“There are only two hard things in computer science: cache invalidation and naming things.”
- Phil Karlton
Caching is the other half of distributed systems

- Option #1: Go to the data (RPC)
  - Problems: server has to process all operations, latency from going to server on every operation, can’t run operations if the server isn’t available

- Option #2: Bring the data to you (caching)
  - Problems: how to keep the data synchronized with the server, what if there is more than one cache
Example: Dropbox, ideally

Client 1

write(f,"A")

f = [A]

write(f, "B")

f = [A,B]

Client 2
Example: Dropbox, real-life

Client 1

write(f,"A")

f = [A]

Client 2

f = []

write(f,"B")

f = [A,B]
Some definitions

• Coherence/consistency models: Does the reality match the ideal?
  • Coherence = guarantee for single object
  • Consistency = guarantee for multiple objects across the system
  • Caveat: literature is not consistent about this terminology! Naming is hard!
• Anomaly: a sequence of operations leading to a state that cannot occur in the ideal system
• Some models:
  • weak consistency: doesn't match the ideal system, could have anomalies
  • eventual consistency: temporary anomalies, but over time (if no further modifications are made), the system will converge
  • sequential consistency/serializable: behaves like the idea system for applications but might not matching external user expectations
  • linearizable: behaves like a single system to users
Consistency Models
Defining Consistency Models

Client 1

write(f, A)

ok

read(f)

?

write(f, B)

Client 2

ok

read(f)

?
Linearizability

Client 1

write(f, A)
ok
read(f)
[A, B]

Client 2

write(f, B)
ok
read(f)
[A, B]
Serializability

Client 1
- write(f, A)
- ok
- read(f)
- [B, A]

Client 2
- write(f, B)
- ok
- read(f)
- [B, A]
Weak Consistency

Client 1
- write(f,A)
- ok
- read(f)
- [B,A]

Client 2
- write(f,B)
- ok
- read(f)
- [A,B]

Note: The diagram illustrates the concept of weak consistency in distributed systems, where different clients can observe the state of a shared resource in an inconsistent manner at different times.
Eventual Consistency

Client 1

write(f, A)
ok
read(f)
[A]

Client 2

write(f, B)
ok
read(f)
[B]
Maintaining Consistency
Example: Single client

Client 1

```
write(f, "A")
```

```
f = []
```

```
ok
```

```
f = [A]
```

```
read(f)
```

```
[]
```

Client 2

```
f = []
```

```
[]
```
Example: Multiple clients

Client 1

| write(f,"A") |
| ok |
| f = [A] |

Client 2

| write(f,"B") |
| ok |
| f = [B] |

Cache

```python
f = []
f = [A]
f = [B]
```
Idea: Cache Invalidations

Client 1

```
write(f,"A")
f = [A]
ok
```

Client 2

```
write(f,"B")
f = [A]
f = [A,B]
ok
```
Cache invalidations

• Need to lock the store and not allow any updates before invalidation to ensure strong consistency

• Slow, especially in a distributed system

• Sometimes practical on processors where caches share a bus, but even modern processors do not provide strong consistency
Idea: Leases

Client 1

write(f, "A")

f = []

ok

get_lease(f)

f = [A]

Client 2

write(f, "B")

f = [A]

f = [A, B]

ok
Cache invalidations

- Need to revoke all read leases before acquiring a write lease
- Requires all to all communication
- Again can be practical on processors where caches can snoop on a bus, but even modern processors do not provide strong consistency
Strong vs. Weak consistency

• Strong is obviously easier for programmers to use but performance can be bad

• What about when you can’t reach some clients for invalidation or to revoke leases? Do you wait for them?

• CAP theorem deals with exactly the case where some clients are unreachable: The system can either give up (A)vailability by blocking or (C)onsistency by going ahead without reaching some of the clients
Maintaining Consistency in systems with sharding
Example: Sharding

write(f, "A")
write(g, "X")
ok
ok
f = []
g = []
if g = [X] append(f, g)
g = [X]
f = [A]
g = [X]
f = []
f = [X]
Idea: Blocking writes

Client 1

write\( (f, "A") \)

ok

write\( (g, "X") \)

ok

Client 2

\( f = [\] \)

\( g = [\] \)

if \( g = [X] \)

append\( (f, g) \)

\( g = [X] \)

\( f = [A] \)

\( f = [A, X] \)
Processor Ordering

• Linearizability and sequential consistency guarantee processor ordering (operations at one client execute in the order they are issued).

• Re-ordering can happen for many reasons: out-of-order execution, network re-ordering

• Again, it is easier to reason about but more expensive

• Another option for this example in a distributed system is transactions to guarantee atomicity: if the write(g) is visible then so is the write(f), otherwise neither