Lecture 4: Linearizability (September 30)

Aurojit Panda

1 Plan

1.1 ADT and reasoning about correctness

1.2 Linearizability

1.3 Seq Consistency

1.4 Locality

1.5 Non-blocking

1.6 Composing linearizable structures

1.7 CAP theorem

1.8 Consequences of CAP

2 Abstract data types and reasoning about correctness under concurrency

Much of this is a repeat of the end of last week, though more formally.

2.1 Abstract data types

Consistency only makes sense w.r.t. operations that apply to something. How to model this in general? We are going to use ADTs: abstract objects that come equipped with a set of operations. For example queues with enqueue and dequeue, or stacks with push and pop.

2.2 Histories

Traces where events are requests and responses from the object. Note this neatly fits into our model.

Why differentiate between requests and responses? Mostly so we can identify and precisely define concurrent processes.

2.2.1 Sequential

Each request is followed by a response: no concurrent operations.

2.2.2 Well-formed: Sequential at each process

Assume that a single process has a single outstanding request at a time. Another way to look at this is that the process blocks after making a request, roughly equivalent to:

```
send(...) # Make request

receive do # Wait for response.
...
end
```

2.2.3 Complete

History H is complete iff all invocations appearing in H have a corresponding response event in H. Given H can drop invocations without responses to get a complete history $complete(H)$

2.2.4 Prefix-closure and sequential specification

What does this look like for a queue?

2.2.5 Operation partial order

Operations in H induce a partial order: for two operations e_0 and e_1 define $e_0 <_H e_1$ if $res(e_0)$ precedes $inv(e_1)$ in H.

3 Linearizability

History H is linearizable if it can be extended to H' so that

- L1: $complete(H')$ is equivalent to some legal sequential history S
- L2 $<_H \subseteq <_S$

Digging into L1 and L2:

- When is $complete(H)$ insufficient? Why do we need to extend to H'?
- What does it mean for $<_H \subseteq <_S$?

3.1 Examples and computing possible schedules

Taking histories from the paper. See lecture video.

4 Sequential Consistency

4.1 Why?

Makes it easier to discuss properties like locality if we can discuss what it means to not meet that property.

4.2 What?

Another consistency level: given history H can extend it to get H' such that $complete(H')$ is equivalent to legal sequential history S such that:

- For all processes p, $H'|p = S|p$.

Put another way: S provides a total order that preserves process order.

Weaker than linearizability; what does that mean? All linearizable histories are sequentially consistent, but not the other way round. How to see this?

5 Locality

5.1 Definition

History H is linearizable iff $H|x$ linearizable for all objects x that appear in x.

Does not hold for sequential consistency.

5.2 Examples

See lecture video/scribble board.

5.3 Proof

Steps: Given:

- history H
- set of objects X.
- $H|x \forall x \in X$ is linearizable, which equivalently gives us $<_x \forall x \in X$ a **total order over all events** in $H|x$. Why is $<_x$ a total order? Well if $H|x$ is linearizable, it is equivalent to some sequential history S which provides a total order over operations in $H|x$, since sequential history means no concurrent operations. Additionally, by definition (L2) we know that $<_{H|x} \subseteq <_S$. We use $<_x$ to refer to $<_S$ in this case. We also get some set of additional events R_x that we have to add to $H|x$ to arrive at S.
- Construct a new extension $H' = H \bigcup_{x \in X} R_x$ by combining all

extentions.

- Compute $<$ over $complete(H')$ such that $< \subseteq <_H$ and $< \subseteq <_x$. How? Take the union of $<_H$ and all $<_x$ and compute the transitive closure. Now need to show that $<$ is a partial order: show that there do not exist $e_1, e_2, \ldots, e_n$ such that $e_1 < e_2 < e_3 < \ldots < e_n$ and $e_n < e_1$. How:

  1. Argue that if e_1 is an event on object x, then at least one of $e_2, \ldots, e_n$ must be an event for another object x'. Why?

     Assume this is not the case. We know that $< \subseteq <_x$, $< \subseteq <_H$, and $<_x$ is a total order. This must mean that $e_{i-1} <_x e_i$ while $e_i <_H e_{i-1}$. Since both events are for the same object x, this also means $e_i <_{H|x} e_{i-1}$. This however contradicts the fact that $<_{H|x} \subseteq <_x$. Hence, any cycle must involve at least two objects.
  2. Now pick the smallest cycle $e_1, e_2, \ldots e_n$ such that e_1 and e_2 are events for different objects.

     First, we will show for this cycle that e_1 and e_n are events on different objects, Do so by showing a stronger thing: if e_1 is an event on object x then none of $e_2, \ldots e_n$ are events on object x. How? Assume that event e_i is the first event after e_1 on object x. By construction $i \neq 2$. We know e_{i-1} and e_i act on different objects (since otherwise both act on x, and

we should be thinking about e_{i-1}). This means they are ordered by $<_H$, which in turn means $response(e_{i-1})$ happens before $inv(e_i)$. Furthermore, since $e_2 < e_{i-1}$ we know that at least $inv(e_2)$ happens before $response(e_{i-1})$ otherwise $e_2, \ldots e_{i-1}$ is a smaller cycle (this is regardless of what objects e_2 and e_i refer to, and is comparing $<_H$ with all of the $\$<_x\$$s). By construction $e_1 <_H e_2$ which means $response(e_1) < inv(e_2)$. This in turn means $response(e_1) < response(e_{i-1}) < inv(e_i)$, which means $e_1 <_H e_i$. But then the cycle $e_1, e_i, \ldots e_n$ is a shorter cycle, which contradicts our assumption that $e_1, ldotse_n$ is the smallest cycle. Thus we now know that e_1 and e_n are on different objects, and are only related by $<_H$.

Second, we show that this cycle is not smallest We have assumed that $e_n <_H e_1$. But $<_H$ is transitive, and $e_1 <_H e_2$ which means $e_n <_H e_2$. This in turn means $e_2, \ldots e_n$ is a shorter cycle. This finally means no such cycle can exist.

6 Non-blocking

6.1 Definition

An operation never has to wait for any other operation to finish (in order to meet the demands of linearizability.)

6.2 Importance

Enables interesting choices in system design. For example, consider a different consistency model, INV, where operations must appear to be executed in **invocation** order. Is INV non-blocking?

Now consider implementing an ADT with operations of different time complexity, and providing INV vs linearizability. What might be some tradeoffs?

7 Composing linearizable data structures

How to build more complex data structures.

7.1 Why?

Just another way to talk about the verification portions of the paper, but this one is more likely to be useful to the average attendee.

7.2 Problem

Despite locality, combining two linearizable data structure to produce a third more complex one is non-trivial. Consider the stack you all got to build while thinking about this lecture.

8 CAP Theorem

8.1 Why?

Two reasons:

- Interesting constraint when building real-world systems, and the response to this constraint has led to many different systems, each of which has presented interesting tradeoffs that developers need to consider. For example, Dynamo, Cassandra, etc.
- Perennial source of internet arguments: knowing this proves you took one of these courses.

8.2 What?

Cannot achieve all three of linearizability, availability and partition tolerance.

8.2.1 Linearizability

Topic of this lecture. Why? Easy to reason about the system.

8.2.2 Availability

Service should eventually generate a response as long as the request can reach a live process.

8.2.3 Parition tolerance

System is correct even when the network is partitioned so that only some of the processes can talk to each other.

8.3 Proof

Proof by indistinguishability, please see the lecture video or scribble board.

It is **important** you understand this proof, we will use the same technique again and again.

9 CAP consequences

Services are often unwilling to give up availability, has led to a whole slew of weaker consistency guarantees, and mechanisms to tune them.

9.1 Eventual consistency

Eventually updates are known to all.

9.2 Causal consistency

Causally linked updates appear together.

9.3 Trading off consistency model

Some systems, e.g., Amazon's Dynamo, provide a way to tradeoff between these extremes. Briefly look at how Amazon is doing this: see lecture video or scribble.