MODULE *ConsistentKV*

This spec is an abstraction of our cache consistency protocol; the interface, along with some additional documentation, can be found in the caching/consistentkv package.

The intent of this protocol is to maintain strong consistency between the underlying database and the cache.

The key idea is the use of a sentinel value, called *Pending*, that prevents cache entries from being populated while a writer is committing to the underlying database.

EXTENDS *Naturals*

CONSTANTS

The server can be in the following three states:

* *Active* servers may be used for all operations
* *SentinelOnly* servers should only be used for operations involving Sentinel values (*e.g.* setting or clearing a *Pending* sentinel)
* *Down* servers may not be used for any operations

Active, *SentinelOnly*, *Down*,

We define the following kinds of processes:

* *Readers* attempt to read from the cache and then populate cache misses
* *Writers* use a two-step protocol to safely invalidate cache entries whenever updating the underlying database
* *Operators* change the state of the server along the following "paths":
  * *Active* → *SentinelOnly* → *Down*
  * *Down* → *SentinelOnly* → Up

Readers, *Writers*, *Operators*,

The set of possible keys and values for entries in the cache and database.

Keys, *Values*,

Sentinel values for entries in the cache that should be treated specially.

* *Deleted* and *Missing* entries are treated as cache misses
* *Pending* entries are treated as cache misses, and also prevent the entry from being populated

Missing means that a key does not appear in the cache, while *Deleted* means that a key is associated with a special "DELETED" value. Because the cache makes extensive use of versions and *CAS* operations, the most important difference between *Missing* and *Deleted* is that *Missing* entries always have version 0, while *Deleted* entries can have any nonzero version.

Deleted, *Missing*, *Pending*

--algorithm *ConsistentKV* **{**

variables

The cache is initially unpopulated.

$cache\_keys\_to\_values = [k \in Keys \mapsto Missing]$,

Missing cache entries have version 0; versions generally increase monotonically, though expiration and server restarts reset may reset entry versions back to 0 (and entry values back to *Missing*).

$cache\_keys\_to\_versions = [k \in Keys \mapsto 0]$;

The *DB* initially maps each key to an arbitrary value.

$db\_keys\_to\_values = [k \in Keys \mapsto \text{CHOOSE } v \in Values : \text{TRUE}]$,

Each Reader and Writer has its own view of the memcache server's current state, and has behavior that depends on this view.

This version of the spec models the behavior of the protocol in the case where the server starts in a *Down* state before transitioning through *SentinelOnly* and *Active*.

$observed\_server\_state = [p \in Readers \cup Writers \mapsto Down]$;

Readers proceed as follows:

* *init_reader* initializes the set of keys to be read
* *get_from_cache* reads existing values and versions from an *Active* server
* *read_from_db* reads cache misses that should be populated (of course the actual code also reads cache misses that don't need to be populated, but the result of those reads are not relevant to this spec so do we not include them)
* *add_to_cache* uses a *CAS* operation to populate cache misses that haven't changed since the previous call to *get_from_cache*

process ($R \in Readers$)
variables
$data\_version\_ids$,
$db\_read\_keys\_to\_values$,
$keys\_to\_populate$,
$keys\_to\_read$,
{
init_reader:
Sets *keys_to_read* to an arbitrary non-empty subset of *Keys*.
with ($keys \in (\text{SUBSET } Keys) \setminus \{\{\}\}$) **{**
$keys\_to\_read := keys$;
} ;

get_from_cache uses *GetMulti* (*i.e.* it only reads from *Active* servers).
get_from_cache:
if ($observed\_server\_state[self] = Active$) **{**
$keys\_to\_populate :=$
$\{k \in keys\_to\_read :$
$\lor\ cache\_keys\_to\_values[k] = Missing$
$\lor\ cache\_keys\_to\_values[k] = Deleted\}$;
$data\_version\_ids :=$
$[k \in keys\_to\_populate \mapsto cache\_keys\_to\_versions[k]]$;
} else {
$keys\_to\_populate := \{\}$;

```
            data_version_ids := ⟨⟩ ;
        } ;
    read_from_db:
        db_read_keys_to_values :=
            [k ∈ keys_to_populate ↦ db_keys_to_values[k]] ;
```

add_to_cache uses *CasMulti* (*i.e.* it only writes to ACTIVE servers).

```
    add_to_cache:
        if ( observed_server_state[self] = Active ) {
            cache_keys_to_values :=
                [k ∈ Keys ↦
                    IF  ∧ k ∈ keys_to_populate
                        ∧ data_version_ids[k] = cache_keys_to_versions[k]
                      THEN db_read_keys_to_values[k]
                      ELSE cache_keys_to_values[k]] ;
            cache_keys_to_versions :=
                [k ∈ Keys ↦
                    IF  ∧ k ∈ keys_to_populate
                        ∧ data_version_ids[k] = cache_keys_to_versions[k]
                      THEN cache_keys_to_versions[k] + 1
                      ELSE cache_keys_to_versions[k]] ;
        } ;
}
```

Writers proceed as follows:

* *init_writer* initializes the set of *keys* → value mappings to be written; we assume that a database transaction has already been started, and that all operations for that transaction, except the final commit, occurred before *init_writer*
* *start_pending* uses the *Pending* sentinel value to prevent cache entries from being populated (with possibly stale values); upon failure or detection of existing *Pending* values, the entire write process (in particular, the database transaction) must be aborted
* *db_commit* (obviously) commits the underlying database transaction and makes the written values available to readers
* *finish_pending* clears out any *Pending* sentinels that were set, so that subsequent reads may populate the cache again

```
process ( W ∈ Writers )
    variables
        cas_error_keys,
        cas_success_keys,
        cas_success_keys_to_versions,
        current_cache_keys_to_values,
        current_cache_keys_to_versions,
        db_write_keys,
        db_write_keys_to_values,
        pending_items,
```

```
{
  init_writer:
    Sets db_write_keys to an arbitrary non-empty subset of Keys.
    with ( keys ∈ (SUBSET Keys) \ {{}} ) {
      db_write_keys := keys ;
    } ;
    Sets db_write_keys_to_values to an arbitrary mapping from keys_to_write to Values.
    with ( keys_to_values ∈ [db_write_keys → Values] ) {
      db_write_keys_to_values := keys_to_values ;
    } ;

  start_pending uses GetSentinels and CasSentinels (i.e. it reads / writes from both Active
  and SentinelOnly servers).
  start_pending:
    skip ;

    get_sentinels:
      if (
        ∨ observed_server_state[self] = Active
        ∨ observed_server_state[self] = SentinelOnly ) {
        current_cache_keys_to_values :=
          [k ∈ db_write_keys ↦ cache_keys_to_values[k]] ;
        current_cache_keys_to_versions :=
          [k ∈ db_write_keys ↦ cache_keys_to_versions[k]] ;
      } else {
        current_cache_keys_to_values := ⟨⟩ ;
        current_cache_keys_to_versions := ⟨⟩ ;
      } ;

    check_already_pending:
      if ( ∃ k ∈ DOMAIN current_cache_keys_to_values :
             current_cache_keys_to_values[k] = Pending ) {
        goto Done ;
      } ;

    cas_sentinels:
      if (
        ∨ observed_server_state[self] = Active
        ∨ observed_server_state[self] = SentinelOnly ) {
        cas_success_keys :=
          {k ∈ DOMAIN current_cache_keys_to_versions :
            current_cache_keys_to_versions[k] = cache_keys_to_versions[k]} ;
        cas_error_keys :=
          {k ∈ DOMAIN current_cache_keys_to_versions :
          ¬k ∈ cas_success_keys} ;
        cache_keys_to_values :=
```

```
                [k ∈ Keys ↦
                  IF k ∈ cas_success_keys
                    THEN Pending
                    ELSE cache_keys_to_values[k]] ;
              cache_keys_to_versions :=
                [k ∈ Keys ↦
                  IF k ∈ cas_success_keys
                    THEN cache_keys_to_versions[k] + 1
                    ELSE cache_keys_to_versions[k]] ;
              cas_success_keys_to_versions :=
                [k ∈ cas_success_keys ↦ cache_keys_to_versions[k]] ;
            } else {
              cas_success_keys := {} ;
              cas_error_keys := {} ;
            } ;

        check_cas_errors:
            if ( cas_error_keys ≠ {} ) {
              goto finish_pending ;
            } ;

    db_commit:
        db_keys_to_values :=
          [k ∈ Keys ↦
            IF k ∈ db_write_keys
              THEN db_write_keys_to_values[k]
              ELSE db_keys_to_values[k]] ;
```

finish_pending uses *SetSentinels* (*i.e.* it writes to both *Active* and *SentinelOnly* servers).

```
    finish_pending:
        if (
            ∨ observed_server_state[self] = Active
            ∨ observed_server_state[self] = SentinelOnly ) {
            cache_keys_to_values :=
              [k ∈ Keys ↦
                IF ∧ k ∈ cas_success_keys
                   ∧ cas_success_keys_to_versions[k] = cache_keys_to_versions[k]
                  THEN Deleted
                  ELSE cache_keys_to_values[k]] ;
            cache_keys_to_versions :=
              [k ∈ Keys ↦
                IF ∧ k ∈ cas_success_keys
                   ∧ cas_success_keys_to_versions[k] = cache_keys_to_versions[k]
                  THEN cache_keys_to_versions[k] + 1
                  ELSE cache_keys_to_versions[k]] ;
        } ;
```

```
    }
```

Operators proceed as follows:

* *to_sentinel_only* changes each client's view of the server from *Down* to *SentinelOnly*, one client at a time
* *to_active* changes each client's view of the server from *SentinelOnly* to *Active*, one client at a time

Note that in this spec, we require that *clients′* views of the server are never more than one "step" apart.

```
    process ( O ∈ Operators )
        variable clients ;
    {
        init_clients_1:
            clients := Readers ∪ Writers ;

        to_sentinel_only:
            while ( clients ≠ {} ) {
                one_client_to_sentinel_only:
                    with ( client ∈ clients ) {
                        observed_server_state[client] := SentinelOnly ;
                        clients := clients \ {client} ;
                    } ;
            } ;

        init_clients_2:
            clients := Readers ∪ Writers ;

        to_active:
            while ( clients ≠ {} ) {
                one_client_to_active:
                    with ( client ∈ clients ) {
                        observed_server_state[client] := Active ;
                        clients := clients \ {client} ;
                    } ;
            } ;
    }
}
```

Consistency ensures that for all keys, the associated value in the cache is either equal to the associated value in the database, or is one of the sentinel values.

$$
\begin{aligned}
&Consistency \triangleq \\
&\quad \forall\, k \in Keys : \\
&\qquad \lor\ cache\_keys\_to\_values[k] = db\_keys\_to\_values[k] \\
&\qquad \lor\ cache\_keys\_to\_values[k] = Deleted \\
&\qquad \lor\ cache\_keys\_to\_values[k] = Missing \\
&\qquad \lor\ cache\_keys\_to\_values[k] = Pending
\end{aligned}
$$

TypeOK provides the following sanity checks:

* *db_keys_to_values* is a mapping from the set of *Keys* to the set of *Values*
* *cache_keys_to_values* is a mapping from the set of *Keys* to the set of *Values*, together with some additional sentinel values
* *cache_keys_to_versions* is a mapping from *Keys* to natural numbers

$$
\begin{aligned}
&\mathit{TypeOK} \triangleq \\
&\quad \land \quad \mathit{db\_keys\_to\_values} \quad \in [\mathit{Keys} \to \mathit{Values}] \\
&\quad \land \quad \mathit{cache\_keys\_to\_values} \quad \in [\mathit{Keys} \to (\mathit{Values} \cup \{\mathit{Deleted}, \mathit{Missing}, \mathit{Pending}\})] \\
&\quad \land \quad \mathit{cache\_keys\_to\_versions} \in [\mathit{Keys} \to \mathit{Nat}]
\end{aligned}
$$

VersionsOK ensures that a key has version 0 if and only if it does not appear in the cache (*i.e.* its value is *Missing*)

$$
\begin{aligned}
&\mathit{VersionsOK} \triangleq \\
&\quad \forall\, k \in \mathit{Keys} : \\
&\qquad \mathit{cache\_keys\_to\_values}[k] = \mathit{Missing} \equiv \mathit{cache\_keys\_to\_versions}[k] = 0
\end{aligned}
$$

* Modification History
* Last modified *Mon Aug* 01 09:12:30 *PDT* 2016 by *elliott*
* Created *Thu Jul* 28 11:36:26 *PDT* 2016 by *elliott*