A Transmission Control Protocol with High Throughput of Using Low Earth-Orbit Satellite to Collect Data from the Floats on Sea Surface

Chia-Sheng Tsai[1*] Yu-Cheng Wang[1] Hsin-Kai Wang[2]

[1] Department of Computer Science and Engineering, Tatung University
Taipei 104, Taiwan, ROC
*Corresponding author: icstsai@gmail.com
starckwang@gmail.com

[2] Research and Development Department, Group, Laboratory, Emerson Network Power (Taiwan) Co. Ltd.
Taipei 105, Taiwan, ROC
memorywang0809@gmail.com

Received 2 September 2014; Revised 20 September 2014; Accepted 11 October 2014

Abstract. For the sea surface salinity (SSS) applications, the data collected by the floats needs to be uploaded as a satellite passes through. The floats distributed on sea surface are equipped with sensors to measure interested information to deliver to visible satellites passing through the sky. Also, a Low Earth-Orbit Satellite use transmission control protocol (TCP) to collect data from the floats on sea surface. Hence, a Low Earth-Orbit satellite communication is an important issue for SSS plan. TCP is a popular connection oriented transport layer protocol used for reliable data transfer in the satellite communication. In the evolution of network technology, the transmission control protocol is an important research topic. It is responsible for data transmissions of end users, flow control, congestion control, and other tasks. The paper focuses on the congestion control scheme. In TCP, the congestion control is used to maximize the data transmission without causing network congestion. There exist different congestion control schemes, whose performance in throughput and fairness varies. Some trade fairness for throughput and some vice versa. Our research goal is to achieve good performance on both throughput and fairness in the satellite communication for SSS plan. Through computer simulations, we are given to understand that our scheme can improve the throughput and preserve the fairness.

Keywords: sea surface salinity (SSS), transmission control protocol (TCP), satellite communication, congestion control

1 Introduction

Global warming is a tough challenge threatening the very existence of our civilization. Many research projects are dedicated to study various aspects of this critical phenomenon. National Aeronautics and Space Administration (NASA) in the United States and its international partners started project Aquarius/SAC-D to study sea surface salinity and its effects on global climate and ocean/ground water circulation [1]. In the Aquarius project, many Argo float sensors capable of collecting and distributing measured data have been built and deployed by voluntary participating countries around the world. In short, the architecture can be divided into three subsystems [2], the satellite, the Argo floats and the data center in the Fig. 1.

In the satellite subsystem, a Low Earth-Orbit (LEO) satellite needs to be launched and put into place to collect the data from the floats. The satellite uses two types of satellite-terrestrial communications [4][5]. One is between the satellite and the Argo floats and another is between the satellite and the ground data centers. When the satellite talks to the floats, the satellite is the sole source for the download channel. However, the upload bandwidth is usually considered contentious. In order to send the measured data, the floats fight to get access to the upload channel. In this paper, we will introduce a proposed scheme to improve the utilization of the upload channel. The main function of Argo floats is to measure and store scientific data on board. Various sensors are built into the float to sample or monitor the desired environmental parameters. Each Argo's float will pump fluid into an external bladder and rise to the surface over about 6 hours while measuring temperature and salinity at typically 10-day intervals [6][7]. The satellite can around the earth three times to complete the data transmission. In Fig. 2, the float descends and drifts for 10 days in around 1000 meters under the water. Then, it dives to around 2000 meters and, then, starts to emerge to the surface. After 2010 year, 70% of floats profile to depths

greater than 1500 meters. Besides, 20% profile to between 1000 meters and 1500meters. As it moves upward, it measures the temperature and sea surface salinity.

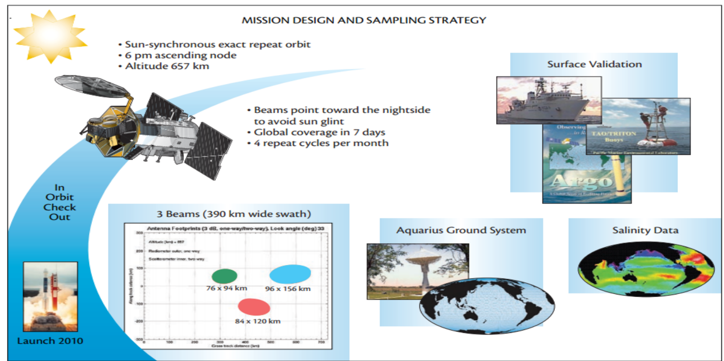

Fig. 1. The Aquarius/SAC-D mission concept [3]

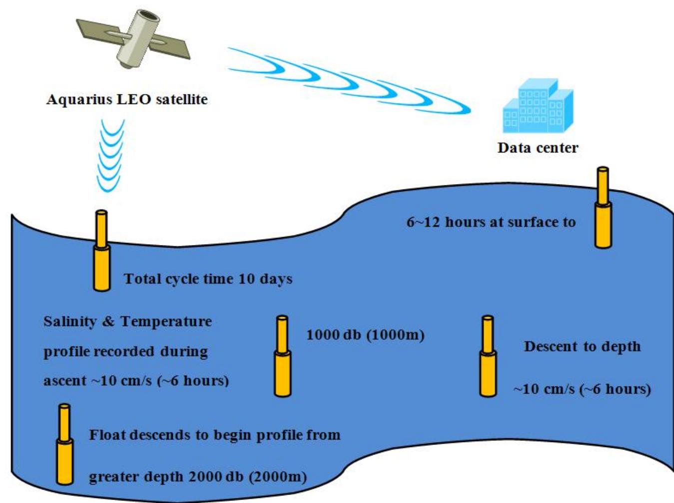

Fig. 2. The three subsystems architecture

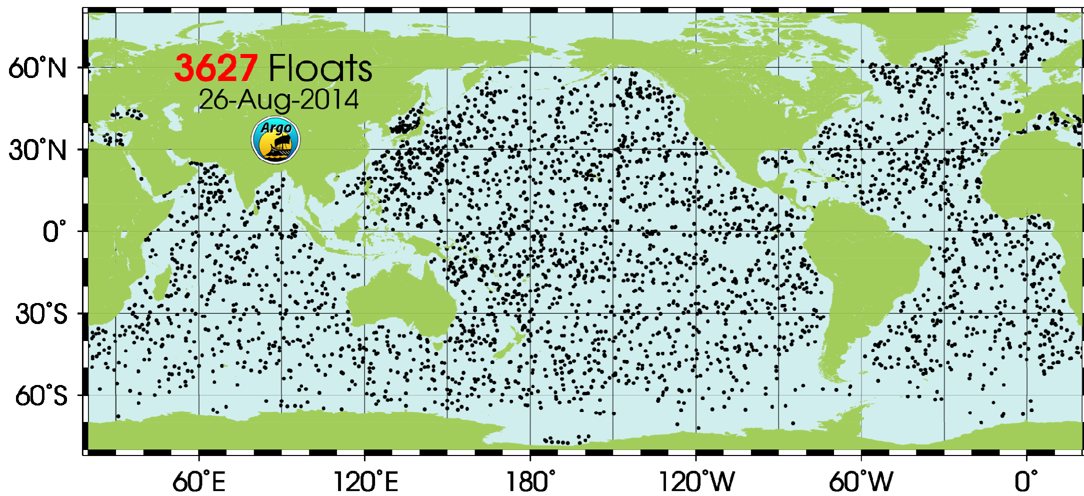

Fig. 3. Positions of the floats [8]

It stays in the surface for about 6 to 12 hours and sends the collected data to the passing Aquarius satellite in the sky [9][10]. In Fig. 3, while the Argo array is currently complete at 3627 floats, to be maintained at that level, national commitments need to provide about 800 floats per year. This paper focuses on how to achieve good performance on both throughput and fairness in the satellite communication for SSS plan. The overall performance of most TCP versions is not satisfactory because some TCP schemes emphasize on throughput while

others focus on fairness. The goal of the paper is to suggest a new scheme that can enhance and balance throughput for connections as segment losses occur or new connections join. Under these circumstances, the resources assigned to the connections will be redistributed rapidly to adapt to the new network environment. This redistribution may cause throughput to drop drastically or imbalance among the connections' bandwidth. Throughput is a measurement of how efficient the bandwidth is used. It is the quantity of correct data received by the destination from the source per time unit. Fairness measures how equally the bandwidth is divided among connections. In the ideal case, N connections share the same bandwidth in the network, each connection should get $1/N$ of network capacity. In order to improve the performance on both throughput and fairness, we modify congestion control and fast recovery mechanisms. We propose a new scheme, TCP Yam, to get better performance in both throughput and fairness. These two performance indicators are adopted to compare the performance of TCP Yam with other versions of TCP. We discuss how they behave under various traffic scenarios. It is observed that many of these TCPs throughput and fairness varies issues. The paper is organized as follows. Section 2 reviews some variations of TCP, such as Tahoe, TCP Reno, TCP Vegas, TCP for High-Speed Networks, and TCP Satellite for Networks. We will also cover the associated network architecture if some TCP versions assume unusual operation environments. The next Section describes our proposed TCP Yam scheme in detail. We give discussion on the design motivation and justification. We will also provide the pseudocode for our proposed scheme. Section 4 presents the performances evaluation of TCP Yam along with the comparisons with other versions of TCP. It shows that TCP Yam achieves good characteristics in both throughput and fairness. Finally, we conclude the paper in Section 5.

2 Related Works

In this Section, we will introduce TCP Tahoe, TCP Reno, TCP Vegas, TCP for High-Speed Networks, and TCP Satellite for Networks. We will give a rough comparison among these schemes before going to detail on each method. TCP Tahoe is the original version of TCP and considered as the prototype for other TCPs. TCP Reno adds a recovery scheme in TCP Tahoe. The recovery scheme enhances the system performance in the bandwidth recovery when the segment losses occur. After TCP Reno, TCP Vegas improves the congestion avoidance scheme. Yet, it suffers from the fairness problem. TCP for High-Speed Networks and TCP Satellite for Networks are two relatively new TCP proposals. TCP for High-Speed Networks suggests a new congestion avoidance scheme and aims to run in high-speed networks. In addition, TCP Satellite for Networks introduces two new supportive schemes, i.e., sudden start and rapid recovery, for satellite networks. It mitigates the ill effect of long propagation delay in the satellite network to improve throughput. But, when TCP Satellite for Networks is applied to a general network, it may cause congestion easily due to some redundant segments generated by the scheme. Now, we will take a closer look at these TCP versions.

2.1 TCP Tahoe

TCP Tahoe is one of the earliest TCP [11]. It comprises of slow start mechanism, congestion avoidance mechanism, duplicate ACK mechanism, time out mechanism, and fast retransmit mechanism [12]. They are discussed one by one in the following.

1. Slow start: Slow start is the first phase of data transmission. After the connection is successfully set by the three-way handshaking, the source sets *cwnd* to one. *cwnd*, denoting the congestion window size, is the number of segments that the source can transmit at the moment. Since it is initially set to 1, the source can transmit one segment to the destination. After transmitting one segment, the destination receives a segment and returns an ACK. When receiving the ACK, the source adds one to *cwnd* (*cwnd* = *cwnd* + 1). Now, the source is allowed to send out two segments since *cwnd* is equal to 2. In approximately one round trip time, two ACKs from the destination reach the source and lead to doubling of the *cwnd*. This behavior continues *cwnd* is doubled in every RTT. RTT, representing round trip time, is the time that the source transmits a segment to the destination and receives an ACK of the expected sequence number. The slow start scheme repeats until a segment loss is detected.

2. Congestion avoidance: This scheme controls *cwnd* to avoid congestion. *cwnd* behaves differently in the congestion avoidance phase than in the slow start phase. In congestion avoidance, *cwnd* is increased one per RTT while it is doubled in slow start. It means that the source adds 1/*cwnd* for each received ACK (*cwnd* = *cwnd* + 1 / *cwnd*). Furthermore, the *cwnd* segments that are allowed to be sent should be transmitted over a RTT to maintain this linear increment of *cwnd*. The congestion avoidance scheme always follows the slow start scheme. The transmission rate of the source increases exponentially in slow start. When it reaches a certain level, the congestion avoidance scheme will be activated and conservatively grows the transmission rate. Specifically, once *cwnd* exceeds the threshold of *ssthresh*, the congestion avoidance scheme is triggered. s*sthresh* is

initially set to infinity and later set to half of *cwnd* when a segment loss is detected. Hence, it represents the half of maximum transmission rate previously achieved by the source without segment loss.

3. Duplicate acknowledgements (dup-ACKs): The mechanism uses repeated ACKs to detect the segment losses. If the source receives three identical ACKs, the source understands that a segment may be lost. Then, the source goes to the fast retransmit phase.

4. Time out: Time out is another method to detect the segment loss. In data transmission, the source sends a segment to the destination and bootstraps the associated timer. When the timer of a segment counts down to zero before receiving the ACK, the source detects a possible segment loss. The initial value of the timer is acquired according to the past RTTs [11]. When there is a segment loss, the source enters into the fast retransmit phase. On the other hand, the timer is terminated when the source receives the ACK of the corresponding segment before its expiration.

5. Fast retransmit: The scheme is used to retransmit the missing segment, set *ssthresh* to half of *cwnd*, and reduce *cwnd* to avoid further segment loss. If the source transmits the segments continually when the segment loss occurs, the network may result further segment loss easily. The fast retransmit phase is carried out in one RTT after the segment loss is detected. Then, *cwnd* is reduced to 1 in the slow start phase.

TCP Tahoe begins with the slow start phase and adds *cwnd* exponentially. When the segment loss occurs, the source goes into the fast retransmit phase. After the retransmission, the source proceeds with the slow start phase. However, this time the source will perform an additional step of comparing *cwnd* with *ssthresh*. If *cwnd* is greater than *ssthresh*, the source switches into the congestion avoidance phase. Otherwise, the slow start phase continues. In the congestion avoidance phase, the source increases *cwnd* linearly until detecting the segment loss.

2.2 TCP Reno

TCP Reno is modified from TCP Tahoe. In TCP Reno, there are six mechanisms, *slow start*, *congestion avoidance*, *duplicate-ACK*, *time out*, *fast recovery*, and *fast retransmit*. In these schemes, *congestion avoidance* and *fast recovery* are different from TCP Tahoe [13]. We will outline the differences of TCP Reno and TCP Tahoe in the following.

In TCP Reno, while a segment is lost, the source sets *ssthresh* to half of *cwnd* (*ssthresh* = *cwnd* /2) and *cwnd* to *ssthresh*. Then, the value of *cwnd* is used in the next congestion avoidance phase. In other words, TCP Reno accelerates the ramp-up of the segment transfer rate by setting a larger *cwnd* value in the beginning of the congestion avoidance phase. In the congestion avoidance phase, *cwnd* starts from *ssthresh* and adds linearly. In TCP Reno, the congestion avoidance phase follows on the fast recovery phase and the fast retransmit phase.

We explain the increment and decrement of *cwnd* in all phases of TCP Reno. First, in the slow start phase, the value of *cwnd* increases exponentially. When losing a segment, the source enters into the fast retransmit phase and the fast recovery phase. Then, the source transmits the segments according to *cwnd* until the segment loss occurs in the congestion avoidance phase.

Now, we discuss the shortcoming in TCP Reno. TCP Reno increases *cwnd* continually except till the segment losses. It uses the segment losses as the indication of congestion. After the segment loss occurs, the source decreases the segment transfer rate. This aggressive reduction in rate can result in lower system utilization. If the source can reduce the transmission rate to avoid congestion and the subsequent rate volatility, the overall throughput may be improved. We introduce those methods such as TCP Vegas and TCP Yam in the later sections.

2.3 TCP Vegas

Lawrence S. Brakmo and Larry L. Peterson propose TCP Vegas in [14]. TCP Vegas modifies the *slow start* scheme, improves the *congestion avoidance* scheme, and reforms the *retransmission* scheme from TCP Reno. These enhanced schemes are overviewed in the following.

1. Slow start: The authors use a new way to expand the bandwidth of the connections. A threshold on *cwnd*, γ, is added to switch from the slow start phase to the congestion avoidance phase. When *cwnd* reaches γ, the network is believed to approach congestion. TCP Vegas slows the segment transfer rate. γ is defined by the source according to the network environments.

$$Expected = WindowSize/BaseRTT \quad (1)$$

$$Diff = Expected - Actual = \left(\frac{WindowSize}{BaseRTT} - \frac{BytesTransmitted}{MeasuredRTT}\right) \quad (2)$$

2. Congestion avoidance: In TCP Vegas, the authors use some parameters to measure congestion. They are shown in Equation 1 and 2 [14]. First, *Expected* is a parameter that stands for the expected sending rate and is

defined by the ratio of *WindowSize/BaseRTT*. *WindowSize* is specified in bytes. The authors use it to represent the number of bytes that the source will be allowed to transmit. In other words, *WindowSize* contains the total number bytes of the next *cwnd* segments. *WindowSize* is calculated by the current sequence number minus the last ACK number. *BaseRTT* is the minimum among all measured RTTs. Hence, *Expected* indicates the maximum sending rate in the following RTT estimated by the source. In Equation 2, *Diff* is the difference between the expected and actual sending rates. TCP Vegas calculates *Actual* as the ratio of *BytesTransmitted* to *MeasuredRTT*. *Actual* represents the sending rate in bytes/second measured from the latest RTT. *BytesTransmitted* is the size in bytes of the successfully transmitted segments which are sent and their associated ACKs are received by the source. It is calculated by the current sequence number minus the final sequence number of the last RTT. *Diff* can be considered as an indicator for the level of congestion. When *Diff* is positive or zero, the source uses two thresholds for *Diff*, α and β to control its sending rate. α is the lower bound of *Diff* while β is the upper bound. α and β are set by the source according to the network environments. If *Diff* is smaller than α, the source increases 1/*cwnd* to use the available bandwidth in the next RTT. Or, the source decreases 1/*cwnd* to prevent the segment loss in the next RTT when *Diff* is greater than β. If *Diff* is between α and β, the source remains unchanged in the value of *cwnd*. In the congestion avoidance phase, the appropriate size of *cwnd* is to keep between α and β. This modification leads to a stable segment transfer rate, avoids rate fluctuation, achieves higher throughput, and seems to prevent congestion more effectively [14]. However, the source resets *BaseRTT* to the latest *MeasuredRTT* when *Diff* is negative. In addition, *BaseRTT* is smaller than *MeasuredRTT* because it is the smallest RTT among all measured RTTs.

3. Retransmission: The retransmission mechanism of TCP Vegas uses the timestamp to achieve more accurate RTT estimation. When receiving a repeated ACK, the source calculates the RTT according to the timestamp and checks it against the timeout value of the ACK. If the RTT is greater than the timeout value, the source retransmits the segment without waiting for three duplicated ACKs.

TCP Vegas has the fairness problems. When a new connection joins into the network, the segment loss of the existing connection may be resulted from the sudden change of network situation. Then, the existing and new connections need to compete for bandwidth and their segment sending rate will change based on the computation of their *cwnd* values. Hence, some of the TCP versions drastically cut the *cwnds* of the existing connections to ensure fair competition. We will revisit this problem and give a former definition of fairness in Section 4. TCP Vegas depends on the accurate calculation of *BaseRTT*. In Equation 2, the value of *WindowSize* may be close to *BytesTransmitted*. It is because *cwnd* increases or decreases rather slowly as we explained in the TCP Vegas' congestion avoidance scheme. When *MeasuredRTT* is much greater than *BaseRTT*, we may conclude that the network approaches congestion. The bandwidth available to the source when the *BaseRTT* is measured may be exhausted by other sources when the current *MeasuredRTT* is estimated. Thus, *cwnd* has to be reduced as we explained above. On the contrary, when *MeasuredRTT* is too small, TCP Vegas increases *cwnd* slowly.

We think it is the strength of the TCP Vegas to control *cwnd* according to the level of congestion. However, the adjustment is gradually made and may result in the wasteful use of the network resources. Another important property of TCP Vegas is that when it coexists with other TCP versions like TCP Reno, its own throughput may degrade. TCP Vegas reduces its sending rate according to *Diff* when it enters the congestion avoidance phase. That is, TCP Vegas starts to curb its segment transfer rate while other TCP versions still try to get the most out of the network.

2.4 TCP for High-Speed Networks (FAST TCP)

In TCP Tahoe and TCP Reno, the value of *cwnd* can go through rapid and drastic changes when a segment loss is detected. FAST TCP was designed to slow down the segment transfer before suffering from the segment loss. Cheng Jin, David X. Wei and Steven H. Low propose FAST TCP in [15]. FAST TCP has a new congestion avoidance scheme for the high-speed network. It uses queueing delay instead of the segment losses as congestion signal while TCP Tahoe and TCP Reno employ the segment losses as the signal. FAST TCP separates the congestion avoidance mechanism of TCP Reno into four components, *data control* mechanism, *window control* mechanism, *burstiness control* mechanism, and *estimation* mechanism. The data control mechanism selects the next segment to transmit from the new segments or the segments that are considered to be lost. The window control decides the number of segments that can be bursted out. The burstiness control mechanism determines when to transmit these segments. These decisions are based on information which is provided by the estimation mechanism. The estimation mechanism collects information of network environments such as RTTs and queueing delay.

$$cwnd \leftarrow \min\{2cwnd, Weight\} \tag{3}$$

$$Weight = (1-w)cwnd + w\left(\frac{BaseRTT}{MeasuredRTT}cwnd + \kappa(cwnd, qdelay)\right) \tag{4}$$

FAST TCP solves two problems in TCP Reno. One of the problems is that the source relies only on one indication, i.e., the segment losses, to detect congestion. On the other hand, FAST TCP uses the queueing delay to estimate the level of congestion. The queueing delay, *qdelay*, is the time that the segments wait in a queue until they can be transmitted by the source. Equation 3 and Equation 4 represent the computation of new *cwnd* value that depends on *BaseRTT*, *MeasuredRTT*, and the queueing delay [15]. In Equation 3, the smaller value between the double *cwnd* and the weighted value is selected as the new *cwnd* value. Twice of the original *cwnd* value imposes an upper bound for the new *cwnd* value. In other words, FAST TCP prevents *cwnd* from increasing exponentially. In Equation 4, the computation of *Weight* can be divided into two parts. They are based on the current *cwnd* value and the traffic and congestion levels. A predefined parameter *w*, ranging between 0 and 1, is used to determine the impact of these two parts toward the final result of *Weight* calculation. If *w* is equal to 1, then the *Weight* will ignore the current *cwnd* value and decide on *Weight* entirely based on the traffic level of the source and congestion level of the network. In contrast, if *w* is close to 0, the *Weight* becomes insensitive to the collected information regarding the traffic and network conditions and heavily depends on the current *cwnd* value.

In the second part of *Weight* calculation, the authors use the ratio of RTTs and the function $\kappa(cwnd,qdelay)$ to assess the network condition. *qdelay* is a positive value which is calculated as that the average RTT decreases the minimum RTT. The network condition can be considered as the congestion and non-congestion. In the network congestion, the *MeasuredRTT* is larger than *BaseRTT* and the *qdelay* is nonzero. If $qdelay \neq 0$, $\kappa(cwnd,qdelay)$ is set to *n*, the number of segment in the queue at the source [15]. Otherwise, it is set to $n*cwnd$. Then, the calculation bases on the multiple *cwnd* plus *n*. The multiple *cwnd* may be a small value because *BaseRTT/MeasuredRTT* is close to 0.

In contrast, if the network is non-congestion, *qdelay* may be zero or nonzero. Thus, if $qdelay=0$, the calculation consists of the multiple *cwnd* and $n*cwnd$. Otherwise, the calculation is a value of the multiple *cwnd* plus *n*. The multiple *cwnd* may equal to the last *cwnd* due to *BaseRTT/MeasuredRTT* is close to 1. Another problem is that the variation of *cwnd* is frequently. The frequent fluctuation of *cwnd* may not acquire the better throughput. Therefore, the source changes *cwnd* according to the minimum RTT and the queueing delay. It provides more information to support data control mechanism, window control mechanism, and burstiness control mechanism. It seems that FAST TCP may run into the problem of using outdated data to determine the segment transfer rate. In other words, when the network environment goes through sudden changes, the source may not acquire the correct queueing delay immediately. It may choose an obsolete value of *cwnd*. When the segments arrive at a router, they have to be processed and transmitted. A router can only process one segment at a time. If the segments arrive faster than the router can process them, i.e., in a burst transmission, the router puts them into the queue until it can transmit them.

2.5 TCP for Satellite Networks (TCP Peach)

TCP Reno has poor throughput in the satellite network [16]. In such a network, we need to consider two characteristics uncommon in other network environments, i.e., the long propagation delay and the high link error rate. The long propagation delay results in longer RTT. The longer RTT decreases the growing speed of cwnd that is used to get the available bandwidth in the slow start phase. In slow start, cwnd grows exponentially per RTT. If RTT is long, then cwnd will not increase as fast as we desire. It is particularly true in the beginning of a connection. Also, the high link error rate can easily lead to segments discard at the destination. When there is no ACK to a segment, the segment loss is detected. The source must reduce the sending rate. Thus, the high link error rate brings the unnecessary fluctuation of sending rate and degradation of throughput. Ian F. Akyildiz, Giacomo Morabito, and Sergio Palazzo propose TCP-Peach in [16].

TCP Peach consists of sudden start, congestion avoidance with timeout and duplicated ACKs detection, and fast retransmit and rapid recovery as shown in Fig. 4. Some of these schemes are borrowed from other TCP versions. Two of them are actually first suggested in TCP Peach. The two new schemes are based on the novel concept of dummy segments. Dummy segments are the low priority segments that are generated by the source as a copy of the last transmitted data segment. And, it is used to increase *cwnd* in a short period of time. The source sets one or more of the six unused bits in the TCP header to distinguish the dummy segments from the real data segments. The type of service (TOS) field is set by the source in the IP header to indicate the type or quality of service that should be provided by the network. For example, in the three-bit precedence subfield of the TOS field, 001 can be set to indicate dummy segments [17]. Thus, the router can discard the dummy segments when a router receives a segment beyond its queue capacity.

However, the source sends the dummy segments to the destination in the sudden start phase and the rapid recovery phase regardless of the network being congested or not. If the network is not congested, the dummy segments are used to quickly increase the bandwidth of the source. Even if the network is congested, the routers can use TOS in the IP header to drop out the dummy segments. When the source receives the ACK of a dummy segment, the congestion window is increased by one. The behavior of sending a great deal of dummy segments

is to increase *cwnd* dramatically for the ACKs of dummy segments. Next, we describe the specifics of the sudden start scheme and the rapid recovery scheme. Sudden start is carried out only in the first RTT in the beginning of connection. At the initiation of sudden start, the source transmits the dummy segments as fast as possible in a RTT. The destination window size is the segments that the receiver can receive.

The source obtains the window size from the widow field of the ACK message. This window size imposes a hard upper bound for the number of segments that can be bursted by the source. In other words, if *cwnd* is greater than the destination window size, the source is allowed to transmit a number of segments as specified by the window size. Rapid recovery solves the serious throughput degradation problem due to the link errors. The rapid recovery scheme uses the parameter, allowed dummy segment number (*adsn*), to raise throughput quickly. *adsn* is the number of dummy segments that the source is allowed to transmit into the network. The source sets the initial value of *adsn* to the unreduced value of *cwnd* right before the segment loss occurs. adsn is different from *cwnd*. The difference between *adsn* and *cwnd* is the type of segments that the source transmits. It should be noted that *cwnd* tracks the maximum number of segments that may be bursted by the source.

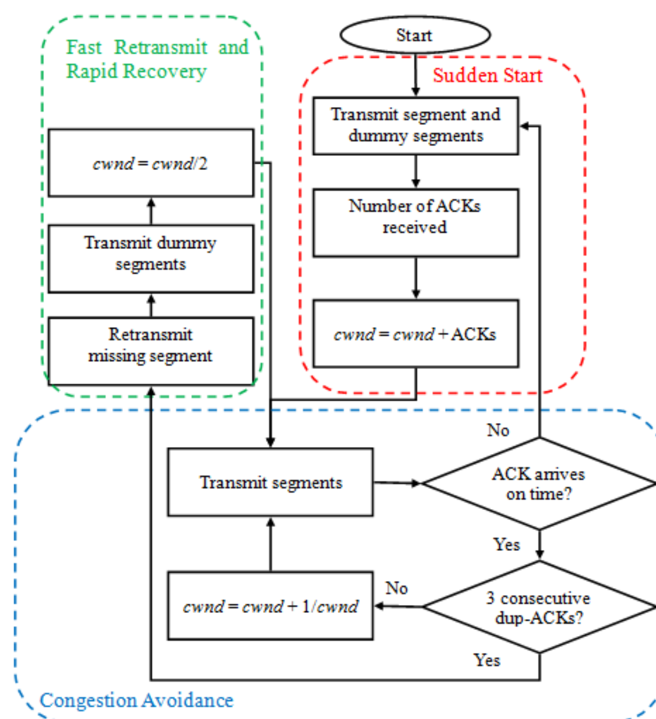

Fig. 4. The TCP Peach Scheme

The source obtains the window size from the widow field of the ACK message. This window size imposes a hard upper bound for the number of segments that can be bursted by the source. In other words, if *cwnd* is greater than the destination window size, the source is allowed to transmit a number of segments as specified by the window size. Rapid recovery solves the serious throughput degradation problem due to the link errors. The rapid recovery scheme uses the parameter, allowed dummy segment number (*adsn*), to raise throughput quickly. *adsn* is the number of dummy segments that the source is allowed to transmit into the network. The source sets the initial value of *adsn* to the unreduced value of *cwnd* right before the segment loss occurs. adsn is different from *cwnd*. The difference between *adsn* and *cwnd* is the type of segments that the source transmits. It should be noted that *cwnd* tracks the maximum number of segments that may be bursted by the source.

These segments are regular uses data and do not include the dummy segments. In short, the source uses different quota, *adsn*, to send the dummy segments. When the source transmits a dummy segment, it decreases one of *adsn* until the value of *adsn* is zero. However, the source receives ACKs of the dummy segment in the next congestion avoidance phase to increase *cwnd* rapidly. There are several drawbacks concerning TCP Peach when it is applied to the network with a better operating environment. The most glaring one is the extra traffic of the dummy segments. It may cause network congestion. In particular, if a segment loss is caused by a congestion instead of link errors, then the additional dummy segments sent in the rapid recovery phase will make the congestion even worse. In some TCP versions such as TCP Tahoe and TCP Reno, they decrease the bandwidth in the fast recovery phase.

Yet, TCP Peach measures the bandwidth to replace the decrease of bandwidth in the rapid recovery phase. Then, the source transmits the segments continually. Understandably, the rapid recovery scheme may usually be carried out due to the link errors in the satellite network. However, such assumption does not hold for other type

of networks. Thus, if the network is congested, the endless increment of *cwnd* may result in congestion frequently and reduce the utilization of network.

3 A High Throughput, Fairness, and Stability Transmission Control Protocol in the Satellite Communication for Sea Surface Salinity Plan

In the Section, we propose a novel scheme TCP Yam. TCP Yam improves the congestion avoidance phase and fast recovery phase from TCP Reno. In Section 2, we discuss several TCP improvements in either throughput or fairness. Or, it spends a cost with the addition of some supporting mechanisms. Hence, we may make a balance between throughput and fairness for no additional cost. TCP Yam combines TCP Reno and TCP Vegas. We use the average congestion window to be the signal when the network is getting congested, not the segment losses.

3.1 TCP Yam's Scheme

TCP Yam contains the *slow start* scheme, the *congestion avoidance* scheme, the *fast retransmit* scheme, the *fast recovery* scheme, the *duplicated ACKs* mechanism, and the *time out* mechanism. The scheme is shown in Fig. 5. The congestion avoidance scheme and the fast recovery scheme are different from TCP Reno. They are modified to balance throughput and fairness.

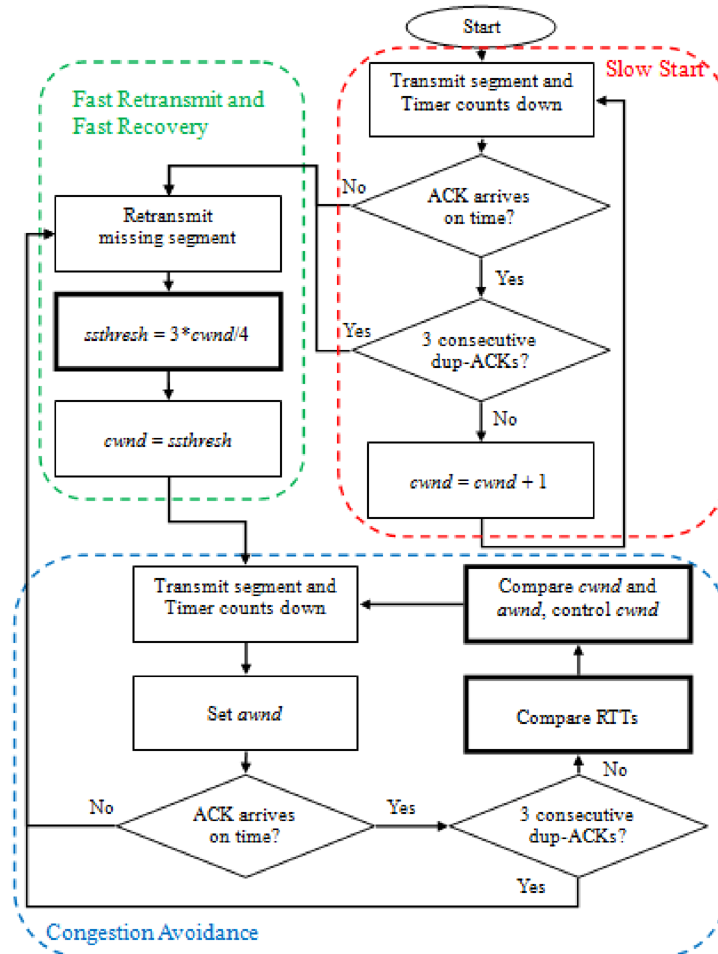

Fig. 5. The TCP Yam Scheme

Firstly, we introduce the problems that we want to resolve in the congestion avoidance scheme. One of the problems is that the source measures congestion by the segment losses such as in TCP Reno. The source may waste the utilization of bandwidth due to bandwidth fluctuation resulted from the segment loss. Another problem is that the source uses the delay-based parameters to be the signal of congestion, i.e., the queueing delay in FAST TCP. Yet, the source may adopt an improper *cwnd* which depends on the obsolete queueing delay or fast

changing network conditions. In some cases, the source may result in a terrible congestion when it calculates a large *cwnd* as the network situation quickly starts to deteriorate. Therefore, we adopt two parameters, RTTs and *awnd*, as the representation of congestion level. We believe these two problems can be addressed by the introduction of these two parameters.

Secondly, the source recovers the bandwidth from the segment loss in the fast recovery. It is reasonable to assume that when *cwnd* is above *ssthresh* the network tends to reach to a congestion state. Of course, *ssthresh* may be changed according to various network conditions. However, the congestion avoidance scheme and the fast recovery scheme are independent with their different objectives. They cooperate with each other to accomplish higher throughput. We can promptly adjust the *cwnd* value according to the ever changing network condition. Once *cwnd* becomes stable or accommodated to a network condition, its value should remain until the network condition changes again.

TCP Yam begins with the slow start phase. The source adds *cwnd* exponentially until losing the segments. In the fast recovery phase, the source sets *ssthresh* to χ. After our experiments, TCP Yam can reach a higher throughput when χ is greater than 1/2 of *cwnd*. When this phase ends, the value of *cwnd* is replaced by *ssthresh*. Then, the source prevents congestion according to *awnd* and the measured RTTs in the congestion avoidance phase. Next, we introduce the details in the fast recovery scheme and the congestion avoidance scheme.

3.2 Fast Recovery

Our fast recovery phase is used to moderate congestion. And, it allows the higher value of *cwnd* to be used in the next congestion avoidance phase. As mentioned earlier, the fast recovery phase is cooperated with the congestion avoidance phase to prevent congestion and raise the throughput. If we can prevent the congestion, we won't need to go through the severe bandwidth reduction in the fast recovery phase, especially for large *cwnd*. We raise the sending rate as high as we can in the congestion avoidance scheme. Now, the fast retransmit and fast recovery schemes are implemented together in the following.

1. Initiation: When detecting the implicit negative ACKs, which are signaled by three duplicated ACKs or timeout, the source sets *ssthresh* to χ, a constant set by the source. According to our experiments, the protocol has good throughput when χ is larger than half of *cwnd*. As shown in Section 4, χ is equal to 3/4**cwnd* seems an attractive choice. However, the initial value of *cwnd* is 1 in the slow start phase. When the network confronts heavily congestions, *ssthresh* may be reduced to a lower value after repeated detection of segment losses. After that, the lower *ssthresh* will be the initial value of *cwnd* in the congestion avoidance phase. The value of *cwnd* increases linearly over time so that the source may not use the bandwidth efficiently. Thus, we define that if *ssthresh* is not greater than 2, TCP Yam must restart from the slow start scheme. As explained previously, the value of *cwnd* increases exponentially in the slow start process.

In the retransmission, we transmit the missing segments according to the ACKs. When retransmitting the missing segments, the source sets *cwnd* to *ssthresh* plus the number of missing segments. We inflate *cwnd* to assure that all the missing segments will be sent in one RTT. If the sum is greater than the *cwnd* value prior to the detection of the segment loss, then the value of *cwnd* remains unchanged even the segment loss is detected. In the next step, we describe how the source reacts to the segment losses when it is in the retransmission phase.

2. Retransmission: After *ssthresh* is correctly set in the previous step, the source only transmits the missing segments excluding any new data segments in this step. In the retransmission, the source may receive the negative ACKs or explicit positive ACKs. If the source receives a negative ACK, the source has to restart the fast recovery phase until a positive ACK arrives. However, if there are three duplicated segment losses, the process compulsively restart from the slow start scheme. We use three duplicated segment losses to detect the excess segment losses that results in the heavily congestion. Then, we describe how the protocol reacts when a positive ACK is received by the source in the next step.

3. Fast Recovery: When a positive ACK arrives, the source sets the value of new *cwnd* to *ssthresh*. The positive ACK can indicate that all the transmitted segments are received by the destination and the network may be in a relatively uncongested state. In other words, this ACK tells the source that all the previously missed segments are sent successfully. When comparing to TCP Reno and Tahoe, TCP Yam is more aggressive in the fast recovery phase since it does not reduce *cwnd* as much when a segment loss is detected. If the congestion avoidance scheme can effective prevent the segment loss, we can use more bandwidth.

3.3 Congestion Avoidance

In the congestion avoidance scheme, we use two parameters, *awnd* and the measured RTTs. These parameters reflect the current situation of the network and can be used to control *cwnd* to prevent congestion. *awnd*, the size of average congestion window, is computed by the three latest values of *cwnd* and refreshed when the positive segment is received. Then, we show the movement of *cwnd* to prevent the segment loss accurately. If the

network approaches congestion gradually, such as the current RTT is greater than the last RTT, we decrease *cwnd* to avoid network congestion. In contrast, we increase *cwnd* carefully so that a new congestion is not produced easily. Therefore, we explain the network situation from the value of *awnd*, *cwnd*, and *ssthresh*. If the source receives a positive ACK, the value of *awnd* is refreshed. In contrast, the source updates the value of *ssthresh* and replaces the new *cwnd* by *ssthresh* in the fast recovery phase. Furthermore, *cwnd* is constantly adjusted according to the measured RTTs and the difference between itself and *awnd*. The congestion avoidance algorithm operates in the following.

1. In the congestion avoidance phase, the initial value of *cwnd* is set in the fast recovery phase. As mentioned above, when the source receives a positive ACK, the new value of *awnd* is computed according to three latest values of the past *cwnd*.

2. After *awnd* and *cwnd* are set in the previous step, the source transmits the segments according to *cwnd* and compares the value of RTT with the last RTT and *awnd* with *cwnd*. The result of this comparison will determine the adjustment made to *cwnd*. The basic idea is the following. When the source believes an imminent congestion, the value of *cwnd* will start to be reduced. Otherwise, the value of *cwnd* will try to be linearly grown. This is an important feature of TCP Yam since no other congestion avoidance scheme allows reduction in the *cwnd*.

(1)、If current RTT ≧ last RTT,
- i. If $cwnd \geqq awnd$, $cwnd = cwnd - 1 / cwnd$.
- ii. If $cwnd < awnd$, $cwnd = cwnd - 1$.

(2)、If current RTT < last RTT,
- i. If $cwnd \geqq awnd$, $cwnd = cwnd + 1 / cwnd$.
- ii. If $cwnd < awnd$, $cwnd = cwnd + 1$.

The details of the above procedure are discussed in the following. First, we consider that the variation of RTTs can represent the current network situation. Therefore, we compare the current RTT with the last RTT. *awnd* is summaries the recent average use of bandwidth by the source. When the value of the current RTT is greater than the last RTT, it is considered that the network is gradually congested. Thus, we reduce *cwnd* to avoid congestion. The decrement of *cwnd* is 1 or 1/*cwnd* which bases on the comparison between *cwnd* and *awnd*. If the current RTT is less than the last RTT, the source increases *cwnd* to utilize more bandwidth. And, when *cwnd* is smaller than *awnd*, the source increases *cwnd* by one. The network has more bandwidth to be utilized than the past network condition according to *awnd*. Or, the value of *cwnd* is added by 1/*cwnd*.

In conclusion, TCP Yam exploits the *awnd* to directly reflect the current trend of *cwnd*. And, it cooperates with the measured RTTs. In one segment loss case, the congestion window reduces for one time. However, in a multiple segment losses situation, the value of *cwnd* reduces over two times. It changes the degree of congestion window is heavier than one segment losses. Hence, we may prevent the segment losses as possible as we can.

4 Performance Evaluation

This Section reports the simulations of TCP Yam and a number of other TCP protocols in different network environments. We use Network Simulation, version 2 (NS2), to be the simulation tool [18] . NS2 is developed by University California Berkeley. We use NS2 with the queue management mechanism that we describe in the next section to simulate the different network environments.

4.1 Simulation Environment and Parameters

We introduce two types of queue management mechanism. First, we apply Drop Tail on the queue of all routers [18]. Drop Tail is a queue management algorithm that is used by the routers to decide when to drop the segments. With Drop Tail, when the queue is filled to the maximum capacity, the newly arriving segments are dropped until the queue has enough capacity to receive.

Another queue management mechanism is random early detection (RED) which is also known as random early drop. This mechanism is used to detect congestion before the queue overflows by dropping the segment with a probability. RED is fair for all the segments in the queue.

After introducing the mechanism that we use in our simulation, we explain the computation of fairness. The fairness problem happens in the networks that have over two connections. The fairness among the connection flows is affected when a new connection arrives. This affect may reduce throughput and change fairness among the connections. According to the other researches, we use Jain's fairness index to calculate the fairness [15]. The parameter b_i is the throughput of each connection flow. In Equation 5, n is the number of connection flows in the network. The range of fairness index is from $1/n$ to 1. However, the best value of fairness index is 1

when each connection flow has the same throughput. Otherwise, when the difference among their throughputs is dramatic, the fairness index is growing worse to $1/n$.

$$\textit{Jain's fairness index:} \frac{\left(\sum_{i=1}^{n} b_i\right)^2}{n \times \left(\sum_{i=1}^{n} b_i^{\,2}\right)} \tag{5}$$

4.2 Static Traffic Scenarios

In the section, we show the performance when TCP Yam and other TCP versions coexist in a network. We set the same executing time for every connection. Specifically, we examine the throughput and fairness among the connections with the same TCP version and difference.

First, we observe the case where there is only one TCP Yam connection. The TCP Yam connection and other relevant network parameters are illustrated in Fig. 6. The bandwidth capacity is 10 Mbps between the two end users. The propagation delay is 1ms for the two links connecting the routers to the two end users. The bottleneck link located between the two routers has the bandwidth capacity of 1 Mbps and the propagation delay of 4ms. The queue sizes are the same for both routers and they contain 18 segments. The queue size of the destination is set to 20 segments. When the destination queue is empty, 20 segments are the number of segments allowed to be received by the destination.

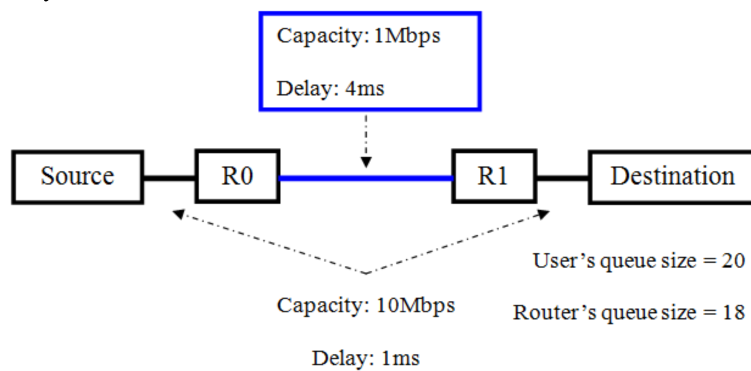

Fig. 6. Single TCP Connection

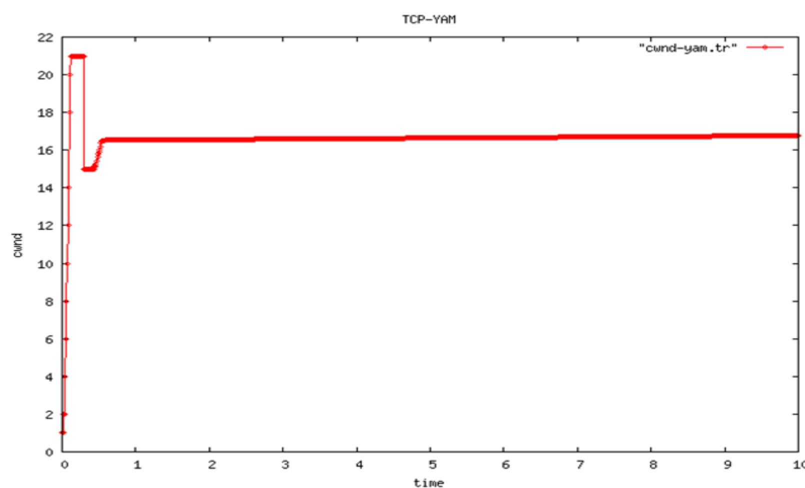

Fig. 7. The Congestion Window of Single TCP YAM Connection

Fig. 7 and Fig. 8 show the charts of *cwnd* and the average throughput. They demonstrate that TCP Yam exhibits stable characteristic in a single connection case and owns a high throughput. In Fig. 7, the x-ray is the time of seconds and y-ray is *cwnd*. Next, Fig. 8, the x-ray is the time of seconds and y-ray is the average throughput of kilo-bits. We record the process per 0.01 second. In Fig. 7, between 0 and 1 there has a falloff

which presents the segment losses. This falloff affects the throughput as Fig. 8. After that, adding the increment dramatically because of the RTT is small.

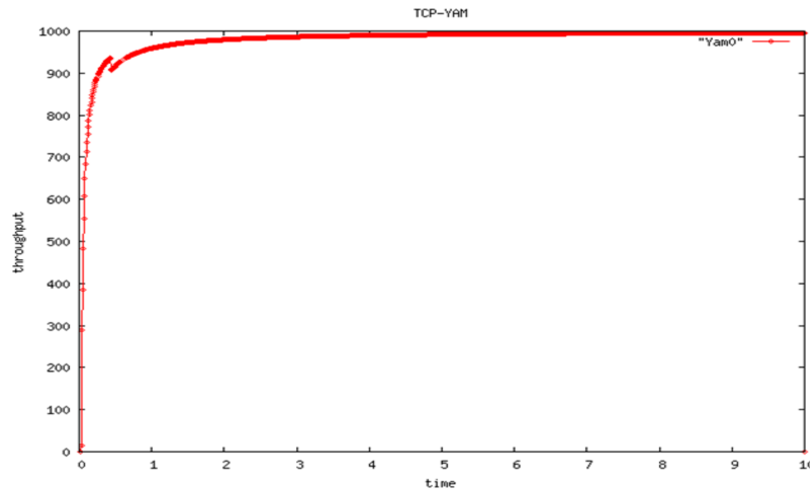

Fig. 8. The Throughput of Single TCP YAM Connection

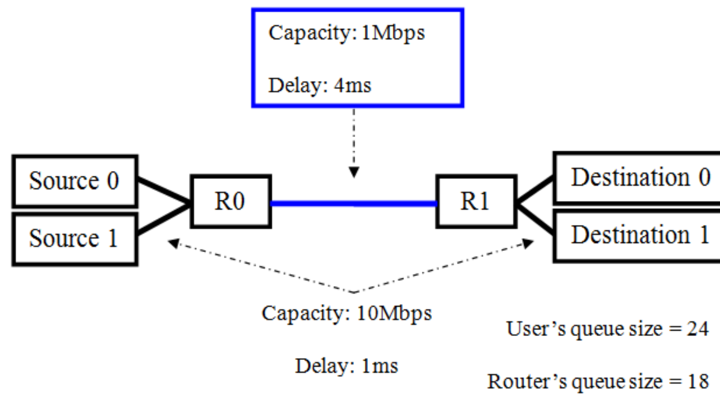

Fig. 9. Two TCP Connections

Secondly, we discuss how two TCP connections interfere with each other in Fig. 9, Fig. 10, Fig. 11, Fig. 12, and Fig. 13. First, we discuss one TCP Yam connection. The network topology is illustrated in Fig. 9. The bandwidth capacity is 10 Mbps between two end users. The propagation delay is 1ms for the two links connecting the routers to two end users. The bottleneck link located between two routers with the bandwidth capacity of 1 Mbps and the propagation delay of 4ms. The queue size of routers is 18 segments. The destination's queue size is set to 24 segments that is the number of segments allowed to receive by the destination.

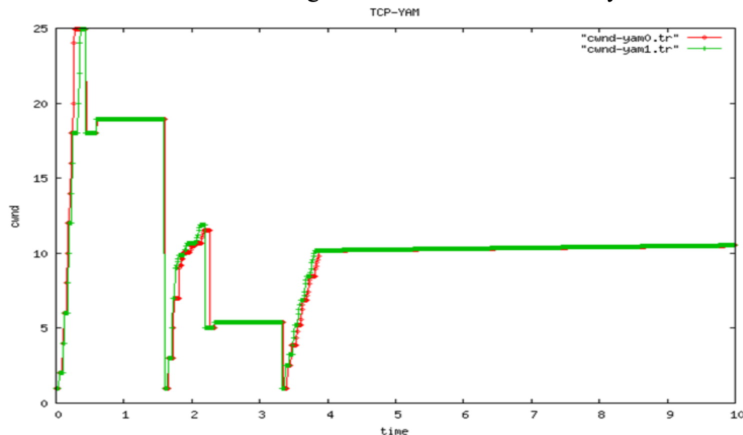

Fig. 10. The Congestion Window of Two TCP YAM Connections

We observe the change of congestion window to present the situation of network and throughput in our simulation. As depicted in Fig. 10 and Fig. 11, TCP Yam achieves fairness between two similar YAM connections.

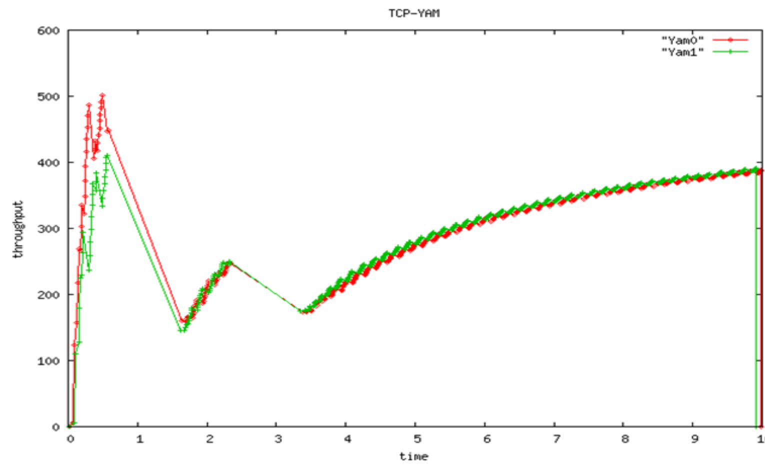

Fig. 11. The Congestion Window of Two TCP YAM Connections

Then, we measure throughput and fairness between TCP Yam and other TCP versions. We compare a TCP Yam connection with a TCP Reno connection. It is shown in Fig. 12 and Fig. 13 that Yam will not be fair against Reno. In fact, the TCP Yam connection will enjoy much higher throughput. This unfairness is caused by TCP Reno's strong focus in reducing *cwnd* dramatically after the segment losses. In the period of 0.3 to 0.7 seconds, there are segment losses. Then, TCP Yam restores the congestion window rapidly and gets a better throughput than TCP Reno. And then, TCP Yam smoothly increases *cwnd* and throughput.

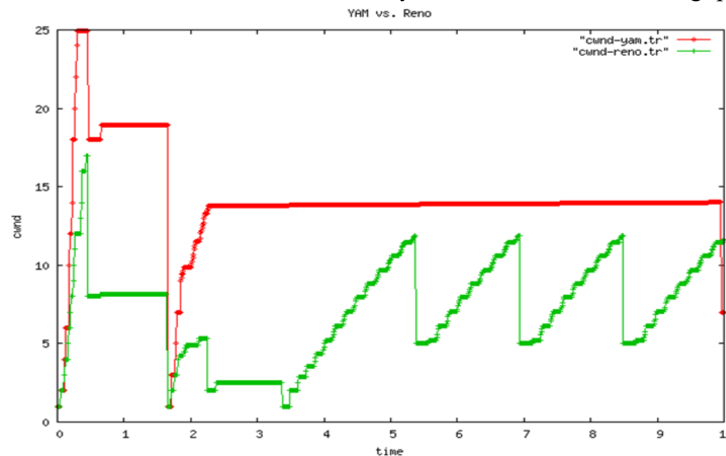

Fig. 12. Congestion Windows of TCP YAM and TCP Reno Connections

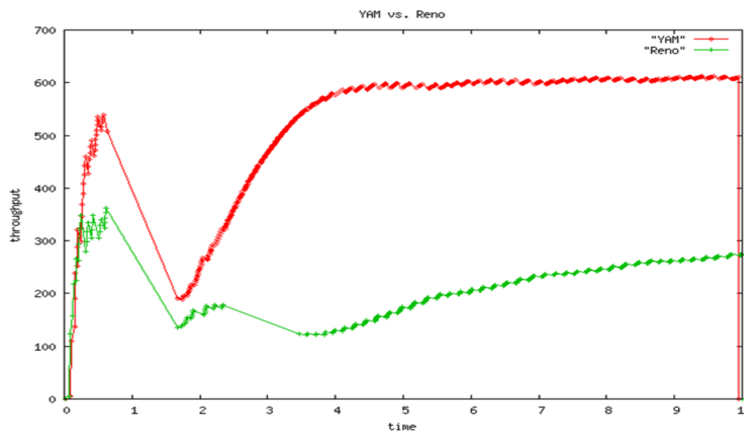

Fig. 13. Throughput of TCP YAM and TCP Reno Connections

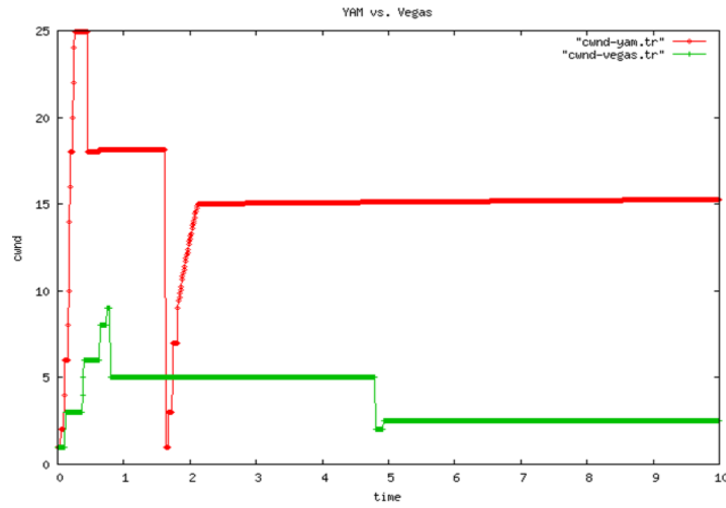

Fig. 14. Congestion Windows of TCP YAM and TCP Vegas Connections

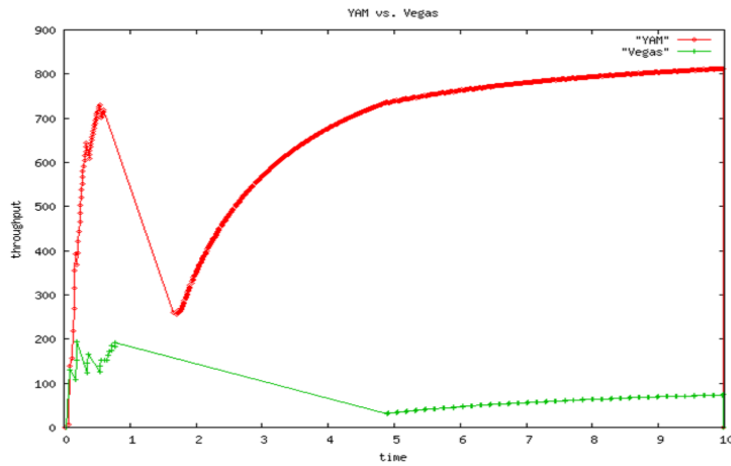

Fig. 15. Throughput of TCP YAM and TCP Vegas Connections

In Fig. 14 and Fig. 15, we compare *cwnd* and throughput between TCP Yam and TCP Vegas connections. TCP Yam and TCP Vegas change the bandwidth immediately by the congestion avoidance mechanism. It is shown that in Fig. 14 and Fig. 15 that Yam has a better fairness to Vegas. In fact, the Yam connection will enjoy much higher throughput. It is unfairness is caused by Vegas' segment avoidance. In the period of 0.3 to 0.7, there are segment losses. We can find that the TCP Vegas' recovery ability is bad. TCP Yam recovery the congestion window quickly and get a high throughput. This behavior confirms the TCP Yam's competitive ability is good.

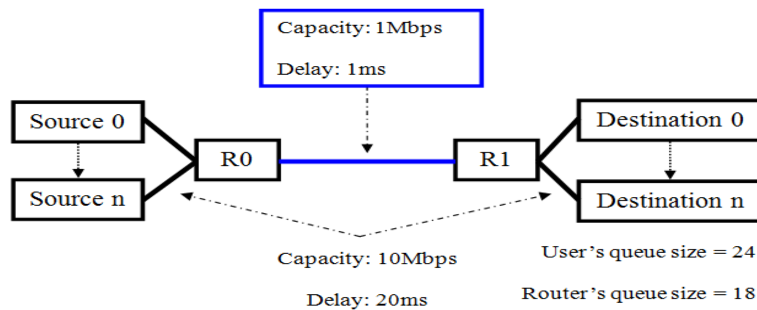

Fig. 16. Numbers of TCP YAM Connections

For the next scenario, the topology of network and implement results are Fig. 16, Fig. 17, and Fig. 18, we change the number of connection flows and prolong the implement time to observe the variation of throughput and fairness. The execution time is one hundred seconds. We record the throughput and the congestion window per 0.01 seconds. Fig. 16 is the topology of structure with some parameters. The bandwidth capacity is 10

Mbps between two end users. The propagation delay is 1ms for the two links connecting the routers to two end users. The bottleneck link located between two routers with the bandwidth capacity of 1 Mbps and the propagation delay of 4ms. The queue size of routers is 18 segments. The destination's queue size is set to 24 segments that is the number of segments allowed to receive by the destination.

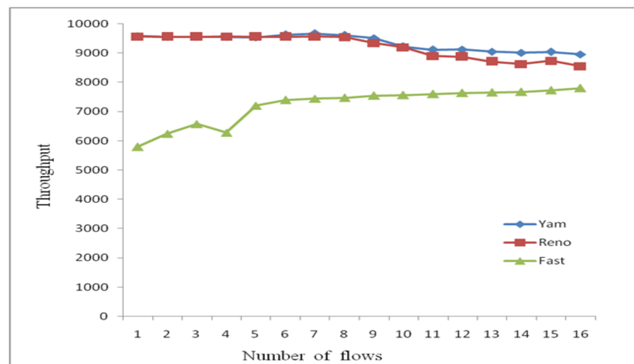

Fig. 17. The Throughput of Different Connections

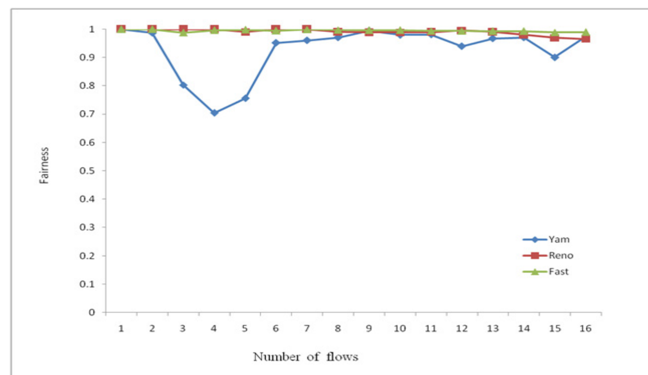

Fig. 18. The Fairness of Different Connections

From Fig. 17, TCP Yam has the similar throughput with TCP Reno. And, FAST TCP has the low throughput in the network environment with the low bandwidth capacity. In Fig. 18, TCP Yam has a lower fairness with number of flows from three to five than TCP Reno and FAST TCP. It may make an unstable situation. However, along with the increase of connection flows, TCP Yam's fairness backs to be similar to TCP Reno and Fast TCP.

4.3 Dynamic Traffic Scenarios

In this session, we show the dynamic network that is along with the different delay time, start time, and the end time among connections. We separately simulate the short implement time and the long implement time to observe the variation of throughput and fairness in these scenarios. And, we change the parameters of network to imitate the real networks. In the experiments, the fairness and throughput of networks may change dramatically for the mix of different propagation delay.

For the next scenario, the topology of network and implement period is Fig. 19 and Fig. 20. It is observed that the three connections behave as expected. There are three connection flows with the same propagation delays of 4 ms. They started and terminated at different times, as Fig. 20. We set the bottleneck capacity as 1 Mbps with the propagation delay of 20 ms. And, the other connections are 10 Mbps. The router's queue size is 18 segments. In Fig. 20, the first and second connections overlap in time and compete for bandwidth. When the first connection terminates, the throughput of the second connection takes off. It ends right before the third connection starts.

The Fig. 21 and Fig. 22 present the situation and throughput of network, this express TCP Yam may conform to the dynamic of network. This experiment shows the movement of congestion window and throughput in short time. Between 5 seconds and 11 seconds, the green flow joins into the network but the red flow cannot reduce immediately in the short time. It results the green flow cannot get enough throughput in early. Before 15 seconds, there is just green flow in the network. And, the green flow has not utilizes the throughput complete. At 15 seconds, the blue flow joins into the network and the green flow leaves the network.

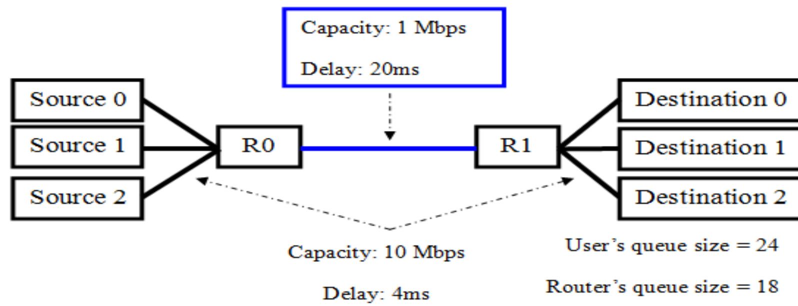

Fig. 19. Dynamic Traffic TCP Connections

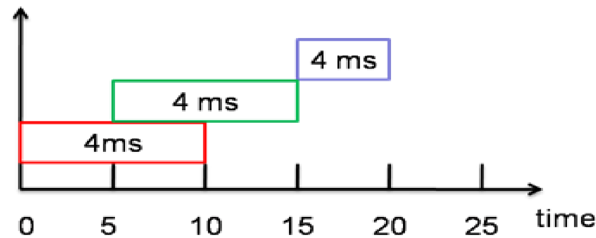

Fig. 20. Dynamic Traffic Pattern

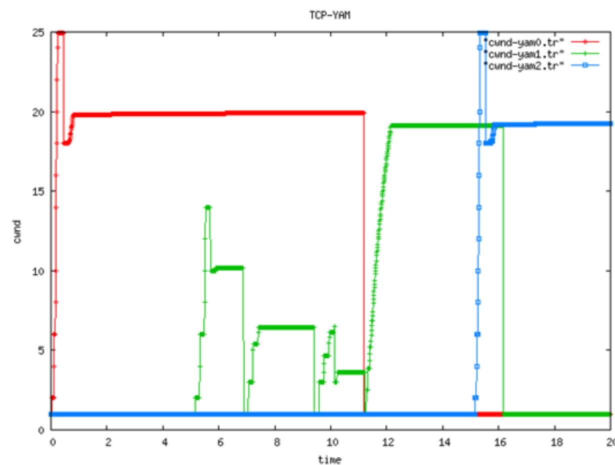

Fig. 21. The Congestion Window of Three TCP YAM Connections

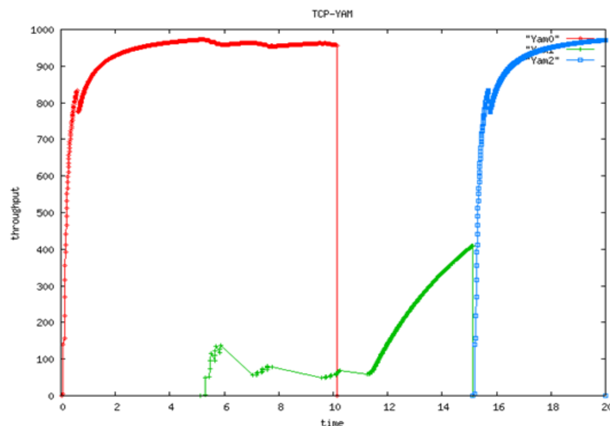

Fig. 22. The Throughput of Three TCP YAM Connections

TCP Yam may have a good throughput whatever the new connections connected or old connections disconnected. We may find that TCP YAM is more suitable on dynamic network than static network. The dynamic network is more satisfy to the real network.

Next, we prolong the implement time and increase the bandwidth capacity, propagation delay, and queue size. The topology of network and implement period is Fig. 23 and Fig. 24. We implement our scheme in high-speed network to observe that the three connections behave as expectation. There are three connection flows with propagation delays of 100, 150, and 200 ms. They started and terminated at different time, as Fig. 24. We set the bottleneck capacity as 400 Mbps. And, the other connections are 1 Gbps. The router's queue size is 2000 segments. The destination's receive window is 3000 segments.

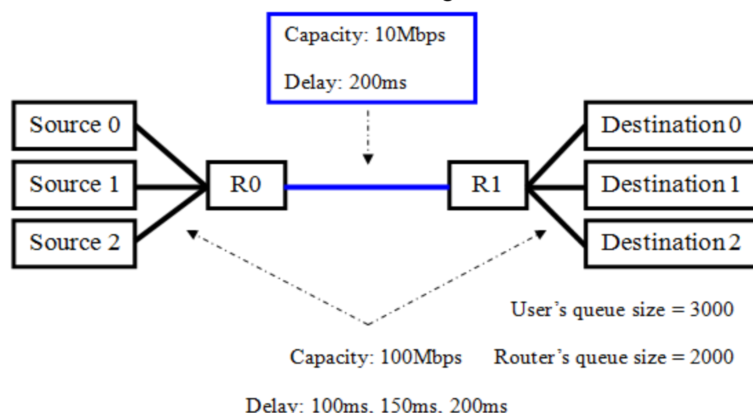

Fig. 23. Dynamic Traffic TCP Connections

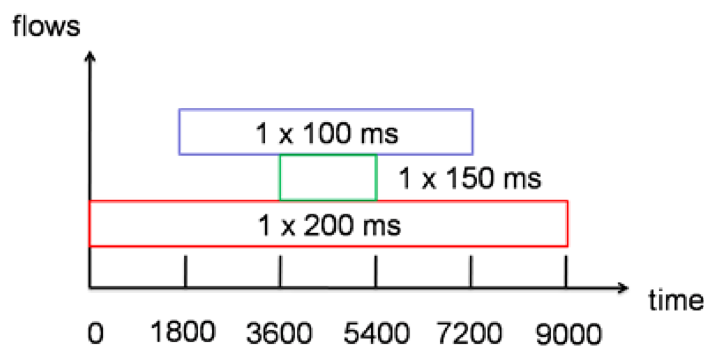

Fig. 24. Dynamic Traffic Pattern in High-Speed Networks

Table 1. The Fairness of Dynamic Traffic

Time	#Sources	YAM	FAST	Reno
1800-3600	2	0.95	0.96	0.68
3600-5400	3	0.971	0.972	0.90
5400-7200	2	0.95	0.96	0.71

From the Table 1, we get a summary result of fairness for three connections. The fairness of TCP Yam is good as FAST TCP in all time period. Fig. 25 presents the average throughput for three connection flows. This express TCP Yam may conform to the dynamic of high-speed network for long time. The first connection, red line, exists for all time. The second connection and third connection, green line and blue line, enter into the network at different time and compete bandwidth with the first connection. Between 3600 seconds and 5400 seconds, it makes heavy transmission for the competition of three connections. However, the green flow gets enough throughputs in competition with other connections. And, the red flow has and blue flow does not influence their average throughputs.

When one connection joins into the network, the order connection may influence the throughput and fairness to let the new connection get the enough throughputs. The average throughput shows the integration of network. Next, we present the throughput per fifty seconds and per twenty seconds. The results are similar to the average throughput but the wave by cycle.

In conclusion, we analyze the implementation cost and performance for some TCP versions. We know that TCP Yam has the lower cost such as TCP Tahoe, TCP Reno, and TCP Vegas. In TCP Peach, it needs the mechanism of telling the lower priority segment apart the data segment. And, it adds the load of processing the segments for the routers. However, FAST TCP separates the mechanism of congestion control into four compo-

nents for individual upgrade. This behavior may result in the more waiting time of selecting the information from the network.

In the performance aspect, TCP Yam's throughput is similar to FAST TCP and higher than TCP Tahoe, TCP Reno and TCP Vegas. However, TCP Yam has the current best fairness such as TCP Reno.

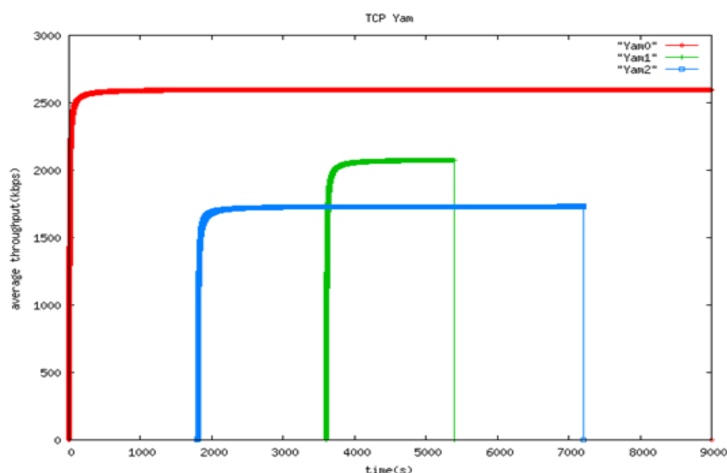

Fig. 25. The Throughput of Three TCP YAM Connections

5 Conclusions and Future Work

There are many existing researches regarding TCP improvements. In this paper, we propose TCP Yam, a fair and high throughput TCP version. We aim to raise the throughput and maintain the fairness among the connections. To achieve these two objectives, we rely on two parameters, *awnd* and the past RTTs, to estimate the level of congestion. Our approach reaches attractive performance with reasonable cost. In other words, it does not require the special schemes or mechanisms to support and other modification of the existing segment header. However, we simulate the results that show the good performance in throughput and fairness.

For this paper, TCP Yam has the both advantages of fairness and throughput. We may reduce *cwnd* to avoid a segment loss and recover to a higher bandwidth when the segment loss occurs. In other word, TCP Yam may effectively decrease the segment losses and availably control *cwnd* to avoid congestion. In our simulations, TCP Yam's throughput is higher than TCP Vegas and TCP Reno. And, it has a fair transmission such as TCP Reno. Therefore, we believe that TCP Yam has the ascendency of using on the networks.

Acknowledgement

This research was partially supported by Tatung University, Taiwan, R.O.C., under grant B103I01031.

References

[1] Argo website. Available: http://www.argo.ucsd.edu/How_Argo_floats.html 2007.

[2] T. W. Yue, Y. C. Wang, W. Yen, "Fast Sensor Identification Technology for Sea Surface Salinity Measurement," in *Proceedings of the 2008 IEEE International Conference on Systems, Man, and Cybernetics*, pp. 2700-2705, 2008.

[3] G. Lagerloef, F.R. Colomb, D. L. Vine, F. Wentz, S. Yueh, C. Ruf, J. Lilly, J. Gunn, Y. Chao, A. Decharon, G. Feldman, C. Swift, "The Aquarius/SAC-D Mission: Designed to Meet the Salinity Remote-Sensing Challenge," *Oceanography*, vol. 21, pp. 69-81, 2008.

[4] M. Muhammad, B. Matteo, D. C. Tomaso, “A Simulation Study of Network-Coding-Enhanced PEP for TCP Flows in GEO Satellite Networks,” in *Proceedings of the 2014 IEEE International Conference on Communications*, pp. 3588-3593, 2014.

[5] Y. Sun, Z. Ji, H. Wang, “TFRC-Satellite: A TFRC Variant with a Loss Differentiation Algorithm for Satellite Networks,” *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 49, pp. 716-725, 2013.

[6] J. Boutin and N. Martin, “ARGO Upper Salinity Measurements: Perspectives for L-Band Radiometers Calibration and Retrieved Sea Surface Salinity Validation,” *IEEE Geoscience and Remote Sensing Letters*, Vol. 3, pp. 202-206, 2006.

[7] Y. S. Chang, H. T. Cheng, H. J. Lai, “Metadata Miner Assisted Integrated Information Retrieval for Argo Ocean Data,” in *Proceedings of IEEE International Conference on System, Man, Cybernetics*, pp. 2930-2935, 2009.

[8] Argo website. Available: http://www.argo.ucsd.edu/index.html 2014.

[9] Aquarius project website. Available: http://aquarius.nasa.gov 2014.

[10] C. S. Tsai and G. F. Yang, “Research on an Assistant Ad Hoc Network to Aid the Measurement of Salinity-Temperature-Depth,” in *Proceedings of IEEE International Conference on System, Man, Cybernetics*, pp. 2689-2693, 2008.

[11] J. Postel, *Transmission Control Protocol – DARPA Internet Program Protocol Specification*, Internet RFC 793, 1981.

[12] M. Allman, V. Paxson, W. Stevens, *TCP Congestion Control*, Internet RFC 2581, 1999.

[13] T. R. Henderson, E. Sahouria, S. McCanne, R. H. Katz, “On Improving Fairness of TCP Congestion Avoidance,” in *Proceedings of 1998 Global Telecommunications Conference*: *The Bridge to Global Integration*, pp. 539-544, 1998.

[14] L. S. Brakmo and L. L. Peterson, “TCP Vegas: End to End Congestion Avoidance on a Global Internet,” *IEEE Journal on Selected Areas in Communication*, Vol. 13, pp. 1465-1480, 1995.

[15] R. K. Jain, D. W. Chiu, W. R. Hawe, *A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Systems*, DEC Research Report TR-301, 1984.

[16] I. F. Akyildiz, G. Morabito, S. Palazzo, “TCP-Peach: A New Congestion Control Scheme for Satellite IP Networks,” *IEEE/ACM Transactions Networking*, Vol. 9, pp. 307-321, 2001.

[17] J. Postel, “Internet Protocol,” *Internet RFC 791*, 1981.

[18] The Network Simulator – ns-2 website. Available: http://www.isi.edu/nsnam/ns/ 2008.