TLA+ specifications of the consistency guarantees provided by Cosmos DB

Murat Demirbas

[2018-11-01 Thu]

Outline

1. Modeling with TLA+
2. Specifying consistency
3. Single client specifications
4. General specifications

Modeling with TLA+

Devil is in details

Any large software system is prone to:

- corner cases
- failed assumptions
- race conditions
- cascading faults

Modeling saves you time

Model first and debug your design exhaustively

- catch the errors at design time, before it is too late
- achieve clarity of thinking
- communicate the design precisely
- see alternative ways to implement the design

TLA+ for modeling

TLA+ is a formal language for describing and reasoning about distributed and concurrent systems

It is developed by Dr. Leslie Lamport (Turing Award 2013)

It is based on mathematical logic, set theory, temporal logic, and is supported by tool set (TLC model checker)

Modeling with TLA+

The math-based formal language provides modeling at high-level, yet with precision

PlusCal provides ease-of-use and readability

The integrated model checker exhaustively debugs your model to the face of concurrency and failures, and produces counterexamples for your candidate invariants

Invariant-based reasoning

Instead of happy-path operational thinking, invariant-based reasoning focuses on *what needs to go right*?

It avoids the complexities & bugs of operational reasoning for concurrent systems

Invariants are specified as safety and liveness properties

Operationalizing TLA+

(1) Model your system/protocol; figure out the invariant

(2) Verify your model wrt to requirements

(3) When you need to add a "feature" to the system:

- first modify your model
- model check, and debug design problems
- implement the correct design

TLA+ is practical

Microsoft Azure Cosmos DB is a globally distributed database

The replication protocol is modeled in TLA+ and is tested for correctness against failures

Other places TLA+ used include

- multi write region modeling
- CRDT modeling
- lease protocol modeling

Specifying consistency

Specifying consistency

Consistency defines how the distributed database behaves when a client reads/writes while other clients write at different replicas

The tighter the consistency guarantees provided, the easier it gets to program/develop over the distributed database

Cosmos DB consistency

Cosmos DB provides 5 well-defined consistency properties to the clients:

- strong consistency
- bounded staleness
- session consistency
- consistent prefix
- eventual consistency

Cosmos DB consistency

Strong consistency provides linearizability since the reads are guaranteed to return the most recent version of an item

Bounded staleness consistency relaxes this a bit; the reads lag behind the writes by at most K prefixes or T interval

Session consistency provides monotonic reads, monotonic writes, read-your-own-writes, and write-follows-reads

When using consistent prefix reads, the updates returned are some prefix of all the updates, with no gaps

Eventually-consistent reads may return out of order reads, which stabilize to the most recent version only after all the writes quiesces

Single client specifications

Single client writing incremented counter values

```
50  process ( client ∈ Clients )
51  variables
52      m = ⟨⟩ ; op = 0 ; chistory = ⟨0⟩ ; ses = 1 ;
53  {
54   CW: while ( op < MaxNumOp ) {
55          op := op + 1 ;
56          send(Cloud, [type ↦ "Write", dat ↦ op, ses ↦ ses, orig ↦ self]) ;
57   CWA:   receive(m)   ;  Ack
58          ses := m.ses ;
59           read
60   CR:    send(Cloud, [type ↦ Consistency, ses ↦ ses, orig ↦ self]) ;
61   CRA:   receive(m) ;   Reply
62          chistory := Append(chistory, m.dat) ;
63          ses := m.ses ;
64      }
65   }
```

Macros for sending and receiving messages

```
1  ---------- MODULE swscop ----------
2  EXTENDS Naturals, Integers, Sequences, FiniteSets, TLC, Bags
3  CONSTANT NumClients, MaxNumOp, Consistency, K
4  ASSUME Consistency ∈ {"Eventual", "Consistent_Prefix", "Session", "Bounded_Staleness", "Strong"}
5  ASSUME MaxNumOp < 10 ∧ NumClients = 1
6  Cloud ≜ 0
7  Clients ≜ 1 .. NumClients
9  --algorithm swscop{
10 variables
11    chan = [n ∈ 0 .. NumClients ↦ ⟨⟩] ;    FIFO channels

13     network functions
14    macro send( des, msg ) {
15        chan[des] := Append(chan[des], msg) ;
16    }

18    macro receive( msg ) {
19        await Len(chan[self]) > 0 ;
20        msg := Head(chan[self]) ;
21        chan[self] := Tail(chan[self]) ;
22    }
```

Cosmos DB as a single process

```
24      process ( cosmosdb ∈ {Cloud} )
25      variables
26          Database = ⟨0⟩ ; msg = ⟨⟩ ;
27      { D: while ( TRUE ) {
28              receive(msg) ;
29              if ( msg.type = "Write" ) {
30                  Database := Append(Database, msg.dat) ;
31      DW:         send(msg.orig, [type ↦ "Ack", dat ↦ Database[Len(Database)], ses ↦ Len(Database)]) ; }
32          else if ( msg.type = "Eventual" )
33      DE:     with ( k ∈ 1 .. Len(Database) )
34                  send(msg.orig, [type ↦ "Reply", dat ↦ Database[k], ses ↦ k]) ;
35          else if ( msg.type = "Consistent_Prefix" )
36      DP:     with ( k ∈ 1 .. Len(Database) )
37                  send(msg.orig, [type ↦ "Reply", dat ↦ Database[k], ses ↦ k]) ;
38          else if ( msg.type = "Session" )
39      DS:     with ( k ∈ msg.ses .. Len(Database) )
40                  send(msg.orig, [type ↦ "Reply", dat ↦ Database[k], ses ↦ k]) ;
41          else if ( msg.type = "Bounded_Staleness" )
42      DB:     with ( k ∈ (IF Len(Database) > K THEN Len(Database) − K ELSE 1) .. Len(Database) )
43                  send(msg.orig, [type ↦ "Reply", dat ↦ Database[k], ses ↦ k]) ;
44          else if ( msg.type = "Strong" )
45      DG:     with ( k = Len(Database) )
46                  send(msg.orig, [type ↦ "Reply", dat ↦ Database[k], ses ↦ k]) ;
47      }
```

Model checking

192 Invariants for single $client(ID = 1)$ writing with op ++

193 $Eventual \triangleq chistory[1][Len(chistory[1])] \in \{Database[Cloud][i] : i \in 1 \,..\, Len(Database[Cloud])\}$

195 $Consistent\_Prefix \triangleq chistory[1][Len(chistory[1])] \in \{Database[Cloud][i] : i \in 1 \,..\, Len(Database[Cloud])\}$

197 $Session \triangleq pc[1] = \text{“CW”} \Rightarrow chistory[1][Len(chistory[1])] \in \{Database[Cloud][i] :$
198 $i \in ses[1] \,..\, Len(Database[Cloud])\}$

200 $Bounded\_Staleness \triangleq pc[1] = \text{“CW”} \Rightarrow chistory[1][Len(chistory[1])] \in \{Database[Cloud][i] :$
201 $i \in (\text{IF } Len(Database[Cloud]) > K \text{ THEN } Len(Database[Cloud]) - K \text{ ELSE } 1) \,..\, Len(Database[Cloud])\}$

203 $Strong \triangleq pc[1] = \text{“CW”} \Rightarrow chistory[1][Len(chistory[1])] = Database[Cloud][Len(Database[Cloud])]$

205

Single client reading, incrementing, writing-back counter values

In this scenario, the client reads value from the database (with one of the 5 consistency levels configured), increments it, and writes it back

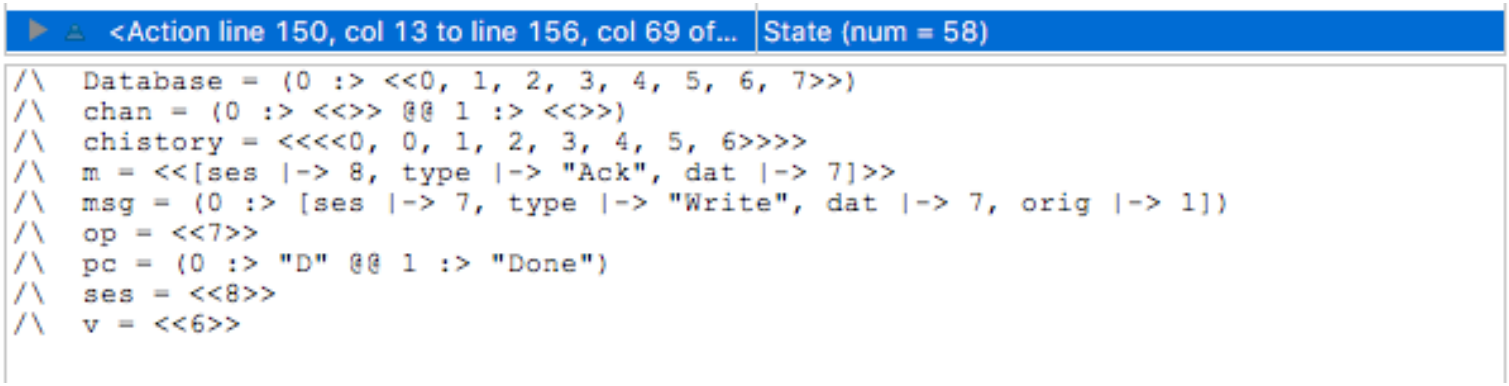
The value stored in the database become consecutive only using the session and strong consistency reads

The client model

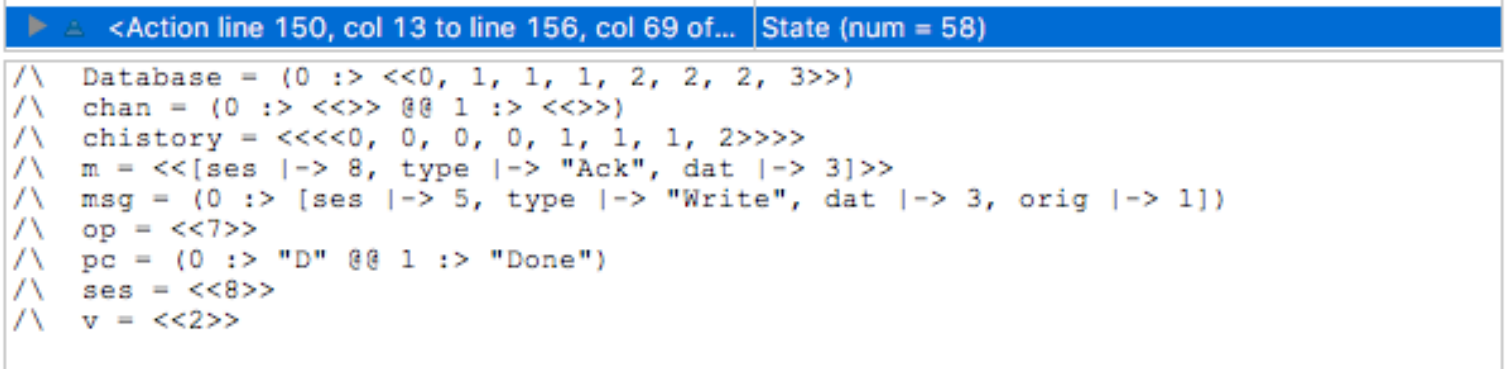
The database model is unchanged; the client is modified

```
50    process ( client ∈ Clients )
51    variables
52        m = ⟨⟩ ; op = 0 ; v = 0 ; chistory = ⟨0⟩ ; ses = 1 ;
53    {
54     CR: while ( op < MaxNumOp ) {
55             send(Cloud, [type ↦ Consistency, ses ↦ ses, orig ↦ self]) ;   read
56     CRA:    receive(m) ;    Reply
57             chistory := Append(chistory, m.dat) ;
58             v := m.dat ;
59             ses := m.ses ;
60              write v + 1
61     CW:     send(Cloud, [type ↦ "Write", dat ↦ v + 1, ses ↦ ses, orig ↦ self]) ;
62     CWA:    receive(m)    ;   Ack
63             ses := m.ses ;
64             op := op + 1 ;
65         }
66     }
```

Strong consistency trace



Bounded consistency trace



General specifications

General specifications

To implement a linearizability checker, the clients share a backchannel: the counter value (or a real time clock)

Multiple clients across different regions put Cosmos DB to test with respect to the consistency guarantees it provides in the presence of concurrent read and writes

For brevity we use shared memory model instead of message passing

Database model exposing regions

The clients perform their writes and reads to their respective local regions

The writes are appended to the `Database[r]` of the corresponding region `r`

The Cosmos DB servers perform *anti-entropy* across the regions and merge their `Database` respecting the happened-before relationship to implement last writer wins

The client

```
142   fair process ( client ∈ Clients )
143   variable session_token = 0 ;
144   numOp = 0 ;
145   {
146       client_actions:
147       while ( numOp < MaxNumOp )
148       {
149           numOp := numOp + 1 ;
150           either
151           {
152               write:
153               value := value + 1 ;
154               write(value) ;
155            }
156           or read: read() ;
157        }
```

The client write operation

```
113   macro write( v )
114   {
115       if ( self[1] ∈ WriteRegions )
116       {
117           when ∀ i, j ∈ Regions : Last(Database[i]) − Last(Database[j]) < Bound ;
118           Database[self[1]] := Append(@, v) ;
119           History := Append(History, [type   ↦ "write",
120                                       data   ↦ v,
121                                       region ↦ self[1],
122                                       client ↦ self]) ;
123           session_token := v ;
124       }
125   }
```

The client read operation

```
128  macro read( )
129  {
130      We check session token for session consistency
131      when Consistency ≠ "session" ∨ Last(Database[self[1]]) ≥ session_token ;
132      We check global value for strong consistency
133      when Consistency ≠ "strong" ∨ ∀ i, j ∈ Regions : Last(Database[i]) = Last(Database[j]) ;
134      History := Append(History, [type   ↦ "read",
135                                  data   ↦ Last(Database[self[1]]),
136                                  region ↦ self[1],
137                                  client ↦ self]) ;
138      session_token := Last(Database[self[1]]) ;
139  }
```

The database

```
164     fair process ( CosmosDB ∈ Servers )
165     {
166         database_action:
167         while ( TRUE )
168         {
169          with ( s ∈ WriteRegions ) {
170             Database[self[1]] := RemoveDuplicates(SortSeq(Database[self[1]] ∘ Database[s], < )) ;
171           }
172          }
173      }
```

Invariant

```
309  Operation in history h is monitonic
310  Monotonic(h) ≜ ∀ i, j ∈ DOMAIN h : i ≤ j ⇒ h[i].data ≤ h[j].data

312  Reads in region r are monotonic
313  MonotonicReadPerRegion(r) ≜ LET reads ≜ [i ∈ {j ∈ DOMAIN History : ∧ History[j].type = "read"
314                                                                     ∧ History[j].region = r}
315                                          ↦ History[i]]
316                              IN  Monotonic(reads)

318  Reads from client c are monotonic
319  MonotonicReadPerClient(c) ≜ LET reads ≜ [i ∈ {j ∈ DOMAIN History : ∧ History[j].type = "read"
320                                                                     ∧ History[j].client = c}
321                                          ↦ History[i]]
322                              IN  Monotonic(reads)

324  MonotonicWritePerRegion(r) ≜ LET writes ≜ [i ∈ {j ∈ DOMAIN History : ∧ History[j].type = "write"
325                                                                      ∧ History[j].region = r}
326                                           ↦ History[i]]
327                               IN  Monotonic(writes)

329  Clients read their own writes
330  ReadYourWrite ≜ ∀ i, j ∈ DOMAIN History : ∧ i < j
331                                            ∧ History[i].type = "write"
332                                            ∧ History[j].type = "read"
333                                            ∧ History[i].client = History[j].client
334                                            ⇒ History[j].data ≥ History[i].data

336  Read the latest writes
337  ReadAfterWrite ≜ ∀ i, j ∈ DOMAIN History : ∧ i < j
338                                             ∧ History[i].type  = "write"
339                                             ∧ History[j].type  = "read"
340                                             ⇒ History[j].data ≥ History[i].data
```

Invariant

$$
\begin{array}{ll}
366 & \mathit{Linearizability} \triangleq \forall\, i, j \in \text{DOMAIN } \mathit{History} : \ \wedge\, i < j \\
367 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \mathit{History}[j].\mathit{data} \geq \mathit{History}[i].\mathit{data} \\
 & \\
369 & \mathit{BoundedStaleness} \triangleq \ \wedge \forall\, i, j \in \mathit{Regions} : \mathit{Last}(\mathit{Database}[i]) - \mathit{Last}(\mathit{Database}[j]) \leq K \\
370 & \qquad\qquad\qquad\qquad\quad \wedge \forall\, r \in \mathit{Regions} : \mathit{MonotonicReadPerRegion}(r) \\
371 & \qquad\qquad\qquad\qquad\quad \wedge \mathit{ReadYourWrite} \\
 & \\
373 & \mathit{ConsistentPrefix} \triangleq \forall\, r \in \mathit{Regions} : \ \wedge \mathit{MonotonicWritePerRegion}(r) \\
374 & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge \mathit{AnyReadPerRegion}(r) \\
 & \\
376 & \mathit{Strong} \triangleq \ \wedge \mathit{Linearizability} \\
377 & \qquad\quad \wedge \mathit{Monotonic}(\mathit{History}) \\
378 & \qquad\quad \wedge \mathit{ReadAfterWrite} \\
 & \\
380 & \mathit{Session} \triangleq \ \wedge \forall\, c \in \mathit{Clients} : \mathit{MonotonicReadPerClient}(c) \\
381 & \qquad\quad \wedge \mathit{ReadYourWrite} \\
 & \\
383 & \mathit{Eventual} \triangleq \forall\, i \in \text{DOMAIN } \mathit{History} : \\
384 & \qquad\quad \text{LET } r \triangleq \mathit{History}[i].\mathit{region} \\
385 & \qquad\quad \text{IN } \ \mathit{History}[i].\mathit{data} \in \{\mathit{Database}[r][j] : j \in \text{DOMAIN } \mathit{Database}[r]\} \cup \{0\}
\end{array}
$$

Links

https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels

https://github.com/Azure/azure-cosmos-tla

https://lamport.azurewebsites.net/tla/tla.html

http://learntla.com