```
1  ---------- MODULE CJupiter ----------
   Model of our own CJupiter protocol.
5  EXTENDS Integers, OT, TLC, FunctionUtils, SequenceUtils
6  ----------
7  CONSTANTS
8      Client,      the set of client replicas
9      Server,      the (unique) server replica
10     Char,        set of characters allowed
11     InitState    the initial state of each replica

13 Replica ≜ Client ∪ {Server}

15 List ≜ Seq(Char ∪ Range(InitState))                all possible lists/strings
16 MaxLen ≜ Cardinality(Char) + Len(InitState)  the max length of lists in any states;
17     We assume that all inserted elements are unique.

19 ClientNum ≜ Cardinality(Client)
20 Priority ≜ CHOOSE f ∈ [Client → 1 .. ClientNum] : Injective(f)
21 ----------
22 ASSUME
23     ∧ Range(InitState) ∩ Char = {}   due to the uniqueness requirement
24     ∧ Priority ∈ [Client → 1 .. ClientNum]
25 ----------
   The set of all operations. Note: The positions are indexed from 1.
30 Rd ≜ [type : {"Rd"}]
31 Del ≜ [type : {"Del"}, pos : 1 .. MaxLen]
32 Ins ≜ [type : {"Ins"}, pos : 1 .. (MaxLen + 1), ch : Char, pr : 1 .. ClientNum]  pr: priority

34 Op ≜ Ins ∪ Del
35 ----------
   Cop: operation of type Op with context
39 Oid ≜ [c : Client, seq : Nat]   operation identifier
40 Cop ≜ [op : Op ∪ {Nop}, oid : Oid, ctx : SUBSET Oid]

   tb: Is cop1 totally ordered before cop2?
   This can be determined according to the serial view (sv) of any replica.
47 tb(cop1, cop2, sv) ≜
48     LET pos1 ≜ FirstIndexOfElementSafe(sv, cop1.oid)
49         pos2 ≜ FirstIndexOfElementSafe(sv, cop2.oid)
50     IN  IF pos1 ≠ 0 ∧ pos2 ≠ 0   at the server or both are remote operations
51         THEN pos1 < pos2          at a client: one is a remote operation and the other is a local operation
52         ELSE pos1 ≠ 0
   OT of two operations of type Cop.
56 COT(lcop, rcop) ≜ [lcop EXCEPT !.op = Xform(lcop.op, rcop.op), !.ctx = @ ∪ {rcop.oid}]
57 ----------
```

```
58  VARIABLES
        For the client replicas:
62      cseq,       cseq[c]: local sequence number at client c ∈ Client
        For all replicas: the n-ary ordered state space
66      css,        css[r]: the n-ary ordered state space at replica r ∈ Replica
67      cur,        cur[r]: the current node of css at replica r ∈ Replica
68      state,      state[r]: state (the list content) of replica r ∈ Replica
        For edge ordering in CSS
72      serial,     serial[r]: the serial view of replica r ∈ Replica about the server
73      cincomingSerial,
74      sincomingSerial,
        For communication between the Server and the Clients:
78      cincoming,  cincoming[c]: incoming channel at the client c ∈ Client
79      sincoming,  incoming channel at the Server
        For model checking:
83      chins       a set of chars to insert
84  ----------------------------------------------------------------------------
85  serialVars ≜ ⟨serial, cincomingSerial, sincomingSerial⟩
86  vars ≜ ⟨chins, cseq, css, cur, state, cincoming, sincoming, serialVars⟩
87  ----------------------------------------------------------------------------
88  comm ≜ INSTANCE CSComm WITH Msg ← Cop
89  commSerial ≜ INSTANCE CSComm WITH Msg ← Seq(Oid),
90                    cincoming ← cincomingSerial, sincoming ← sincomingSerial
91  ----------------------------------------------------------------------------
        A css is a directed graph with labeled edges, represented by a record with node field and edge field.
        Each node is characterized by its context, a set of oids. Each edge is labeled with an operation.
98  IsCSS(G) ≜
99      ∧ G = [node ↦ G.node, edge ↦ G.edge]
100     ∧ G.node ⊆ (SUBSET Oid)
101     ∧ G.edge ⊆ [from : G.node, to : G.node, cop : Cop]

103 EmptySS ≜ [node ↦ {{}}, edge ↦ {}]

105 TypeOK ≜
        For the client replicas:
109     ∧ cseq ∈ [Client → Nat]
        For edge ordering in CSS:
113     ∧ serial ∈ [Replica → Seq(Oid)]
114     ∧ commSerial!TypeOK
        For all replicas: the n-ary ordered state space
118     ∧ ∀ r ∈ Replica : IsCSS(css[r])
119     ∧ cur ∈ [Replica → SUBSET Oid]
120     ∧ state ∈ [Replica → List]
```

```
                For communication between the server and the clients:
124               ∧ comm ! TypeOK
                For model checking:
128               ∧ chins ⊆ Char
129 ├──────────────────────────────────────────────────────────────────────┤
130 Init ≜
                For the client replicas:
134               ∧ cseq = [c ∈ Client ↦ 0]
                For the server replica:
138               ∧ serial = [r ∈ Replica ↦ ⟨⟩]
139               ∧ commSerial ! Init
                For all replicas: the n-ary ordered state space
143               ∧ css  = [r ∈ Replica ↦ EmptySS]
144               ∧ cur = [r ∈ Replica ↦ {}]
145               ∧ state = [r ∈ Replica ↦ InitState]
                For communication between the server and the clients:
149               ∧ comm ! Init
                For model checking:
153               ∧ chins = Char
154 ├──────────────────────────────────────────────────────────────────────┤
    Locate the node in rcss (the css at replica r ∈ Replica) that matches the context ctx of cop.
158 Locate(cop, rcss) ≜ CHOOSE n ∈ rcss.node : n = cop.ctx
    Take union of two state spaces ss1 and ss2.
162 ss1 ⊕ ss2 ≜ [node ↦ ss1.node ∪ ss2.node, edge ↦ ss1.edge ∪ ss2.edge]
    xForm: Iteratively transform cop with a path through the css at replica r ∈ Replica, following
    the first edges.
167 xForm(cop, r) ≜
168     LET rcss ≜ css[r]
169         u ≜ Locate(cop, rcss)
170         v ≜ u ∪ {cop.oid}
171         RECURSIVE xFormHelper(_, _, _, _, _, _)
172          'h' stands for "helper"; xcss: eXtra css created during transformation
173         xFormHelper(uh, vh, coph, xcss, xcoph, xcurh) ≜
174             IF uh = cur[r]
175              THEN ⟨xcss, xcoph, xcurh⟩
176              ELSE LET fedge ≜ CHOOSE e ∈ rcss.edge :
177                                 ∧ e.from = uh
178                                 ∧ ∀ uhe ∈ rcss.edge :
179                                     (uhe.from = uh ∧ uhe ≠ e) ⇒ tb(e.cop, uhe.cop, serial[r])
180                       uprime ≜ fedge.to
181                       fcop ≜ fedge.cop
182                       coph2fcop ≜ COT(coph, fcop)
183                       fcop2coph ≜ COT(fcop, coph)
```

```
                        vprime ≜ vh ∪ {fcop.oid}
                IN      xFormHelper(uprime, vprime, coph2fcop,
                          [xcss EXCEPT !.node = @ ∪ {vprime},
                            !.edge = @ ∪ {[from ↦ vh, to ↦ vprime, cop ↦ fcop2coph],
                                          [from ↦ uprime, to ↦ vprime, cop ↦ coph2fcop]}],
                          coph2fcop, vprime)
  IN    xFormHelper(u, v, cop, [node ↦ {v}, edge ↦ {[from ↦ u, to ↦ v, cop ↦ cop]}], cop, v)
```

Perform cop at replica $r \in Replica$.

```
Perform(cop, r) ≜
  LET xform ≜ xForm(cop, r)   xform: ⟨xcss, xcop, xcur⟩
      xcss ≜ xform[1]
      xcop ≜ xform[2]
      xcur ≜ xform[3]
  IN  ∧ css' = [css EXCEPT ![r] = @ ⊕ xcss]
      ∧ cur' = [cur EXCEPT ![r] = xcur]
      ∧ state' = [state EXCEPT ![r] = Apply(xcop.op, @)]
```

Client $c \in Client$ issues an operation op.

```
DoOp(c, op) ≜   op: the raw operation generated by the client c ∈ Client
    ∧ cseq' = [cseq EXCEPT ![c] = @ + 1]
    ∧ LET cop ≜ [op ↦ op, oid ↦ [c ↦ c, seq ↦ cseq'[c]], ctx ↦ cur[c]]
      IN  ∧ Perform(cop, c)
          ∧ comm!CSend(cop)

DoIns(c) ≜
  ∃ ins ∈ {op ∈ Ins : op.pos ∈ 1 .. (Len(state[c]) + 1) ∧ op.ch ∈ chins ∧ op.pr = Priority[c]} :
    ∧ DoOp(c, ins)
    ∧ chins' = chins \ {ins.ch}   We assume that all inserted elements are unique.
    ∧ UNCHANGED ⟨serialVars⟩

DoDel(c) ≜
  ∃ del ∈ {op ∈ Del : op.pos ∈ 1 .. Len(state[c])} :
    ∧ DoOp(c, del)
    ∧ UNCHANGED ⟨chins, serialVars⟩

Do(c) ≜
    ∨ DoIns(c)
    ∨ DoDel(c)
```

Client $c \in Client$ receives a message from the $Server$.

```
Rev(c) ≜
    ∧ comm!CRev(c)
    ∧ Perform(Head(cincoming[c]), c)
    ∧ commSerial!CRev(c)
    ∧ serial' = [serial EXCEPT ![c] = Head(cincomingSerial[c])]
    ∧ UNCHANGED ⟨chins, cseq⟩
```

```
The Server receives a message.
SRev ≜
    ∧ comm!SRev
    ∧ LET cop ≜ Head(sincoming)
      IN  ∧ Perform(cop, Server)
          ∧ comm!SSendSame(cop.oid.c, cop)   broadcast the original operation
          ∧ serial' = [serial EXCEPT ![Server] = Append(@, cop.oid)]
          ∧ commSerial!SSendSame(cop.oid.c, serial'[Server])
    ∧ UNCHANGED ⟨chins, cseq, sincomingSerial⟩
Next ≜
    ∨ ∃ c ∈ Client : Do(c) ∨ Rev(c)
    ∨ SRev
Fairness: There is no requirement that the clients ever generate operations.
Fairness ≜
    WF_vars(SRev ∨ ∃ c ∈ Client : Rev(c))

Spec ≜ Init ∧ □[Next]_vars   ∧ Fairness (We care more about safety.)
The compactness of CJupiter: the CSSes at all replicas are the same.
Compactness ≜
    comm!EmptyChannel ⇒ Cardinality(Range(css)) = 1

THEOREM Spec ⇒ Compactness
\* Modification History
\* Last modified Tue Dec 04 18:34:22 CST 2018 by hengxin
\* Created Sat Sep 01 11:08:00 CST 2018 by hengxin
```