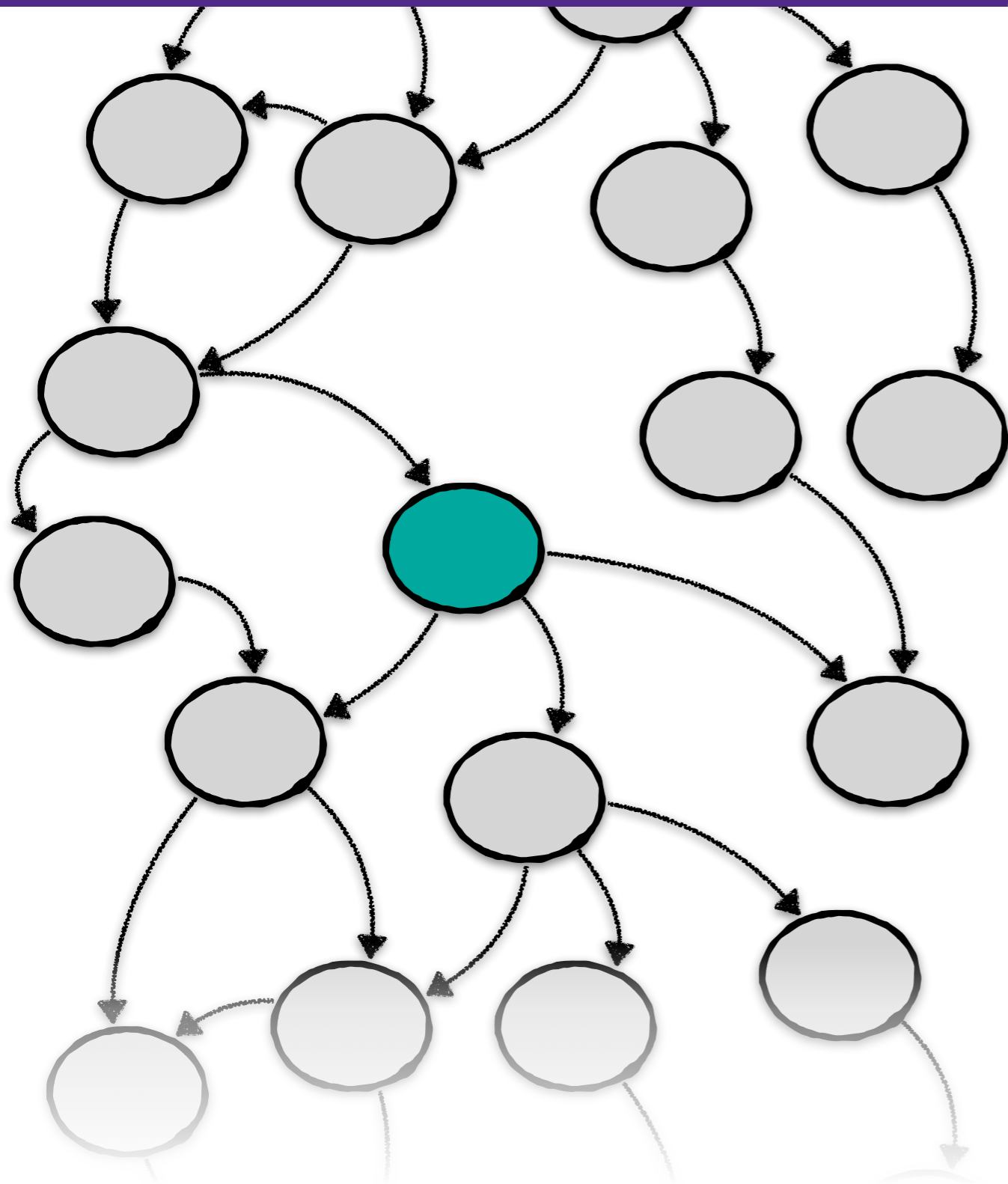


# Punctuated Search



- BFS is limited primarily by the depth to which it can search.
- First, the model checker finds a state matching an intermediate constraint. Then, resumes checking starting from the new state.
- Repeatable, allows for scripting complex searches

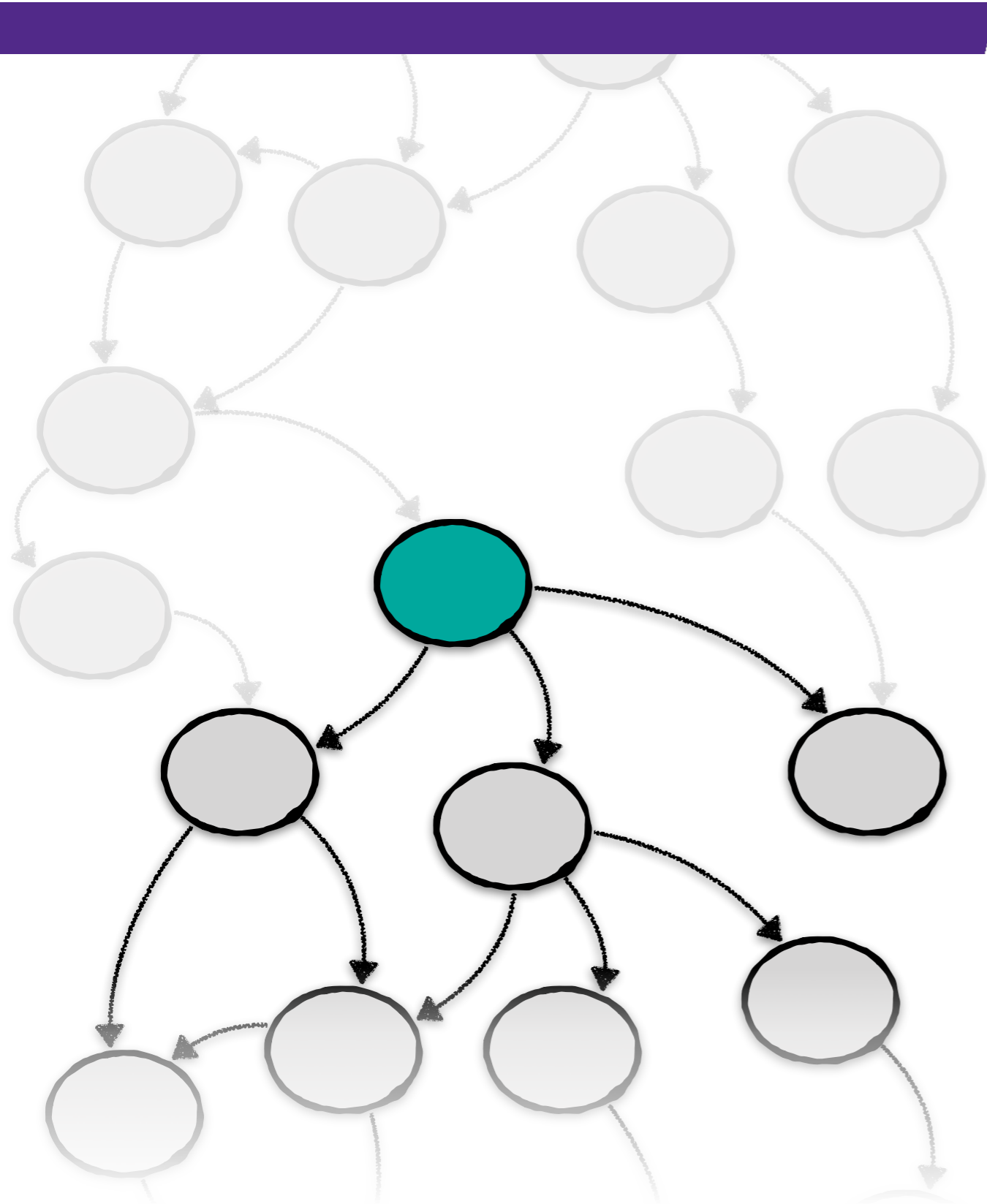
# Punctuated Search



- BFS is limited primarily by the depth to which it can search.
- First, the model checker finds a state matching an intermediate constraint. Then, resumes checking starting from the new state.
- Repeatable, allows for scripting complex searches

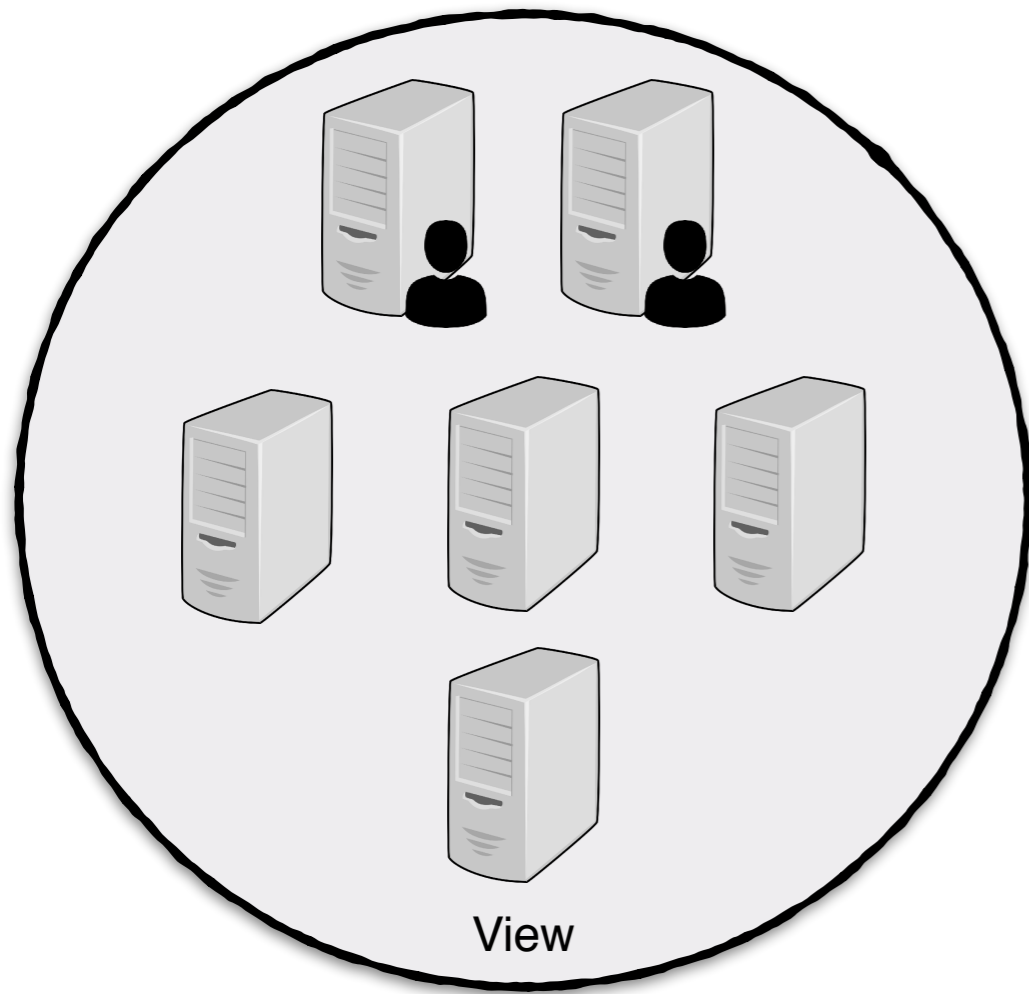


# Punctuated Search

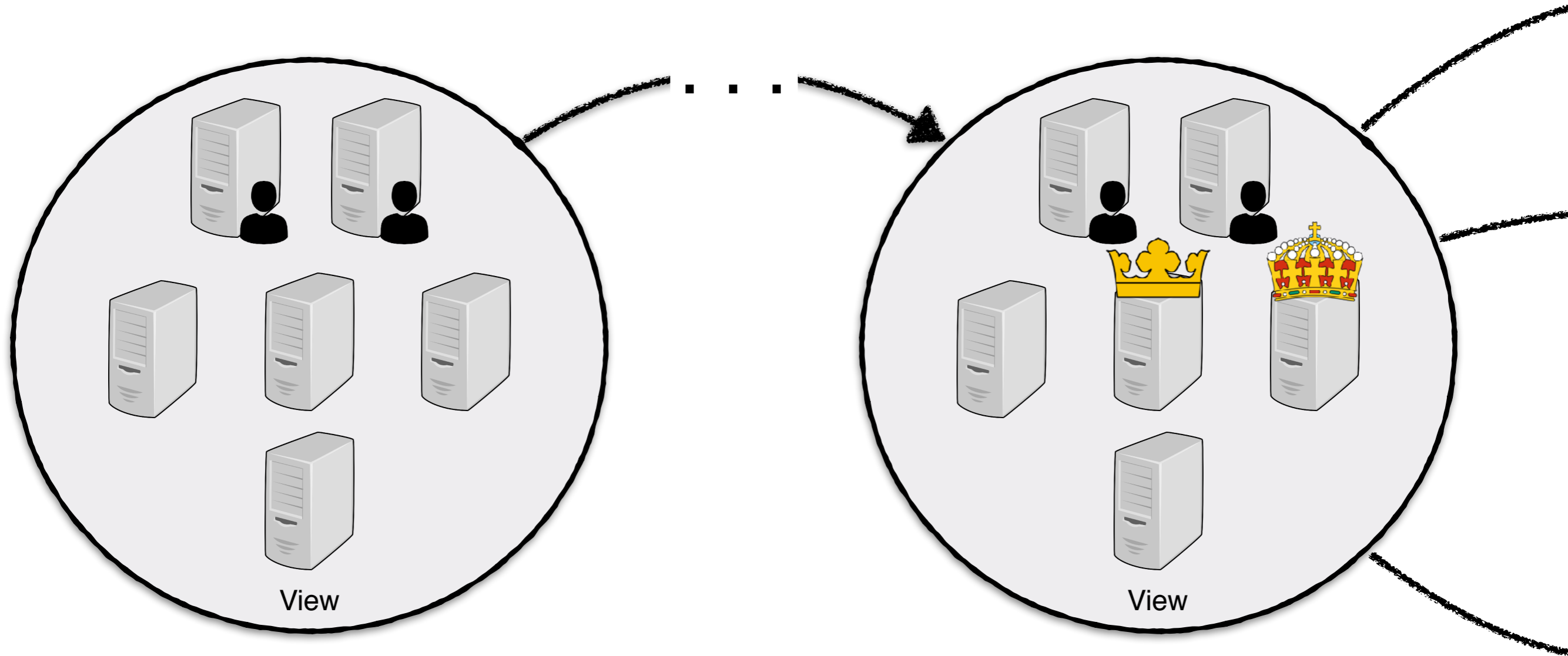


- BFS is limited primarily by the depth to which it can search.
- First, the model checker finds a state matching an intermediate constraint. Then, resumes checking starting from the new state.
- Repeatable, allows for scripting complex searches

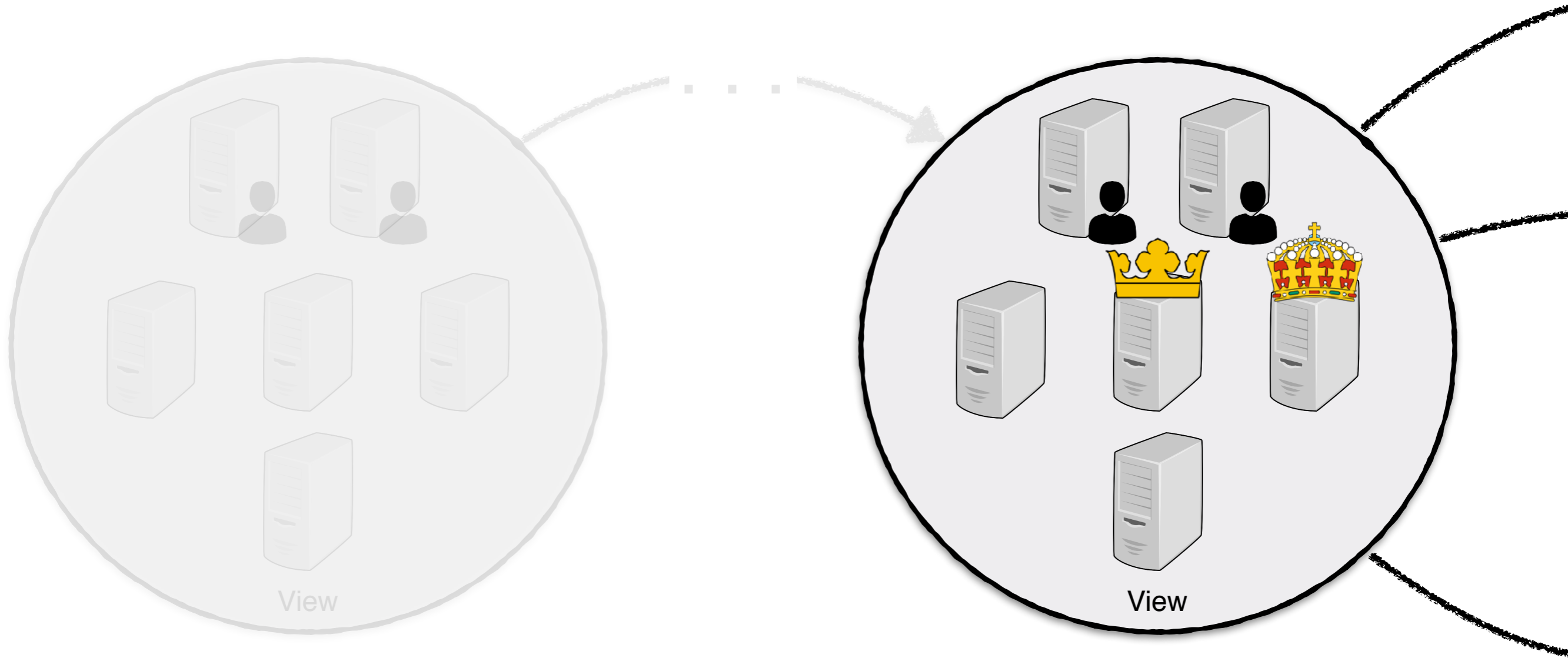
# Punctuated Search Example: Primary-backup



# Punctuated Search Example: Primary-backup



# Punctuated Search Example: Primary-backup



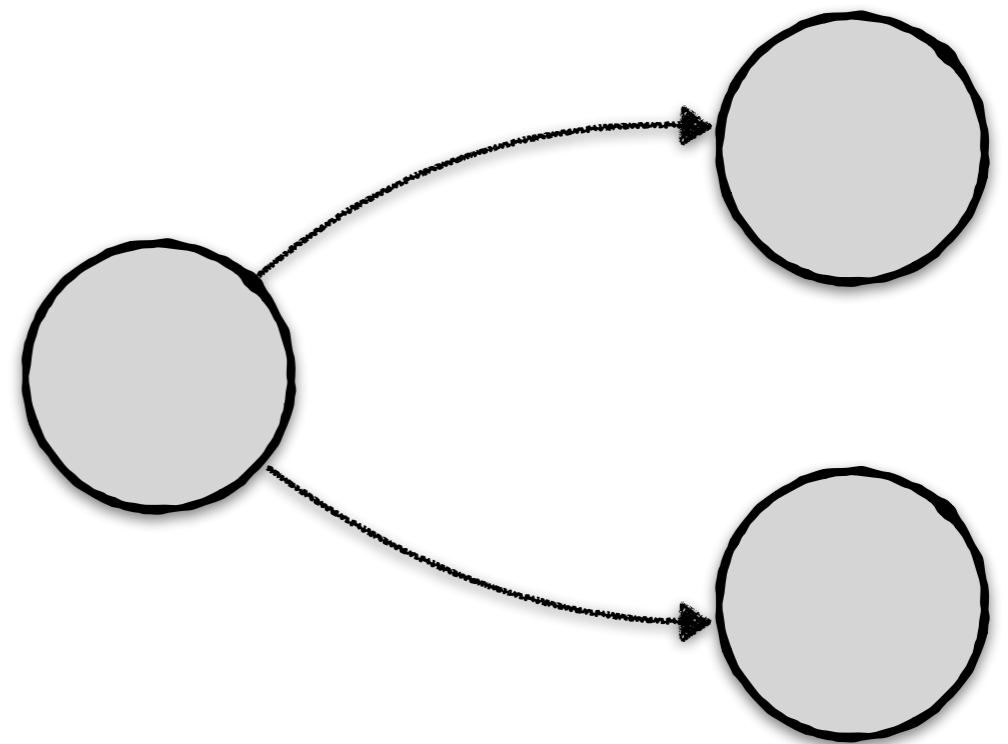
# Simplifying Implementation: Testing Determinism

- Key assumption: nodes are deterministic.
- Some sources of non-determinism are non-obvious.
- DSLabs has flag to check handler determinism, facilitating correct implementation.



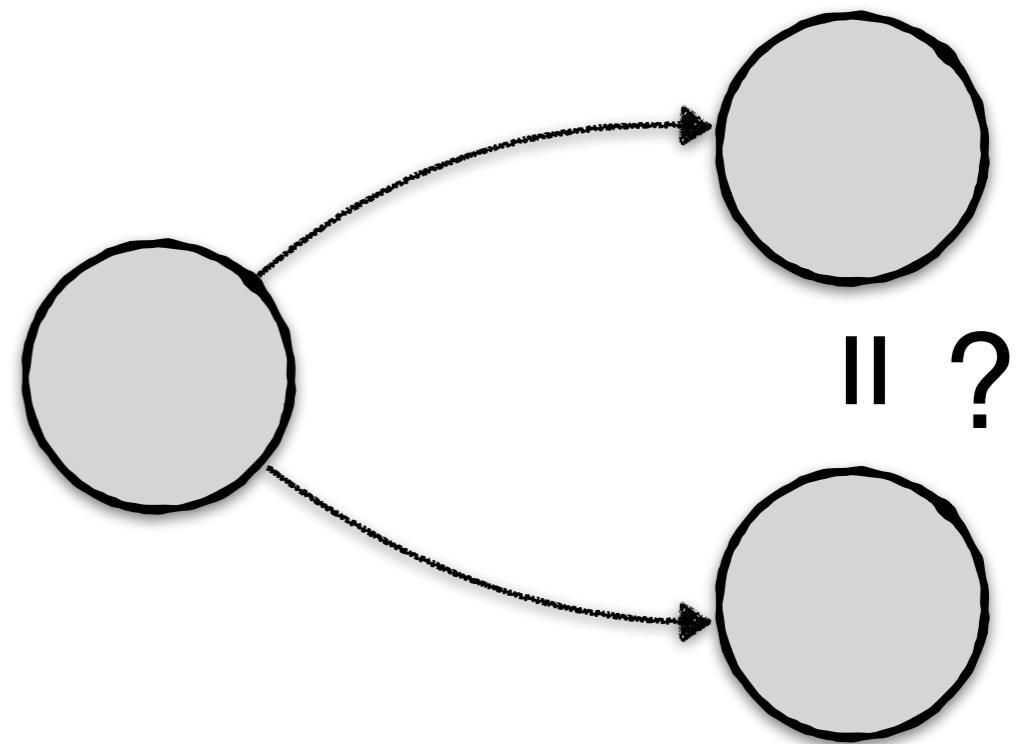
# Simplifying Implementation: Testing Determinism

- Key assumption: nodes are deterministic.
- Some sources of non-determinism are non-obvious.
- DSLabs has flag to check handler determinism, facilitating correct implementation.



# Simplifying Implementation: Testing Determinism

- Key assumption: nodes are deterministic.
- Some sources of non-determinism are non-obvious.
- DSLabs has flag to check handler determinism, facilitating correct implementation.



# Designing Systems for Model Checking



- Performance of model checking is implementation-dependent; runtime optimizations can reduce checkability.
- Our advice to students:
  - Favor simplicity.
  - Keep and send minimal state.
  - Ensure system can make progress with minimal steps.



# Outline

1. The DSLabs programming model
2. Model checking strategies and optimizations
3. Understandability and Oddity visual debugger
4. Experiences

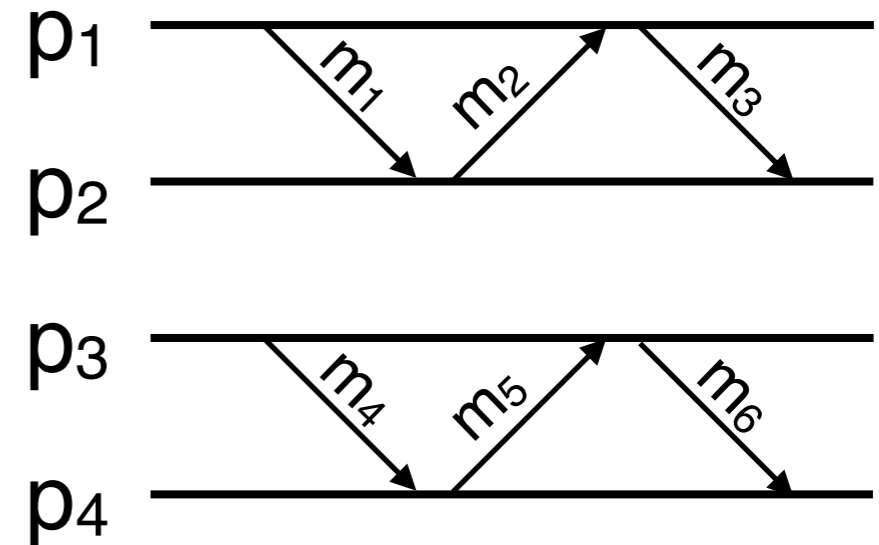
# Producing Understandable Traces

- A trace is a linearization of an execution returned by model checker, demonstrating invariant violation.
- BFS used by model checker could return any minimal length trace.
- DSLabs performs a depth-first topological sort of the event graph before returning traces to students



# Producing Understandable Traces

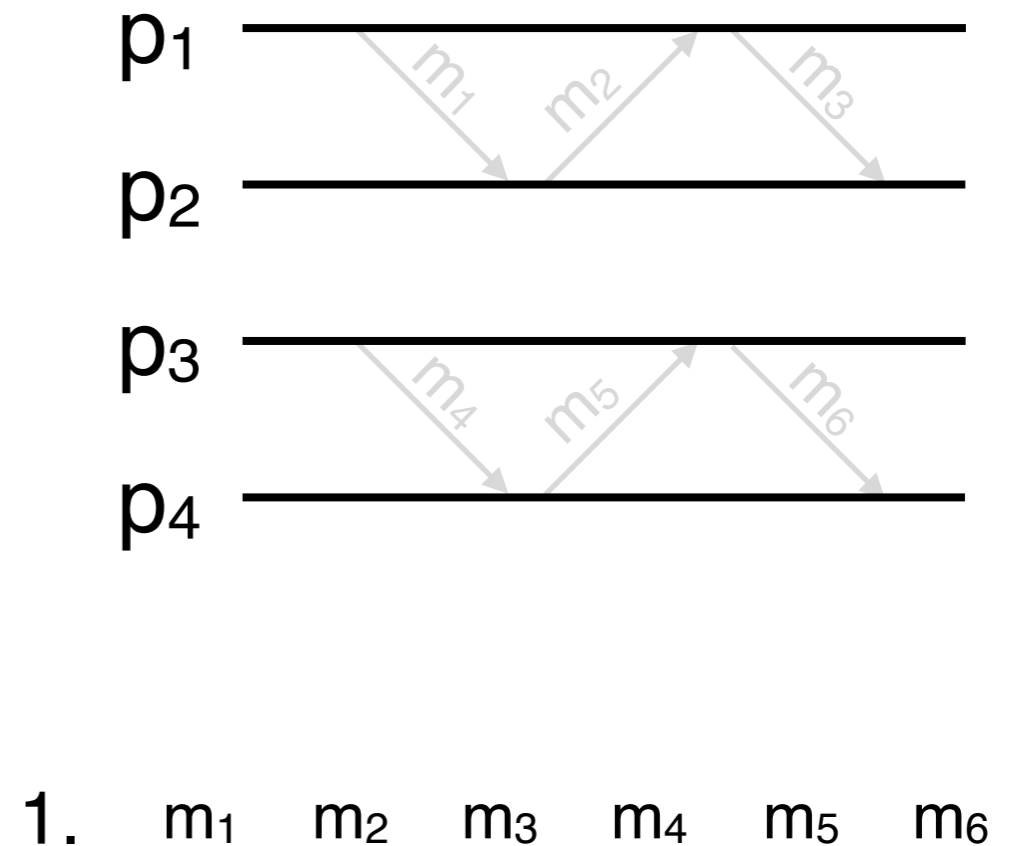
- A trace is a linearization of an execution returned by model checker, demonstrating invariant violation.
- BFS used by model checker could return any minimal length trace.
- DSLabs performs a depth-first topological sort of the event graph before returning traces to students



1. m<sub>1</sub> m<sub>2</sub> m<sub>3</sub> m<sub>4</sub> m<sub>5</sub> m<sub>6</sub>

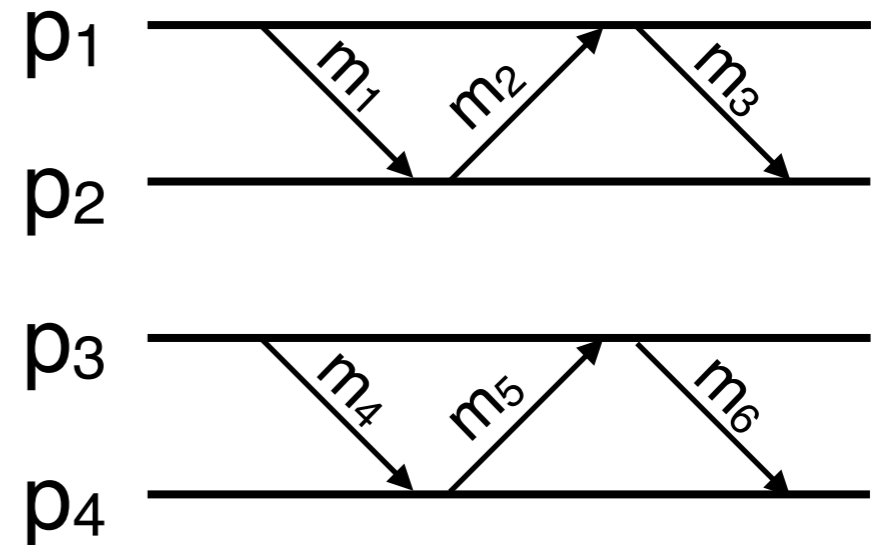
# Producing Understandable Traces

- A trace is a linearization of an execution returned by model checker, demonstrating invariant violation.
- BFS used by model checker could return any minimal length trace.
- DSLabs performs a depth-first topological sort of the event graph before returning traces to students



# Producing Understandable Traces

- A trace is a linearization of an execution returned by model checker, demonstrating invariant violation.
- BFS used by model checker could return any minimal length trace.
- DSLabs performs a depth-first topological sort of the event graph before returning traces to students

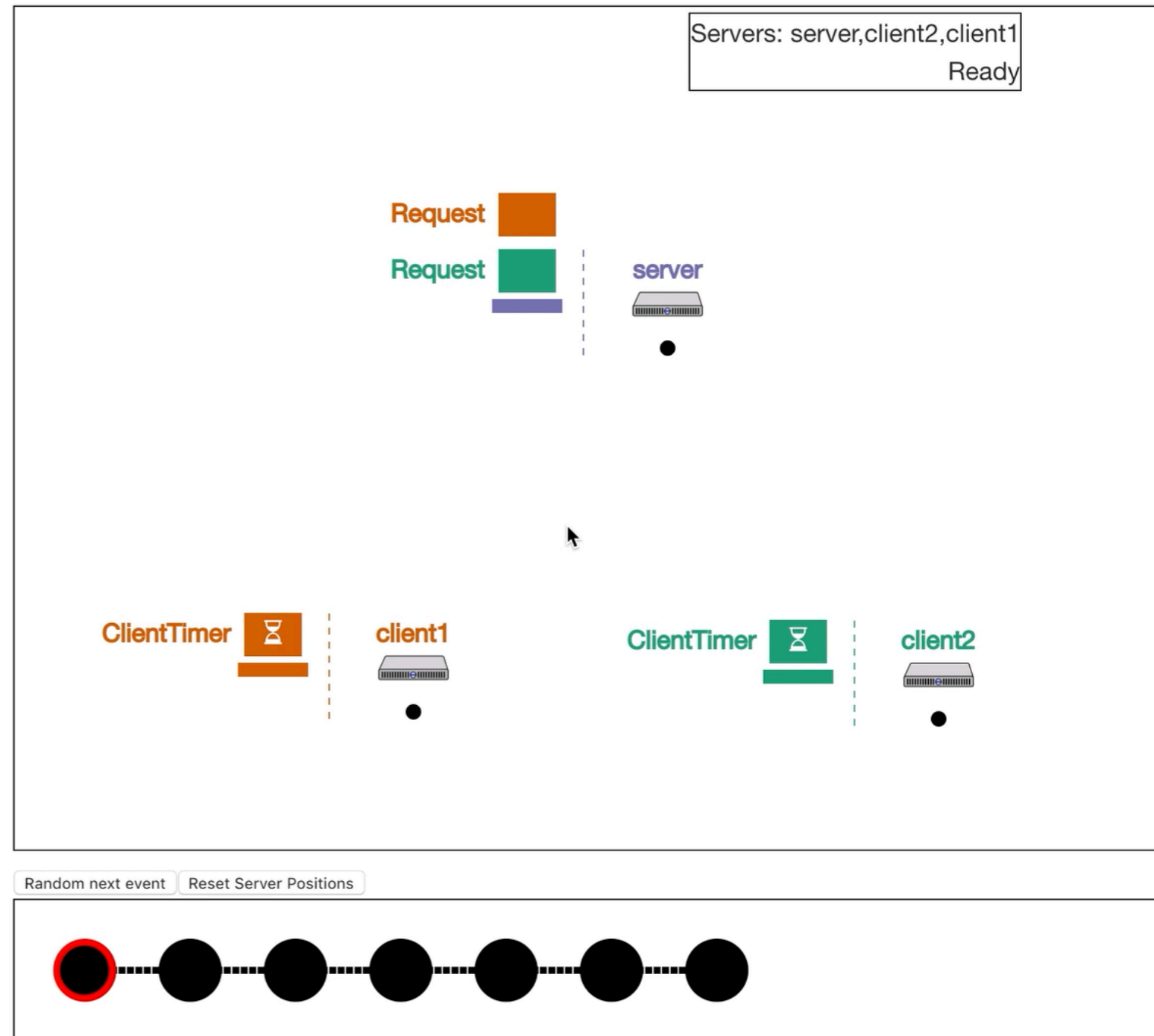


1. m<sub>1</sub> m<sub>2</sub> m<sub>3</sub> m<sub>4</sub> m<sub>5</sub> m<sub>6</sub>

2. m<sub>1</sub> m<sub>4</sub> m<sub>5</sub> m<sub>2</sub> m<sub>6</sub> m<sub>3</sub>

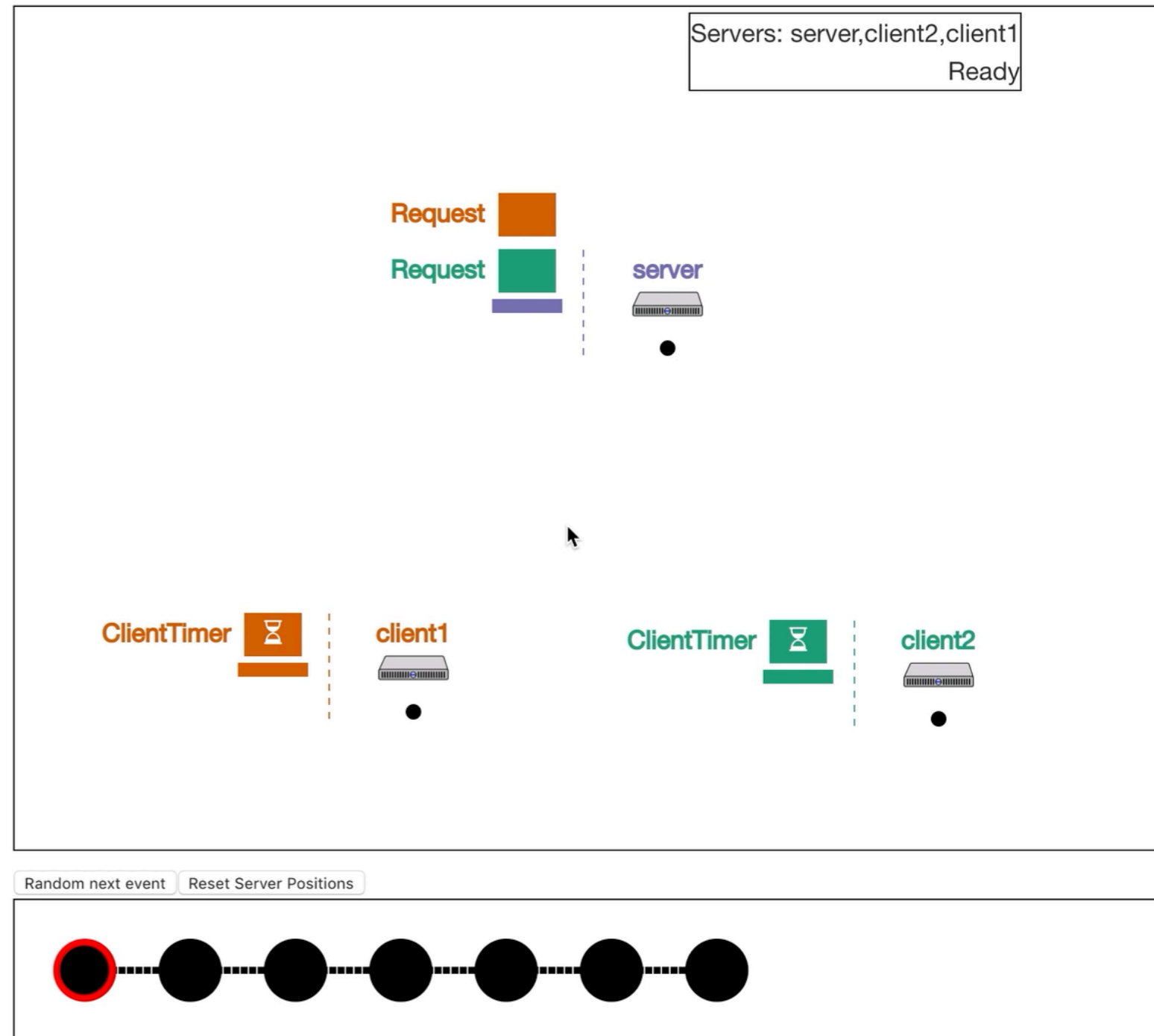
# Oddity

- Allows exploration from initial state or invariant-violating trace
- Lets students interactively explore states, examine messages and nodes
- Can "time-travel," explore alternate histories



# Oddity

- Allows exploration from initial state or invariant-violating trace
- Lets students interactively explore states, examine messages and nodes
- Can "time-travel," explore alternate histories



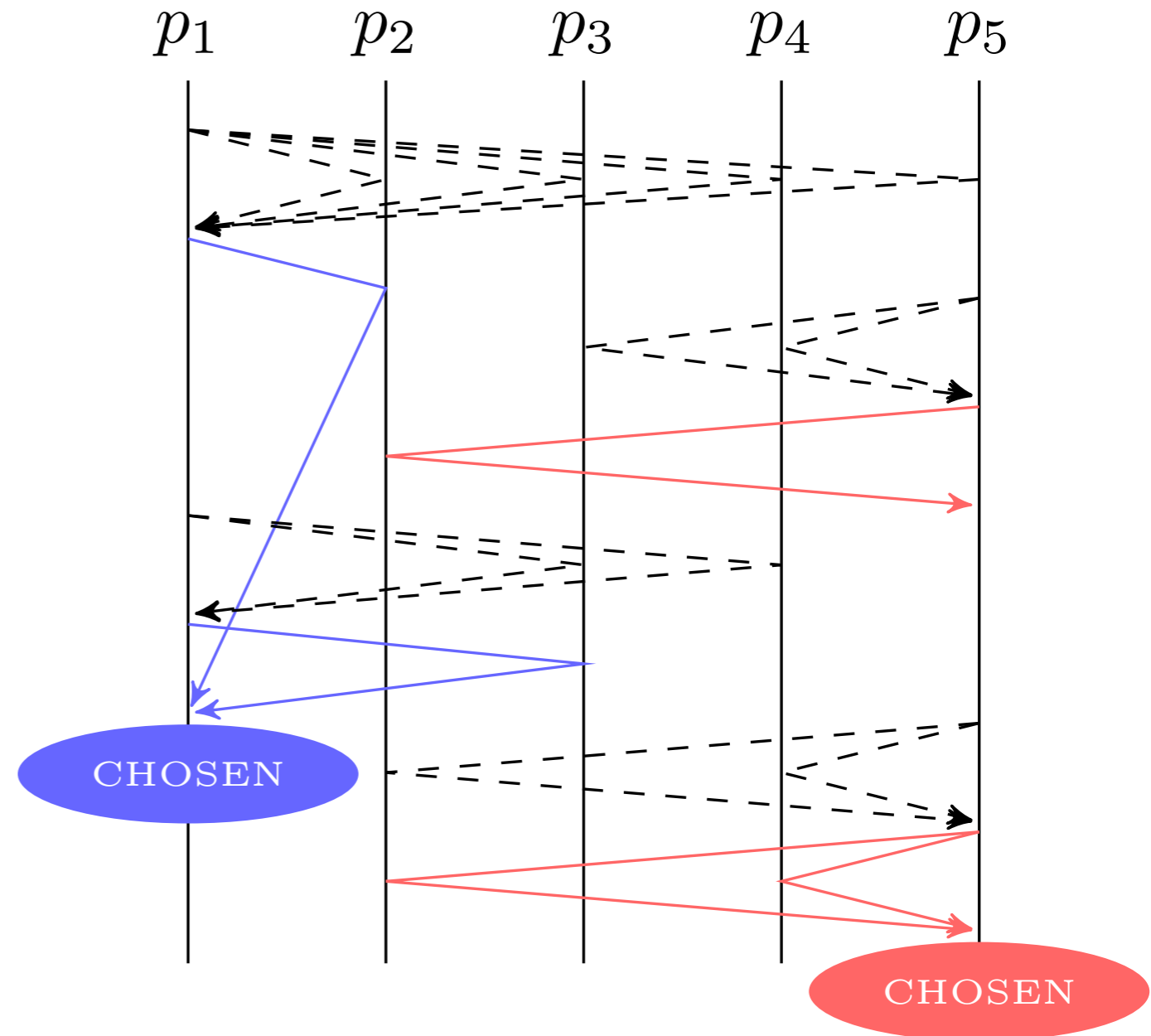
# Outline

1. The DSLabs programming model
2. Model checking strategies and optimizations
3. Understandability and Oddity visual debugger
4. **Experiences**

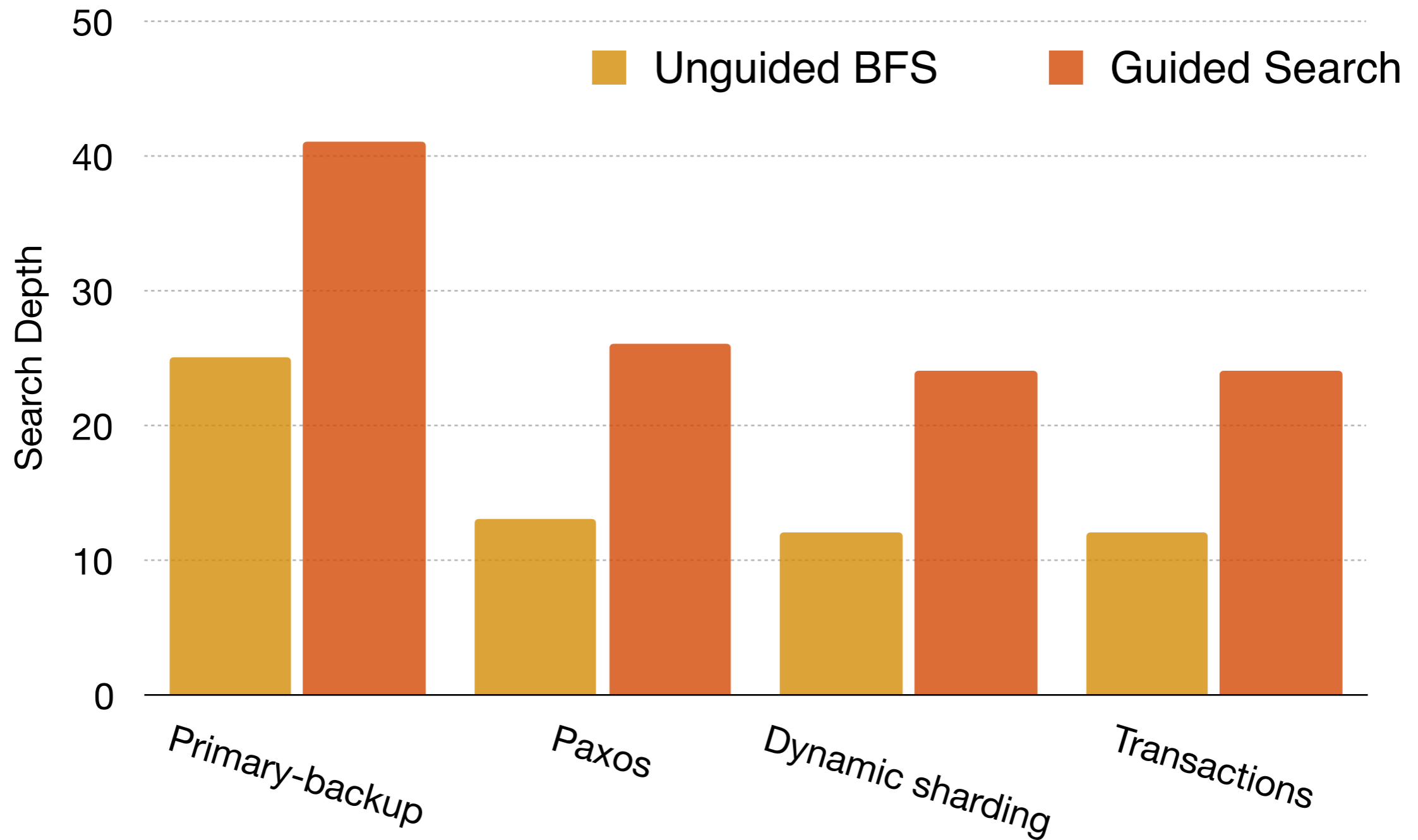


# Can Guided Searches Find Bugs?

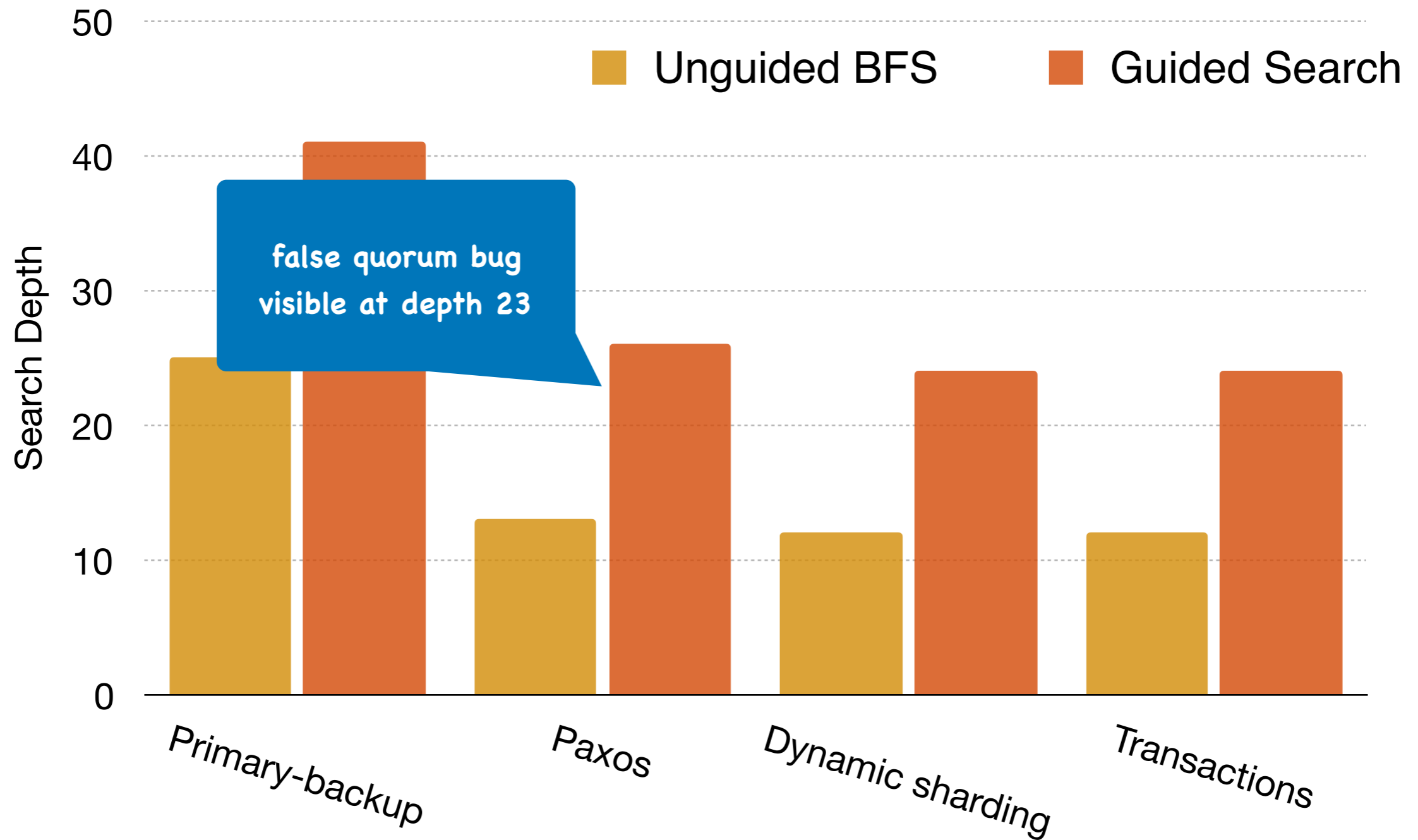
- Naïve BFS can't find the example false quorum bug.
- Random exploration takes an average of 12 hours.
- Guided search for this type of bug takes just 18 seconds.



# Can Guided Search Improve Model Checking Thoroughness?



# Can Guided Search Improve Model Checking Thoroughness?

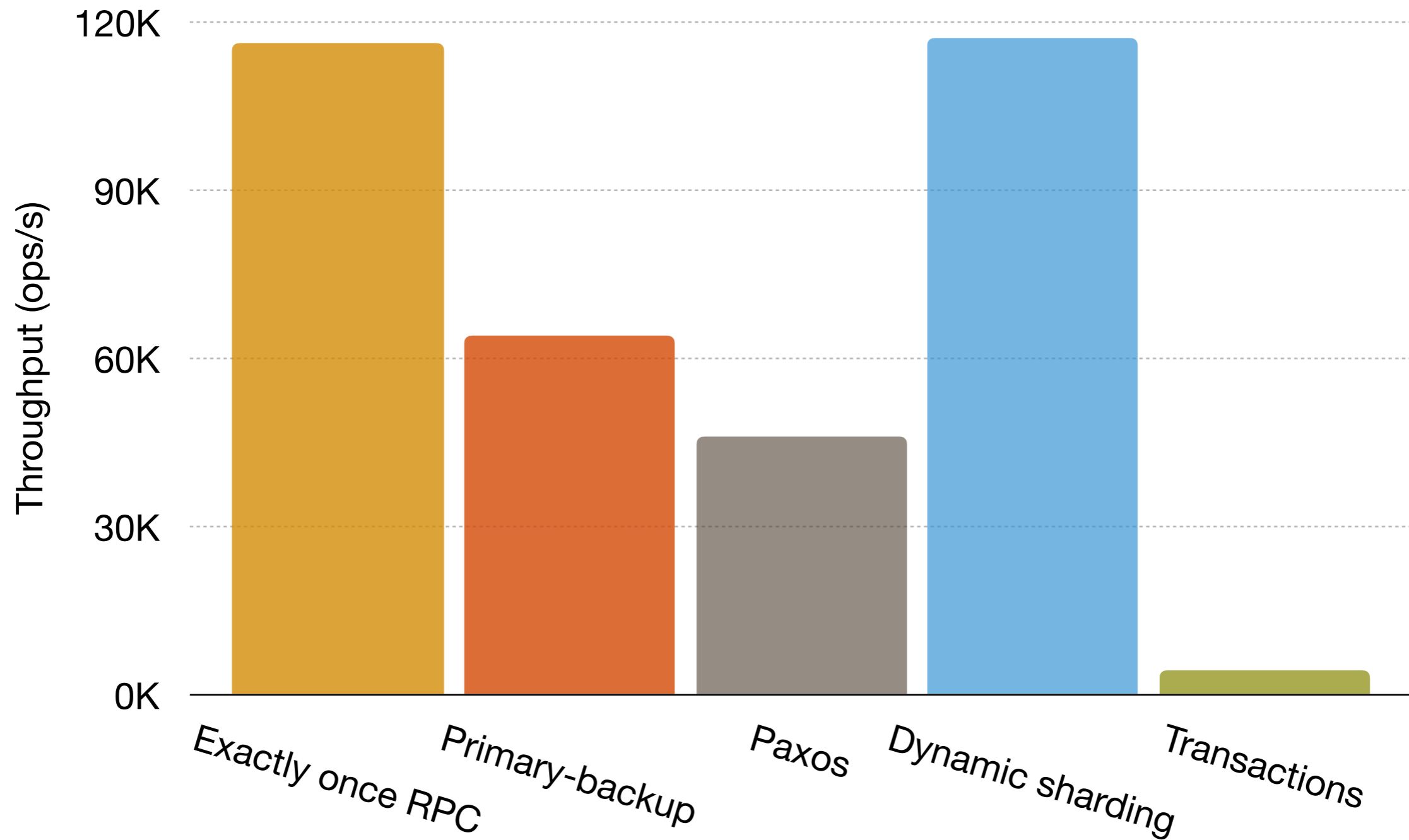


# Are Students Able to Debug Their Systems?

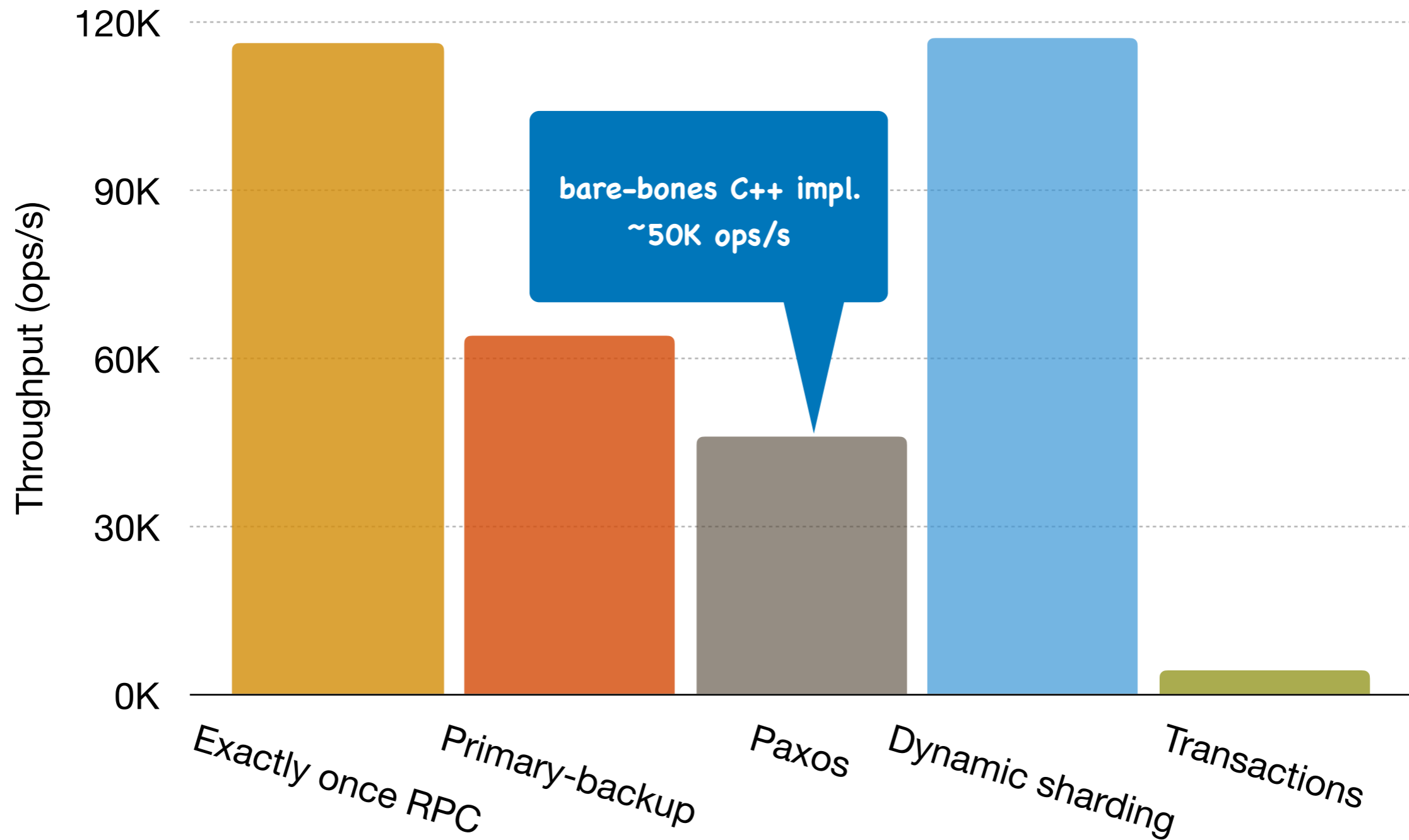


- Based on opt-in telemetry: over 150 invariant-violations examined with Oddity
- Almost all of these fixed before submission
- Only 25 submissions (across all assignments) found to violate invariants, 38 unable to pass searches for progress

# Can Students Build Runnable, Performant Systems?



# Can Students Build Runnable, Performant Systems?



# Does DSLabs Encourage "Distributed Thinking"?



- We want to encourage a distributed systems mindset: focus on invariants, rather than normal case.
- Model checking centers the distributed programming environment, finds "rare" errors.
- Visual debugger reinforces the programming model.

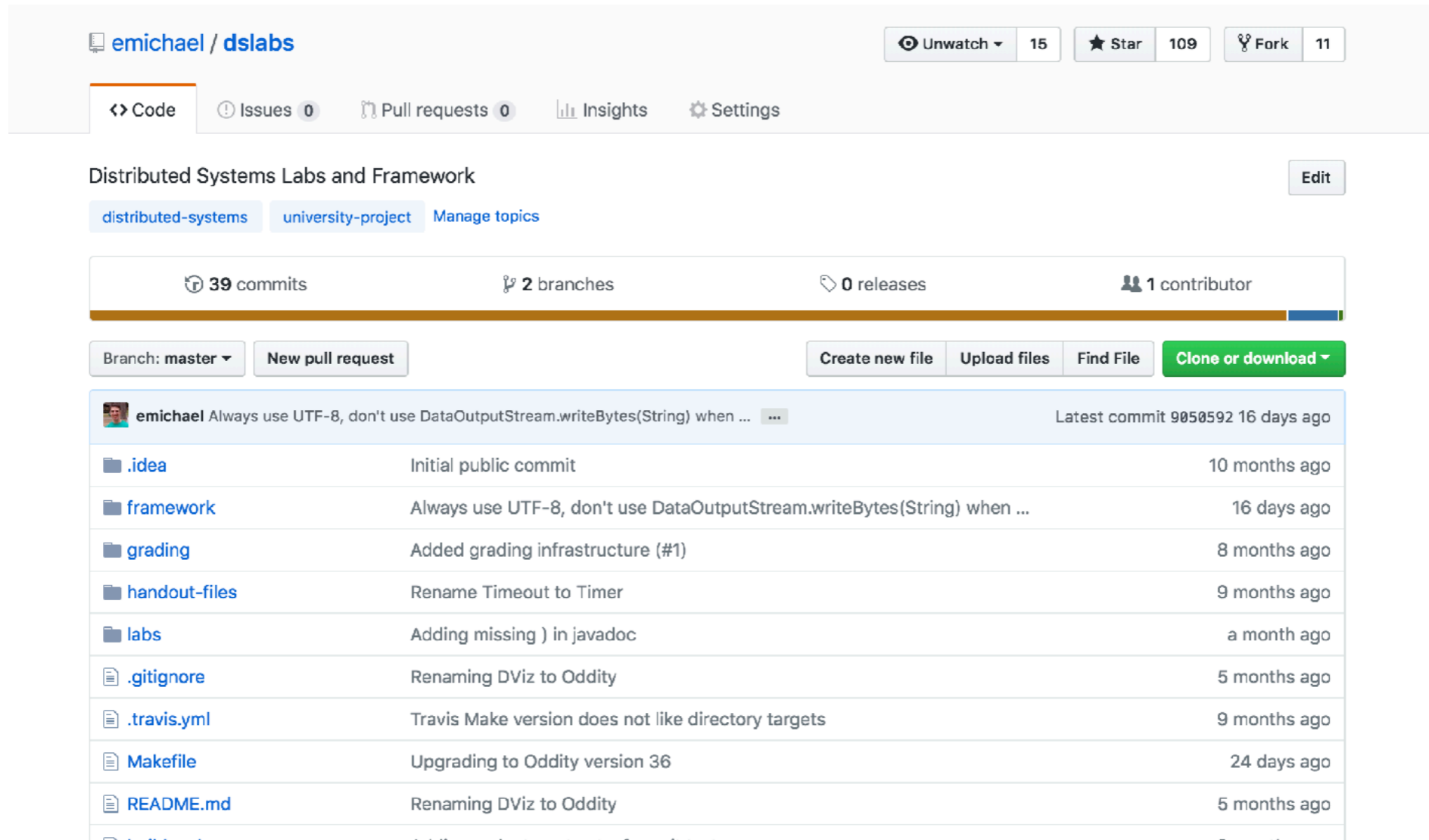
# Summary



- DSLabs, a new framework for building distributed systems assignments:
  - ❖ Uses efficient model checking based on guided search techniques,
  - ❖ Allows instructors to design model checking tests for student implementations,
  - ❖ Includes tools for debugging, understanding errors when they occur.
- DSLabs has been invaluable at UW, helped us scale undergraduate distributed systems to 200 students per quarter.



# Thanks for Listening!



The screenshot shows the GitHub repository page for 'emichael / dslabs'. At the top, it displays the repository name, a navigation bar with 'Code', 'Issues 0', 'Pull requests 0', 'Insights', and 'Settings', and interaction buttons for 'Unwatch 15', 'Star 109', and 'Fork 11'. Below this, the repository description 'Distributed Systems Labs and Framework' is shown with an 'Edit' button and topic tags 'distributed-systems' and 'university-project'. A summary bar indicates '39 commits', '2 branches', '0 releases', and '1 contributor'. The main content area shows a commit history table with columns for the commit author, message, and date.

Commit	Message	Time
emichael	Always use UTF-8, don't use DataOutputStream.writeBytes(String) when ...	Latest commit 9050592 16 days ago
	.idea	Initial public commit 10 months ago
	framework	Always use UTF-8, don't use DataOutputStream.writeBytes(String) when ... 16 days ago
	grading	Added grading infrastructure (#1) 8 months ago
	handout-files	Rename Timeout to Timer 9 months ago
	labs	Adding missing ) in javadoc a month ago
	.gitignore	Renaming DViz to Oddity 5 months ago
	.travis.yml	Travis Make version does not like directory targets 9 months ago
	Makefile	Upgrading to Oddity version 36 24 days ago
	README.md	Renaming DViz to Oddity 5 months ago



<https://github.com/emichael/dslabs>

[emichael@cs.washington.edu](mailto:emichael@cs.washington.edu)

Feedback, issues,  
pull-requests welcome