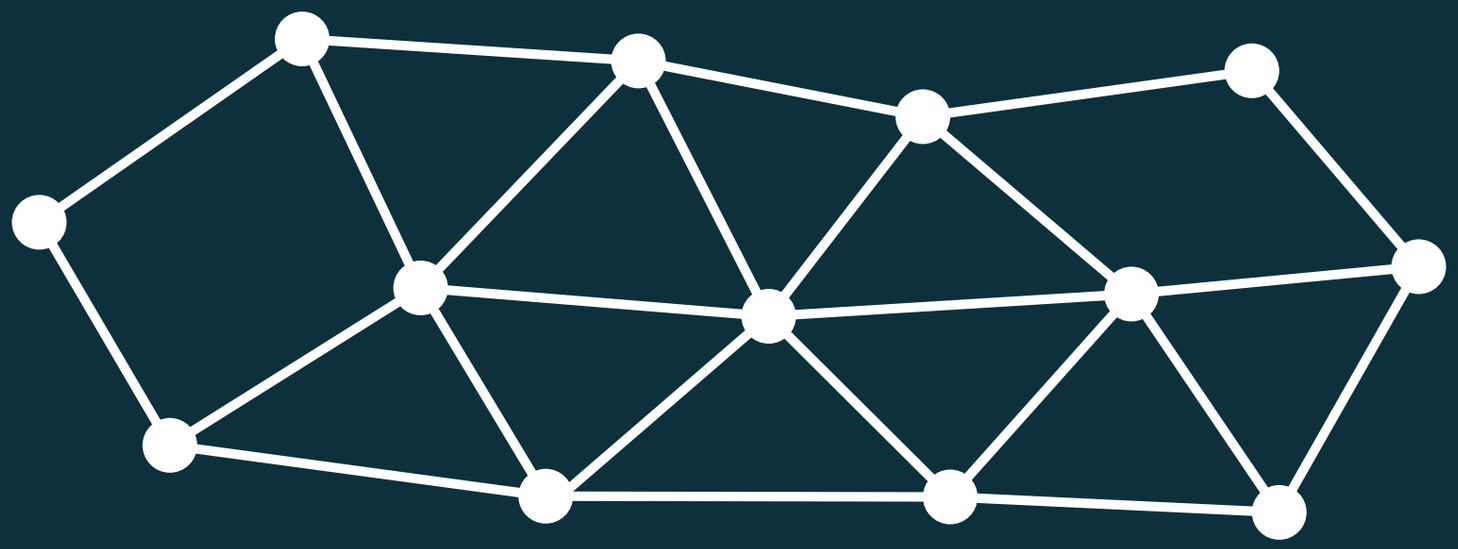
Deterministic Simulation Testing of Distributed Systems

For Great Justice

will.wilson@foundationdb.com

@FoundationDB

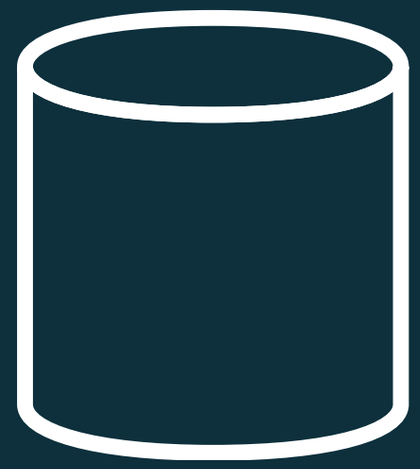
Debugging distributed systems is terrible



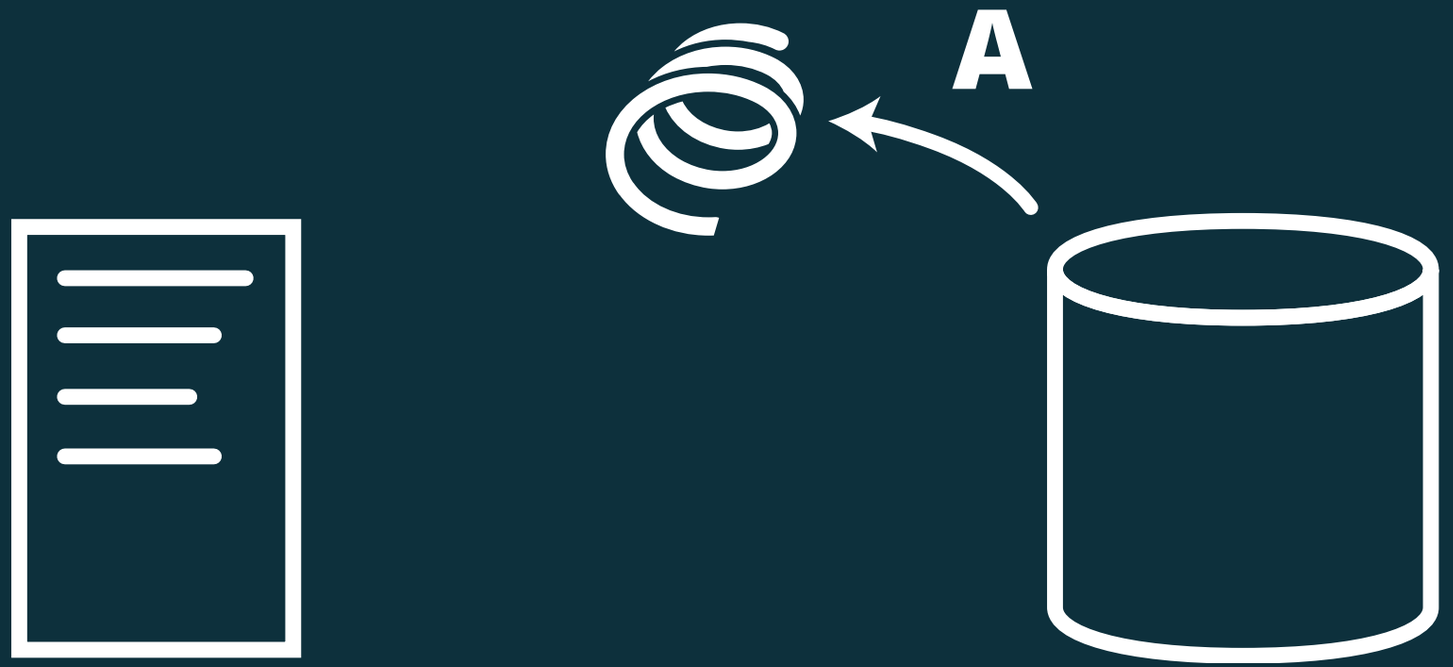
First of all, distributed systems tend to be complicated...

... but it’s worse than that

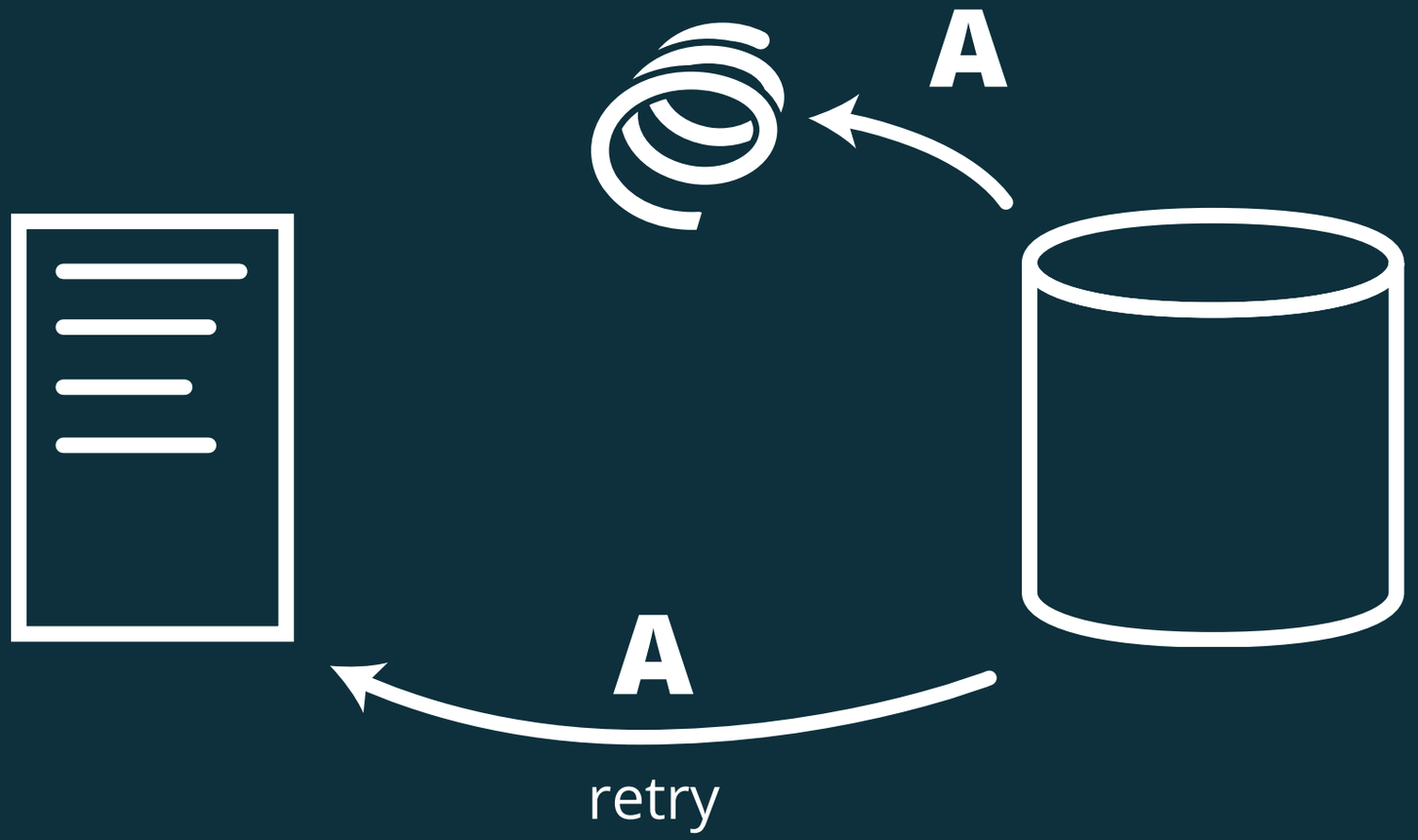
Repeatability? Yeah, right



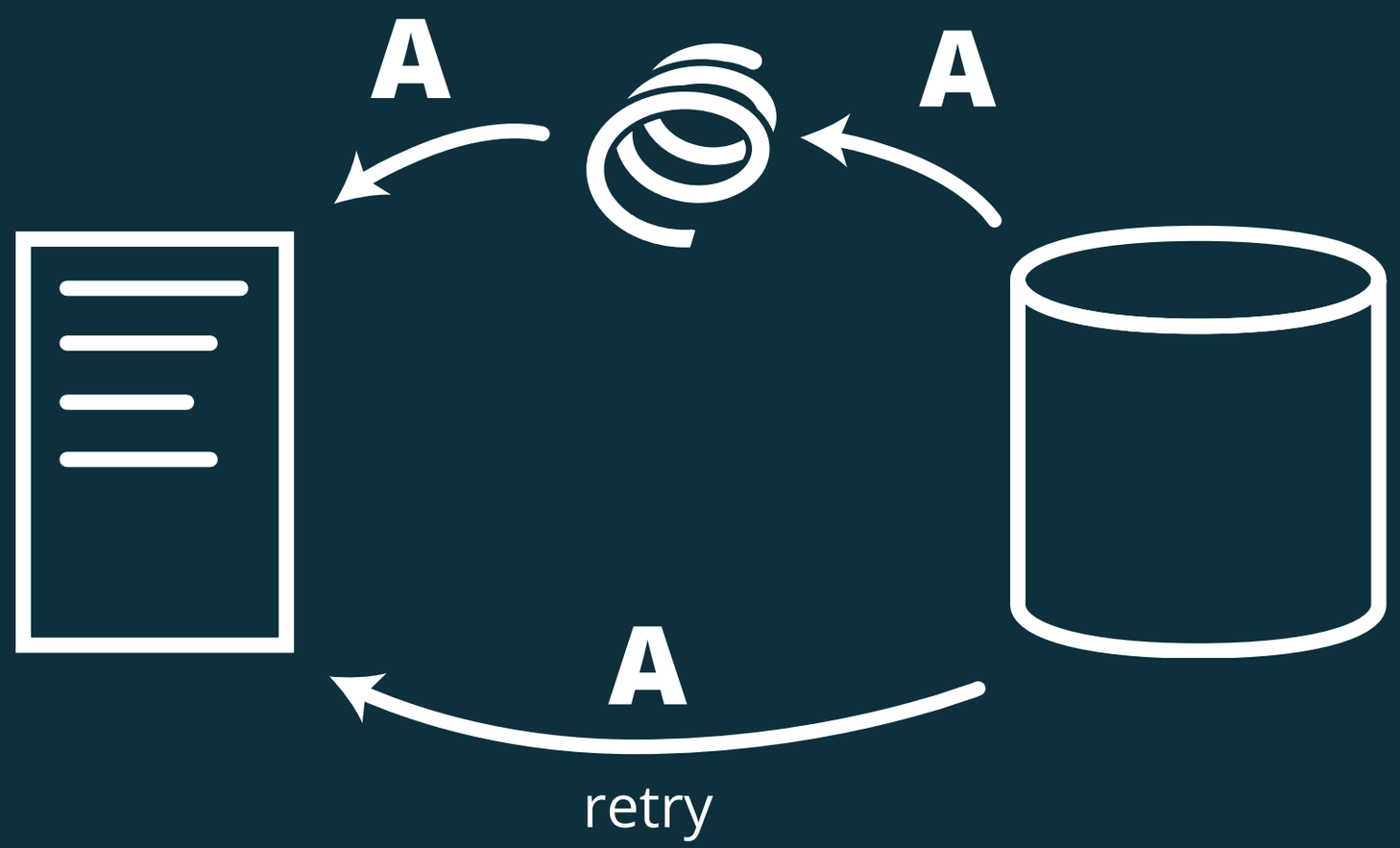
Repeatability? Yeah, right



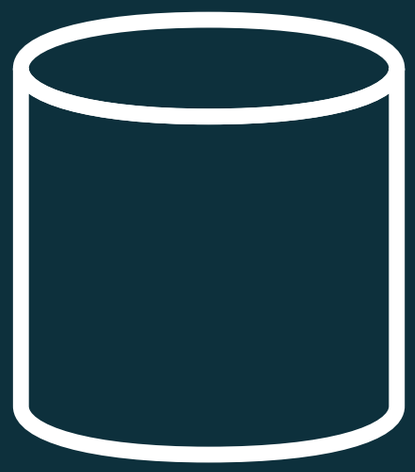
Repeatability? Yeah, right



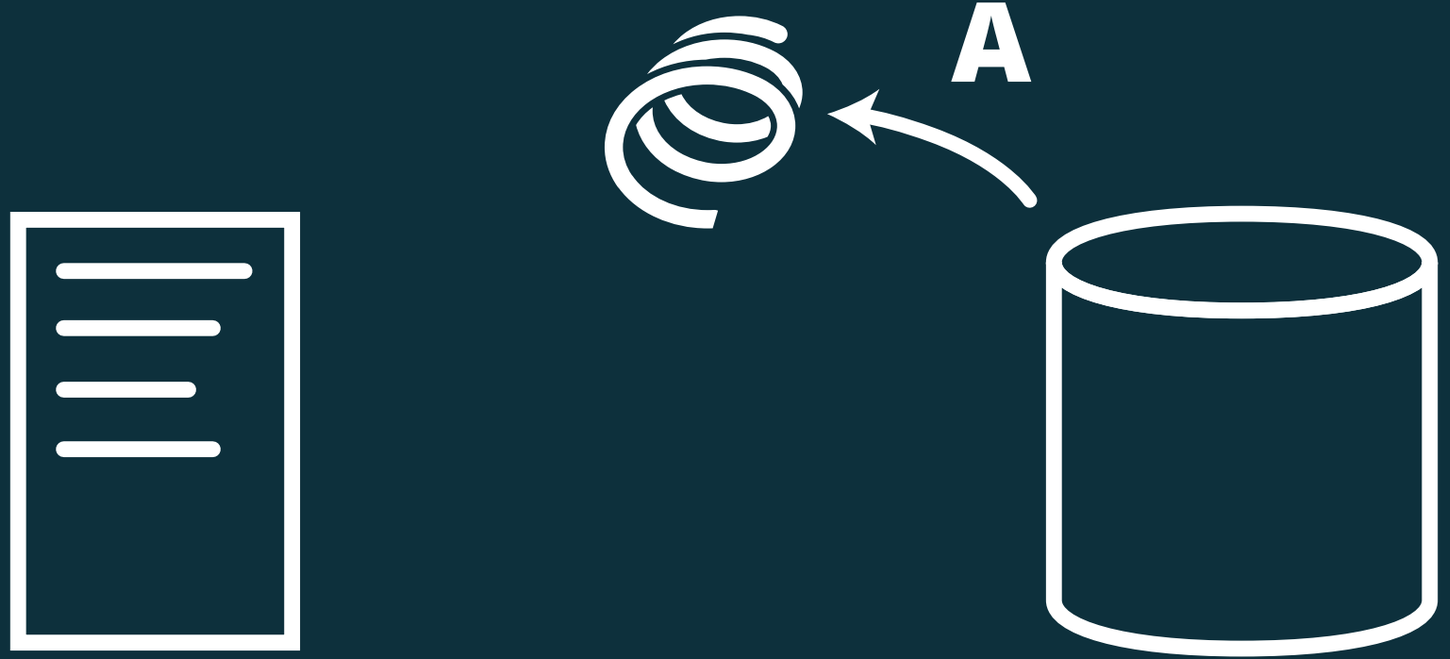
Repeatability? Yeah, right



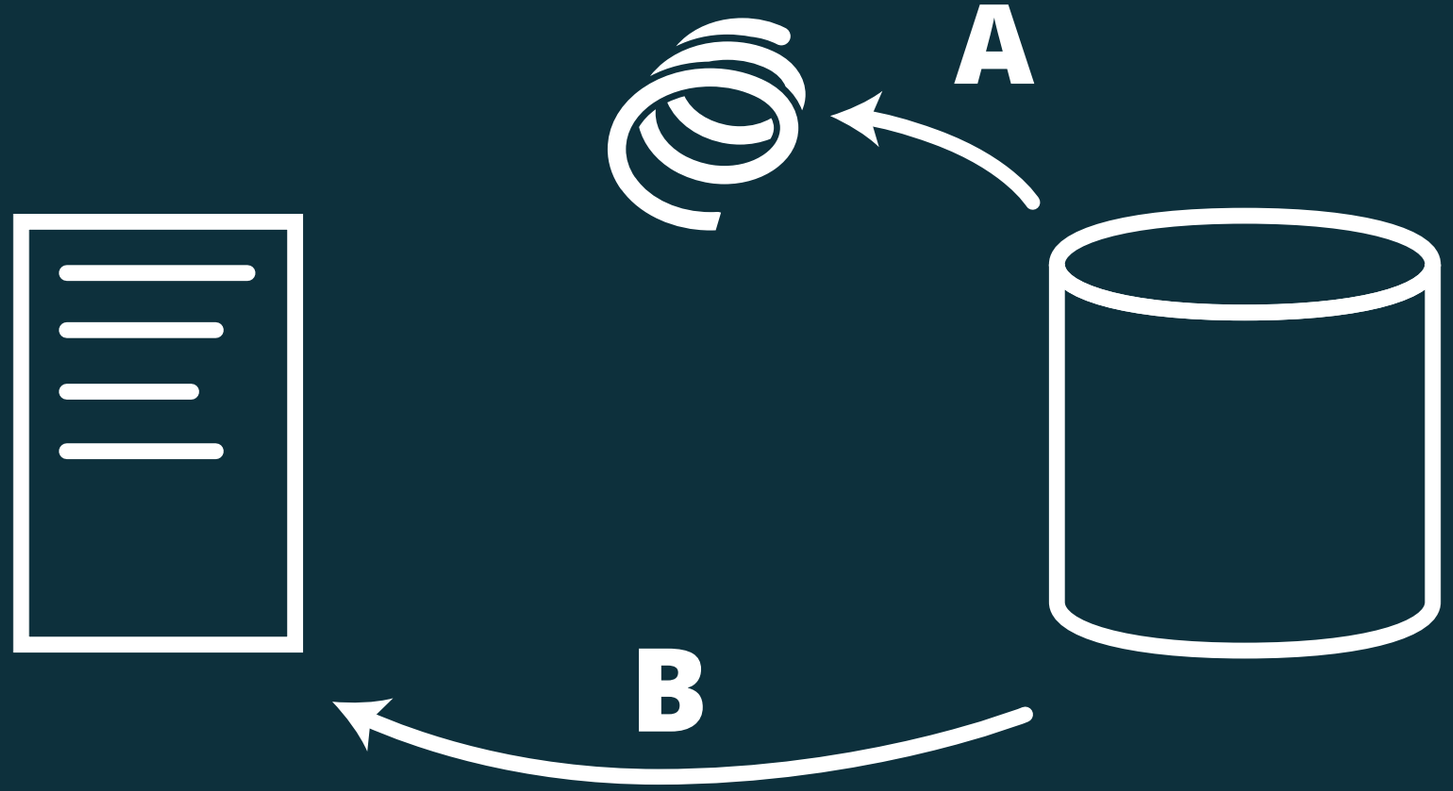
Lack of repeatability is a special case of non-determinism



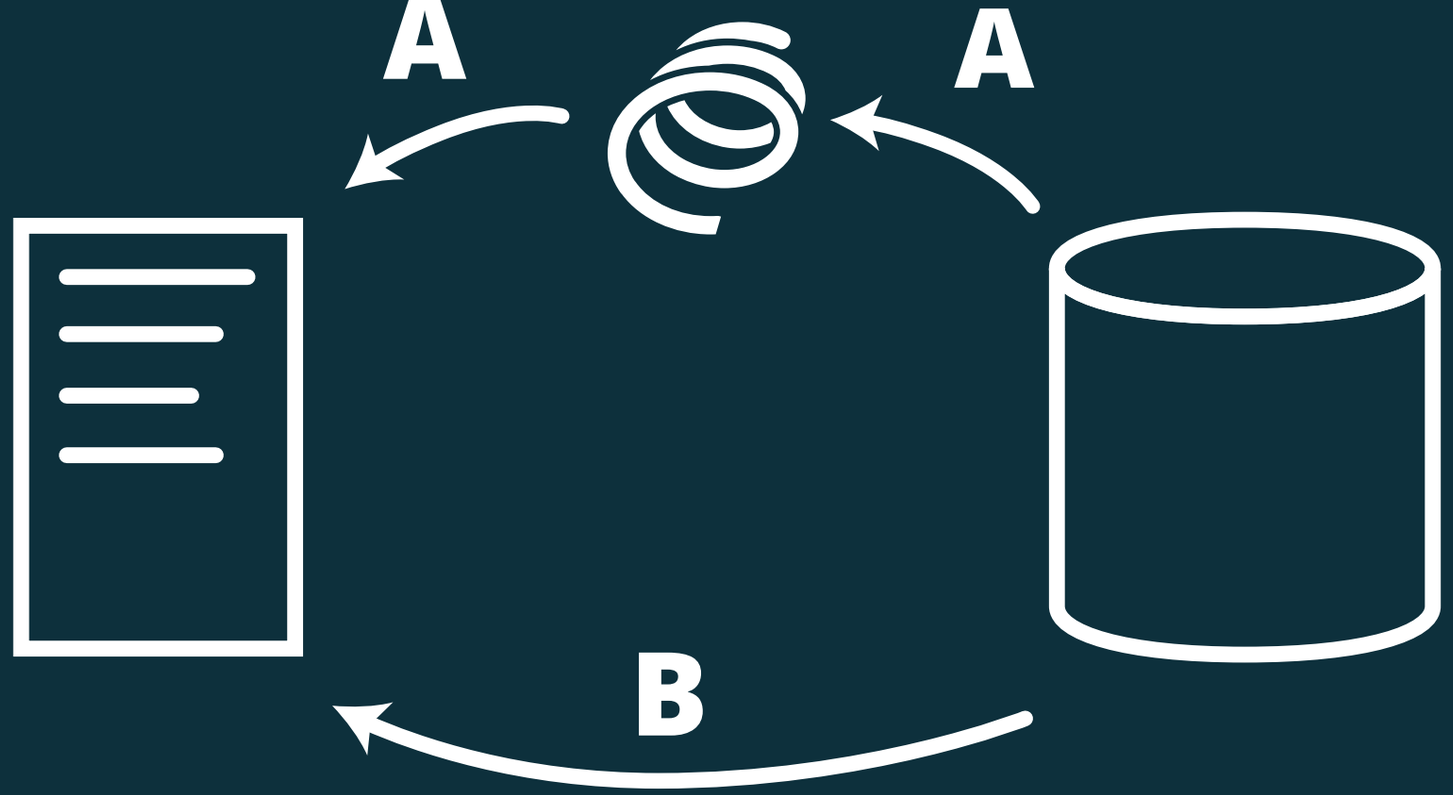
Lack of repeatability is a special case of non-determinism



Lack of repeatability is a special case of non-determinism



Lack of repeatability is a special case of non-determinism



FoundationDB is a distributed stateful system with ridiculously strong guarantees

ACID!

Don’t debug your system,

debug a simulation instead.

3 INGREDIENTS
of deterministic simulation

1. Single-threaded pseudo-concurrency
2. Simulated implementation of all external communication
3. Determinism

1 Single-threaded pseudo-concurrency

Flow

A syntactic extension to C++ , new keywords and actors, “compiled” down to regular C++ w/ callbacks

1 Single-threaded pseudo-concurrency

```
ACTOR Future<float> asyncAdd(Future<float> f,
                             float offset) {
  float value = wait( f );
  return value + offset;
}
```

1 Single-threaded pseudo-concurrency

```
class AsyncAddActor : public Actor, public FastAllocated<AsyncAddActor> {
public:
	AsyncAddActor(Future<float> const& f,float const& offset) : Actor("AsyncAddActor"),
		f(f),
		offset(offset),
		ChooseGroupbody1W1_1(this, &ChooseGroupbody1W1)
	{ }
	Future<float> a_start() {
		actorReturnValue.setActorToCancel(this);
		auto f = actorReturnValue.getFuture();
		a_body1();
		return f;
	}
private: // Implementation
...
};

Future<float> asyncAdd( Future<float> const& f, float const& offset ) {
	return (new AsyncAddActor(f, offset))->a_start();
}
```

1 Single-threaded pseudo-concurrency

```
int a_body1(int loopDepth=0) {
        try {
            ChooseGroupbody1W1_1.setFuture(f);
            loopDepth = a_body1alt1(loopDepth);
        }
        catch (Error& error) {
            loopDepth = a_body1Catch1(error, loopDepth);
        } catch (...) {
            loopDepth = a_body1Catch1(unknown_error(), loopDepth);
        }
        return loopDepth;
}
int a_body1cont1(float const& value,int loopDepth) {
        actorReturn(actorReturnValue, value + offset);
        return 0;
}
int a_body1when1(float const& value,int loopDepth) {
        loopDepth = a_body1cont1(value, loopDepth);
        return loopDepth;
}
```

1 Single-threaded pseudo-concurrency

```
template <class T, class V>
	void actorReturn(Promise<T>& actorReturnValue, V const& value) {
		T rval = value; // explicit copy, needed in the case the return is a state variable of the actor
		Promise<T> rv = std::move( actorReturnValue );
		// Break the ownership cycle this,this->actorReturnValue,this->actorReturnValue.sav,this->actorReturnValue.sav->actorToCancel
		rv.clearActorToCancel();
		try {
			delete this;
		} catch (...) {
			TraceEvent(SevError, "ActorDestructorError");
		}
		rv.send( rval );
	}
```

1 Single-threaded pseudo-concurrency

Programming language	Time in seconds
Ruby (using threads)	1990 sec
Ruby (using queues)	360 sec
Objective C (using threads)	26 sec
Java (using threads)	12 sec
Stackless Python	1.68 sec
Erlang	1.09 sec
Go	0.87 sec
Flow	0.075 sec

2 Simulated Implementation

INetwork

IConnection

IAsyncFile

...

SimNetwork

SimConnection

SimFile

...

2 Simulated Implementation

```
void connect( Reference<Sim2Conn> peer, NetworkAddress peerEndpoint ) {
        this->peer = peer;
        this->peerProcess = peer->process;
        this->peerId = peer->dbgid;
        this->peerEndpoint = peerEndpoint;

        // Every one-way connection gets a random permanent latency and a random send buffer
for the duration of the connection
        auto latency = g_clogging.setPairLatencyIfNotSet( peerProcess->address.ip, process-
>address.ip, FLOW_KNOBS->MAX_CLOGGING_LATENCY*g_random->random01() );
        sendBufSize = std::max<double>( g_random->randomInt(0, 5000000), 25e6 * (latency + .002) );
        TraceEvent("Sim2Connection").detail("SendBufSize", sendBufSize).detail("Latency", latency);
    }
```

2 Simulated Implementation

```
ACTOR static Future<Void> receiver( Sim2Conn* self ) {
	loop {
		if (self->sentBytes.get() != self->receivedBytes.get())
			Void _ = wait( g_simulator.onProcess( self->peerProcess ) );
		while ( self->sentBytes.get() == self->receivedBytes.get() )
			Void _ = wait( self->sentBytes.onChange() );
		ASSERT( g_simulator.getCurrentProcess() == self->peerProcess );
		state int64_t pos = g_random->random01() < .5 ? self->sentBytes.get() : g_random->randomInt64( self->receivedBytes.get(), self->sentBytes.get()+1 );
		Void _ = wait( delay( g_clogging.getSendDelay( self->process->address, self->peerProcess->address ) ) );
		Void _ = wait( g_simulator.onProcess( self->process ) );
		ASSERT( g_simulator.getCurrentProcess() == self->process );
		Void _ = wait( delay( g_clogging.getRecvDelay( self->process->address, self->peerProcess->address ) ) );
		ASSERT( g_simulator.getCurrentProcess() == self->process );
		self->receivedBytes.set( pos );
		Void _ = wait( Void() );  // Prior notification can delete self and cancel this actor
		ASSERT( g_simulator.getCurrentProcess() == self->process );
	}
}
```

2 Simulated Implementation

```
// Reads as many bytes as possible from the read buffer into [begin,end) and returns the number of bytes read (might be 0)
	// (or may throw an error if the connection dies)
	virtual int read( uint8_t* begin, uint8_t* end ) {
		rollRandomClose();

		int64_t avail = receivedBytes.get() - readBytes.get();  // SOMEDAY: random?
		int toRead = std::min<int64_t>( end-begin, avail );
		ASSERT( toRead >= 0 && toRead <= recvBuf.size() && toRead <= end-begin );
		for(int i=0; i<toRead; i++)
			begin[i] = recvBuf[i];
		recvBuf.erase( recvBuf.begin(), recvBuf.begin() + toRead );
		readBytes.set( readBytes.get() + toRead );
		return toRead;
	}
```

2 Simulated Implementation

```
void rollRandomClose() {
	if (g_simulator.enableConnectionFailures && g_random->random01() < .00001) {
		double a = g_random->random01(), b = g_random->random01();
		TEST(true);  // Simulated connection failure
		TraceEvent("ConnectionFailure", dbgid).detail("MyAddr", process-
>address).detail("PeerAddr", peerProcess->address).detail("SendClosed", a > .33).detail("RecvClosed",
a < .66).detail("Explicit", b < .3);
		if (a < .66 && peer) peer->closeInternal();
		if (a > .33) closeInternal();
		// At the moment, we occasionally notice the connection failed immediately.  In
principle, this could happen but only after a delay.
		if (b < .3)
			throw connection_failed();
	}
}
```

3 Determinism

We have (1) and (2), so all we need to add is none of the control flow of anything in your program depending on anything outside of your program + its inputs.

3 Determinism

Use unseeds to figure out whether what happened was deterministic

The Simulator

Test files

```
testTitle=SwizzledCycleTest
    testName=Cycle
    transactionsPerSecond=1000.0
    testDuration=30.0
    expectedRate=0.01

    testName=RandomClogging
    testDuration=30.0
    swizzle = 1

    testName=Attrition
    machinesToKill=10
    machinesToLeave=3
    reboot=true
    testDuration=30.0

    testName=ChangeConfig
    maxDelayBeforeChange=30.0
    coordinators=auto
```

Other Simulated disasters

In parallel with the workloads, we run some other things like:

- broken machine
- clogging
- swizzling
- nukes
- dumb sysadmin
- etc.

You are looking for bugs…
the universe is also
looking for bugs

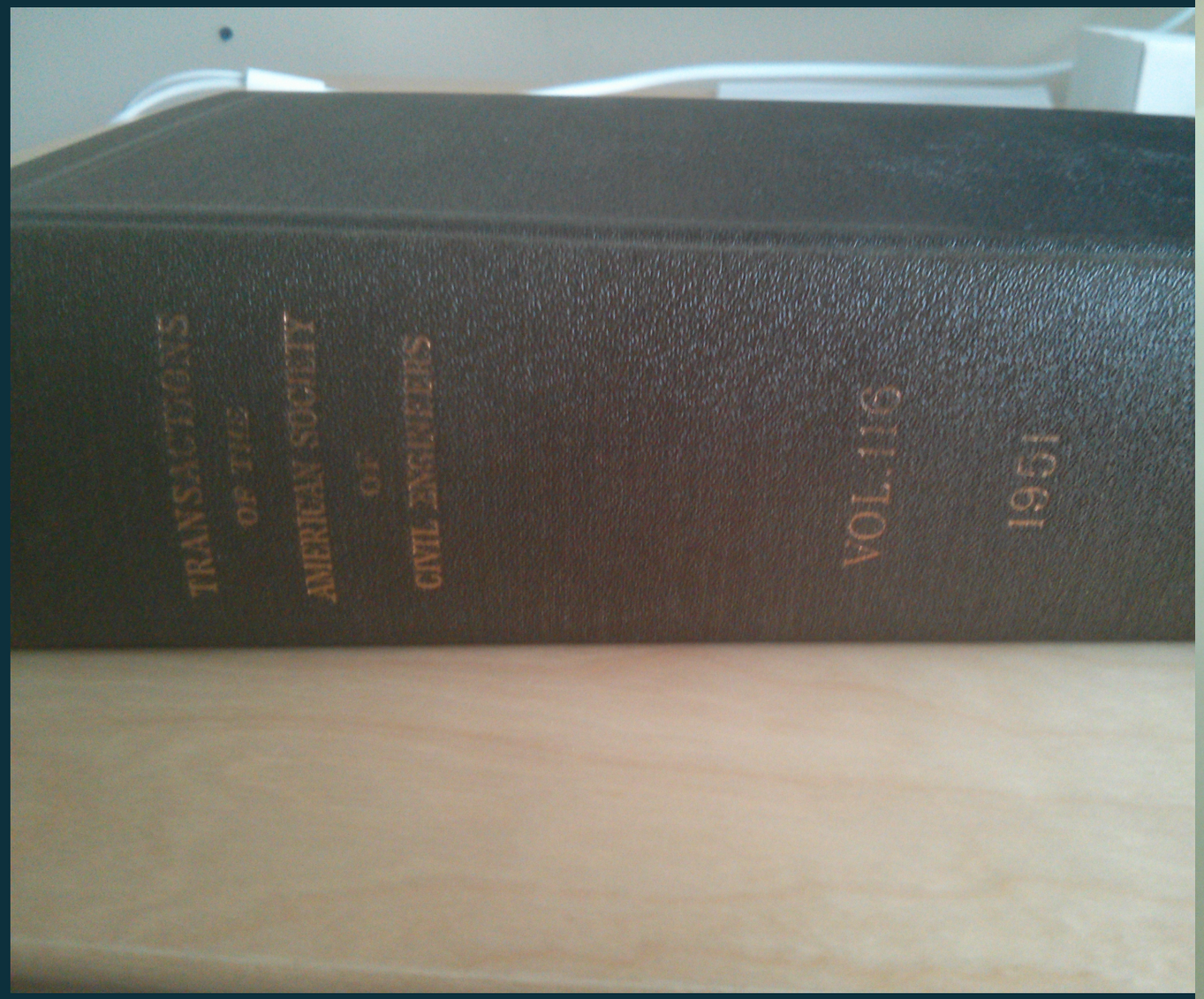
How can you make your
CPUs more efficient than
the universe’s?

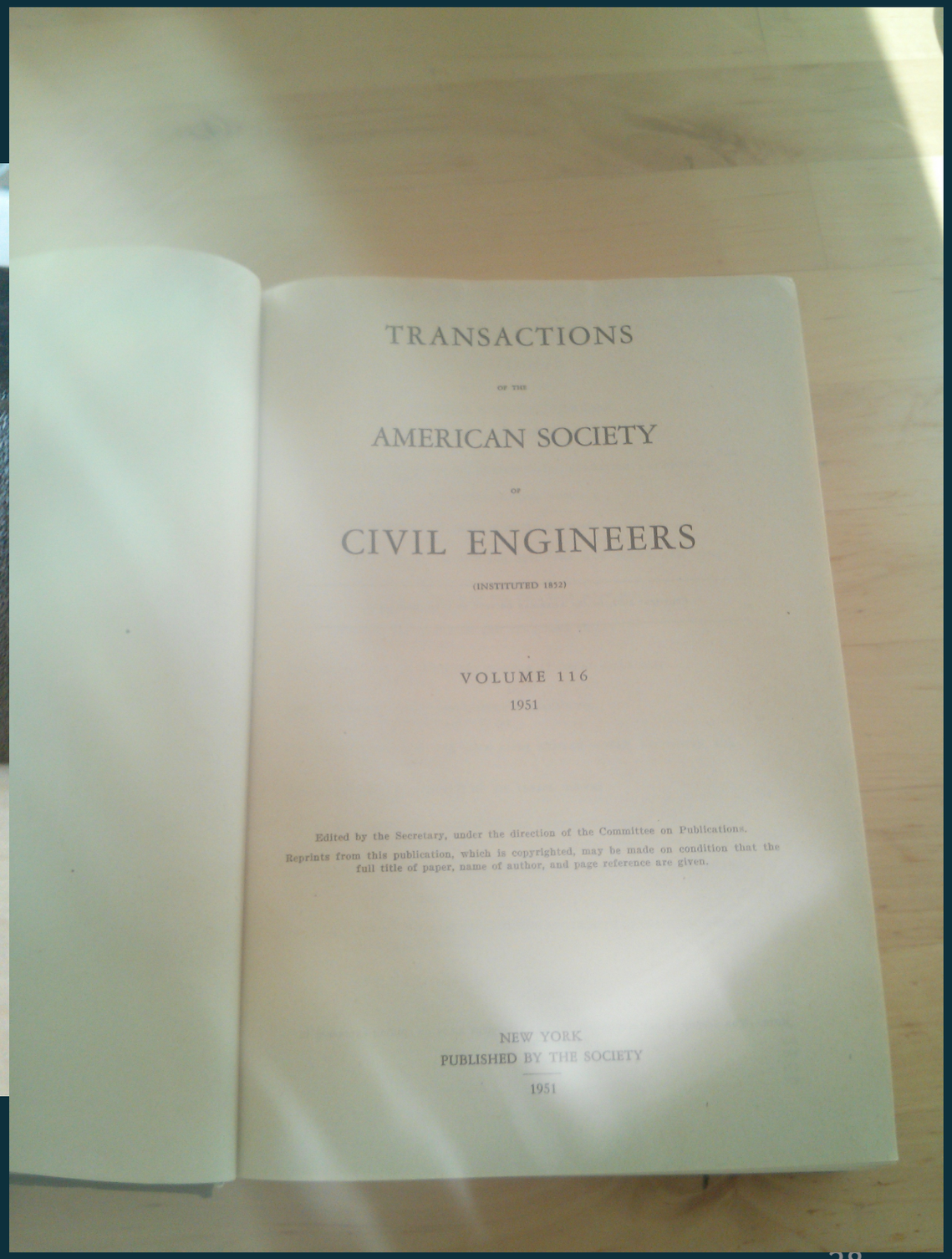
1. Disasters here happen more frequently than in the real world

2. Buggify!

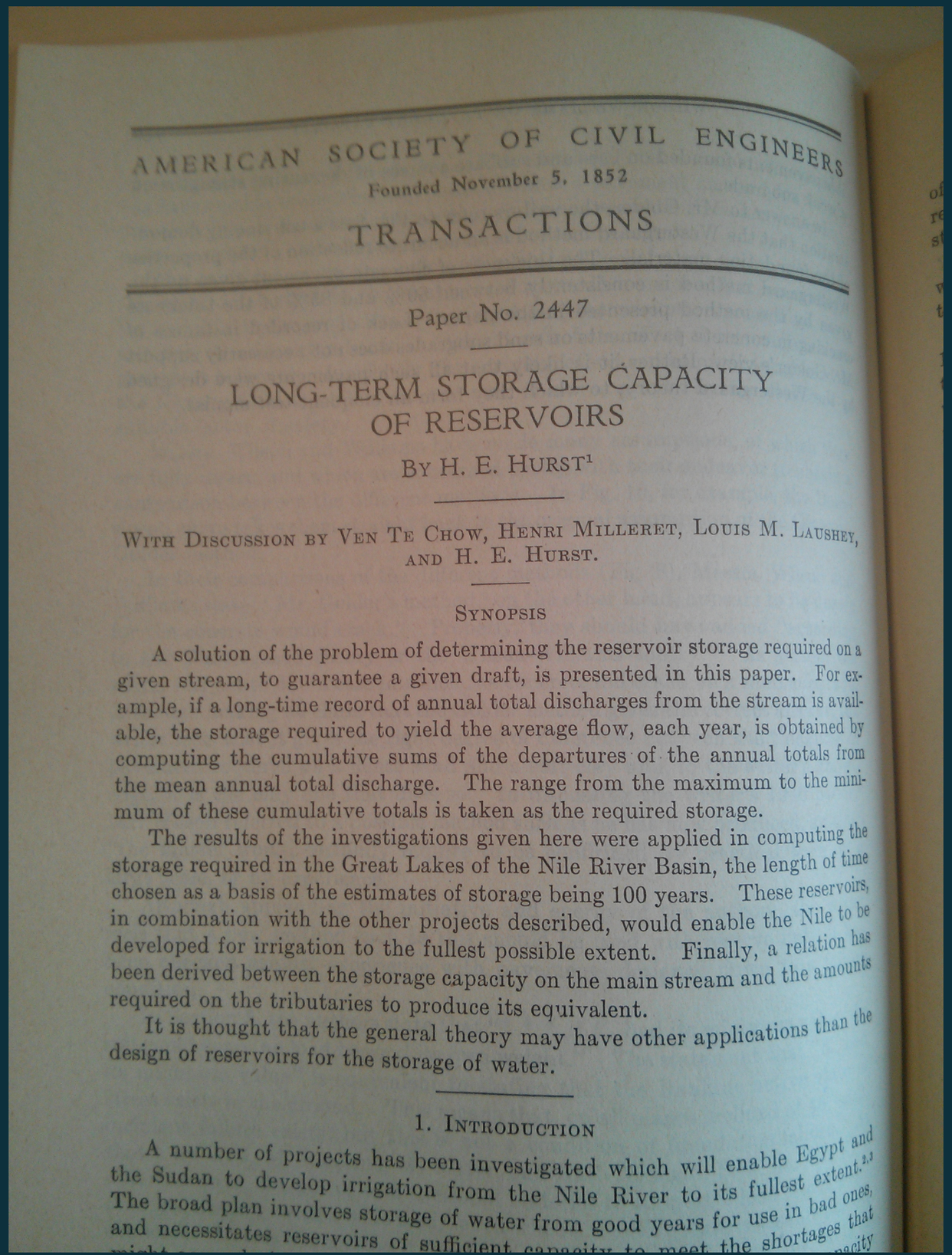
```
if (BUGGIFY) toSend = std::min(toSend, g_random->randomInt(0, 1000));

when( Void _ = wait( BUGGIFY ? Never() : delay( g_network->isSimulated() ? 20 :
900 ) ) ) {
	req.reply.sendError( timed_out() );
}
```

3. The Hurst exponent





3. The Hurst exponent



AMERICAN SOCIETY OF CIVIL ENGINEERS

Founded November 5, 1852

TRANSACTIONS

Paper No. 2447

LONG-TERM STORAGE CAPACITY OF RESERVOIRS

By H. E. Hurst[1]

With Discussion by Ven Te Chow, Henri Milleret, Louis M. Laushey, and H. E. Hurst.

Synopsis

A solution of the problem of determining the reservoir storage required on a given stream, to guarantee a given draft, is presented in this paper. For example, if a long-time record of annual total discharges from the stream is available, the storage required to yield the average flow, each year, is obtained by computing the cumulative sums of the departures of the annual totals from the mean annual total discharge. The range from the maximum to the minimum of these cumulative totals is taken as the required storage.

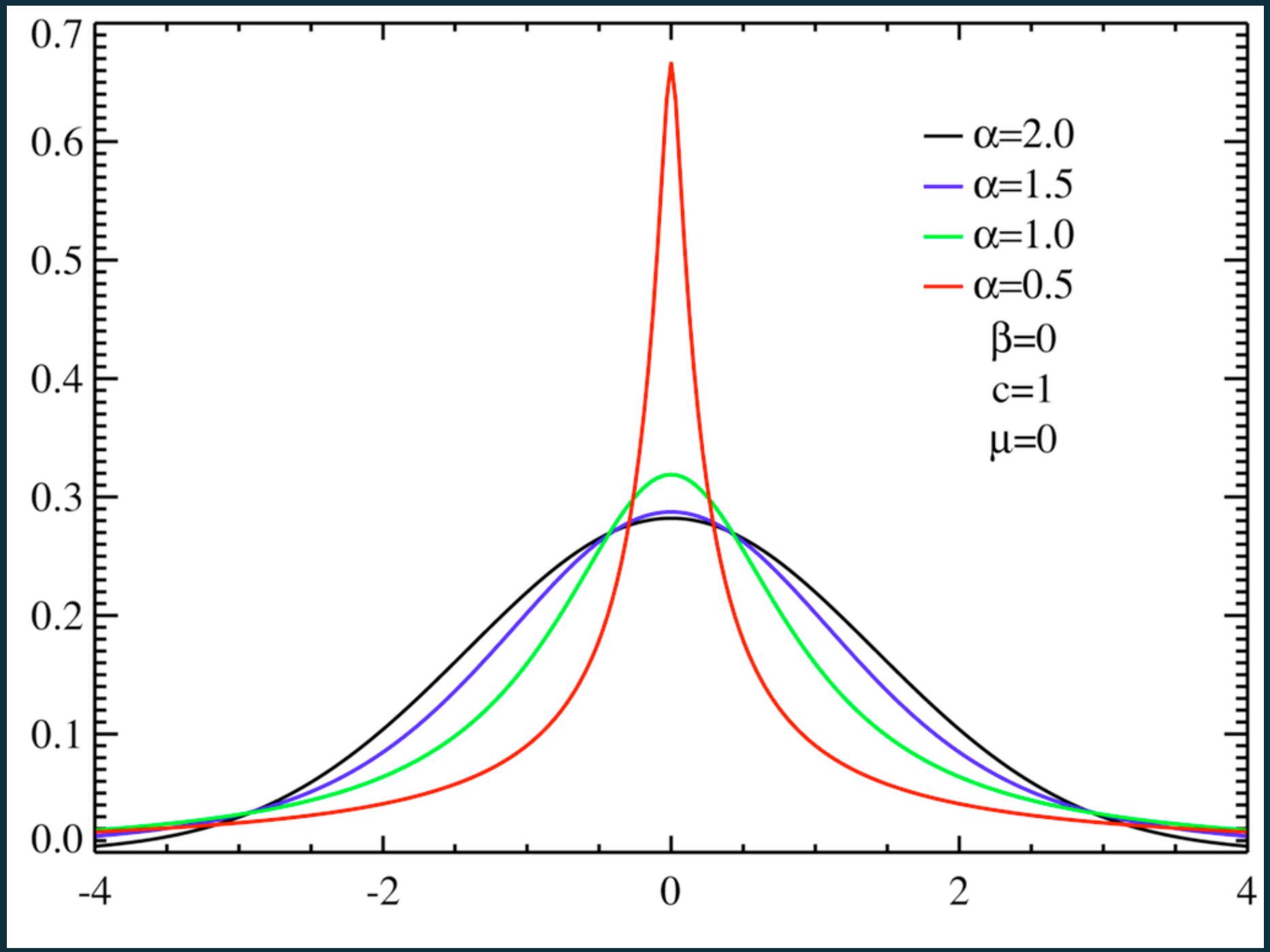
The results of the investigations given here were applied in computing the storage required in the Great Lakes of the Nile River Basin, the length of time chosen as a basis of the estimates of storage being 100 years. These reservoirs, in combination with the other projects described, would enable the Nile to be developed for irrigation to the fullest possible extent. Finally, a relation has been derived between the storage capacity on the main stream and the amounts required on the tributaries to produce its equivalent.

It is thought that the general theory may have other applications than the design of reservoirs for the storage of water.

1. Introduction

A number of projects has been investigated which will enable Egypt and the Sudan to develop irrigation from the Nile River to its fullest extent.[2,3] The broad plan involves storage of water from good years for use in bad ones, and necessitates reservoirs of sufficient capacity to meet the shortages that

3. The Hurst exponent

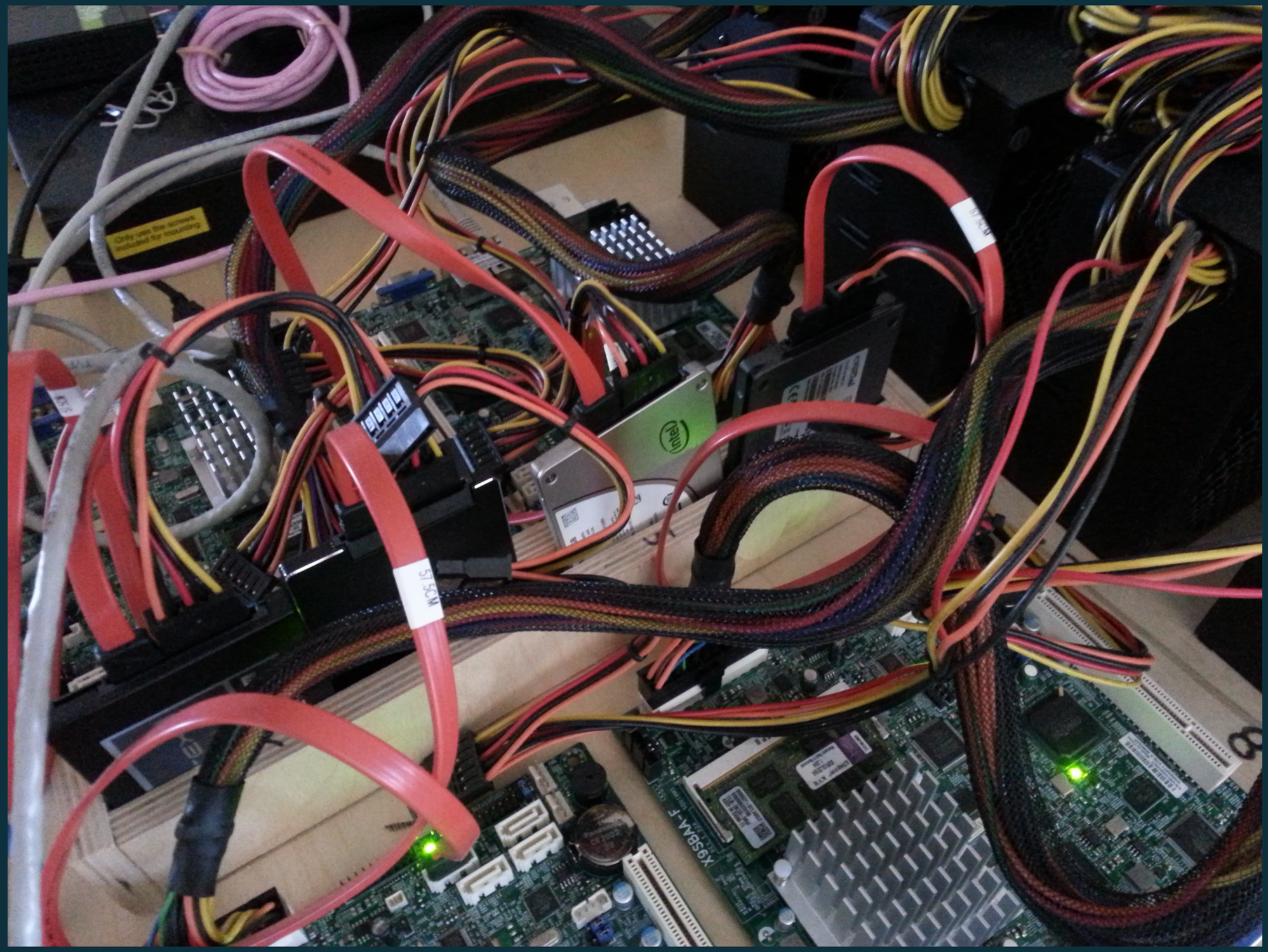


We estimate we've run the equivalent of *trillions of hours* of "real world" tests

BAD NEWS:

We broke debugging (but at least it’s possible)

Backup: Sinkhole



Two “bugs” found by Sinkhole

1. Power safety bug in ZK
2. Linux package managers writing conf files don’t sync

Future directions

1. Dedicated “red team”
2. More hardware
3. Try to catch bugs that “evolve”
4. More real world testing

Questions?

FOUNDATIONDB

will.wilson@foundationdb.com

@willwilsonFDB

Data Partitioning

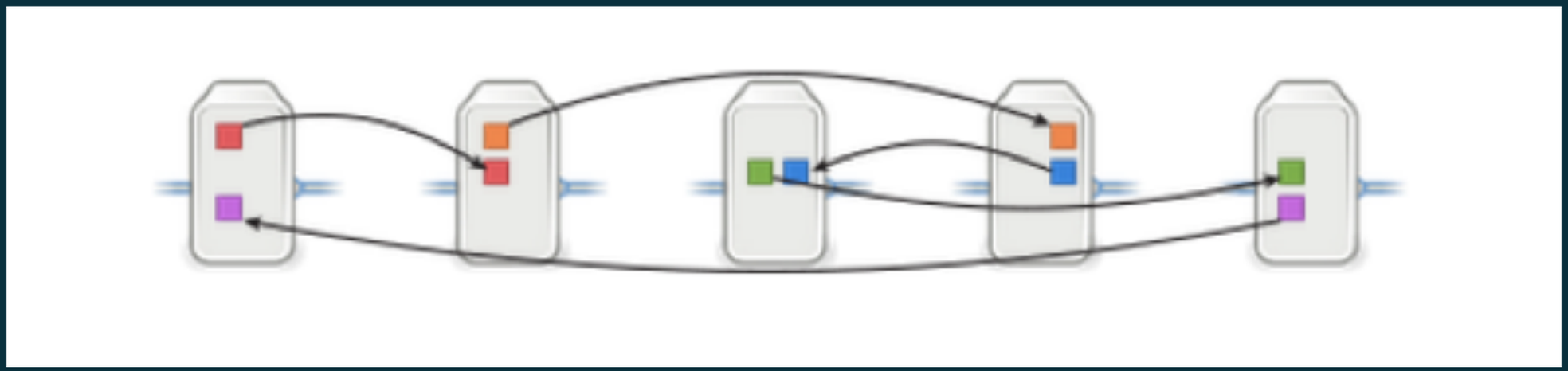
Keyspace is partitioned onto different machines for scalability

Data Replication

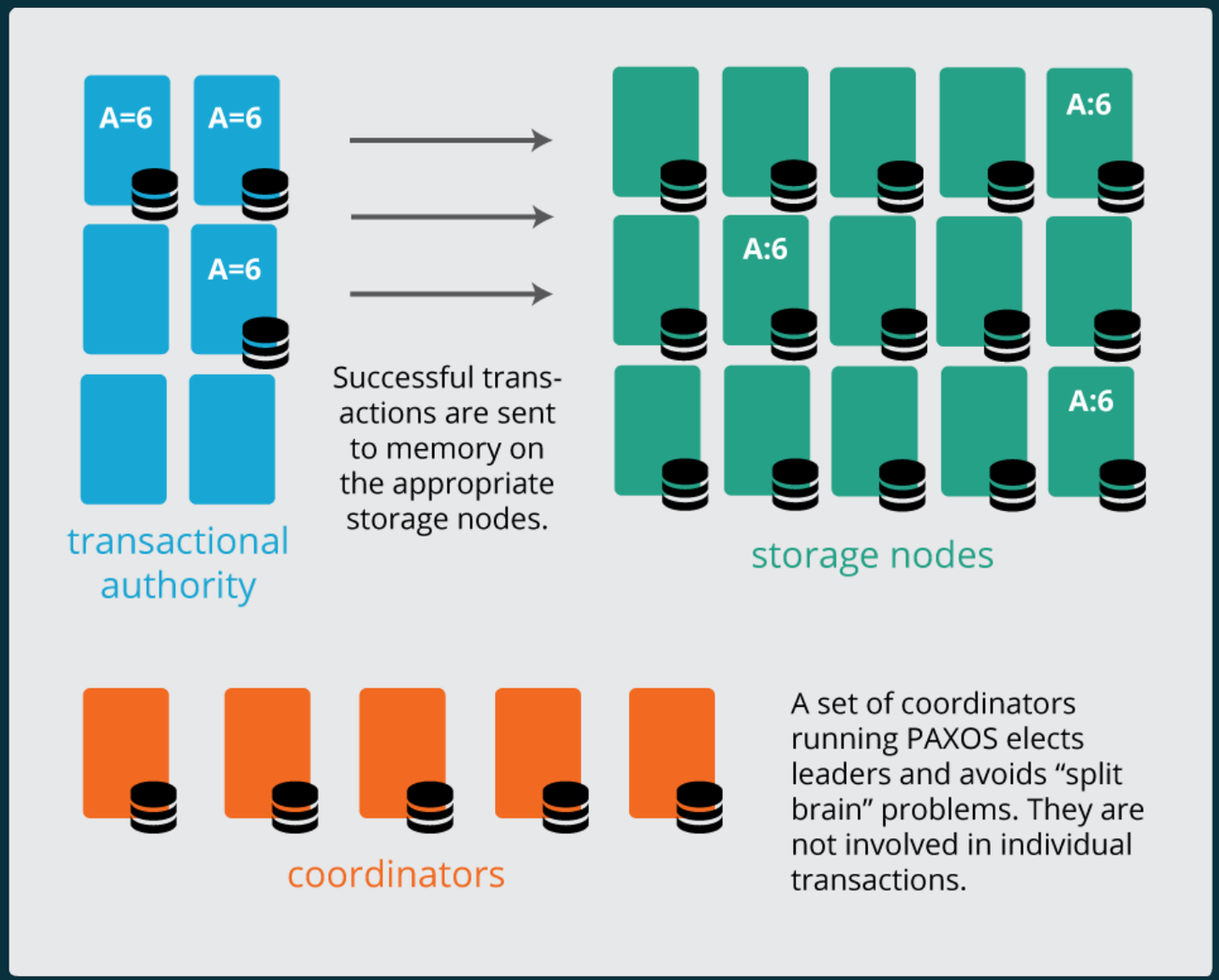
The same keys are copied onto different machines for fault tolerance

FoundationDB

Partitioning + replication for both scalability and fault tolerance

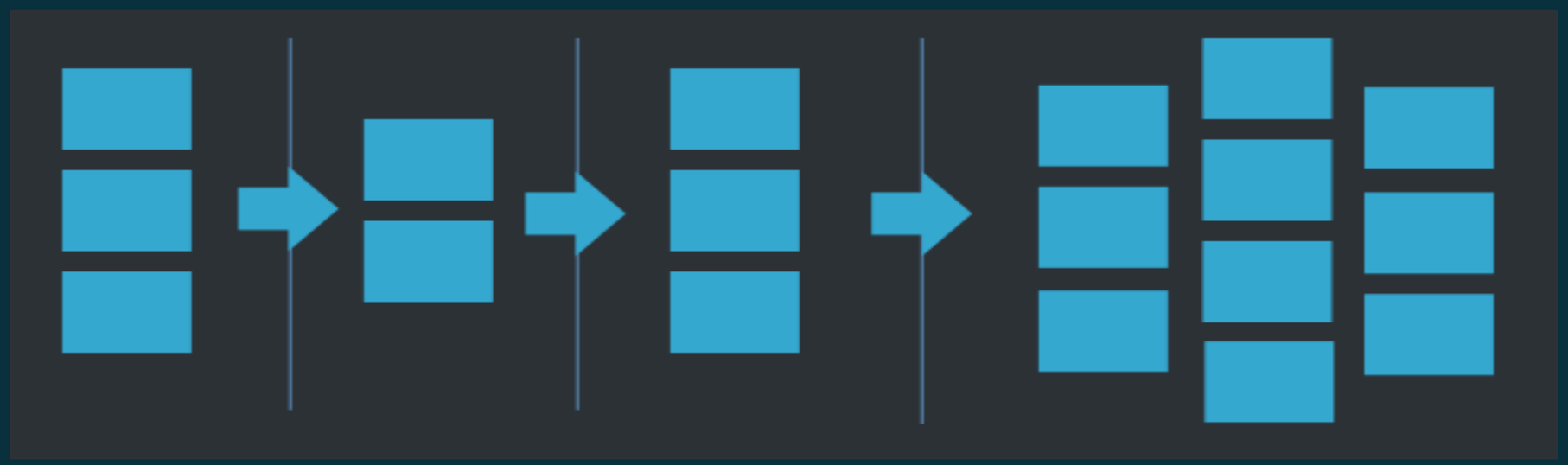


Architecture of FoundationDB



Transaction Processing

- Decompose the transaction processing pipeline into stages
- Each stage is distributed



Pipeline Stages

- Accept client transactions
- Apply conflict checking
- Write to transaction logs
- Update persistent data structures

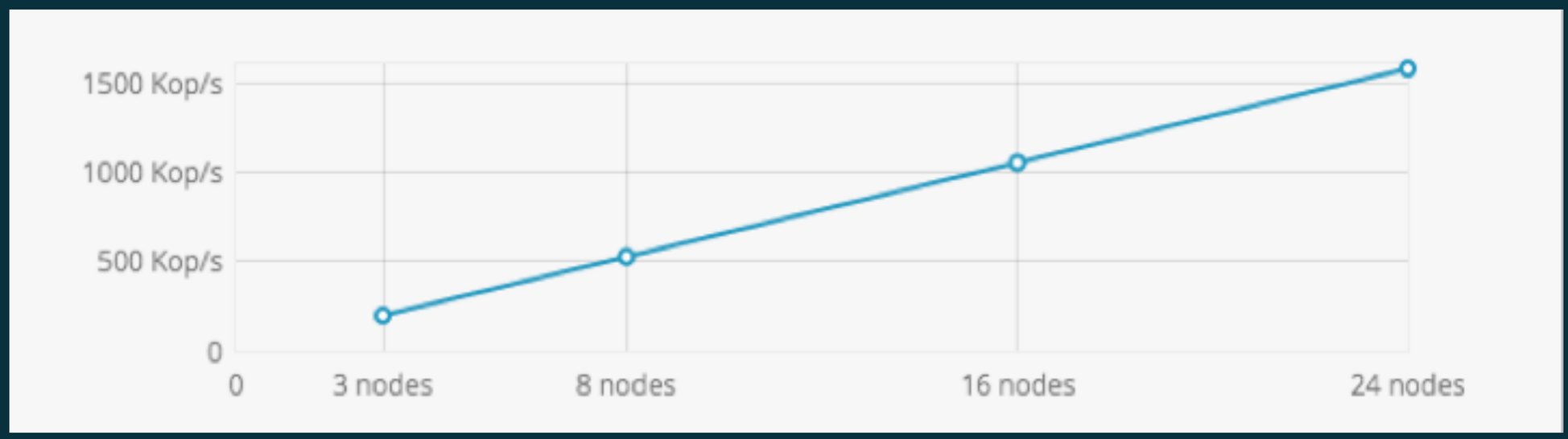
High Performance!

Performance Results

- 24 machine cluster
- 100% cross-node transactions
- Saturates its SSDs

890,000 ops/sec

Scalability Results



Performance by cluster size