qa-tcp.lisp

```
; Measuring TCP roundtrip time with SRTT (using Karn's algorithm)

; foldmap: (func init Xs) -> Ys. Where:
; func: (Y X) -> Y; init: Y; Xs: [X]; Ys: [Y]
(defun foldmap (func init Xs)
  (if (null Xs) nil  ; (list init)
    (let ((Y (apply func (list init (car Xs)))))
      (cons Y (foldmap func Y (cdr Xs))))))

; foldmap2: (funcY funcZ initZ Xs) -> ((Y, Z)s). Where:
; funcY: (Y X) -> Y; funcZ: (Z Y X) -> Z; initZ: Z; Xs: [X]; Ys: [Y]
(defun foldmap2 (funcXY funcXZ initY initZ Xs)
  (if (null Xs) nil
    (let ((Yn (apply funcXY (list initY (car Xs))))
          (Zn (apply funcXZ (list initZ initY (car Xs)))))
      (cons (list Yn Zn) (foldmap2 funcXY funcXZ Yn Zn (cdr Xs))))))

; SRTT_i = (\alpha * SRTT_{i-1}) + ((1 - \alpha) * S_{i})
; where:
;   SRTT_i: Smoothed Round-Trip Time at time i
;   S_i   : Sample RTT at time i
;   \alpha: Smoothing factor, 0 < \alpha < 1
;   SRTT_0: Initial value of SRTT

; and, to determine the retransmission timeout (RTO) value:
;  RTO_i = \beta * SRTT_i, use beta > 1

; (srtt  srtt_{i-1} sample \alpha)
; returns the new SRTT value
(defun srtt_next (srtt sample alpha)
  (+ (* alpha srtt) (* (- 1 alpha) sample)))

; then, redefine srttListFold using foldmap and
; a partial application of srtt_next
(defun srttListFold (fsrtt srtt0 Ss alpha)
  (foldmap
    (lambda ; partial apply alpha
      (srtt0 sample) ; now this is a function takes (Y X)->Y.
      (apply fsrtt (list srtt0 sample alpha)))
    srtt0 Ss))

; a-1
; A TCP sneder's SRTT is 180 ms,
; but then a routing change occurs,
```

```
; fter which all measured RTTs are 110ms.
; How many measurements of the new RTT are required
; before SRTT drops below 130ms?

(defun srtt_uniform_target (fsrttNext init unisample alpha target)
  (labels ((srtt_uniform_target_inner (srttNow target comparator iter)
    (if (funcall comparator srttNow target) iter
      (srtt_uniform_target_inner
        (funcall fsrttNext srttNow unisample alpha)
        target comparator (+ 1 iter)))))
  (srtt_uniform_target_inner init target #'< 0)))

; rttvar <- (1 - \beta) * RTTVAR + \beta * |SRTT - R'|
(defun rttvar_next (rttvar srtt sample beta)
  (+ (* (- 1 beta) rttvar)
     (* beta (abs (- sample srtt)))))

; use foldmap2 since need to update both srtt and rttvar
; (foldmap2 funcXY funcXZ initY initZ Xs)
; fsrtt: (srtt sample alpha) -> srttNext
; frttvar: (rttvar srtt sample beta) -> rttvarNext
(defun rttvarListFold (fsrtt frttvar srtt0 rttvar0 Ss alpha beta)
  (foldmap2
    ; funcXY: sample -> srttNext
    (lambda (srtti R) (funcall fsrtt srtti R alpha))
    ; funcXZ: sample -> rttvarNext
    (lambda (rttvari srtti R) (funcall frttvar rttvari srtti R beta))
    ; initY: new srtt; initZ: new rttvar
    (funcall fsrtt srtt0 (car Ss) alpha)
    (funcall frttvar rttvar0 srtt0 (car Ss) beta)
    ; Xs: samples
   Ss))
; e.g.:
; (rttvarlistfold #'srtt_next #'rttvar_next 180 90 '(110 110 110) 0.8 0.7)

; after srtt and rttvar is calculated,
; RTO <- SRTT + 4 * RTTVAR
(defun rto_next (srtt rttvar)
  (+ srtt (* 4 rttvar)))

; rtoListFold: (srtt rttvar srtt0 rttvar0 Ss alpha beta) -> RTOs
; this is simple, first get listfold from rttvarListFold
; then just do a map to get the RTOs
(defun rtoTripletListFold (fsrtt frttvar srtt0 rttvar0 Ss alpha beta)
 (let ((srttrttvarL
         (rttvarListFold fsrtt frttvar srtt0 rttvar0 SS alpha beta)))
```

```
  (let ((rtoL (mapcar (lambda (x) (apply #'rto_next x)) srttrttvarL)))
    (mapcar (lambda (t2 one) (list (car t2) (cadr t2) one))
            srttrttvarL rtoL))))

; just pull out the rto from the triplet
(defun rtoListFold (fsrtt frttvar srtt0 rttvar0 Ss alpha beta)
  (mapcar (lambda (triplet) (caddr triplet))
          (rtoTripletListFold fsrtt frttvar srtt0 rttvar0 Ss alpha beta)))
; (rtolistfold #'srtt_next #'rttvar_next 180 90 '(110 110 110) 0.8 0.7)

; get the triplets of (srtt, rttvar, rto)
; from next sample
(defun rtoTriplet (fsrtt frttvar frto srtt rttvar sample alpha beta)
  (let ((rttvarNext (funcall frttvar rttvar srtt sample beta))
        (srttNext (funcall fsrtt srtt sample alpha)))
    (let ((rtoNext (funcall frto srttNext rttvarNext)))
      (list srttNext rttvarNext rtoNext))))

; a-2:
; find out how many measurements (of uniformal R) are required
; to bring RTO below (T) ms.
; rto_uniform_target

; first, need a func that runs the srtt_next and rttvar_next enough times
; for that uniform R value so it converges.
; returns the triplet of (srtt, rttvar, rto)
(defun rto_uniform_ntimes
  (fsrttNext frttvar frto usually unusual mod alpha beta n)
  (labels ((rto_uniform_ntimes_inner (srtt rttvar iter)
    (let ((sample (if (= (mod iter mod) 0) unusual usually)))
      (let ((triplet (rtoTriplet fsrttNext frttvar frto srtt
                                 rttvar sample alpha beta)))
        (if (< iter n)
          (rto_uniform_ntimes_inner (car triplet) (cadr triplet) (+ 1 iter))
          triplet)))))
  (rto_uniform_ntimes_inner 0 0 0)))

; then, found out max mod that brings RTO below target
(defun q2_query
  (n target)
    (labels ((q2_query_inner (iterMod)
      (let ((triplet (rto_uniform_ntimes #'srtt_next #'rttvar_next #'rto_next
                      200 600 (+ 1 iterMod) 0.8 0.7 n)))
        (if (< (caddr triplet) target) (- iterMod 1)
          (q2_query_inner (+ 1 iterMod))))))
            (q2_query_inner 1)))
```

qa-out.lisp

```
(load "qa-tcp.lisp")
(let ((sample_c (srtt_uniform_target #'srtt_next 180 110 0.8 130)))
  (let ((srttList
          (srttListFold 'srtt_next 180
            (loop repeat sample_c collect 110) 0.8)))
    (format t "~d" "a-1-1:")(terpri)
    (format t "~d" "need ")(write sample_c)(format t "~d" " samples")
    (format t "~d" " to drop below 130ms")(terpri)
    (format t "~d" "srtt for first ")(write sample_c)
    (format t "~d" " samples:")(terpri)
    (write srttList)(terpri)))

(let ((sample_c (q2_query 10000 600)))
  (terpri) (terpri)
  (format t "~d" "a-1-2:") (terpri)
  (format t "~d" "need ") (write sample_c) (format t "~d" " samples") (terpri)
  (format t "~d" " so next 600ms RTT will trigger a timeout") (terpri)
  (format t "~d" " the first ")(write sample_c)(format t "~d" " samples," )
  (format t "~d" " (srtt, rttvar, rto) are")

  (write (loop for i from 1 to sample_c collect
           (rto_uniform_ntimes #'srtt_next #'rttvar_next #'rto_next
                               200 600 i 0.8 0.7 10000)))
  (terpri)(terpri)
  (format t "~d" "the first 60 samples in the smallest exceeding case are:")
  (terpri)
  (write (rtoTripletListFold #'srtt_next #'rttvar_next 0 0
    (loop for i from 1 to 60 collect (if (= (mod i (+ sample_c 1)) 0) 200 600))
    0.8 0.7)))
```