Linearizability & CAP
Announcements

• No hours this week.
Announcements

• No hours this week.

• Sorry am traveling starting tomorrow.
Announcements

• No hours this week.
  • Sorry am traveling starting tomorrow.
• Lab 1 goes out next week.
Announcements

• No hours this week.
  • Sorry am traveling starting tomorrow.
• Lab 1 goes out next week.
• On requiring summaries vs adding labs.
Linearizability
Concurrency not Distributed Systems?

- Linearizability isn't necessarily about being in a distributed setting.
Concurrency not Distributed Systems?

- Linearizability isn't necessarily about being in a distributed setting.
- Need to worry about operation order even within a single machine.
Concurrency not 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.
Concurrency not 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.
Concurrency not 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?
Reasoning about Concurrent Operations

- What is the problem?
- Tend to specify correctness in terms of sequential behavior
Reasoning about Concurrent Operations

• What is the problem?

• Tend to specify correctness in terms of sequential behavior

\[
\begin{align*}
X \quad & \quad X \\
& \quad \text{enqueue}(X)
\end{align*}
\]
Reasoning about Concurrent Operations

• What is the problem?

• Tend to specify correctness in terms of sequential behavior

\[
\begin{array}{c c}
X & Y \\
\end{array}
\]

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

• What is the problem?

• Tend to specify correctness in terms of sequential behavior

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

• What is the problem?

• Tend to specify correctness in terms of sequential behavior

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

• What is the problem?

• Tend to specify correctness in terms of sequential behavior

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

• What is the problem?

• Tend to specify correctness in terms of sequential behavior

```
enqueue(X)
ingenue(Y)
dequeue()
genue(Z)
dequeue()
dequeue()
```
Reasoning about Concurrent Operations

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>enqueue(X)</td>
<td>enqueue(Y)</td>
</tr>
<tr>
<td>dequeue()</td>
<td>dequeue()</td>
</tr>
<tr>
<td>enqueue(Z)</td>
<td></td>
</tr>
<tr>
<td>dequeue()</td>
<td></td>
</tr>
<tr>
<td>enqueue()</td>
<td></td>
</tr>
<tr>
<td>dequeue()</td>
<td></td>
</tr>
</tbody>
</table>
Reasoning about Concurrent Operations
Reasoning about Concurrent Operations

$100
NYU: Deposit $100
Reasoning about Concurrent Operations

$70
NYU: Deposit $100
Amazon: Withdraw $30
Reasoning about Concurrent Operations

$10
NYU: Deposit $100
Amazon: Withdraw $30
Amtrak: Withdraw $80
Reasoning about Concurrent Operations

$70
NYU: Deposit $100
Amazon: Withdraw $30
Amtrack: Withdraw $80
Amtrack: Refund $80
Reasoning about Concurrent Operations

$60
NYU: Deposit $100
Amazon: Withdraw $30
Amtrack: Withdraw $80
Amtrack: Refund $80
Xi'an: Withdraw $10
Reasoning about Concurrent Operations

$60
NYU: Deposit $100
Amazon: Withdraw $30
Amtrack: Withdraw $80
Amtrack: Refund $80
Xi'an: Withdraw $10

$0
Reasoning about Concurrent Operations

$60
NYU: Deposit $100
Amazon: Withdraw $30
Amtrack: Withdraw $80
Amtrack: Refund $80
Xi'an: Withdraw $10

$30
Amazon: Withdraw $30
Reasoning about Concurrent Operations

$60
NYU: Deposit $100
Amazon: Withdraw $30
Amtrack: Withdraw $80
Amtrack: Refund $80
Xi'an: Withdraw $10

$110
Amazon: Withdraw $30
Amtrack: Withdraw $80
Reasoning about Concurrent Operations

$60
NYU: Deposit $100
Amazon: Withdraw $30
Amtrack: Withdraw $80
Amtrack: Refund $80
Xi'an: Withdraw $10

$120
Amazon: Withdraw $30
Amtrack: Withdraw $80
Xi'an: Withdraw $10
Reasoning about Concurrent Operations

$60
NYU: Deposit $100
Amazon: Withdraw $30
Amtrack: Withdraw $80
Amtrack: Refund $80
Xi'an: Withdraw $10

$40
Amazon: Withdraw $30
Amtrack: Withdraw $80
Xi'an: Withdraw $10
Amtrack: Refund $80
Reasoning about Concurrent Operations

$60
NYU: Deposit $100
Amazon: Withdraw $30
Amtrack: Withdraw $80
Amtrack: Refund $80
Xi'an: Withdraw $10

$60
Amazon: Withdraw $30
Amtrack: Withdraw $80
Xi'an: Withdraw $10
Amtrack: Refund $80
NYU: Deposit $100
Reasoning about Concurrent Operations
Reasoning about Concurrent Operations

Process 1

Process 2
Reasoning about Concurrent Operations

Process 1

Process 2

enqueue(X)
Reasoning about Concurrent Operations

X

enqueue(X)

Process 1       Process 2
Reasoning about Concurrent Operations

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

enqueue(X) | enqueue(Y)

Process 1 | Process 2
Reasoning about Concurrent Operations

enqueue(X)  enqueue(Y)
  dequeue()

Y

Process 1  Process 2
Reasoning about Concurrent Operations

Process 1
enqueue(X)  
dequeue()

Process 2
enqueue(Y)
Reasoning about Concurrent Operations

Process 1
enqueue(X)
dequeue()

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

Z

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

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

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

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

Correct?
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.
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.
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.
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.
Concurrency Model

• What sets of ordering are valid?
Concurrency Model

• What sets of ordering are valid?

• Possible concerns:
Concurrency Model

• What sets of ordering are valid?

• Possible concerns:
  • Does the ordering need to match wall clock time?
Concurrency 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?
Concurrency 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?
Concurrency 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?

```plaintext
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()  
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()
A: q.deq()
B: q.deq()
B: q.OK(x)
A: q.OK(z)```

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

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

No
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)
A: q.OK()
A: q.deq()
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(z)
A: q.OK(y)
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(x)
A: q.OK()
A: q.deq()
B: q.deq()
B: q.deq()
B: q.OK(y)
A: q.OK(z)
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(op1)}</th>
<th>\text{res(op1)}</th>
<th>\text{inv(op2)}</th>
<th>\text{res(op2)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 2</td>
<td>\text{inv(op3)}</td>
<td>\text{res(op3)}</td>
<td>\text{inv(op4)}</td>
<td>\text{res(op4)}</td>
</tr>
</tbody>
</table>
Sequential Consistency

Process 1  \[\text{inv(op1)} \quad \text{res(op1)} \quad \text{inv(op2)} \quad \text{res(op2)}\]
Process 2  \[\text{inv(op3)} \quad \text{res(op3)} \quad \text{inv(op4)} \quad \text{res(op4)}\]

\[\text{inv(op1)} \quad \text{res(op1)} \quad \text{inv(op3)} \quad \text{res(op3)} \quad \text{inv(op2)} \quad \text{res(op2)} \quad \text{inv(op4)} \quad \text{res(op4)}\]
## Sequential Consistency

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

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

Process 1  \( \text{inv(op1)} \ \text{res(op1)} \ \text{inv(op2)} \ \text{res(op2)} \)
Process 2  \( \text{inv(op3)} \ \text{res(op3)} \ \text{inv(op4)} \ \text{res(op4)} \)

\[
\begin{align*}
\text{inv(op1)} & \ \text{res(op1)} & \ \text{inv(op3)} & \ \text{res(op3)} & \ \text{inv(op2)} & \ \text{res(op2)} & \ \text{inv(op4)} & \ \text{res(op4)} \\
\text{inv(op1)} & \ \text{res(op1)} & \ \text{inv(op5)} & \ \text{res(op5)} & \ \text{inv(op2)} & \ \text{res(op2)} & \ \text{inv(op4)} & \ \text{res(op4)} \\
\text{inv(op1)} & \ \text{res(op1)} & \ \text{inv(op4)} & \ \text{res(op4)} & \ \text{inv(op2)} & \ \text{res(op2)} & \ \text{inv(op3)} & \ \text{res(op3)}
\end{align*}
\]
Sequential Consistency
Sequential Consistency

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

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

p.ok(y)
q.deq()
q.ok(x)

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

Process B
p.enq(x)
p.ok(y)
q.deq()
q.ok(x)

p
q
Sequential Consistency

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

Process A

p.enq(x)
p.ok(Y)

q.deq()
q.ok(X)

Process B

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

Process B
X

p
q
Sequential Consistency

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

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

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

Process B
- B: q.enq(y)
- B: q.OK()
- B: p.enq(y)
- B: p.OK()
- B: q.deq()
- B: q.OK(x)
Sequential Consistency

A: p.enq(x)
P: p.OK(
q.enq(Y)
q.OK( )

B: q.enq(y)
P: q.OK()
q.deq()
q.ok(X)

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

Process A Process B
Sequential Consistency

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: p.enq(x)</td>
<td>p.enq(x)</td>
</tr>
<tr>
<td>A: p.OK()</td>
<td>p.OK()</td>
</tr>
<tr>
<td>B: q.enq(y)</td>
<td>q.enq(y)</td>
</tr>
<tr>
<td>B: q.OK()</td>
<td>q.OK()</td>
</tr>
<tr>
<td>A: q.enq(x)</td>
<td>p.enq(x)</td>
</tr>
<tr>
<td>A: q.OK()</td>
<td>p.OK()</td>
</tr>
<tr>
<td>B: p.enq(y)</td>
<td>q.enq(y)</td>
</tr>
<tr>
<td>B: p.OK()</td>
<td>q.OK()</td>
</tr>
<tr>
<td>A: q.deq()</td>
<td>q.deq()</td>
</tr>
<tr>
<td>A: q.OK(y)</td>
<td>q.ok(X)</td>
</tr>
<tr>
<td>B: q.deq()</td>
<td></td>
</tr>
<tr>
<td>B: q.OK(x)</td>
<td></td>
</tr>
</tbody>
</table>
Sequential Consistency

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

Process A

Process B
Sequential Consistency

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

Process A

Process B
Sequential Consistency

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

p.enq(x)          q.enq(y)          p.enq(X)
q.enq(X)          p.ok(Y)          q.enq(Y)
q.OK()            q.OK()            q.deq()            q.ok(X)

Process A          Process B
Sequential Consistency

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

Process B
- p.enq(x)
- q.enq(Y)
- p.OK() q.OK()
- q.enq(X) p.enq(Y)
- q.OK() q.OK()
- q.deq() q.ok(X)
## Sequential Consistency

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: p.enq(x)</td>
<td>p.enq(x)</td>
</tr>
<tr>
<td>A: p.OK()</td>
<td>p.0K()</td>
</tr>
<tr>
<td>B: q.enq(y)</td>
<td>q.enq(X)</td>
</tr>
<tr>
<td>B: q.OK()</td>
<td>q.0K()</td>
</tr>
<tr>
<td>A: q.enq(x)</td>
<td>p.enq(Y)</td>
</tr>
<tr>
<td>A: q.OK()</td>
<td>p.0K()</td>
</tr>
<tr>
<td>B: p.enq(y)</td>
<td>q.deq()</td>
</tr>
<tr>
<td>B: p.OK()</td>
<td>q.0K()</td>
</tr>
<tr>
<td>A: p.deq()</td>
<td>q.ok(X)</td>
</tr>
<tr>
<td>A: p.OK(y)</td>
<td></td>
</tr>
<tr>
<td>B: q.deq()</td>
<td></td>
</tr>
<tr>
<td>B: q.OK(x)</td>
<td></td>
</tr>
</tbody>
</table>

- A: p.enq(x)  
- A: p.OK()   
- B: q.enq(y)  
- B: q.OK()   
- A: q.enq(x)  
- A: q.OK()   
- B: p.enq(y)  
- B: p.OK()   
- A: p.deq()  
- A: p.OK(y)  
- B: q.deq()  
- B: q.OK(x)
Sequential Consistency

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

p.enq(x)  q.enq(Y)
p.OK()  q.OK()
q.enq(X)  p.enq(Y)
q.OK()  p.OK()
p.deq()  q.deq()
p.OK(y)  q.OK(x)

Process A  Process B
Sequential Consistency

A: p.enq(x)    p.enq(x)    q.enq(Y)
A: p.OK()      p.OK()      q.OK()
B: q.enq(y)    q.enq(x)    p.enq(Y)
B: q.OK()      q.OK()      p.OK()
A: q.enq(x)    p.deq()    q.deq()
A: q.OK()      p.ok(Y)    q.ok(X)
B: p.enq(y)    B: q.deq()
B: p.OK()      B: q.OK(x)
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()
B: p.enq(y)
B: p.OK()
A: p.deq()
A: p.OK(y)
B: q.deq()
B: q.OK(x)

Process B

p
q
Sequential Consistency

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

Process A | Process B
Sequential Consistency

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

Process A          Process B
Sequential Consistency

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

p
q
Y

Process A

Process B
Sequential Consistency

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

Process A

Process B
Sequential Consistency

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

Process A | Process B
---|---

p | q
Y | Y
Sequential Consistency

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

Y
Y

p
q
Sequential Consistency

A: \text{p.enq}(x)
A: \text{p.OK}()
B: \text{q.enq}(y)
B: \text{q.OK}()
A: \text{q.enq}(x)
A: \text{q.OK}()
B: \text{p.enq}(y)
B: \text{p.OK}()
A: \text{p.deq}()
A: \text{p.OK}(y)
B: \text{q.deq}()
B: \text{q.OK}(x)

\text{Process A} \quad \text{Process B}
Sequential Consistency

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

p.enq(x)  q.enq(y)
q.OK()    p.OK()
Sequential Consistency

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

Process A | Process B
---|---
p | q
X | Y
p.enq(x) | q.enq(Y)
p.OK() | q.OK()
p.enq(Y) | p.OK()
Sequential Consistency

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

Process A                Process B

\[\text{X} \quad \text{Y}\]

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

Process B

p.enq(x)  q.enq(y)
p.OK()    q.OK()
p.enq(x)  q.enq(y)
q.OK()    p.OK()
p.enq()   q.enq()
p.OK(y)   p.OK()
q.enq()   p.enq()
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

B: p.enq(y)
B: p.OK()
A: q.deq()
A: p.ok(Y)
B: q.deq()
B: q.ok(X)

A: p.enq(x)
B: q.enq(y)
A: q.enq(x)
B: p.enq(y)
A: p.enq(y)
B: q.enq(X)
A: q.enq(X)
B: q.enq(Y)
A: p.enq(Y)
B: q.enq(Y)
A: q.enq(Y)
B: q.enq(Y)
A: p.enq(Y)
B: q.enq(Y)
A: p.enq(Y)
B: q.enq(Y)
Sequential Consistency

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

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

Red cross indicates invalid execution order.
Serializability and Strict Serializability

- Common in databases, will deal with in a few classes.
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 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.
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.
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.
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:
How to Enforce a Consistency Model?

• In almost all cases control two things:

• When does some change (due to an operation) become visible?
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.
Building a Linearizable Queue

- Need to ensure linearizability.
- Need to ensure concurrent processes do not see corrupted data.
Building a Linearizable Queue

• Need to ensure linearizability.

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

```go
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
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
A Source of Internet Arguments

- Eric Brewer gave a keynote at PODC 2000

- "Towards Robust Distributed Systems"
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.
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:
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
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
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 you read

• An attempt to formalize this concept.

• What is consistency?
What you read

• An attempt to formalize this concept.

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

- A common proof technique in distributed systems.
Indistinguishability

• A common proof technique in distributed systems.
Indistinguishability

- A common proof technique in distributed systems.
Indistinguishability

- A common proof technique in distributed systems.

\[\text{get}(x)\]

Alice \quad Bob
Indistinguishability

• A common proof technique in distributed systems.
Indistinguishability

- A common proof technique in distributed systems.
Indistinguishability

- A common proof technique in distributed systems.

```
Alice
write(x = 2)
```

```
Bob
get(x)
```

```
Alice
get(x)
```

```
Bob
get(x)
```
Fair Schedules

- What is a fair schedule?
Fair Schedules

• What is a fair schedule?

• Concern about what packets are dropped or lost.
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.
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.
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:
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.
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.
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.
Partial Synchrony

• There are bounds on message delay and processing time.

• Bounds are not known a-priori.
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.
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.
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.
Weaker Consistency Models

• In the last decade trends towards weaker consistency models.

• Prefer availability over consistency.
Weaker Consistency Models

- In the last decade trends towards weaker consistency models.
- Prefer availability over consistency.
- Also helps performance: possibly respond without blocking.
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.
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.
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: Lunch?
B: Taco Bell?
Eventual Consistency

• Operations eventually become visible.
• No ordering guarantees beyond that.
Eventual Consistency

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

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

A
B: Lunch?
A: Taco Bell?

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

C
A: Taco Bell
B: Lunch?
Eventual Consistency

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

A

B: Lunch?
A: Taco Bell?

B

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

C

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

• Operations eventually become visible.

• No ordering guarantees beyond that.

A

B: Lunch?
A: Taco Bell?

B

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

C

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

• Operations eventually become visible.

• No ordering guarantees beyond that.

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

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

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

A

B: Lunch?
A: Taco Bell?

B

C
Causal Consistency

- Operations eventually become visible.
- Order preserves causality

A
B: Lunch?  
A: Taco Bell?

B
B: Lunch?  
A: Taco Bell
Causal Consistency

• Operations eventually become visible.

• Order preserves causality
Causal Consistency

- Operations eventually become visible.

- Order preserves causality
Causal Consistency

- Operations eventually become visible.
- Order preserves causality
Causal Consistency

- Operations eventually become visible.
- Order preserves causality
Causal Consistency

- Operations eventually become visible.
- Order preserves causality
Causal Consistency

- Operations eventually become visible.

- Order preserves *causality*
Relaxing Consistency

• Pros:
Relaxing Consistency

• Pros:
  • Availability, performance.
Relaxing Consistency

• Pros:
  • Availability, performance.

• Cons:
Relaxing Consistency

• Pros:
  • Availability, performance.

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

• Pros:
  • Availability, performance.

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

• Research Questions:
Relaxing Consistency

• Pros:
  • Availability, performance.

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

• Research Questions:
  • When is a given consistency model appropriate?
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.