MODULE *MapCache*

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *TLC*

An empty value
CONSTANT *Nil*

The set of clients
CONSTANT *Client*

The set of possible keys
CONSTANT *Key*

The set of possible values
CONSTANT *Value*

An update entry type
CONSTANT *Update*

A tombstone entry type
CONSTANT *Tombstone*

The system state
VARIABLE *state*

The maximum version assigned to an event
VARIABLE *stateVersion*

The cache state
VARIABLE *cache*

The maximum version propagated to the cache
VARIABLE *cacheVersion*

A bag of pending cache entries
VARIABLE *cachePending*

A 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
$TypeInvariant \triangleq$
$\quad \land \forall\, c \in Client :$

$$\land \forall\, k \in Key :$$
$$\quad \land \forall\, r \in \text{DOMAIN } reads[c][k] :$$
$$\qquad r > 1 \Rightarrow reads[c][k][r] \geq reads[c][k][r-1]$$

This section models helpers for managing the system and cache state

Drop a key from the domain of a function

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

Put an entry in the given function

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

This section models the cache. When a client updates the map, it defers updates to the cache to be performed asynchronously. The cache also listens for events coming from other clients.

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.

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

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.

$$Cache(c,\, e) \triangleq$$
$$\land \text{LET } entry \triangleq cachePending[c][e]$$
$$\quad \text{IN}$$
$$\qquad \land \lor \land entry.version > cacheVersion[c]$$
$$\qquad\quad \land \lor entry.key \notin \text{DOMAIN } cache[c]$$
$$\qquad\qquad \lor \land entry.key \in \text{DOMAIN } cache[c]$$
$$\qquad\qquad\quad \land entry.version > cache[c][entry.key].version$$
$$\qquad\quad \land cache' = [cache \text{ EXCEPT } ![c] = PutEntry(cache[c],\, entry)]$$
$$\qquad\quad \lor \land \lor entry.version \leq cacheVersion[c]$$
$$\qquad\qquad \lor \land entry.key \in \text{DOMAIN } cache[c]$$
$$\qquad\qquad\quad \land entry.version \leq cache[c][entry.key].version$$
$$\qquad\quad \land \text{UNCHANGED } \langle cache \rangle$$
$$\qquad \land cachePending' = [cachePending \text{ EXCEPT } ![c] = [v \in \text{DOMAIN } cachePending[c] \setminus \{entry.version\} \mapsto$$
$$\land \text{UNCHANGED } \langle state,\, stateVersion,\, cacheVersion,\, events,\, reads \rangle$$

Enqueue a cache update event 'e' for all clients

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

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.

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

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

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

This section models the method calls for the Map primitive. Map entries can be created, updated, deleted, and read. When the map state is changed, events are enqueued for the client, and the learner updates the cache.

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.

```
Get(c, k) ≜
    ∧ ∨ ∧ k ∈ DOMAIN cache[c]
        ∧ reads' = [reads EXCEPT ![c][k] = Append(reads[c][k], cache[c][k].version)]
        ∧ UNCHANGED ⟨cache⟩
      ∨ ∧ k ∉ DOMAIN cache[c]
        ∧ k ∈ DOMAIN state
        ∧ LET entry ≜ state[k]
          IN
            ∧ ∨ ∧ entry.version > cacheVersion[c]
                ∧ cache' = [cache EXCEPT ![c] = PutEntry(cache[c], entry)]
              ∨ ∧ entry.version ≤ cacheVersion[c]
                ∧ UNCHANGED ⟨cache⟩
            ∧ reads' = [reads EXCEPT ![c][k] = Append(reads[c][k], state[k].version)]
      ∨ ∧ k ∉ DOMAIN cache[c]
        ∧ k ∉ DOMAIN state
        ∧ reads' = [reads EXCEPT ![c][k] = Append(reads[c][k], cacheVersion[c])]
        ∧ UNCHANGED ⟨cache⟩
    ∧ UNCHANGED ⟨state, stateVersion, cachePending, cacheVersion, events⟩
```

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) ≜
    ∧ stateVersion' = stateVersion + 1
    ∧ LET entry ≜ [type ↦ Update, key ↦ k, value ↦ v, version ↦ stateVersion']
      IN
        ∧ state' = PutEntry(state, entry)
        ∧ EnqueueEvent(entry)
        ∧ DeferCache(c, entry)
    ∧ UNCHANGED ⟨cache, cacheVersion, reads⟩
```

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) ≜
    ∧ k ∈ DOMAIN state
    ∧ stateVersion' = stateVersion + 1
    ∧ LET entry ≜ [type ↦ Tombstone, key ↦ k, value ↦ Nil, version ↦ stateVersion']
      IN
        ∧ state' = DropKey(state, k)
        ∧ EnqueueEvent(entry)
        ∧ DeferCache(c, entry)
    ∧ UNCHANGED ⟨cache, cacheVersion, reads⟩
```

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

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

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

* Modification History
* Last modified *Tue Feb* 11 13:30:38 *PST* 2020 by *jordanhalterman*
* Created *Mon Feb* 10 23:01:48 *PST* 2020 by *jordanhalterman*