Dealing with Data Inconsistencies in GFS

Learn how applications and GFS deal with data inconsistencies.

We'll cover the following

- Inconsistencies dealt with by applications
  - Undefined regions
  - Padding and record duplicates
- Inconsistencies dealt with by GFS
  - Stale data
  - Data corruption

Inconsistencies dealt with by applications

In the previous lesson, we saw the states of file regions after data mutations (random write/record appends). In some cases, the file region becomes undefined or inconsistent. Let’s look at how the applications using GFS deal with these cases.

Undefined regions

Undefined regions are produced due to the concurrent execution of random write operations on the overlapping region. GFS doesn't serialize concurrent writes. Applications that want to use random writes need to be wary of conflicting regions. There can be many ways to avoid undefined regions at the application level. One way is to serialize the concurrent random writes at the same offset at the application level and always write the same length records so that one write operation completely overwrites the previous write (to avoid mixing data from multiple writes producing undefined regions). Otherwise, concurrent writes can be applied to nonoverlapping regions without an issue.

The application using the GFS needs to somehow control the concurrent writes to the same regions by its multiple threads if it can't live with those undefined regions. If we make an analogy with the OS world, we can think of an undefined region as a critical section that must be protected (for example, via locks) for it to be in the same condition. What clients can do is acquire a lock for the region (for example, by using Chubby), and whoever has that lock can write. In this way, the region will never be undefined because one client will write at a particular time, and what that client wrote is not intermingled with other's data (assuming same-length records).

> **Note:** The global mutation order is first set out by the manager's chosen lease grant order, and then within a lease by the primary's assignment of serial numbers to mutations.

Padding and record duplicates

For record append operations, GFS guarantees that it will append the data at least once as an atomic unit. But, we've seen that retrying the append operation adds padding and produces record duplicates in between the defined regions. However, it wouldn't happen often.We don't want the GFS system to take care of removing padding and record duplicates because that will slow it down. Therefore, it is the client's responsibility. The GFS client code is extended to identify and discard extra padding and record duplicates. We can identify undefined regions and record duplicates in the manner described below:

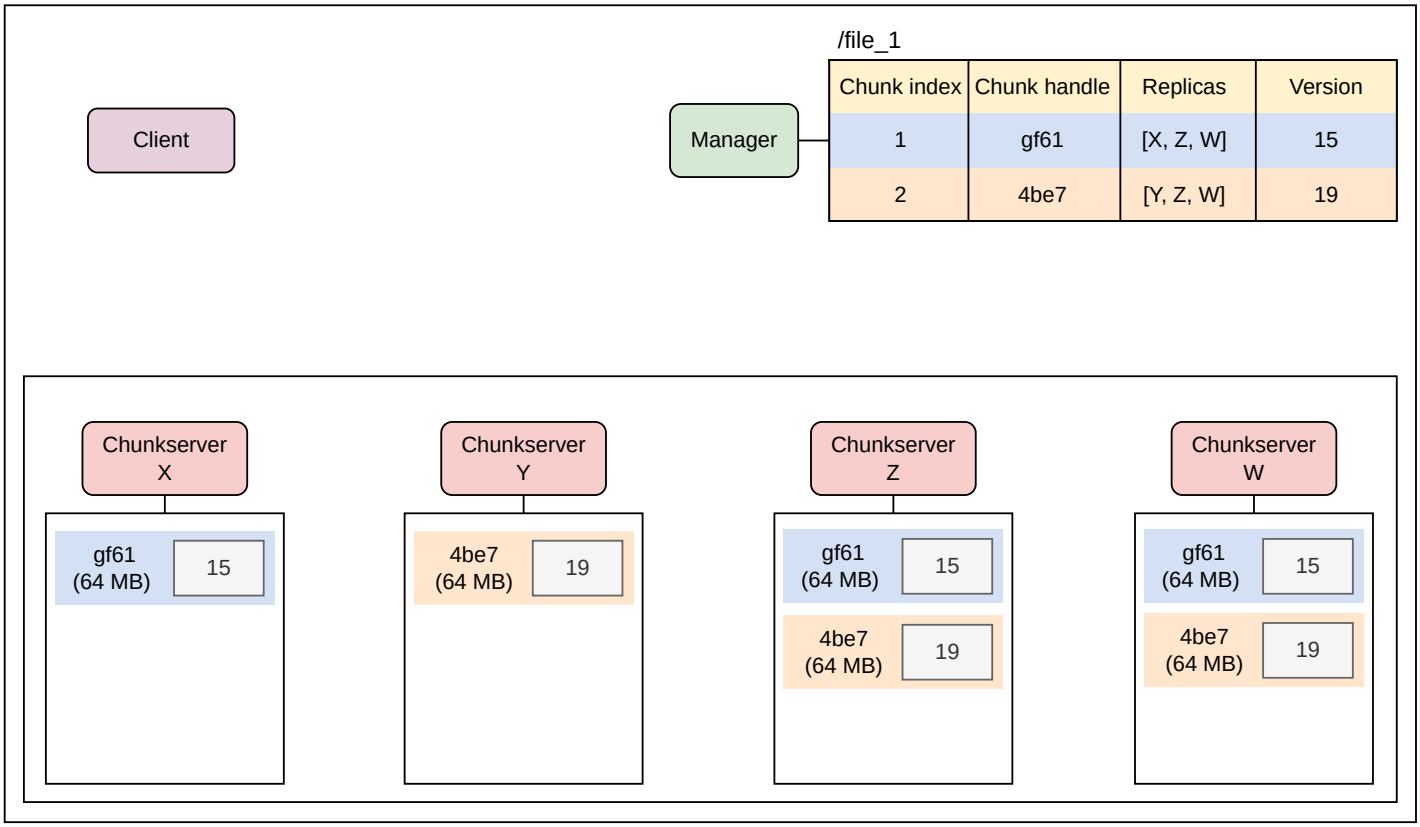
- To determine if the region is defined, records are written with the checksum of the writes.
- To find out duplicates, applications can put in monotonically increasing numbers.

Inconsistencies dealt with by GFS

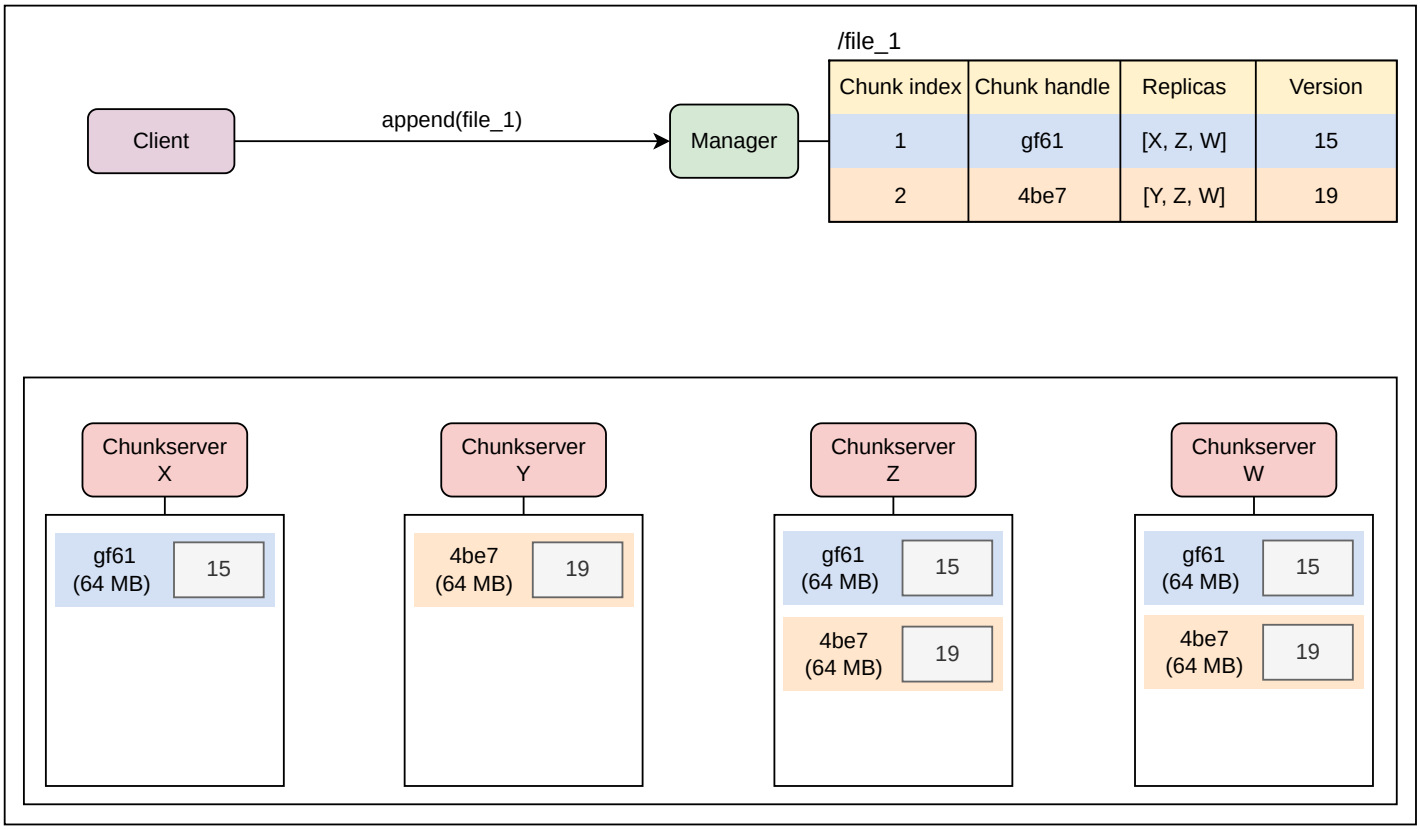
The failed mutations resulting in inconsistent data among replicas and data inconsistencies produced due to disk failures are handled by GFS itself. Let's see how GFS copes with these data inconsistencies.

Stale data

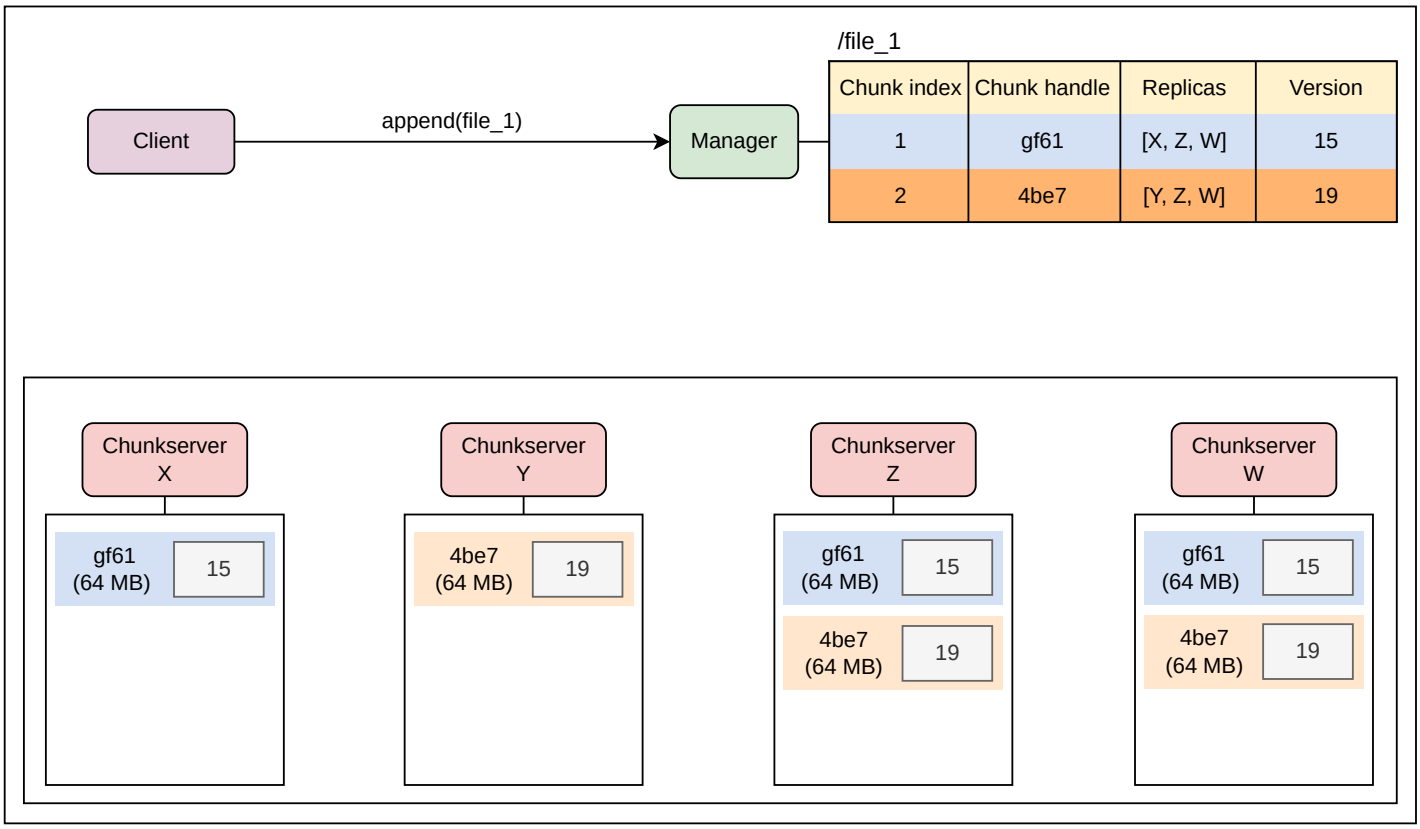
The failed mutations result in data inconsistency among replicas because some replicas couldn't apply the mutation due to a failure at the chunkservers. Replicas that miss the mutation become stale and such replicas shouldn't be used for serving the read requests or to apply any further mutation. If we do, the clients will get different data from replicas. GFS detects stale replicas using chunk version numbers and deletes such replicas as soon as possible. Let's see how chunk version numbers are maintained and who does it.



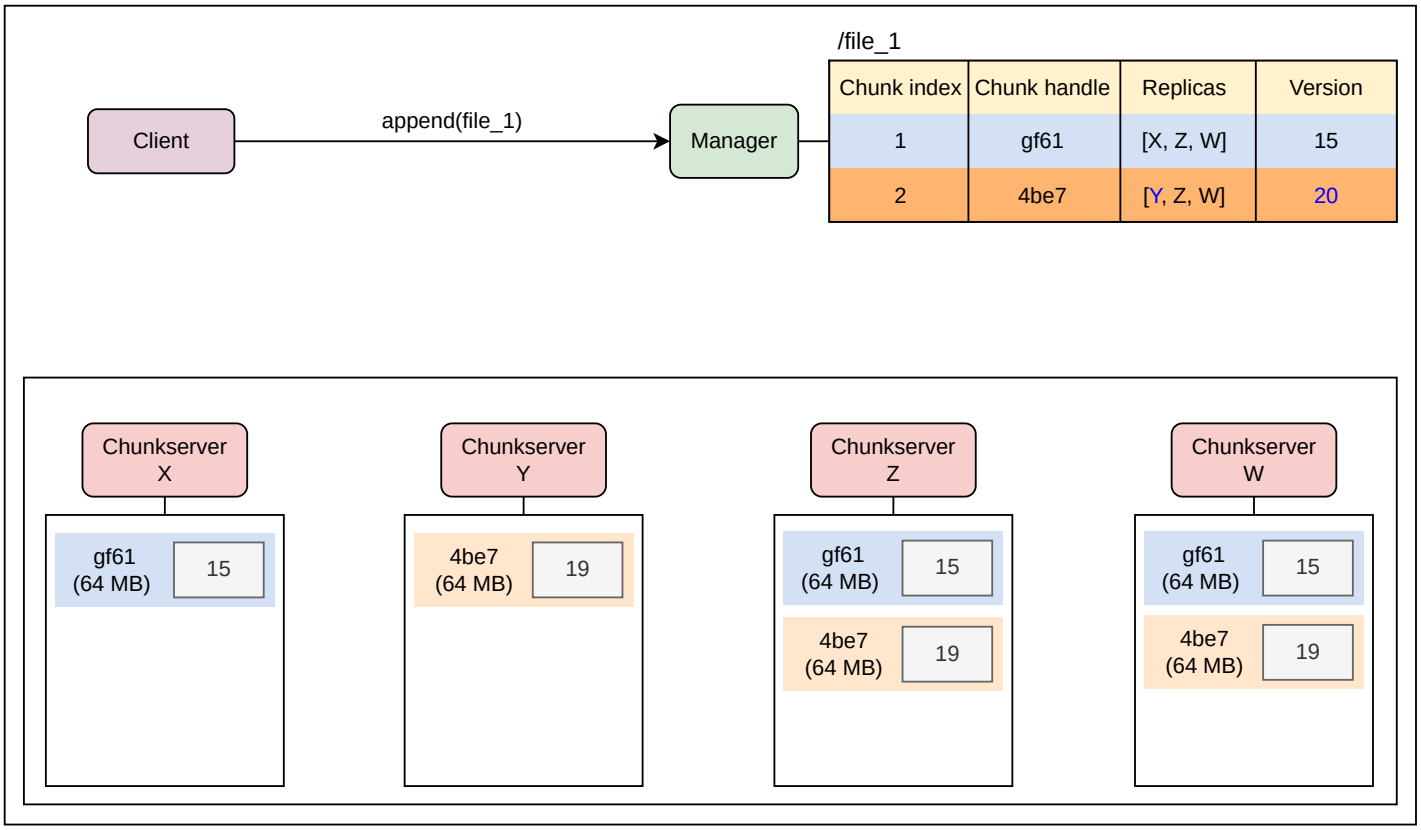
There are two chunks for file_1. The metadata is stored in the manager node, a chunk number is maintained for each chunk at the manager, and chunkservers also update the chunk number following instructions from the manager



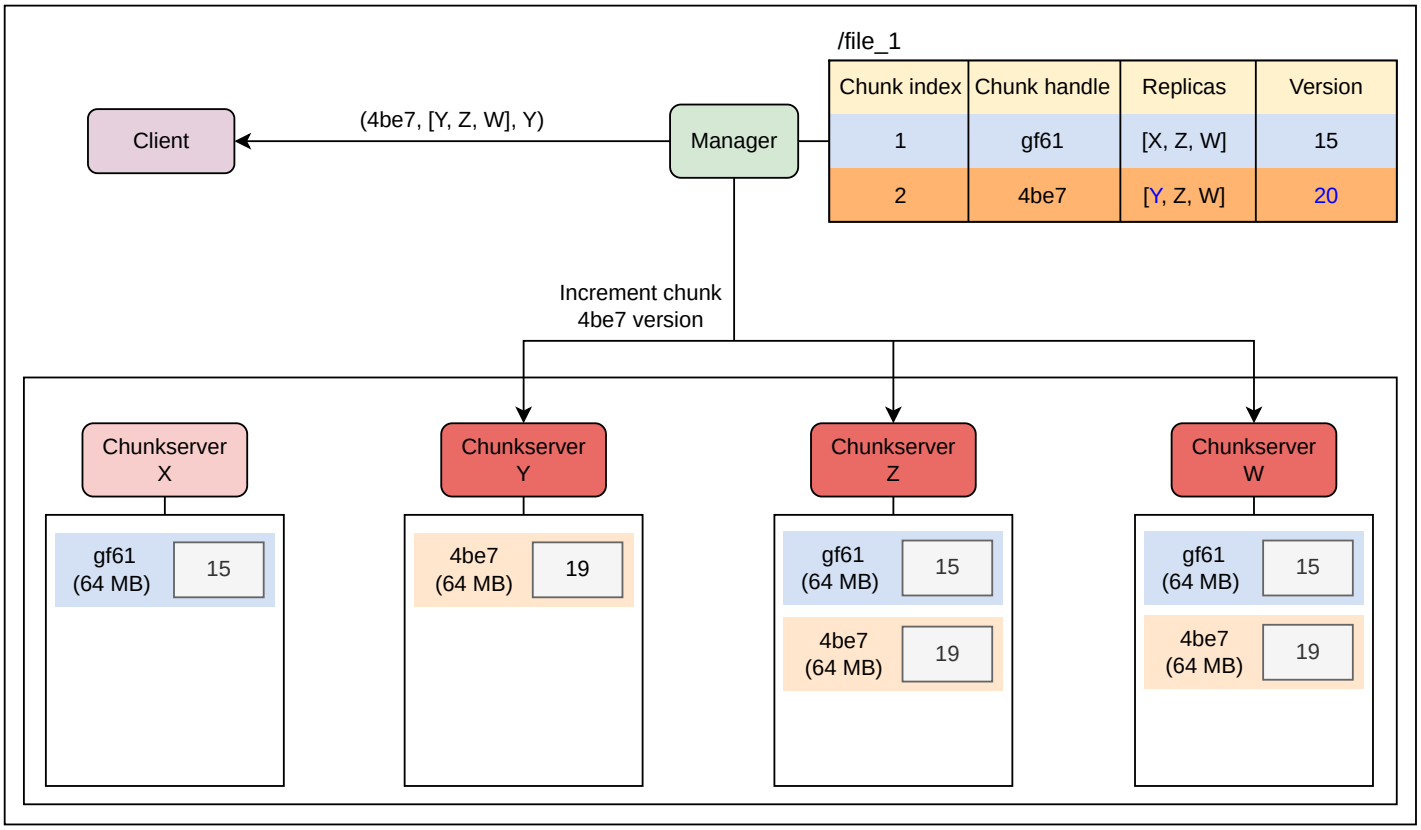
A client asks the manager for a chunk lease and replicas to perform an append operation



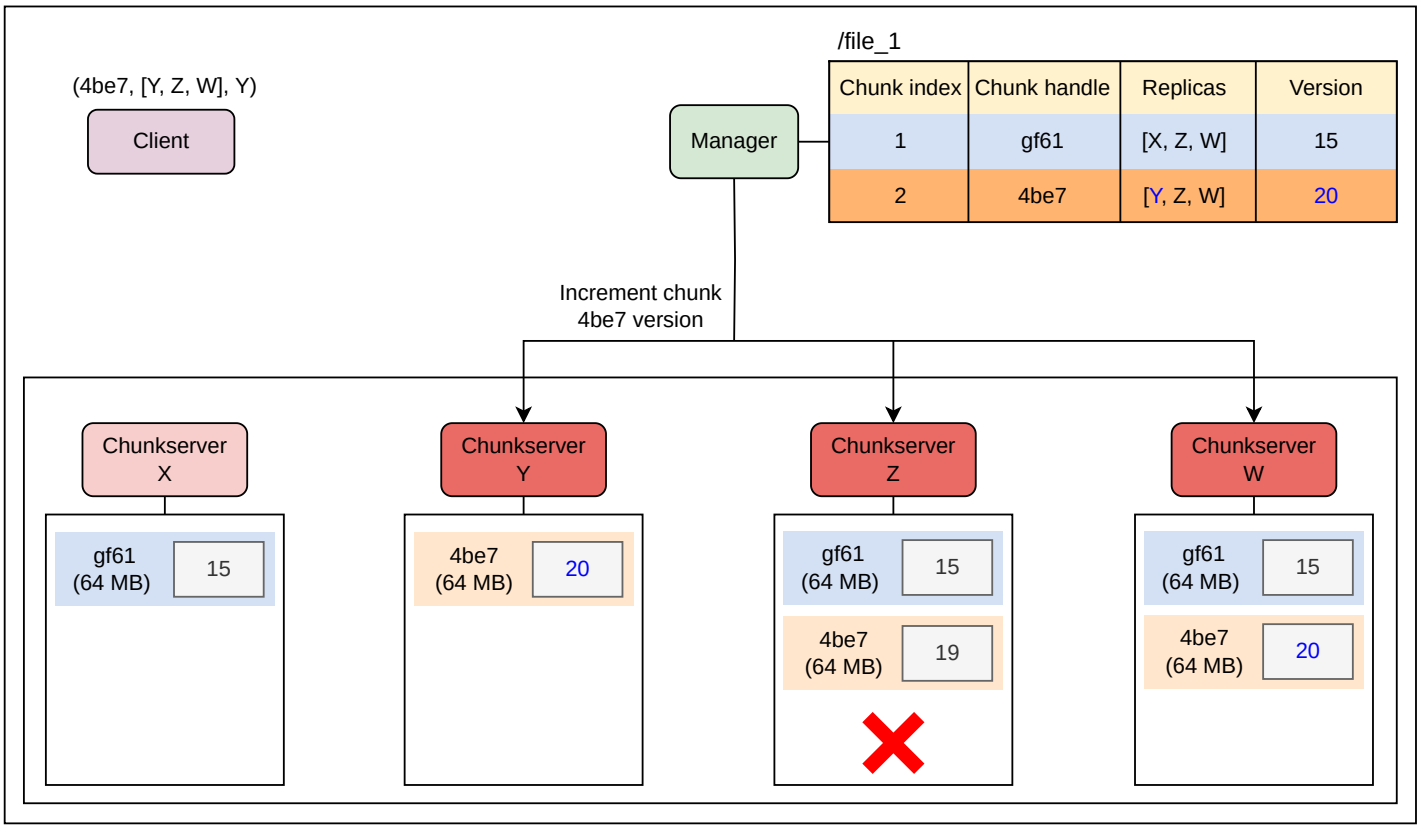
The operation was append so the manager finds the last chunk of the file, and check if any of its replicas already has a lease



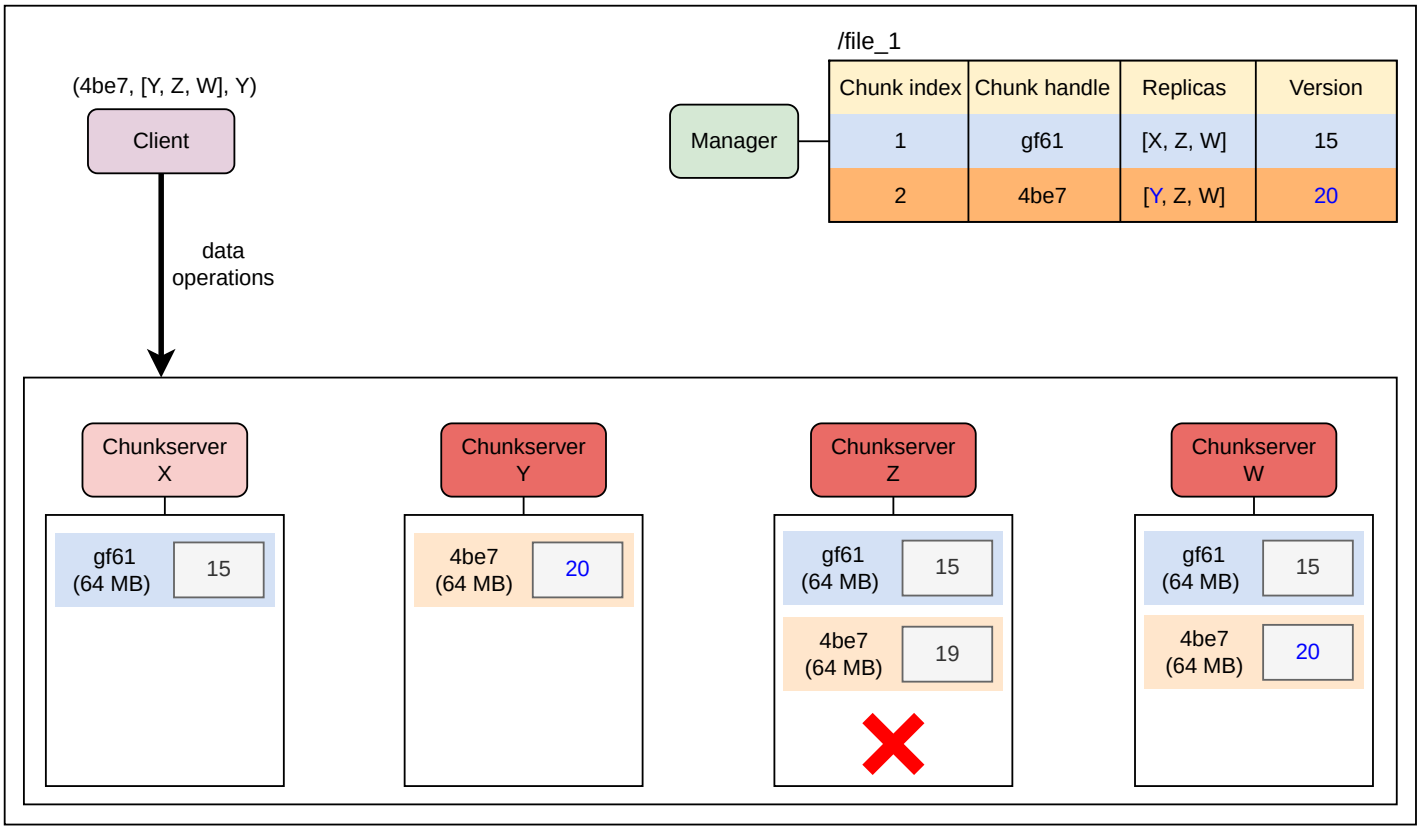
Assuming that no replica is currently on lease, the manager grants a lease to replica Y, and increments the chunk version number



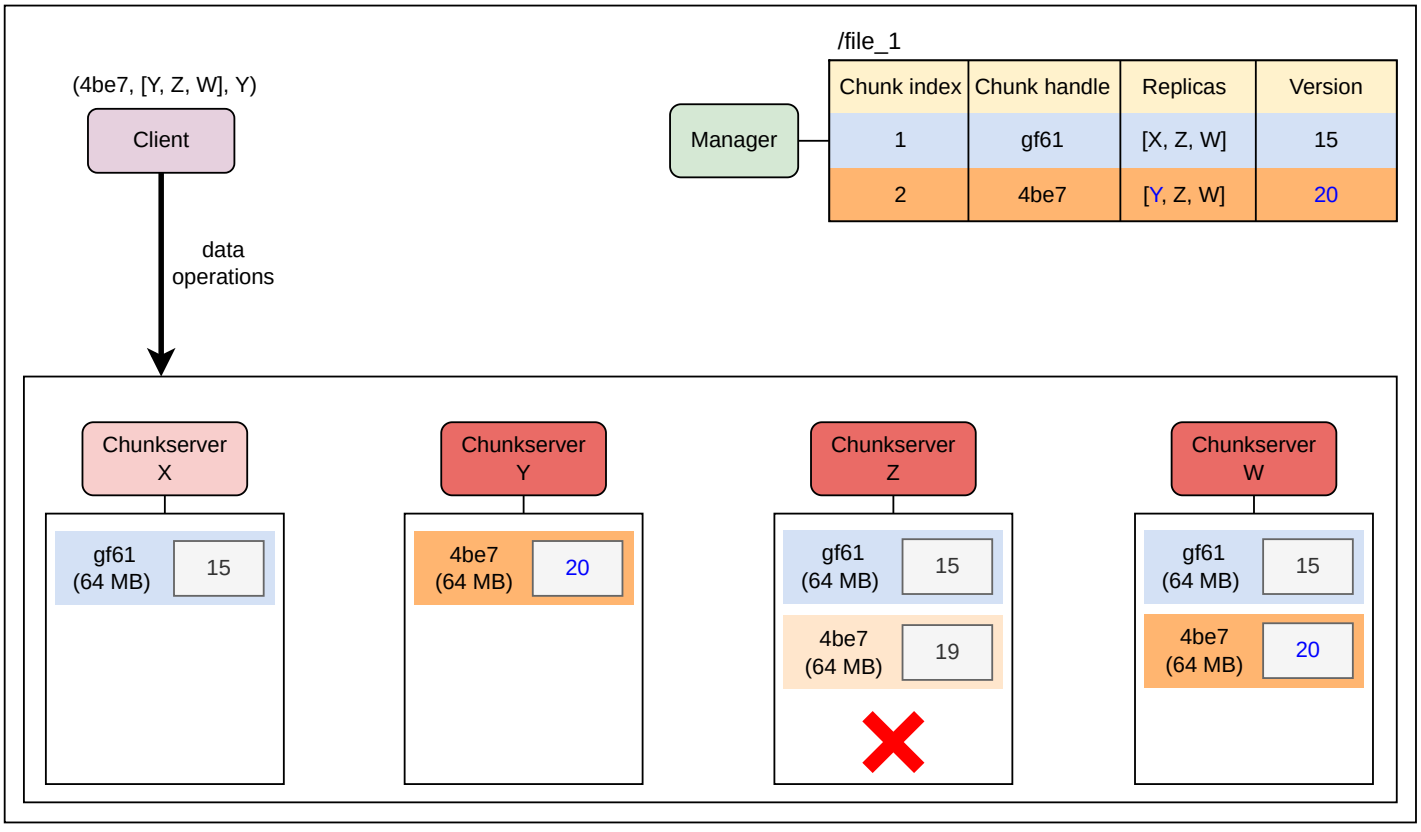
The manager asks all replicas to update the chunk version number



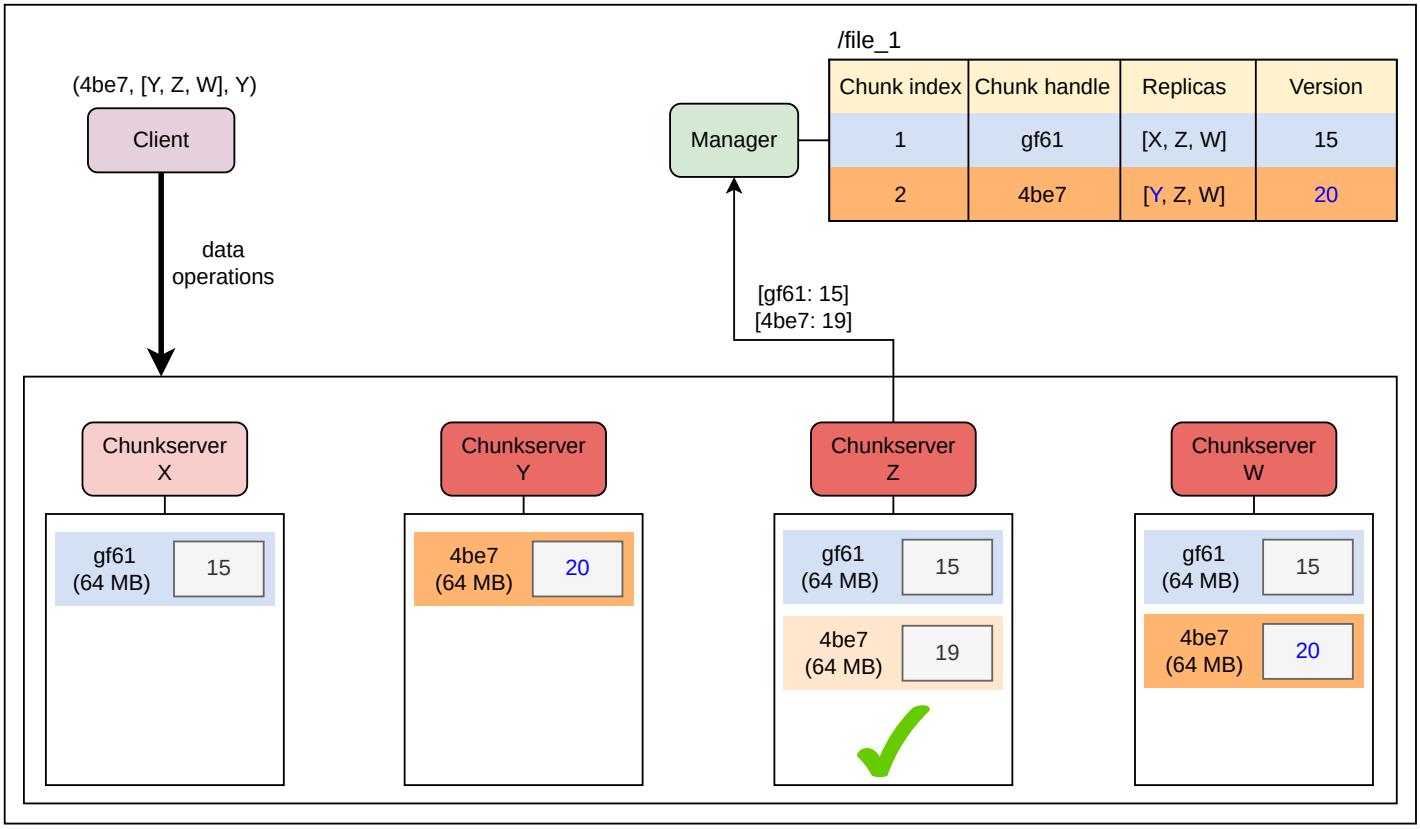
Replica Y and W update the version, but replica Z is unavailable



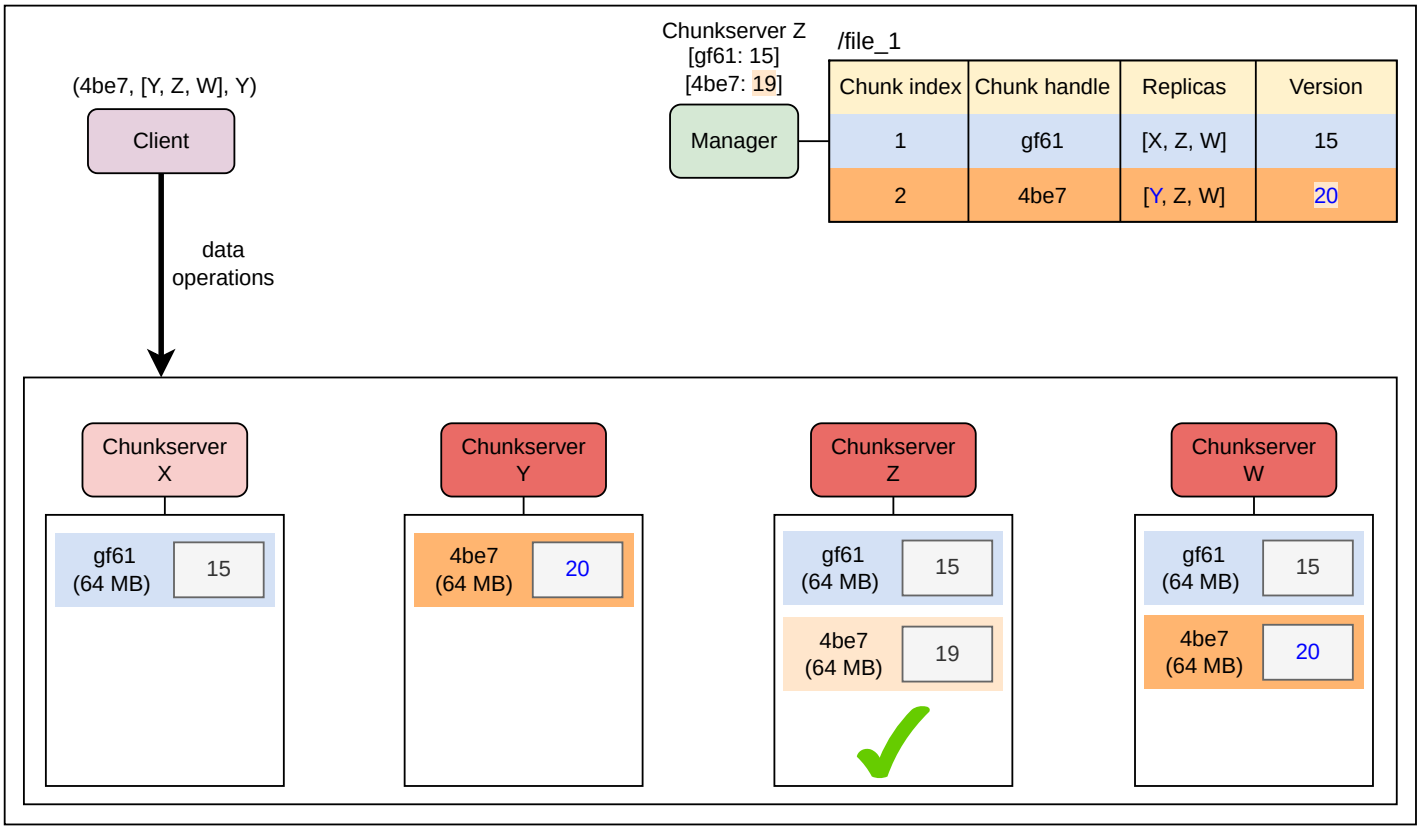
The client performs the data operation on the chunk



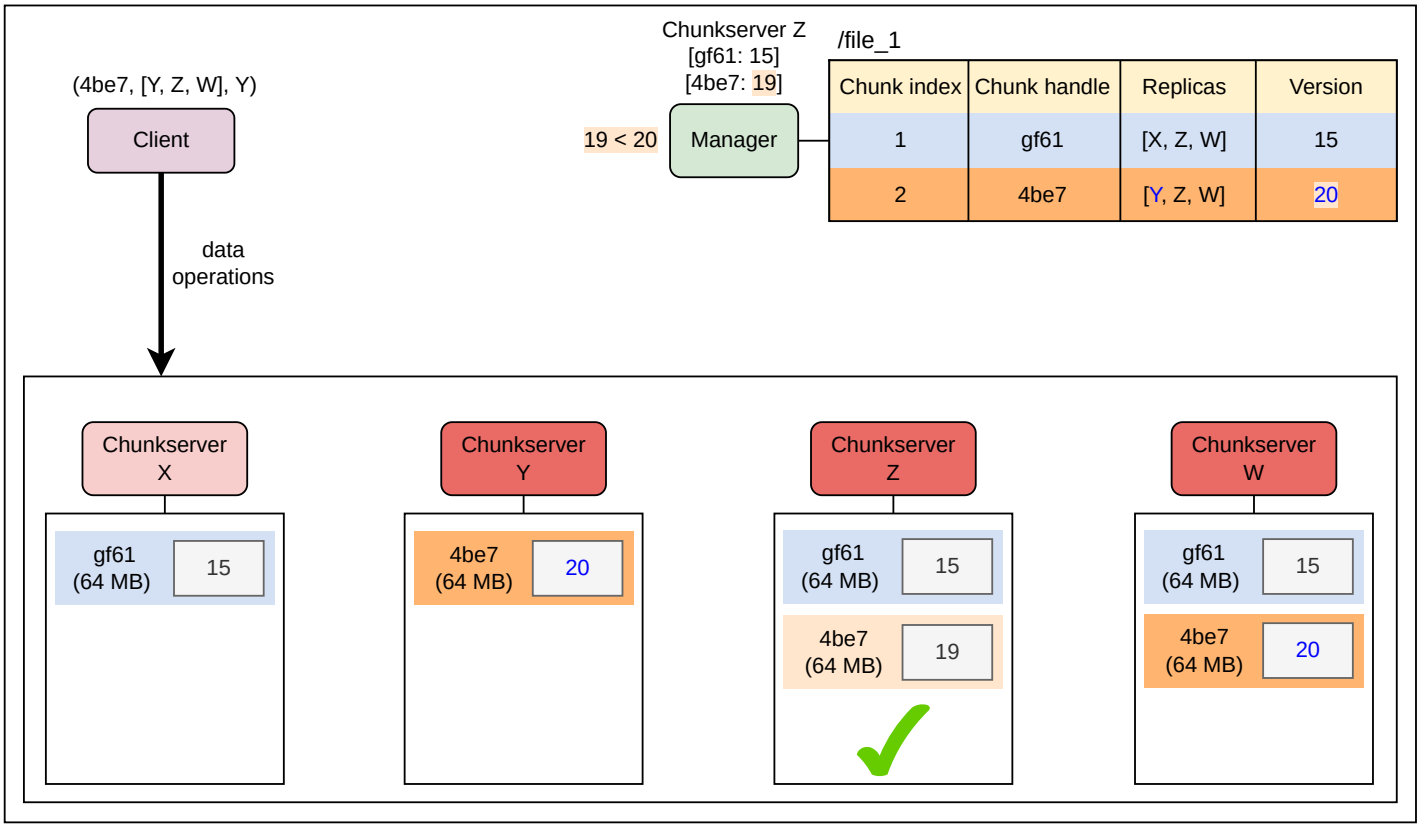
The primary replica Y and the secondary replica W perform the mutations on chunk 4be7, but Z is unable to



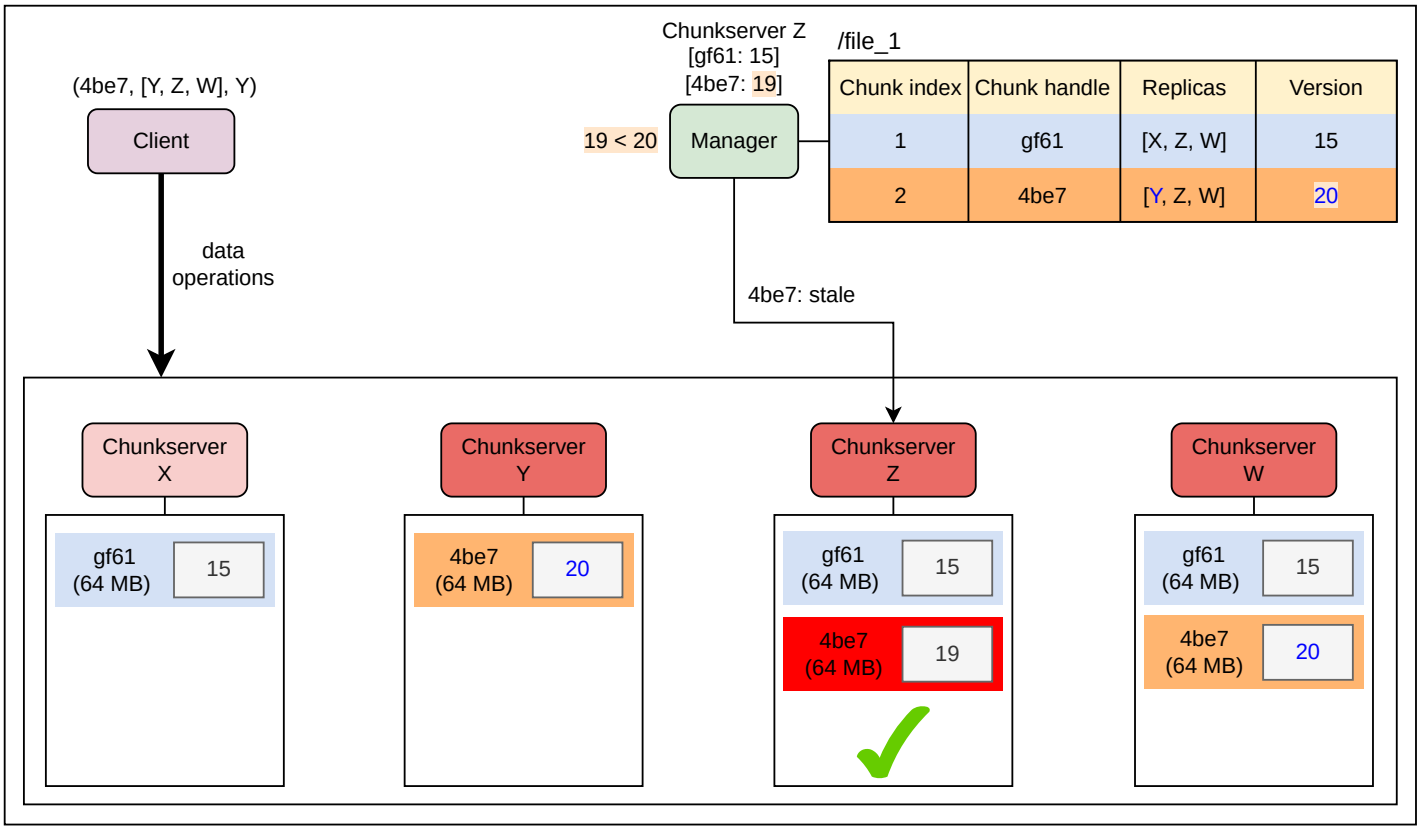
Replica Z recovers from the failure and reports its state to the manager



The manager verifies the version numbers for chunks



The version number for chunk 4be7at chunkserver Z is less than the current version of 4be7 at the manager node



The manager tells the chunkserver Z that 4be7 is stale. Chunkserver Z will then update the chunk data by reading from valid replicas

The manager node maintains each chunk's version number. It increases the version number of a chunk upon granting a new lease and asks all chunk replicas to update the chunk version number. Suppose a replica is not available because the chunkserver holding it is down. In that case, the chunk version number won't get updated on that replica. When the chunkserver restarts after a failure, it sends a heartbeat message to the manager to report its state, including the set of chunks and their version number. If the manager finds a replica with a version number lower than the one it has for that replica, then it marks that replica as stale. The stale replicas are not used to serve any user requests and are removed in regular garbage collection.

> The chunk's version number information is stored persistently.

Furthermore, the manager includes in its response the version number of a chunk when a client requests a read or write operation on that chunk. While performing the operation on the chunkserver, the version number is verified, and the operation is performed only if the version numbers match.

Point to ponder

Question

Since the metadata information is cached on the client side, can the readers still read the stale data?

Hide Answer

The timeframe for which the metadata information is cached, is bounded by the timeout parameter and the next time the file is opened. The metadata is removed from the cache once the time frame reaches its limit. This doesn't totally solve the problem. Readers can still read the stale data during a small window.

Moreover, GFS is built on the assumption that most of its files are append-only, so the returned data would be the premature end of the file (this will happen when client-cached metadata is stale). In append-only files, data is appended at the end of the file, and the previous data in the file remains the same, so if a client reads the old version of the file, it will contain the correct data, but some data from the end will be missing. On the other