Implementing caches (part II)

Doug Woos
Logistics notes

Textbook chapter for Friday—no (graded) discussion
  - Only up to (not including) three-phase commit
Last time

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Invalidation</th>
<th>Leases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write policy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write-through</td>
<td>AFS (Andrew FS)</td>
<td>DNS</td>
</tr>
<tr>
<td>Write-back</td>
<td>Sprite</td>
<td>NFS</td>
</tr>
</tbody>
</table>
Rule for caches and shards

Due to Sarita Adve

Suppose each process specifies ops in some order

Sequentially consistent if:

1. Ops applied in processor order, and
2. All ops to a single key are serialized (as if to a single copy)

So how do we ensure ops go to a single copy?
Write-through invalidations

Track all reading caches

On a write:
- Send invalidations to all caches
- Each cache invalidates, responds
- Wait for all invalidations, do update
- Return

Reads can proceed:
- If there is a cached copy
- If no write waiting at server
Write-back caches
Processor cache coherence protocols
Leases and weak consistency
Write-back invalidations

Track all reading and writing caches

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

- Enforce processor order, avoid deadlock
 MSI/MESI

Protocols used for processor caches
Similar to protocol used e.g. in Sprite
Useful to understand
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:
Client

```
put (k1, f(data))
put (done1, true)
while (get(done1) == false)
    ;
put (k2, g(get(k1)));
put (done2, true)
while (get(done2) == false)
    ;
```

Server

```
k1 = 0
k2 = 0
done1 = false
done2 = false
```
```
put (k1, f(data))
put (done1, true)
while(get(done1) == false);
put (k2, g(get(k1)));
put (done2, true)
while(get(done2) == false);

rslt = h(get(k1), get(k2))
```
Client

put (k1, f(data))
put (done1, true)

while(get(done1) == false)
    put (k2, g(get(k1));
    put (done2, true)

Client

rslt = h(get(k1), get(k2))

Server

k1 = 0
k2 = 0
done1 = false
done2 = false
Client

```
put (k1, f(data))
put (done1, true)
while (get(done1) == false);
put (k2, g(get(k1)));
put (done2, true)
```

Server

```
k1 = 0
k2 = 0
done1 = false
done2 = false
```

Client

```
while (get(done2) == false);
rlst = h(get(k1), get(k2))
```

Client
```
put (k1, f(data))
put (done1, true)
while (get(done1) == false) {
    put (k2, g(get(k1)))
    put (done2, true)
}

rslt = h(get(k1), get(k2))
```
Client

```
put (k1, f(data))
put (done1, true)
while(get(done1) == false)
    put (k2, g(get(k1)));
    put (done2, true)
```

Server

```
k1 = 42; M
k1 = 0
k2 = 0
done1 = false; S
done2 = false
```

Client

```
while(get(done2) == false)
    rslt = h(get(k1), get(k2))
```
Client

put (k1, f(data))
put (done1, true)
while(get(done1) == false)
    put (k2, g(get(k1)));
    put (done2, true)
while(get(done2) == false)
    rslt = h(get(k1), get(k2))

Server

k1 = 42; M

k1 = 0
k2 = 0
done1 = false
done2 = false

done1 = false; S

done2 = false; S
Client

\[ \text{put} \ (k1, f(data)) \]
\[ \text{put} \ (done1, true) \]

\[ \text{while(get(done1) == false)} \]
\[ \quad \text{;} \]
\[ \text{put} \ (k2, g(get(k1))) \]
\[ \text{put} \ (done2, true) \]

\[ rslt = h(get(k1), get(k2)) \]

Server

\[ k1 = 42; M \]
\[ \text{done1} = \text{false}; I \]
\[ \text{done2} = \text{false}; S \]
Client

\[
\text{put } (k_1, f(\text{data}))
\]
\[
\text{put } (\text{done1, true})
\]

Client

\[
\text{while(get(done1) == false)}
\]
\[
; \\
\text{put } (k_2, g(\text{get(k1)})); \\
\text{put } (\text{done2, true})
\]

Client

\[
\text{while(get(done2) == false)}
\]
\[
; \\
r\text{slt } = h(\text{get(k1), get(k2)})
\]

Server

\[
\text{k1 } = 42; M \\
\text{done1 } = \text{true}; M \\
\text{k1 } = 0 \\
\text{k2 } = 0 \\
\text{done1 } = \text{false} \\
\text{done2 } = \text{false}
\]
Client

put (k1, f(data))
put (done1, true)

while(get(done1) == false)
    ;
put (k2, g(get(k1)));
put (done2, true)

Client

rslt = h(get(k1), get(k2))

Server

k1 = 42; M
done1 = true; M
done1 = false; l

done2 = false; S

done2 = false
Client

put (k1, f(data))
put (done1, true)
while (get(done1) == false) {
    put (k2, g(get(k1));
    put (done2, true)
}

Server

k1 = 42; M
done1 = true; S

Client

done1 = false; l

Client

done2 = false; S

while (get(done2) == false) {
    rslt = h(get(k1), get(k2))
}
Client

put \((k_1, f(data))\)
put \((\text{done1}, \text{true})\)

while (get(\text{done1}) == \text{false})
{
put \((k_2, g(\text{get}(k_1)))\);
put \((\text{done2}, \text{true})\);
}

rslt = h(\text{get}(k_1), \text{get}(k_2))

Server

k1 = 42; M
done1 = true; S

k1 = 0
done1 = true

k2 = 0
done2 = false

Client

done1 = true; S

Client

done2 = false; S
Client

put (k1, f(data))
put (done1, true)

while(get(done1) == false)
    put (k2, g(get(k1)));
put (done2, true)

rslt = h(get(k1), get(k2))

Server

k1 = 42; M
done1 = true; S

Client

k1 = 0
k2 = 0
done1 = true
done2 = false

Client

done1 = true; S

done2 = false; S
Client

\[
\begin{align*}
\text{put } (k1, f(data)) \\
\text{put } (\text{done1}, \text{true}) \\
\text{while}(\text{get}(\text{done1}) \neq \text{false}) \\
\quad \text{put } (k2, g(get(k1))) \\
\text{put } (\text{done2}, \text{true}) \\
rslt = h(get(k1), get(k2))
\end{align*}
\]

Server

\[
\begin{align*}
k1 &= 42; \ M \\
done1 &= \text{true}; \ S
\end{align*}
\]

while(get(done1) == false) \\
\[
\begin{align*}
\text{while}(\text{get}(\text{done2}) \neq \text{false}) \\
rslt &= h(get(k1), get(k2))
\end{align*}
\]

\[
\begin{align*}
done1 &= \text{true}; \ S \\
done2 &= \text{false}; \ S
\end{align*}
\]
Client

\[
\text{put } (k1, f(data)) \text{;}
\]

\[
\text{put } (\text{done1}, \text{true}) \text{;}
\]

\[
\text{while(} \text{get(}\text{done1} \text{)} = \text{false} \text{)} \text{;}
\]

\[
\text{put } (k2, g(\text{get}(k1))) \text{;}
\]

\[
\text{put } (\text{done2}, \text{true}) \text{;}
\]

\[
\text{rslt} = h(\text{get}(k1), \text{get}(k2))
\]

Server

\[
\begin{align*}
\text{k1} &= 42; \ S \\
\text{done1} &= \text{true}; \ S \\
\text{k2} &= 0 \\
\text{done2} &= \text{false}
\end{align*}
\]
put (k1, f(data)))
put (done1, true)

while(get(done1) == false)
;
put (k2, g(get(k1)));
put (done2, true)

rslt = h(get(k1), get(k2))

k1 = 42; S
done1 = true; S
k1 = 42; S

Server
k1 =42
k2 = 0
done1 = true
done2 = false

Client

Client

Client
```
put (k1, f(data))
put (done1, true)
while(get(done1) == false)
    put (k2, g(get(k1)))
    put (done2, true)

rslt = h(get(k1), get(k2))
```
MSI

Invalid

Shared

Modified
MSI

- Invalid
- Modified

Read miss

- Shared
MSI

- Invalid
- Shared
- Modified

Remote write
MESI

Motivation:
- Common pattern: i++ (read, then a 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>
data = get(k1)
put (k1, f(data))
put (done1, true)

while(get(done1) == false) ;
put (k2, g(get(k1)));
put (done2, true)

while(get(done2) == false) ;
rslt = h(get(k1), get(k2))

k1 is “Exclusive” to N. America after first read
Can modify without sync
Caching implementations

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Write policy</th>
<th>Invalidation</th>
<th>Lease</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write-through</td>
<td></td>
<td>AFS (Andrew FS)</td>
<td>DNS</td>
</tr>
<tr>
<td>Write-back</td>
<td></td>
<td>Sprite</td>
<td>NFS</td>
</tr>
</tbody>
</table>
Strong leases

Read request: key, TTL (time to live)
When server returns:
- It won’t accept writes to the key
- For TTL seconds after reply sent
Client invalidates its cache after TTL seconds
- From when request was sent
Strong leases

For write-through:

- Server queues writes until all leases expire
- Avoid starvation: don’t accept new reads

For write-back:

- Cache can get a write lease (exclusive)
- Server queues read requests until lease expires
Clock issues

How long should the server wait on a lease?
How long should the client wait on a lease?
What about clock skew?

- Add $\epsilon$ on server, subtract $\epsilon$ on client
Strong leases vs. Invalidations

What are advantages/disadvantages of each?
Strong leases vs. Invalidations

What are advantages/disadvantages of each?

- Strong leases potentially slower

- What if a cache fails when it has a key? Strong leases provide better availability

Can combine techniques

- Short lease on entire cache, periodically revalidated

- All keys invalidated on failure (after lease)
Weak leases

Cache values 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
Discussion

“Complexity” as a downside

Do the scalability/performance issues mentioned in the paper exist today?

Why do we use NFS?
send_money(user1, user2, amount) {
    Begin_Transaction();
    if (user1.balance - amount >= 0) {
        user1.balance = user1.balance - amount;
        user2.balance = user2.balance + amount;
        Commit_Transaction();
    } else {
        Abort_Transaction();
    }
}
Next time

How to ensure that transactions are atomic?
Even when data are sharded?
Even when nodes can fail?