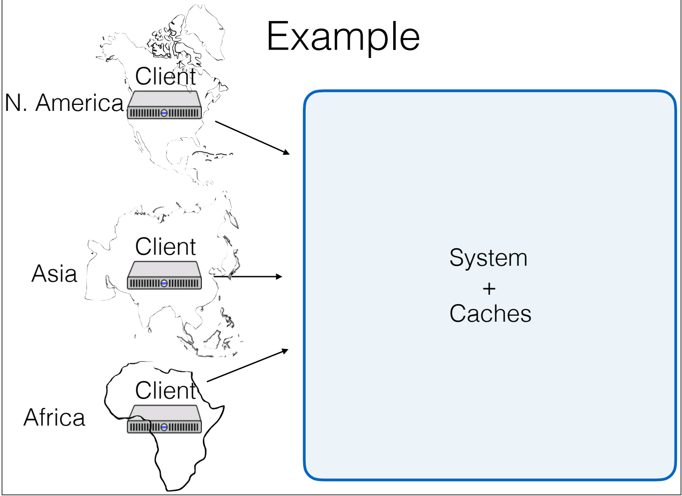
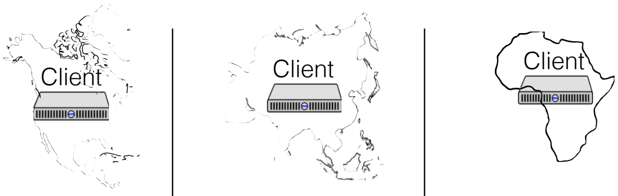
Implementing caches

Doug Woos

Example





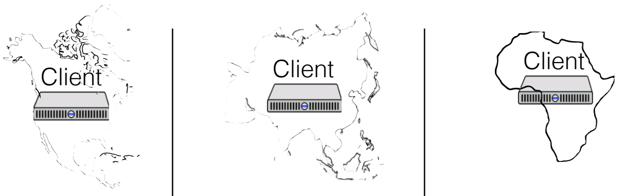
```
put (k1, f(data))
put (done1, true)
```

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

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

What if clients use a sharded key-value store to coordinate their output?

Or CPUs use memory to coordinate?



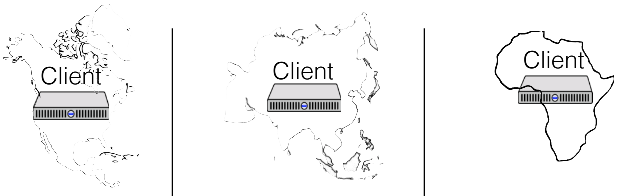
```
put (k1, f(data))
put (done1, true)
```

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

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

Write buffering: Can we start to write *done1* before we finish write to *k1*?

Yes, if order enforced at the server



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

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

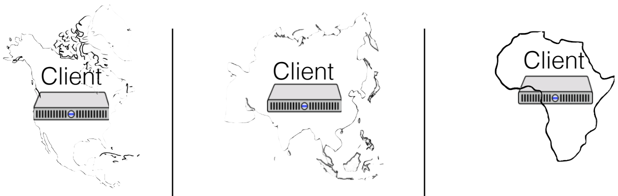
```
while(get(done2) == false)
     ;
rslt = h(get(k1), get(k2))
```

Write buffering: Can we start to write *done1* before we finish write to *k1*?

Yes, if order enforced at the server

No, if sharded and want linearizability: must serialize writes

---



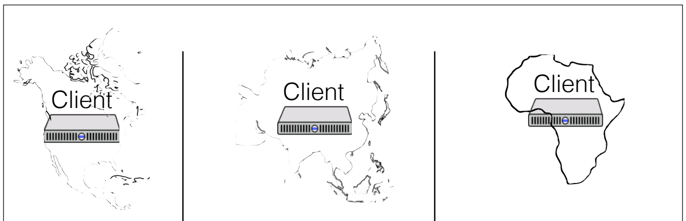
```
put (k1, f(data))
put (done1, true)
```

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

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

What if caches can hold out of date data?

What might go wrong?



```
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))
```

Asia: done1 = true, cached (old) k1

Africa: done2 = true, cached (old) k1 and k2

Africa: done2 = true, k2 correct, cached k1 (!)

Rule for caches and shards

Suppose each process specifies operations in some order

Sequentially consistent if:

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

How do we ensure #2?

- Can study each memory location in isolation

Invalidations vs. Leases

Invalidations

- Track where data is cached
- When doing a write, invalidate all (other) locations
- Data can live in multiple caches during reads

Leases

- Permission to serve data for some time period
- (if weak) eventually consistent
- (if strong) Wait until lease expires before update

Write-through vs. write-back

Write-through

- Writes go to the server
- Caches only hold clean data

Write-back

- Writes go to cache
- Dirty cache data written to server when necessary

Write-through vs. write-back

Mechanism **Write policy**	**Invalidations**	**Leases**
Write-through	AFS (Andrew FS)	DNS
Write-back	Sprite	NFS

Write-through invalidations

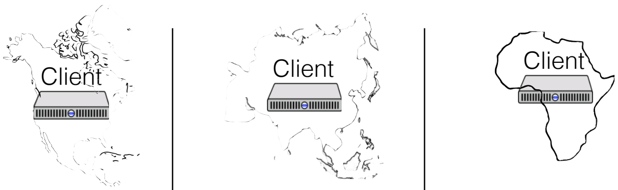
Track all caches with read copies

On a write:

- Send invalidations to all caches with a copy
- Each cache invalidates, responds
- Wait for all invalidations, do update
- Return

Reads can proceed:

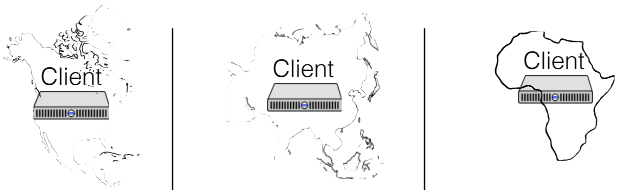
- If there is a cached copy
- or if cache miss and no write waiting at server



```
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))
```

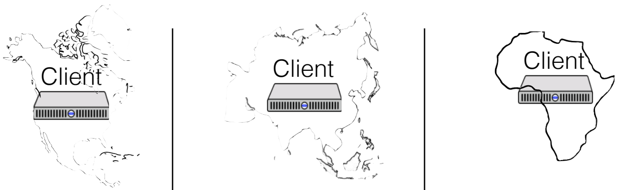


```
put (k1, f(data))
put (done1, true)
```

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

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

done1=false



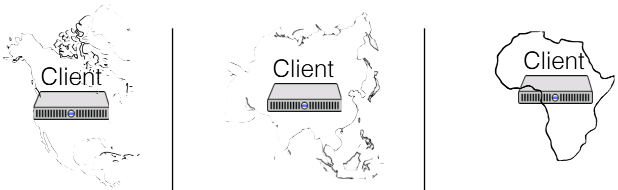
```
put (k1, f(data))
put (done1, true)
```

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

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

done1=false

done2=false



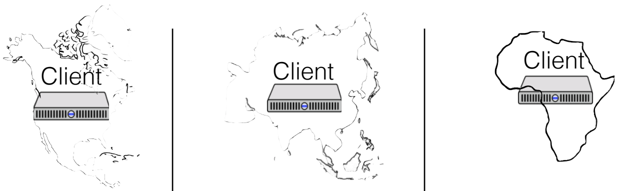
```
put (k1, f(data))
put (done1, true)
```

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

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

done1=false

done2=false



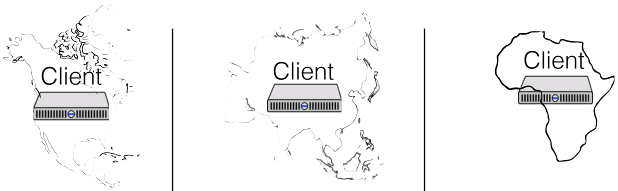
```
put (k1, f(data))
put (done1, true)
```

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

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

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))
```

~~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))
```

done1=true

done2=false

Questions

While a write is waiting on invalidations, can clients read old values from caches?

Questions

While a write is waiting on invalidations, can the writing client perform a different write?

Questions

While a write is waiting on invalidations, can the server process a read to a different location?

Questions

While a write is waiting on invalidations, can the server process a read to the same location?

Questions

While a write is waiting on invalidations, can the server process a write to a different location?

Questions

While a write is waiting on invalidations, can the server process a write to the same location?

More Questions

Why does the server wait until write is applied before returning to the client?

Why queue incoming requests during a write?

How much directory state is needed at server?

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

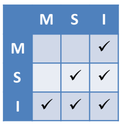
Useful to understand

MSI

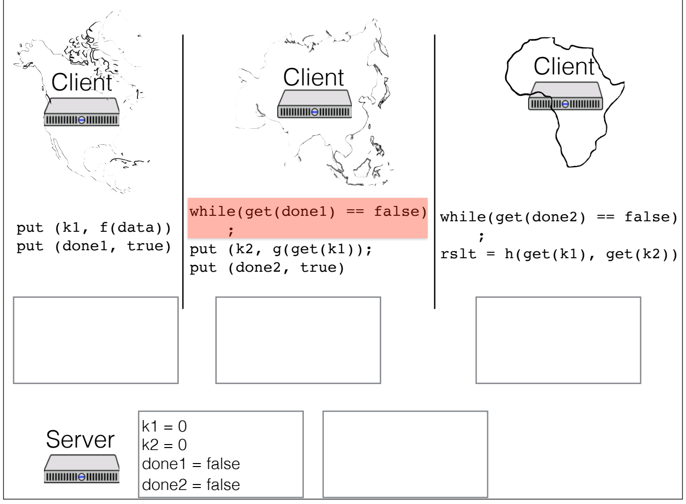
Three cache states:

- **M**odified: this is the only copy, it's dirty
- **S**hared: this is one of many copies, it's clean
- **I**nvalid

Allowed states between pairs of caches:



	M	S	I
M			✓
S		✓	✓
I	✓	✓	✓

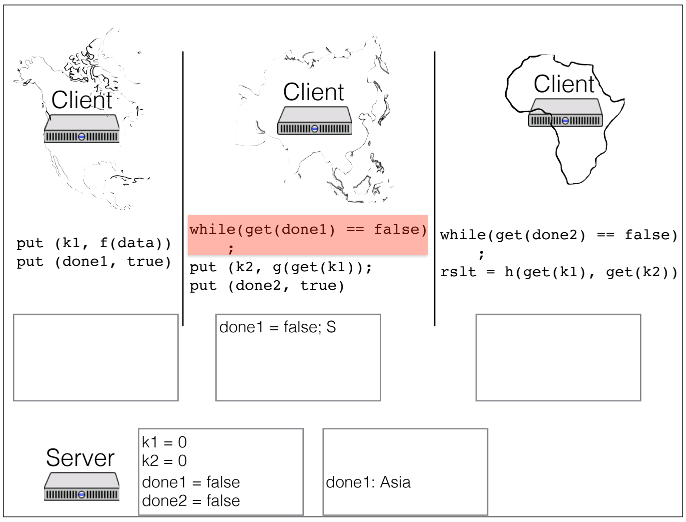


```
put (k1, f(data))
put (done1, true)
```

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

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

k1 = 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))
```

done1 = false; S

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

done1: Asia

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

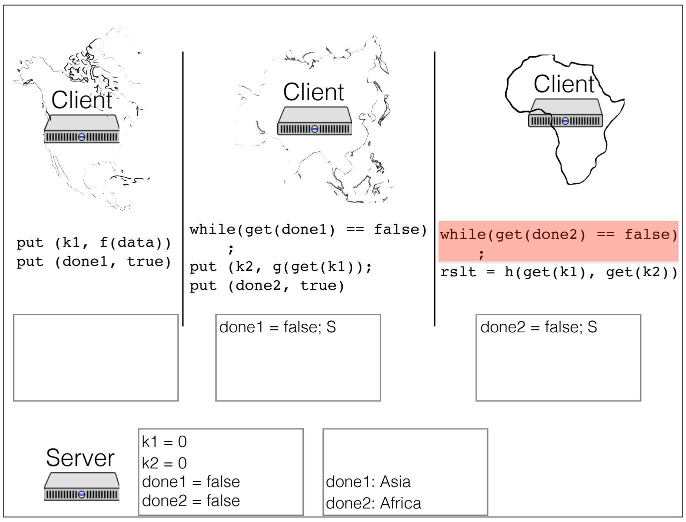
```
while(get(done1) == false)
    ;
put (k2, g(get(k1)));
put (done2, true)
```

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

done1 = false; S

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

done1: Asia



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

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

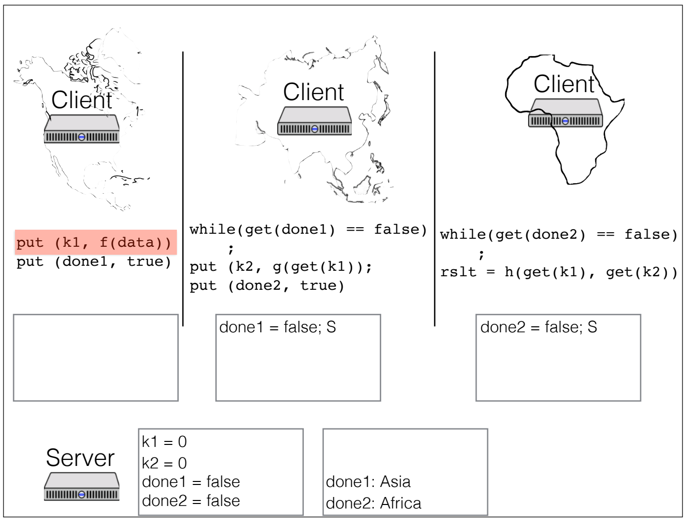
```
while(get(done2) == false)
    ;
rslt = h(get(k1), get(k2))
```

done1 = false; S

done2 = false; S

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

done1: Asia
done2: Africa



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

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

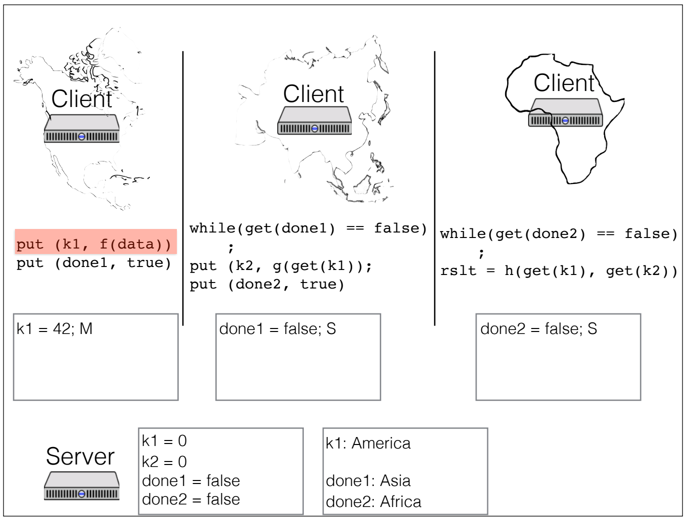
```
while(get(done2) == false)
    ;
rslt = h(get(k1), get(k2))
```

done1 = false; S

done2 = false; S

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

done1: Asia
done2: Africa



```
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 = 42; M

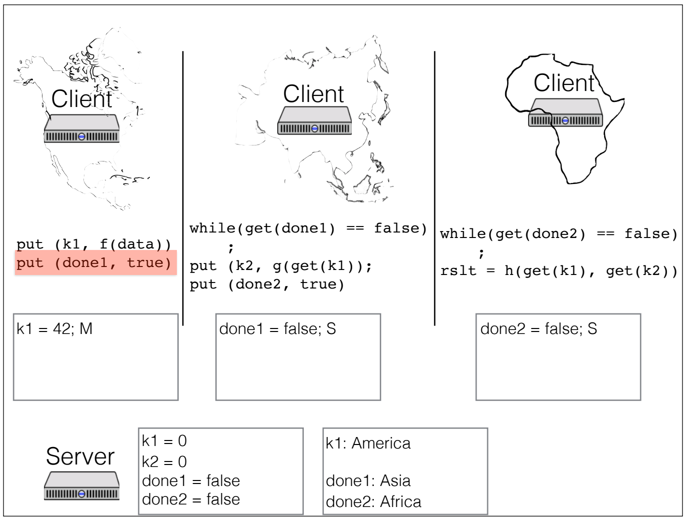
done1 = false; S

done2 = false; S

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

k1: America

done1: Asia
done2: Africa



```
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 = 42; M

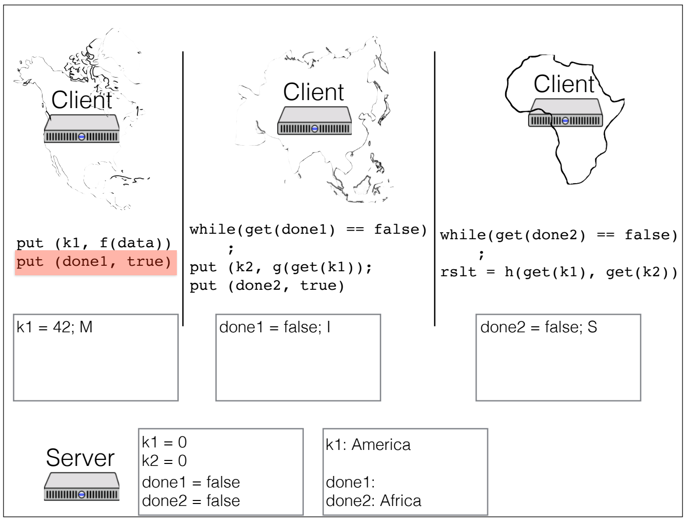
done1 = false; S

done2 = false; S

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

k1: America

done1: Asia
done2: Africa



```
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 = 42; M

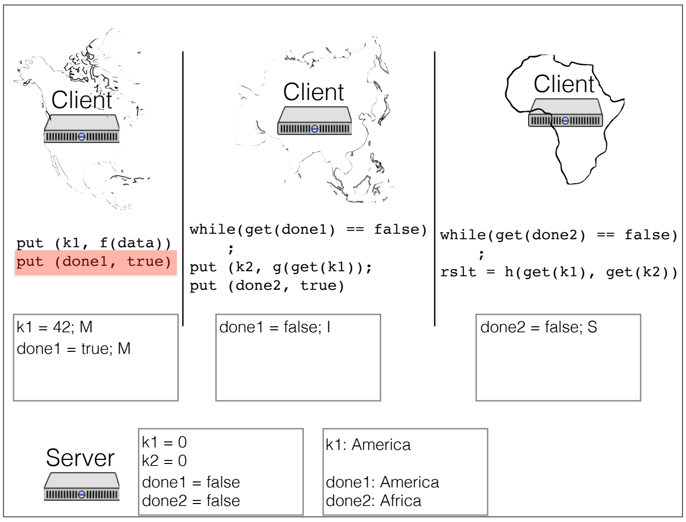
done1 = false; I

done2 = false; S

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

k1: America

done1:
done2: Africa



```
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 = 42; M
done1 = true; M

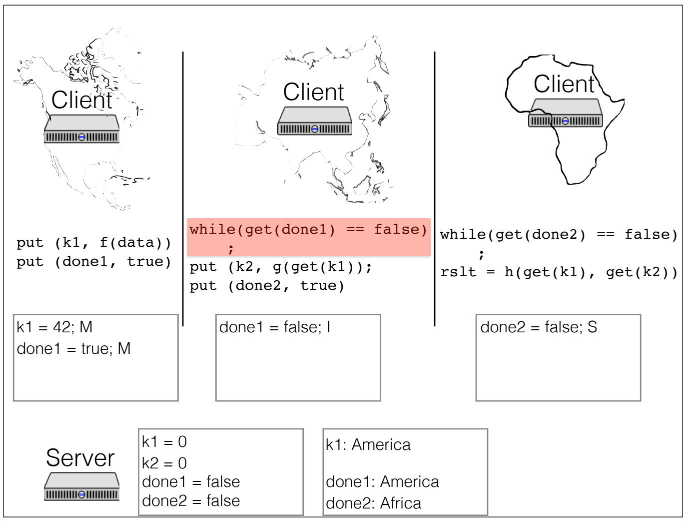
done1 = false; I

done2 = false; S

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

k1: America

done1: America
done2: Africa



```
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 = 42; M
done1 = true; M

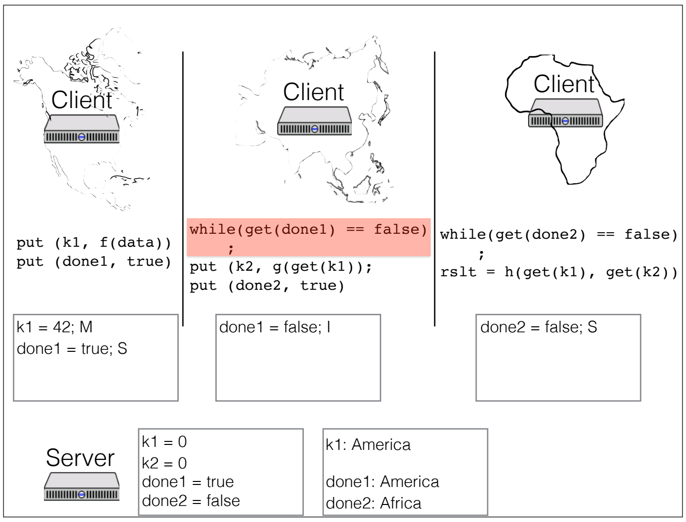
done1 = false; I

done2 = false; S

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

k1: America

done1: America
done2: Africa



```
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 = 42; M
done1 = true; S

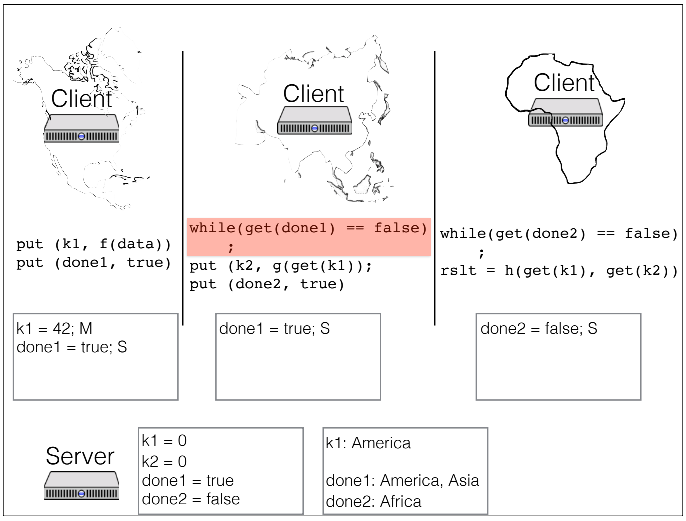
done1 = false; I

done2 = false; S

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

k1: America

done1: America
done2: Africa



```
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 = 42; M
done1 = true; S

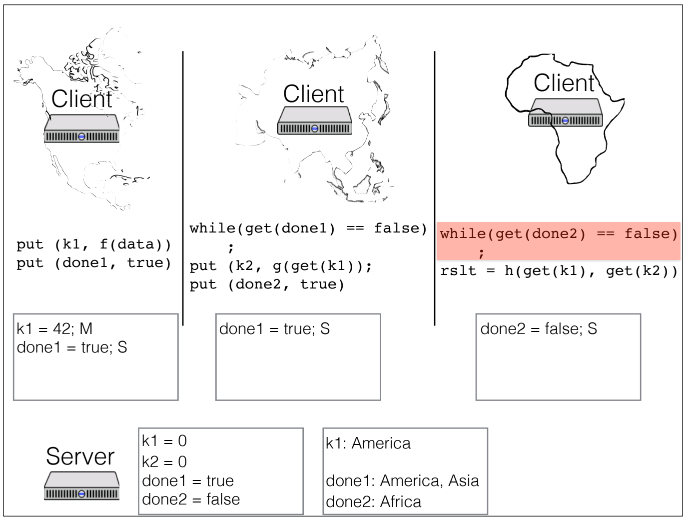
done1 = true; S

done2 = false; S

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

k1: America

done1: America, Asia
done2: Africa



```
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 = 42; M
done1 = true; S

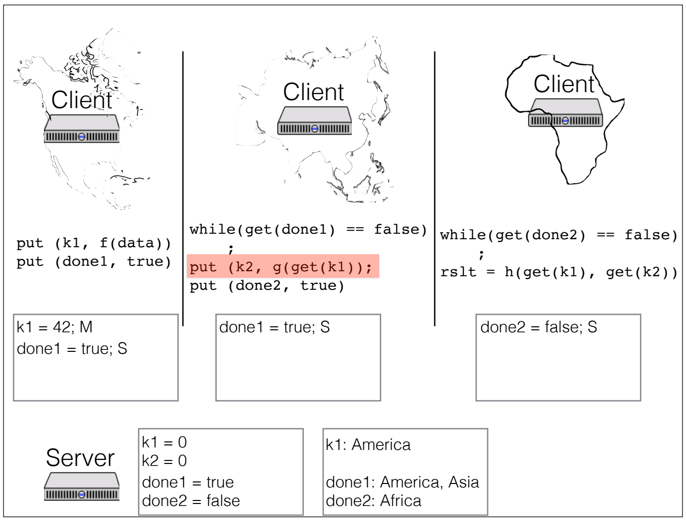
done1 = true; S

done2 = false; S

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

k1: America

done1: America, Asia
done2: Africa



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

k1 = 42; M
done1 = true; S

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

done1 = true; S

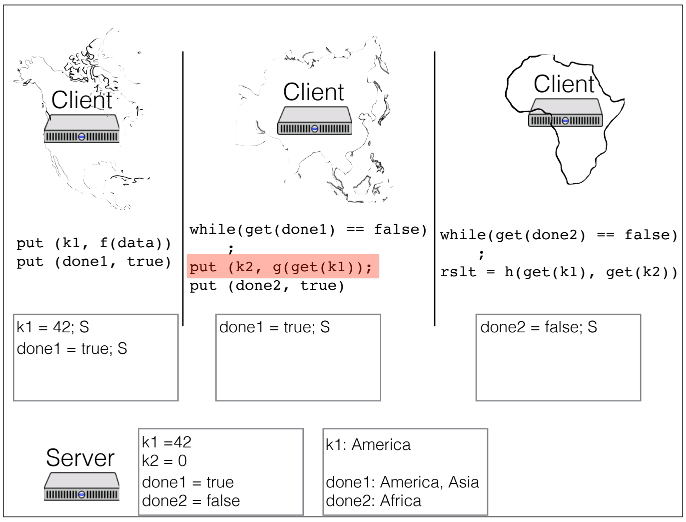
```
while(get(done2) == false)
      ;
rslt = h(get(k1), get(k2))
```

done2 = false; S

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

k1: America

done1: America, Asia
done2: Africa



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

k1 = 42; S
done1 = true; S

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

done1 = true; S

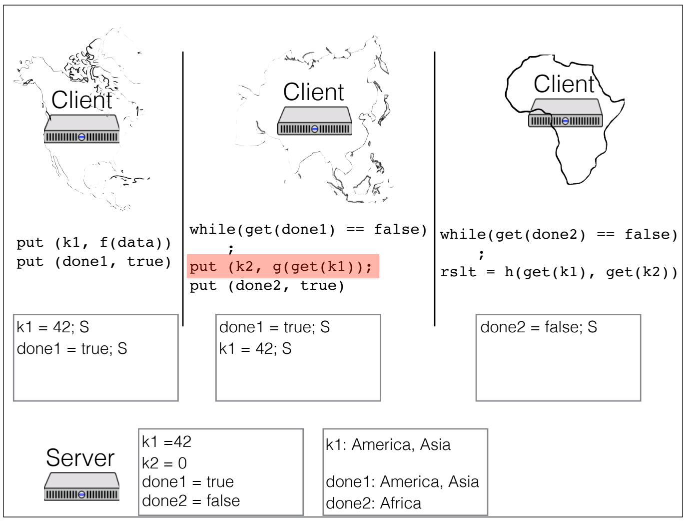
```
while(get(done2) == false)
      ;
rslt = h(get(k1), get(k2))
```

done2 = false; S

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

k1: America

done1: America, Asia
done2: Africa



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

k1 = 42; S
done1 = true; S

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

done1 = true; S
k1 = 42; S

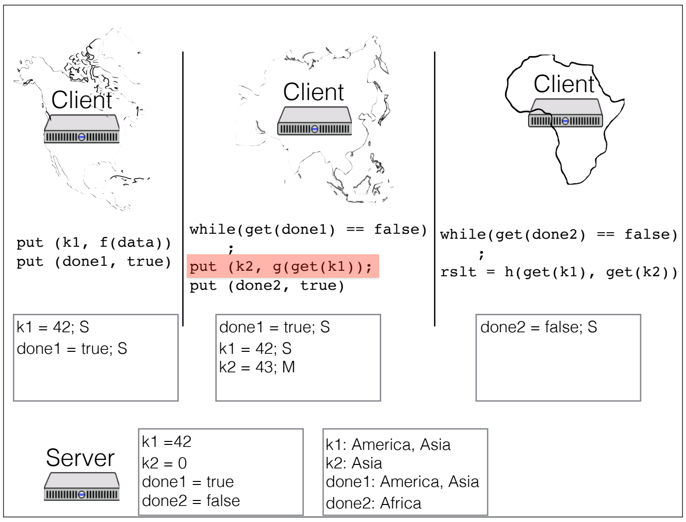
```
while(get(done2) == false)
      ;
rslt = h(get(k1), get(k2))
```

done2 = false; S

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

k1: America, Asia

done1: America, Asia
done2: Africa



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

k1 = 42; S
done1 = true; S

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

done1 = true; S
k1 = 42; S
k2 = 43; M

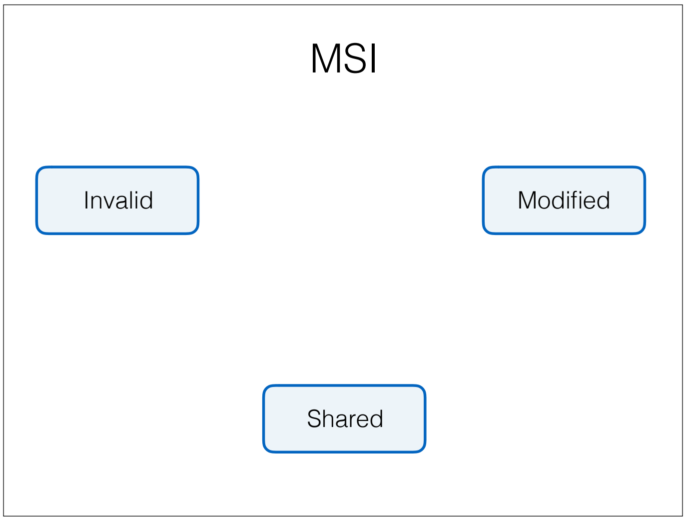
```
while(get(done2) == false)
      ;
rslt = h(get(k1), get(k2))
```

done2 = false; S

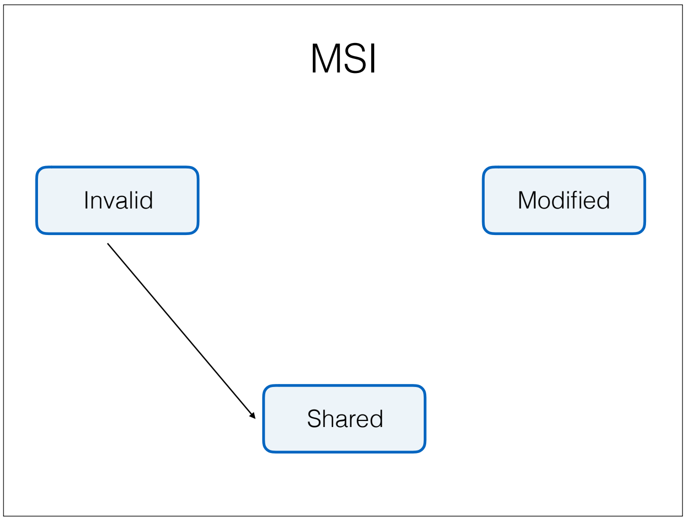
k1 =42
k2 = 0
done1 = true
done2 = false

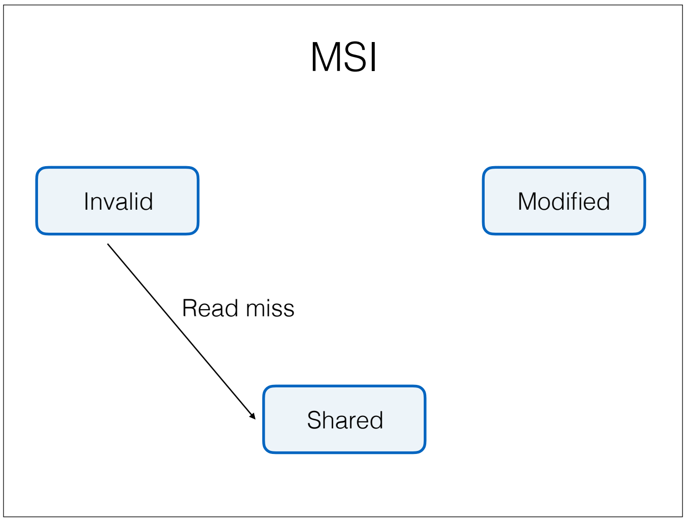
k1: America, Asia
k2: Asia
done1: America, Asia
done2: Africa

MSI



MSI





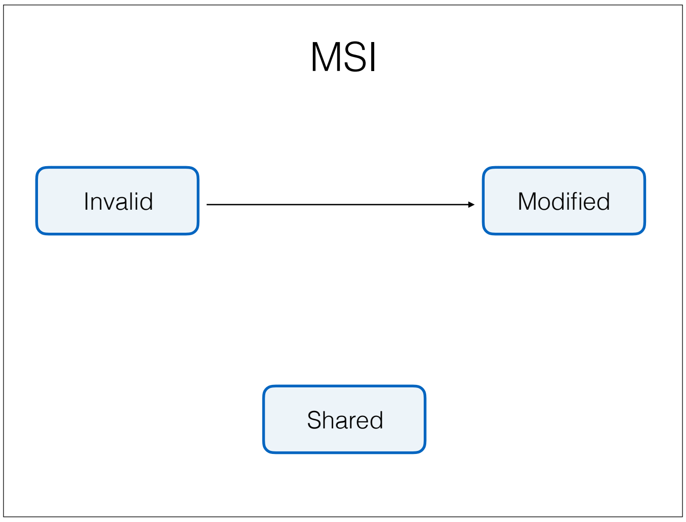
MSI

Invalid

Modified

Read miss

Shared

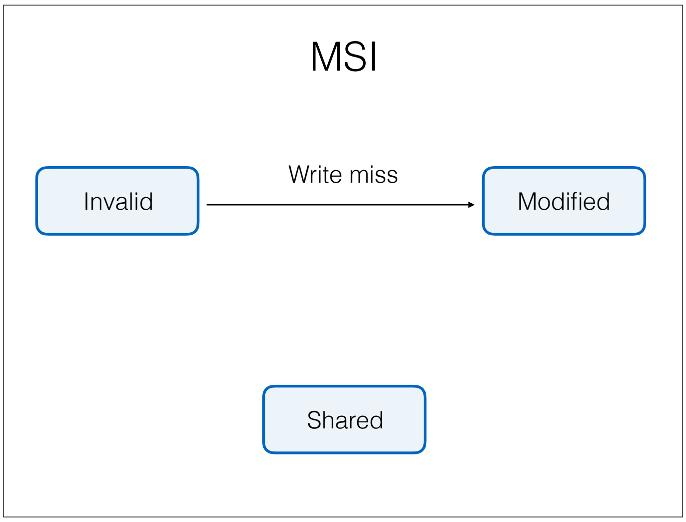


MSI

Invalid

Modified

Shared



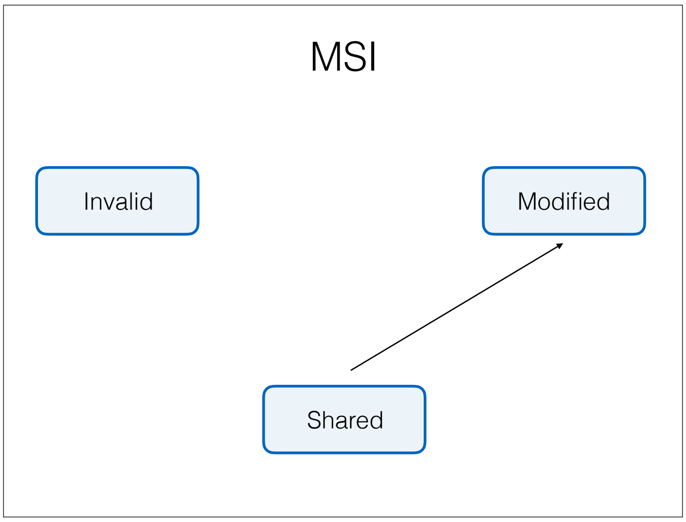
MSI

Invalid

Write miss

Modified

Shared

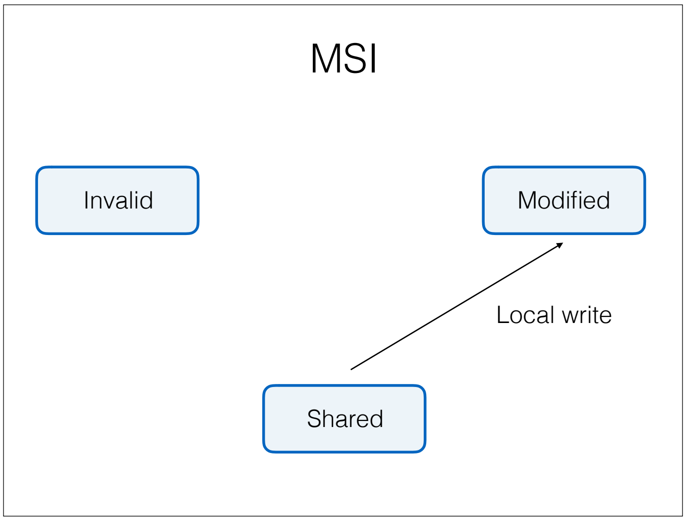


MSI

Invalid

Modified

Shared



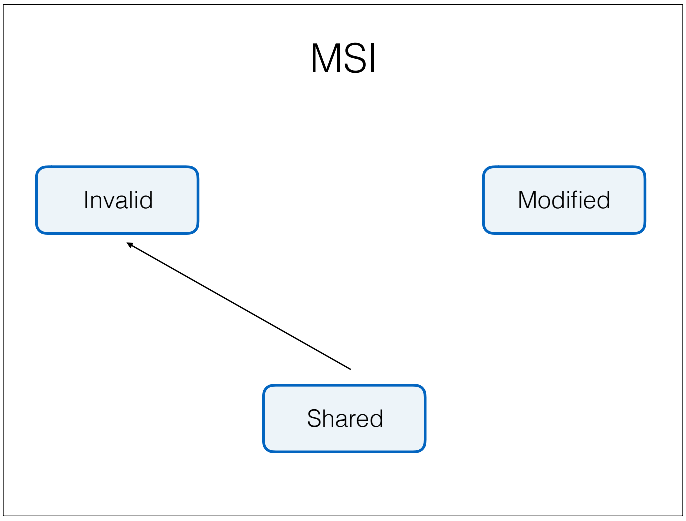
MSI

Invalid

Modified

Local write

Shared



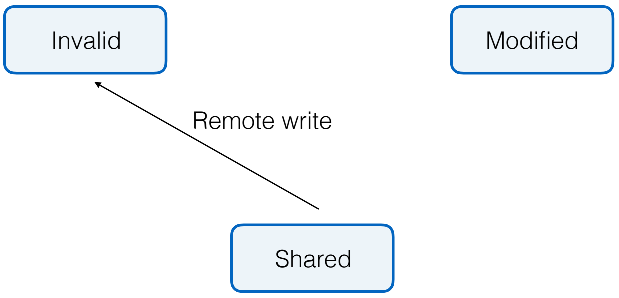
MSI

Invalid

Modified

Shared

MSI



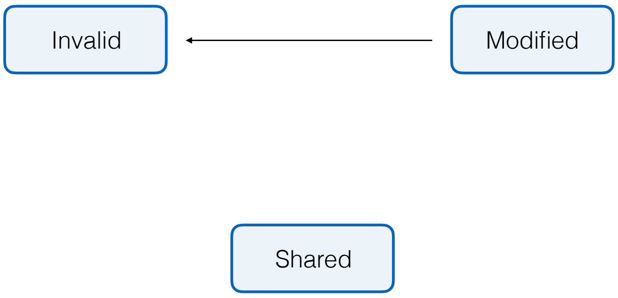
Invalid

Modified

Remote write

Shared

MSI

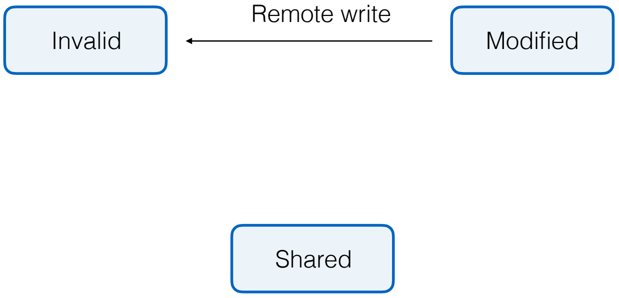


Invalid

Modified

Shared

MSI



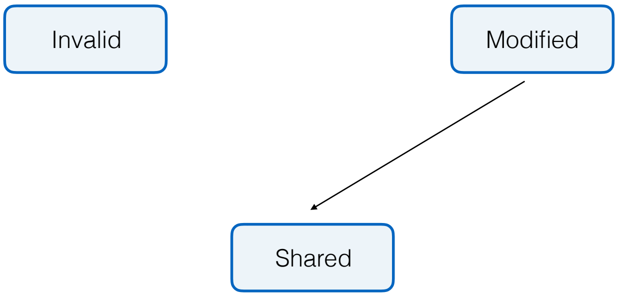
Invalid

Remote write

Modified

Shared

MSI

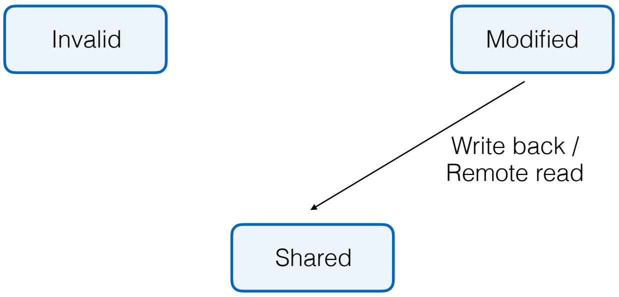


Invalid

Modified

Shared

MSI



Invalid

Modified

Write back /
Remote read

Shared

MESI

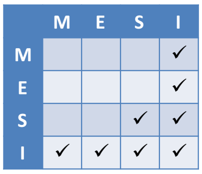
Motivation:

- 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:

- **M**odified: this is the only copy, it’s dirty
- **E**xclusive: this is the only copy, it’s clean
- **S**hared: this is one of many copies, it’s clean
- **I**nvalid

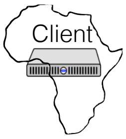
MESI allowed states



	M	E	S	I
M				✓
E				✓
S			✓	✓
I	✓	✓	✓	✓

```
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

False Sharing

Expensive to keep track of MESI for every memory location

Instead, coarse-grained record-keeping

- On CPUs, at the cache line granularity
- In file systems, at the granularity of a file/file block

What if two clients try to modify different memory locations in the same cache line, concurrently?

- Cache line can only be “modified" 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

Software Transactions (CPUs)

Often want multiple instructions to execute atomically

- Critical section, supported in hardware

May involve multiple cache lines

Execute normally: acquire cache lines in MESI state

If remote miss during the software transaction

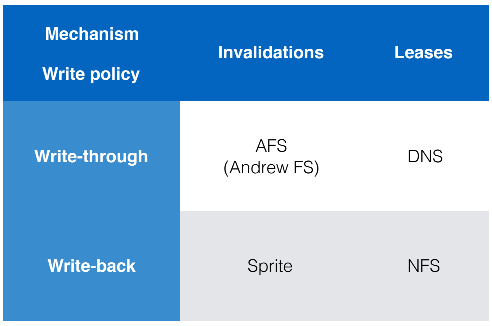
- Abort transaction, erase modifications, and try again

If reach end of software transaction without remote miss

- Success!

Distributed transactions (with node failures) next time!

Caching implementations



Mechanism Write policy	Invalidations	Leases
Write-through	AFS (Andrew FS)	DNS
Write-back	Sprite	NFS

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

Assumes bounded physical clock sync

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 ε on server, subtract ε 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 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

Discussion

“Complexity” as a downside

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

Why do we use NFS?