Overview

This documentation gives a short introduction into how watches are implemented in FoundationDB and what kind of guarantees a client is getting. Furthermore, this document should give some insight into back practices.

An equivalent Functionality implemented in the Client

Understanding how watches are implemented is valuable to understand its performance implications. But in order to use watches correctly, one needs to understand the guarantees that watches provide. Understanding this part is easier if you think of watches as an optimization for a client feature.

Let's assume you want to be informed whenever the value for a certain key changes. You could do this without a watch by polling:

```
@fdb.transactional
def key_changed(tr, key, value):
    v = tr[key]
    if v ≠ value:
        return True
    return False

def watch_key(db, key, value, watch_sleep_time):
    while not key_changed(db, key, value):
        time.sleep(watch_sleep_time)
```

So this will just query the value in a loop and return as soon as the value changed (and keeps changed long enough for it to be observable). If you use this method you will quickly find that finding a good value for `watch_sleep_time` is very hard: if the value is very small each watch will put significant load onto the storages within the cluster. If the value is too large, it will take much longer until a change is observed.

It is important to understand that the only thing that database watches solve is that one can basically set this hypothetic `watch_sleep_time` parameter to `0` without paying for the polling. Or phrased differently: think of watches as an optimization rather than a feature.

Implementation Overview

In general there are two parts of this implementation: one part is implemented in the client and one is implemented on the server side. These play together to achieve the right functionality.

Watches are registered through a transaction and are registered with a storage server. The storage will keep all watches in a map internally. For each mutation that is executed on the storage, it will then look up corresponding

watches from that map and, if there are any, will send the version at which the change was made back to the client.

Since the initial implementation of watches there have been a few optimizations that were done. The first is on the client side, where watches are tracked for a given key and a request is only sent to the storage server if one has not been sent for a previous watch with the same key/value pair. The second optimization is on the server, where the duplication of key/value pairs is reduced by collapsing the actors for same key/value watches.

Guarantees

A watch might get notified on changes. However, you can miss changes for two reasons: * The value changed more than once while the client or a storage server was in some faulty state and the watch had to be reregistered. In this case the watch might only fire once. * Watches are subjected to the ABA-problem: if a value changes twice and after the second update it has the same value as before, the client might not be notified of that update.

Failure Handling

Client Failures

If a client fails, the storage will keep that watch until it will be triggered the next time as the storage server will not be made aware of the fact that the client died. This is problematic, as a client could watch a key that never changes. To handle this case the storage server will eventually time out any watch that didn't see any writes for a long time (usually after ~15 minutes). If the client didn't fail, it will receive a timeout from the server and can reregister the watch (it will do so automatically).

Server Failures

Watches on storage servers are ephemeral. So if a storage server fails, it will lose all current watches. While clients are waiting for a watch reply, they will ping the server (by default once a second) - if this times out they will automatically reregister the watch on a different server.

Usage

The client side code for registering a watch and using the watch is often in this pattern:

```
 // First transaction waits for a change to a key
state Reference<ReadYourWritesTransaction> tr(new ReadYourWritesTransaction(cx));
 state Future<Void> watchFuture;
 loop {
     try {
         ... // Optional<Value> value = wait(tr->get(someKey));

         watch = tr->watch(watchKey);
         wait(tr->commit());
         wait(watchFuture);
         break;
     } catch (Error& e) {
         wait(tr->onError(e));
     }
 }

 // Second transaction reads the key
 tr->reset();
 loop {
     try {
         // Optional<Value> value = wait(tr->get(watchKey));
         ...
         wait(tr->commit());
         break;
     } catch (Error& e) {
         wait(tr->onError(e));
     }
 }
```

Note the first transaction `wait` on the transaction commit and then `wait` on the `watchFuture`. Specifically, the `tr->commit()` in the first transaction sets up the `watchFuture` to receive changes from the committed version of the first transaction. As a result, if another transaction `Tx` modifies the `watchKey` after the first transactions, the above pattern guarantees that when `wait(watchFuture)` returns, `Tx` has already committed and the second transaction runs after it (thus will see the effect of `Tx`).