Q: What is linearizability?

A: Linearizability is one way to define correctness for how a service behaves in the face of concurrent client requests. Roughly speaking, it specifies that the service should appear as if it executes requests one at a time as they arrive.

Linearizability is defined on "histories": traces of actual client requests and server responses, annotated by the times at which clients send and receive each message. Linearizability tells you if an individual history is legal; we can say that a service is linearizable if every history it can generate is linearizable.

For each client request, the request message to the server and the corresponding response message are separate elements in the history, each message appearing at the time at which the client sent or received it. So histories make request concurrency and network delays explicit.

A history is linearizable if you can assign a "linearization point" (a time) to each operation, where each operation's point lies between the times the client sent the request and received the response, and the history's response values are the same as you'd get if you executed the requests one at a time in point order. If no assignment of linearization points satisfies these two requirements, the history is not linearizable.

An important consequence of linearizability is that the service has freedom in the order in which it executes concurrent (overlapping-in-time) operations. In particular, if requests from client C1 and C2 are concurrent, the server could execute C2's request first even if C1 send its request message before C2. On the other hand, if C1 received a response before C2 sent its request, linearizability requires the service to act as if it executed C1's request before C2's (i.e. C2's operation is required to observe the effects of C1's operation, if any).

Q: How do linearizability checkers work?

A: A simple linearizability checker would try every possible order (or choice of linearization points) to see if one is valid according to the rules in the definition of linearizability. Because that would be too slow on big histories, clever checkers avoid looking at clearly impossible orders (e.g. if a proposed linearization point is before the operation's start time), decompose the history into sub-histories that can be checked separately when that's possible, and use heuristics to try more likely orders first.

These papers describe the techniques; I believe Knossos is based on the first paper, and Porcupine adds ideas from the second paper:

    http://www.cs.ox.ac.uk/people/gavin.lowe/LinearizabiltyTesting/paper.pdf
    https://arxiv.org/pdf/1504.00204.pdf

Q: Do services implement linearizability using linearizability checkers?

A: No; checkers are only used as part of testing.

Q: So how do services implement linearizability?

A: If the service is implemented as a single server, with no replication or caching or internal parallelism, it's nearly enough for the service to execute client requests one a time as they arrive. The main complication comes from clients that re-send requests because

they think the network has lost messages: for requests with side-effects, the service must take care not to execute any given client request more than once. More complex designs are required if the service involves replication or caching.

Q: Do you know of any examples of real world systems tested with Porcupine or similar testing frameworks?

A: Such testing is common -- for example, have a look at https://jepsen.io/analyses; Jepsen is an organization that has tested the correctness (and linearizability, where appropriate) of many storage systems.

For Porcupine specifically, here's an example:

  https://www.vldb.org/pvldb/vol15/p2201-zare.pdf

Q: What are other consistency models?

A: Look for

    eventual consistency
    causal consistency
    fork consistency
    serializability
    sequential consistency
    timeline consistency

And there are others from the worlds of databases, CPU memory/cache systems, and file systems.

In general, different models differ in how intuitive they are for application programmers, and how much performance you can get with them. For example, eventual consistency allows many anomalous results (e.g. even if a write has completed, subsequent reads might not see it), but in a distributed/replicated setting can be implemented with higher performance than linearizability.

Q: Why is linearizability called a strong consistency model?

A: It is strong in the sense of forbidding many situations that might surprise application programmers.

For example, if I call put(x, 22), and my put completes, and nobody else writes x, and subsequently you call get(x), you're guaranteed to see no value other than 22. That is, reads see fresh data.

As another example, if no-one is writing x, and I call get(x), and you call get(x), we won't see different values.

These properties are not true of some other consistency models we'll look at, for example eventual and causal consistency. These latter models are often called "weak".

Q: What do people do in practice to ensure their distributed systems are correct?

A: I suspect that thorough testing is the most common plan.

Use of formal methods is also common; have a look here for some examples:

https://arxiv.org/pdf/2210.13661.pdf

https://assets.amazon.science/67/f9/92733d574c11ba1a11bd08bfb8ae/how-amazon-web-services-

uses-formal-methods.pdf

https://dl.acm.org/doi/abs/10.1145/3477132.3483540

https://www.ccs.neu.edu/~stavros/papers/2022-cpp-published.pdf

https://www.cs.purdue.edu/homes/pfonseca/papers/eurosys2017-dsbugs.pdf

Q: Why is linearizability used as a consistency model versus other ones, such as eventual consistency?

A: People do often build storage systems that provide consistency weaker than linearizability, such as eventual and causal consistency.

Linearizability has some nice properties for application writers:

* reads always observe fresh data.
* if there are no concurrent writes, all readers see the same data.
* on most linearizable systems you can add mini-transactions like test-and-set (because most linearizable designs end up executing operations on each data item one-at-a-time).

Weaker schemes like eventual and causal consistency can allow higher performance, since they don't require all copies of data to be updated right away. This higher performance is often the deciding factor. However, weak consistency introduces some complexity for application writers:

* reads can observe out-of-date (stale) data.
* reads can observe writes out of order.
* if you write, and then read, you may not see your write, but instead see stale data.
* concurrent updates to the same items aren't executed one-at-a-time, so it's hard to to implement mini-transactions like test-and-set.

Q: How do you decide where little orange line for linearizability goes - the linearization point for an operation? On the diagram it looks like it's randomly drawn somewhere within the body of the request?

A: The idea is that, in order to show that an execution is linearizable, you (the human) need to find places to put the little orange lines (linearization points). That is, in order to show that a history is linearizable, you need to find an order of operations that conforms to these requirements:

* All function calls have a linearization point at some instant between their invocation and their response.

* All functions appear to occur instantly at their linearization point, behaving as specified by the sequential definition.

So, some placements of linearization points are invalid because they lie outside of the time span of a request; others are invalid because they violate the sequential definition (for a key/value store, a violation means that a read does not observe the most recently written value, where "recent" refers to linearization points).

In complex situations you may need to try many combinations of orders of linearization points in order to find one that demonstrates that the history is linearizable. If you try them all, and none works, then the history is not linearizable.

Q: Is it ever the case that if two commands are executing at the same

time, we are able to enforce a specific behavior such that one command always executes first (as in it always has an earlier linearization point)?

A: In a linearizable storage service (e.g. GFS, or your Lab 3), if requests from multiple clients arrive at about the same time, the service can choose the order in which it executes them. Although in practice most services execute concurrent requests in the order in which the request packets happen to arrive on the network.

The notion of linearization point is part of one strategy for checking whether a history is linearizable. Actual implementations do not usually involve an explicit notion of linearization point. Instead, they usually just execute incoming requests in some serial (one-at-a-time) order. You can view each operation's linearization point as occuring somewhere during the time in which the service is executing the request.

Q: What other types of consistency checks can we perform that are stronger? Somehow, linearizability doesn't intuitively *feel* very helpful, since you can be reading different data even when you execute two commands at the same time.

A: True, linearizability is reminiscent of using threads in a program without using locks -- any concurrent accesses to the same data are races. It's possible to program correctly this way but it requires care.

The next strongest consistency notions involve transactions, as found in many databases, which effectively lock any data used. For programs that read and write multiple data items, transactions make programming easier than linearizability. "Serializability" is the name of one consistency model that provides transactions.

However, transaction systems are significantly more complex, slower, and harder to make fault-tolerant than linearizable systems.

Q: What makes verifying realistic systems involve "huge effort"?

A: Verification means proving that a program is correct, that it is guaranteed to conform to some specification. It turns out that proving significant theorems about complex programs is difficult -- much more difficult than ordinary programming.

You can get a feel for this by trying the labs for this course:

  https://6826.csail.mit.edu/2020/

Q: From the reading assigned, most of the distributed systems are not formally proven correct. So how does a team decide that a framework or system is well tested enough to ship as a real product?

A: It's a good idea to start shipping product, and getting revenue, before the point at which your company would run out of money and go bankrupt. People test as much as they can before that point, and usually try to persuade a few early customers to use the product (and help reveal bugs) with the understanding that it might not work correctly. Maybe you are ready to ship when the product is functional enough to satisfy many customers and has no known major bugs.

Independent of this, a wise customer will also test software that they depend on. No serious organization expects any software to be bug-free.

Q: Why not use the time at which the client sent the command as the linearization point? I.e. have the system execute operations in the order that clients sent them?

A: It's hard to build a system that guarantees that behavior -- the start time is the time at which the client code issued the request, but the service might not receive the request until much later due to network delays. That is, requests may arrive at the service in an order that's quite different from the order of start times. The service could in principle delay execution of every arriving request in case a request with an earlier issue time arrives later, but it's hard to get that right since networks do not guarantee to bound delays. And it would increase delays for every request, perhaps by a lot. That said, Spanner, which we'll look at later, uses a related technique.

A correctness specification like linearizability needs to walk a fine line between being lax enough to implement efficiently, but strict enough to provide useful guarantees to application programs. "Appears to execute operations in invocation order" is too strict to implement efficiently, whereas linearizability's "appears to execute somewhere between invocation and response" is implementable though not as straightforward for application programmers.

Q: Is it a problem that concurrent get()s might see different values if there's also a concurrent put()?

A: It's often not a problem in the context of storage systems. For example, if the value we're talking about is my profile photograph, and two different people ask to see it at the same time that I'm updating the photo, then it's quite reasonable for them to see different photos (either the old or new one).

Some storage systems provide more sophisticated schemes, notably transactions, that make this easier. Transactions automatically lock data to prevent concurrent read/write. "Serializability" is the name of one consistency model that provides transactions. However, transaction systems are more complex, slower, and harder to make fault-tolerant than linearizable systems.