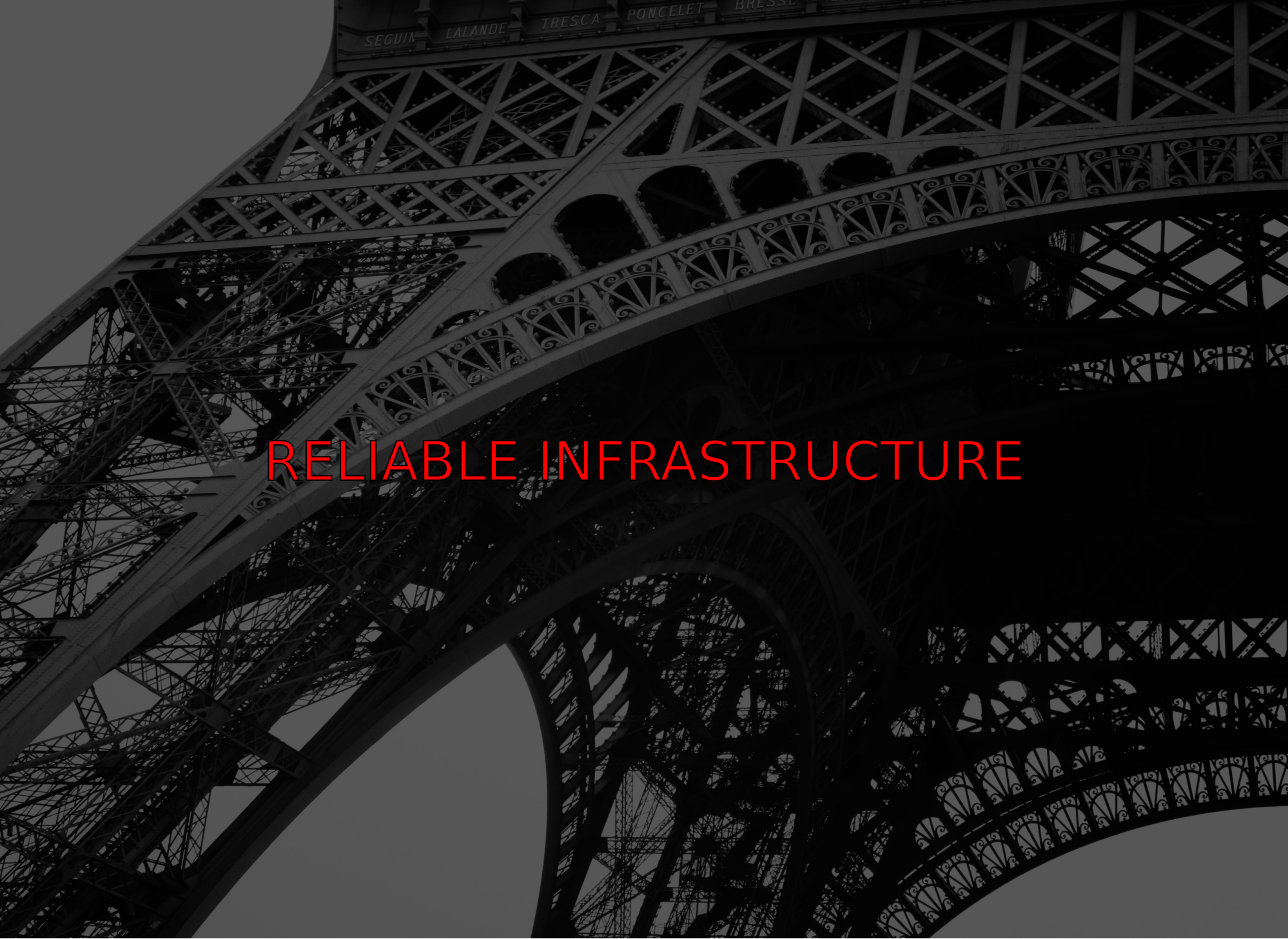
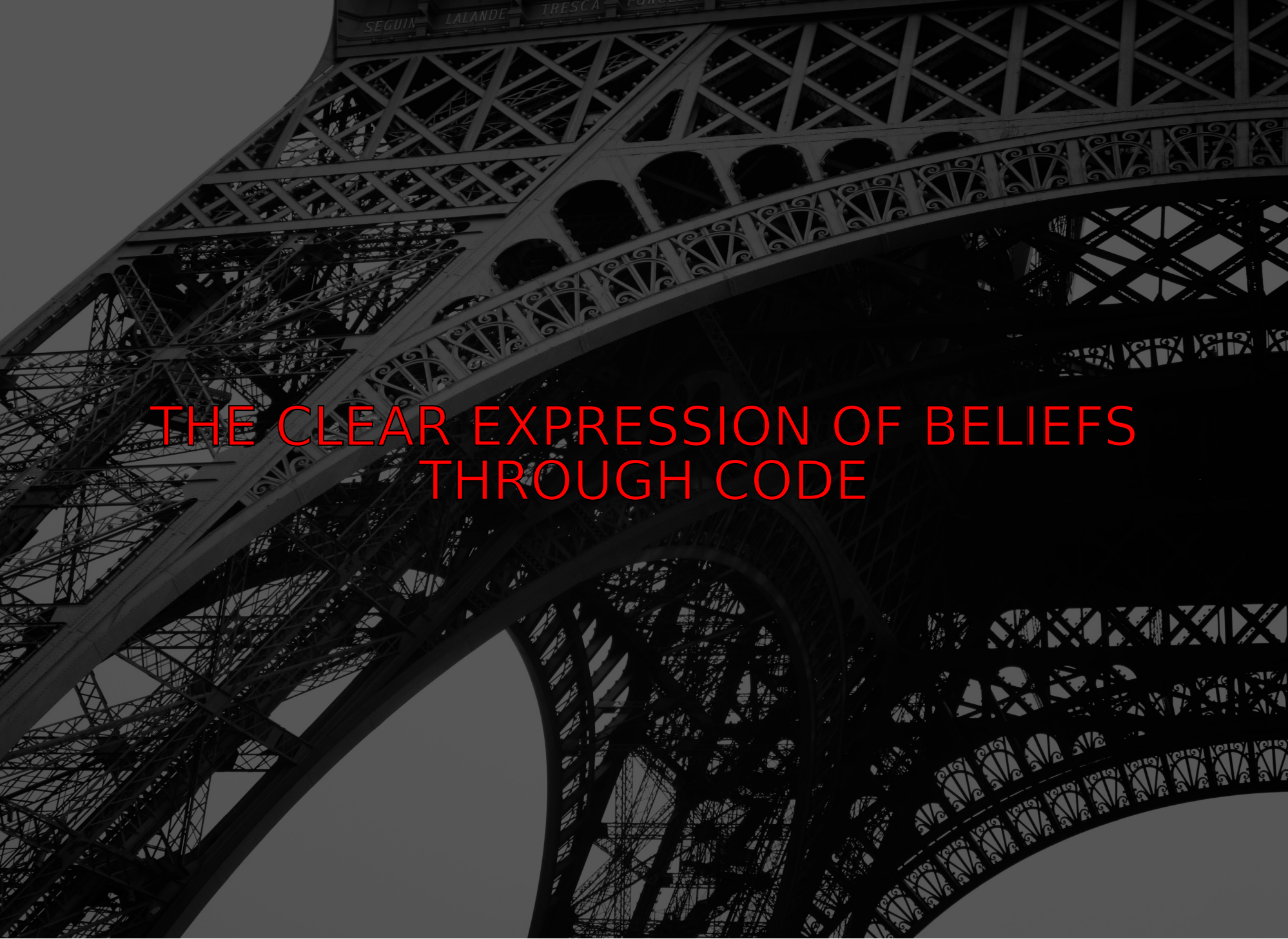


RELIABLE INFRASTRUCTURE



THE CLEAR EXPRESSION OF BELIEFS THROUGH CODE

WHO AM I?

distributed systems, databases, infrastructure

sled: database construction and testing kit

I hack on Open Source Coin, aiming to better fund and empower free software

you might know me as spacejam

ROADMAP

- brains, code, reality
- fuzzing our assumptions
- chaos on the net
- cool tips, advanced topics

NON-ROADMAP (COVERED WELL ELSEWHERE)

- metrics and introspection
- capacity planning
- Brendan Gregg: Systems Performance: Enterprise and Cloud
- Google: Site Reliability Engineering
- Martin Kleppmann: Designing Data-Intensive Applications

THIS IS YOUR BRAIN

everything we know is subject to bias

everything we build reflects these biases

we tend not to ask our beliefs to "pay rent"

we don't do cache invalidation

see "Rationality: From AI to Zombies", this is all about confronting assumptions

problem: our code reflects our biases, our tests are often biased similarly

solution: don't write tests. write expectations.

have the machine generate random test cases

make beliefs explicit, force them to pay rent

PROPERTY TESTING

CRATE: PROPTEST

```
// this property is false, but perhaps
// not unreasonable to expect to be true
proptest! {
  #[test]
  fn mult_and_div(ref a in any::<usize>()) {
    let result = (a * 5) / 5;
    assert_eq!(result, a);
  }
}
```

CRATE: PROPTEST

```
$ cargo test
test mult_and_div ... FAILED
Test failed: attempt to multiply with overflow;
minimal failing input: ref a = 3689348814741910324
test result: FAILED. 0 passed; 1 failed
```

crates.io/crates/**proptest**

```
$ cat proptest-regressions/main.txt
# Seeds for failure cases proptest has
# generated. It is automatically read
# and these particular cases re-run before
# any novel cases are generated.

# shrinks to ref a = 3689348814741910324
xs 4050946508 1278147119 4151624343 875310407
```

wonderful for testing codecs, serialization, compression, or any set of operations that should retain equality.

```
proptest! {
  #[test]
  fn compress_roundtrip(ref s in ".*") {
    let result = decompress(compress(s));
    assert_eq!(result, s);
  }
}
```

it's easy to generate more structured input, too

```
proptest! {
  #[test]
  fn parses_all_valid_dates(
    ref s in "[0-9]{4}-[0-9]{2}-[0-9]{2}"
  ) {
    parse_date(s).unwrap();
  }
}
```

im-rs has some great examples of using proptest!

```
#[test]
fn exact_size_iterator(ref vector in vector(i32::ANY)) {
  let mut should_be = vector.len();
  let mut it = vector.iter();
  loop {
    assert_eq!(should_be, it.len());
    match it.next() {
      None => break,
      Some(_) => should_be -= 1,
    }
  }
  assert_eq!(0, it.len());
}
```

CONFIGURATION IS A GREAT TARGET

crates.io/crates/**sparse-bitfield**

```
proptest! {
  #[test]
  fn doesnt_crash(
    bit in 0usize..1_000_000,
    page_sz_exponent in 0usize..30
  ) {
    let page_sz = 1 << page_sz_exponent;
    let mut bits = Bitfield::new(page_sz);
    assert_eq!(bits.set(bit, true), Change::Changed);
    assert_eq!(bits.get(bit), true);
  }
}
```

But how do we apply random testing to more complex systems?

MODEL-BASED TESTING

1. write a simplified model of a system
2. apply random sequences of operations on both the implementation and the model
3. if the model and implementation behave differently, it means we've found a problem
4. the library will rerun the test, dropping out operations until it finds a minimal failing sequence (the machine just wrote a regression test for you!!!)

LET'S BUILD A TREE!

```
pub struct Tree<K, V> {
    root: Option<Node<K, V>>,
}

impl<K, V> Tree<K, V> {
  pub fn insert(&mut self, key: K, value: V) {}
  pub fn get(&self, key: &K) -> Option<&V> {
    None
  }
}
```

crates.io/crates/**model**

```
model! {
  Model          => let mut map  = BTreeMap::new(),
  Implementation => let mut tree = MyTree::new(),
  Set(k: usize::ANY, v: 0usize..10) => {
    map.insert(k, v);
    tree.set(k, v);
  },
  Get(k: usize::ANY) => {
    assert_eq!(map.get(&k), tree.get(&k))
  }
}
```

crates.io/crates/**model**

```
$ cargo test
running 1 test
test test_model ... FAILED
Test failed: assertion failed: `(left == right)
  left: `Some(0)`,
  right: `None`;
minimal failing input: [Set((2, 0)), Get(2)]
test result: FAILED. 0 passed; 1 failed
```

crates.io/crates/**model**

```
model! {
  Model =>          let mut map = BTreeMap::new(),
  Implementation => let mut db  = DB::open("my.db"),
  Set(k: usize, v: usize) => {
    assert_eq!(map.insert(k, v), db.set(k, v))
  },
  Get(k: usize) => {
    assert_eq!(map.get(&k), db.get(&k))
  },
  Restart() => {
    drop(db);
    db = DB::open("my.db");
  }
```

BONUS: LINEARIZABILITY!!!

operations "happen" at one time after they start

operations "happen" at one time before they end

can be tested by looking for a single order of operations that results in the same return values

crates.io/crates/**model**

```
linearizable! {
  Implementation => let i = AtomicUsize::new(0),
  BuggyAdd(usize)(v in 0usize..4) -> usize {
    let current = i.load(Ordering::SeqCst);
    i.store(current + v, Ordering::SeqCst);
    current + v
  }
}
```

TO LEARN MORE ABOUT MODEL-BASED TESTING

- Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane
- Finding Race Conditions in Erlang with QuickCheck and PULSE

But what about things that do IO?

FAULT INJECTION

problem: chaos_of(world) > chaos_of(laptop)

solution: make failures happen intentionally!

QUICK ASIDE: FOCUS

almost all (92%) of the catastrophic system failures are the result of incorrect handling of non-fatal errors explicitly signaled in software.

...

Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems (OSDI '14)

QUICK ASIDE: FOCUS

in 58% of the catastrophic failures, the underlying faults could easily have been detected through simple testing of error handling code.

Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems (OSDI '14)

WE SHOULD TRIGGER ERRORS THAT WE CODE FOR

```
if let Err(e) = do_something_with_io() {
  error!("we detected a problem: {}", e);
  // TODO(not me) deal with it!
  panic!(e);
}
```

crates.io/crates/**fail**

```
fn do_something_with_io() -> Result<(), ()> {
  fail_point!("might_fail", |_| Err(()));
  Ok(())
}

#[test]
fn test() {
  fail::setup();
  fail::cfg("might_fail", "return").unwrap();

  do_something_with_io().expect_err("should return Err");
}
```

gotcha: make sure only one thread is running failpoint tests at a time using this!

ADVANCED TECHNIQUE: COMBINE QUICKCHECK WITH FAIL

crates.io/crates/**pagecache**

```
impl Arbitrary for Op {
  fn arbitrary<G: Gen>(g: &mut G) -> Op {
    let fail_points = vec![
        "zero garbage segment",
        "buffer write",
        "write_config crc",
        "trailer write",
        "snap write mv",
    ];

    Op::FailPoint(*g.choose(&fail_points).unwrap())
  }
}
```

TO LEARN MORE ABOUT TESTING OPERATIONS ON FILES

- All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications (OSDI '14)

That's nice, but my system uses sockets to communicate.

NETWORK SIMULATION

- Jepsen has found bugs in many of the most popular distributed systems
- but it requires learning clojure and it can take a long time to run
- if we design our distributed systems to be testable, we can run thousands of partition tests per second
- by making these tests cheap to run, we can catch race conditions in a second or two instead of hours or weeks

SIMULATION

crates.io/crates/**paxos**

```
// Reactor is a trait for building simulable systems.
pub trait Reactor: Debug + Clone {
    type Peer: std::net::ToSocketAddrs;
    type Message;

    fn receive(
        &mut self,
        at: SystemTime,
        from: Self::Peer,
        msg: Self::Message,
    ) -> Vec<(Self::Peer, Self::Message)>;
}
```

SIMULATION

crates.io/crates/**paxos**

```
#[derive(Debug, Clone)]
struct Cluster {
  peers: HashMap<Addr, ReactorNode>,
  partitions: Vec<Partition>,
  in_flight: BinaryHeap<ScheduledMessage>,
  client_responses: Vec<ScheduledMessage>,
}
```

SIMULATION

crates.io/crates/**paxos**

```
// impl Cluster
fn step(&mut self) {
  let pop = self.in_flight.pop();
  if let Some((at, dest, from, m_in)) = pop {
    for (to, m_out) in dest.receive(now, from, m_in) {
      let at = generate_latency();
      if self.is_partitioned(at, &*to, dest) {
        continue; // it's partitioned, don't send
      }
      self.in_flight.push(at, dest, m_out);
    }
  }
}
```

Simulation sounds nice in practice, but my system is way too complex to run into bugs very frequently...

LINEAGE-DRIVEN FAULT INJECTION

- LDFI observes what goes right without faults, then induces targeted faults starting from the last successful operation, working backwards
- "Fault-tolerance is redundancy in space and time" - Peter Alvaro
- quite feasible with a simulator-friendly system!
- crate incoming!

ANTI-NONDETERMINISM

- clocks, random number generators, thread scheduling are common sources of nondeterminism
- threads can be scheduled using linux realtime priorities
- files can be wrapped with a mutation log that keeps track of syncs and can be "crashed"
- crate: deterministic

MISC TIPS

- use assert! and debug_assert! on non-trivial things! this makes our "fuzzers" extremely effective
- try not to use unwrap() everywhere, at least use expect("helpful message") to speed up debugging
- when propagating errors, include context that helps you get back to the root

GO BREAK SYSTEMS