v1.1.0 (2017-07-17)

|  |  |
| --- | --- |
| // === SENDER ===  // functions must be synchronized!  **init**() {  **var** id = 0, expectedAck = 0 // (\*)  **var** connected = **false**  **var** srtt = 0, totalTicks = 0  **var** crcid = 0  timer(trigger, *PERIOD*)  }  **connect**(crc\_id) { // (\*\*)  crcid = crc\_id  id = expectedAck = *SYN*  srtt = *DEFAULT\_SRTT*  totalTicks = 0  connected = **false**  deleteAll(buffer)  add(buffer[id])  write(id, crcid, **nil**)  }  **send**(msg) {  **if** (connected) {  id = next(id)  add(buffer[id])  buffer[id].startRtt = totalTicks  buffer[id].msg = msg  write(id, crcid, msg)  } **else** error(“not connected”)  }  **onAck**(ack) {  **if** (buffer[ack] != **nil**) {  buffer[ack].ack = **true**  **if** (ack == expectedAck) {  **var** rtt = totalTicks – buffer[ack].startRtt + 1  **var** newSrtt = *RC*\*srtt + (1-*RC*)\*rtt  srtt = min(max(newSrtt, *MIN\_SRTT*), *MAX\_SRTT*)  accept()  }  }  **if** (ack == *SYN*) {  connected = **true**  info(“sender connected”)  }  }  **accept**() {  **if** (buffer[expectedAck] != **nil**) {  **if** buffer[expectedAck].ack {  delete(buffer[expectedAck])  expectedAck = next(expectedAck)  accept()  }  }  }  **trigger**() {  totalTicks++  **var** i = expectedAck  **if** (buffer[i] != **nil** && !buffer[i].ack) {  **if** (buffer[i].attempt > *MAX\_ATTEMPTS*) {  connected = **false**  deleteAll(buffer)  error(“connection failed”)  **return**  } **else if** (buffer[i].ticks==buffer[i].nextRepeat) {  buffer[i].attempt++  buffer[i].nextRepeat+=*AC*\*srtt\*buffer[i].attempt  **if** (buffer[i].attempt > 1) {  buffer[i].startRtt = totalTicks  write(i, crcid, buffer[i].msg)  }  }  buffer[i].ticks++  }  }  // (\*) if (RING\_ALGORITHM) then a closed ring data type is recommended (e.g. for uint8: 255+1 = 0; 3-255 = 4)  // (\*\*) crcid must be generated on client with Crypto Random | // === RECEIVER ===  // if the receiver is being run by several threads then functions must be synchronized  **init**() {  **var** expected = 0, connected = **false**  **var** pending = 0  }  **onMsg**(id, crcid, msg) {  **if** (id == *SYN*) {  writeAck(id, crcid)  deleteAll(buffer)  expected = next(id)  connected = **true**  pending = 0  info(“receiver connected”) // (\*)  } **else** **if** (connected) {  writeAck(id, crcid)  **if** (id == expected) {  onReceived(crcid, msg) // (\*\*)  expected = next(id)  pending = 0  accept(crcid)  } **else** **if** (after(id, expected)) {  **if** (++pending < *MAX\_PENDING*) {  add(buffer[id)  buffer[id] = msg  } **else** {  connected = **false**  deleteAll(buffer)  error(“connection failed”)  }  }  }  }  **accept**(crcid) {  **if** (buffer[expected] != **nil**) {  onReceived(crcid, buffer[expected]) // (\*\*)  delete(buffer[expected])  expected = next(expected)  accept(crcid)  }  }  // (\*) if you make full-duplex protocol, you can call server’s sender.connect() from here  // (\*\*) onReceived() must NOT throw exceptions |
| // === COMMON ===  **const** *SYN* = 0, *MAX\_ATTEMPTS* = 8, *PERIOD* = 10ms  **const** *MAX\_PENDING* = 5  **const** *MIN\_SRTT* = 2, *DEFAULT\_SRTT* = 6, *MAX\_SRTT* = 12  **const** *RC* = 0.8, *AC* = 2.5  **next**(n) {  **if** (*RING\_ALGORITHM*)  result = n+1 // (n+1) % N (\*)  **else** result = n+1  **return** result != *SYN* ? result : next(result)  }  **after**(x, y) {  **if** (*RING\_ALGORITHM*)  **return** (y-x) > N/2 // (y-x+N) % N > N/2 (\*)  **return** x > y  }  // (\*) formula OK if your data type is a closed ring |
| // === ACRONYMS ===  // SYN = synchronization ID  // RTT = round trip time (ticks)  // SRTT = smoothed RTT  // AC = assurance coefficient  // RC = RTT coefficient  // CRC\_ID = crypto random connection id |