```
1  ---------- MODULE XJupiter ----------
   Specification of the Jupiter protocol described in CSCW'2014 by Yi Xu, Chengzheng Sun, and
   Mo Li. We call it XJupiter, with 'X' for "Xu".
7  EXTENDS Integers, OT, TLC, AdditionalFunctionOperators, AdditionalSequenceOperators
8  ----------
9  CONSTANTS
10     Client,      the set of client replicas
11     Server,      the (unique) server replica
12     Char,        set of characters allowed
13     InitState    the initial state of each replica

15 Replica ≜ Client ∪ {Server}

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

21 ClientNum ≜ Cardinality(Client)
22 Priority ≜ CHOOSE f ∈ [Client → 1 .. ClientNum] : Injective(f)

24  direction flags
25 Local ≜ 0
26 Remote ≜ 1
27 ----------
28 ASSUME
29     ∧ Range(InitState) ∩ Char = {}   due to the uniqueness requirement
30     ∧ Priority ∈ [Client → 1 .. ClientNum]
31 ----------
   The set of all operations. Note: The positions are indexed from 1.
36 Rd ≜ [type : {"Rd"}]
37 Del ≜ [type : {"Del"}, pos : 1 .. MaxLen]
38 Ins ≜ [type : {"Ins"}, pos : 1 .. (MaxLen + 1), ch : Char, pr : 1 .. ClientNum]  pr: priority

40 Op ≜ Ins ∪ Del
41 ----------
   Cop: operation of type Op with context
45 Oid ≜ [c : Client, seq : Nat]   operation identifier
46 Cop ≜ [op : Op ∪ {Nop}, oid : Oid, ctx : SUBSET Oid]

   OT of two operations of type Cop.
51 COT(lcop, rcop) ≜ [lcop EXCEPT !.op = Xform(lcop.op, rcop.op), !.ctx = @ ∪ {rcop.oid}]
52 ----------
53 VARIABLES
       For the client replicas:
57     cseq,      cseq[c]: local sequence number at client c ∈ Client
```

The $2D$ state spaces (ss, for short). Each client maintains one $2D$ state space. The server maintains n $2D$ state spaces, one for each client.

63 css, $css[c]$: the $2D$ state space at client $c \in Client$
64 $ccur$, $cur[c]$: the current node of $css[c]$
65 sss, $sss[c]$: the $2D$ state space maintained by the $Server$ for client $c \in Client$
66 $scur$, $scur[c]$: the current node of $sss[c]$

For all replicas

70 $state$, $state[r]$: state (the list content) of replica $r \in Replica$

For communication between the $Server$ and the Clients:

74 $cincoming$, $cincoming[c]$: incoming channel at the client $c \in Client$
75 $sincoming$, incoming channel at the $Server$

For model checking:

79 $chins$ a set of chars to insert

80 ----

81 $comm \triangleq$ INSTANCE $CSComm$ WITH $Msg \leftarrow Cop$

82 ----

83 $eVars \triangleq \langle chins \rangle$ variables for the environment
84 $cVars \triangleq \langle cseq \rangle$ variables for the clients
85 $cssVars \triangleq \langle css, ccur \rangle$ variables for $2D$ state spaces at clients
86 $sssVars \triangleq \langle sss, scur \rangle$ variables for $2D$ state spaces at the $Server$
87 $commVars \triangleq \langle cincoming, sincoming \rangle$ variables for communication
88 $vars \triangleq \langle eVars, cVars, commVars, cssVars, sssVars, state \rangle$ all variables

89 ----

A $2D$ state space is a directed graph with labeled edges. It is represented by a record with node field and edge field. Each node is characterized by its context, a set of operations. Each edge is labeled with an operation and a direction flag indicating whether this edge is LOCAL or REMOTE. For clarity, we denote edges by records instead of tuples.

98 $IsSS(G) \triangleq$
99 $\land G = [node \mapsto G.node, edge \mapsto G.edge]$
100 $\land G.node \subseteq (\text{SUBSET } Oid)$
101 $\land G.edge \subseteq [from : G.node, to : G.node, cop : Cop, lr : \{Local, Remote\}]$

103 $TypeOK \triangleq$

For the client replicas:

107 $\land cseq \in [Client \to Nat]$

For the $2D$ state spaces:

111 $\land \forall c \in Client : IsSS(css[c]) \land IsSS(sss[c])$
112 $\land ccur \in [Client \to \text{SUBSET } Oid]$
113 $\land scur \in [Client \to \text{SUBSET } Oid]$
114 $\land state \in [Replica \to List]$

For communication between the server and the clients:

118 $\land comm\,!\,TypeOK$

For model checking:

```
122     ∧ chins ⊆ Char
123 ├──────────────────────────────────────────────────────────────────────┤
    The Init predicate.
127 Init ≜
    For the client replicas:
131     ∧ cseq = [c ∈ Client ↦ 0]
    For the 2D state spaces:
135     ∧ css = [c ∈ Client ↦ [node ↦ {{}}, edge ↦ {}]]
136     ∧ ccur = [c ∈ Client ↦ {}]
137     ∧ sss = [c ∈ Client ↦ [node ↦ {{}}, edge ↦ {}]]
138     ∧ scur = [c ∈ Client ↦ {}]
    For all replicas:
142     ∧ state = [r ∈ Replica ↦ InitState]
    For communication between the server and the clients:
146     ∧ comm!Init
    For model checking:
150     ∧ chins = Char
151 ├──────────────────────────────────────────────────────────────────────┤
    Locate the node in the 2D state space ss which matches the context ctx of cop.
155 Locate(cop, ss) ≜ CHOOSE n ∈ (ss.node) : n = cop.ctx

    xForm: iteratively transform cop with a path through the 2D state space ss at some client,
    following the edges with the direction flag d.
162 xForm(cop, ss, cur, d) ≜
163     LET u ≜ Locate(cop, ss)
164         v ≜ u ∪ {cop.oid}
165         RECURSIVE xFormHelper(_, _, _, _)
166          'h' stands for "helper"; xss: eXtra ss created during transformation
167         xFormHelper(uh, vh, coph, xss) ≜
168             IF uh = cur
169              THEN xss
170              ELSE LET e ≜ CHOOSE e ∈ ss.edge : e.from = uh ∧ e.lr = d
171                       uprime ≜ e.to
172                       copprime ≜ e.cop
173                       coph2copprime ≜ COT(coph, copprime)
174                       copprime2coph ≜ COT(copprime, coph)
175                        vprime ≜ vh ∪ {copprime.oid}
176                   IN   xFormHelper(uprime, vprime, coph2copprime,
177                            [xss EXCEPT !.node = @ ∘ ⟨vprime⟩,
178                                        the order of recording edges here is important
179                                        so that the last one is labeled with the final transformed operation
180                                       !.edge = @ ∘ ⟨[from ↦ vh, to ↦ vprime, cop ↦ copprime2coph, lr ↦
181                                                    [from ↦ uprime, to ↦ vprime, cop ↦ coph2copprime,
```

```
182     IN   xFormHelper(u, v, cop, [node ↦ ⟨v⟩,
183                                  edge ↦ ⟨[from ↦ u, to ↦ v, cop ↦ cop, lr ↦ 1 − d]⟩])
184 ⊢──────────────────────────────────────────────────────────────────────────────────────⊣
    Client c ∈ Client perform operation cop guided by the direction flag d.
188 ClientPerform(cop, c, d) ≜
189     LET xss ≜ xForm(cop, css[c], ccur[c], d)
190         xn ≜ xss.node
191         xe ≜ xss.edge
192         xcur ≜ Last(xn)
193         xcop ≜ Last(xe).cop
194     IN   ∧ css′ = [css EXCEPT ![c].node = @ ∪ Range(xn),
195                               ![c].edge = @ ∪ Range(xe)]
196          ∧ ccur′ = [ccur EXCEPT ![c] = xcur]
197          ∧ state′ = [state EXCEPT ![c] = Apply(xcop.op, @)]
    Client c ∈ Client issues an operation op.
201 DoOp(c, op) ≜   op: the raw operation generated by the client c ∈ Client
202         ∧ cseq′ = [cseq EXCEPT ![c] = @ + 1]
203         ∧ LET cop ≜ [op ↦ op, oid ↦ [c ↦ c, seq ↦ cseq′[c]], ctx ↦ ccur[c]]
204             IN   ∧ ClientPerform(cop, c, Remote)
205                  ∧ comm!CSend(cop)

207 DoIns(c) ≜
208     ∃ ins ∈ Ins :
209         ∧ ins.pos ∈ 1 .. (Len(state[c]) + 1)
210         ∧ ins.ch ∈ chins
211         ∧ ins.pr = Priority[c]
212         ∧ chins′ = chins \ {ins.ch}   We assume that all inserted elements are unique.
213         ∧ DoOp(c, ins)
214         ∧ UNCHANGED ⟨sssVars⟩

216 DoDel(c) ≜
217     ∃ del ∈ Del :
218         ∧ del.pos ∈ 1 .. Len(state[c])
219         ∧ DoOp(c, del)
220         ∧ UNCHANGED ⟨sssVars, eVars⟩

222 Do(c) ≜
223         ∨ DoIns(c)
224         ∨ DoDel(c)
    Client c ∈ Client receives a message from the Server.
228 Rev(c) ≜
229         ∧ comm!CRev(c)
230         ∧ LET cop ≜ Head(cincoming[c])   the received (transformed) operation
231             IN   ClientPerform(cop, c, Local)
232         ∧ UNCHANGED ⟨eVars, cVars, sssVars⟩
```

```
The Server performs operation cop.
ServerPerform(cop) ≜
    LET c ≜ cop.oid.c
      xss ≜ xForm(cop, sss[c], scur[c], Remote)
       xn ≜ xss.node
       xe ≜ xss.edge
     xcur ≜ Last(xn)
     xcop ≜ Last(xe).cop
    IN  ∧ sss' = [cl ∈ Client ↦
                    IF cl = c
                     THEN [sss[cl] EXCEPT !.node = @ ∪ Range(xn),
                                          !.edge = @ ∪ Range(xe)]
                     ELSE LET scurcl ≜ scur[cl]
                              scurclprime ≜ scurcl ∪ {cop.oid}
                          IN  [sss[cl] EXCEPT !.node = @ ∪ {scurclprime},
                                              !.edge = @ ∪ {[from ↦ scurcl, to ↦ scurclprime,
                                                            cop ↦ xcop, lr ↦ Remote]}]
                ]
        ∧ scur' = [cl ∈ Client ↦
                    IF cl = c THEN xcur ELSE scur[cl] ∪ {cop.oid}]
        ∧ state' = [state EXCEPT ![Server] = Apply(xcop.op, @)]
        ∧ comm!SSendSame(c, xcop)   broadcast the transformed operation
The Server receives a message.
SRev ≜
    ∧ comm!SRev
    ∧ LET cop ≜ Head(sincoming)
      IN  ServerPerform(cop)
    ∧ UNCHANGED ⟨eVars, cVars, cssVars⟩
The next-state relation.
Next ≜
    ∨ ∃ c ∈ Client : Do(c) ∨ Rev(c)
    ∨ SRev
The Spec. (TODO: Check the fairness condition.)
Spec ≜ Init ∧ □[Next]_vars ∧ WF_vars(Next)
```

* Modification History
* *Last* modified *Wed Oct* 10 15:00:09 *CST* 2018 by *hengxin*
* Created *Tue Oct* 09 16:33:18 *CST* 2018 by *hengxin*