Optimal Offline TCP Sender Buffer Management Strategy

Mugurel Ionut Andreica, Nicolae Tapus

Optimal Offline TCP Sender Buffer Management Strategy

Mugurel Ionut Andreica*, Nicolae Tapus*
* *Politehnica University of Bucharest, Computer Science Department, Bucharest, Romania*
{mugurel.andreica, nicolae.tapus}@cs.pub.ro

Abstract

The Transmission Control Protocol (TCP) uses a sliding window in order to enforce flow control. The receiver advertises its available buffer space to the sender, which cannot transmit more data than the advertised space. Transmitted data is first copied from application buffers into TCP buffers and from there it is sent through the network. In this paper we propose a model which characterizes the sender's behavior throughout the duration of a TCP conversation. The model is suitable in the case of powerful, but variably loaded senders, slow receivers, fast connections and moderate amounts of data transmitted. For this model we present an $O(n \cdot \log^2 n)$ algorithm which computes the minimum processing time spent by the sender, if the window sizes advertised by the receiver and the sender's load are known in advance. The solution is based on an algorithmic framework for the segment tree data structure, which we introduce in this paper.

1. Introduction

TCP uses a sliding window mechanism in order to enforce flow control and not overwhelm the receiver with too much data. This mechanism is particularly useful when a fast, powerful sender communicates with a slow receiver or with one having limited resources (small amounts of buffer space). However, these situations are rather stressful for the sender, which needs to copy data from application buffers into TCP buffers many times. If the sender is loaded by many applications performing different tasks, copying data between buffers might not always take the same amount of time. Because of this, it might happen that the sender becomes a performance bottleneck.

In this paper we propose a model for characterizing the sender's behavior throughout the life time of a TCP connection. Based on this model, we developed an $O(n \cdot \log^2 n)$ algorithm for computing the minimum total processing time for the sender when the window sizes advertised by the receiver and the system load of the sender are known in advance. The algorithm can only be used offline, either when accurate estimates of the required parameters are known, or when detailed traces of TCP conversations are available.

The rest of the paper is structured as follows. In Section 2 we present our model for the sender's behavior and, based on it, we define the problem of minimizing the total processing time. In Section 3 we present the algorithm which computes the minimum total processing time on the sender side. In Section 4 we present related work, while in Section 5 we draw some conclusions and present future work.

2. The TCP Sender Behavior Model

After the initial three-way handshake, the TCP sender sends only as much data as the last window size advertised by the receiver. We will assume that throughout the TCP conversation, the receiver advertised its window size n times. The window size at the i^{th} advertisement is $w_i \geq 0$. After each advertisement i is received, the sender sends the next w_i bytes of data to the receiver and waits for the next advertisement. We will assume that the total amount of transmitted bytes is equal to the sum of the window sizes.

When receiving the i^{th} advertisement, the sender already has in its TCP buffer the next $bb_i \geq 0$ bytes to be sent. If $bb_i < w_i$, the sender will copy cb_i bytes ($w_i - bb_i \leq cb_i$) into its TCP buffer and then send w_i bytes to the receiver. The sender may choose to copy some bytes into the TCP buffer even when $bb_i \geq w_i$. The time needed to copy x bytes into the TCP buffer at the time of the i^{th} advertisement is

$$t_{copy,i}(x)=t_{setup,i} + t_{byte,i}\cdot x \text{ , } x>0. \quad \textbf{(1)}$$

There are two components comprised in the copy time. The first one ($t_{setup,i}$) is the setup time needed to initiate the transfer. The second one ($t_{byte,i}$) is the time required for copying a byte from the application buffer into the TCP buffer. If no byte is copied, then the copy time is 0 ($t_{copy,i}(0)=0$). The two time parameters ($t_{setup,i}$ and $t_{byte,i}$) depend on the system load, which is variable. We will call a *time step* the moment when a window size advertisement is received. Given the values w_i, $t_{setup,i}$ and $t_{byte,i}$ for each of the n time steps, the processing time depends on the number of bytes cb_i copied between the application buffer and the TCP buffer at each step. We will assume that the TCP buffer capacity is very large (infinite) compared to the total number of bytes transferred. Thus, it would be possible for the sender to copy all the bytes in the TCP buffer during the first time step, if such a strategy were considered convenient. Although buffer space may be large enough for the data transferred on a single TCP connection, we must consider the fact that this buffer might be shared by several connections. We will take this into consideration by adding another parameter sc_i to every time step. This parameter represents the cost of storing one byte in the TCP buffer from time step i to time step i+1. sc_i will also be expressed in time units, because using up one byte of the TCP buffer increases the processing times of other TCP connections, which will have less buffer space at their disposal. Thus, this parameter represents the amount by which the processing times of other TCP conversations increase if one byte is stored for the current connection in the TCP buffer from time step i to the next. The total processing time of the sender is:

$$TPT = \sum_{i=1}^{n} t_{copy,i}(cb_i) + \sum_{i=2}^{n}\sum_{j=1}^{i-1} sc_{i-1}\cdot(cb_j - w_j)\cdot \quad \textbf{(2)}$$

Given the values of all the parameters at each time step, the total processing time depends only on the number of bytes copied at each step. The values cb_i for which TPT is minimum define an optimal sender buffer management strategy. It is interesting that, with another meaning given to the parameters, this problem is equivalent to the economic lot sizing problem [4,5,6].

3. An Efficient Algorithm

Solving the problem starts with a non-trivial observation. Let's assume that at the end of time step j, the TCP buffer contains X bytes (after copying the cb_j bytes planned for that time step). Let's also assume that among these X bytes, $X_1>0$ were copied in the buffer at time step i_1 and $X_2>0$ were copied in the buffer at time step i_2. The processing times incurred by the X_1 and X_2 bytes are given by the equations below, where A_1, B_1, A_2 and B_2 can be inferred easily:

$$PT_1=A_1+(t_{byte,i1}+sc_{i1} + sc_{i1+1}+\ldots+sc_{j-1})\cdot X_1= \quad \textbf{(3)}$$
$$A_1+B_1\cdot X_1,\ 0\le A_1\le t_{setup,i1},\ B_1\ge 0 \text{ ,}$$
$$PT_2=A_2+(t_{byte,i2}+sc_{i2}+sc_{i2+1}+\ldots+sc_{j-1})\cdot X_2= \quad \textbf{(4)}$$
$$A_2+B_2\cdot X_2,\ 0\le A_2\le t_{setup,i2},\ B_2\ge 0 \text{ .}$$

The processing time incurred by the X_1+X_2 bytes is $PT_{12}=PT_1+PT_2$. If all the X_1+X_2 bytes had been copied in the TCP buffer at time step i_1, the processing time would have been $PT_A=A_1+B_1\cdot(X_1+X_2)$. Similarly, if all the X_1+X_2 bytes had been copied in the TCP buffer at time step i_2, the processing time would have been $PT_B=A_2+B_2\cdot(X_1+X_2)$. We will show that either PT_A or PT_B must be less than or equal to PT_{12}. We have

$$PT_A-PT_{12}=B_1\cdot X_2-B_2\cdot X_2-A_2=(B_1-B_2)\cdot X_2-A_2 \text{ ,} \quad \textbf{(5)}$$
$$PT_B-PT_{12}=(B_2-B_1)\cdot X_1-A_1. \quad \textbf{(6)}$$

It is obvious that either $B_1-B_2\le 0$ or $B_2-B_1\le 0$, i.e. either $PT_A\le PT_{12}$ or $PT_B\le PT_{12}$. Thus, in an optimal solution, the time steps 1,…,n can be split in a number K of intervals $[l_1=1, r_1]$, $[l_2=r_1+1, r_2]$, …, $[l_K=r_K-1+1, r_K=n]$, such that at every time step l_i, the number of bytes copied in the TCP buffer is

$$cb_{l_i} = \sum_{j=l_i}^{r_i} w_j \quad \textbf{(7)}$$

At the time steps j which are not the first time steps of some interval, no bytes will be copied in the TCP buffer and the required bytes will already be there. We will first present a simple $O(n^2)$ dynamic programming solution. Then, we will introduce the segment tree data structure and we will show how we can use it in order to improve the time complexity to $O(n\cdot\log^2 n)$.

3.1. An $O(n^2)$ dynamic programming solution

We will compute an array *mintpt*, where *minpt[i]* is the minimum total processing time if i were the last time step. For each step i, all the possible steps j are considered, such that j is the first time step of the interval ending at i. For each value of j, the total processing time is computed and the minimum value is maintained. The pseudocode is given below:

```
mintpt[0]=0
for i = 1 to n
  mintpt[i]=+Infinity; wtotal=0; storagept=0
  for j = i downto 1
    storagept=storagept+sc_j·wtotal
    wtotal=wtotal+w_j
    totalpt=t_setup,j+t_byte,j·wtotal+mintpt[j-1]+storagept
    if (totalpt<mintpt[i]) then
      mintpt[i]=totalpt
```

3.2. The Segment Tree Data Structure

The segment tree [7] is a binary tree structure used for performing operations on an array v with n cells. Each cell i ($1 \le i \le n$) contains a value v_i. Each node p of the tree has an associated interval [p.left, p.right], corresponding to an interval of cells. If the node p is not a leaf, then it has two sons: the left son (p.lson) and the right son (p.rson). The interval of the left son is [p.left, mid] and the right son's interval is [mid+1, p.right], where *mid=floor((p.left+p.right)/2).*

The leaves are those nodes whose associated interval contains only one cell. The interval of the root node is [1,n]. It is obvious that the height of the segment tree is O(log(n)). The tree can be built in O(n) time. Query operations consist of computing a function on the values of a range of cells [a,b] (range query) or on retrieving the value of a single cell (point query).

Range Query(a, b): compute $qFunc(v_a, v_{a+1},...,v_b)$.

Analogously, we have point and range updates:

Range Update(u, a, b): v_i=*uFunc*(u, v_i), $a \le i \le b$.

The *qFunc* function must be binary and associative, i.e. $qFunc(v_a,..,v_b)=qFunc(v_a,qFunc(v_{a+1},..,qFunc(v_{b-1}, v_b)..))$ and $qFunc(a,qFunc(b,c))=qFunc(qFunc(a,b),c)$. We must also have $uFunc(x,y)=uFunc(y,x)$. Only values v_i with O(1) size are considered (numbers and tuples with a fixed number of elements). In order to use the segment tree, we introduce here an algorithmic framework, consisting of the functions from Table 1.

Table 1. Segment Tree Framework Functions

Update Fuctions	Query Functions
STpointUpdate	STpointQuery
STrangeUpdate	STrangeQuery
STpointUpdateNode	STpointQueryNode
STrangeUpdateNodeFit	STrangeQueryNodeFit
STrangeUpdateNodeIncl	STrangeQueryNodeIncl

In order to perform a range update, we call the *STrangeUpdate* function with the root of the segment tree as the node argument, the update parameter and the update interval. If the update interval is equal to the node's interval, then the *STrangeUpdateNodeFit* function is called; otherwise, if the intersection between the interval of one of the node's sons and the update interval is non-empty, the function is called with that son as the node argument and with the interval intersection as the update interval. The function visits O(log(n)) tree nodes. A range query (*STrangeQuery*) works similarly. A point query (*STpointQuery*) or update (*STpointUpdate*) on a position i traverses the tree upwards, from the leaf with the interval [i,i] towards the root. The leaf node is either found directly or by traversing the tree downwards from the root towards the leaf. When using range queries and range updates together, we must also have a "multiplication" operator *mop* which computes the effects of an update upon the result of a query on a range of points. *qFunc* and *uFunc* must be able to handle *uninitialized* values. If one of their two

arguments is *uninitialized*, the functions must simply return the other argument. In the following examples, we will not present this part.

STpointUpdate(node, u, i):

```
// node is the tree leaf with the interval [i,i]
while (node≠null) do
  STpointUpdateNode(node, u)
  node=node.parent
```

STpointUpdateNode(node, u):

```
if (node is a leaf) then
  node.qagg=uFunc(u, node.qagg)
else node.qagg=qFunc(node.lson.qagg, node.rson.qagg)
```

STrangeUpdate(node, u, a, b):

```
if ((a=node.left) and (node.right=b)) then
  // the update "stopped" at this node
  STrangeUpdateNodeFit(node, u)
else
  lson, rson = left and right son of the current tree node
  if ((a≤lson.right) and (lson.left≤b)) then
   STrangeUpdate(lson,u,max(a,lson.left),min(b, lson.right))
  if ((a≤rson.right) and (rson.left≤b)) then
   STrangeUpdate(rson,u,max(a,rson.left),min(b rson.right))
  STrangeUpdateNodeIncl(node, u, a, b)
```

STrangeUpdateNodeFit(node, u):

```
node.uagg=uFunc(u, node.uagg)
// update the query aggregate
node.qagg=uFunc(mop(u,node.left,node,right),node.qagg))
```

STrangeUpdateNodeIncl(node, u, a, b):

```
node.qagg=uFunc(mop(node.uagg, node.left, node.right), qFunc(node.lson.qagg, node.rson.qagg))
```

STpointQuery(node, i):

```
// node is the tree leaf with the interval [i,i]
q=node.qagg; node=node.parent
while (node≠null) do
  q=uFunc(STpointQueryNode(node), q)
  node=node.parent
return q
```

STpointQueryNode(node):

```
return node.uagg
```

STrangeQuery(node, a, b):

```
if (a=node.left and node.right=b) then
  // the query "stopped" at this node
  return STrangeQueryNodeFit(node)
else
  q=uninitialized
  if ((a≤node.lson.right) and (node.lson.left≤b)) then
    q=qFunc(q, STrangeQuery(node.lson, max(a, node.lson. left), min(b, node.lson.right))
  if ((a≤node.rson.right) and (node.rson.left≤b)) then
    q=qFunc(q, STrangeQuery(node.rson, max(a,node.rson. left), min(b, node.rson.right))
  return uFunc(STrangeQueryNodeIncl(node, a, b), q)
```

STrangeQueryNodeFit(node):

```
return node.qagg
```

STrangeQueryNodeIncl(node, a, b):

```
return mop(node.uagg, a, b)
```

Each node of the tree has a pointer to its parent (this pointer is *null* for the root of the tree) and stores two values: *uagg*, an update aggregate and *qagg*, a query aggregate. *node.uagg* is the aggregate value of all the update

parameters of the *STrangeUpdateNodeFit* function calls on *node*. At any moment, *node.qagg* is the answer to the range query on the interval [node.left, node.right] if the updates which "stopped" at higher levels are ignored. When building the tree, *qagg* is set to v_i (for a leaf with the interval [i,i]), while for nodes p which are not leaves, *p.qagg* is set to *qFunc(p.lson. qagg, p.rson.qagg). qagg* is modified by the update functions. In *STrangeUpdateIncl*, *qagg* is recomputed from scratch, after an update changed the *qagg* values of the node's sons. Alternatively, for some update and query functions, we could modify qagg directly:

node.qagg=**uFunc**(**mop**(*u, a, b*), *node.qagg*)

uagg and *qagg* are used together only when range queries are used together with range updates. In the case of point queries with range updates, only the *uagg* values are meaningful; similarly, only the *qagg* values are meaningful in the case of point updates with range queries. Common update and query functions can be easily integrated into the framework. For example, with *uFunc(x,y)=(x+y)*, *qFunc(x,y)=(x+y)* and *mop(u,a,b)= u·(b-a+1)*, we can support point and range sum queries, together with point and range addition updates. For *uFunc(x,y)=x+y*, *qFunc(x,y)=min(x,y)* and *mop(u,a,b) =u*, we can support point and range minimum queries, together with point and range addition updates. We can also consider point and range multiplication updates, *uFunc(x,y)=x·y*, with point and range queries: *qFunc(x,y)=x·y* (with *mop(u,a,b)=*u^{b-a+1}), *qFunc(x,y)= min(x,y)* and *qFunc(x,y)=(x+y)* (with *mop(u,a,b)=u*). With *mop(u,a,b)=u*, we can support range queries and updates for some bit functions (where v_i=0 or 1). For *uFunc(x,y)=(x or y)* or *uFunc(x,y)=(x and y),* we can have *qFunc(x,y)=(x and y)* and *qFunc(x,y)=(x or y)*. For the *and* update, we can also have *qFunc(x,y)=(x xor y)*. We can support range *xor* updates and queries (*uFunc(x,y) = qFunc(x,y) = (x xor y)*), but with *mop(u,a,b)=if (((b-a+1) mod 2)=0) then 0 else u*. In order to obtain any combination of bit functions, we notice that the result of a query depends only on the number of 0 and 1 values (cnt_0, cnt_1) in the query range: if (cnt_1>0) then *or* returns 1; if (cnt_1 mod 2=1) then *xor* returns 1; if (cnt_0=0) then *and* returns 1. Thus, we will work with (cnt_0, cnt_1) tuples as values. We will also consider the *conceptual values* cv_i, which are the numerical values we conceptually work with. We have v_i=(1-cv_i, cv_i). A query asks for the number of 0 and 1 conceptual values in the query range and an update changes this number according to the bit function used. Any combination of point and range queries and updates is supported with the functions below:

bitTupleQuery(($cnt_{0,x}$, $cnt_{1,x}$), ($cnt_{0,y}$, $cnt_{1,y}$)):
return ($cnt_{0,x}$+$cnt_{0,y}$, $cnt_{1,x}$+$cnt_{1,y}$)

bitTupleUpdate((1-u, u), (cnt_0, cnt_1), func):
if *(func=and)* **and** *(u=0)* **then return** *(cnt_0+cnt_1, 0)*
else if *(func=or)* **and** *(u=1)* **then return** *(0, cnt_0+cnt_1)*
else if *(func=xor)* **and** *(u=1)* **then return** *(cnt_1, cnt_0)*
else return *(cnt_0, cnt_1)*

For other types of operations, the framework can only support combinations like point queries with range updates or range queries with point updates. For instance, if the update function has the effect of setting all the values in a range to the same value *s* (range set), we will again need to work with tuples: the values v_i and the update parameters u will have the form *(numerical value, time_stamp)*. We need to have a *timestamp()* function which returns increasing values upon successive calls. We can use a global counter as a time stamp, which is incremented at every call. The initial numerical values are assigned an initial time stamp and every update parameter gets a more recent time stamp. The update function is:

uFunc((v_x, t_x), (v_y, t_y)):
if (t_x>t_y) **then return** *(v_x, t_x)* **else return** *(v_y, t_y)*

With these definitions, a point query function call on a position i will return the last update parameter on the path from the leaf with the [i,i] interval, to the root.

A useful range query function (used together with point updates) is finding the maximum sum segment (interval of consecutive cells) fully contained in a range of cells [a,b] (see [3] for this problem without updates). Conceptually, the value of a cell i is a number cv_i, but in the framework we will use tuples consisting of 4 values: *(totalsum, maxlsum, maxrsum, maxsum)*. Assuming that these values correspond to an interval of cells [c,d], we have the following definitions:

$$\text{totalsum} = \sum_{p=c}^{d} cv_p \qquad \text{maxlsum} = \max_{c-1 \le q \le d} \sum_{p=c}^{q} cv_p$$

$$\text{maxrsum} = \max_{c \le q \le d+1} \sum_{p=q}^{d} cv_p \qquad \text{maxsum} = \max_{\substack{c \le q \le d \\ q-1 \le r \le d}} \sum_{p=q}^{r} cv_p$$

In the framework, a value v_i will be a tuple corresponding to the interval [i,i]. If $cv_i<0$, then $v_i=(cv_i, 0, 0, 0)$; otherwise, $v_i=(cv_i, cv_i, cv_i, cv_i)$. The point update function changes the value of cv_i of a cell i and then recomputes v_i. The *qFunc* function is given below:

qFunc((t_x,ml_x,mr_x,m_x), (t_y,ml_y,mr_y,m_y)):
return $(t_x+t_y, max\{ml_x, t_x+ml_y\}, max\{mr_y, t_y+mr_x\}, max\{m_x, m_y, mr_x+ml_y\})$

3.3. The O(n·log²n) solution

This solution is based on the segment tree data structured, presented previously. Basically, we want to compute the same array *mintpt* as in the other solutions. When computing mintpt[i], we can choose the time step j, which is the beginning of the time step interval ending at i, from the set {1,2,…,i}. We will define the family of functions $f_j(x)$, representing the minimum total processing time for the first x time steps, if x is considered the last time step and j is the first time step of the interval ending at x. For each i, we will have to find the function f_j whose $f_j(i)$ value is minimum. We will first introduce two new arrays, scp and wp, representing the prefix sums of the arrays sc and w:

$$scp[i] = \sum_{j=1}^{i} sc_j \quad \textbf{(8)}, \qquad wp[i] = \sum_{j=1}^{i} w_j \quad \textbf{(9)}.$$

A function f_j is defined on the interval [j-1,n]. The first value, $f_j(j-1)$ does not have a practical meaning, as a function f_j is considered only at the time steps j,..,n; it is introduced to simplify the analysis. We have:

$$f_j(j-1)=\text{mintpt}[j-1]+t_{setup,j}\,, \qquad \textbf{(10)}$$

$$f_j(j) = \text{mintpt}[j-1]+t_{setup,j}+t_{byte,j}\cdot w_j = f_j(j-1)+t_{byte,j}\cdot w_j\,. \quad \textbf{(11)}$$

$$df_j(x) = f_j(x) - f_j(x-1) = w_x \cdot \left(t_{byte,j} + \sum_{p=j}^{x-1} sc_p \right),\ x \ge j \quad \textbf{(12)}$$

The difference between two consecutive values of a function f_j ($df_j(x)$) contains the processing time of copying w_x bytes in the TCP buffer at time step j and the sum of processing times incurred by storing the w_x bytes in the TCP buffer until time step x. Using the prefix sum arrays, the difference can be rewritten:

$$df_j(x)=t_{byte,j}\cdot w_x + (scp[x-1] - scp[j-1])\cdot w_x = \quad \textbf{(13)}$$
$$scp[x-1]\cdot w_x + (t_{byte,j} - scp[j-1])\cdot w_x.$$

The difference is now composed of two terms: the term $scp[x-1]\cdot w_x$, which depends only on the point at which the function is evaluated and a term which is composed of two factors, one of which is constant for a given function f_j and the other one depends only on the point where the function is evaluated. The factor which is constant for a function f_j will be denoted by

$$p_j=t_{byte,j}-scp[j-1]. \qquad \textbf{(14)}$$

We will now slightly change the definitions of the functions and remove the term $scp[x-1]\cdot w_x$. This term does not influence the relative ordering of the values of the functions f_j. After computing *mintpt[n]* using the new definitions of the functions, we will add at the end the sum of all the excluded terms:

$$S_{terms} = \sum_{i=1}^{n} scp[i-1]\cdot w_i \qquad \textbf{(15)}$$

With the new definitions, the equation for $df_j(x)$ is $df_j(x)=f_j(x)-f_j(x-1)=p_j\cdot w_x$. The initial values $f_j(j-1)$ do not change. If we associate to each time step i an x-coordinate wp[i], we can change the definitions of the functions further and obtain some new functions g_j, defined on the interval [wp[j-1], wp[n]]: $g_j(x)=g_j(wp[j-1])+p_j\cdot(x-wp[j-1])$, where $g_j(wp[j-1])=f_j(j-1)$. It is easy to see that the relationship between the functions g_j and f_j is: $g_j(wp[x])=f_j(x)$.

The functions g_j are half lines and, thus, have the following useful property: the values of each function g_j are the globally minimum values among all the functions either on an interval of x-coordinates [lx_j, rx_j] or none of its values is a global minimum. The proof is easy. Let's assume that the function g_j has the globally minimum values on two disjoint intervals [lx_{j1}, rx_{j1}] and [lx_{j2}, rx_{j2}], with $lx_{j2}>rx_{j1}$. There are two possibilities: The first one is that there exists some function g_k, such that $g_k(x)>g_j(x)$, for $x\le rx_{j1}-\varepsilon$ and $g_k(x)<g_j(x)$ for $x\ge rx_{j1}$. In order for this to happen, the function g_k must have a slope p_k which is smaller than the slope p_j of the function g_j. But if this is the case, then $g_k(x)<g_j(x)$, for any $x\ge rx_{j1}$, so function g_j can never become minimum again. The second possibility is that a function g_k "started" at $x=rx_{j1}$ (that is, rx_{j1} is the first point on its definition domain) and its values is minimum. But, from the

way the functions are defined, the first value of a function g_k is equal to the minimum value of the functions g_p (p<k), plus $t_{setup,j}$ (which is positive), so $g_k(rx_{j1})$ cannot be the globally minimum value.

With the observation that each function has globally minimum values on at most one interval, we can use a segment tree for storing half lines. The n+1 cells for which the segment tree is built correspond to the points wp[1], wp[2], …, wp[n] and wp[n+1]=wp[n]+1. The queries will be point queries and the update operations will be of the type *"range set"*. Thus, each value of the segment tree consists of a pair (numerical value, timestamp). The pseudocode is given below:

compute *scp, wp (scp[0]=wp[0]=0) and* S_{terms}
mintpt[0]=0
for *i = 1* **to** *n*
 ginit[i]=mintpt[i-1]+$t_{setup,i}$
 p[i]=$t_{byte,i}$*-scp[i-1]*
 // find the interval [lx_i,rx_i] on which g_i is globally minimum
 [lx_i*,* rx_i*]=***find_interval(*$\underline{i}$**)**
 if (*$lx_i \le rx_i$*) **then**
  STrangeUpdate(*segment_tree_root , u(i),* lx_i *,* rx_i**)**
 *mintpt[i]=***get_min(***i***)**
return *mintpt[n]+*S_{terms}

The *u(i)* argument of the update function is *(i, timestamp())*. The *get_min* function returns the globally minimum value of the functions g_j at the point wp[i].

get_min(i):
*(k,t)=***STpointQuery(***leaf node with the interval [i,i], i***)**
return *ginit[k]+(wp[i]-wp[k-1])·p[k]*

In the *find_interval* function we binary search for the first time step lx_i (between i and n) where the value $g_i(wp[lx_i])$ is the smallest among all the functions' values. In a similar manner, the last time step rx_i is binary searched, too. In order to find lx_i, we first need to observe how the function g_i's values change relative to the globally minimum value of the other functions. In general (excluding particular cases), $g_i(wp[i-1])$ is larger than the minimum value. Then, the difference between $g_i(x)$ and the minimum value at point x decreases until $g_i(x)$ becomes smaller than the former minimum value at point x. The function g_i is minimum until $x=rx_i$, after which the difference between $g_i(x)$ and the minimum value at point x increases, for $x>rx_i$. This type of behavior suggests that a binary search on the differences between two consecutive values of the function $h(x)=g_i(x)$*-get_min(x)* is appropriate. In order to handle values $w_i=0$, we use an array *wpnext*, where *wpnext[i]* ($i \le n$) is the next position j>i, such that wp[j]>wp[i], i.e. *wpnext[i]=if (wp[i+1]> wp[i]) then (i+1) else wpnext[i+1]*. We will also repeatedly decrease the value of n, until $w_n>0$ (or n=0).

left=i; right=n; lx_i*=n+1*
while (*left*$\le$*right*) **do**
 mid = (left+right) div 2 // integer division
 gi_mid_1 = ginit[i]+(wp[mid]-wp[i-1])·p[i]
 dg=p[i]·(wp[wpnext[mid]]-wp[mid])
 fmin1=get_min(mid); fmin2=get_min(wpnext[mid])
 dmin=fmin2-fmin1
 if (*gi_mid_1<fmin1*) **then** *{* lx_i*=mid; right=mid-1 }*
 else if (*dg<dmin)* **then** *left=mid+1*
 else *right=mid-1*

In the end, lx_i contains the left endpoint of the interval in which g_i is minimum (or $lx_i=n+1$ if such an interval does not exist). rx_i is computed analogously.

4. Related Work

TCP buffer management strategies have been proposed in many papers, for optimizing different performance metrics [1,2]. As far as we know, TCP sender buffer management has not been addressed from the perspective presented in this section. The (uncapacitated) economic lot sizing problem and different variations of it were studied extensively in many papers [4,5,6] and optimal O(n·log(n)) algorithms were proposed for solving it. The segment tree data structure [7] is used for solving many problems; computational geometry, online scheduling, advance

reservations and optimization problems are only a few domains in which the segment tree is used. Dynamic programming techniques based on the use of the segment tree were presented in [8].

5. Conclusions and Future Work

In this paper we introduced a new model for the behavior of the sender in a TCP conversation. Using this model, we presented a new $O(n \cdot \log^2 n)$ dynamic programming algorithm which computes the sender's minimum total processing time, when the receiver's advertised window sizes and the sender's system load are known in advance. In this paper we focused only on the theoretical aspects of the buffer management strategy (model definition and algorithm efficiency); thus, we leave the practical validation of our model for future work. Also as part of our future work, we intend to devise efficient, competitive, online algorithms for the problem. The model is similar to the uncapacitated single-item economic lot sizing problem, which was studied in many papers, but was never connected to the optimization of the TCP buffer management strategy. Although optimal $O(n \cdot \log(n))$ algorithms are known for this problem, our algorithm is easier to implement and the techniques employed by the algorithm are of interest by themselves, as they can be used for solving other problems, too. Moreover, we also introduced a novel, easy to use, algorithmic framework for using the segment tree data structure in any application.

6. References

[1] A. Cohen, and R. Cohen, "A dynamic approach for efficient TCP buffer allocation", *IEEE Transactions on Computers 51*, IEEE Press, 2002, pp. 303-312.

[2] D. Goldenberg, M. Kagan, R. Ravid, and M.S. Tsorkin, "Zero copy sockets direct protocol over Infiniband – preliminary implementation and performance analysis", *Proceedings of the 13th IEEE Symposium on High Performance Interconnects*, IEEE Press, 2005, pp. 128-137.

[3] K.-Y. Chen, and K.-M. Chao, "On the range maximum-sum segment query problem", *Discrete Applied Mathematics*, *155*, Elsevier, 2007, pp. 2043-2052.

[4] G.R. Bitran, and H.H. Yanasse, "Computational complexity of the capacitated lot size problem", *Manage. Sci. 28*, 1982, pp. 1174-1186.

[5] A. Federgruen, and M. Tzur, "A simple forward algorithm to solve general dynamic lot sizing models with n periods in O(nlogn) or O(n) time", *Manage. Sci. 37*, 1991, pp. 909-925.

[6] A. Wagelmans, S. Van Hoesel, and A. Kolen, "An O(nlogn) algorithm that runs in linear time in the Wagner-Whitin case", *Oper. Res. 40*, 1992, pp. 145-156.

[7] J.L. Bentley, "Solutions to Klee's Rectangle Problems", Technical Report, Carnegie-Mellon University, 1977.

[8] A. Aggarwal, and T. Tokuyama, "Consecutive interval query and dynamic programming on intervals", *Discrete Applied Mathematics 85*, Elsevier, 1998, pp. 1-24.