TCP-Real
Improving Real-time Capabilities of TCP over Heterogeneous Networks

C. Zhang and V. Tsaoussidis
College of Computer Science
Northeastern University
Boston, MA 02115
{czhang,vassilis}@ccs.neu.edu

ABSTRACT

We present a TCP-compatible and -friendly protocol which abolishes three major shortfalls of TCP for reliable multimedia applications over heterogeneous networks: (i) ineffective bandwidth utilization, (ii) unnecessary congestion-oriented responses to wireless link errors (e.g., fading channels) and operations (e.g. handoffs), and (iii) wasteful window adjustments over asymmetric, low-bandwidth reverse paths. We propose TCP-Real, a high-throughput transport protocol that minimizes transmission-rate gaps, thereby enabling better performance and reasonable playback timers. In TCP-Real, the receiver decides with better accuracy about the appropriate size of the congestion window. Slow Start and timeout adjustments are used whenever congestion avoidance fails; however, rate and timeout adjustments are cancelled whenever the receiving rate indicates sufficient availability of bandwidth. We detail the protocol design and we report significant improvement on the performance of the protocol with time-constrained traffic, wireless link errors and asymmetric paths.

1. INTRODUCTION

The standard Transmission Control Protocol (TCP) has some expedient properties that match the requirements of reliable best effort service over wired networks. However, these properties cannot render TCP the protocol of choice for real-time communications over heterogeneous networks. Congestion control dominates the behavior of the protocol, even when errors are caused by transmission deficiencies. In the presence of non-congestive conditions, TCP might under-utilize the available bandwidth. False congestion-oriented responses due to transmission errors or asymmetric paths [6], with rapid downward window adjustments, undermine the protocol's eligibility for time-constrained applications that rely on smooth playback timers.

The most well-known and widely-used versions of TCP are Tahoe and Reno[1]. The congestion-control algorithm introduced by Tahoe includes Slow Start, Congestion Avoidance, and Fast Retransmit. The congestion window effectively grows exponentially (slow start) until a threshold is reached. Beyond that point additive increase (congestion avoidance) takes over. When retransmission timeout event occurs, the congestion window is set to double the maximum segment size. In Fast Retransmit, a number of successive duplicate acknowledgements (dacks) trigger off a retransmission without waiting for the associated timeout event to occur. Then the slow start is applied. TCP Reno introduces Fast Recovery in conjunction with Fast Retransmit. Fast Recovery effectively set the congestion window to half its previous value, rather than performing Slow Start, after the retransmitted segment gets acknowledged. TCP NewReno[9] introduces the concept of Partial Acknowledgement, which is an indication of multiple segment drops in presence of dacks. In such case, Fast Recovery procedure goes on re-transmitting multiple dropped segments until the absence of Partial Acknowledgements.

Since TCP's approach to error detection is based on mechanisms that only confirm that a segment is missing (i.e., 3-DACKs, timeouts), the nature of the error is not detected and does not determine alternative recovery strategies. The protocol's behavior is dominated by congestion control, even when errors are caused by transient random errors, transmission burst errors, and handoffs. However, recovery from non-congestion errors using the standard congestion control mechanism results in unnecessary degraded performance. The congestion control mechanisms rapidly reduce the window and re-adapt slowly after the error conditions are over. Communication time is extended and transmission rate fluctuation cannot conform to some applications' time-constrained patterns of data processing. Similar anomalies have been observed not only in the context of wireless networks but also in wired. For example, congested or asymmetric reverse paths that might carry the receiver-generated acknowledgments downgrade the transmission rates and the bandwidth utilization of the forward path. The reason is that in standard TCP the recovery strategy is dominated by the Round-Trip-Time (RTT) measurements. Hence, the applicability of TCP for multimedia applications over heterogeneous networks is limited for two reasons: (i) throughput is degraded, and (ii) data transmission cannot always conform to the time constraints of some applications.

The above limitations can be outlined precisely by the protocol's behavior in conjunction with the token bucket algorithm. The algorithm can determine an upper bound for TCP's throughput, hence alleviating the concerns of fairness. It also presents a challenge for TCP to demonstrate its capability to exploit the *available* bandwidth. Given a token rate r and a depth B, TCP is faced with the opportunity to consume r tokens per time interval. The window backward adjustments circumscribe further the throughput; tokens might not be consumed within the time interval and new tokens will be wasted due to the limitation of the token depth. Clearly, the number of tokens consumed per RTT constitutes an appropriate metric for the evaluation of the protocol's performance. We present results using this metric in section 4.

A well-designed version of TCP, which is however focused on sender-based congestion avoidance, is TCP Vegas [3]. Vegas defines a BaseRTT to be the minimum of all measured RTTs, and ExpectedRate to be the ratio of congestion window and BaseRTT. The sender measures the ActualRate based on the sample RTTs. If the difference between the ExpectedRate and ActualRate is below a threshold α, the congestion window increases linearly during the next RTT; if the difference exceeds another threshold β, TCP Vegas decreases the congestion window linearly during the next RTT. According to [3], Vegas achieves better transmission rates than Reno and Tahoe. Currently, it does not contribute to error detection and recovery in wired/wireless networks. Its RTT-based window adjustments do not allow for improving the protocol's behavior over asymmetric paths.

The authors in [2] studied TCP interaction with IETF control-load services for reservation-based applications. Compliant packets are sent marked while non-compliant and best-effort packets are sent unmarked. Their proposal extends RED by setting for marked packets a much lower dropping probability than for unmarked. For reservation-based connections, the congestion window consists of two parts: a reserved part equal to the product of reserved rate and the estimated RTT, and a variable part that tries to estimate the residual capacity and share it with other active connections.

The Wave-and-Wait Protocol (WWP) [4, 5] is a TCP-incompatible lightweight transport protocol running on top of IP. WWP introduces the concept of "wave". A wave consists of a predetermined number of fixed-sized data segments sent side by side, where the number of segments comprising a wave is set according to a "wave level". The less the perceived congestion risk, the higher the wave level and hence, the more segments a wave contains. A wave is effectively the congestion window with two additional attributes: (i) its size is fixed during each RTT and corresponds to the data receiving rate (i.e., it is not based on the acknowledgements received by the sender), and (ii) its size is published to both peers. The receiver decides about the next wave level according to the current wave level and the actual transmission time of the wave. It notifies the sender about the next wave level expected by using negative-SACK. WWP is a high-throughput energy-saving transport protocol and owns a number of advantages over standard TCP on heterogeneous networks.

A major component of TCP-Real is the wave-based communication pattern. Unlike the sender-based indistinct window manipulation, the wave-based communication enables both the sender and the receiver to use a limpid communication pattern. We show that, in the context of wired/wireless heterogeneous computing, a wave-based communication in conjunction with appropriate modifications to the error control mechanism could cancel TCP's inflexible behavior. TCP-Real enables accurate rate adjustments initiated by the receiver, possibility to distinguish the nature of the error, and appropriate recovery strategies within the frame of fair behavior and friendly rates. We show that transmission gaps can be reduced and throughput can be increased, thereby enabling better performance and feasible playback timers. Furthermore, asymmetric-link-caused behavior can be avoided. Our modification requires no infrastructure changes and is designed to work with standard TCP and to collaborate well with reservation mechanisms.

2. TCP-Real
2.1 Protocol Strategy and Justification

The basic idea of TCP-Real is to incorporate into TCP the concept of "wave" from WWP without changing the semantics of TCP and without violating the established standards of Additive-Increase/Multiplicative-Decrease-based congestion control[1] during congestion. The reason to introduce the concept of wave is that in order for the receiver to effectively estimate network congestion based on the successive segments reaching it, it needs some knowledge of the sender's pattern of transmission of these segments. Since TCP sender sends packets side by side within the congestion window every RTT, from receiver's point of view, the sender sends packets in waves. In TCP-Real, the sender's congestion window size is controlled by the receiver, rather than the sender itself. The receiver measures the data-receiving rate and adjusts the wave level according to the change of data receiving rate, which reflects network conditions. The lower the perceived rate, the higher the wave level suggested by the receiver, and vice versa. The receiver notifies the sender about the new wave level, using an option attached to the corresponding ACK. When the sender receives the ACK, it extracts the option field and changes its congestion window accordingly.

TCP-Real is a transport level solution that requires no modification at the routers. The contribution of TCP-Real is based on the accuracy of congestion level estimation and the subsequent appropriate recovery. Since the up-to-date congestion condition is monitored by measuring the data-receiving rate, TCP-Real receiver can not only know whether there is congestion, but also estimate more precisely the level of congestion. Taking advantages of this property, TCP-Real is designed to avoid congestion[2]. With Slow Start and Congestion Avoidance mechanisms used by TCP Reno/Tahoe, the sender continuously increases the sending window until the packet loss caused by congestion occurs, where it rapidly reduce the sending window. With TCP-Real, the sender can adjust the sending window before the packet loss occurs, thereby the fluctuation of the transmission rate is smaller.

---

[1] A recent paper [7] discusses the inefficiencies of AIMD, although the issue is still a subject of discussion.
[2] Source-based congestion avoidance was first introduced in TCP-Vegas.

The above modifications also constitute the foundation for an efficient recovery strategy over heterogeneous networks, which increases the throughput and reduce the transmission gap of real time applications. When transient random errors occur on wireless links, the data-receiving rate is unaffected. That is, an error might occur while the congestion level could not justify a detected packet drop. The sender then could avoid window adjustments backwards and transmit conservatively or aggressively, depending on the level of congestion and the density of the error detected. Our design currently is relatively more conservative: upon a timeout TCP-Real always backs off as Tahoe and Reno, but adjusts rapidly upwards to the appropriate wave level when the timeout was caused by transient random errors.

TCP-Real's receiver-based rate control renders the sender capable of recovering quickly after an error condition is over. This is a nice mechanism against burst errors and handoffs of wireless links. After the burst error is over, the receiver can estimate the congestion level faster and the sender's congestion window will be set accordingly by the receiver. With Slow Start and Congestion Avoidance mechanism and in presence of relatively large delay-bandwidth product, it takes more round trips to reach the appropriate size of the sending window.

In order to avoid the wasteful window adjustments downward over asymmetric links, the sender needs additional equipment: to decouple the timeout mechanism and the RTT from the window size. That is, in standard TCP, when there is an acknowledgement loss, timeout is extended and congestion window is reduced. In TCP-Real, the timeout can be extended, but the window size could remain the same or even increase. The reasoning behind this strategic modification is that the sender needs to extend the timeout based on the RTT measurements, in order to accommodate the potential delays of the reverse path and avoid an early timeout. However, only the perceived congestion level of the forward path will determine the sender's congestion window size.

In practice, the wave-based communication of TCP introduces a novelty: both the sender and the receiver are aware of the current window size (wave level), since the congestion window is now included in the header. As it is exemplified by flow control that requires the Advertised Window to be included in the header, the receiver now communicates with the sender including also the Congestion Window in the header, by means of the wave level. However, although the Advertised Window indicates the number of bytes permitted for transmission, the Congestion Window indicates the number of segments. Note that both flow and congestion measurements are taken at the receiver; the sender uses a Sending Window, which is the minimum of the two distinct windows, minus the data that is already in transit. Hadn't we been worried about TCP's semantics, we could have combined the two windows into a single Sending Window manipulated at the receiver and replacing the Advertised Window.

2.2 Protocol Implementation

TCP-Real extends TCP-Reno. In TCP-Real, the receiver computes the data transmission rate by collecting the current wave-length of data. The receiver also records t_f and t_l, the arriving time of the first and last segment in the wave, respectively. The data-receiving rate can therefore be computed as the ratio of the wave size to wave receiving time. The wave receiving time is the difference between t_f and t_l, and could be much smaller than the RTT. The wave size used in calculations is actually the "expected" wave size, not the "received" wave size, since loss of corrupted segments in a wave needs to be counted. Packet loss due to random transient errors should not affect the computed rate, which is used to measure the congestion level; packet loss due to congestion could be detected by the change in the receiving rate anyway.

The receiver determines the next wave level according to the change of the data-receiving rate over time. Whenever a new receiving rate is computed, it is compared against the previous one. If the rate increases (decreases), which means the network gets less (more) congested, the next wave level should be adjusted to higher (lower) levels. The two weights, preserve_weight and adjust_weight indicate the inclination to adjust conservatively or aggressively, respectively.

```
if ( previous_rate > current_rate){
        if (current_wave_level > wave_level_threshold)
                next_wave_level = (0.618 + 0.382*(current_rate /
                                previous_rate)) * current_wave_level ;
        else
                next_wave_level = (0.382 + 0.618*(current_rate /
                                previous_rate)) * current_wave_level ;
}
else{
        if (current_wave_level > wave_level_threshold)
                next_wave_level = (0.382 + 0.618*( current_rate /
                                previous_rate)) * current_wave_level ;
        else
                next_wave_level = (0.5 + 0.5*(current_rate /
                                previous_rate)) * current_wave_level ;
}
if (next_wave_level>max_wave_level)
        next_wave_level>max_wave_level;
```

Figure 1. Wave level adjustment algorithm

```
rate_ratio=current_rate/previous_rate;
next_wave_level = current_wave_level*
 (preserve_weight+adjust_weight*rate_ratio);
where  preserve_weight + adjust_rate = 1.
```

According to the paradigm of Slow Start and Congestion Avoidance, the sender's congestion window should expand aggressively (conservatively) when the window size is relatively small (large). We further propose in TCP-Real that the congestion window decreases aggressively (conservatively) when the window size is large (small). Notice that here the congestion window decreases before a packet loss occurs. Hence, the weights in the above formula need to be set dynamically depending on the relative size of current wave level, with a wave level threshold serving as judging reference. Whenever a receiver wave-decision-timeout event occurs (see below), the threshold is set to half the wave level prior to the timeout. Figure 1 shows the wave level adjustment algorithm, with weights set dynamically in different cases.

The receiver inserts the TCP-REAL option in the header every time a segment is acknowledged, to notify the sender about the next wave level. The TCP_REAL option is four bytes long. The second two-byte content of the option contains the next wave level suggested by the receiver. The sender extracts the TCP_REAL option from the ACK's header and sets its congestion window accordingly.   Since the identical TCP_REAL option is repeated in every ACK before the wave level changes, the probability not to deliver the wave-level information to the sender is pretty low. However, the sender should have some kind of a self-adjusting mechanism activated upon the lack of all wave-level signals sent by the receiver. A bit is associated with the sender's retransmission timer. Before the timer expires, if any DACK is received, the bit is set. Upon a timeout, the sender retransmits the packet but reduces the congestion window to two segments, like Slow Start, only if the bit is not set. This is taken as an indication that the sender did not receive any wave information during that time. Thus, reliable transmission and congestion control are effectively decoupled.

If the last segment in the wave is lost, the receiver will not be able to compute the receiving rate (t_l unavailable) and update the next wave level until the sender timeouts and retransmits. Note that the sender needs the information most when congestion develops. In such case, the receiver needs to update the wave level information before it completes collecting the current wave of data. Therefore, a wave-level-decision timer needs to be set at the time the receiver first receives any segment within a wave. The length of the timer is set to two times the expected wave receiving time (size of the current wave divided by the previous data-receiving rate). After the timer expires, the receiver computes the receiving rate when the next segment arrives, and updates the wave level information immediately. Then the newly updated wave is sent back with the ACK triggered.

3. TESTING

The protocol is implemented using the x-kernel protocol framework [8]. In our experiments we used 24 wave levels, ranging from 1 to 24. For wave level i, the corresponding congestion window size is `2*i*max_segment_size`. The minimum wave level corresponds to the size of 2 segments, while the maximum wave level corresponds to the maximum congestion window. The number is selected to be sufficient to fill in a relatively large Delay*Bandwidth product. The tests were carried out in a single session, with client and server running on two directly connected dedicated hosts. 5M bytes original data was sent from one end host to another in a single session. The data was selected to be sufficiently large to avoid significant deviations of measurements. We measured the protocol's goodput performance as:

```
Goodput = Original data / Connection time
```

In order to simulate the error conditions, we developed a new x-kernel "virtual protocol", which was configured between TCP and IP. The protocol's core mechanism consists of a two-state continuous Markov chain. Each state has a mean sojourn time m_i, a drop rate r_i ($i = 1,2$) whose values are set by user. Thus when it visits state i, the mechanism remains there for an exponentially-distributed amount of time with mean m_i, during which it randomly drops segments with probability r_i. In our experiments, one states were always configured with a zero drop rate. Thus, simulated error conditions during a given experiment alternated between "On" and "Off" phase during which drop actions were in effect and were suspended, respectively. The two states have the same sojourn time, 1 second, which is sufficiently large to permit a full window recovery under "clear" network conditions. Note that our model uses packet error rate (PER). The error rate presented in the figures and tables in the next section denotes the PER during the "ON" phase and not the PER of the entire connection.

We developed another x-kernel protocol VREAL, configured on top of TCP, to simulate a playback-based application with data rate 1Mbps. We present here results with a playback time interval of 40ms. That is, our application attempts to read and consume 5KB each time, 25 times a second. Because of the sending window fluctuation and transmission gaps of TCP, there are playback instances when the data is unavailable to the application. The percentage of the application successful attempts to read 5KB from the playback buffer is used to measure the protocol's real-time performance. Playback time interval of 100ms was also tested.

In order to reveal the dynamics of TCPs with real time traffic, we also simulate a token bucket at the sender side, with a depth of 20KB and a rate of 20 tokens per 30ms. Each token corresponds to a packet of 1 KB. The token consumption rate every RTT is measured to see the source's capability to fully utilize the transmission speed the token bucket allows. The token consumption rate per RTT was contrasted to the permitted rate in order to demonstrate the protocol's capability to exploit the available bandwidth.

4. RESULTS AND ANALYSIS

4.1 Goodput Performances

Our experiments first demonstrate that the Goodput is improved over heterogeneous networks with a short one-way propagation delay of 15 ms. With dropping rate of state ON varying from 0.0 to 0.5, the goodput comparison of TCP Real/Reno/Tahoe is shown in Table 1 and Figure 2. As we can see, TCP-Real outperforms both Tahoe and Reno. Even with random transient errors, Reno and Tahoe's congestion control mechanism

unnecessarily reduce the congestion window and adjust the timeout value. In TCP Real, in case the receiving rate does not justify a drop due to congestion, the congestion window is not adjusted[3]. This behavior of TCP-Real also results in improved goodput given the time constraints set for the experiment.
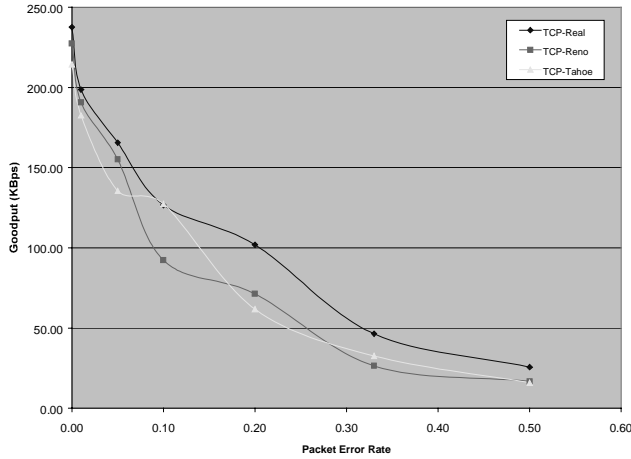

Figure 2 TCP Goodput with Link Error

It can be observed from Figure 3 and Table 2 that when the virtual protocol is only configured in the reverse direction in order to represent an asymmetric path, TCP-Real prevails again: It outperforms Tahoe and Reno, since the sender's congestion window is now controlled by the congestion level of the forward path estimated by the receiver.
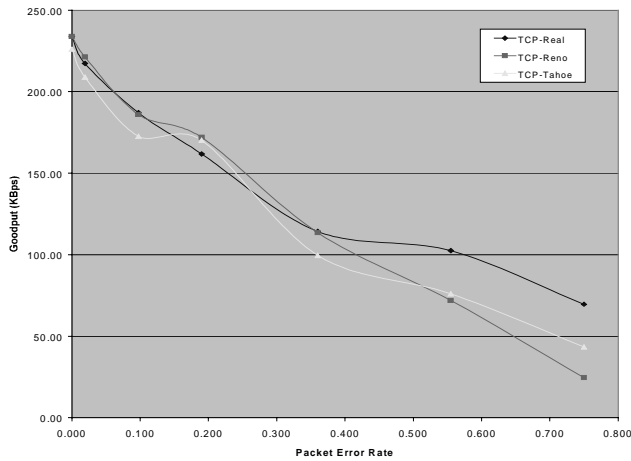

Figure 3 TCP Goodput with Asymmetric Links

Parameters were also set to reflect the impact of bursty or handoff conditions on protocol and application performance. The mean sojourn time was set to 5 seconds for state OFF and to 0.5 for state ON. The dropping rate for state ON was always 0.99, and 50M bytes original data was sent during each session. Data size is now selected larger due to the significant standard deviation observed with the previous setting. The results in Table 3 show

---

3 In the event of a timeout the window is, in fact, adjusted as in Tahoe and readjusts appropriately when the conditions do not call for congestion control.

that goodput for TCP Real, Reno and Tahoe is 594.67KBps, 310.98KBps and 308.75KBps, respectively. This confirms the allegation that TCP-Real has enhanced re-adapting capability after a bursty error condition is over, compared to Tahoe and Reno. In TCP-Real, the receiver can estimate the congestion-level/disconnection more precisely within one RTT. Tahoe and Reno require more RTTs for recovery even though the network conditions might not justify a conservative behavior.
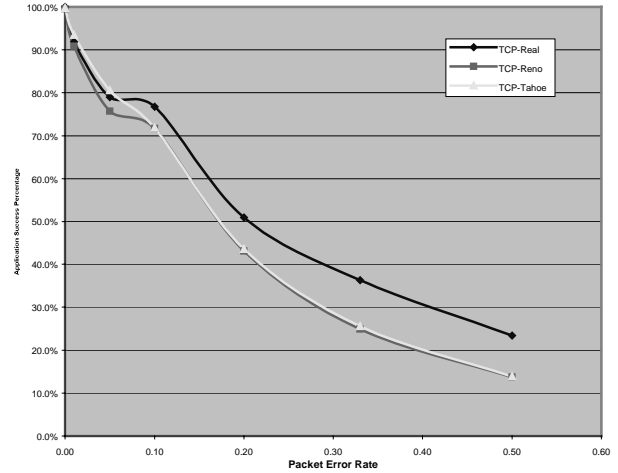
Figure 4. Application Success Percentage with Link Errors
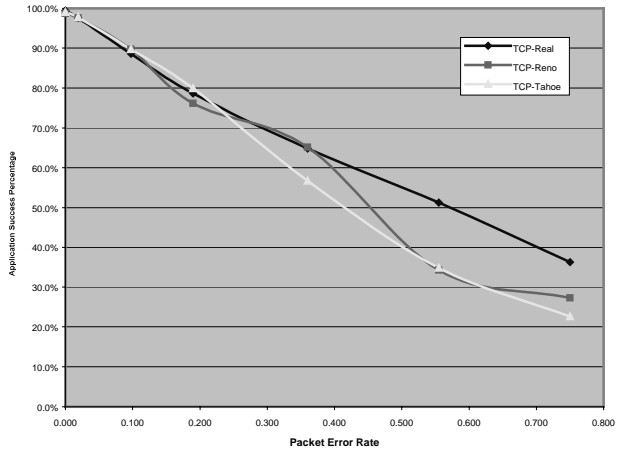
Figure 5. Application Success Percentage over Asymmetric Links

4.2 Application Success Percentage

The application success percentage comparison under the different scenarios described above is shown in Table 4, Table 5, Table 6 and Figures 4 and 5. The result for playback time interval of 100ms is about the same, as shown in Figure 11 and 12 in the appendix. As expected, not only the goodput of TCP is improved, but also the time-constrained application performs significantly better: TCP-Real reduces unnecessary transmission gaps that dominate the application's performance. TCP-Real's congestion avoidance mechanism enables the sender to adjust the sending window before a packet loss due to congestion occurs. Since the fluctuation of the transmission rate is smaller, the transmission
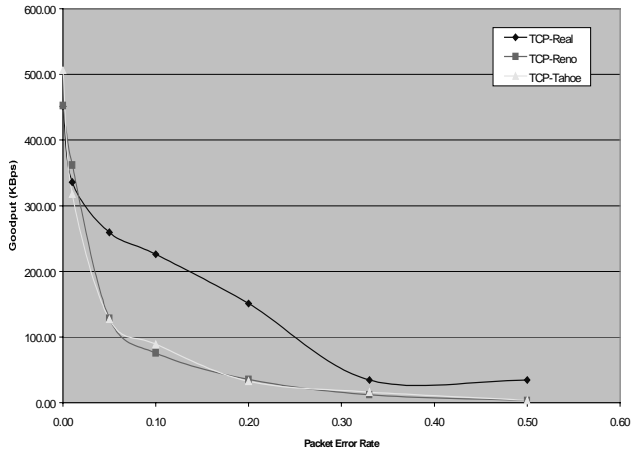
Figure 6. TCP Goodput with Link Error and High Propagation Delay
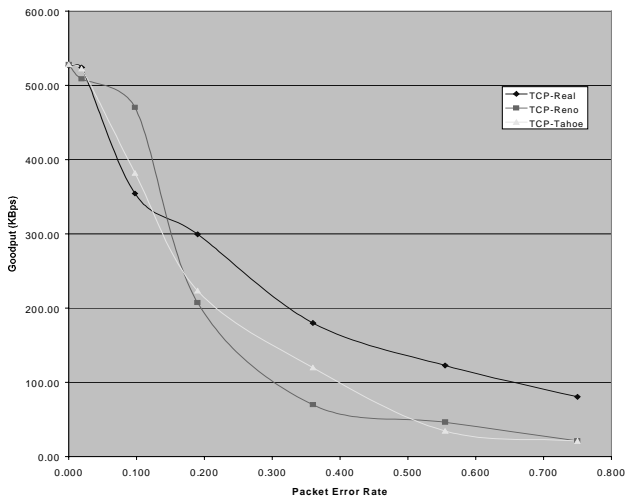
Figure 7. TCP Goodput with Asymmetric Link and High Propagation Delay
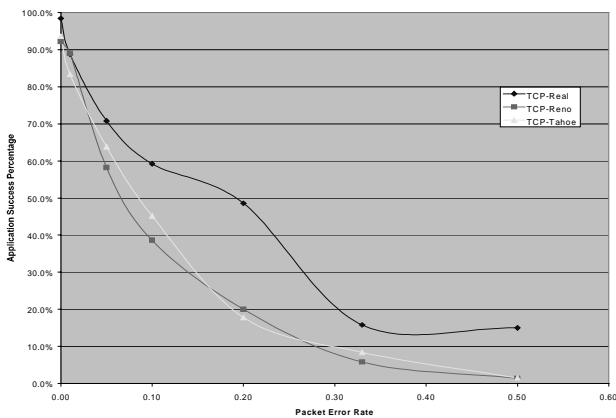
Figure 8. Application Success Percentage with Link Error and High Propagation Delay
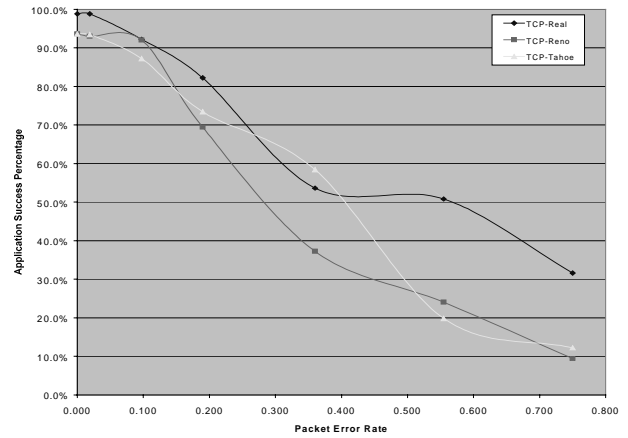
Figure 9. Application Success Percentage with Asymmetric Link and High Propagation Delay

4.3 Impact of Propagation Delay

The above experiments are repeated with a longer one-way propagation delay of 50 ms to simulate WANs or wireless link with high propagation delay. Such conditions highlight further Reno's and Tahoe's inappropriate recovery strategy. With longer propagation delay, the relative performance gain of TCP-Real is amplified, as shown in Table 7-12 and Figure 6-9.

4.4 Token Consumption

We also present the token bucket test as described in section 3, with the 15ms delay and 20% packet dropping rate. Recall that this is the error rate during the "ON" phase. Traces of token consumption every RTT are plotted in Figure 10. Whenever a link error occurs, Reno and Tahoe rapidly reduce the window size and hence the token consumption rates. Instead, with TCP-Real, the token consumption is relatively stable. Furthermore, the average of consumed tokens per RTT (11.12Kbytes) is significantly higher compared to Reno (5.82 Kbytes) and Tahoe (3.65 Kbytes).
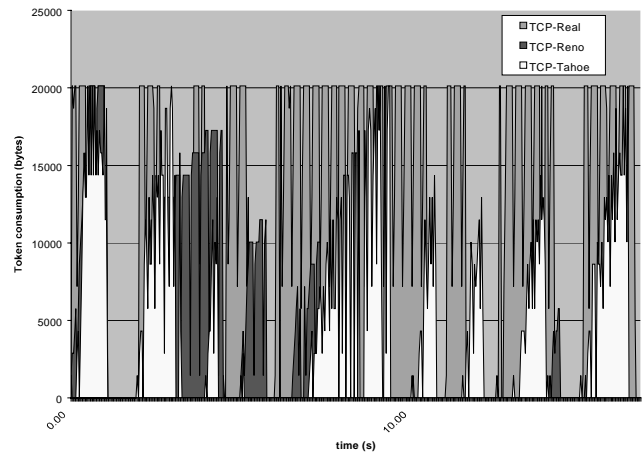
Figure 10. Token Consumption

gaps are reduced and the real time application experiences better performance.

5. CONCLUSIONS AND FUTURE WORK

We have presented a receiver-oriented TCP that remedies major shortfalls of standard TCP for multimedia services over heterogeneous networks. The protocol attempts to avoid congestion and RTT-based window adjustments. It applies an efficient error recovery whenever packet drops are not due to congestion and decouples the RTT and the timeout from the size of congestion window. Under congestion, the protocol exhibits a conservative behavior respecting the established standards of fairness and stability. TCP-Real's predetermined communication patterns are well suited to reservation mechanisms. Our future work will attempt to demonstrate this argument with RSVP. Furthermore, we plan to investigate the protocol's behavior in collaboration with other network components or devices that attempt to avoid/control congestion.

6. ACKNOWLEDGEMENTS

7. REFERENCES

1. M. Allman, V. Paxson, W. Stevens, "TCP Congestion Control", RFC2581, April 1999

2. W. Feng, D. Kandlur, S. Saha and K. Shin, "Understanding TCP Dynamics in an Integrated Service Internet", NOSSDAV '97, May 1997.

3. L.S. Brakmo and L.L. Peterson, "TCP Vegas: End to End Congestion Avoidance on a Global Internet", IEEE Journal on Selected Areas in Communications, 13(8):1465-1480, Oct 1995

4. V. Tsaoussidis, H. Badr, R. Verma, "Wave and Wait Protocol: An energy-saving Transport Protocol for Mobile IP-Devices", In Proceedings of the 7th IEEE International Conference on Network Protocols, 1999, Toronto, Canada.

5. V. Tsaoussidis, A. Lahanas and C. Zhang, "The Wave & Probe Communication Mechanisms", Journal of Supercomputing, Kluwer Academic Publishers, Vol. 20, No 2, September 2001.

6. H. Balakrishnan, V. Padmanabhan, and R. Katz, "The Effects of Asymmetry in TCP Performance", Proceedings of the 3rd ACM/IEEE Mobicom Conference, September 1997.

7. S. Gorinsky and H. Vin, "Additive Increase Appears Inferior", Technical Report TR2000-18, Department of Computer Sciences, The University of Texas at Austin, May 2000.

8. The X-Kernel: http://www.princeton.edu/xkernel

9. S. Floyd, and T. Henderson, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC2582, April 1999

APPENDIX

Figure 11. Application Success Percentage with Link Error (with playback interval of 100ms)

Figure 12. Application Success Percentage over Asymmetric Link (with playback interval 100ms)

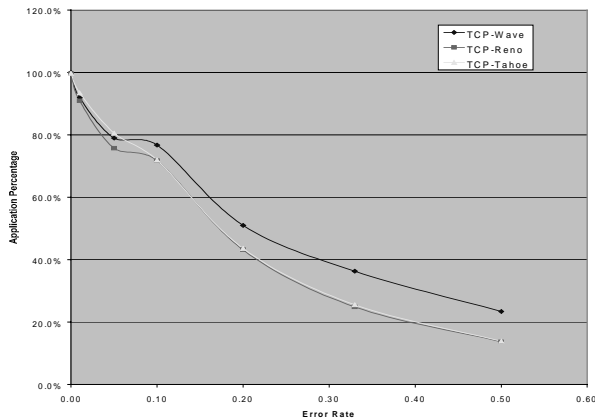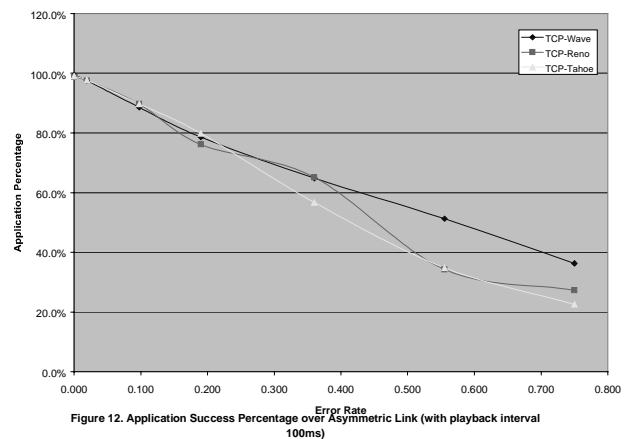| Tabel 1 TCP Goodput with Link Error				
Test Settings	Protocol	Dropping Rate	Transmission Time (s)	Goodput (Kbps)
1	TCP-Real	0.00	22.06	237.65
1	TCP-Reno	0.00	23.06	227.36
1	TCP-Tahoe		24.46	214.35
2	TCP-Real	0.01	26.39	198.68
2	TCP-Reno	0.01	27.49	190.73
2	TCP-Tahoe		28.71	182.62
3	TCP-Real	0.05	31.68	165.51
3	TCP-Reno	0.05	33.80	155.11
3	TCP-Tahoe		38.69	135.51
4	TCP-Real	0.10	41.39	126.66
4	TCP-Reno	0.10	56.85	92.23
4	TCP-Tahoe		41.14	127.45
5	TCP-Real	0.20	51.42	101.96
5	TCP-Reno	0.20	73.49	71.34
5	TCP-Tahoe		84.73	61.88
6	TCP-Real	0.33	112.96	46.41
6	TCP-Reno	0.33	198.48	26.42
6	TCP-Tahoe		160.54	32.66
7	TCP-Real	0.50	204.45	25.64
7	TCP-Reno	0.50	312.58	16.77
7	TCP-Tahoe		328.35	15.97

Table 2. TCP Goodput over Asymmetric Links				
Test Settings	Protocol	Reverse Path Dropping Rate	Transmission Time (s)	Goodput (KBps)
1	TCP-Real	0.000	22.42	233.87
1	TCP-Reno	0.000	22.41	233.91
1	TCP-Tahoe		23.19	226.08
2	TCP-Real	0.019	24.13	217.29
2	TCP-Reno	0.019	23.70	221.26
2	TCP-Tahoe		25.10	208.90
3	TCP-Real	0.098	28.02	187.09
3	TCP-Reno	0.098	28.19	186.00
3	TCP-Tahoe		30.37	172.62
4	TCP-Real	0.190	32.39	161.87
4	TCP-Reno	0.190	30.48	171.99
4	TCP-Tahoe		30.81	170.15
5	TCP-Real	0.360	45.87	114.30
5	TCP-Reno	0.360	46.20	113.49
5	TCP-Tahoe		52.63	99.62
6	TCP-Real	0.555	51.16	102.47
6	TCP-Reno	0.555	72.87	71.95
6	TCP-Tahoe		68.94	76.05
7	TCP-Real	0.750	75.30	69.62
7	TCP-Reno	0.750	213.42	24.57
7	TCP-Tahoe		120.53	43.50

Table 3. TCP Goodput with Handoff				
Test Settings	Protocol	Dropping Rate	Transmission Time (s)	Goodput (KBps)
1	TCP-Real	0.99	95.38	549.67
1	TCP-Reno	0.99	168.59	310.98
1	TCP-Tahoe		169.81	308.75

Table 4. Application Success Percentage with Link Errors			
Test Settings	Protocol	Dropping Rate	Application
1	TCP-Real	0.00	100.0%
1	TCP-Reno	0.00	99.6%
1	TCP-Tahoe		99.7%
2	TCP-Real	0.01	91.9%
2	TCP-Reno	0.01	90.9%
2	TCP-Tahoe		93.4%
3	TCP-Real	0.05	79.0%
3	TCP-Reno	0.05	75.7%
3	TCP-Tahoe		80.6%
4	TCP-Real	0.10	76.7%
4	TCP-Reno	0.10	71.7%
4	TCP-Tahoe		71.9%
5	TCP-Real	0.20	51.0%
5	TCP-Reno	0.20	43.1%
5	TCP-Tahoe		43.6%
6	TCP-Real	0.33	36.3%
6	TCP-Reno	0.33	24.9%
6	TCP-Tahoe		25.6%
7	TCP-Real	0.50	23.4%
7	TCP-Reno	0.50	13.7%
7	TCP-Tahoe		13.9%

Table 5. Application Success Percentage over Assymmetric Links			
Test Settings	Protocol	Reverse PathDropping Rate	Application Percentage
1	TCP-Real	0.000	99.4%
1	TCP-Reno	0.000	98.7%
1	TCP-Tahoe		99.0%
2	TCP-Real	0.019	97.6%
2	TCP-Reno	0.019	97.6%
2	TCP-Tahoe		97.7%
3	TCP-Real	0.098	88.6%
3	TCP-Reno	0.098	89.7%
3	TCP-Tahoe		89.7%
4	TCP-Real	0.190	78.6%
4	TCP-Reno	0.190	76.1%
4	TCP-Tahoe		79.9%
5	TCP-Real	0.360	64.9%
5	TCP-Reno	0.360	65.1%
5	TCP-Tahoe		56.7%
6	TCP-Real	0.555	51.3%
6	TCP-Reno	0.555	34.3%
6	TCP-Tahoe		34.9%
7	TCP-Real	0.750	36.3%
7	TCP-Reno	0.750	27.3%
7	TCP-Tahoe		22.6%

Table 6. Application Success Percentage with Handoff			
Test Settings	Protocol	Dropping Rate	Application Percentage
1	Real	0.99	72.57%
1	TCP-Reno	0.99	57.29%
1	TCP-Tahoe		62.57%

Table 7. TCP Goodput over WAN with Link Error				
Test Settings	Protocol	Dropping Rate	Transmission Time (s)	Goodput (KBps)
1	TCP-Real	0.00	16.61	315.61
1	TCP-Reno	0.00	11.59	452.44
1	TCP-Tahoe		10.36	506.26
2	TCP-Real	0.01	15.61	335.91
2	TCP-Reno	0.01	14.50	361.68
2	TCP-Tahoe		16.50	317.67
3	TCP-Real	0.05	20.21	259.42
3	TCP-Reno	0.05	40.76	128.63
3	TCP-Tahoe		41.19	127.30
4	TCP-Real	0.10	23.21	225.87
4	TCP-Reno	0.10	69.30	75.66
4	TCP-Tahoe		58.92	88.98
5	TCP-Real	0.20	34.72	151.00
5	TCP-Reno	0.20	148.44	35.32
5	TCP-Tahoe		160.30	32.71
6	TCP-Real	0.33	152.11	34.47
6	TCP-Reno	0.33	430.43	12.18
6	TCP-Tahoe		324.40	16.16
7	TCP-Real	0.50	152.71	34.33
7	TCP-Reno	0.50	1635.26	3.21
7	TCP-Tahoe		1431.15	3.66

Table 8. TCP Goodput over WAN with Asymmetric Link Errors				
Test Settings	Protocol	Reverse Path	Transmission	Goodput
1	TCP-Real	0.000	16.96	309.10
1	TCP-Reno	0.000	9.93	528.20
1	TCP-Tahoe		9.93	528.20
2	TCP-Real	0.019	16.00	327.64
2	TCP-Reno	0.019	10.31	508.52
2	TCP-Tahoe		10.03	522.51
3	TCP-Real	0.098	14.80	354.25
3	TCP-Reno	0.098	11.15	470.30
3	TCP-Tahoe		13.73	381.91
4	TCP-Real	0.190	17.51	299.46
4	TCP-Reno	0.190	25.29	207.29
4	TCP-Tahoe		23.47	223.39
5	TCP-Real	0.360	29.13	179.97
5	TCP-Reno	0.360	75.23	69.69
5	TCP-Tahoe		43.78	119.74
6	TCP-Real	0.555	42.76	122.62
6	TCP-Reno	0.555	113.32	46.27
6	TCP-Tahoe		152.22	34.44
7	TCP-Real	0.750	65.15	80.47
7	TCP-Reno	0.750	250.34	20.94
7	TCP-Tahoe		250.84	20.90

Table 9. TCP Goodput with Handoff over WAN				
Test Settings	Protocol	Dropping Rate	Connection Time (s)	Goodput (KBps)
1	TCP-Real	0.99	208.75	251.16
1	TCP-Reno	0.99	461.62	113.58
1	TCP-Tahoe		508.79	103.05

Table 10. Application Success Percentage over WAN with Link			
Test Settings	Protocol	Dropping Rate	Application Percentage
1	TCP-Real	0.00	98.8%
1	TCP-Reno	0.00	92.0%
1	TCP-Tahoe		93.8%
2	TCP-Real	0.01	88.3%
2	TCP-Reno	0.01	88.0%
2	TCP-Tahoe		83.0%
3	TCP-Real	0.05	72.8%
3	TCP-Reno	0.05	56.8%
3	TCP-Tahoe		59.0%
4	TCP-Real	0.10	63.8%
4	TCP-Reno	0.10	38.3%
4	TCP-Tahoe		48.0%
5	TCP-Real	0.20	52.5%
5	TCP-Reno	0.20	19.0%
5	TCP-Tahoe		19.3%
6	TCP-Real	0.33	17.5%
6	TCP-Reno	0.33	5.3%
6	TCP-Tahoe		8.5%
7	TCP-Real	0.50	15.3%
7	TCP-Reno	0.50	1.3%
7	TCP-Tahoe		1.3%

Table 11. Application Success Percentage over WAN with Asymmetric Link Erros			
Test Settings	Protocol	Dropping Rate	Application Percentage
1	TCP-Real	0.000	98.8%
1	TCP-Reno	0.000	93.6%
1	TCP-Tahoe		93.6%
2	TCP-Real	0.019	98.8%
2	TCP-Reno	0.019	93.0%
2	TCP-Tahoe		93.4%
3	TCP-Real	0.098	92.2%
3	TCP-Reno	0.098	92.0%
3	TCP-Tahoe		87.2%
4	TCP-Real	0.190	82.2%
4	TCP-Reno	0.190	69.4%
4	TCP-Tahoe		73.4%
5	TCP-Real	0.360	53.6%
5	TCP-Reno	0.360	37.2%
5	TCP-Tahoe		58.4%
6	TCP-Real	0.555	50.8%
6	TCP-Reno	0.555	24.0%
6	TCP-Tahoe		19.8%
7	TCP-Real	0.750	31.6%
7	TCP-Reno	0.750	9.4%
7	TCP-Tahoe		12.2%

Table 12. Application Success Percentage over WAN with Handoff			
Test Settings	Protocol	Dropping Rate	Application Percentage
1	TCP-Real	0.99	64.71%
1	TCP-Reno	0.99	45.00%
1	TCP-Tahoe		43.29%