Leases and Cache Coherence
Leases

Lease - a time-limited right to do something
  - can be renewed
    - unlike Paxos, depends on loosely synchronized clocks

Lease fault tolerance
  - if lease holder or network fails, wait for lease to expire
    - plus epsilon to account for clock drift
  - hand lease to someone new
Paxos as Lease Server

Paxos group as fault tolerant view server

- grant lease to primary
- primary serves requests
- revoke lease if not renewed
- grant lease to new primary

Design pattern used in GFS, BigTable, …
Primary election in Chubby, Zookeeper

```java
x = Open("/BigTable/primary")
if (TryAcquire(x) == success) {
    // I'm the primary, tell everyone
    SetContents(x, my-address)
} else {
    // I'm not the primary, find out who is
    primary = GetContents(x)
    // also set up notifications
    // in case the primary changes
}
```
Example

Client

App

App

Chubby

Paxos
Example

Client

App

App

TryAcquire

Chubby

Paxos
Example

Client

App

App

Chubby

Paxos

OK
Example
Example
Example

Client

Primary

Backup

Chubby

Paxos
Example

Client

Primary

Backup

Chubby

Paxos
Example

Client → Requests → Primary

Backup

Chubby

Paxos
What if Primary Fails?

Client

Backup

Primary

Chubby

Paxos
What if Primary Fails?

Client

Backup

Primary

Chubby

Paxos
What if Primary Fails?

Client

Primary

Backup

TryAcquire

Chubby

Paxos
What if Primary Fails?

Client → Backup → Chubby → Paxos

Primary

X

OK
What if Primary Fails?

Client

Primary

X

Chubby

Paxos
Primary Backup With Leases

What if the old primary didn’t crash?
Client sends request to old primary
What keeps old primary from performing op?
Primary Backup With Leases

What if the old primary didn’t crash?
Client sends request to old primary
What keeps old primary from performing op?
Old primary demotes itself if it doesn’t renew lease
Primary Backup with Leases

No possibility of split brain

Reads can occur at the primary!
  - no need to talk to backup

Writes can be logged to storage layer
  - on failure, new primary reads latest changes from storage layer
  - backup is optimization to speed recovery
Fault Tolerant Caching with Leases

Linearizability with caches is another use of leases

Cache obtains lease (ex: read-only)

No one can modify data until lease expires or is revoked

Once lease expires, ok for server to change
Caching With Leases

Client

Cache 1

Cache 2

Chubby

Paxos
Caching With Leases
Caching With Leases
Caching With Leases

Client → Get x → Cache 1

Client → Cache 2

Server
Caching With Leases

Client

Cache 1

Get x

Server

Cache 2

Client

Client

Client

Client
Caching With Leases

Cache 1

x = 3, for t

Cache 2

Cache 1 has x, for t
Caching With Leases

Client

Cache 1

x = 3, for t

Cache 2

Server

Cache 1 has x, for t

Client
Caching With Leases

Client
Cache 1
Cache 2
Server

Cache 1 has x, for t

x = 3, for t

Get x
Caching With Leases

Server

Cache 1

Cache 2

Client

Client

Client

Client

Cache 1 has x, for t

Get x

Cache 1 has x, for t
Caching With Leases

Client

Cache 1

x = 3, for t

x = 3, for t

Cache 2

Cache 1, 2 has x, for t

Server
Caching With Leases

Client

Cache 1

$\text{x} = 3$, for $t$

Cache 2

$\text{x} = 3$, for $t$

Server

Cache 1,2 has $\text{x}$, for $t$
Caching With Leases

Why give out cache leases with same values of \( t \)?

Why give out cache leases with different values of \( t \)?
Caching With Leases

Why give out cache leases with same values of t?
- less state at server
- can reclaim leases at same time

Why give out cache leases with different values of t?
- caches all ask for new lease at same time
Caching With Leases

Client → Cache 1
  x = 3, for t

Client → Cache 2
  x = 3, for t

Client → Server
  Get x

Cache 1, 2 has x, for t
Caching With Leases

Client → Cache 1
  \( x = 3 \), for \( t \)

Client → Cache 2
  \( x = 3 \), for \( t \)

Cache 1,2 has \( x \), for \( t \)

Server
Caching With Leases

Can clients cache values too?
Caching With Leases

Can clients cache values too?

Yes! Leases can be delegated.

Caches keep track as to which clients have which data.
Caching With Leases

Client

Put x=4

Cache 1
x = 3, for t

Cache 2
x = 3, for t

Server

Cache 1,2 has x, for t
Caching With Leases

Server

Cache 1

Cache 2

Put $x=4$

Client

Cache 1,2 has $x$, for $t$

Client

$\times=3$, for $t$

Client

$\times=3$, for $t$

Client
Caching With Leases

Client

Cache 1
  x = 3, for t

Cache 2
  x = 3, for t

Server

Cache 1,2 has x, for t
Caching With Leases

Client

Cache 1

x = 3, for t

Cache 2

x = 3, for t

Server

Cache 1,2 has x, for t

Client

Get x

Client
Caching With Leases

Client

Cache 1
x = 3, for t

Cache 2
x = 3, for t

Client

Cache 1,2 has x, for t

Server
Caching With Leases

No one has copy of x
Ok to change x
Caching With Leases

Client

Put $x=4$

Client

Cache 1

$x=3$, for $t$

Cache 2

$x=3$, for $t$

Server

Cache 1, 2 has $x$, for $t$

OR
Caching With Leases

Client

Cache 1

Put $x = 4$

Server

Cache 2

$x = 3, \text{ for } t$

Cache 1,2 has $x, \text{ for } t$

Client

Client

Client

Client
Caching With Leases

Cache 1

x = 3, for t

Cache 2

Cache 2 has x, for t
Caching With Leases

Cache 1

Cache 2

Server

Cache 2 has \( x \), for \( t \)
Caching With Leases

Client

Cache 1

Cache 2

Server

OK!

Cache 2 has x, for t
Caching With Leases

Client

Cache 1

Cache 2

Server

No one has copy of x
Ok to change x
Caching With Leases

Why can’t we leave the old value on cache 1 while we shoot down other copies?

Why can’t we just update the old value on cache 1 and then shoot down the other copies?
Caching With Leases

Why can’t we leave the old value on cache 1 while we shoot down other copies?

Why can’t we just update the old value on cache 1 and then shoot down the other copies?

Linearizability: as if there is only one copy
- implement by having only one copy for updates
- many copies ok when no one is updating
Caching with Invalidation

Cache obtains lease (read-only)
No one can modify data until lease expires or is revoked
Server gets update
Forwards invalidation (revoke) to every node with copy
Wait for response from all (or timeout)
OK to proceed with change
Terminology

Cache coherence: keeping caches up to date
- can be linearizable, or weaker semantics
Write through: caches hold read-only data
- write sent to store, store revokes copies
Write back: caches can hold read-only or modified data
- write to cache, cache asks store to revoke
- subsequent writes faster
MSI

Three cache states:

- **Modified**: this is the only copy, it’s dirty
- **Shared**: this is one of many copies, it’s clean
- **Invalid**

Allowed states between pairs of caches:
Write Back Fault Tolerance?

Write back: caches can hold modified data

What happens when cache fails? Lose data?

Option 1: checkpoint/restart if any cache fails
  - appropriate for background computations
  - CPU cache coherence is write-back

Option 2: log local changes to replicas
  - identical to lease to a primary (primary logs changes), except fine-grained leases
Caching With Leases

Client → Get x → Cache 1

Client → Cache 2

Client → Cache 1

Client → Cache 2

Server
Caching With Leases

Client

Cache 1

Get x

Cache 2

Server

Client

Client

Client
Caching With Leases

Cache 1

Cache 2

Cache 1 has x, t, shared

x=3, t, shared

Server

Client

Client

Client

Client
Caching With Leases

Cache 1 has x, t, shared

x = 3, t, shared
Caching With Leases

Client

Cache 1

Server

Cache 1 has x, t, shared

Client

Cache 2

Put x, 4

x = 3, t, shared

Client

Client

Client
Caching With Leases

Client

Put x, 4

x = 3, t, shared

Cache 2

Cache 1

Cache 1 has x, t, shared

Server
Caching With Leases

Client

Cache 1

Need x, dirty

Server

Cache 2

Cache 1 has x, t, shared

x = 3, t, shared

Client

Client

Client

Client
Caching With Leases

Cache 1
x = 3, t, shared

Cache 2

Cache 1 has x, t, dirty

Client

Client

Client

Client

Server
Caching With Leases

Cache 1

\(x = 4, t, \text{dirty}\)

Cache 1 has \(x, t, \text{dirty}\)
Caching With Leases

Client

Cache 1

Server

Cache 2

Cache 1 has x, t, dirty

x = 4, t, dirty

ok!
Why does cache 1 wait until other copies are revoked and write is applied before returning ok to client?
Caching With Leases

Client  \rightarrow \text{Put } x, 5 \rightarrow \text{Cache 1}

$\text{Cache 1 has } x, t, \text{dirty}$

$\text{Cache 2}$

Client  \rightarrow \text{Cache 2}

$\text{Client}$

$\text{Server}$

$\text{Client}$

$\text{Client}$

$\text{Client}$
Caching With Leases

Client

Cache 1

x = 5, t, dirty

Cache 2

Cache 1 has x, t, dirty

Server
Caching With Leases

Client

Cache 1

Cache 2

Server

Cache 1 has x, t, dirty

x = 5, t, dirty

ok!
Caching With Leases

Client

Cache 1

x = 5, t, dirty

Cache 2

Cache 1 has x, t, dirty

Server

Client

get x
Caching With Leases

Client

Cache 1

x = 5, t, dirty

Cache 2

get x

Cache 1 has x, t, dirty

Server
Caching With Leases

Cache 1 has x, t, dirty

Revoke x shared

Server

Cache 1

x = 5, t, dirty

Cache 2

Cache 1 has x, t, dirty

Client

Client

Client

Client
Caching With Leases

Client

Cache 1

x = 5, t, shared

Cache 2

Server

Cache 1 has x, t, dirty

Client

Client

Client

Client
Caching With Leases

Cache 1
x = 5, t, shared
ok! x = 5

Cache 2

Cache 1 has x, t, dirty

Server
Caching With Leases

Client

Cache 1

x = 5, t, shared

Cache 2

ok! x = 5

Cache 1,2 has x, t, shared

Server
Caching With Leases

Client → Cache 1
  x = 5, t, shared

Client → Cache 2
  x = 5, t, shared

Cache 1, 2 has x, t, shared

Server
Questions

While a write to x is waiting on invalidations, can other clients read old values of x from their caches?
Questions

While a write to x is waiting on invalidations, can the server perform a read to y != x?
Questions

While a write to $x$ is waiting on invalidations, can the server perform a write (from another cache) to $y \neq x$?
Questions

While a write to $x$ is waiting on invalidations, can the server perform a write (from another cache) to $y = x$?
Write Back Cache Coherence

On a write:

- Send invalidations to all caches
- Each cache invalidates, responds (possibly with updated data)
- Wait for all invalidations
- Return

Reads can proceed when there is a local copy

Order requests carefully at server, avoid deadlock
MSI

Invalid → Read miss → Shared → Modified
MSI

Invalid → Modified

Shared
MSI

Invalid \(\rightarrow\) Write miss \(\rightarrow\) Modified

Shared
MSI

Invalid

Shared

Modified
MSI

Invalid

Modified

Local write

Shared
MSI

Invalid

Shared

Modified
MSI

Invalid

Remote write

Shared

Modified
MSI

Invalid

Shared

Modified
MSI

- Invalid
- Modified
- Shared

Write back / Remote read
MESI

Motivation:
- Common pattern: read, then write
- MSI inefficient when doing a read and then a write
- If no one else has a copy, can “claim” it with the read

Four cache states:
- **Modified**: this is the only copy, it’s dirty
- **Exclusive**: this is the only copy, it’s clean
- **Shared**: this is one of many copies, it’s clean
- **Invalid**
MESI allowed states

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>E</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>I</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
False Sharing

Expensive to keep track of MESI for every memory location

Instead, coarse-grained record-keeping

- CPUs: cache line granularity
- File systems: file/file block granularity

What if two clients try to modify different memory locations in same block, concurrently?

- Cache line can only be “dirty” in one at a time
- Correct behavior, but slow
Atomic Read-Modify-Write

RMW needed to implement spinlocks and other sync
Request cache line exclusive/modified
Delay concurrent remote read/write misses until entire operation completes
Multi-key Transactions

Often want to read/modify multiple keys atomically

Acquire cache lines in MESI state

If remote miss during transaction
  - Abort, erase modifications, and try again
  - Or delay until done

If reach end of transaction without remote miss
  - Success!
Weak leases

Cache valid until lease expires

Allow writes, other reads simultaneously

Semantics?
Weak leases

Examples: NFS, DNS, web browsers

Advantages
  - Stateless at server (don’t care who is caching)
  - Reads, writes always processed immediately

Disadvantages
  - Consistency model (!!!)
  - Overhead of revalidations
  - Synchronized revalidations