Distributed Systems 1

CUCS Course 4113
https://systems.cs.columbia.edu/ds1-class/

Instructor: Roxana Geambasu

Testing and Model Checking Distributed Systems

Properties

Want to ensure certain properties of distributed systems

- Safety (correctness at every step)
- Liveness (eventually, something will happen)
- Performance (something will happen within a certain time or with a certain amount of resource)

Question: How do we ensure our DS achieves properties?

This Lecture

Part 1: Testing

- Usually checks safety in a best effort way, but it's applied directly on implementation.

Part 2: Model Checking

- Comprehensive checking of safety and liveness properties, but usually applied on design, not on the implementation.

Part 3: Benchmarking and Evaluation

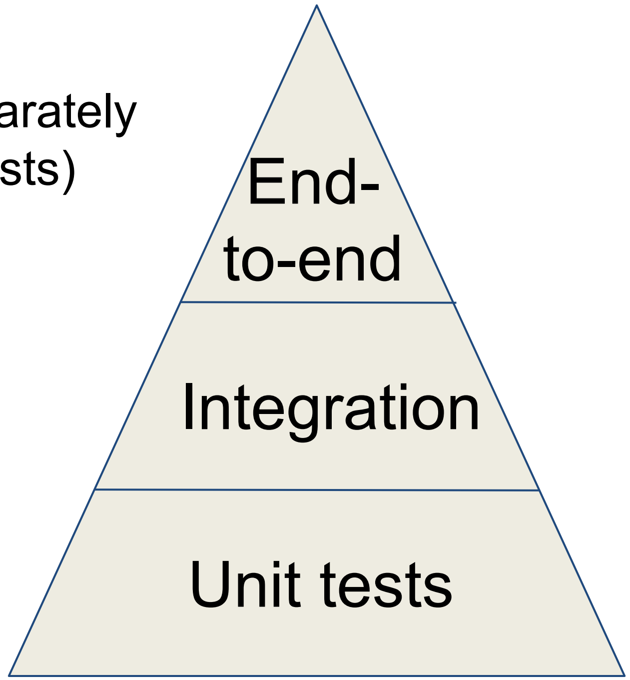
- Measures performance properties of systems (latency, throughput, resource utilization, energy consumption, etc…) under realistic or stress load.

Part 1: Testing

Slides inspired from: https://www.youtube.com/watch?v=hQSCnJ3kj2M.

Testing Pyramid

- Unit tests:
  - Basis to catch most bugs pre-production
  - Test every function, module, microservice separately
  - Stub all other components (mocks, contract tests)
  - Shoot for >95% line coverage
- Integration:
  - Test multiple integrated components, still with some stubbing for external deps
  - Often rely on growing list of scenarios
- End-to-end:
  - Often ran on deployment in production(-like) environment, often with mirrored traffic



Testing in Production

- Pre-production testing is critical, but insufficient
  - Conditions can change dramatically in production
  - Different combos of protocols/protocol versions, ongoing migrations of dependencies, different workload patterns, different configs, …

- But testing in production raises challenges, so it needs to be done carefully and support in the code!

Types of Tests in Production

Deploy

↓

Release

↓

Operate

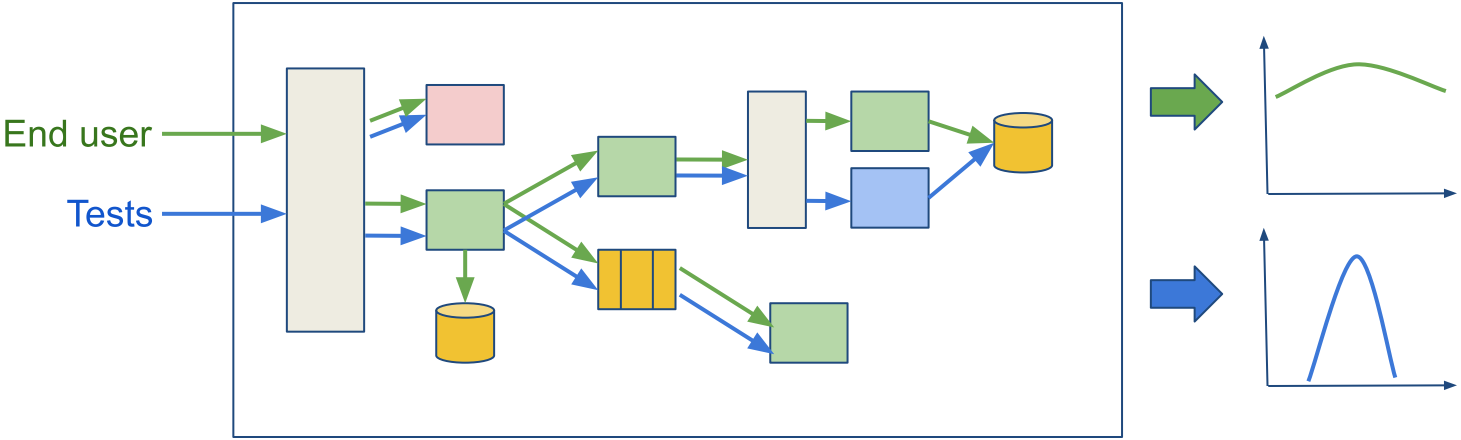
- End-to-end integration test cases
- Shadowing/traffic mirroring
- Canary deployments
- Chaos engineering
- Real user monitoring

Risks of Testing in Production

- User impact
- State poisoning
- Traffic saturation
- Telemetry data skew
- Misfired alerts

The application needs to be **aware of** (code for) tests being performed in production

Test Labeling



- Test label is propagated across services per request
- Services and routing layer are aware of test label
- Supported by RPC tracing systems (e.g., OpenTelemetry)

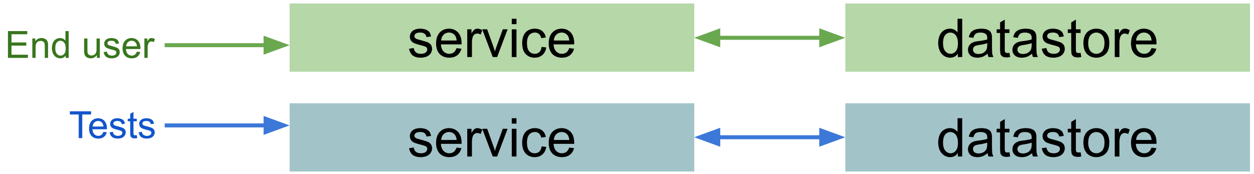
Fixing the Risks

- User impact → Test before releasing
- State poisoning → Separate writes to datastores
- Traffic saturation → Implement QoS based on test label
- Telemetry data skew → Mark telemetry with test label
- Misfired alerts → Exclude test telemetry from alerts

Example: OpenTelemetry

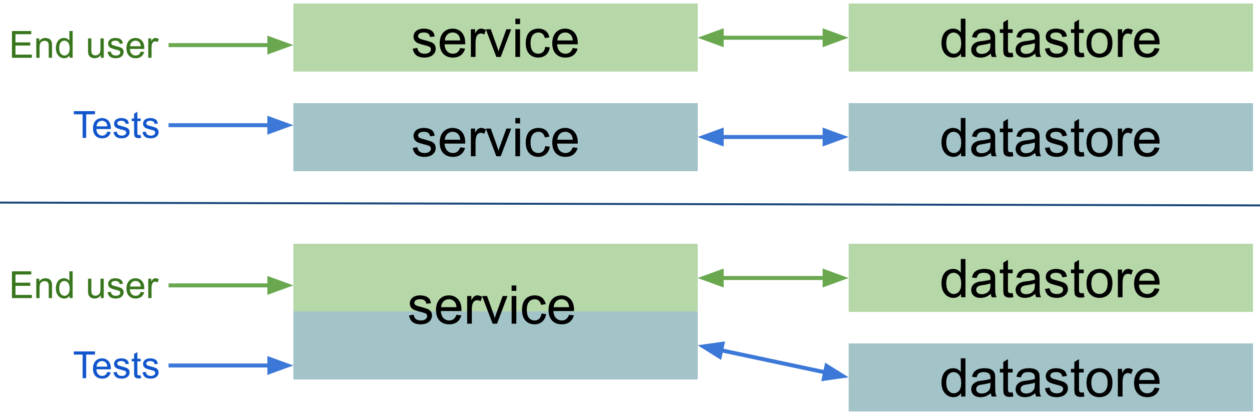
```
func TestIntegration(t *testing.T) {
  tracer := global.TraceProvider().GetTracer("")
  ctx := distributedcontext.NewContext(context.Background(), key.String("tenancy", "test"))
  ctx, span := tracer.Start(ctx, t.Name())
  defer span.End()

 //  …  test case
}
```

Managing State



Single-tenant services
Single-tenant datastores

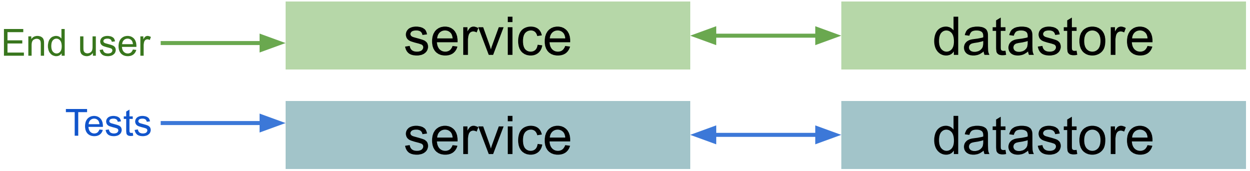
Managing State



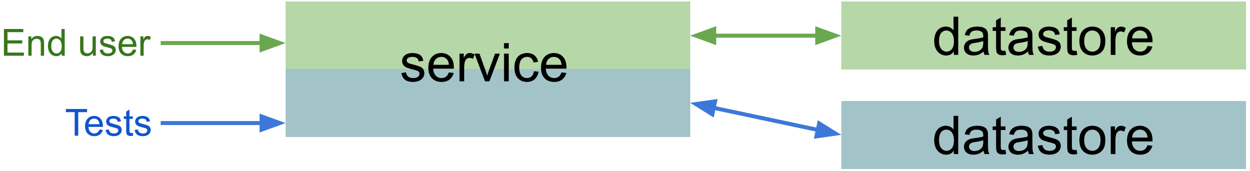
Single-tenant services
Single-tenant datastores

Multi-tenant service
Single-tenant datastores

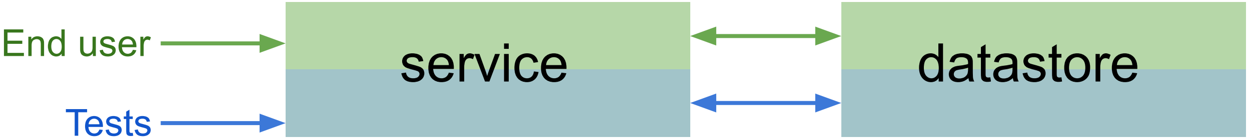
Managing State



Single-tenant services
Single-tenant datastores

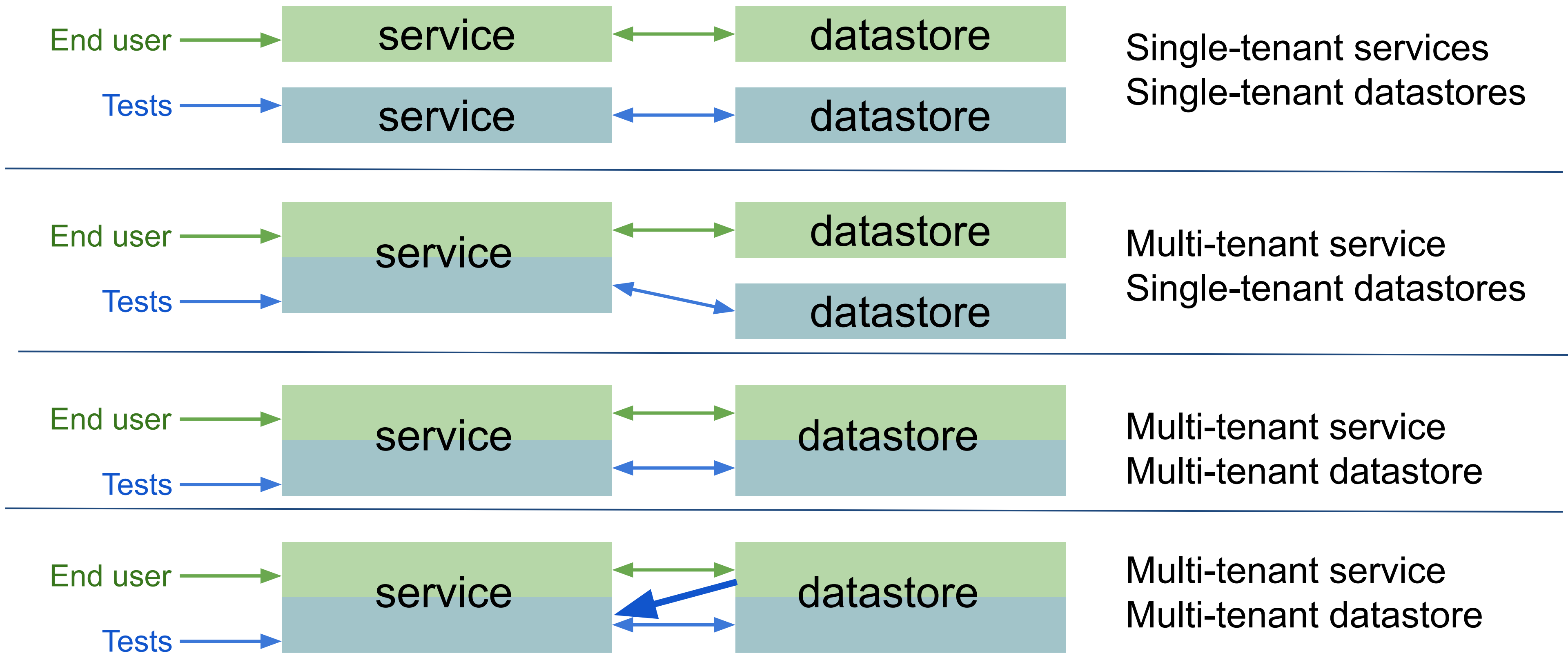


Multi-tenant service
Single-tenant datastores



Multi-tenant service
Multi-tenant datastore

Managing State



Managing Telemetry Data

With OpenTelemetry:

```
// Init measure
meter := global.MeterProvider().GetMeter(“”)
tenancyKey := key.New(“tenancy“)
measure := meter.NewInt64Measure(“myMeasure”, metric.WithKeys(tenancyKey))

// Extract tenancy from distributed context
var labels []core.KeyValue
if tenancyValue, ok := distributedcontext.FromContext(ctx).Value(“tenancy”); ok {
  labels = append(labels, core.KeyValue{Key: tenancyKey, Value: tenancyValue})
}

// Attach labels to measurement
measure.Record(ctx, 123, meter.Labels(labels…))
```

Chaos Engineering

- Principles:
  - Aggressively experiment on a system to build confidence in the system's capability to withstand turbulent conditions in production.
  - Start by defining 'steady state' as some measurable output of a system that indicates normal behavior.
  - Hypothesize that this steady state will continue in both the control group and the experimental group.
  - Introduce variables that reflect real world events (like servers that crash, hard drives that malfunction, network connections that are severed, datacenters that go down (!), etc.).
  - Try to disprove the hypothesis by looking for a difference in steady state between the control group and the experimental group.
- Neflix has good tools and a book about this. E.g.: Chaos Monkey.

More Testing Resources

A good index of testing frameworks, practices, and research can be found here:

https://github.com/asatarin/testing-distributed-systems

Part 2: Model Checking

Slides inspired from: https://www.hillelwayne.com/talks/distributed-systems-tlaplus/

Model Checking: Topics

- Motivation
- TLA+ Examples

Model Checking: Topics

- Motivation
- TLA+ Examples

DS Testing is HARD

- What does 95% line coverage in unit tests tell you?
- Often, failures are non-deterministic
  - You run multiple times, but what if not enough times?

- Reasons DS testing is hard:
  - Challenge 1: Concurrency
  - Challenge 2: Non-determinism (including due to failures)

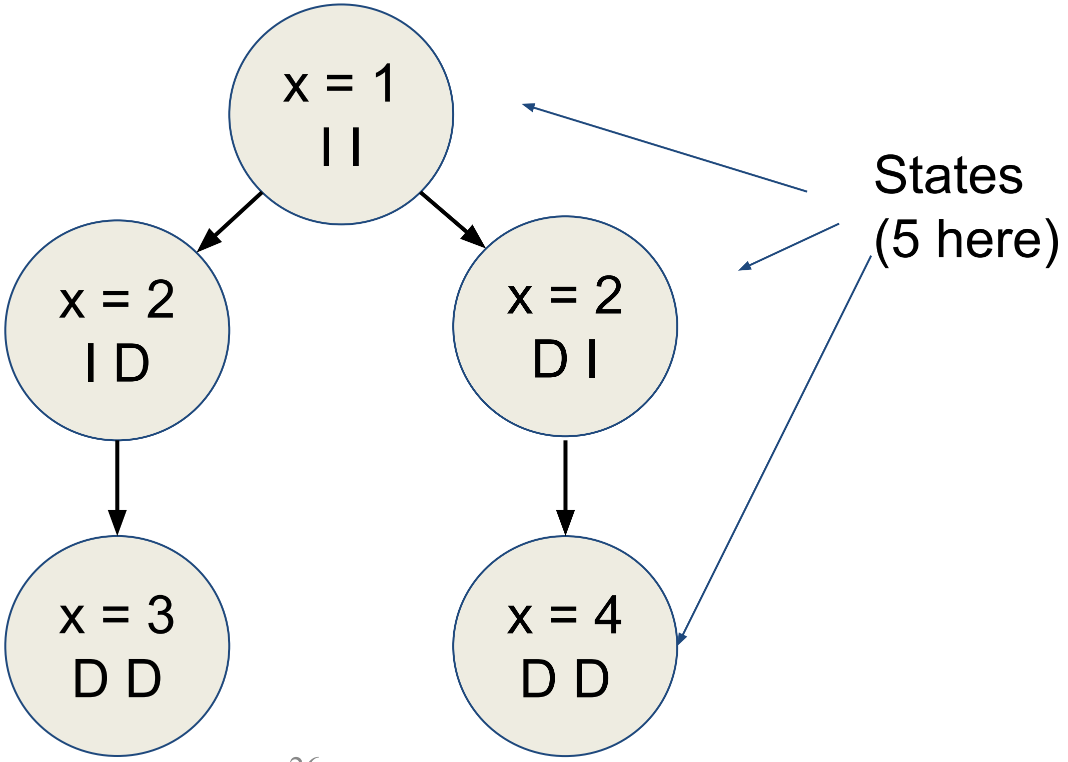
Testing Challenge 1: Concurrency

Example

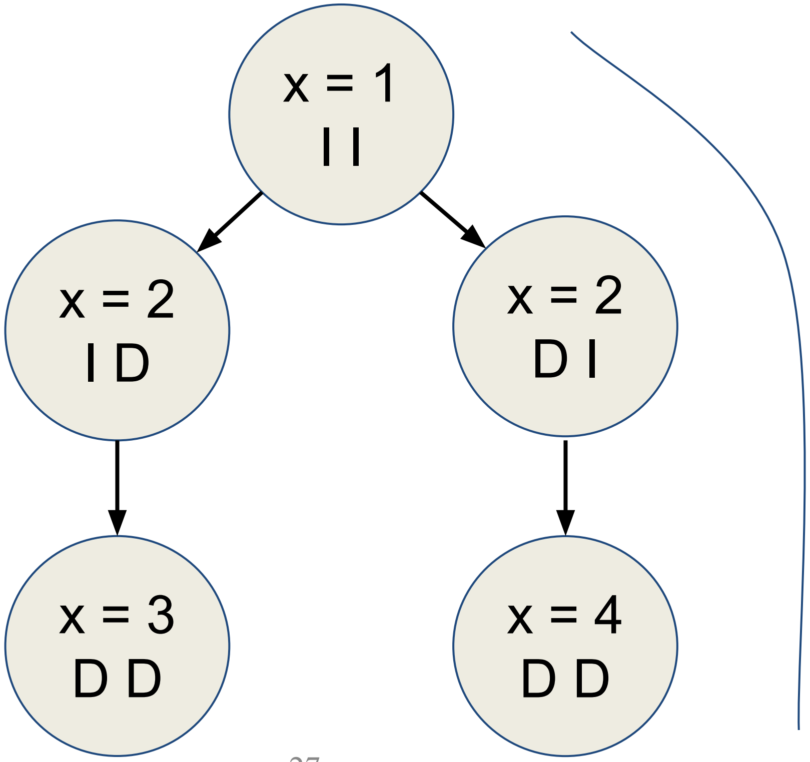
global x = 1

process 1	**process 2**
x = x+1	x = x*2

State & Behavior Spaces



State & Behavior Spaces



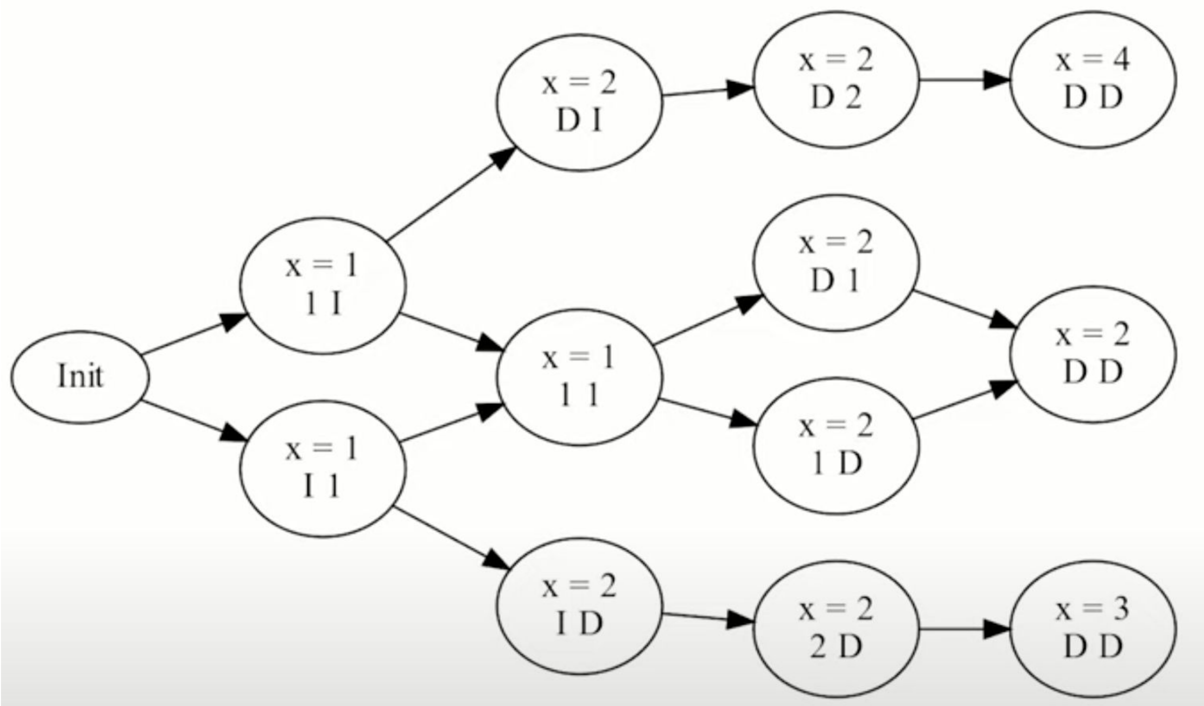
Behavior, a.k.a., history or execution (2 here)

Small Increase in Concurrency ⇒ Large Increase in State Space

global x = 1

process 1	**process 2**
local tmp = x	local tmp = x
x = tmp+1	x = tmp*2

State/Behavior Space



13 states
6 behaviors

(from that small amount of added concurrency!)

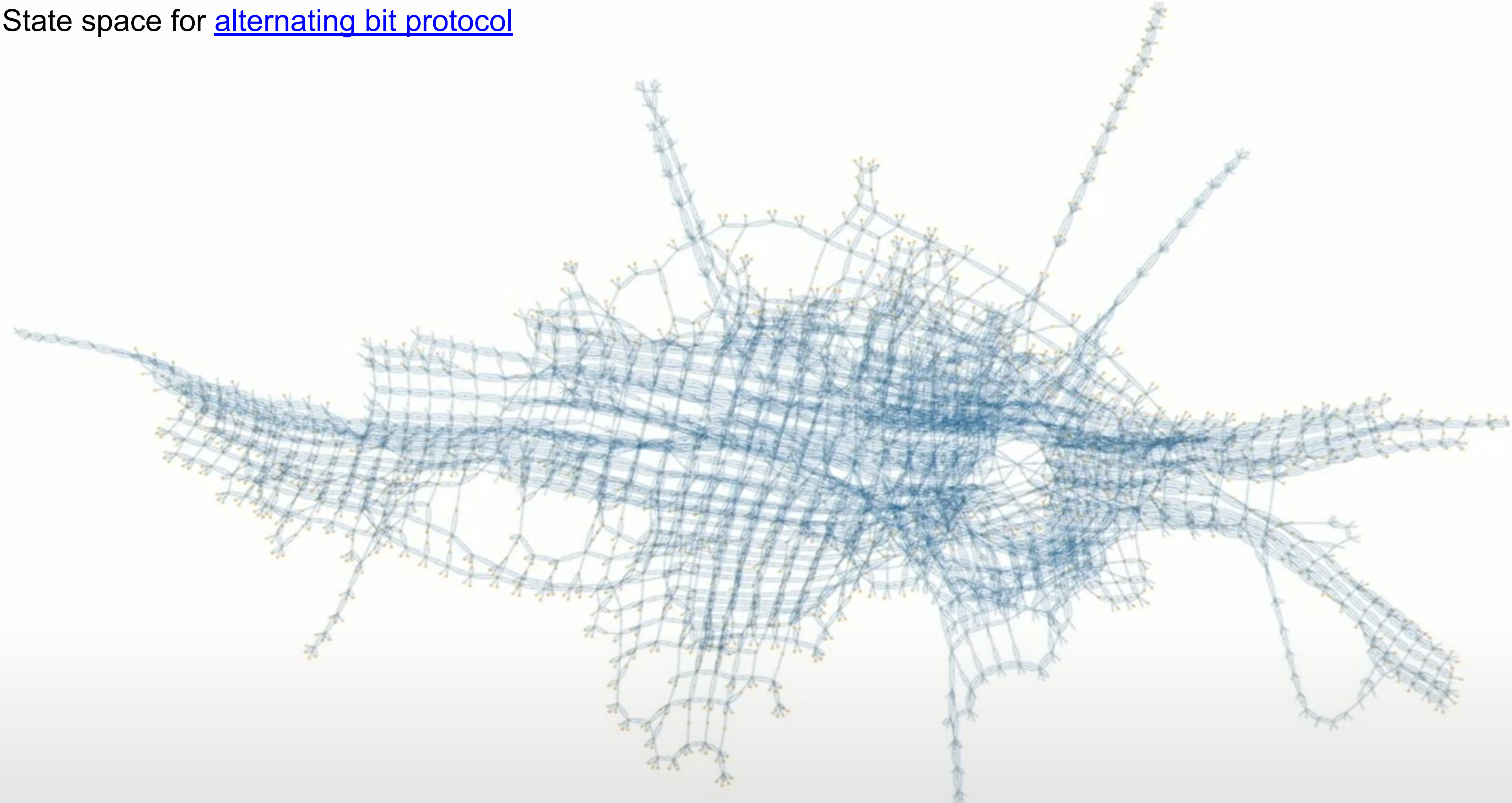
State Space for Example

n = num processes
m = num steps per process

Number of states:
$m\ n * (m\ n)! / m!^n$

Thus, adding one more process means we'd have 540 states, an order of magnitude increase!

State space for alternating bit protocol



Example state space visualization taken from here (more protocols available)

Testing Challenge 2: Non-Determinism

Previous Example with Failures

global x = 1

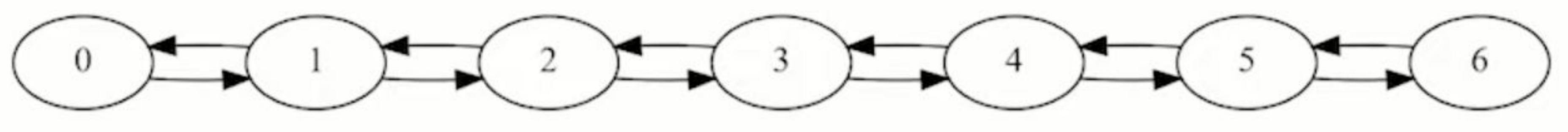
process 1	**process 2**
local tmp = x	local tmp = x
either	**either**
x = tmp+1	x = tmp*2
or	**or**
crash	**crash**

Another Example (deterministic for now)

```
x = 0

while true:
  if x < 6:
    x = x + 1

while true:
  if x > 0:
    x = x - 1
```

Q: How many states and behaviors?

Another Example (deterministic for now)



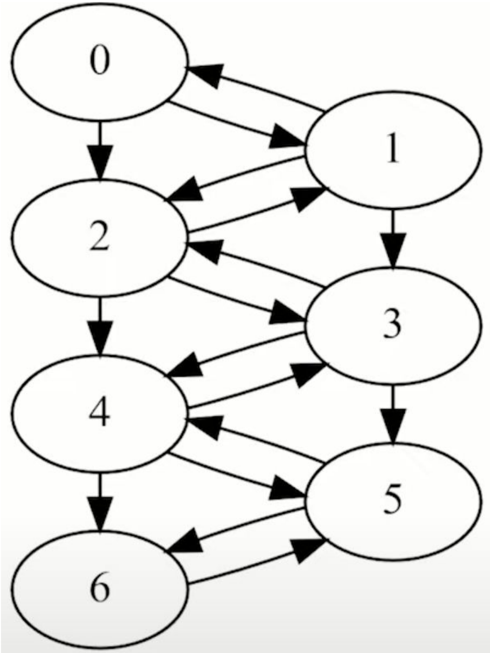
7 states
14 distinct behaviors

Add a Little Non-Determinism

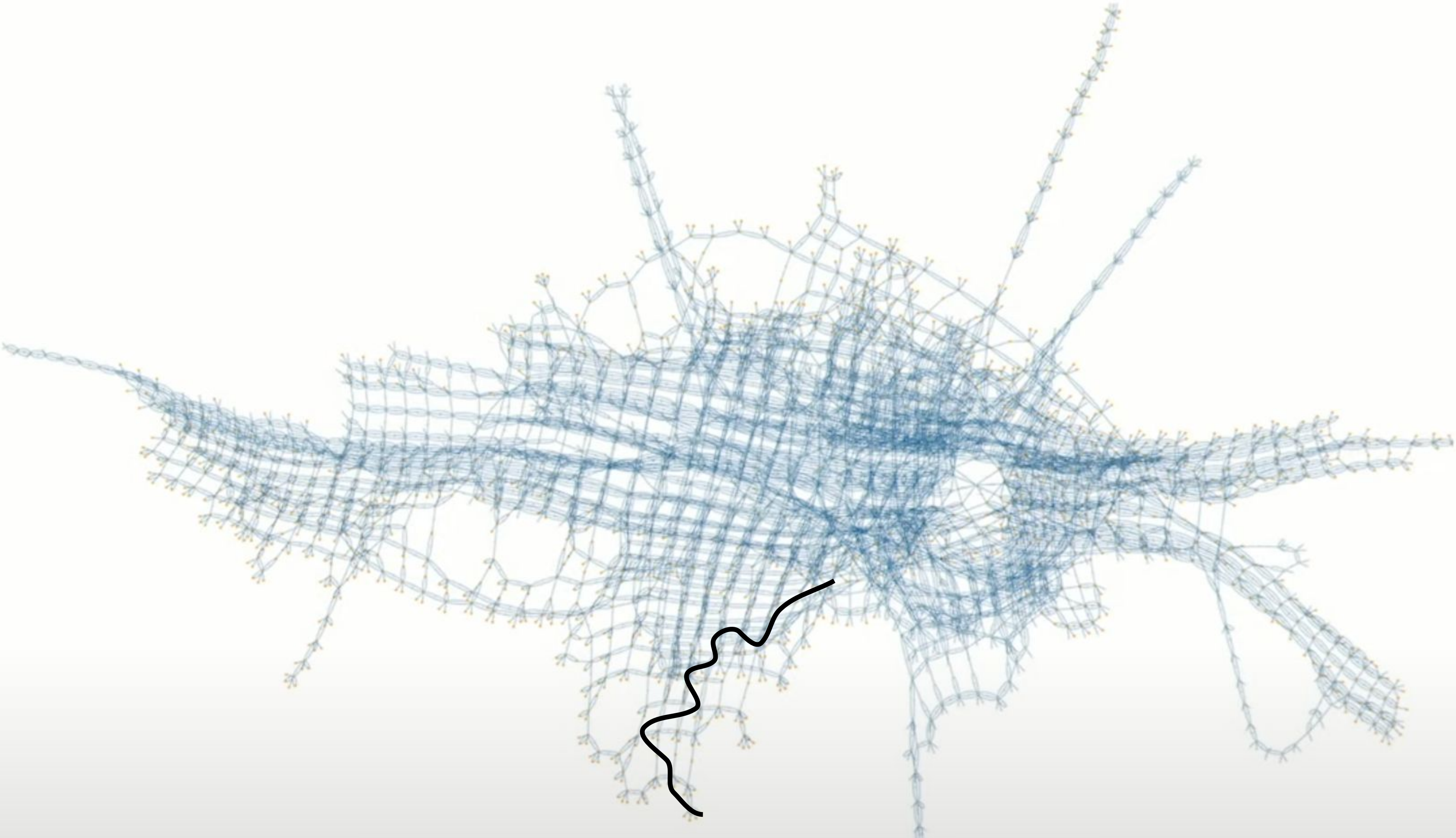
```
x = 0

while true:
  if x < 6:
     x = x + 1
  OR
  if x < 5:
     x = x + 2
```

```
while true:
  if x > 0:
     x = x - 1
```

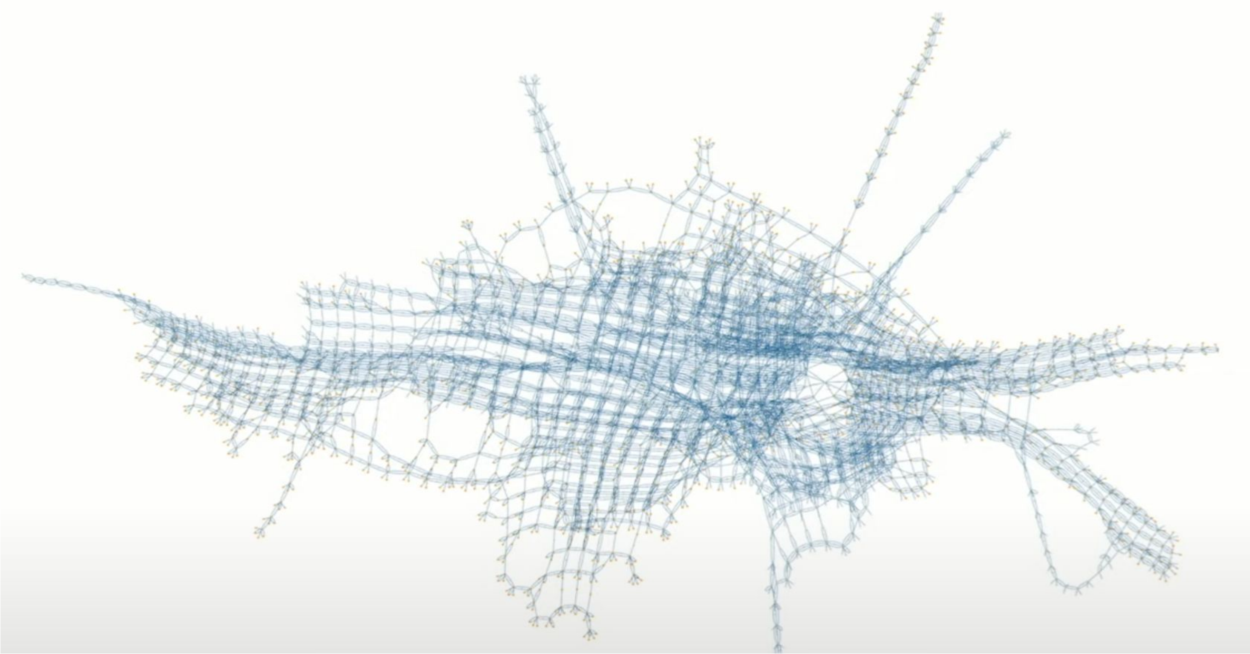
A Little Non-Determinism ⇒ Large Increase in Behavior Space



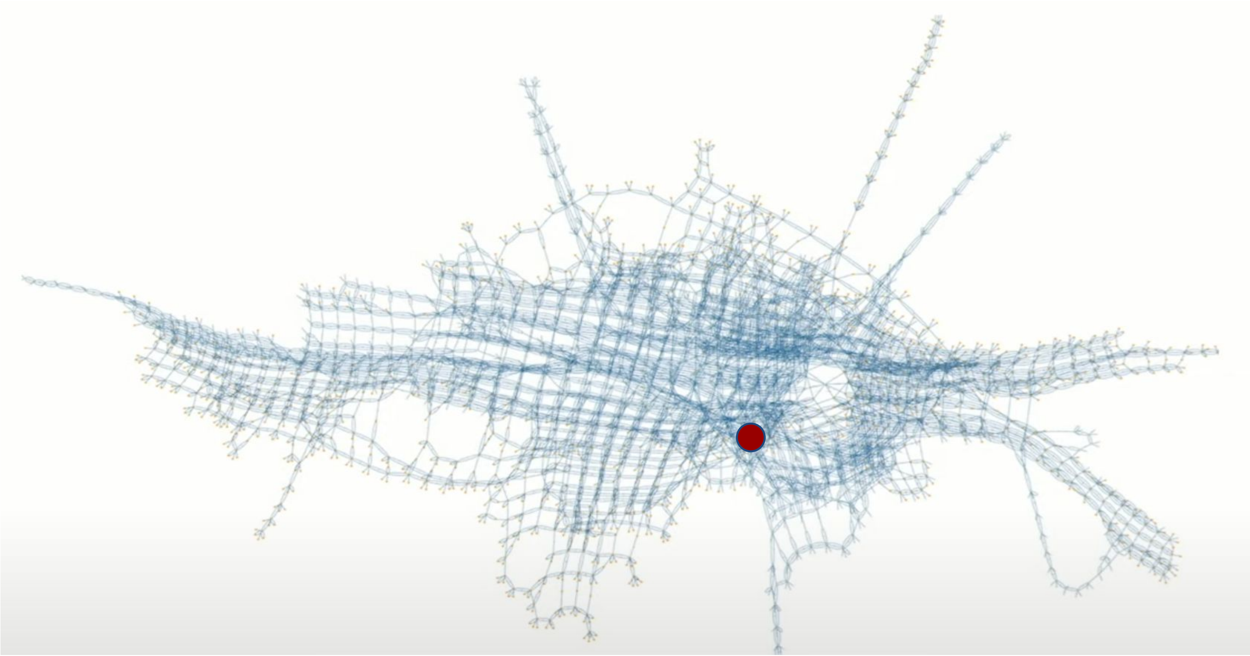
- No new states
- But 5 new edges
- ~100 distinct behaviors, an order of magnitude increase!



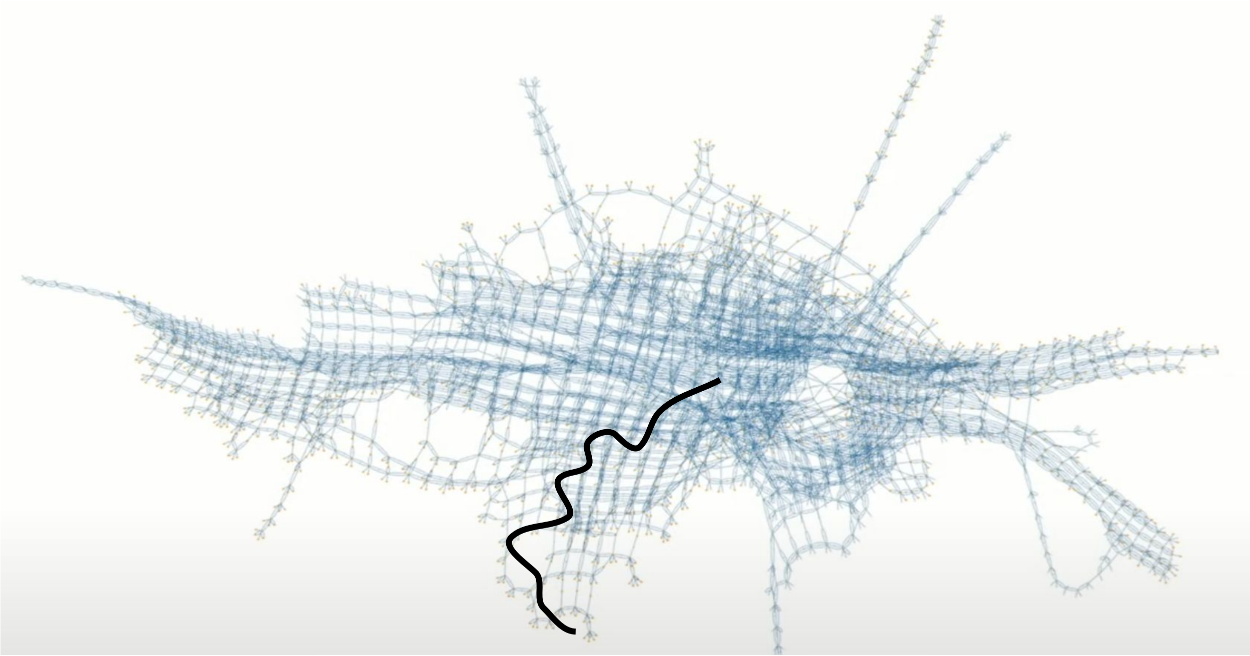
Both States and Behaviors Can be Invalid



Both States and Behaviors Can be Invalid



Both States and Behaviors Can be Invalid



Do We Need to Check All of Them?

Assume: bad state/behavior occurs 1 / 1,000,000,000 events.

If your system executes 100 events / second (it’s not much!)

Then, how long before system reaches a bad state/behavior?

Do Invalid States/Behaviors Matter?

1 / 1,000,000,000 events
* 100 events / second

3-4 per year !!

For bad events, such as crashes, corruptions, outages, that’s a lot!

Solutions

- Better programming abstractions, interfaces, protocols.
- Examples:
  - Type safe langs
  - CAS
  - Transactions
  - Locks
  - Semaphors
  - CRDTs
  - Paxos
  - RAFT
  - …

Solutions

- Better programming abstractions, interfaces, protocols.
- Examples:
  - Type safe langs
  - CAS
  - Transactions
  - Locks
  - Semaphors
  - CRDTs
  - Paxos
  - RAFT
  - …

These all reduce the space (by half? a third?), but still too difficult to reason about and test large programs thoroughly.

Bigger Problem: Design Issues

- State/behavior explosions occur due to **design**, not implementation (think “complexity”).
- Programming abstractions focus on implementation, not design.
  - If a design has flaws, any implementation will have flaws.
- But how do we test designs?
  - A lot of frameworks for specifying designs exist (e.g., UML, pseudocode, whiteboard :)), but most aren’t testable.

Formal Specification and Model Checking

(Ideally before implementing your system:)

1. Write a **specification** of the system in a formal specification language (think math).
2. Specify correctness properties as **invariants** on states or behaviors.
3. Use a **model checker** to exhaustively check that every state/behavior of the system, within a bounded range of configurations, satisfies your invariants.

Model Checking: Topics

- Motivation
- TLA+ Examples

TLA+ (Temporal Logic of Actions)

- Mathematical formalism for specifying asynchronous DSes.
  - Everything is expressed as logical formulas.
- Developed by Leslie Lamport: “Best way to describe things precisely is with simple mathematics.”
  - E.g: `\E x \in S : \A y \in S : y <= x`
- Useful for eliminating fundamental design bugs, which are hard to find and expensive to correct in code.
  - But also useful for other things, like understanding systems better, comparing designs, … [Geambasu+08]

TLA+ Suite

- **TLA+**, the formalism:
  - Adopts the state machine perspective of a DS.
  - Inherently assumes concurrency, non-determinism, and failures, but you can specify constraints (called “fairness”).
  - Supports two types of properties:
    - **Safety**: Must hold for all states (e.g., at any state, at least one server has the committed data).
    - **Liveness**: Must eventually must hold (e.g., eventually all replicas have the committed data).
- **TLC**, the model checker:
  - Verifies that all states/behaviors satisfy the properties within a bounded configuration space.
- **PlusCal**: imperative wrapper around TLA+ math.

State Machine Spec

- "State" in TLA+ refers to the **entire, global state** of the specified DS.
- A DS specification consists of:
  - A set of potential initial states (**Init**)
  - Next-state relation (**Next**): The set of all the possible transitions among pairs of states.
    - These are defined as logical formulas that may or may not become true ("fire") in any given state.
    - TLC will try them all in any given state. Those that "fire" will be followed and may create new states.
  - Then, **Spec == Init /\ []Next**
- Properties are also specified as logical formulas, with two operators:
  - Safety: **Spec ⇒ [](logical_formula)** (for all states)
  - Liveness: **Spec ⇒ <>(logical_formula)** (eventually)

TLA+ Is Useful and USED!

- AWS
- Amazon
- Azure
- Xbox
- eSpark Learning
- Sutori
- Elastic
- Mongo
- ING
- Auxon
- OSOCO
- OpenStack
- CockroachDB
- Protocol Labs
- Facebook
- Cigna
- Shopify
- Auth0
- Several clients under NDA
- Like 80 blockchain companies

See Lamport's list of TLA+ industrial uses.

Amazon’s Experience

Paper: “Why Amazon Chose TLA+” Chris Newcombe Amazon, Inc., 2014.

Talk: “The Evolution of Testing Methodology at AWS: From Status Quo to Formal Methods with TLA+” Tim Rath, 2015.

“**Why Amazon is using formal methods.** Amazon builds many sophisticated distributed systems that store and process data on behalf of our customers. In order to safeguard that data we rely on the correctness of an ever-growing set of algorithms for replication, consistency, concurrency-control, fault tolerance, auto-scaling, and other coordination activities. Achieving correctness in these areas is a major engineering challenge as these algorithms interact in complex ways in order to achieve high-availability on cost-efficient infrastructure whilst also coping with relentless rapid business-growth. We adopted formal methods to help solve this problem.”

Example Amazon Uses (from talk)

- DynamoDB
  - Replication protocols
  - Membership handling
  - Quorum Configuration Changes
- Other AWS projects [8]
  - Low level distributed network protocol
  - Internal distributed lock manager
  - S3, EC2, EBS system management algorithms

Example 1:
Basic Paxos in TLA+

https://github.com/neoschizomer/Paxos/blob/master/Paxos.tla
(look at code)

Example 2:
Simple Protocol in PlusCal

https://www.hillelwayne.com/talks/distributed-systems-tlaplus/
(play video starting at minute 22)