Linearizability & CAP
Announcements

• No hours this week.
  • Sorry am traveling starting tomorrow.

• Lab 1 goes out next week.

• On requiring summaries vs adding labs.
Linearizability
Concurrent Systems or Distributed Systems?

- Linearizability isn’t necessarily about being in a distributed setting.
- Need to worry about operation order even within a single machine.
- Consider multicore, multiple processes, and other sources of concurrency.
- A property where we are not considering anything about failures.
- That comes with the CAP bit later.
Two Core Ideas

• Reasoning about concurrent operations.

• Building concurrent data structures from others.
Reasoning about Concurrent Operations

• What is the problem?

• Tend to specify correctness in terms of sequential behavior

  X Y Z
  enqueue(X)
enqueue(Y)
dequeue()
enqueue(Z)
dequeue()
dequeue( )
Reasoning about Concurrent Operations

Process 1
enqueue(X)
dequeue()
enqueue(Z)
dequeue()

Process 2
enqueue(Y)
dequeue()
Reasoning about Concurrent Operations

$700
- NYU: Deposit $100
- Amazon: Withdraw $30
- Amtrack: Withdraw $80
- Amtrack: Refund $80
- Xi'an: Withdraw $10

$600
- Amazon: Withdraw $30
- Amtrack: Withdraw $80
- Xi'an: Withdraw $10
- Amtrack: Refund $80
- NYU: Deposit $100
Reasoning about Concurrent Operations

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
</table>

Process 1
- enqueue(X)
- dequeue()
- enqueue(Z)
- dequeue()

Process 2
- enqueue(Y)
- dequeue()
Reasoning about Concurrent Operations

Correct?

Any concerns with always using locks?
Reasoning about Concurrent Operations

• Would like to reason about operations without requiring a lock.
  • Locks require all other threads of execution to block, wait their turn.
  • Limited benefit for performance.
• Also brings on questions about granularity of locks.
Concurrent Model

- What sets of ordering are valid?

Possible concerns:

- Does the ordering need to match wall clock time?
- Do we need to preserve ordering for operations in a process?
- Do we need to preserve ordering for operations across objects?
- ...
Linearizability

- Real Time: An operation takes effect between invocation and return.
  - Changes must be visible after return.
- Local: If history for each object is sequential then entire history is sequential.
When are histories linearizable?
Is Linearizable?

A: q.enq(x)
A: q.OK()
B: q.enq(y)
B: q.OK()
A: q.enq(z)
B: q.deq()
B: q.OK(x)
A: q.OK()
A: q.deq()
B: q.deq()
B: q.OK(y)
A: q.OK(z)

Yes

A: q.enq(x)
A: q.OK()
B: q.enq(y)
B: q.OK()
A: q.enq(z)
B: q.deq()
B: q.OK(y)
A: q.OK()
A: q.deq()
B: q.deq()
B: q.OK(x)
A: q.OK(z)

No

A: q.enq(x)
A: q.OK()
B: q.enq(y)
B: q.OK()
A: q.enq(z)
B: q.deq()
B: q.OK(y)
A: q.OK()
A: q.deq()
B: q.deq()
B: q.OK(x)
A: q.OK(z)

Yes
Sequential Consistency

- Operations in a single process happen in the same order.
- Globally operations happen in some sequential order across processes.

Process 1  |  inv(op1)  |  res(op1)  |  inv(op2)  |  res(op2)
----------|-----------|------------|-----------|------------
Process 2  |  inv(op3)  |  res(op3)  |  inv(op4)  |  res(op4)
Sequential Consistency

<table>
<thead>
<tr>
<th>Process 1</th>
<th>\text{inv}(\text{op1})</th>
<th>\text{res}(\text{op1})</th>
<th>\text{inv}(\text{op2})</th>
<th>\text{res}(\text{op2})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 2</td>
<td>\text{inv}(\text{op3})</td>
<td>\text{res}(\text{op3})</td>
<td>\text{inv}(\text{op4})</td>
<td>\text{res}(\text{op4})</td>
</tr>
</tbody>
</table>

- \text{inv}(\text{op1}) \text{ res}(\text{op1}) \text{ inv}(\text{op3}) \text{ res}(\text{op3}) \text{ inv}(\text{op2}) \text{ res}(\text{op2}) \text{ inv}(\text{op4}) \text{ res}(\text{op4})
- \text{inv}(\text{op1}) \text{ res}(\text{op1}) \text{ inv}(\text{op2}) \text{ res}(\text{op2}) \text{ inv}(\text{op3}) \text{ res}(\text{op3}) \text{ inv}(\text{op4}) \text{ res}(\text{op4})
- \text{inv}(\text{op1}) \text{ res}(\text{op1}) \text{ inv}(\text{op4}) \text{ res}(\text{op4}) \text{ inv}(\text{op2}) \text{ res}(\text{op2}) \text{ inv}(\text{op3}) \text{ res}(\text{op3})

- \text{inv}(\text{op1}) \text{ res}(\text{op1}) \text{ inv}(\text{op3}) \text{ res}(\text{op3}) \text{ inv}(\text{op2}) \text{ res}(\text{op2}) \text{ inv}(\text{op4}) \text{ res}(\text{op4})
- \text{inv}(\text{op1}) \text{ res}(\text{op1}) \text{ inv}(\text{op2}) \text{ res}(\text{op2}) \text{ inv}(\text{op3}) \text{ res}(\text{op3}) \text{ inv}(\text{op4}) \text{ res}(\text{op4})
- \text{inv}(\text{op1}) \text{ res}(\text{op1}) \text{ inv}(\text{op4}) \text{ res}(\text{op4}) \text{ inv}(\text{op2}) \text{ res}(\text{op2}) \text{ inv}(\text{op3}) \text{ res}(\text{op3})
Sequential Consistency

- Not real time. Why?
- Not local. Why?
Sequential Consistency

Process A
A: p.enq(x)
A: p.OK()
B: q.enq(y)
B: q.OK()
A: q.enq(x)
A: q.OK()

Process B
A: q.enq(x)
A: q.OK()
B: p.enq(y)
B: p.OK()
A: p.enq(y)
A: p.OK(y)
B: p.deq()
B: q.deq()
B: q.OK(x)

Red X indicates an illegal order: p.ok(Y) before p.deq()
Sequential Consistency

Process A
- p.enq(x)
- p.OK()
- p.deq()
- p.ok(Y)

Process B
- q.enq(y)
- q.OK()
- q.deq()
- q.ok(X)

The sequence highlighted with an X indicates a violation of sequential consistency.
Serializability and Strict Serializability

• Common in databases, will deal with in a few classes.

• Basic extension: consider multiple operations at a time rather than one operation.

• Serializability: Multiple operations occur in some order.
  • Make it appear like a group of operations committed at the same time.

• Strict Serializability: Serializability + require everything is real time.
  • Hard to implement in practice (without giving up on performance).
Two Core Ideas

- Reasoning about concurrent operations.
- Building concurrent data structures from others.
How to enforce a consistency model?
How to Enforce a Consistency Model?

• In almost all cases control two things:
  • When does some change (due to an operation) become visible?
  • When is a process allowed to take a step?
Building a Linearizable Queue

- Need to ensure linearizability.

- Need to ensure concurrent processes do not see corrupted data.

type CQueue struct {
    l *sync.Mutex
    q Queue
}

func (q *CQueue) Enque(val) ... {
    q.l.Lock()
    defer q.l.Unlock()
    return q.q.Enque(val)
}

func (q *CQueue) Deque(val) ... {
    q.l.Lock()
    defer q.l.Unlock()
    return q.q.Dequeue()
}
Building a Linearizable Queue

type CQueue struct {
    back: int32
    items: []*Item
}

func (q *CQueue) Enq(v: Item) {
    i := atomic.AddInt32(&q.back, 1)
    i = i - 1
    atomic.StorePointer(&v, &q.items[i])
}

func (q *CQueue) Deq() {
    for {
        range := atomic.LoadInt32(&q.back)
        for i = 0; i < range; i++ {
            x := atomic.SwapPointer(&q.items[i], nil)
            if x != nil { return *x }
        }
    }
}
Building a Linearizable Queue

- Are both queues correct?
- Why prefer one or the other queue?
CAP Theorem
A Source of Internet Arguments

- Eric Brewer gave a keynote at PODC 2000
- "Towards Robust Distributed Systems"
- Based on experiences building systems at Berkeley and Inktomi.

Statement: For any distributed shared-data system pick two of:

- Consistency
- Availability
- Partition Tolerance
What you read

• An attempt to formalize this concept.

• What is consistency?
  • Unspecified in original talk. Gilbert and Lynch go with Linearizability.

• What is availability?
  • System should respond to every request.

• What is partition tolerance?
  • System should continue to operate despite network partitions.
Indistinguishability

• A common proof technique in distributed systems.
Indistinguishability

- A common proof technique in distributed systems.

![Diagram showing indistinguishability](image)
Fair Schedules

• What is a fair schedule?

• Concern about what packets are dropped or lost.
  • Could choose to only drop packets of a certain type or from a certain node.

• Fairness means that any message should have a chance to go through.

• Precise statement:
  • If a node sends a message infinitely often, it must be received infinitely often.
Why Does Fairness Matter Here?
Partial Synchrony

• Meant to provide a more accurate model of the network in reality.

• Networks are not always evil, not always dropping or loosing packets.

• Originally proposed by Dwork, Lynch and Stockmeyer
Partial Synchrony

- There are bounds on message delay and processing time.
  - Bounds are not known a-priori.
- After some finite period of time (globally) these bounds hold.
  - When is not known a-priori.
- Seemingly adds very little information to the system but enables algorithms.
Why does partial synchrony help here?
Weaker Consistency Models

• In the last decade trends towards weaker consistency models.
  
  • Prefer availability over consistency.
  
  • Also helps performance: possibly respond without blocking.
  
  • Adopted by datastores like MongoDB, CouchDB, etc.
  
  • One of the hallmarks of the NoSQL movement.
  
• Look at a couple of these weaker consistency models here.
Eventual Consistency

- Operations eventually become visible.
- No ordering guarantees beyond that.

A
B: Lunch?
A: Taco Bell?
C: Agreed
B: Taco Bell sux

B
B: Lunch?
A: Taco Bell
B: Taco Bell sux

C
A: Taco Bell
B: Lunch?
B: Taco Bell sux
C: Agreed
Causal Consistency

• Operations eventually become visible.

• Order preserves causality
Relaxing Consistency

• Pros:
  • Availability, performance.

• Cons:
  • Hard to program? Hard to reason about correctness?

• Research Questions:
  • When is a given consistency model appropriate?
  • How to improve developer productivity given weaker consistency models?
Conclusion

- Consistency models are a way to reason about when events take effect.
- Both necessary when building systems and when reasoning about systems.