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 $key \to value$ 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 reads for the client, used by the model checker to verify sequential consistency
VARIABLE *reads*

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

The type invariant checks that the client's reads never go back in time

$$
\begin{array}{l}
\mathit{TypeInvariant} \triangleq \\
\quad \land \forall\, c \in \mathit{Client} : \\
\qquad \land \forall\, k \in \mathit{Key} : \\
\qquad\quad \land \forall\, r \in \text{DOMAIN } \mathit{reads}[c][k] : \\
\qquad\qquad r > 1 \Rightarrow \mathit{reads}[c][k][r] \geq \mathit{reads}[c][k][r-1]
\end{array}
$$

This section models helpers for managing the system and cache state

Drop a key from the domain of a function

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

Put an entry in the given function

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

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 threads by the operating system.

$$
\begin{array}{l}
\mathit{DeferCache}(c,\, e) \triangleq \\
\quad \mathit{cachePending}' = [\mathit{cachePending} \text{ EXCEPT } ![c] = \mathit{cachePending}[c] \mathbin{@@} (e.\mathit{version} :> e)]
\end{array}
$$

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

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

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.

$$
\begin{array}{l}
\mathit{Cache}(c,\, e) \triangleq \\
\quad \land \text{LET } \mathit{entry} \triangleq \mathit{cachePending}[c][e] \\
\qquad \text{IN} \\
\qquad\quad \land \lor \land \mathit{entry}.\mathit{version} > \mathit{cacheVersion}[c] \\
\qquad\qquad\quad \land \lor \mathit{entry}.\mathit{key} \notin \text{DOMAIN } \mathit{cache}[c]
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad \lor \land\ entry.key \in \textsc{domain}\ cache[c] \\
\qquad\qquad\qquad\quad \land\ entry.version > cache[c][entry.key].version \\
\qquad\qquad \land\ cache' = [cache\ \textsc{except}\ ![c] = PutEntry(cache[c],\ entry)] \\
\qquad \lor \land \lor\ entry.version \leq cacheVersion[c] \\
\qquad\qquad\qquad \lor \land\ entry.key \in \textsc{domain}\ cache[c] \\
\qquad\qquad\qquad\quad \land\ entry.version \leq cache[c][entry.key].version \\
\qquad\qquad \land\ \textsc{unchanged}\ \langle cache \rangle \\
\quad \land\ RemoveCacheDeferral(c,\ entry) \\
\land\ \textsc{unchanged}\ \langle state,\ stateVersion,\ cacheVersion,\ events,\ reads \rangle
\end{array}
$$

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.

$$
\begin{array}{l}
EnqueueEvent(e) \triangleq \\
\quad events' = [i \in Client \mapsto Append(events[i],\ e)]
\end{array}
$$

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.

$$
\begin{array}{l}
Learn(c) \triangleq \\
\quad \land\ Cardinality(\textsc{domain}\ events[c]) > 0 \\
\quad \land\ \textsc{let}\ entry \triangleq events[c][1] \\
\qquad \textsc{in} \\
\qquad\quad \land \lor \land\ entry.key \in \textsc{domain}\ cache[c] \\
\qquad\qquad\qquad \land\ entry.version \geq cache[c][entry.key].version \\
\qquad\qquad\qquad \land \lor \land\ entry.type = Update \\
\qquad\qquad\qquad\qquad\quad \land\ cache' = [cache\ \textsc{except}\ ![c] = PutEntry(cache[c],\ entry)] \\
\qquad\qquad\qquad\quad \lor \land\ entry.type = Tombstone \\
\qquad\qquad\qquad\qquad\quad \land\ cache' = [cache\ \textsc{except}\ ![c] = DropKey(cache[c],\ entry.key)] \\
\qquad\qquad \lor \land \lor\ entry.key \notin \textsc{domain}\ cache[c] \\
\qquad\qquad\qquad\quad \lor \land\ entry.key \in \textsc{domain}\ cache[c] \\
\qquad\qquad\qquad\qquad \land\ entry.version < cache[c][entry.key].version \\
\qquad\qquad\qquad \land\ \textsc{unchanged}\ \langle cache \rangle \\
\qquad\quad \land\ cacheVersion' = [cacheVersion\ \textsc{except}\ ![c] = entry.version] \\
\quad \land\ events' = [events\ \textsc{except}\ ![c] = SubSeq(events[c],\ 2,\ Len(events[c]))] \\
\quad \land\ \textsc{unchanged}\ \langle state,\ stateVersion,\ cachePending,\ reads \rangle
\end{array}
$$

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.

$$
\begin{array}{l}
Evict(c,\ k)\ \triangleq \\
\quad \land\ k \in \text{DOMAIN}\ cache[c] \\
\quad \land\ cache[c][k].version \leq cacheVersion[c] \\
\quad \land\ cache' = [cache\ \text{EXCEPT}\ ![c] = DropKey(cache[c],\ k)] \\
\quad \land\ \text{UNCHANGED}\ \langle state,\ stateVersion,\ cachePending,\ cacheVersion,\ events,\ reads \rangle
\end{array}
$$

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. When the map is updated, write steps do not atomically cache updates but instead defer them to be cached in a separate step. This models the reordering of threads by the *OS*. The *Get* step models a read of either the cache or a consensus service. When reading from the consensus service, the *Get* step does cache entries atomically since reads can be reordered without side effects.

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.
Note that the state is read and the cache is updated in a single step. To guarantee consistency, implementations must update the cache before returning.

$$
\begin{array}{l}
Get(c,\ k)\ \triangleq \\
\quad \land\ \lor\ \land\ k \in \text{DOMAIN}\ cache[c] \\
\qquad\quad \land\ reads' = [reads\ \text{EXCEPT}\ ![c][k] = Append(reads[c][k],\ cache[c][k].version)] \\
\qquad\quad \land\ \text{UNCHANGED}\ \langle cache \rangle \\
\qquad \lor\ \land\ k \notin \text{DOMAIN}\ cache[c] \\
\qquad\quad \land\ k \in \text{DOMAIN}\ state \\
\qquad\quad \land\ \text{LET}\ entry\ \triangleq\ state[k] \\
\qquad\quad\ \ \text{IN} \\
\qquad\quad\quad \land\ cache' = [cache\ \text{EXCEPT}\ ![c] = PutEntry(cache[c],\ entry)] \\
\qquad\quad\quad \land\ reads' = [reads\ \text{EXCEPT}\ ![c][k] = Append(reads[c][k],\ state[k].version)] \\
\qquad \lor\ \land\ k \notin \text{DOMAIN}\ cache[c] \\
\qquad\quad \land\ k \notin \text{DOMAIN}\ state \\
\qquad\quad \land\ \text{LET}\ entry\ \triangleq\ [type \mapsto Tombstone, \\
\qquad\quad\qquad\qquad\qquad\ key \mapsto k, \\
\qquad\quad\qquad\qquad\qquad\ value \mapsto Nil, \\
\qquad\quad\qquad\qquad\qquad\ version \mapsto stateVersion] \\
\qquad\quad\ \ \text{IN} \\
\qquad\quad\quad cache' = [cache\ \text{EXCEPT}\ ![c] = PutEntry(cache[c],\ entry)] \\
\qquad\quad \land\ reads' = [reads\ \text{EXCEPT}\ ![c][k]\ \ = Append(reads[c][k],\ stateVersion)] \\
\quad \land\ \text{UNCHANGED}\ \langle state,\ stateVersion,\ cachePending,\ cacheVersion,\ events \rangle
\end{array}
$$

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'.

$Put(c,\ k,\ v) \triangleq$
$\land\ stateVersion' = stateVersion + 1$
$\land$ LET $entry \triangleq [type \mapsto Update,\ key \mapsto k,\ value \mapsto v,\ version \mapsto stateVersion']$
IN
$\land\ state' = PutEntry(state,\ entry)$
$\land\ EnqueueEvent(entry)$
$\land\ DeferCache(c,\ entry)$
$\land$ UNCHANGED $\langle cache,\ cacheVersion,\ reads\rangle$

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'.

$Remove(c,\ k) \triangleq$
$\land\ k \in$ DOMAIN $state$
$\land\ stateVersion' = stateVersion + 1$
$\land$ LET $entry \triangleq [type \mapsto Tombstone,\ key \mapsto k,\ value \mapsto Nil,\ version \mapsto stateVersion']$
IN
$\land\ state' = DropKey(state,\ k)$
$\land\ EnqueueEvent(entry)$
$\land\ DeferCache(c,\ entry)$
$\land$ UNCHANGED $\langle cache,\ cacheVersion,\ reads\rangle$

$Init \triangleq$
$\land$ LET $nilEntry \triangleq [type \mapsto Nil,$
$key \mapsto Nil,$
$value \mapsto Nil,$
$version \mapsto Nil]$
IN
$\land\ state = [i \in \{\} \mapsto nilEntry]$
$\land\ stateVersion = 0$
$\land\ cache = [c \in Client \mapsto [i \in \{\} \mapsto nilEntry]]$
$\land\ cachePending = [c \in Client \mapsto [i \in \{\} \mapsto nilEntry]]$
$\land\ cacheVersion = [c \in Client \mapsto 0]$
$\land\ events = [c \in Client \mapsto [i \in \{\} \mapsto nilEntry]]$
$\land\ reads = [c \in Client \mapsto [k \in Key \mapsto \langle\rangle]]$

$Next \triangleq$
$\lor \exists\, c \in Client :$
$\exists\, k \in Key :$
$Get(c,\ k)$
$\lor \exists\, c \in Client :$
$\exists\, k \in Key :$

$$
\begin{array}{l}
\quad \exists\, v \in \mathit{Value} : \\
\qquad \mathit{Put}(c,\, k,\, v) \\
\lor\, \exists\, c \in \mathit{Client} : \\
\quad \exists\, k \in \mathit{Key} : \\
\qquad \mathit{Remove}(c,\, k) \\
\lor\, \exists\, c \in \mathit{Client} : \\
\quad \exists\, e \in \text{DOMAIN } \mathit{cachePending}[c] : \\
\qquad \mathit{Cache}(c,\, e) \\
\lor\, \exists\, c \in \mathit{Client} : \\
\quad \mathit{Learn}(c) \\
\lor\, \exists\, c \in \mathit{Client} : \\
\quad \exists\, k \in \mathit{Key} : \\
\qquad \mathit{Evict}(c,\, k)
\end{array}
$$

$$\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\langle \mathit{vars} \rangle}$$

* Modification History
* Last modified *Wed Feb* 12 16:52:38 *PST* 2020 by *jordanhalterman*
* Created *Mon Feb* 10 23:01:48 *PST* 2020 by *jordanhalterman*