
MODULE *MapCache*

EXTENDS *Naturals, FiniteSets, Sequences, TLC*

An empty value
 CONSTANT *Nil*

The set of clients to model
 CONSTANT *Client*

The set of possible keys in the map
 CONSTANT *Key*

The set of possible values in the map
 CONSTANT *Value*

An update entry identifier
 CONSTANT *Update*

A tombstone entry identifier
 CONSTANT *Tombstone*

*The system state, modelled as a strongly consistent consensus service
 using a single mapping of $key \rightarrow value$ pairs*
 VARIABLE *state*

*A sequential version number, used by the consensus service to assign logical
 timestamps to entries*
 VARIABLE *stateVersion*

The cache state
 VARIABLE *cache*

The maximum version propagated to the cache
 VARIABLE *cacheVersion*

An unordered bag of pending cache entries
 VARIABLE *cachePending*

A strongly ordered sequence of update events
 VARIABLE *events*

The history of operations
 VARIABLE *history*

INSTANCE *MapHistory* WITH *history* \leftarrow *history*

vars \triangleq $\langle state, stateVersion, cache, cachePending, cacheVersion, events, history \rangle$

This section models helpers for managing the system and cache state

Drop a key from the domain of a function

$$\text{DropKey}(s, k) \triangleq [i \in \text{DOMAIN } s \setminus \{k\} \mapsto s[i]]$$

Put an entry in the given function

$$\begin{aligned} \text{PutEntry}(s, e) &\triangleq \\ &\text{IF } e.\text{key} \in \text{DOMAIN } s \text{ THEN} \\ &\quad [s \text{ EXCEPT } ![e.\text{key}] = e] \\ &\text{ELSE} \\ &\quad s @@@ (e.\text{key} :> e) \end{aligned}$$

This section models the map cache. When a client updates the map, it defers updates to the cache to be performed in a separate step. The cache also listens for events coming from a consensus service, which must provide sequentially consistent event streams. Entries are arbitrarily evicted from the cache.

Defer an entry 'e' to be cached asynchronously on client 'c'

Updates are deferred in no particular order to model the potential reordering of concurrent thhistory by the operating system.

$$\begin{aligned} \text{DeferCache}(c, e) &\triangleq \\ \text{cachePending}' &= [\text{cachePending} \text{ EXCEPT } ![c] = \text{cachePending}[c] @@@ (e.\text{version} :> e)] \end{aligned}$$

Remove a deferred entry 'e' from cache deferrals for client 'c'

$$\begin{aligned} \text{RemoveCacheDeferral}(c, e) &\triangleq \\ \text{cachePending}' &= [\text{cachePending} \text{ EXCEPT } ![c] = \\ &\quad [v \in \text{DOMAIN } \text{cachePending}[c] \setminus \{e.\text{version}\} \mapsto \text{cachePending}[c][v]]] \end{aligned}$$

Cache an entry 'e' on client 'c'

The entry is read from the pending cache entries. An entry will only be updated in the cache if the entry version is greater than the cache propagation version, ensuring the cache cannot go back in time.

Note that removals are inserted into the cache as tombstones to be removed once updates have been propagated via event queues.

$$\begin{aligned} \text{Cache}(c, e) &\triangleq \\ &\wedge \text{LET } \text{entry} \triangleq \text{cachePending}[c][e] \\ &\text{IN} \\ &\quad \wedge \vee \wedge \text{entry.version} > \text{cacheVersion}[c] \\ &\quad \wedge \vee \text{entry.key} \notin \text{DOMAIN } \text{cache}[c] \\ &\quad \vee \wedge \text{entry.key} \in \text{DOMAIN } \text{cache}[c] \\ &\quad \quad \wedge \text{entry.version} > \text{cache}[c][\text{entry.key}].\text{version} \\ &\quad \wedge \text{cache}' = [\text{cache} \text{ EXCEPT } ![c] = \text{PutEntry}(\text{cache}[c], \text{entry})] \\ &\quad \wedge \text{Record}(c, \text{entry.key}, \text{entry.version}) \\ &\quad \vee \wedge \vee \text{entry.version} \leq \text{cacheVersion}[c] \\ &\quad \quad \vee \wedge \text{entry.key} \in \text{DOMAIN } \text{cache}[c] \\ &\quad \quad \quad \wedge \text{entry.version} \leq \text{cache}[c][\text{entry.key}].\text{version} \\ &\quad \wedge \text{UNCHANGED } \langle \text{cache}, \text{history} \rangle \end{aligned}$$

$\wedge \text{RemoveCacheDeferral}(c, \text{entry})$
 $\wedge \text{UNCHANGED } \langle \text{state}, \text{stateVersion}, \text{cacheVersion}, \text{events} \rangle$

Enqueue a cache update event 'e' for all clients

Events are guaranteed to be delivered to clients in the order in which they occurred in the consensus layer, so we model events as a simple strongly ordered sequence.

$\text{EnqueueEvent}(e) \triangleq$
 $\text{events}' = [i \in \text{Client} \mapsto \text{Append}(\text{events}[i], e)]$

Learn a map update from the event queue of 'c'

The learner learns the first entry in the event queue for client 'c'.

If the key is already in the cache, the learner updates the key only if the update version is at least as great as the cached version.

If the key is not present in the map, the entry is cached.

Tombstone types are removed from the cache. Entry types are inserted.

Once caching is complete, the 'cacheVersion' is updated to ensure the deferred cache remains consistent.

$\text{Learn}(c) \triangleq$
 $\wedge \text{Cardinality}(\text{DOMAIN } \text{events}[c]) > 0$
 $\wedge \text{LET } \text{entry} \triangleq \text{events}[c][1]$
 IN
 $\wedge \vee \wedge \text{entry.key} \in \text{DOMAIN } \text{cache}[c]$
 $\wedge \text{entry.version} \geq \text{cache}[c][\text{entry.key}].\text{version}$
 $\wedge \vee \wedge \text{entry.type} = \text{Update}$
 $\wedge \text{cache}' = [\text{cache} \text{ EXCEPT } ![c] = \text{PutEntry}(\text{cache}[c], \text{entry})]$
 $\vee \wedge \text{entry.type} = \text{Tombstone}$
 $\wedge \text{cache}' = [\text{cache} \text{ EXCEPT } ![c] = \text{DropKey}(\text{cache}[c], \text{entry.key})]$
 $\vee \wedge \vee \text{entry.key} \notin \text{DOMAIN } \text{cache}[c]$
 $\vee \wedge \text{entry.key} \in \text{DOMAIN } \text{cache}[c]$
 $\wedge \text{entry.version} < \text{cache}[c][\text{entry.key}].\text{version}$
 $\wedge \text{UNCHANGED } \langle \text{cache} \rangle$
 $\wedge \text{cacheVersion}' = [\text{cacheVersion} \text{ EXCEPT } ![c] = \text{entry.version}]$
 $\wedge \text{events}' = [\text{events} \text{ EXCEPT } ![c] = \text{SubSeq}(\text{events}[c], 2, \text{Len}(\text{events}[c]))]$
 $\wedge \text{UNCHANGED } \langle \text{state}, \text{stateVersion}, \text{cachePending}, \text{history} \rangle$

Evict a map key 'k' from the cache of client 'c'

To preserve consistency, each key for each client must be retained until updates prior to the key version have been propagated to the client. If keys are evicted before updates have been propagated to the cache, evicting a key can allow a concurrent *Put* or *Remove* to cache an older entry.

$\text{Evict}(c, k) \triangleq$
 $\wedge k \in \text{DOMAIN } \text{cache}[c]$
 $\wedge \text{cache}[c][k].\text{version} \leq \text{cacheVersion}[c]$
 $\wedge \text{cache}' = [\text{cache} \text{ EXCEPT } ![c] = \text{DropKey}(\text{cache}[c], k)]$
 $\wedge \text{UNCHANGED } \langle \text{state}, \text{stateVersion}, \text{cachePending}, \text{cacheVersion}, \text{events}, \text{history} \rangle$

This section models the method calls for the Map primitive. Map entries can be created, updated, deleted, and read. The *Put* and *Remove* steps model writes to a consensus service. Steps do not atomically cache reads/updates but instead defer them to be cached in a separate step. This models the reordering of threads by the OS.

Get an entry for key 'k' in the map on client 'c'
 If the key is present in the cache, read from the cache.
 If the key is not present in the cache, read from the system state and update the cache if the system entry version is greater than the cache version.
 If the key is neither present in the cache or the system state, read the cache version.
 If the step reads from the cache, the cached version is added to the history. Otherwise, if the step reads from the consensus service, the cache is updated in a separate step before recording the value.

$$\begin{aligned}
 \text{Get}(c, k) &\triangleq \\
 &\wedge \vee \wedge k \in \text{DOMAIN } \text{cache}[c] \\
 &\quad \wedge \text{Record}(c, k, \text{cache}[c][k].\text{version}) \\
 &\quad \wedge \text{UNCHANGED } \langle \text{cachePending} \rangle \\
 &\vee \wedge k \notin \text{DOMAIN } \text{cache}[c] \\
 &\quad \wedge k \in \text{DOMAIN } \text{state} \\
 &\quad \wedge \text{DeferCache}(c, \text{state}[k]) \\
 &\quad \wedge \text{UNCHANGED } \langle \text{history} \rangle \\
 &\vee \wedge k \notin \text{DOMAIN } \text{cache}[c] \\
 &\quad \wedge k \notin \text{DOMAIN } \text{state} \\
 &\quad \wedge \text{DeferCache}(c, [\text{type} \mapsto \text{Tombstone}, \\
 &\quad \quad \text{key} \mapsto k, \\
 &\quad \quad \text{value} \mapsto \text{Nil}, \\
 &\quad \quad \text{version} \mapsto \text{stateVersion}]) \\
 &\quad \wedge \text{UNCHANGED } \langle \text{history} \rangle \\
 &\wedge \text{UNCHANGED } \langle \text{state}, \text{stateVersion}, \text{cache}, \text{cacheVersion}, \text{events} \rangle
 \end{aligned}$$

Put key 'k' and value 'v' pair in the map on client 'c'
 Increment the system state version and insert the entry into the system state.
 Enqueue update events to notify all clients and defer a local cache update to client 'c'.

$$\begin{aligned}
 \text{Put}(c, k, v) &\triangleq \\
 &\wedge \text{stateVersion}' = \text{stateVersion} + 1 \\
 &\wedge \text{LET } \text{entry} \triangleq [\text{type} \mapsto \text{Update}, \text{key} \mapsto k, \text{value} \mapsto v, \text{version} \mapsto \text{stateVersion}'] \\
 &\text{IN} \\
 &\quad \wedge \text{state}' = \text{PutEntry}(\text{state}, \text{entry}) \\
 &\quad \wedge \text{EnqueueEvent}(\text{entry}) \\
 &\quad \wedge \text{DeferCache}(c, \text{entry}) \\
 &\wedge \text{UNCHANGED } \langle \text{cache}, \text{cacheVersion}, \text{history} \rangle
 \end{aligned}$$

Remove key 'k' from the map on client 'c'
 Increment the system state version and remove the entry from the system state.

Enqueue tombstone events to notify all clients and defer a local cache update
to client 'c'.

$$\begin{aligned}
\text{Remove}(c, k) &\triangleq \\
&\wedge k \in \text{DOMAIN } \text{state} \\
&\wedge \text{stateVersion}' = \text{stateVersion} + 1 \\
&\wedge \text{LET } \text{entry} \triangleq [\text{type} \mapsto \text{Tombstone}, \text{key} \mapsto k, \text{value} \mapsto \text{Nil}, \text{version} \mapsto \text{stateVersion}'] \\
&\text{IN} \\
&\quad \wedge \text{state}' = \text{DropKey}(\text{state}, k) \\
&\quad \wedge \text{EnqueueEvent}(\text{entry}) \\
&\quad \wedge \text{DeferCache}(c, \text{entry}) \\
&\wedge \text{UNCHANGED } \langle \text{cache}, \text{cacheVersion}, \text{history} \rangle
\end{aligned}$$

$$\begin{aligned}
\text{Init} &\triangleq \\
&\wedge \text{LET } \text{nilEntry} \triangleq [\text{type} \mapsto \text{Nil}, \\
&\quad \text{key} \mapsto \text{Nil}, \\
&\quad \text{value} \mapsto \text{Nil}, \\
&\quad \text{version} \mapsto \text{Nil}] \\
&\text{IN} \\
&\quad \wedge \text{state} = [i \in \{\} \mapsto \text{nilEntry}] \\
&\quad \wedge \text{stateVersion} = 0 \\
&\quad \wedge \text{cache} = [c \in \text{Client} \mapsto [i \in \{\} \mapsto \text{nilEntry}]] \\
&\quad \wedge \text{cachePending} = [c \in \text{Client} \mapsto [i \in \{\} \mapsto \text{nilEntry}]] \\
&\quad \wedge \text{cacheVersion} = [c \in \text{Client} \mapsto 0] \\
&\quad \wedge \text{events} = [c \in \text{Client} \mapsto [i \in \{\} \mapsto \text{nilEntry}]] \\
&\wedge \text{history} = [c \in \text{Client} \mapsto [k \in \text{Key} \mapsto \langle \rangle]]
\end{aligned}$$

$$\begin{aligned}
\text{Next} &\triangleq \\
&\vee \exists c \in \text{Client} : \\
&\quad \exists k \in \text{Key} : \\
&\quad \quad \text{Get}(c, k) \\
&\vee \exists c \in \text{Client} : \\
&\quad \exists k \in \text{Key} : \\
&\quad \quad \exists v \in \text{Value} : \\
&\quad \quad \quad \text{Put}(c, k, v) \\
&\vee \exists c \in \text{Client} : \\
&\quad \exists k \in \text{Key} : \\
&\quad \quad \text{Remove}(c, k) \\
&\vee \exists c \in \text{Client} : \\
&\quad \exists e \in \text{DOMAIN } \text{cachePending}[c] : \\
&\quad \quad \text{Cache}(c, e) \\
&\vee \exists c \in \text{Client} : \\
&\quad \text{Learn}(c) \\
&\vee \exists c \in \text{Client} : \\
&\quad \exists k \in \text{Key} :
\end{aligned}$$

$Evict(c, k)$

$Spec \stackrel{\Delta}{=} Init \wedge \Box[Next]_{\langle vars \rangle}$

* Modification History

* Last modified Sun Feb 16 17:55:24 PST 2020 by jordanhalterman

* Created Mon Feb 10 23:01:48 PST 2020 by jordanhalterman