MODULE *backpressure*

EXTENDS *FiniteSets*, *Integers*, *Sequences*, *TLC*

$Null \triangleq 0$
$Cowns \triangleq 1 \,..\, 3$ # *TODO*: 4
$MaxMessageCount \triangleq 4$ # *TODO*: 4
$MaxMessageSize \triangleq 3$
$OverloadThreshold \triangleq 2$
$PriorityLevels \triangleq \{-1, 0, 1, 2\}$

$Pick(s) \triangleq \text{CHOOSE } x \in s : \text{TRUE}$
$Min(s) \triangleq \text{CHOOSE } x \in s : \forall\, y \in s \setminus \{x\} : y > x$
$Max(s) \triangleq \text{CHOOSE } x \in s : \forall\, y \in s \setminus \{x\} : y < x$
$Range(f) \triangleq \{f[x] : x \in \text{DOMAIN } f\}$
$Subsets(s, min, max) \triangleq$
  $\{x \in \text{SUBSET } s : (Cardinality(x) \geq min) \land (Cardinality(x) \leq max)\}$
RECURSIVE $Concat(\_)$
$Concat(s) \triangleq \text{IF } s = \{\} \text{ THEN } \langle\rangle \text{ ELSE LET } x \triangleq Pick(s) \text{IN } x \circ Concat(s \setminus \{x\})$

VARIABLES *fuel*, *queue*, *scheduled*, *running*, *mutor*, *muted*, *blocker*
$vars \triangleq \langle fuel, queue, scheduled, running, mutor, muted, blocker\rangle$

$Messages \triangleq \text{UNION } \{Range(queue[c]) : c \in Cowns\}$
$EmptyQueue(c) \triangleq Len(queue[c]) = 0$
$Overloaded(c) \triangleq Len(queue[c]) \geq OverloadThreshold$
$Enqueue(c, m) \triangleq c :> Append(queue[c], m)$
$Dequeue(c) \triangleq c :> Tail(queue[c])$

RECURSIVE $Blockers(\_)$
$Blockers(c) \triangleq$
  IF $blocker[c] = Null$ THEN $\{\}$
  ELSE $\{blocker[c]\} \cup Blockers(blocker[c])$

$Running(c) \triangleq \exists\, k \in Cowns : running[k] \land c \in Head(queue[k])$

$AcquiredBy(a, b) \triangleq$
  $\land\, a < b$
  $\land\, \exists\, c \in Cowns : a \in \text{UNION } Range(queue[b])$
  $\land\, b = Min(\{c \in Cowns : a \in \text{UNION } Range(queue[b])\})$
$Acquired(c) \triangleq \exists\, k \in Cowns : AcquiredBy(c, k)$

$MutedBy(a, b) \triangleq$
  $\land\, muted[a]$
  $\land\, \exists\, m \in Range(queue[b]) : (b \notin m) \land (a \in m)$

$Init \triangleq$
  $\land\, fuel = MaxMessageCount$

$\land\ queue = [c \in Cowns \mapsto \langle\{c\}\rangle]$
$\land\ scheduled = [c \in Cowns \mapsto \text{TRUE}]$
$\land\ running = [c \in Cowns \mapsto \text{FALSE}]$
$\land\ mutor = [c \in Cowns \mapsto Null]$
$\land\ muted = [c \in Cowns \mapsto \text{FALSE}]$
$\land\ blocker = [c \in Cowns \mapsto Null]$

$Terminating \triangleq$
$\land\ \forall\, c \in Cowns : EmptyQueue(c)$
———
$\land$ UNCHANGED $vars$

$ExternalReceive(cown) \triangleq$
$\land\ fuel > 0$
———
$\land$ UNCHANGED $\langle scheduled,\ running,\ mutor,\ muted,\ blocker\rangle$
$\land\ fuel' = fuel - 1$
\# Receive a message from an external source
$\land\ \exists\, others \in Subsets(\{c \in Cowns : c > cown\},\ 0,\ MaxMessageSize - 1) :$
$queue' = Enqueue(cown,\ \{cown\} \cup others)$ @@ $queue$

$Acquire(cown) \triangleq$
$\land\ scheduled[cown]$
$\land\ \neg running[cown]$
$\land\ \neg EmptyQueue(cown)$
$\land\ cown \in Head(queue[cown])$
$\land\ cown < Max(Head(queue[cown]))$
———
$\land$ UNCHANGED $\langle fuel,\ running,\ mutor\rangle$
\# *Unschedule* and forward the message to the next *cown*.
$\land$ LET
$msg \triangleq Head(queue[cown])$
$next \triangleq Min(\{c \in msg : c > cown\})$
IN
$\land\ queue' = Enqueue(next,\ msg)$ @@ $Dequeue(cown)$ @@ $queue$
$\land\ blocker' = (cown :> next)$ @@ $blocker$
$\land$ IF $Overloaded(cown)$ THEN
$\land\ scheduled' = (next :> \text{TRUE})$ @@ $(cown :> \text{FALSE})$ @@ $scheduled$
$\land\ muted' = (next :> \text{FALSE})$ @@ $muted$
ELSE
$\land$ UNCHANGED $\langle muted\rangle$
$\land\ scheduled' = (cown :> \text{FALSE})$ @@ $scheduled$

$Unmute(cown) \triangleq$
$\land\ scheduled[cown]$
$\land\ \neg running[cown]$

$\land \neg EmptyQueue(cown)$
$\land cown \notin Head(queue[cown])$
———
$\land$ UNCHANGED $\langle fuel, running, mutor, blocker\rangle$
Remove message from queue.
$\land queue' = Dequeue(cown) \mathbin{@@} queue$
Reschedule muted cowns.
$\land muted' = [c \in Head(queue[cown]) \mapsto \text{FALSE}] \mathbin{@@} muted$
$\land scheduled' = [c \in Head(queue[cown]) \mapsto \text{TRUE}] \mathbin{@@} scheduled$

$PreRun(cown) \triangleq$
$\land scheduled[cown]$
$\land \neg running[cown]$
$\land \neg EmptyQueue(cown)$
$\land cown = Max(Head(queue[cown]))$
———
$\land$ UNCHANGED $\langle fuel, queue, scheduled, mutor, muted\rangle$
Set *max cown* in current message to running
$\land running' = (cown :> \text{TRUE}) \mathbin{@@} running$
$\land blocker' = [c \in Head(queue[cown]) \mapsto Null] \mathbin{@@} blocker$

$Send(cown) \triangleq$
$\land running[cown]$
$\land fuel > 0$
———
$\land$ UNCHANGED $\langle scheduled, running, muted, blocker\rangle$
$\land fuel' = fuel - 1$
Select set of receivers
$\land \exists\, receivers \in Subsets(Cowns, 1, MaxMessageSize) :$
LET
$next \triangleq Min(receivers)$
$senders \triangleq Head(queue[cown])$
$mutors \triangleq \{c \in receivers : Overloaded(c)\}$
IN
Place message for receivers in the first receiver's queue.
$\land queue' = Enqueue(next, receivers) \mathbin{@@} queue$
Set *mutor* if any receiver is overloaded and there are no receivers in the set of senders.
$\land$ IF
$\land mutors \neq \{\}$
$\land mutor[cown] = Null$
$\land (senders \cap receivers) = \{\}$
THEN $mutor' = (cown :> Min(mutors)) \mathbin{@@} mutor$
ELSE UNCHANGED $\langle mutor\rangle$

$PostRun(cown) \triangleq$
$\land running[cown]$

```
  ∧ UNCHANGED ⟨fuel, blocker⟩
  ∧ running' = (cown :> FALSE) @@ running
  ∧ mutor' = (cown :> Null) @@ mutor
  ∧ LET msg ≜ Head(queue[cown]) IN
      # Mute if mutor is set and no running cowns are overloaded.
      ∧ IF (mutor[cown] ≠ Null) ∧ (∀ c ∈ msg : ¬Overloaded(c)) THEN
          ∧ muted' = [c ∈ msg ↦ TRUE] @@ muted
          ∧ scheduled' = [c ∈ msg ↦ FALSE] @@ scheduled
          # Send unmute message to mutor
          ∧ queue' = Enqueue(mutor[cown], msg) @@ Dequeue(cown) @@ queue
        ELSE
          ∧ UNCHANGED ⟨muted⟩
          ∧ scheduled' = [c ∈ msg ↦ TRUE] @@ scheduled
          ∧ queue' = Dequeue(cown) @@ queue

RunStep(cown) ≜
    ∨ ExternalReceive(cown) \* # Very expensive check
  ∨ Acquire(cown)
  ∨ Unmute(cown)
  ∨ PreRun(cown)
  ∨ Send(cown)
  ∨ PostRun(cown)

Next ≜ ∃ c ∈ Cowns : RunStep(c)

Spec ≜
  ∧ Init
  ∧ □[Next ∨ Terminating]_vars
  ∧ ∀ c ∈ Cowns : WF_vars(RunStep(c))

 # Properties

 # Ensure that the termination condition is reached by the model.
Termination ≜ ◇□(∀ c ∈ Cowns : EmptyQueue(c))

 # Invariants

 # Ensure that the model produces finite messages.
MessageLimit ≜ Cardinality(Messages) ≤ (Cardinality(Cowns) + MaxMessageCount)

 # Cowns are acquired by one running message at a time.
UniqueAcquisition ≜
  LET msgs ≜ Concat({⟨Head(queue[c])⟩ : c ∈ {k ∈ Cowns : running[k]}})
  IN   Cardinality(Range(msgs)) = Len(msgs)

 # Each queue has at most one token message.
LoneToken ≜ ∀ c ∈ Cowns : Len(SelectSeq(queue[c], LAMBDA m : m = {})) ≤ 1
```

```
 # A running cown must be scheduled and be the max cown in the message at the head of its queue.
RunningImplication ≜ ∀ c ∈ Cowns : running[c] ⇒
  ∧ scheduled[c]
  ∧ c = Max(Head(queue[c]))
  ∧ ∀ k ∈ Head(queue[c]) : (k < c) ⇒ ¬scheduled[k]

 # A muted cown is not scheduled or running.
MutedImplication ≜ ∀ c ∈ Cowns : muted[c] ≡
  ∧ ∃ k ∈ Cowns : MutedBy(c, k)
  ∧ ¬scheduled[c]
  ∧ ¬Running(c)

 # A muted cown exists in an unmute message in the queue of at least one mutor.
MutedOnce ≜
 ∀ m ∈ {c ∈ Cowns : muted[c]} :
   Cardinality({c ∈ Cowns : MutedBy(m, c)}) > 0

 # A cown may only be acqiured by one message.
AcquiredOnce ≜
 ∀ a ∈ {c ∈ Cowns : Acquired(c)} :
   Cardinality({c ∈ Cowns : AcquiredBy(a, c)}) = 1

 # An acquired cown is acquired by a cown in its blocker set.
AcquiredByBlocker ≜ ∀ ⟨a, b⟩ ∈ Cowns × Cowns :
  AcquiredBy(a, b) ⇒ b ∈ Blockers(a)

 # An overloaded cown doesn't exist in a muted cown's queue.
OverloadedNotInMutedQueue ≜ ∀ ⟨o, m⟩ ∈ Cowns × Cowns :
  Overloaded(o) ∧ muted[m] ⇒ o ∉ UNION Range(queue[m])
```

```
(*
\* https://github.com/tlaplus/Examples/blob/master/specifications/TransitiveClosure/TransitiveClosure.tla#L114

TC(R) ≜
 LET
  S ≜ {r[1] : r ∈ R} ∪ {r[2] : r ∈ R}
  RECURSIVE TCR(_)
  TCR(T) ≜
   IF T = {} THEN R
    ELSE
     LET
      r ≜ CHOOSE s ∈ T: TRUE
      RR ≜ TCR(T \ {r})
     IN
      RR ∪ {⟨s, t⟩ ∈ S × S : ⟨s, r⟩ ∈ RR ∧ ⟨r, t⟩ ∈ RR}
 IN
  TCR(S)
```

```
CylcicTransitiveClosure(R(_, _)) ≜
  LET s ≜ {⟨a, b⟩ ∈ Cowns × Cowns : R(a, b)}
  IN  ∃ c ∈ Cowns: ⟨c, c⟩ ∈ TC(s)
*)
```