

# Testing Distributed Systems for Linearizability

4 Jun 2017 — shared on [Lobsters](#), [Reddit](#), and [Twitter](#)

Distributed systems are challenging to implement correctly because they must handle concurrency and failure. Networks can delay, duplicate, reorder, and drop packets, and machines can fail at any time. Even when designs are proven correct on paper, it is difficult to avoid subtle bugs in implementations.

Unless we want to use formal methods<sup>1</sup>, we have to test systems if we want assurance that implementations are correct. Testing distributed systems is challenging, too. Concurrency and nondeterminism make it difficult to catch bugs in tests, especially when the most subtle bugs surface only under scenarios that are uncommon in regular operation, such as simultaneous machine failure or extreme network delays.

## Correctness

Before we can discuss testing distributed systems for correctness, we need to define what we mean by “correct”. Even for seemingly simple systems, specifying exactly how the system is supposed to behave is an involved process<sup>2</sup>.

Consider a simple key-value store, similar to [etcd](#), that maps strings to strings and supports two operations: `Put(key, value)` and `Get(key)`. First, we consider how it behaves in the sequential case.

## Sequential Specifications

We probably have a good intuitive understanding of how a key-value store is supposed to behave under sequential operation: `Get` operations must reflect the result of applying all previous `Put` operations. For example, we could run a `Put("x", "y")` and then a subsequent `Get("x")` should return `"y"`. If the operation returned, say, a `"z"`, that would be incorrect.

More formal than an English-language description, we can write a specification for our key-value store as executable code:

```
class KVStore:
    def __init__(self):
        self._data = {}
```

```
def put(self, key, value):  
    self._data[key] = value  
  
def get(self, key):  
    return self._data.get(key, "")
```

The code is short, but it nails down all the important details: the start state, how the internal state is modified as a result of operations, and what values are returned as a result of calls on the key-value store. The spec solidifies some details like what happens when `Get()` is called on a nonexistent key, but in general, it lines up with our intuitive definition of a key-value store.

## Linearizability

Next, we consider how our key-value store can behave under concurrent operation. Note that the sequential specification does **not** tell us what happens under concurrent operation. For example, the sequential spec doesn't say how our key-value store is allowed to behave in this scenario:

It's not immediately obvious what value the `Get("x")` operation should be allowed to return. Intuitively, we might say that because the `Get("x")` is concurrent with the `Put("x", "y")` and `Put("x", "z")`, it can return either value or even `""`. If we had a situation where another client executed a `Get("x")` much later, we might say that the

operation must return "z" , because that was the value written by the last write, and the last write operation was not concurrent with any other writes.

We formally specify correctness for concurrent operations based on a sequential specification using a consistency model known as **linearizability**. In a linearizable system, **every operation appears to execute atomically and instantaneously at some point between the invocation and response**. There are other consistency models besides linearizability, but many distributed systems provide linearizable behavior: linearizability is a strong consistency model, so it's relatively easy to build other systems on top of linearizable systems.

Consider an example history with invocations and return values of operations on a key-value store:

This history **is linearizable**. We can show this by explicitly finding linearization points for all operations (drawn in orange below). The induced sequential history, `Put("x", "0") , Get("x") -> "0" , Put("x", "1") , Get("x") -> "1"` , is a correct history with respect to the sequential specification.

In contrast, this history is **not** linearizable:

There is no linearization of this history with respect to the sequential specification: there is no way to assign linearization points to operations in this history. We could start assigning

linearization points to the operations from clients 1, 2, and 3, but then there would be no way to assign a linearization point for client 4: it would be observing a stale value. Similarly, we could start assigning linearization points to the operations from clients 1, 2, and 4, but then the linearization point of client 2's operation would be after the start of client 4's operation, and then we wouldn't be able to assign a linearization point for client 3: it could legally only read a value of "" or "0" .

## Testing

With a solid definition of correctness, we can think about how to test distributed systems. The general approach is to test for correct operation while randomly injecting faults such as machine failures and network partitions. We could even simulate the entire network so it's possible to do things like cause extremely long network delays. Because tests are randomized, we would want to run them a bunch of times to gain assurance that a system implementation is correct.

## Ad-hoc testing

How do we actually test for correct operation? With the simplest software, we test it using input-output cases like `assert(expected_output == f(input))` . We could use a similar approach with distributed systems. For example, with our key-value store, we could have the following test where multiple clients are executing operations on the key-value store in parallel:

```
for client_id = 0..10 {
  spawn thread {
    for i = 0..1000 {
      value = rand()
      kvstore.put(client_id, value)
      assert(kvstore.get(client_id) == value)
    }
  }
}
wait for threads
```

It is certainly the case that if the above test fails, then the key-value store is not linearizable. However, this test is not that thorough: there are non-linearizable key-value stores that would always pass this test.

## Linearizability

A better test would be to have parallel clients run **completely random operations**: e.g. repeatedly calling `kvstore.put(rand(), rand())` and `kvstore.get(rand())`, perhaps limited to a small set of keys to increase contention. But in this case, how would we determine what is “correct” operation? With the simpler test, we had each client operating on a separate key, so we could always predict exactly what the output had to be.

When clients are operating concurrently on the same set of keys, things get more complicated: we can't predict what the output of every operation has to be because there isn't only one right answer. So we have to take an alternative approach: we can test for correctness by recording an entire history of operations on the system and then checking if the history is linearizable with respect to the sequential specification.

## Linearizability Checking

A linearizability checker takes as input a sequential specification and a concurrent history, and it runs a decision procedure to check whether the history is linearizable with respect to the spec.

## NP-Completeness

Unfortunately, linearizability checking is NP-complete. The proof is actually quite simple: we can show that linearizability checking is in NP, and we can show that an NP-hard problem can be reduced to linearizability checking. Clearly, linearizability checking is in NP: given a linearization, i.e. the linearization points of all operations, we can check in polynomial time if it is a valid linearization with respect to the sequential spec.

To show that linearizability checking is NP-hard, we can reduce the subset sum problem to linearizability checking. Recall that in the subset sum problem, we are given a set  $S = \{s_1, s_2, \dots, s_n\}$  of non-negative integers and a target value  $t$ , and we have to determine whether there exists a subset of  $S$  that sums to  $t$ . We can reduce this problem to linearizability checking as follows. Consider the sequential spec:

```
class Adder:
    def __init__(self):
        self._total = 0

    def add(self, value):
        self._total += value

    def get(self):
        return self._total
```

And consider this history:

This history is linearizable if and only if the answer to the subset sum problem is “yes”. If the history is linearizable, then we can take all the operations `Add(s_i)` that have linearization points before that of the `Get()` operation, and those correspond to elements  $s_i$  in a subset whose sum is  $t$ . If the set does have a subset that sums to  $t$ , then we can construct a linearization by having the operations `Add(s_i)` corresponding to the elements  $s_i$  in the subset take place before the `Get()` operation and having the rest of the operations take place after the `Get()` operation.

## Implementation

Even though linearizability checking is NP-complete, in practice, it can work pretty well on small histories. Implementations of linearizability checkers take an executable specification along with a history, and they run a search procedure to try to construct a linearization, using tricks to constrain the size of the search space.


There are existing linearizability checkers like Knossos, which is used in the Jepsen test system. Unfortunately, when trying to test an implementation of a distributed key-value store that I had written, I couldn’t get Knossos to check my histories. It seemed to work okay on histories with a couple concurrent clients, with about a hundred history events in total, but in my tests, I had tens of clients generating histories of thousands of events.

To be able to test my key-value store, I wrote Porcupine, a fast linearizability checker implemented in Go. Porcupine checks if histories are linearizable with respect to executable specifications written in Go. Empirically, Porcupine is thousands of times faster than Knossos. I was able to use it to test my key-value store because it is capable of checking histories of thousands of events in a couple seconds.

## Effectiveness

Testing linearizable distributed systems using fault injection along with linearizability checking is an effective approach.

To compare ad-hoc testing with linearizability checking using Porcupine, I tried testing my distributed key-value store using the two approaches. I tried introducing different kinds of design bugs into the implementation of the key-value store, such as modifications that would result in stale reads, and I checked to see which tests failed. The ad-hoc tests caught some of the most egregious bugs, but the tests were incapable of catching the more subtle bugs. In contrast, I couldn't introduce a single correctness bug that the linearizability test couldn't catch.

- 
1. Formal methods can provide strong guarantees about the correctness of distributed systems. For example, the UW PLSE research group has recently verified an implementation of the Raft consensus protocol using the Coq proof assistant. Unfortunately, verification requires specialized knowledge, and verifying realistic systems involves huge effort. Perhaps one day systems used in the real world will be proven correct, but for now, production systems are tested but not verified. 
  2. Ideally, all production systems would have formal specifications. Some systems that are being used in the real world today do have formal specs: for example, Raft has a formal spec written in TLA+. But unfortunately, the majority of real-world systems do not have formal specs. 