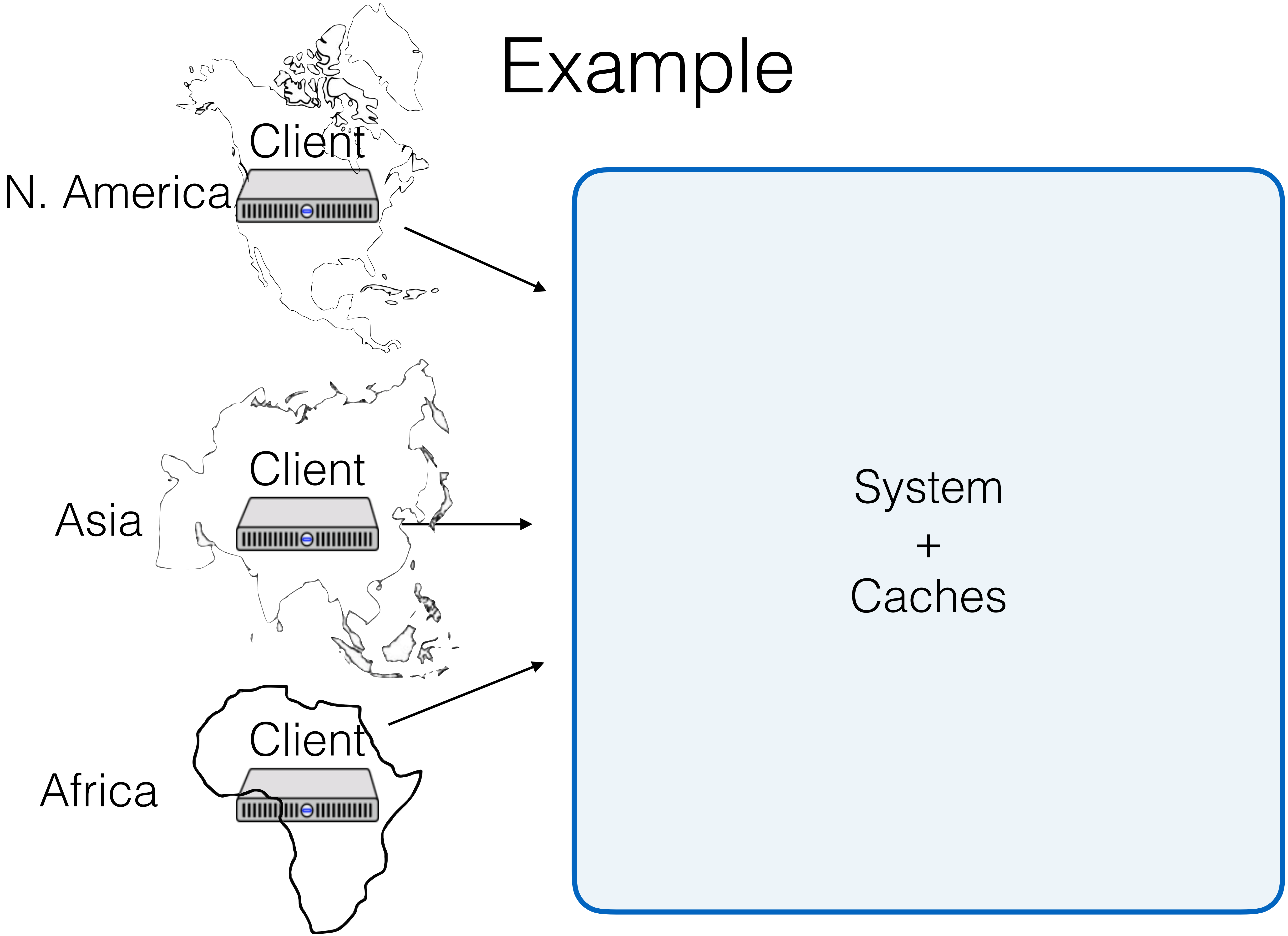
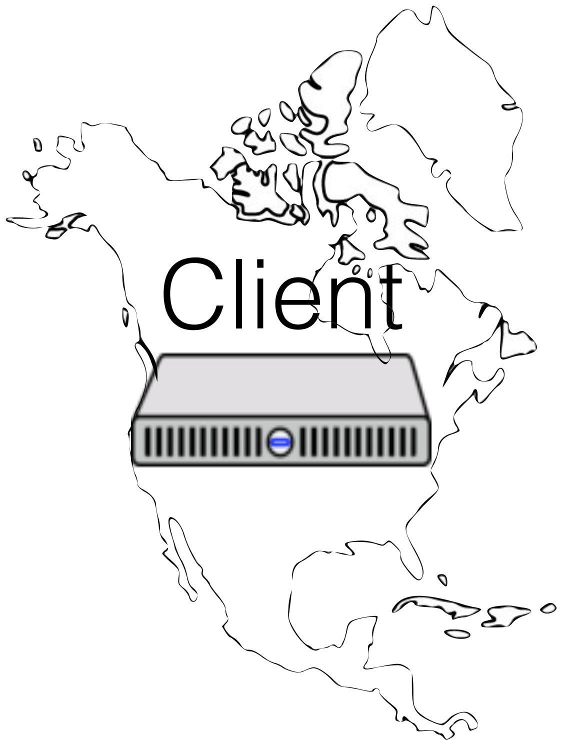
Caches & Memcache

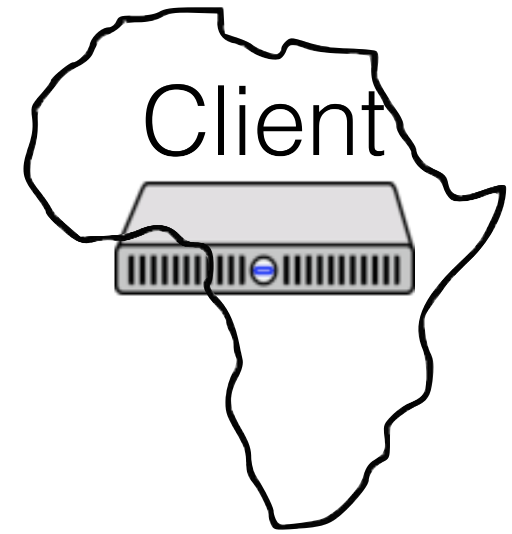
Example





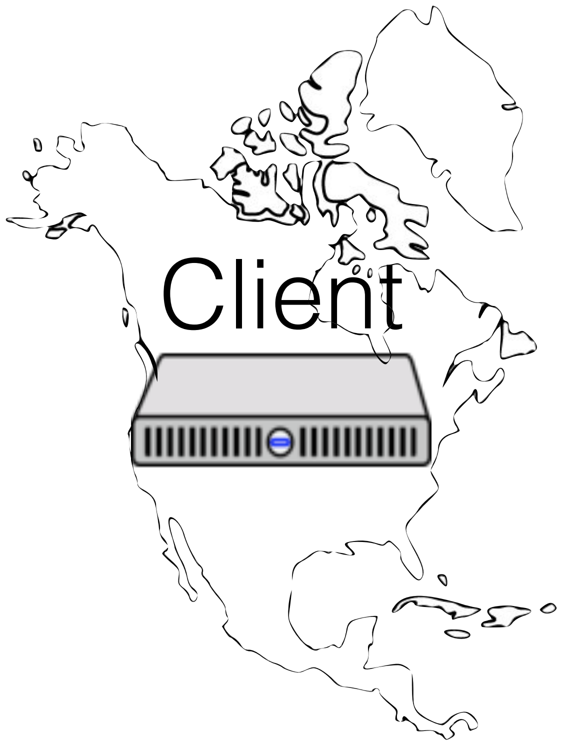

```
put (k1, f(data))
put (done1, true)
```


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



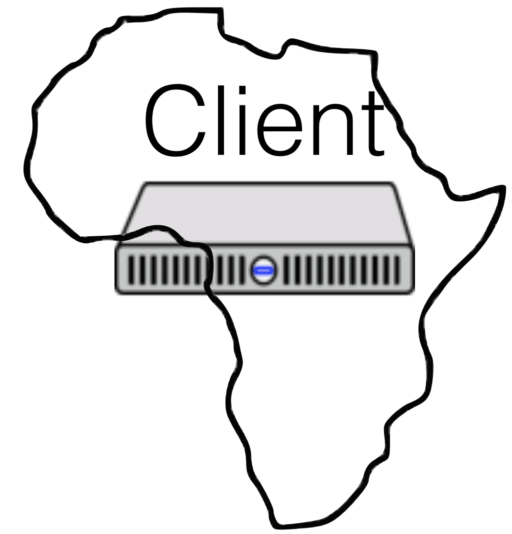

```
while(get(done2) == false)
    ;
rslt = h(get(k1), get(k2))
```

Assume that clients use a sharded key-value store to coordinate their output



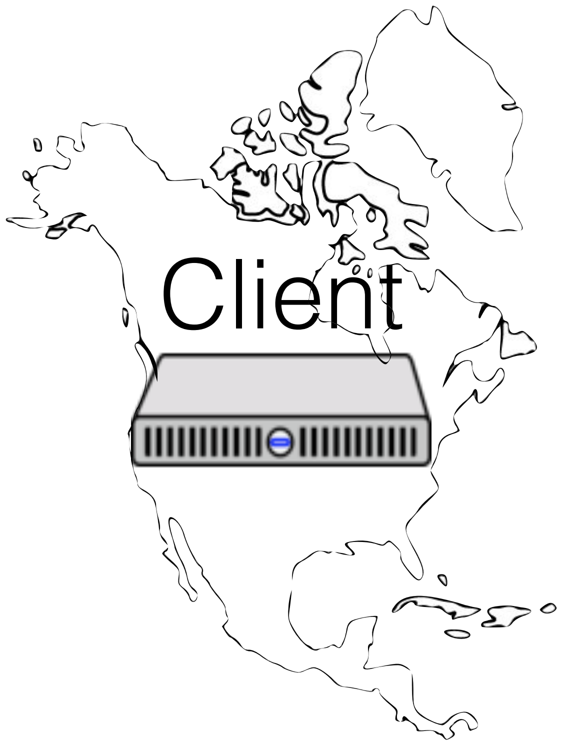

```
put (k1, f(data))
put (done1, true)
```


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



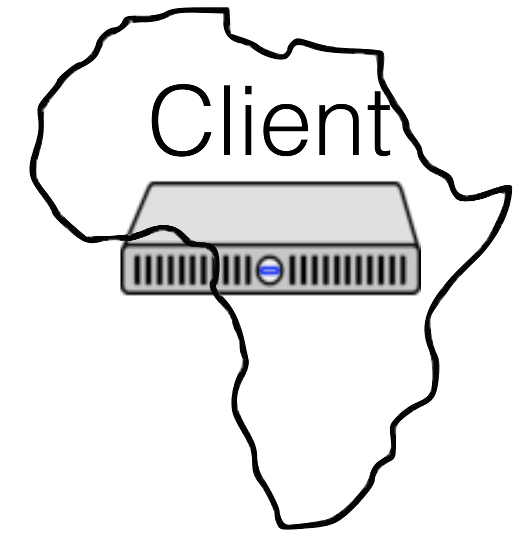

```
while(get(done2) == false)
    ;
rslt = h(get(k1), get(k2))
```

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




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

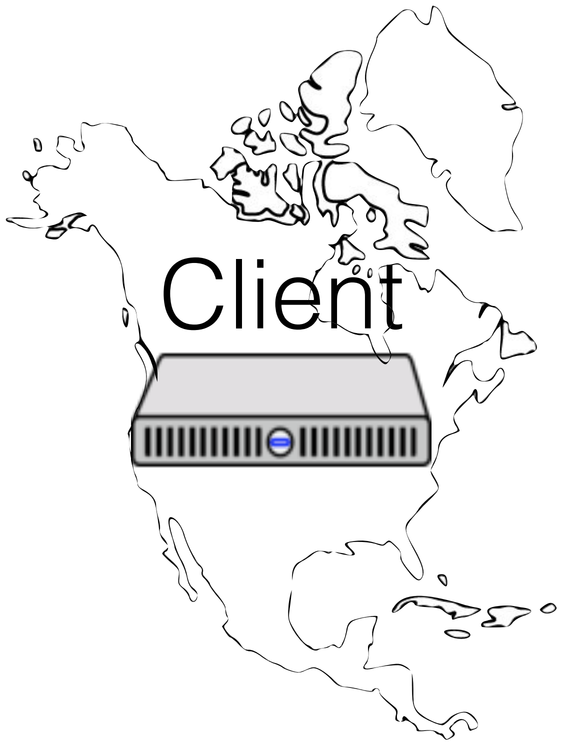

```
while(get(done1) == false)
    ;
put (k2, g(get(k1));
put (done2, true)
```




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

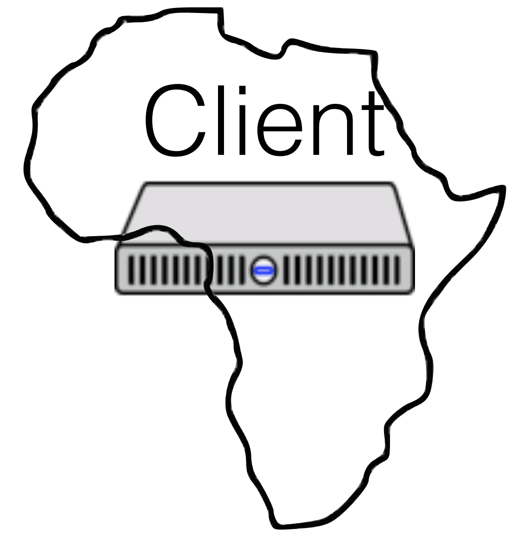
Write buffering: Can we start to write *done1* before we finish write to *k1*?

No, if sharded and want linearizability: must serialize writes




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

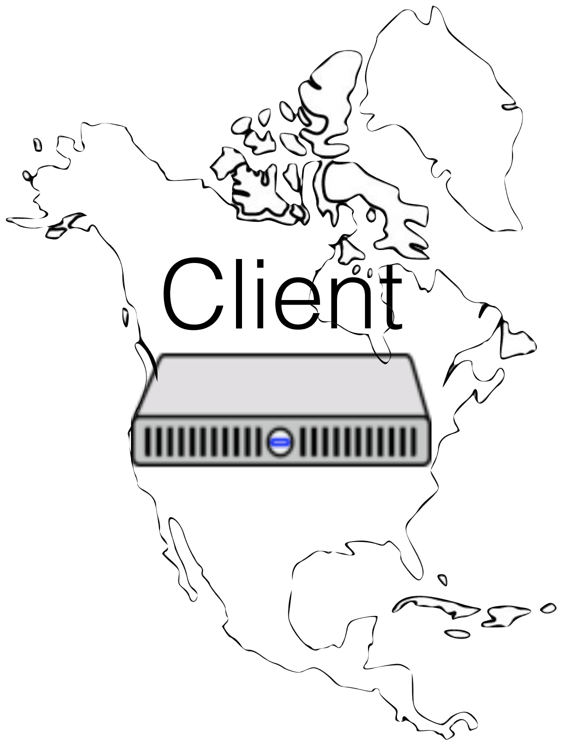

```
while(get(done1) == false)
    ;
put (k2, g(get(k1));
put (done2, true)
```




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

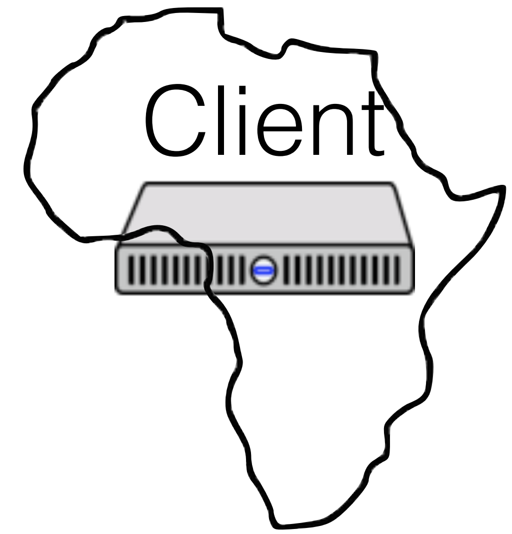
What if caches can hold out of date data?

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 (!)

Rules for caches and shards

Correct execution 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 serialize 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
- 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

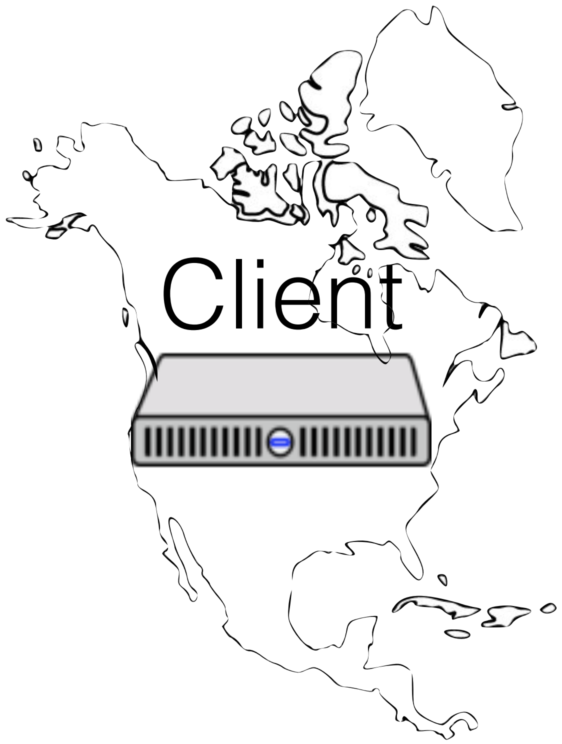
Track all caches with read copies

On a write:

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

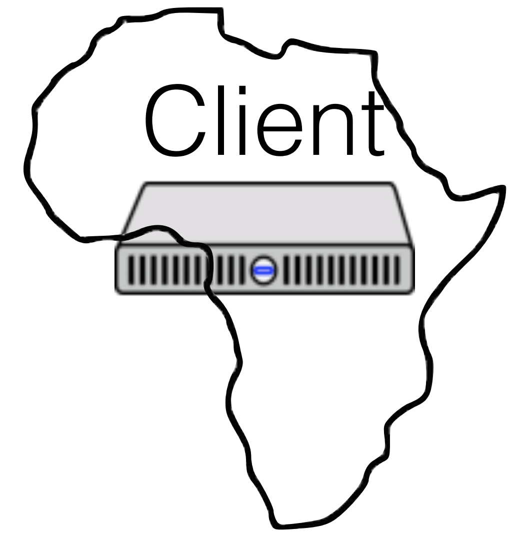
Reads can proceed:

- If there is a cached copy
- or if cache miss, read at server




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

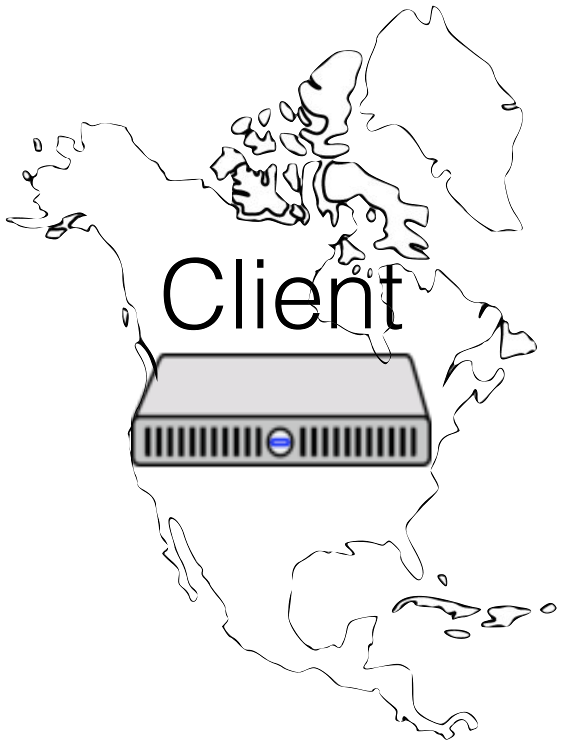

```
while(get(done1) == false)
    ;
put (k2, g(get(k1));
put (done2, true)
```




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

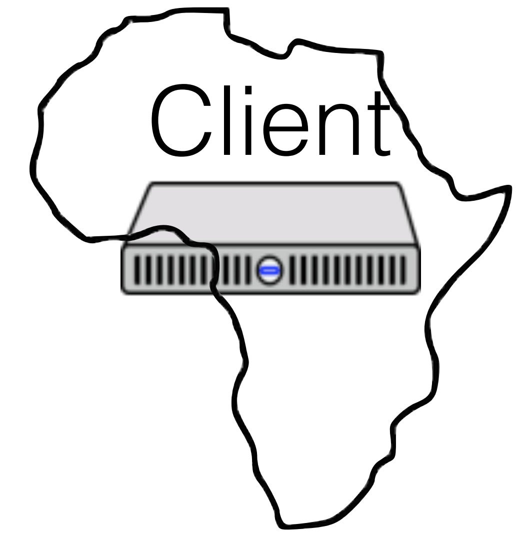
Server

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

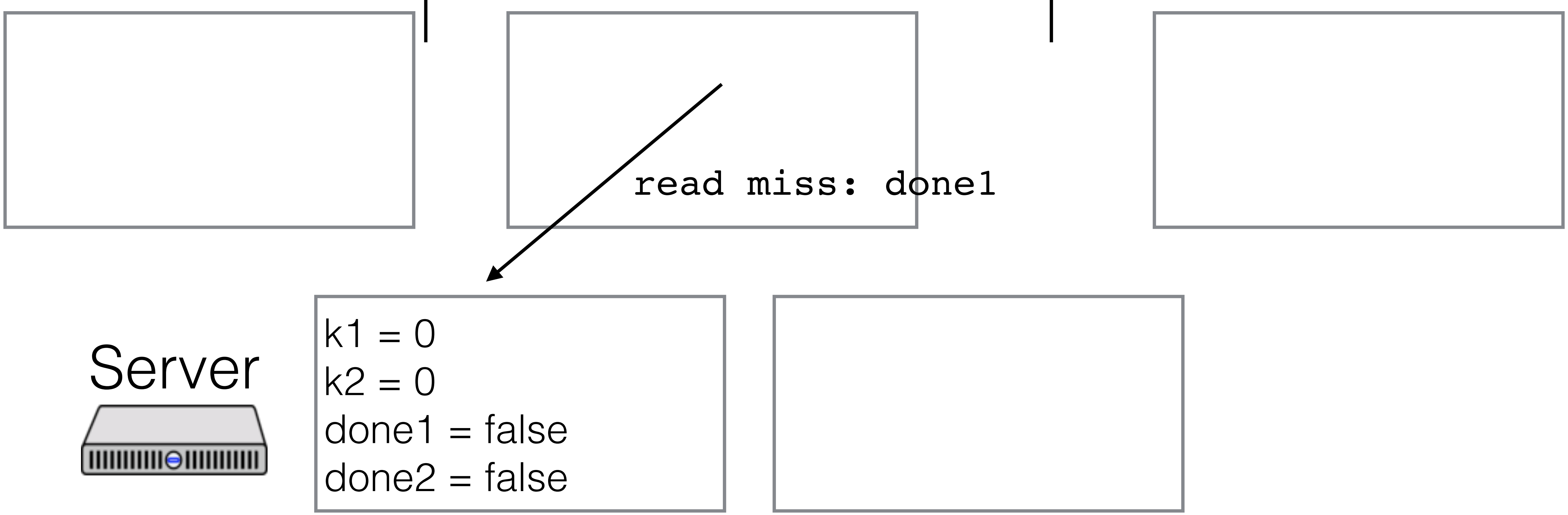


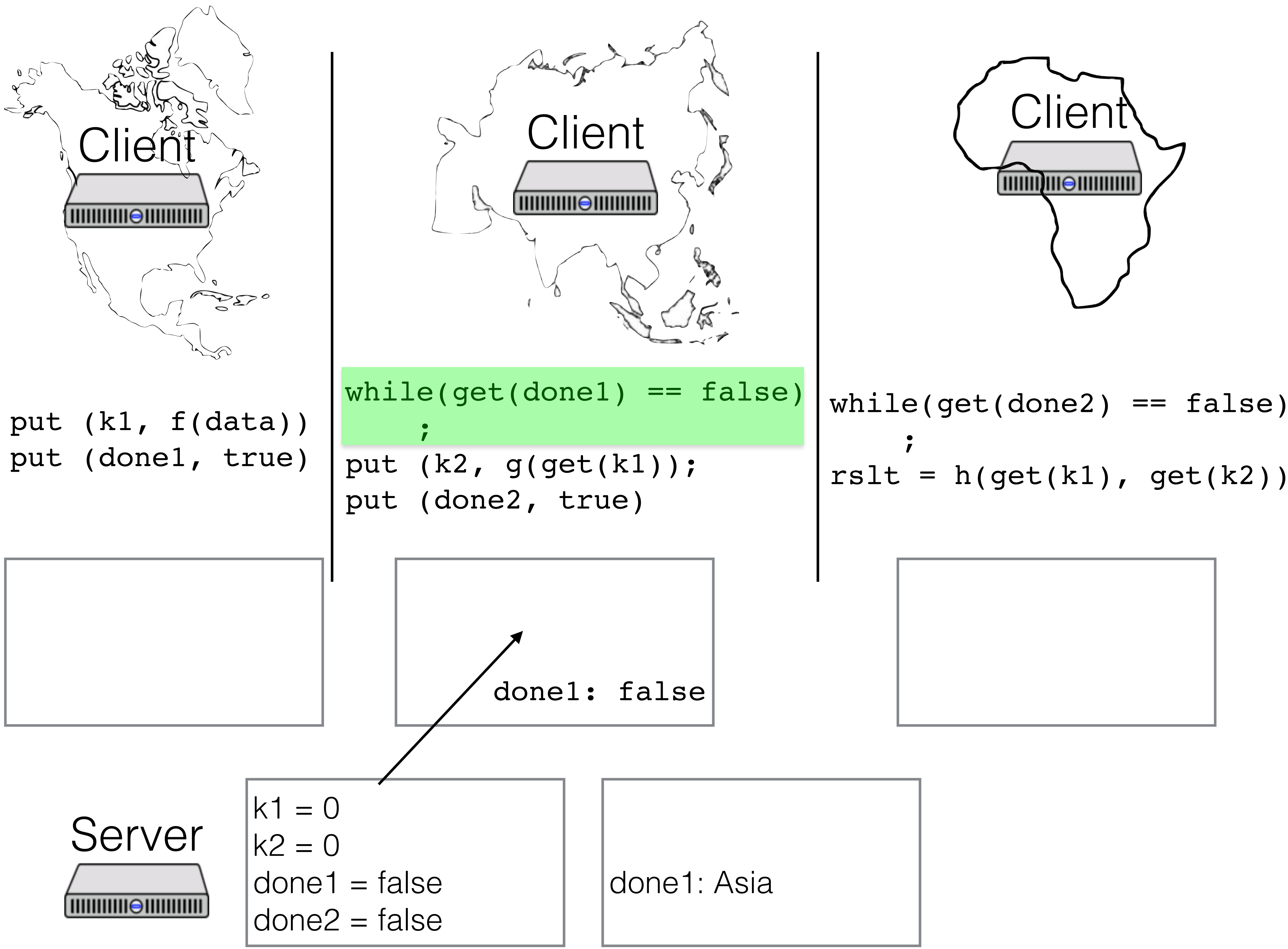

```
put (k1, f(data))
put (done1, true)
```

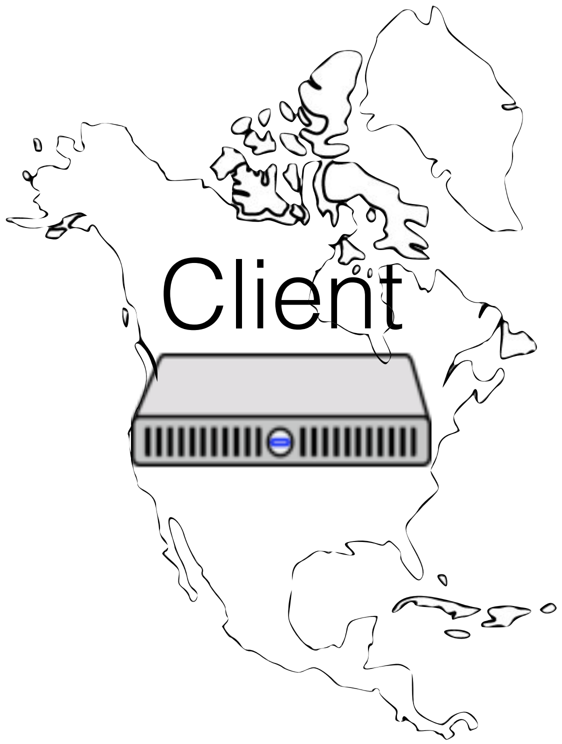

```
while(get(done1) == false)
    ;
put (k2, g(get(k1));
put (done2, true)
```




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

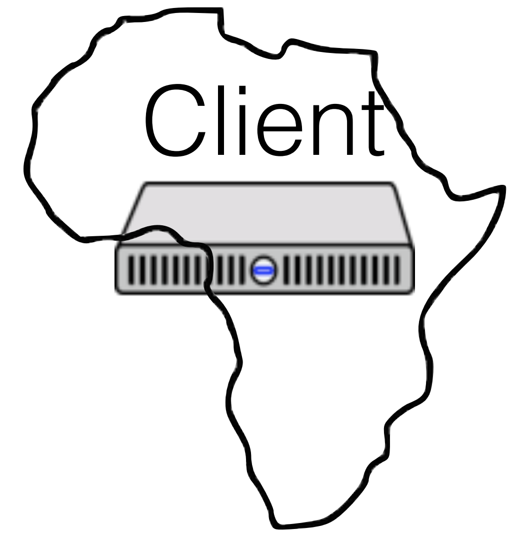



Client
Client
Client
put (k1, f(data))
put (done1, true)
while(get(done1) == false)
;
put (k2, g(get(k1));
put (done2, true)
while(get(done2) == false)
;
rslt = h(get(k1), get(k2))
done1: false
Server
k1 = 0
k2 = 0
done1 = false
done2 = false
done1: Asia




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


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



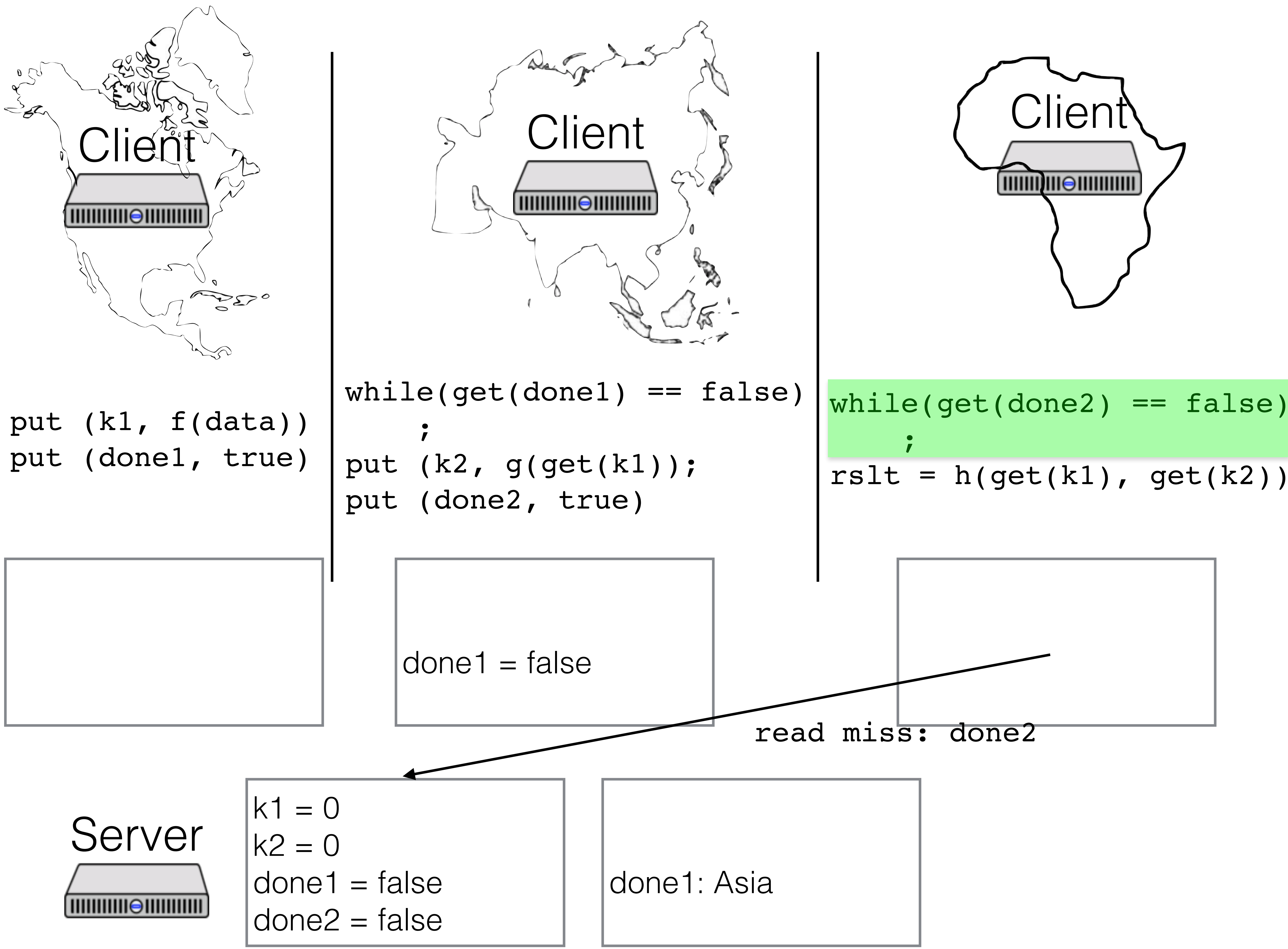

```
while(get(done2) == false)
    ;
rslt = h(get(k1), get(k2))
```

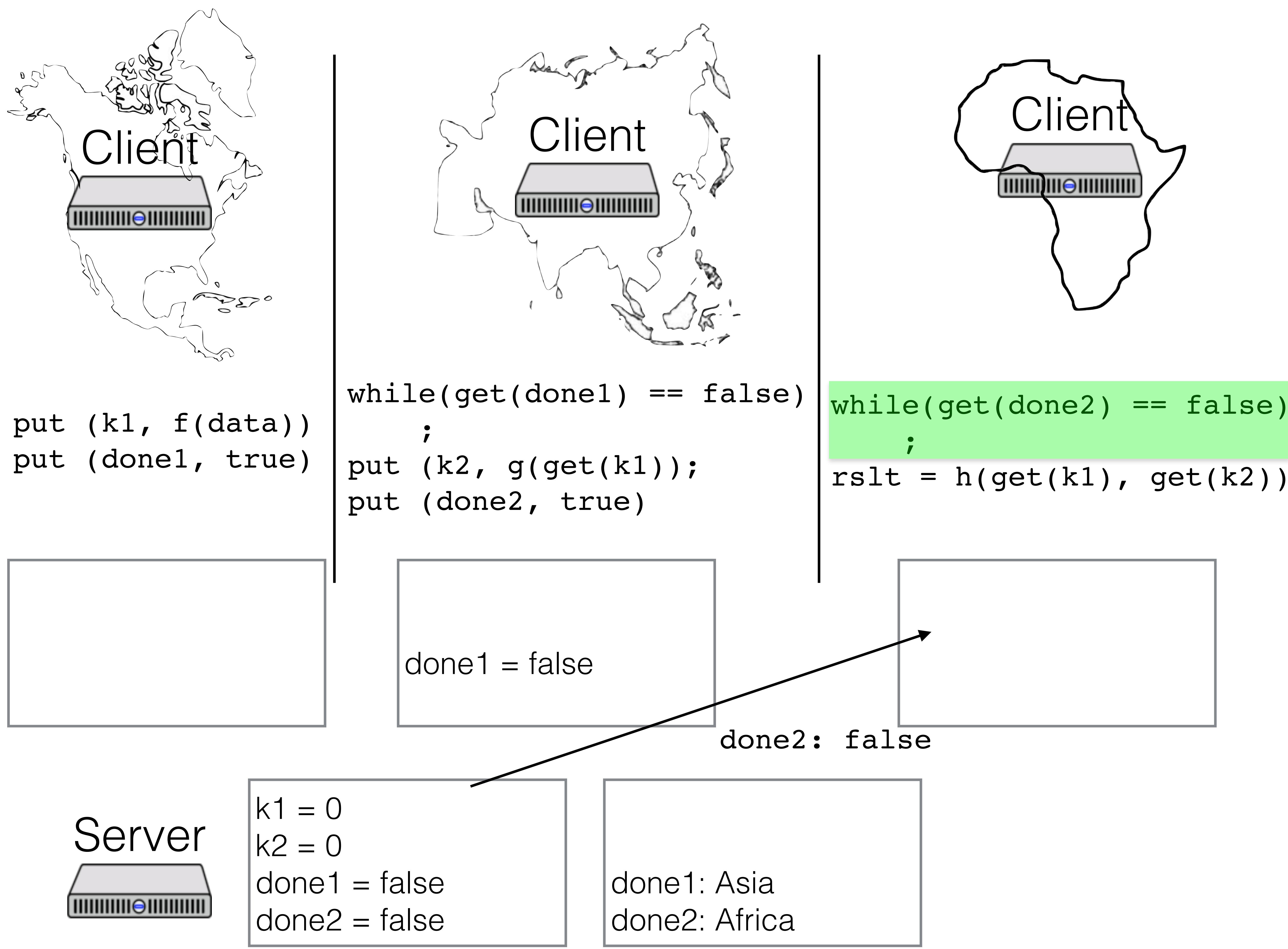
done1 = false

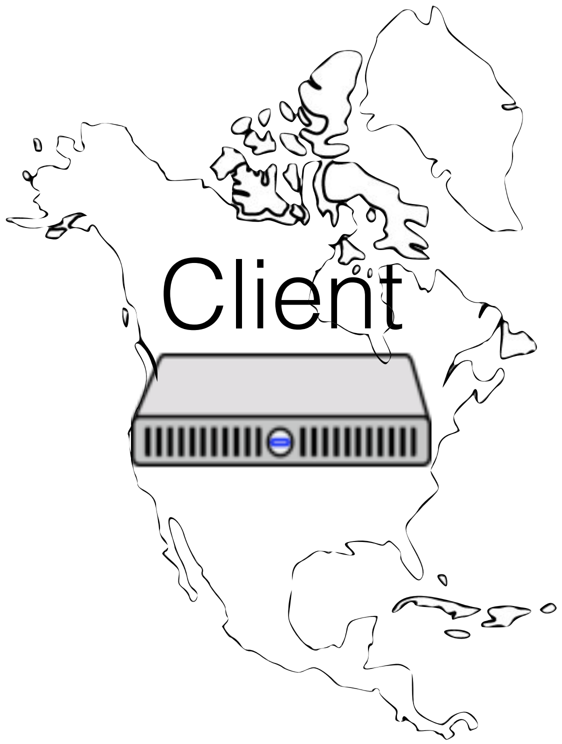
Server

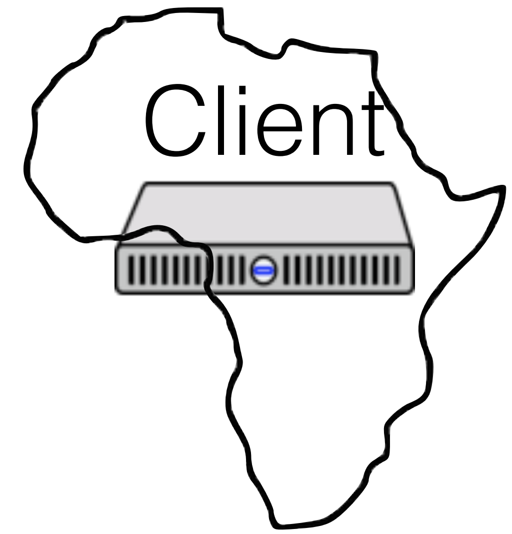
k1 = 0
k2 = 0
done1 = false
done2 = false

done1: Asia


Client
Client
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))
done1 = false
read miss: done2
Server
k1 = 0
k2 = 0
done1 = false
done2 = false
done1: Asia


Client
Client
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))
done1 = false
done2: false
Server
k1 = 0
k2 = 0
done1 = false
done2 = false
done1: Asia
done2: Africa





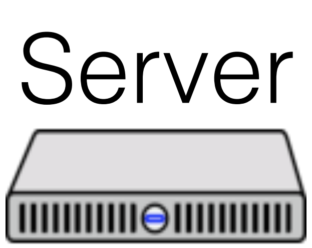

```
put (k1, f(data))
put (done1, true)
```

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

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

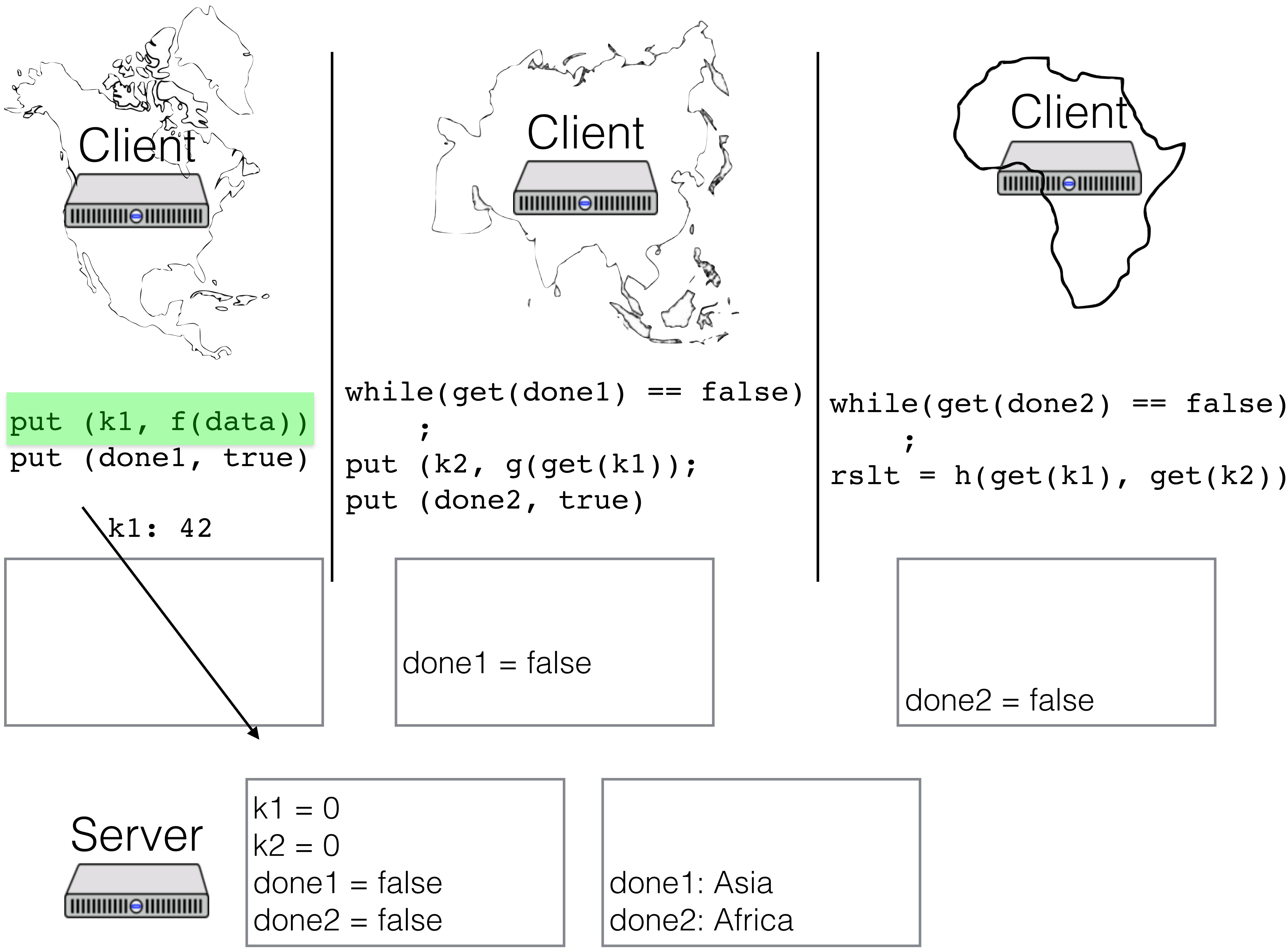
done1 = false

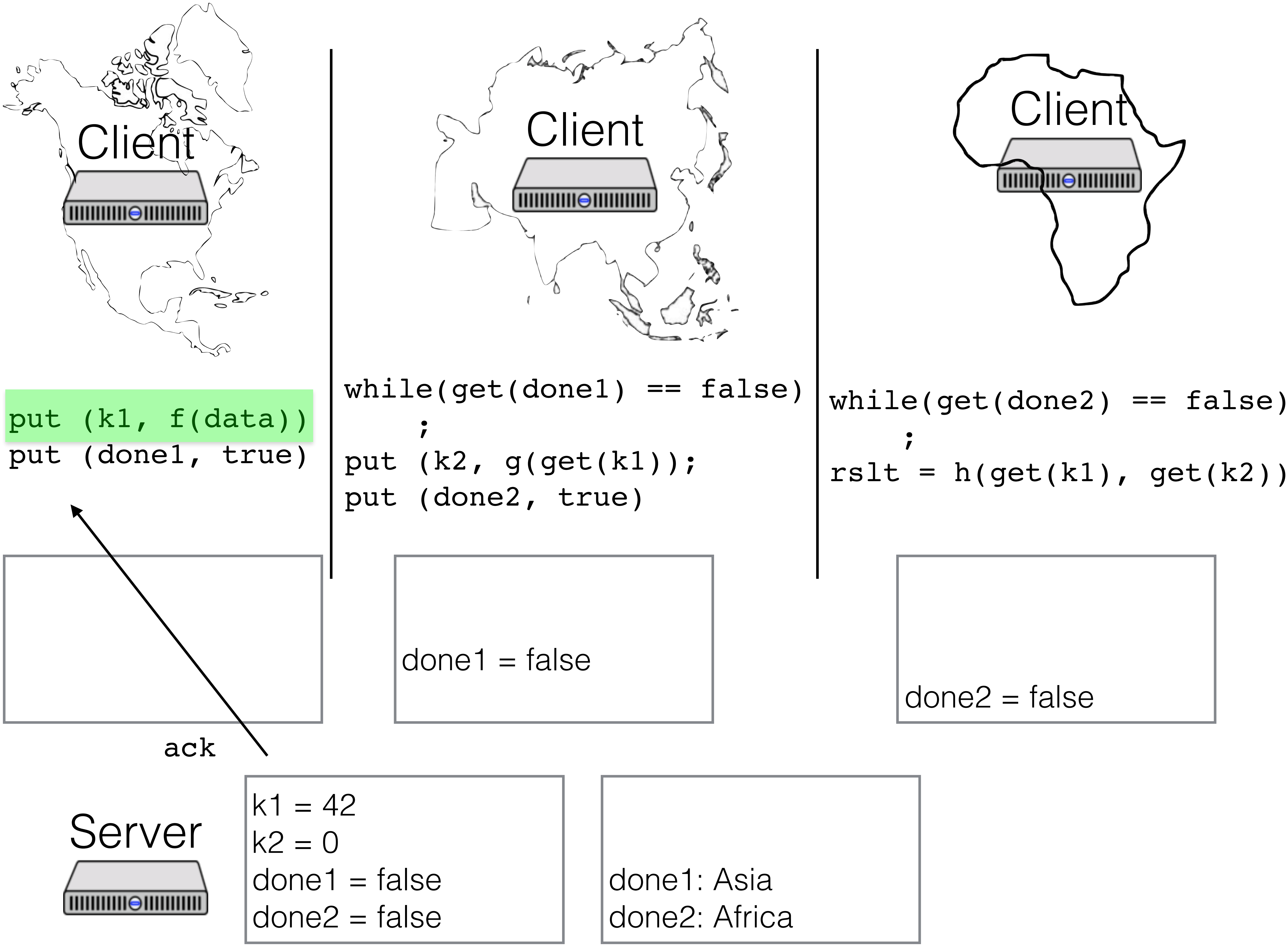
done2 = false

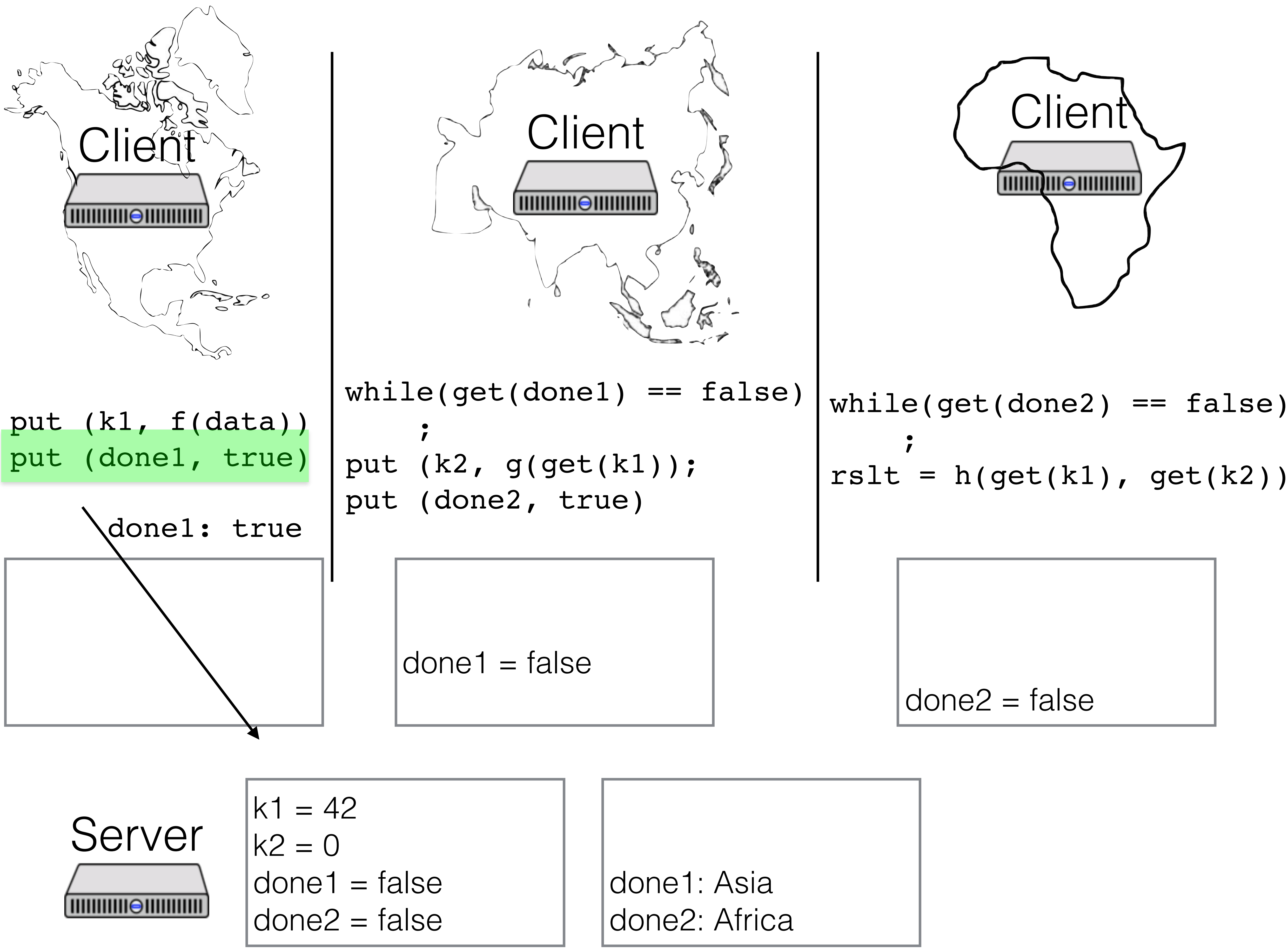


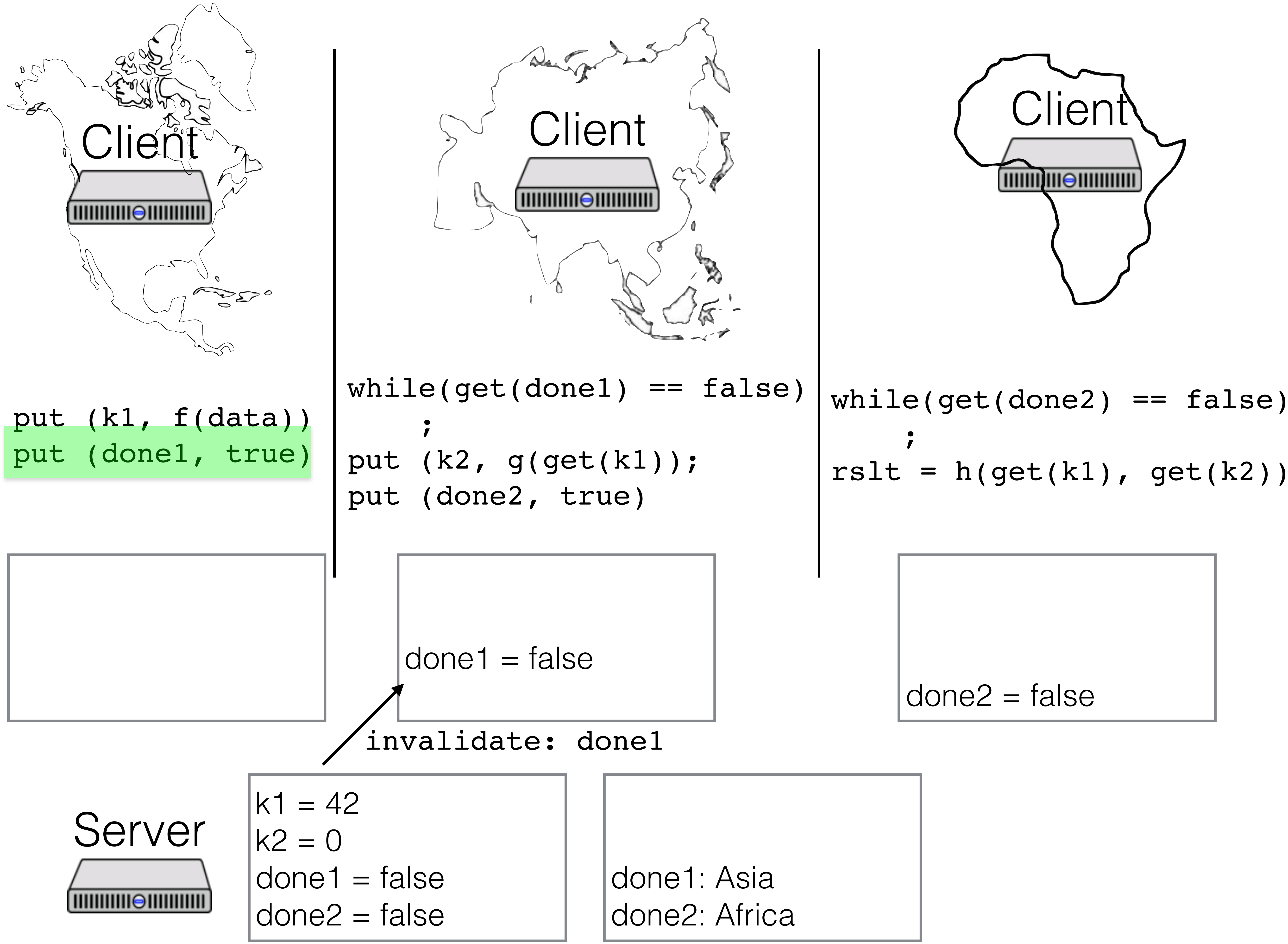
k1 = 0
k2 = 0
done1 = false
done2 = false

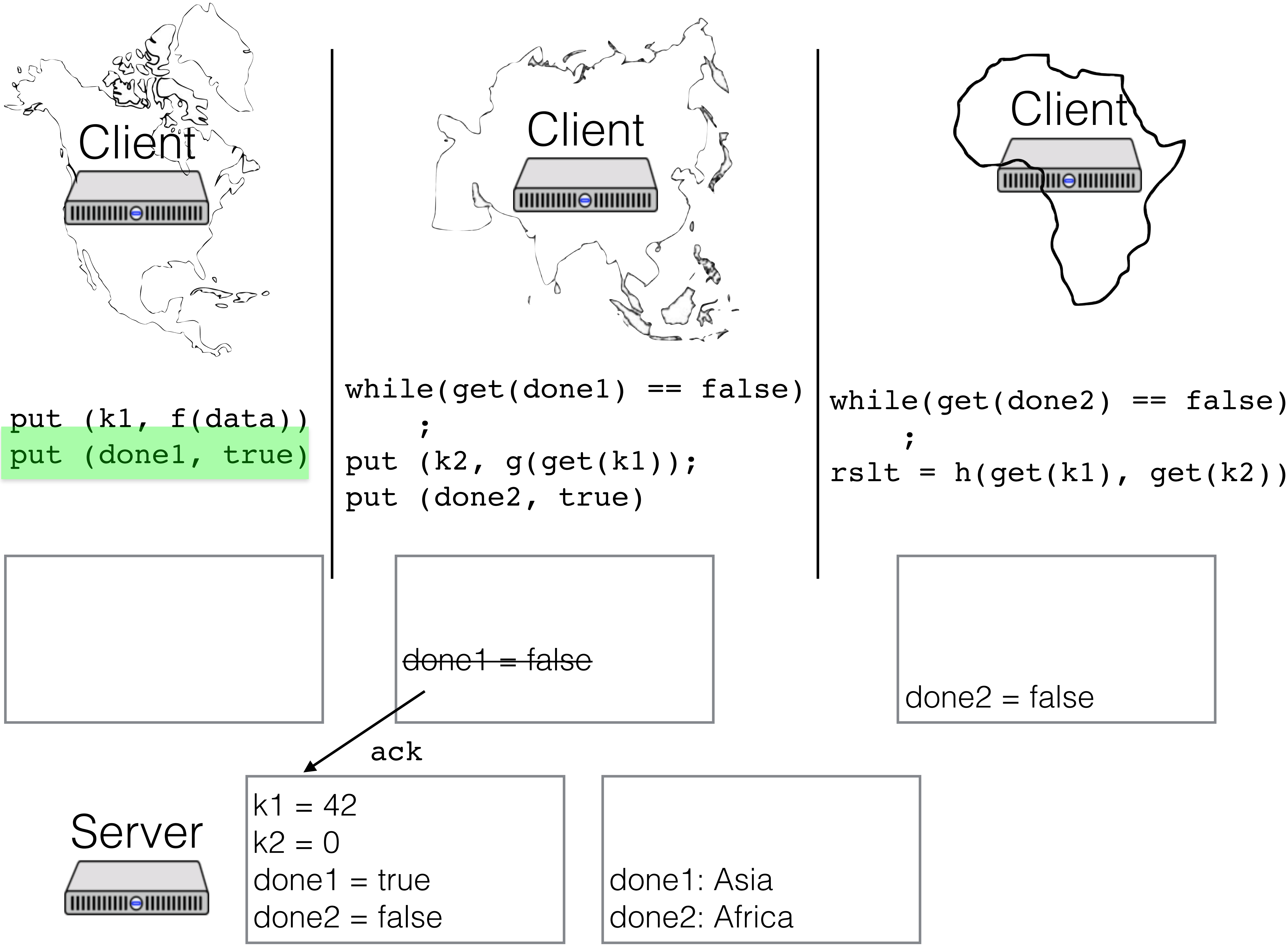
done1: Asia
done2: Africa

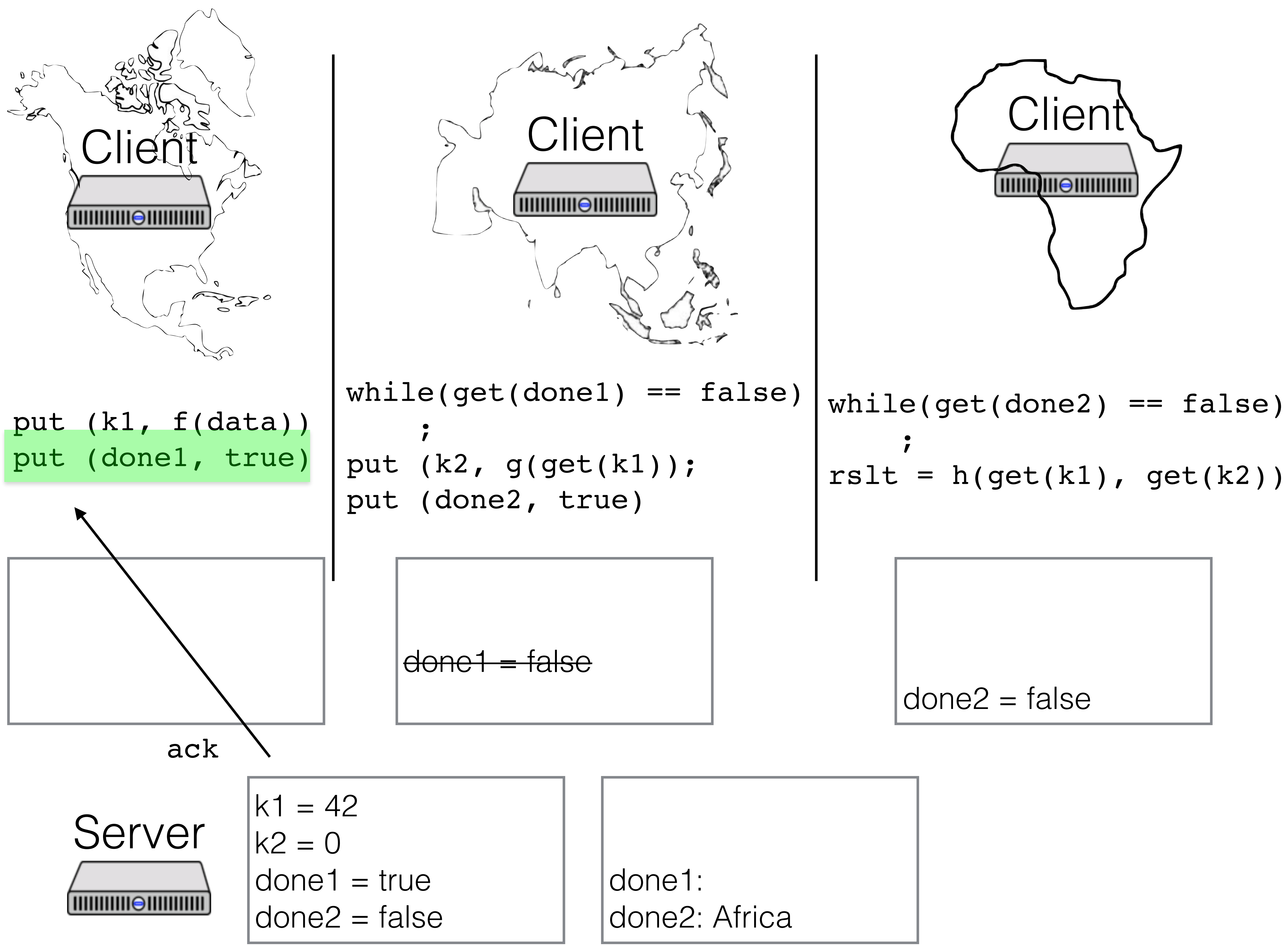

Client
Client
Client
put (k1, f(data))
put (done1, true)
while(get(done1) == false)
;
put (k2, g(get(k1));
put (done2, true)
while(get(done2) == false)
;
rslt = h(get(k1), get(k2))
k1: 42
done1 = false
done2 = false
Server
k1 = 0
k2 = 0
done1 = false
done2 = false
done1: Asia
done2: Africa

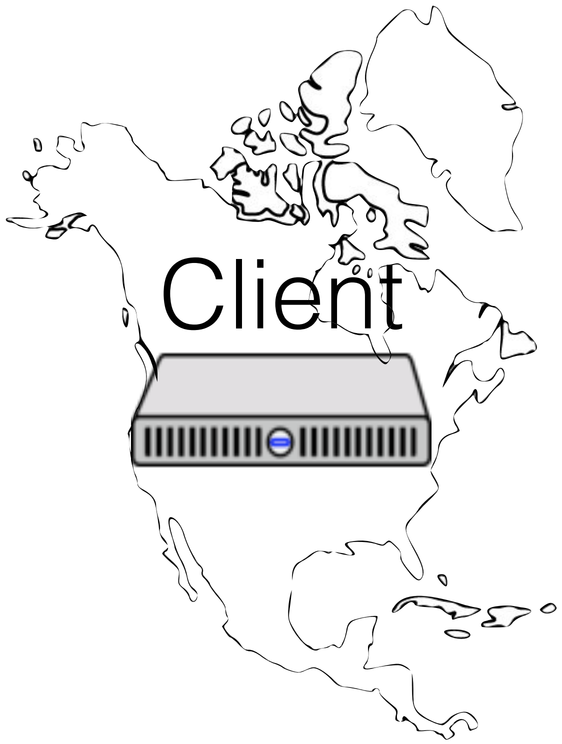

Client
Client
Client
put (k1, f(data))
put (done1, true)
while(get(done1) == false)
;
put (k2, g(get(k1));
put (done2, true)
while(get(done2) == false)
;
rslt = h(get(k1), get(k2))
done1 = false
done2 = false
ack
Server
k1 = 42
k2 = 0
done1 = false
done2 = false
done1: Asia
done2: Africa

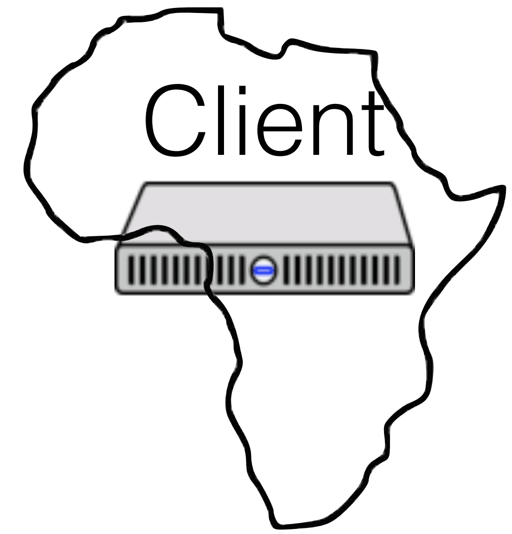

Client
Client
Client
put (k1, f(data))
put (done1, true)
done1: true
while(get(done1) == false)
;
put (k2, g(get(k1));
put (done2, true)
while(get(done2) == false)
;
rslt = h(get(k1), get(k2))
done1 = false
done2 = false
Server
k1 = 42
k2 = 0
done1 = false
done2 = false
done1: Asia
done2: Africa


Client
Client
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))
done1 = false
done2 = false
invalidate: done1
Server
k1 = 42
k2 = 0
done1 = false
done2 = false
done1: Asia
done2: Africa


Client
Client
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))
done1 = false
done2 = false
ack
Server
k1 = 42
k2 = 0
done1 = true
done2 = false
done1: Asia
done2: Africa


Client
Client
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))
done1 = false
done2 = false
ack
Server
k1 = 42
k2 = 0
done1 = true
done2 = false
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))
```

done1 = true

done2 = false

Server

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

done1: Asia
done2: Africa

Questions

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

Questions

While a write to key k from client C is waiting on invalidations, can C perform another write to a different key m?

Questions

While a write to key k from client C is waiting on invalidations, can the server perform a read from a different client D to a different key m?

Questions

While a write to key k from client C is waiting on invalidations, can the server perform a read to k from a different client D?

Questions

While a write to key k from client C is waiting on invalidations, can the server perform a write from client D to the same key?

Facebook’s Memcache Service

Facebook's Scaling Problem

- Rapidly increasing user base
  - Small initial user base
  - 2x every 9 months
  - 2013: 1B users globally
- Users read/update many times per day
  - Increasingly intensive app logic per user
  - 2x I/O every 4-6 months
- Infrastructure has to keep pace

Scaling Strategy

Adapt off the shelf components where possible

Fix as you go

- no overarching plan

Rule of thumb: Every order of magnitude requires a rethink

Facebook Three Layer Architecture

- Application front end
  - Stateless, rapidly changing program logic
  - If app server fails, redirect client to new app server
- Memcache
  - Lookaside key-value cache
  - Keys defined by app logic (can be computed results)
- Fault tolerant storage backend
  - Stateful
  - Careful engineering to provide safety and performance
  - Both SQL and NoSQL

Workload

- Each user's page is unique
  - draws on events posted by other users
- Users not in cliques
  - For the most part
- User popularity is zipf
  - Some user posts affect very large #'s of other pages
  - Most affect a much smaller number

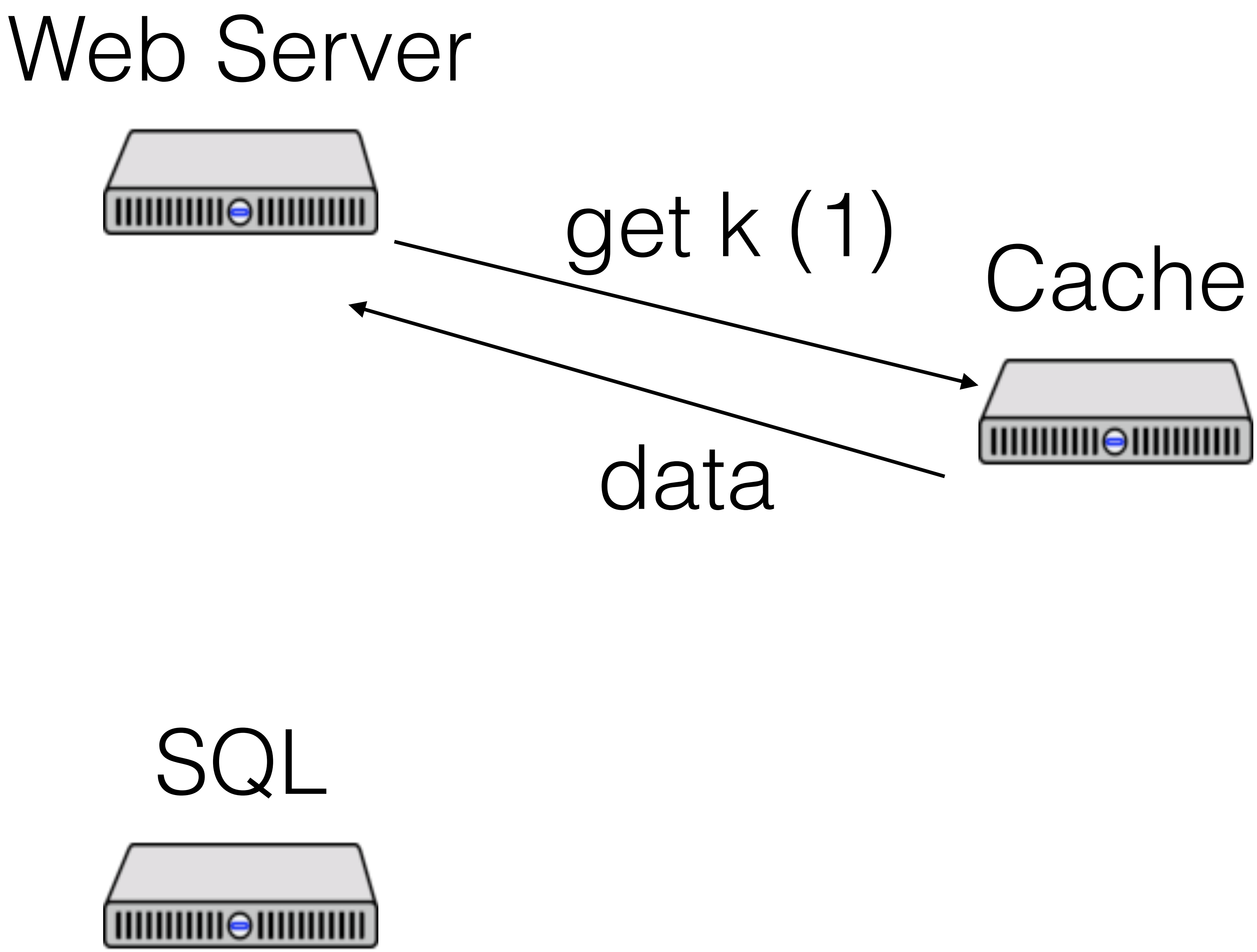
Scale By Caching: Memcache

Sharded in-memory key-value cache

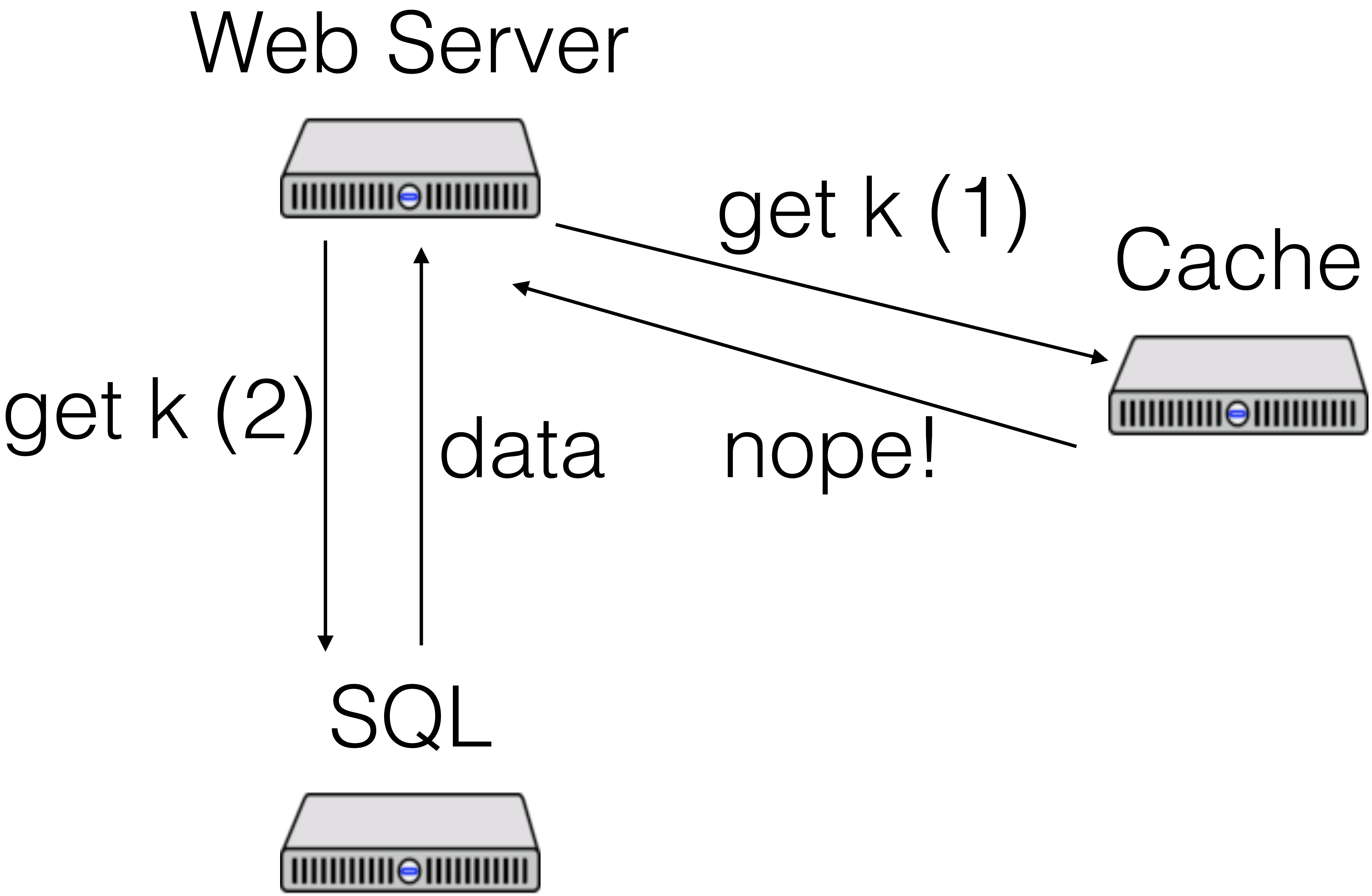
- Key, values assigned by application code
- Values can be data, result of computation
- Independent of backend storage architecture (SQL, noSQL) or format
- Design for high volume, low latency

Lookaside architecture

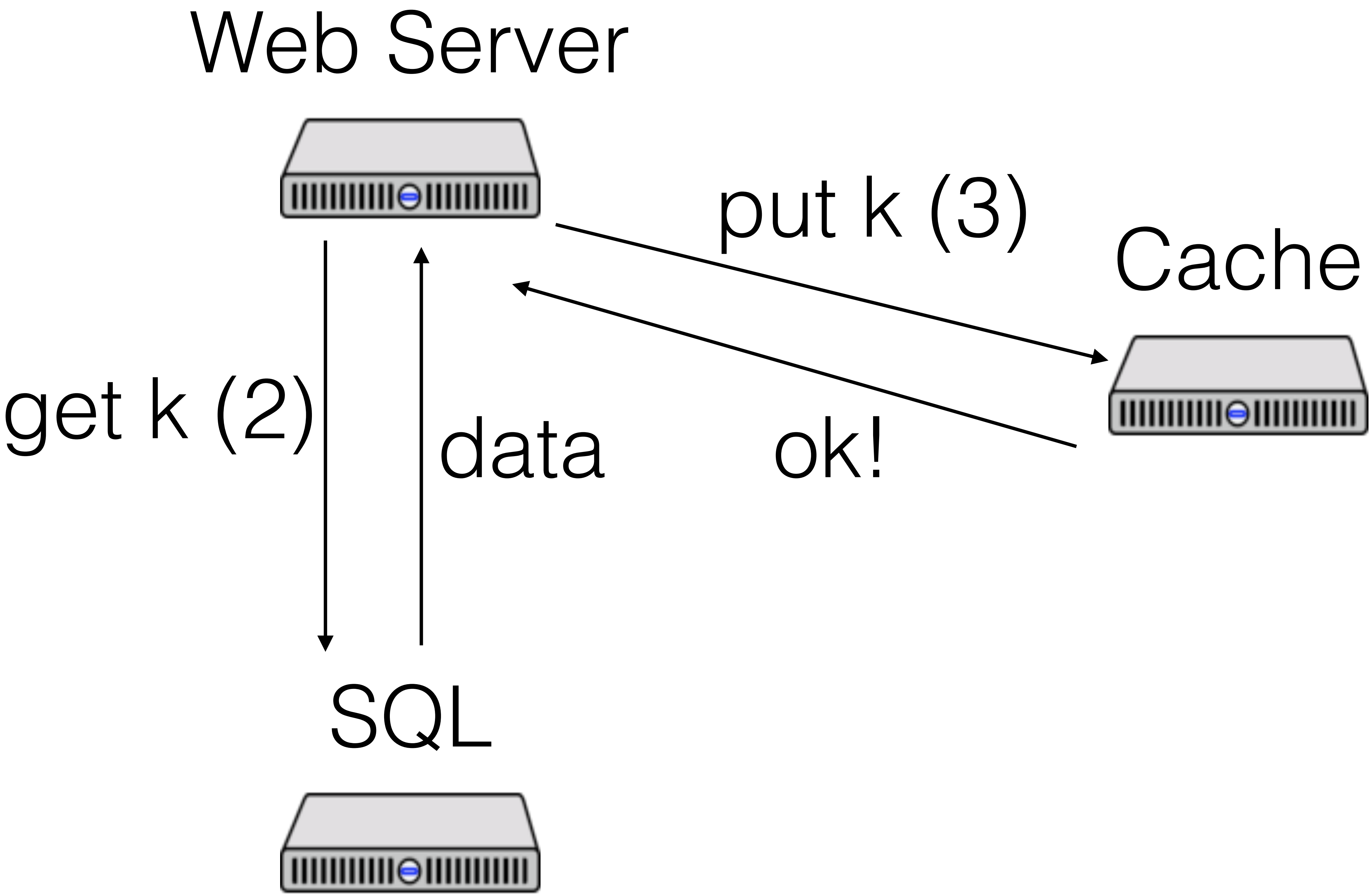
Lookaside Read



Lookaside Read



Lookaside Read



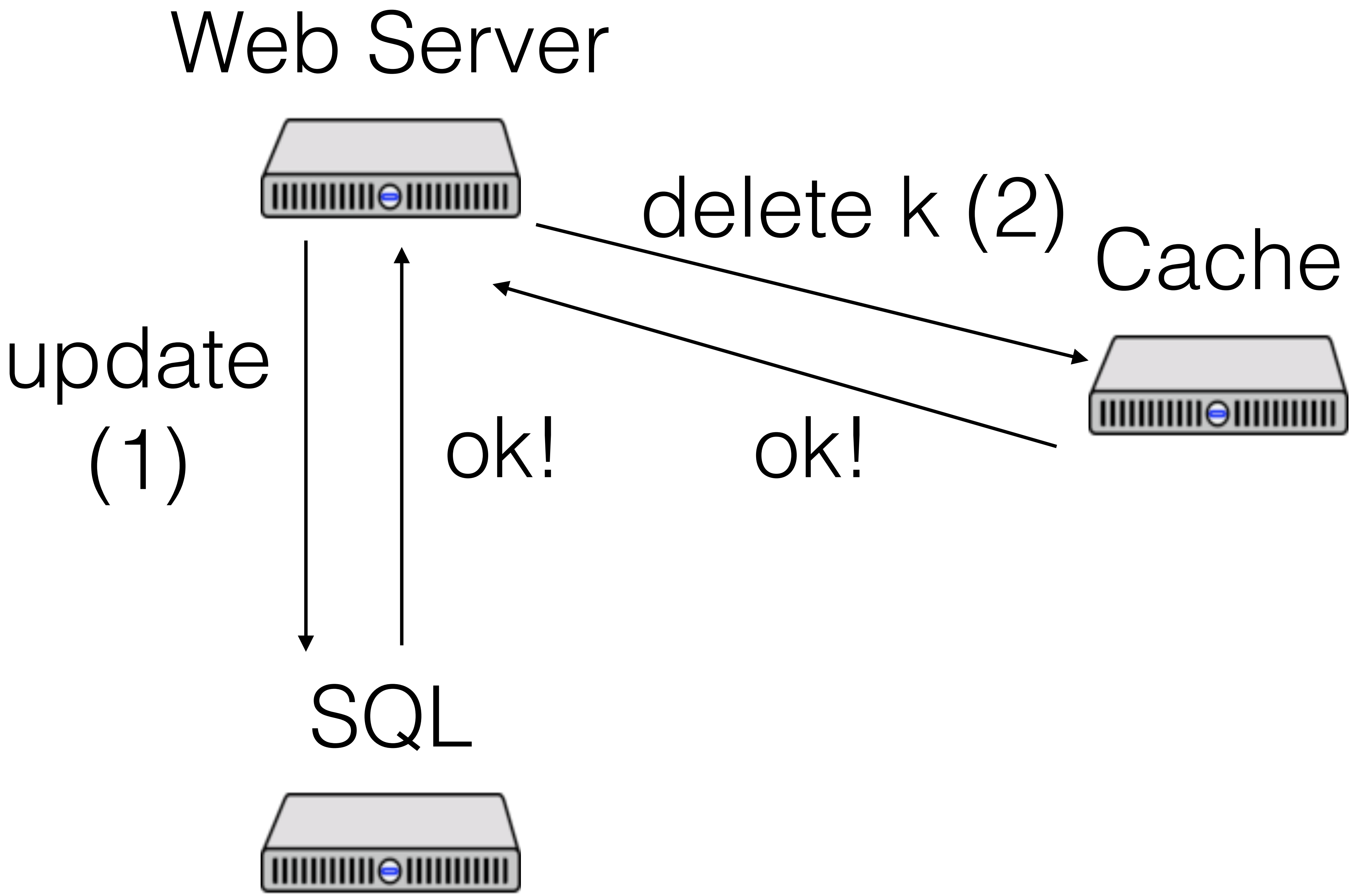
Lookaside Operation (Read)

- Webserver needs key value
- Webserver requests from memcache
- Memcache: If in cache, return it
- If not in cache:
  - Return error
  - Webserver gets data from storage server
  - Possibly an SQL query or complex computation
  - Webserver stores result back into memcache

Question

What if swarm of users read same key at the same time?

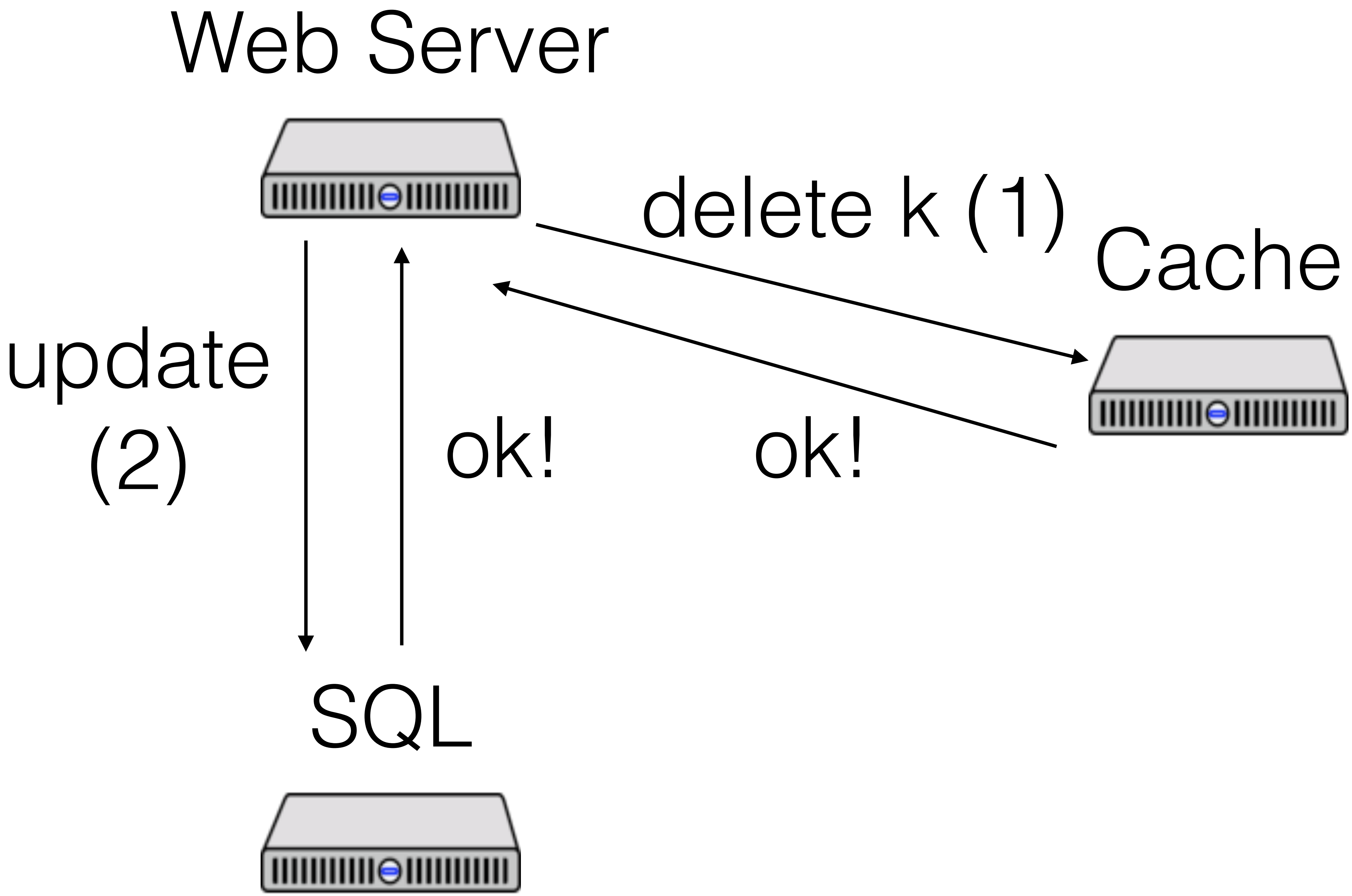
Lookaside Write



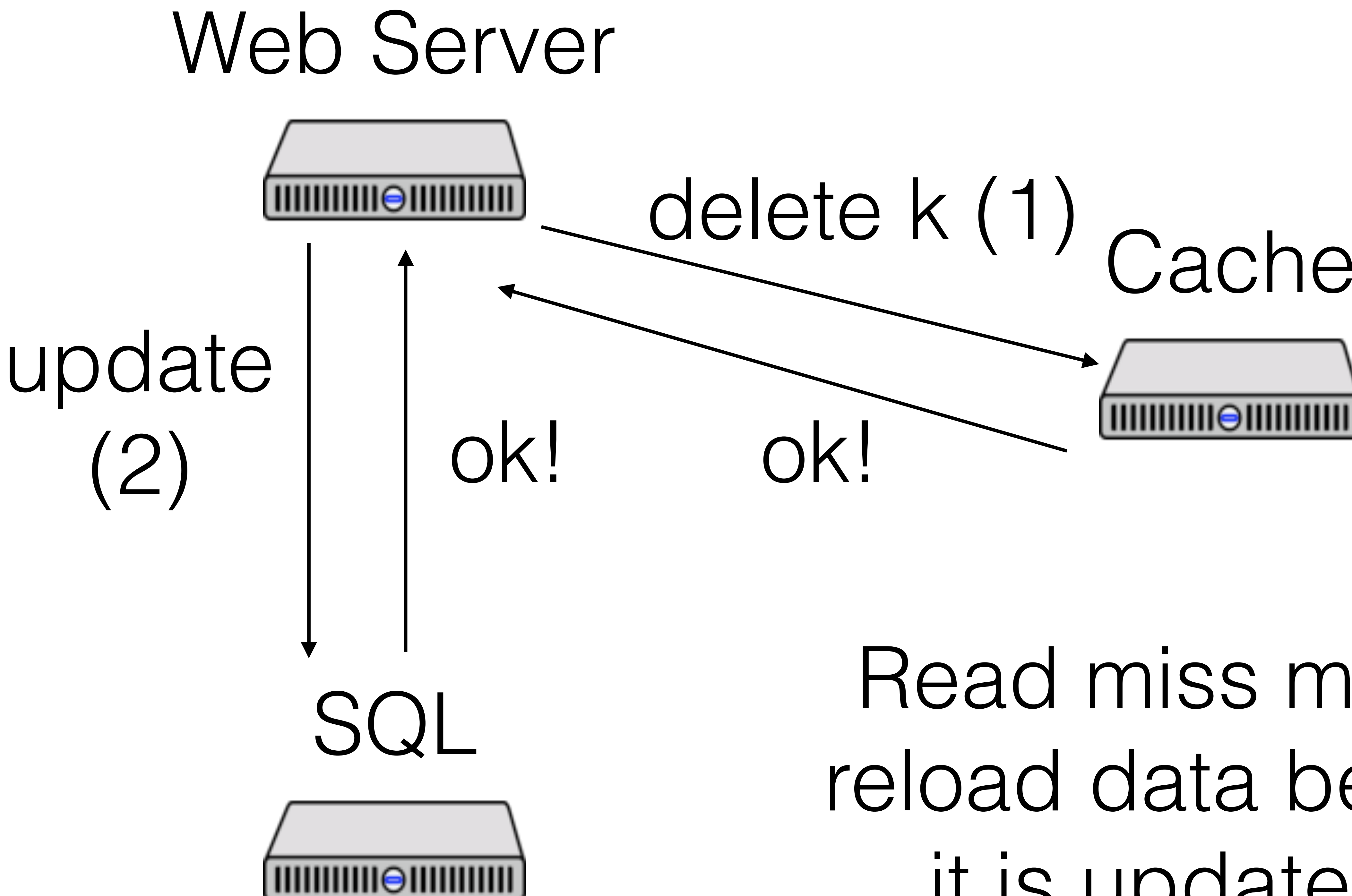
Lookaside Operation (Write)

- Webserver changes a value that would invalidate a memcache entry
  - Could be an update to a key
  - Could be an update to a value used to derive some key value
- Client puts new data on storage server
- Client invalidates entry in memcache

Why Not Delete then Update?



Why Not Delete then Update?



Read miss might reload data before it is updated.

Memcache Consistency

Is memcache linearizable?

Example

Webserver: Reader

Read cache

If missing,

  Fetch from database

  Store back to cache

Webserver: Writer

Change database

Delete cache entry

Interleave any # of readers/writers

Example

Webserver: Reader

Read cache

Webserver: Writer

Change database

Delete cache entry

Memcache Consistency

Is the lookaside protocol eventually consistent?

Example

- Read cache
- Read database

- change database
- Delete entry

- Store back to cache

Lookaside With Leases

Goals:

- Reduce (eliminate?) per-key inconsistencies
- Reduce cache miss swarms

On a read miss:

- leave a marker in the cache (fetch in progress)
- return timestamp
- check timestamp when filling the cache
- if changed means value has (likely) changed: don't overwrite

If another thread read misses:

- find marker and wait for update (retry later)

Question

What if web server crashes while holding lease?

Question

Is lookaside with leases linearizable?

Example

Webserver: Reader

Read cache

Webserver: Writer

Change database

Delete cache entry

Question

Is lookaside with leases eventually consistent?

Example

Webserver: Reader

Webserver: Writer

Change database

Read cache

CRASH!

(before Delete cache entry)

Question

Would this be made "more correct"?

- read misses obtain lease
- writes obtain lease (prevent reads during update)

Except that

- FB replicates popular keys (need lease on every copy?)
- memcache server might fail, or appear to fail by being slow (e.g., to some nodes, but not others)

Latency Optimizations

Concurrent lookups

- Issue many lookups concurrently
- Prioritize those that have chained dependencies

Batching

- Batch multiple requests (e.g., for different end users) to the same memcache server

Incast control:

- Limit concurrency to avoid collisions among RPC responses

More Optimizations

Return stale data to web server if lease is held

- No guarantee that concurrent requests returning stale data will be consistent with each other

Partitioned memory pools

- Infrequently accessed, expensive to recompute
- Frequently accessed, cheap to recompute
- If mixed, frequent accesses will evict all others

Replicate keys if access rate is too high

- Implication for consistency?

Gutter Cache

When a memcache server fails, flood of requests to fetch data from storage layer

- Slows users needing any key on failed server
- Slows other users due to storage server contention

Solution: backup (gutter) cache

- Time-to-live invalidation (ok if clients disagree as to whether memcache server is still alive)
- TTL is eventually consistent