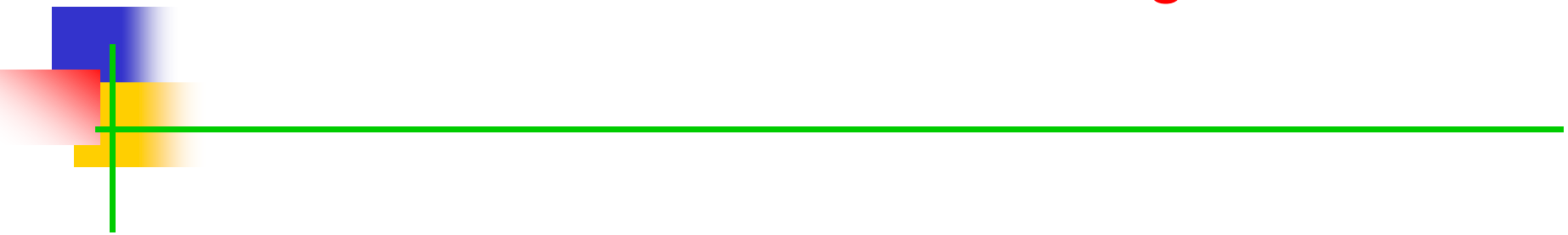
Distributed Caching



Introduction

- Caching is essential in a distributed system.
- Makes the results of expensive queries and computations available for reuse.
- Caches exist in many places in an application.
- Two types of caching will be covered:
  - Application caching
  - Web caching

Application Caching

- Application caching is designed to improve request responsiveness by storing the results of queries and computations in memory so they can be subsequently served by later requests.
  - Newspapers articles (how?)
  - Concert seating map
  - Hourly weather forecast.

Application Caching

- Caching benefits?
  - Heavy read traffic reduction
  - Computation costs reduction
- Caching requires additional resources, and hence cost, to store cached results.
- Well designed caching schemes are low-cost compared to upgrading database and service nodes to cope with higher request loads.
- 3% of infrastructure at Twitter is dedicated to application-level caches.

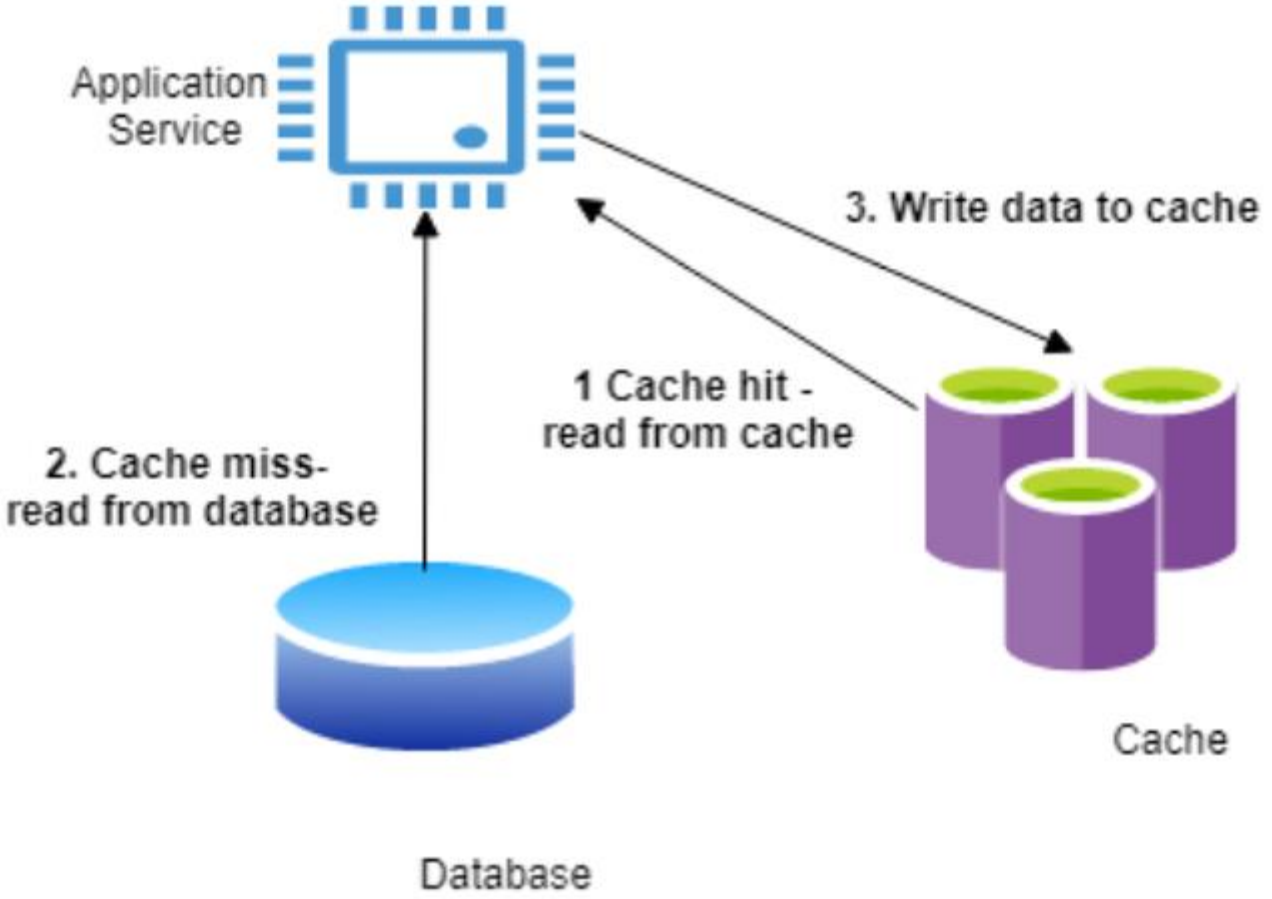
Application Caching

- Utilize cache engines. The two predominant technologies in this area are:
  - Memcached (Supported by Netflix)
  - Redis
- Such cache engines are distributed in-memory hash tables designed for arbitrary data (strings, objects) representing the results of database queries or downstream service API calls.
- Possible storage?
  - User session data, database queries

Application Caching

- The cache appears to application services as a single store, and objects are allocated to individual cache servers using a hash function on the object key

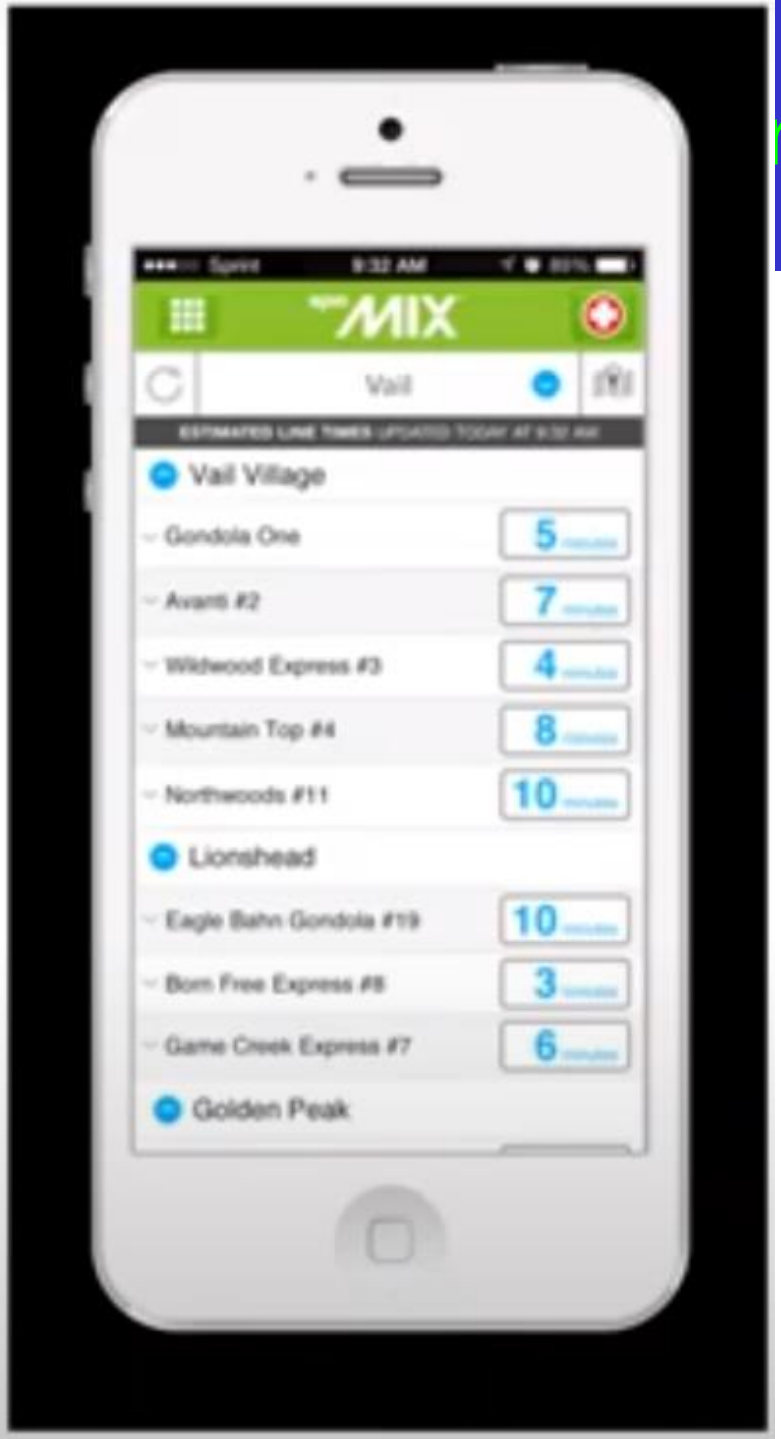
Application Caching



Application Caching - **Example**

- At a busy winter resort, skiers and boarders can use their mobile app to get an estimate of the lift wait times across the resort.
- This enables them to plan and avoid congested areas where they will have to wait to ride a lift for say 15 minutes (or sometimes more!).
- How would the company calculate the estimated wait time to get a lift?
- Is such calculation expensive? Why?
- What needs to be cached?

Appli mple


Sprint
9:32 AM
MIX
Vail
Vail Village
Gondola One
5
Avanti #2
7
Wildwood Express #3
4
Mountain Top #4
8
Northwoods #11
10
Lionshead
Eagle Bahn Gondola #19
10
Born Free Express #8
3
Game Creek Express #7
6
Golden Peak

Application Caching - **Example**

```
1.  public class LiftWaitService {
2.
3.    public List getLiftWaits(String resort) {
4.      List liftWaitTimes = cache.get("liftwaittimes:" + resort);
5.       if (liftWaitTimes == null) {
6.         liftWaitTimes = skiCo.getLiftWaitTimes(resort);
7.         // add result to cache, expire in 300 seconds
8.         cache.put("liftwaittimes:" + resort, liftWaitTimes, 300);
9.       }
10.     return liftWaitTimes;
11.   }
12. }
```

TimeToLive (TTL)

Application Caching - **Example**

- A cache hit on a fast network will take maybe a millisecond – much faster than the lift wait times calculation.
- Hence if we get N requests in a 5 minute period, N-1 requests are served from the cache.
  Imagine if N is 10000?
- This is a lot of expensive calculations saved, and CPU cycles that your database can use to process other queries.

Application Caching Design

- Application caching can provide significant throughput boosts, reduced latencies, and increased client application responsiveness.
- The key to achieving these desirable qualities is to satisfy as many requests as possible from the cache.
- The general design principle is to maximize the cache hit rate and minimize the cache miss rate.
- When items are updated regularly, the cost of cache misses can negate the benefits of the cache

Application Caching Design

- It is needed to monitor the cache usage once a service is in production to ensure the hit and miss rates are in line with design expectations.
- Caches will provide both management utilities and APIs to enable monitoring of the cache usage characteristics.
- For example, memcached makes a large number of statistics available, including the hit and miss counts

```
STAT get_hits 98567
STAT get_misses 11001
```

Caching Patterns

- Application level caching known as cache-aside pattern (**Why?**)
- Alternatives exist, commonly supported by databases
  - Read-through
  - Write-through
  - Write-behind

Caching Patterns

- Read-through: The application satisfies all requests by accessing the cache. If the data required is not available in the cache, a loader is invoked to access the data systems and load the results in the cache for the application to utilize
- Write-through: The application always writes updates to the cache. When the cache is updated, a writer is invoked to write the new cache values to the database. **When the database is updated**, the application can complete the request.
- Write-behind: Like write-through, except the application does not wait for the value to be written to the database from the cache. (Pros/Cons)

Cache-aside versus (Read-through, Write-through, Write-behind)?

- Read/Write-through/behind simplify application logic (how?)
  - Require a cache augmented with an application specific handler to preform database reads/writes (e.g., Amazon's DynamoDB Accelerator (DAX))
- DAX: AWS launched Amazon DynamoDB Accelerator (**DAX**), a highly available, in-memory cache for Amazon DynamoDB.

Cache-aside versus (Read-through, Write-through, Write-behind)?

- How does DAX work?
  - The application developer points their application at the DAX endpoint instead of at the DynamoDB endpoint
  - DAX seamlessly intercepts the API calls that an application normally makes to DynamoDB so that both read and write activity are reflected in the DAX cache.
  - Cache failure?

Caching Patterns

- For cache-aside:
  - More complex programming model
  - Resilient to cache failure

Web Caching

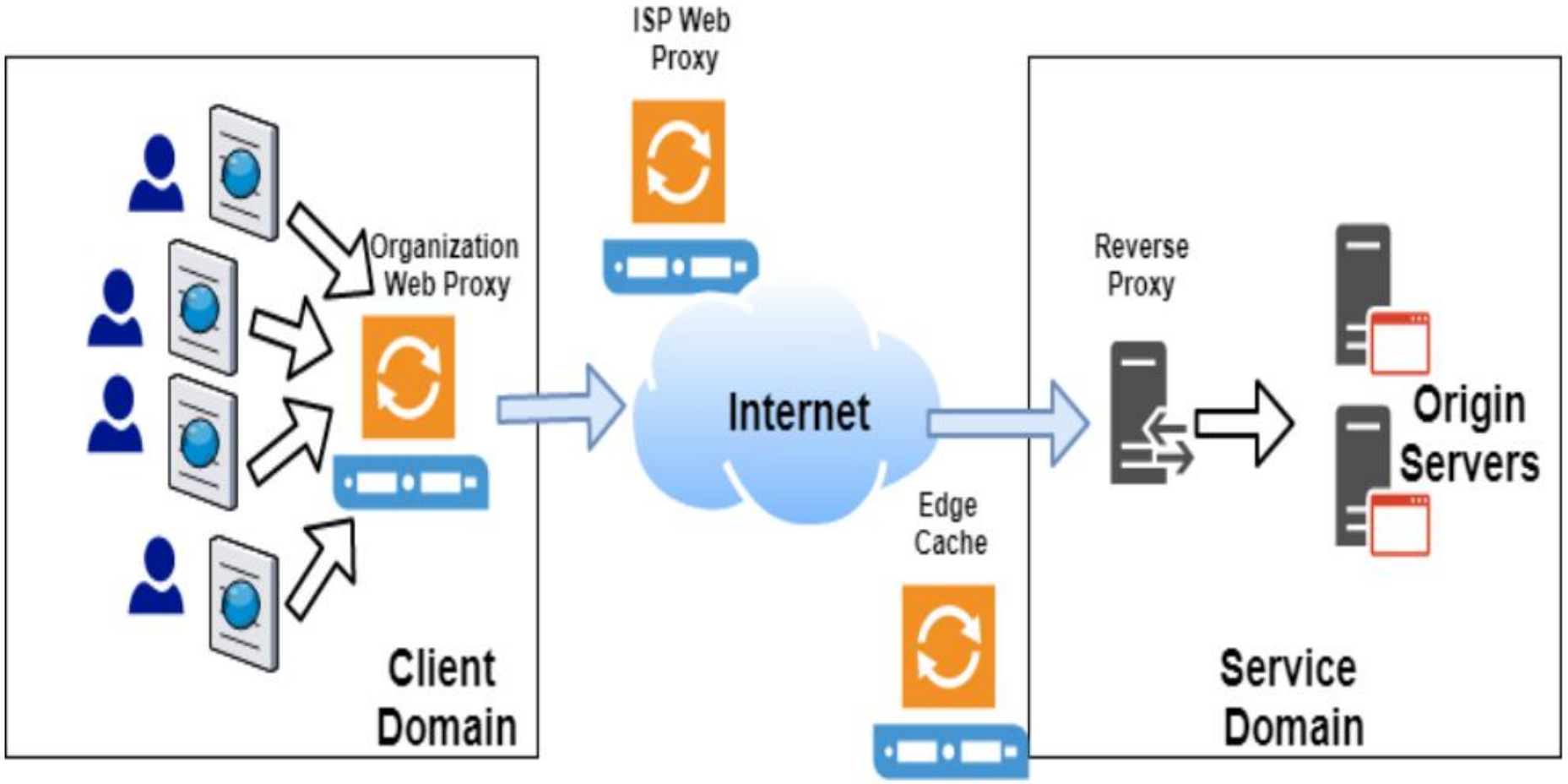


Figure 6-2. Web Caches in the Internet

Web Caching

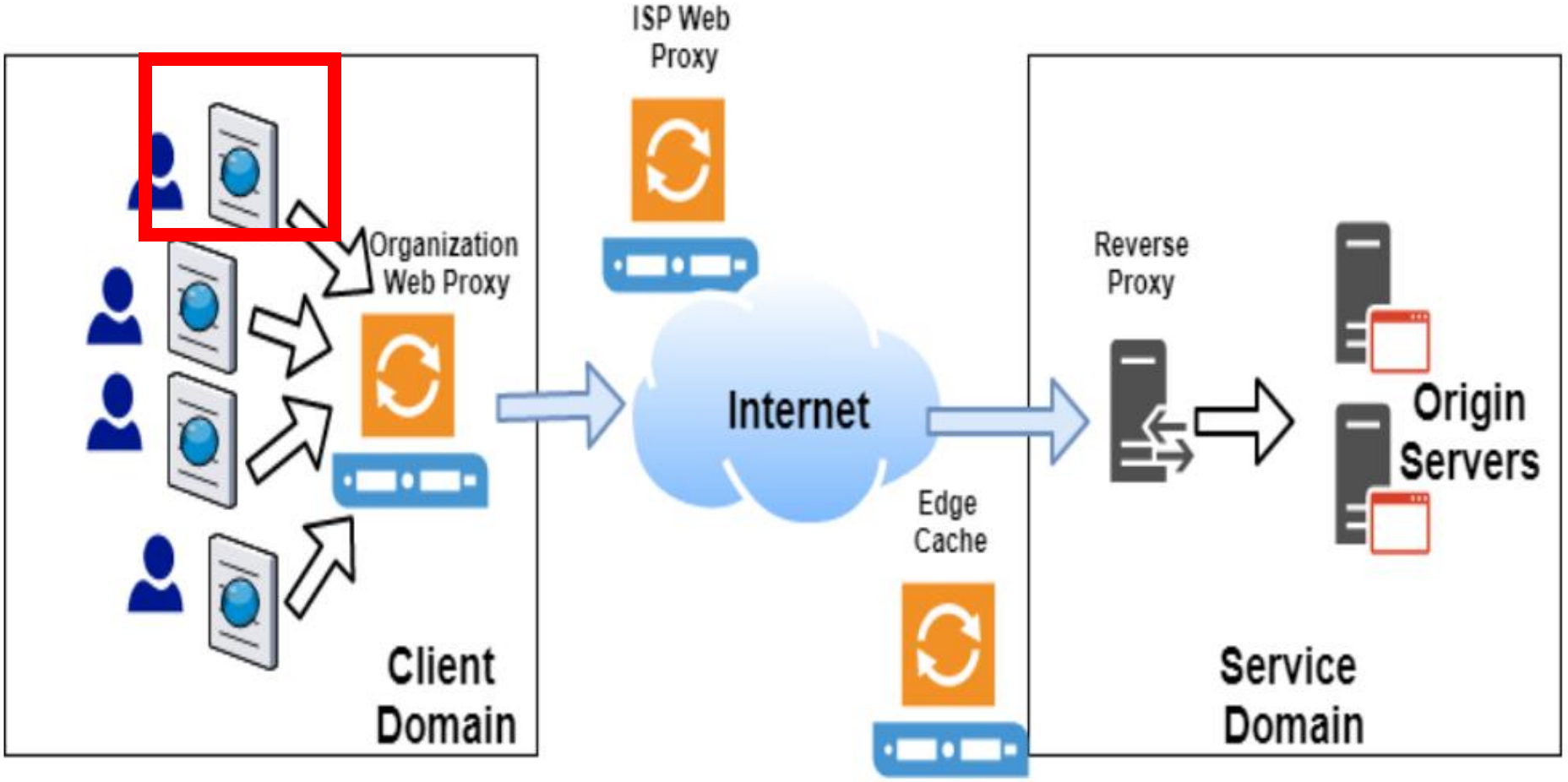


Figure 6-2. Web Caches in the Internet

Web Caching

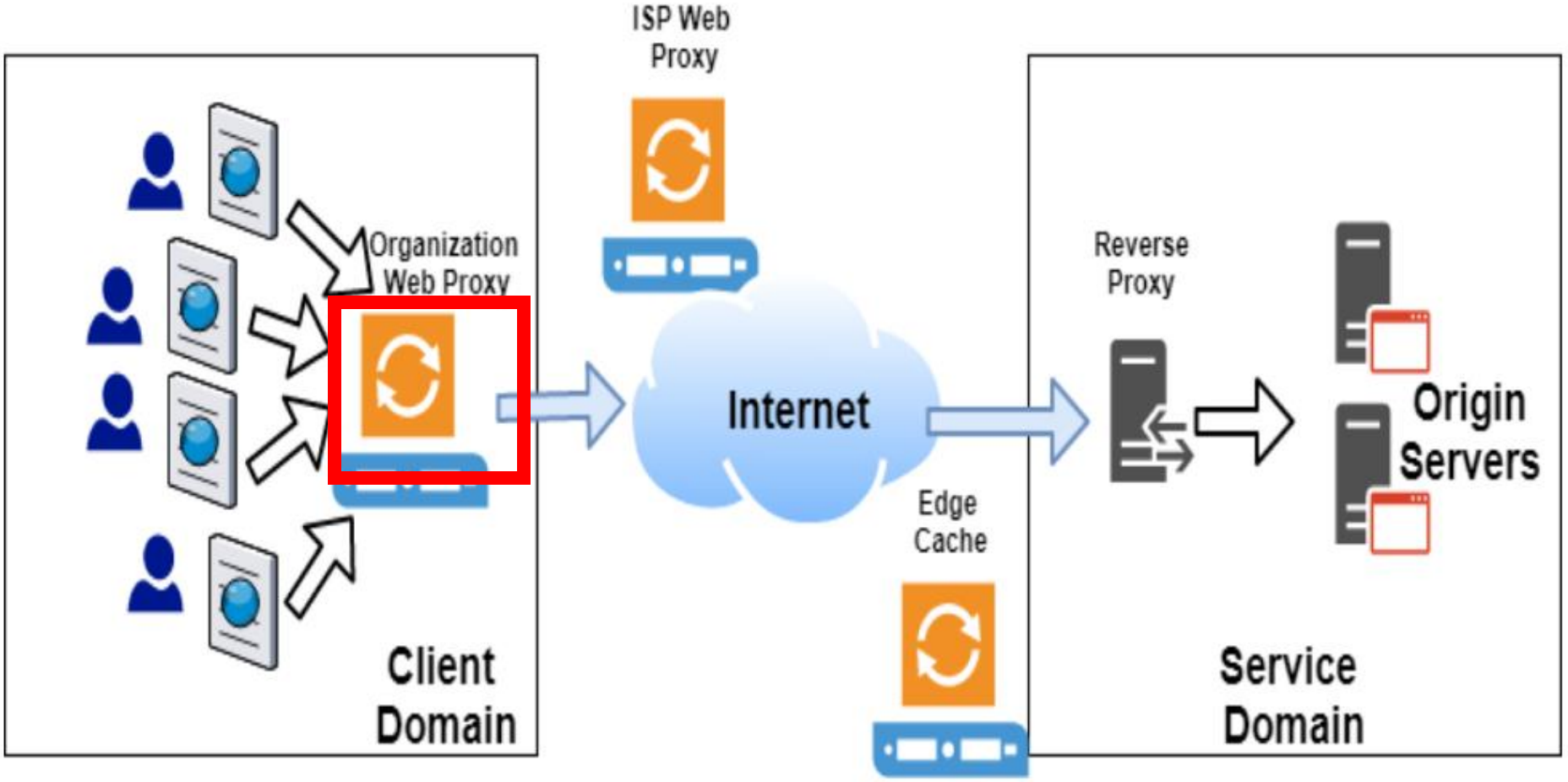


Figure 6-2. Web Caches in the Internet

Web Caching

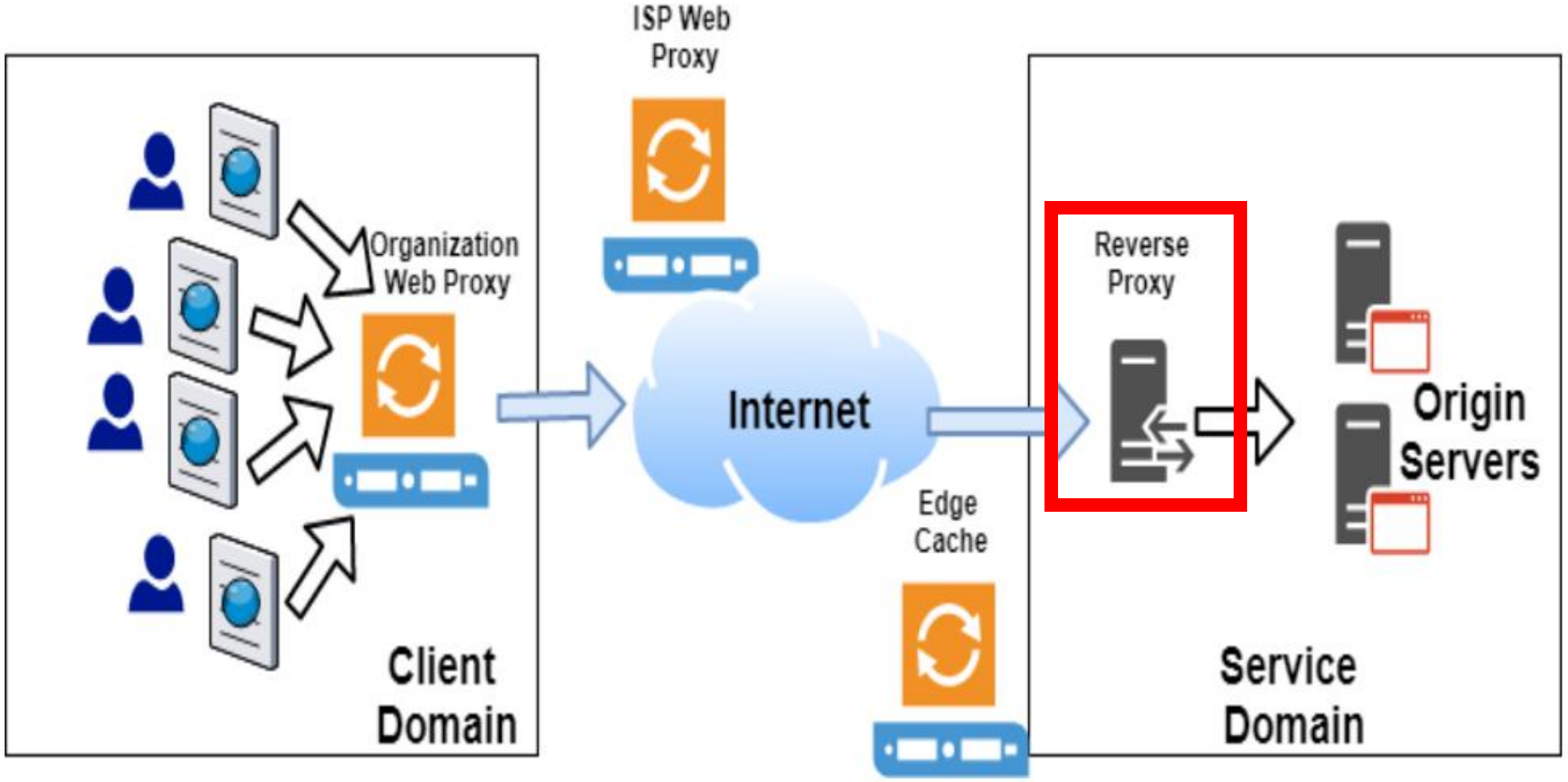


Figure 6-2. Web Caches in the Internet

Web Caching

- Web browser caches are also known as private caches (for a single user).
- Organizational and ISP proxy caches are shared caches that support requests from multiple users.
- Edge caches: live at various strategic geographical locations globally, so that they cache frequently accessed data close to clients. (Example?)
-

Consistency

- Strong consistency versus Week consistency
- Strong consistency: All replicas must exhibit the same value to clients at all times. (Example application?)
- Eventual consistency: Some replicas may be stale. Clients may see inconsistent values when reading the same value. (Example application?)

Eventual Consistency

Promotes availability over consistency

Optimistic

Consistency achieved with some latency (eventually)

Eventual Consistency

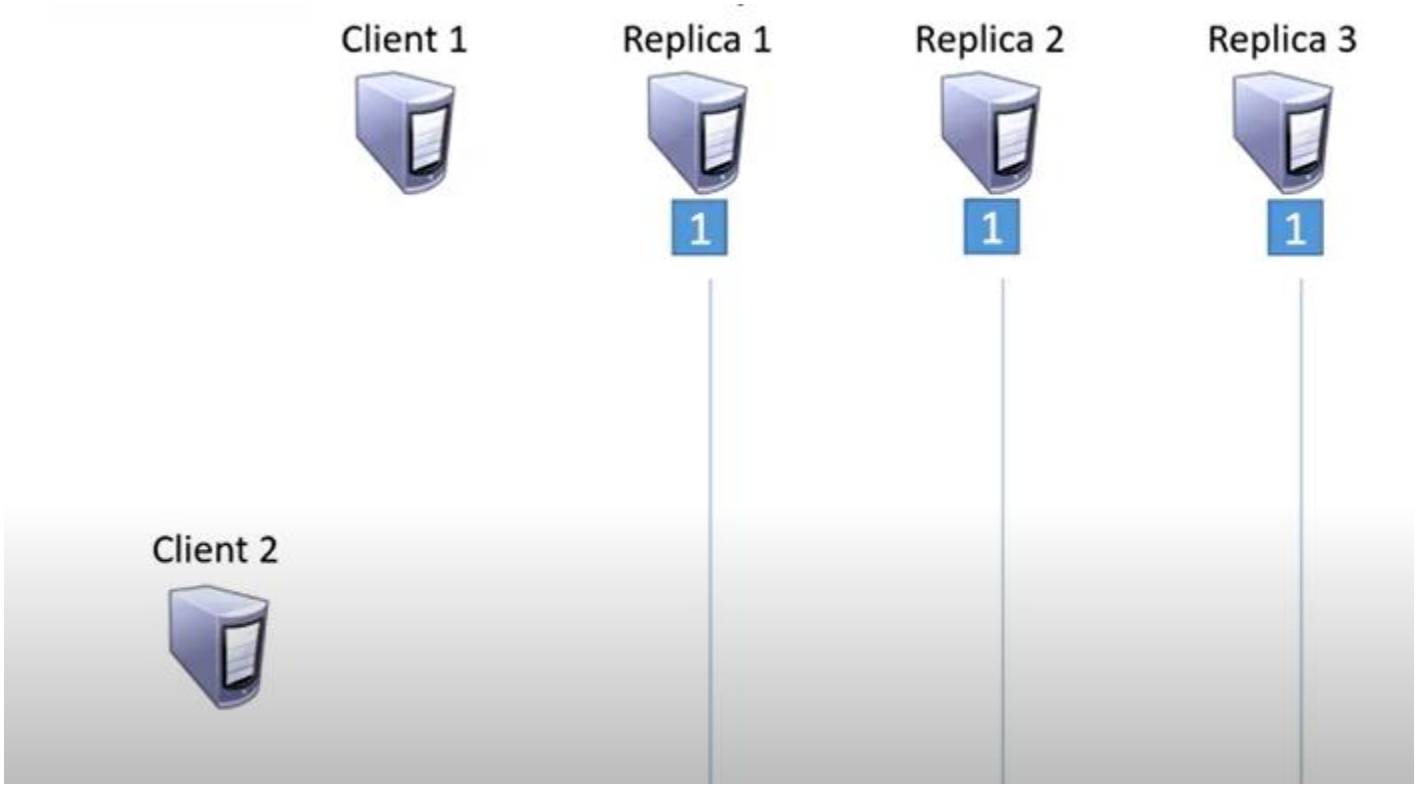
Availability achieved through partitioning and replication

Requires successful write to one or more replicas

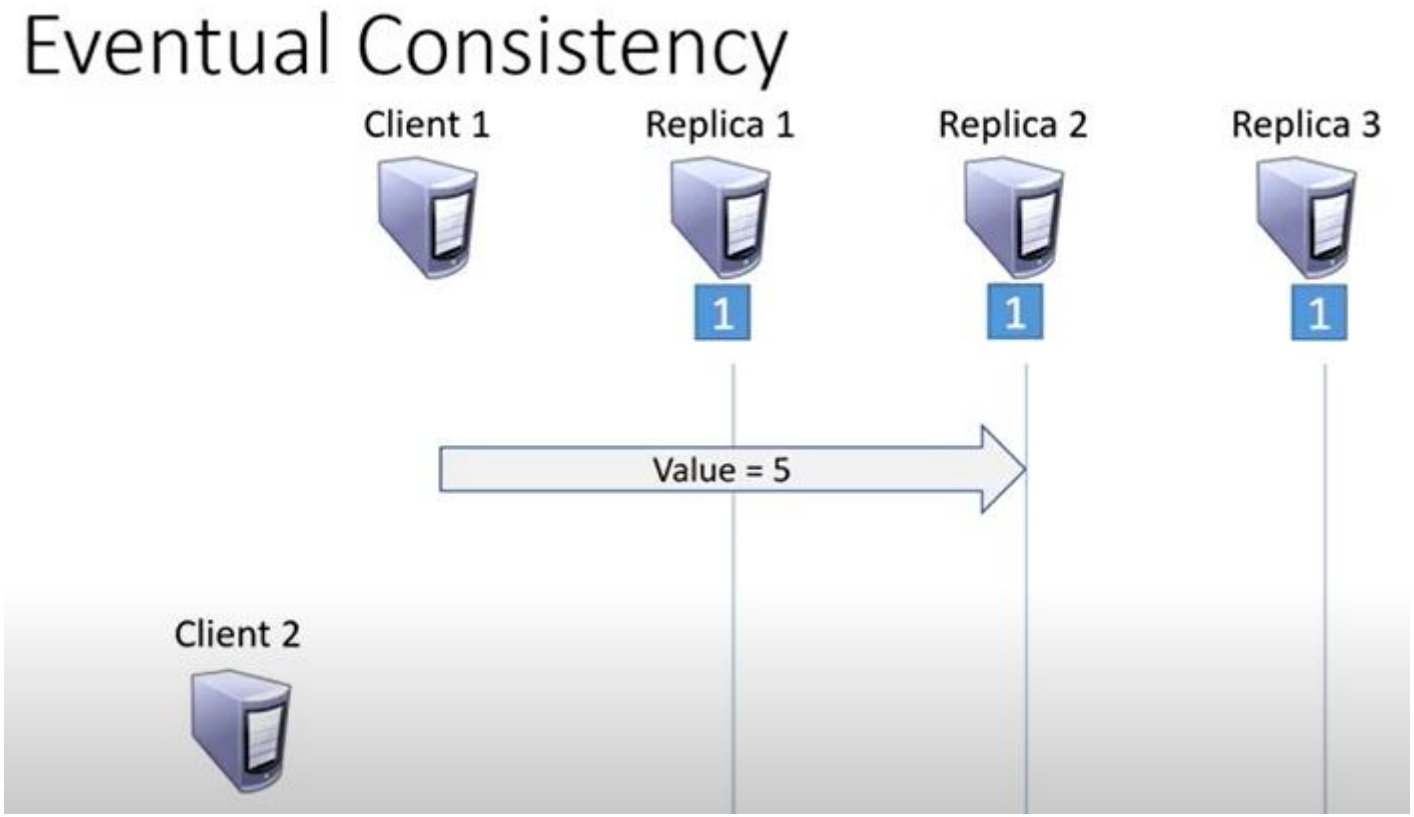
Consistency achieved through background mechanisms

Latency for achieving consistency affected by number of replicas

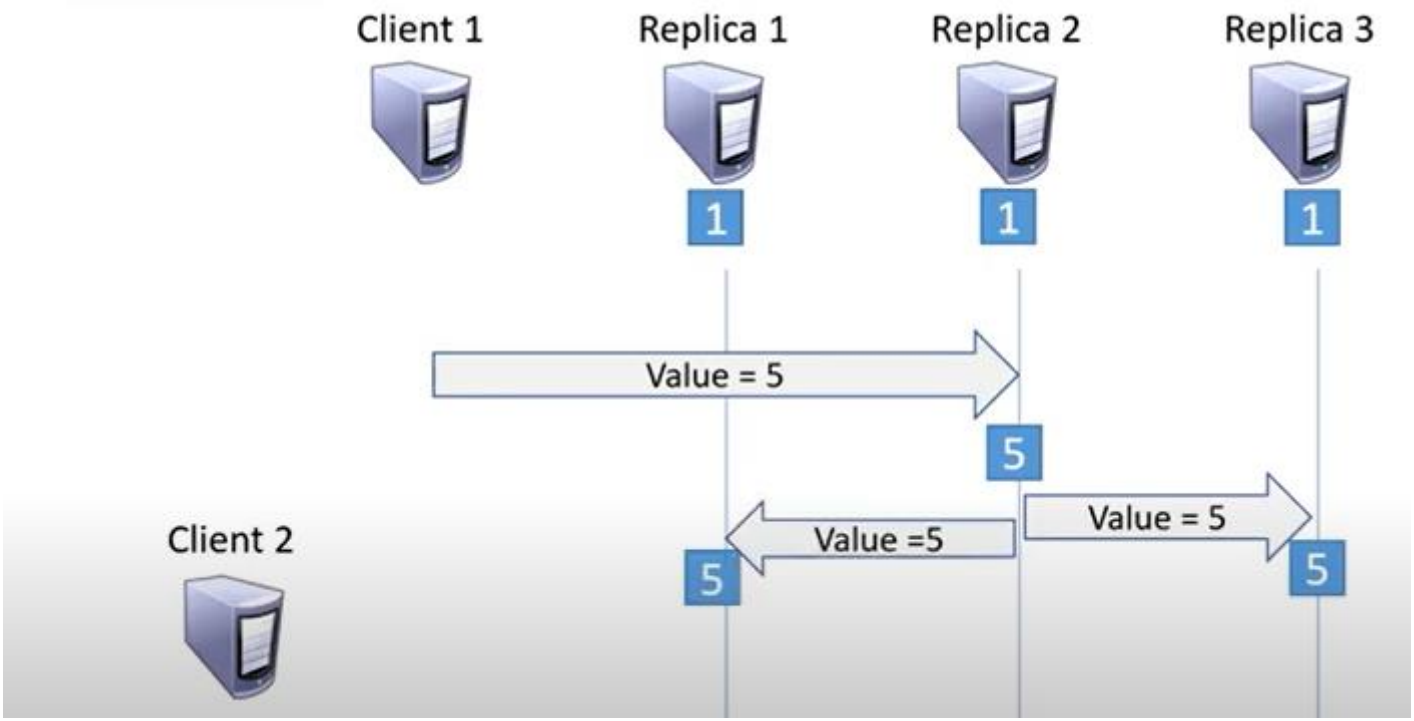
Eventual Consistency **Example**



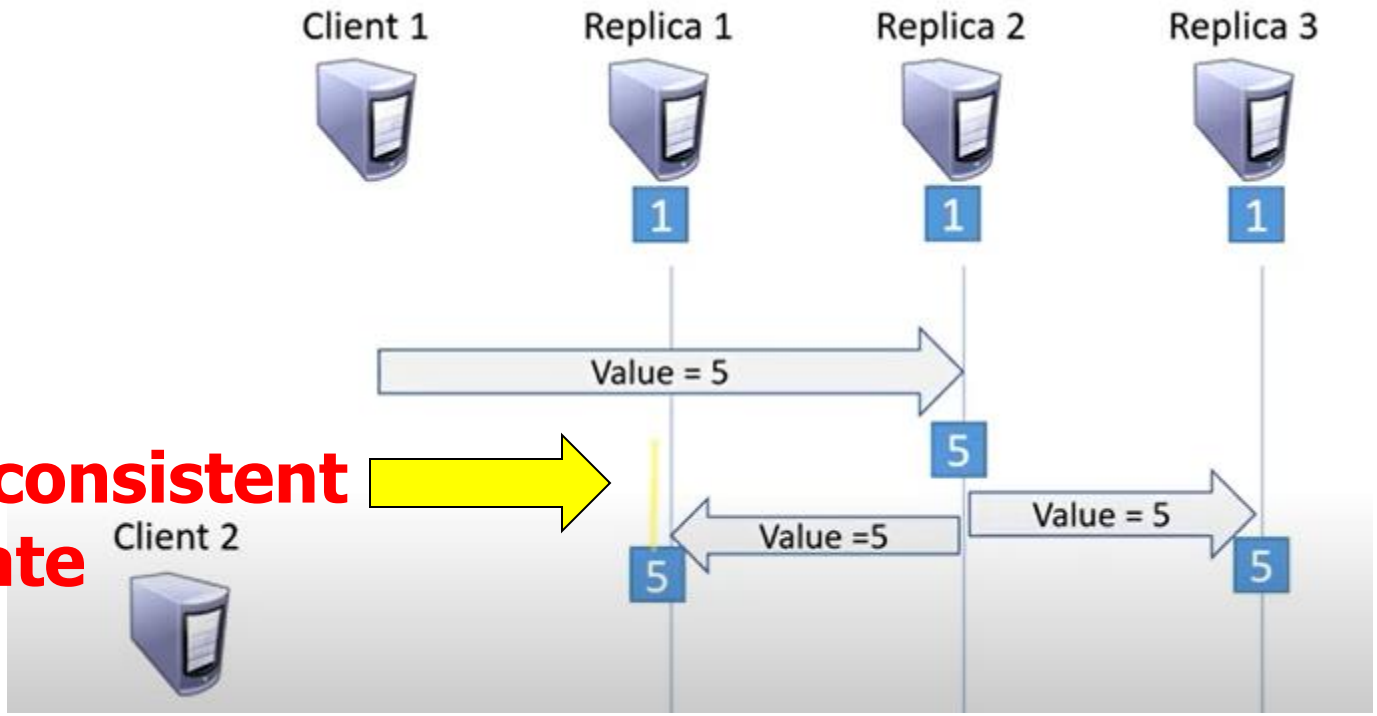
Eventual Consistency **Example**



Eventual Consistency **Example**

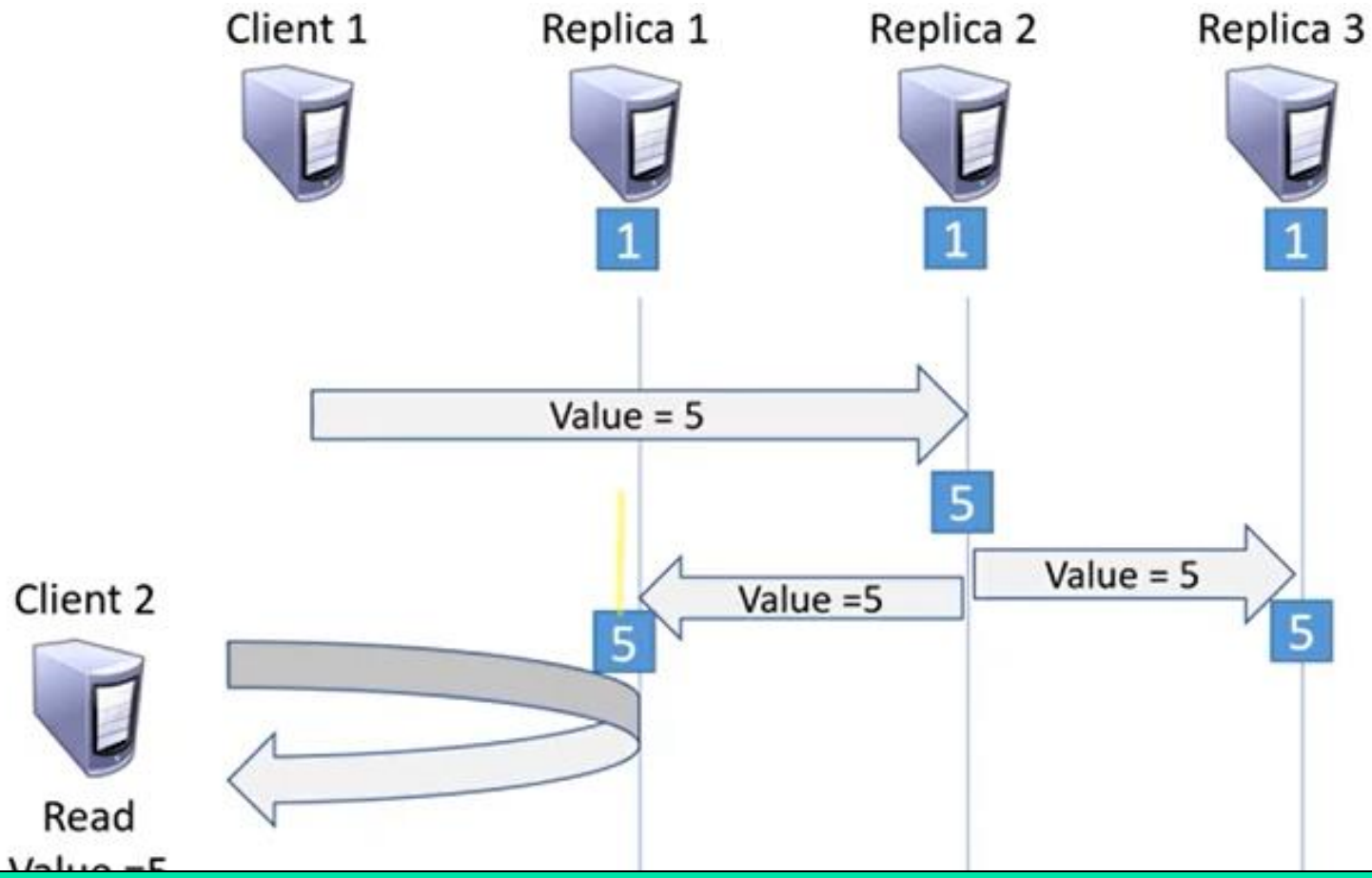


Eventual Consistency **Example**



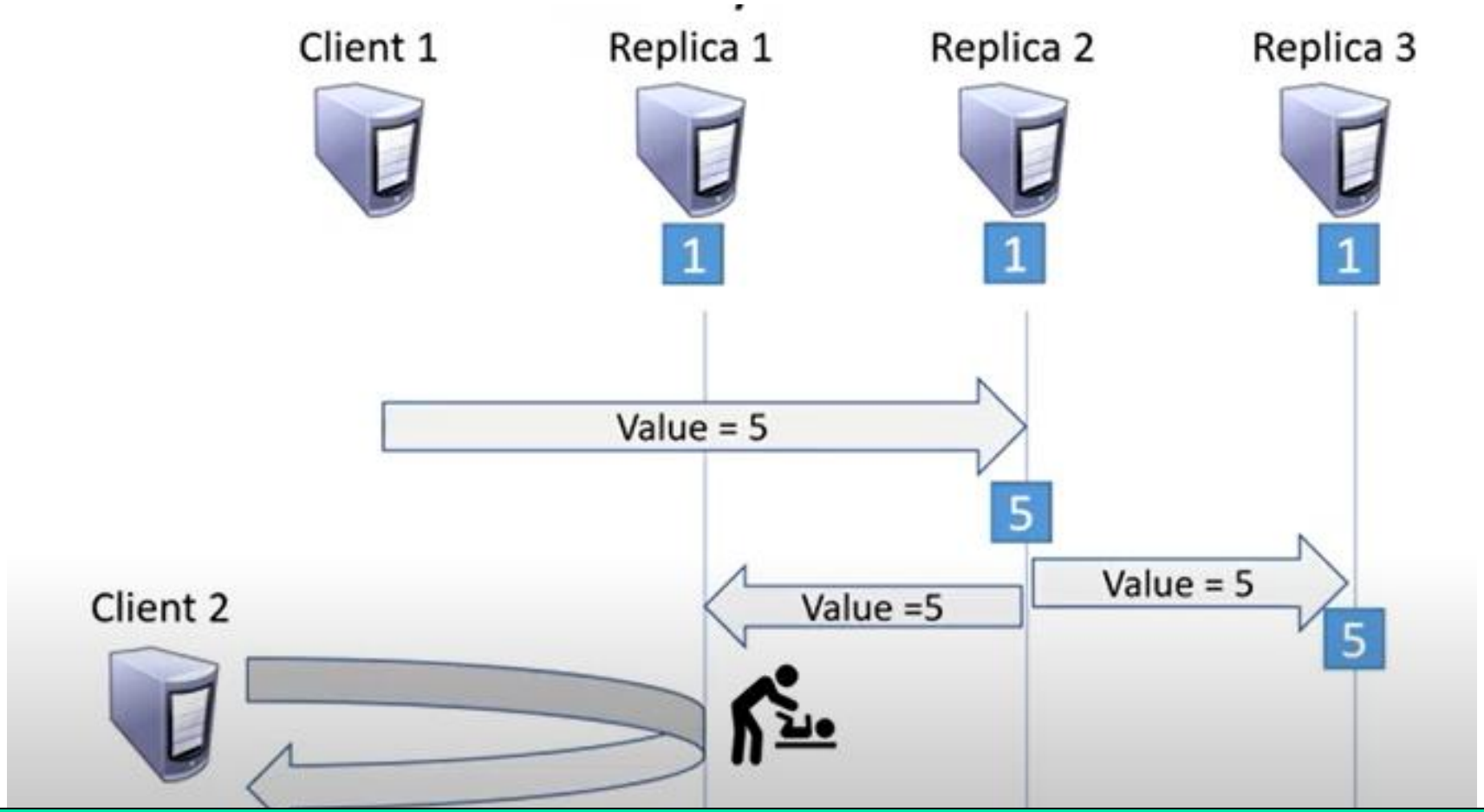
Inconsistent state

Eventual Consistency **Example**



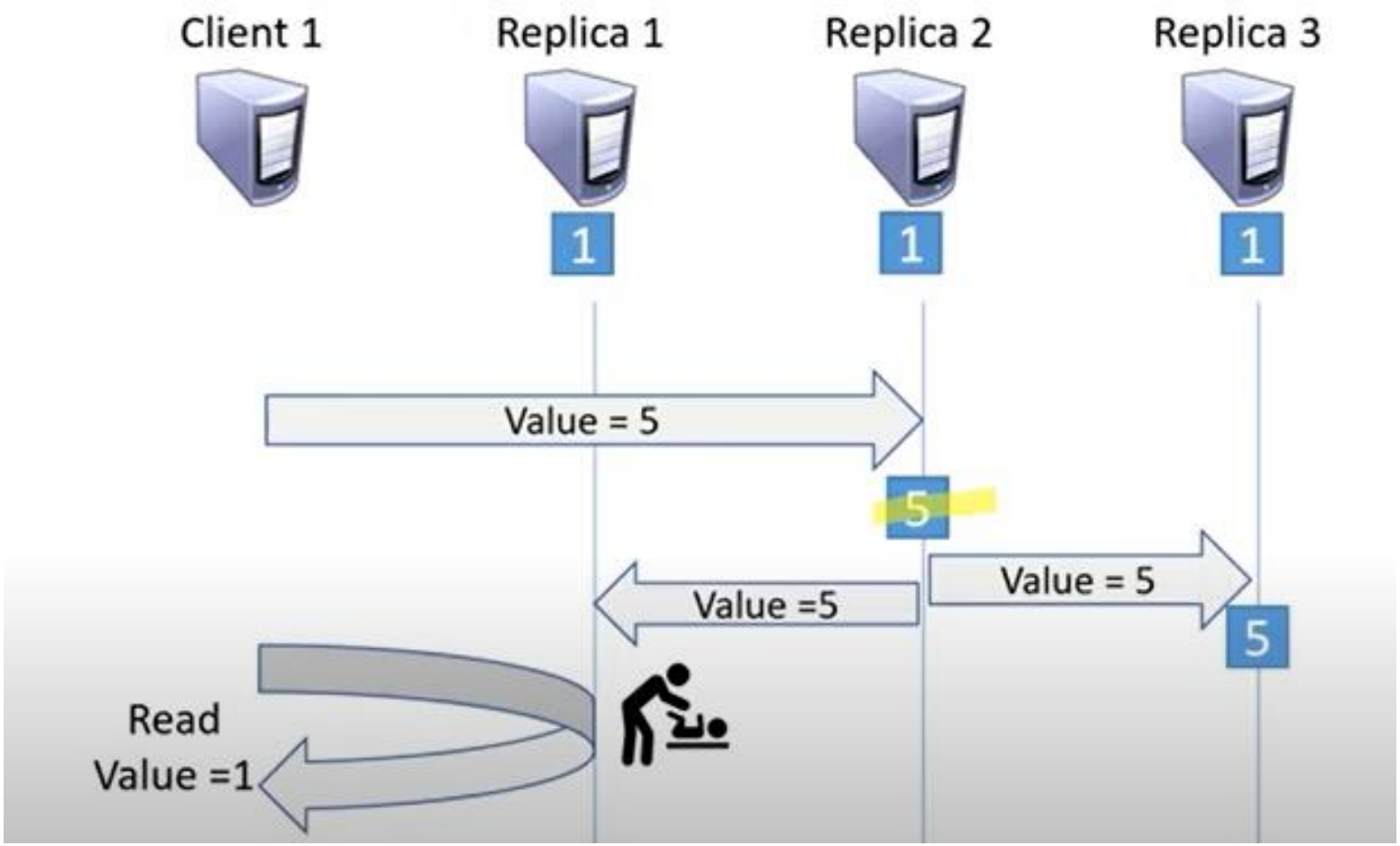
This reading is a happy scenario as it is after the inconsistency period

Eventual Consistency **Example**



Unhappy scenario: When replica 2 tries to update replica 1 the network is busy or for some other reason the value doesn't get updated before client 2 requests to read value

Eventual Consistency - NOT reading your own writes Example



Required Readings

- Chapter 6: Distributed Caching, from the textbook: “Foundations of Scalable Systems”, Ian Gorton, O’reilly Media Inc., 2022.