Storage servers are the primary source of data for client requests. Each storage server is responsible for certain key-ranges (aka shards). As more writes come in and the storage server starts getting full or certain shards becomes write hot, the data distribution layer starts splitting those shards and move them around to different storage servers. Data distribution indicates so to the storage servers by writing special private mutations which when processed on the storage server take special actions to change the shard boundaries. More details on the data distribution internals can be found here: https://github.com/apple/foundationdb/blob/master/design/data-distributor-internals.md

update(): Main actor that keeps running on a storage server. It pulls mutations from the transaction log system (TLog) and applies those mutations to the storage server.

Private Mutations: Private mutations exist for certain in-frequent events. As the name suggests, they are private to a specific storage server. Private mutations are off of the end of the database and are injected by **ApplyMetadataMutation()** to tell the storage server to do something special. e.g. the most common private mutation is to change the shard boundaries.

update() calls **applyPrivateData()** to apply these private mutations. Which takes special action based on the type of private mutation. Here, we will focus on the private mutations that change the shard boundaries.

It could be the addition of a new shard, or the deletion of an existing shard. Please note that, we don't allow overlapping or conflicting shard boundaries at the storage server level. The data distribution layer makes sure that the storage servers only receive non-overlapping key-ranges.

There is a set of two private mutations to indicate a change in the shard boundaries. First mutation in the set will contain begin key of the shard range and second will contain end key of the shard range. After establishing this range, it calls **changeServerKeys()** which drives the addition or removal of a shard from the storage server. It is responsible for splitting or coalescing intersecting shard ranges as well. This only happens when there is an addition of a new shard ranges that intersects with an in-progress delete range. Because the deletion takes some time and new ranges overlapping ranges might get added during that duration. changeServerKeys() then calls **fetchKeys()** to fetch the data for the newly added key-range, from other storage servers. It does so by starting a database transaction and issuing a normal range read.

fetchKeys() fetches data at an established **fetchVersion** which is the current version of the storage server when it processes the mutation which caused the fetch to be initiated. After fetching, it applies the fetched data directly to the storage engine.

While fetchKeys() is in progress, the storage server might receive new

updates to the keys that fall in the range that's being currently added in fetchKeys(). fetchKeys() keeps track of them separately, outside the storageServer while it's in progress. After fetchKeys() is done fetching and applying the new key-range to the storage engine, it waits for durablization of that data.

And then, it chooses the **transferredVersion** that ensures that * The transferredVersion can be mutated in versionedData * The transferredVersion isn't yet committed to storage (so we can write the availability status change) * The transferredVersion is <= the version of any of the updates in batch, and if there is an equal version, its mutations haven't been processed yet

Then it transfers that list of separately tracked new updates to update() actor to be applied to the storage queue at transferredVersion. There is a complicated song and dance that takes place between the update() and fetchKeys() actors.

The final step in fetchKeys() is to wait until the transferredVersion gets durablized. This essentially means waiting until the new shard data gets committed and durable.

Just a side note that the applyMutation() code is re-used, to both apply the mutation to the storage server, as well as track the newly coming updates while fetchKeys() is in progress. This happens based on the shard state. If the shard is in "readWrite" state, the mutation is applied to the storage server, else if the shard is in "Adding" state, the mutation is added to tracked shard state.