```
┌──────────────────── MODULE AJupiter ────────────────────┐
 Model checking the Jupiter protocol presented by Attiya and others.
EXTENDS OT, TLC
├──────────────────────────────────────────────────────────┤
CONSTANTS
    Client,        the set of client replicas
    Server,        the (unique) server replica
    InitState,     the initial state of each replica
    Cop            Cop[c]: operations issued by the client c ∈ Client

 OpToIssue ≜ {op ∈ SUBSET }
ASSUME
    ∧ InitState ∈ List
    ∧ Cop ∈ [Client → Seq(Op)]

VARIABLES
    For model checking:
    cop,          cop[c]: operations issued by the client c ∈ Client

    For the client replicas:
    cbuf,        cbuf[c]: buffer (of operations) at the client c ∈ Client
    crec,        crec[c]: the number of new messages have been received by the client c ∈ Client
                       since the last time a message was sent
    cstate,     cstate[c]: state (the list content) of the client c ∈ Client

    For the server replica:
    sbuf,       sbuf[c]: buffer (of operations) at the Server, one per client c ∈ Client
    srec,       srec[c]: the number of new messages have been ..., one per client c ∈ Client
    sstate,     sstate: state (the list content) of the server Server

    For communication between the Server and the Clients:
    cincoming,    cincoming[c]: incoming channel at the client c ∈ Client
    sincoming     incoming channel at the Server
├──────────────────────────────────────────────────────────┤
comm ≜ INSTANCE CSComm
├──────────────────────────────────────────────────────────┤
cVars ≜ ⟨cop, cbuf, crec, cstate⟩
sVars ≜ ⟨sbuf, srec, sstate⟩
 FIXME: subscript error (Don't know why yet!)
 vars ≜ cVars ∘ sVars ∘ ⟨cincoming, sincoming⟩
vars ≜ ⟨cop, cbuf, crec, cstate, sbuf, srec, sstate, cincoming, sincoming⟩    all variables
├──────────────────────────────────────────────────────────┤
TypeOK ≜
    ∧ cop ∈ [Client → Seq(Op)]
    For the client replicas:
    ∧ cbuf  ∈ [Client → Seq(Op ∪ {Nop})]
    ∧ crec  ∈ [Client → Nat]
    ∧ cstate ∈ [Client → List]
    For the server replica:
    ∧ sbuf  ∈ [Client → Seq(Op ∪ {Nop})]
    ∧ srec  ∈ [Client → Nat]
    ∧ sstate ∈ List
    For communication between the server and the clients:
    ∧ comm!TypeOK
├──────────────────────────────────────────────────────────────────────┤
The Init predicate.
Init ≜
    ∧ cop = Cop
    For the client replicas:
    ∧ cbuf  = [c ∈ Client ↦ ⟨⟩]
    ∧ crec  = [c ∈ Client ↦ 0]
    ∧ cstate = [c ∈ Client ↦ InitState]
    For the server replica:
    ∧ sbuf  = [c ∈ Client ↦ ⟨⟩]
    ∧ srec  = [c ∈ Client ↦ 0]
    ∧ sstate = InitState
    For communication between the server and the clients:
    ∧ comm!Init
├──────────────────────────────────────────────────────────────────────┤
Client c ∈ Client issues an operation op.
Do(c) ≜
      ∧ cop[c] ≠ ⟨⟩
      ∧ LET op ≜ LegalizeOp(Head(cop[c]), cstate[c])    preprocess illegal operations
          IN   ∧ PrintT(c ∘ ": Do " ∘ ToString(op))
               ∧ cstate′ = [cstate EXCEPT ![c] = Apply(op, @)]
               ∧ cbuf′ = [cbuf EXCEPT ![c] = Append(@, op)]
               ∧ comm!CSend([c ↦ c, ack ↦ crec[c], op ↦ op])
      ∧ crec′ = [crec EXCEPT ![c] = 0]
      ∧ cop′ = [cop EXCEPT ![c] = Tail(@)]    consume one operation
      ∧ UNCHANGED sVars

Client c ∈ Client receives a message from the Server.
Rev(c) ≜
      ∧ comm!CRev(c)
      ∧ crec′ = [crec EXCEPT ![c] = @ + 1]
      ∧ LET m ≜ Head(cincoming[c])
            cBuf ≜ cbuf[c]    the buffer at client c ∈ Client
            cShiftedBuf ≜ SubSeq(cBuf, m.ack + 1, Len(cBuf))    buffer shifted
```

```
                    xop ≜ XformOpOps(m.op, cShiftedBuf)  transform op vs. shifted buffer
                    xcBuf ≜ XformOpsOp(cShiftedBuf, m.op)  transform shifted buffer vs. op
              IN   ∧ cbuf′ = [cbuf EXCEPT ![c] = xcBuf]
                   ∧ cstate′ = [cstate EXCEPT ![c] = Apply(xop, @)]  apply the transformed operation xop
          ∧ UNCHANGED ⟨sbuf, srec, sstate, cop⟩   NOTE: sVars ∘ ⟨cop⟩ is wrong!
├──────────────────────────────────────────────────────────────────────┤
The Server receives a message.
SRev ≜
    ∧ comm!SRev
    ∧ LET m ≜ Head(sincoming)   the message to handle with
          c ≜ m.c               the client c ∈ Client that sends this message
          cBuf ≜ sbuf[c]        the buffer at the Server for client c ∈ Client
          cShiftedBuf ≜ SubSeq(cBuf, m.ack + 1, Len(cBuf))  buffer shifted
          xop ≜ XformOpOps(m.op, cShiftedBuf)  transform op vs. shifted buffer
          xcBuf ≜ XformOpsOp(cShiftedBuf, m.op)  transform shifted buffer vs. op
      IN  ∧ srec′ = [cl ∈ Client ↦
                        IF cl = c
                         THEN srec[cl] + 1  receive one more operation from client c ∈ Client
                         ELSE 0]  reset srec for other clients than c ∈ Client
          ∧ sbuf′ = [cl ∈ Client ↦
                        IF cl = c
                         THEN xcBuf  transformed buffer for client c ∈ Client
                         ELSE Append(sbuf[cl], xop)]  store transformed xop into other clients' bufs
          ∧ sstate′ = Apply(xop, sstate)  apply the transformed operation
          ∧ comm!SSend(c, srec, xop)
    ∧ UNCHANGED cVars
├──────────────────────────────────────────────────────────────────────┤
The next-state relation.
Next ≜
    ∨ ∃ c ∈ Client : Do(c) ∨ Rev(c)
    ∨ SRev
The Spec.
Spec ≜ Init ∧ □[Next]_vars ∧ WF_vars(Next)
├──────────────────────────────────────────────────────────────────────┤
The safety properties to check: Eventual Convergence (EC), Quiescent Consistency (QC), Strong
Eventual Convergence (SEC), Weak List Specification, (WLSpec), and Strong List Specification,
(SLSpec).

Eventual Consistency (EC)

Quiescent Consistency (QC)
QConvergence ≜ ∀ c ∈ Client : cstate[c] = sstate
QC ≜ comm!EmptyChannel ⇒ QConvergence

THEOREM Spec ⇒ □QC
```

Strong Eventual Consistency (*SEC*)

Weak *List* Consistency (*WLSpec*)

Strong *List* Consistency (*SLSpec*)

\ ∗ Modification History
\ ∗ *Last* modified Sat *Jul* 07 21:26:52 *CST* 2018 by *hengxin*
\ ∗ Created Sat *Jun* 23 17:14:18 *CST* 2018 by *hengxin*