CS 525M – Mobile and Ubiquitous Computing Seminar

Improving TCP Performance over Wireless Networks at the Link Layer

Christina Parsa &
J.J. Garcia-Luna-Aceves

Josh Schullman

TULIP

- TCP interprets packet loss as congestion!
  - Slow Start, Congestion Avoidance Visualization
- **T**ransport **U**naware **L**ink **I**mprovement **P**rotocol
  - *Service Aware*, not *Protocol Aware*
  - Half-Duplex oriented
  - Stateless!
    - Decisions made on a per-destination basis
  - Maintains local recovery of all lost packets
    - Sliding window
    - Lost packet retransmission handled by sender’s link
  - Exploits TCP timeouts

Related Work

- Link-Layer
  - AIRMAIL
    - Sends entire window of data prior to ACK response
    - Reduces ACK bandwidth consumption, power usage by mobile device
    - Must wait for end of window transmission for error correction; may lead to TCP timeouts
- Split Connection
  - Split Source/Base/Mobile Receiver
    - Base station buffers, acknowledges packets to source not yet ACK’ed by receiver. Violates TCP!!!
- Proxy
  - Proxy inserted between Sender/Receiver e.g., Snoop
    - Packet Sniffer, retransmits packets when detecting duplicate ACKs.

Service Basics…

- Reliable Service
  - RLP (reliable link-level packet)
    - Guarantees in-order delivery w/out duplicates in a given timeout window
  - TCP data ± TCP ACK (TACK)
- Unreliable Service
  - ULP (unreliable link-level packet)
  - TACK only
    - Assumption: +1 TACKs in transit
  - UDP packet
  - Link-level ACK (LACK)

Basic TULIP Operation

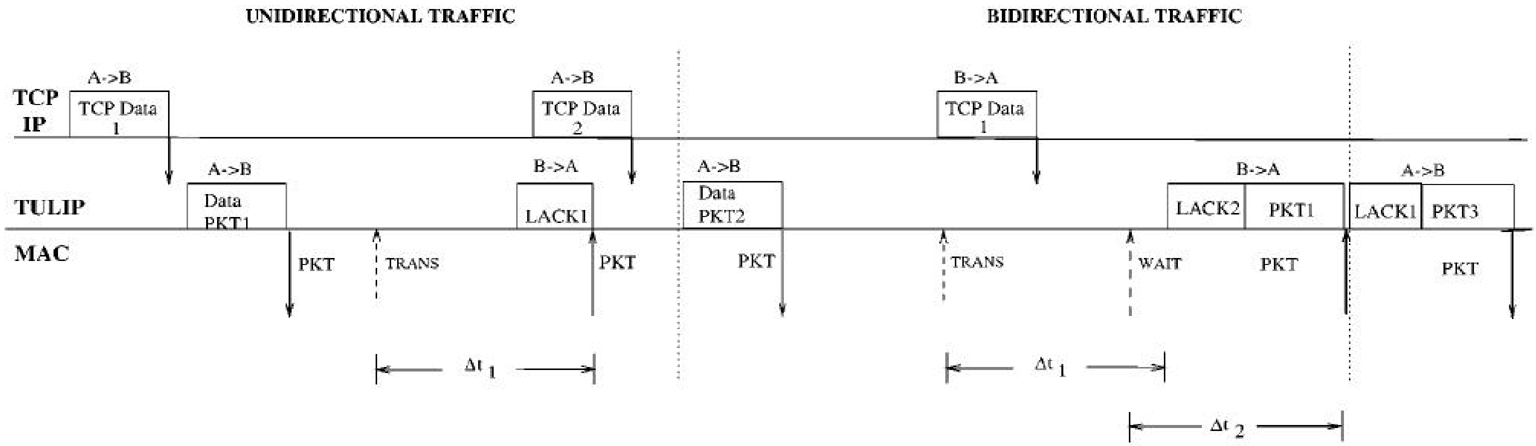


Figure 1. Packet interleaving over half-duplex link in TULIP from the perspective of source A.

- Packet interleaving requires transmission pacing per link, by maximum propagation delay (τ)
- At most, one packet in-transit at MAC layer
  - TRANS: transmission started
    - Send next packet after Δt_1 time
    - $\Delta t_1 = t_{PCK} + 2\tau + t_{ACK} + 2t_{TR} + 2t_c + t_p$
  - WAIT: additional time to wait (Δt_2)
    - Allows self-regulation during bi-di transfer

Flow Control / Error Recovery

- Transmitter utilizes sliding window (size *W*)
- Sequence numbers assigned modulo 2W
- Sender/Receiver maintain buffer pools (*W*)
- UnACKed transmission buffer (sender)
- Retransmission list

Sender Algorithm

```
Definition of Terms
    ACK = received pkt is an ACK
    WAIT = RTS received by MAC layer
    TRANS = MAC has acquired channel and
    pkt is to be transmitted
    macState = 1 if MAC layer has a packet,
    0 otherwise
    S = {SN_min, ..., SN_max}
    W = window size

Initialization
    Initialize SN_min and SN_max to 0

[This procedure is called when the sender receives
a signal or LACK packet from the MAC layer]
procedure receive_from_MAC ( incoming_pkt or signal )
    begin
        switch packet or signal type
            LACK:
                cancel Timer T1
                process_received_ack()
                if(data to send)
                    send_packet()
            WAIT:
                cancel Timer T1
                set timer T2 = Δt2(RTS.data_length)
            TRANS:
                set timer T1 = Δt1(mypacket.length)
                macState=0
    end process_receive_from_MAC

[This procedure is executed upon timer expiration]
procedure process_timer_expire
    begin
        if (data to send) AND (macState==0)
            send_packet()
            macState=1
        else
            return
    end process_timer_expire

procedure process_received_ack ( incoming_pkt )
    begin
        if pkt.CumAck ∈ S
            SN_min = (pkt.CumAck + 1)mod 2W
        end if
        if pkt.BitVector ≠ ∅
            free_received_packets ( incoming_pkt.BitVector)
            create new retransmission list
        end if
    end process_received_ack

procedure send_packet ( )
    begin
        if (untransmitted packets remain in Retransmission list)
            send next pkt in list
            else if (Window is not exhausted) AND (new pkt available)
                    if (new pkt is RDP) AND |S| < W
                        SN_max = (SN_max + 1)mod 2W
                        send packet with sn = SN_max
                    else if (new pkt us URDP)
                        send new packet
                else if (Retransmission List exists)
                    retransmit first pkt in list, i.e. start over
                    else (no retransmission list)
                        retransmit oldest unacknowledged packet
    end send_packet
```

Figure 2. Complete sender algorithm.

Receiver Algorithm

```
Initialization
    CumACK = −1
    BitVector = {0, ···, 0}

[this procedure is called when a pkt is received]
procedure process_incoming_pkt ( incoming_pkt.sn )
    begin
        if incoming_pkt.sn ∈ {CumACK + 1, ···, CumACK + W}
            if incoming_pkt.sn = (CumACK+1) mod 2W
                release to network layer
                release any other in sequence packets
                for each packet released
                    shift left BitVector
                CumACK ← LastReleased.sn
            else if packet not in buffer
                accept packet into buffer
                set corresponding bit in Bit Vector
            else
                drop packet
                return
            if (noDataPkt in Queue)
                prepare_ACK_pkt( CumACK, BitVector )
                send_ACK_pkt( sender_address )
            else
                prepare_PiggyBack_ACK_pkt( CumACK, BitVector, DataPkt )
                send_PiggyBack_ACK_pkt( sender_address, Piggyback_dgParms )
            end if
        else
            drop packet
        end if
    end process_incoming_pkt
```

Figure 3. Complete receiver algorithm.

Sample Transmission

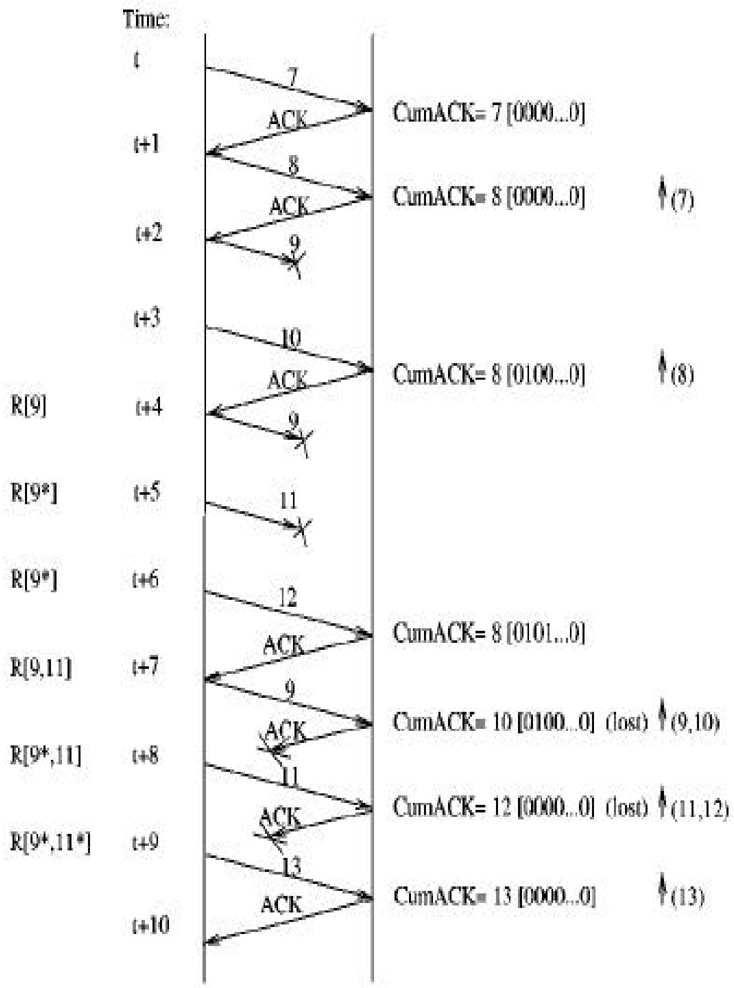


Figure 6. Example of transmission. Window size = 8.

- Retransmission list
  - *R[sn_i, … , sn_n]*
  - *R[sn_i*]*
- Bit Vector
  - Represents Negative ACKs
  - CumACK *N*[0100…0]
    - Sequence *N+1* NACK'ed

MAC-level Acceleration

- Reduce transmission delays via cooperative TULIP/MAC interaction
- FAMA receives data packet, sends to TULIP
- TULIP notifies FAMA of packet payload
  - If size == 0, send ACK
  - Else if size <= 40, send packet + ACK
  - Else, send RTS to request channel
  - Why 40 bytes? Large enough to carry a TACK
- Eliminates assumption that all packets are +40 bytes
  - In doing so, reduces MAC-level overhead to acquire the channel

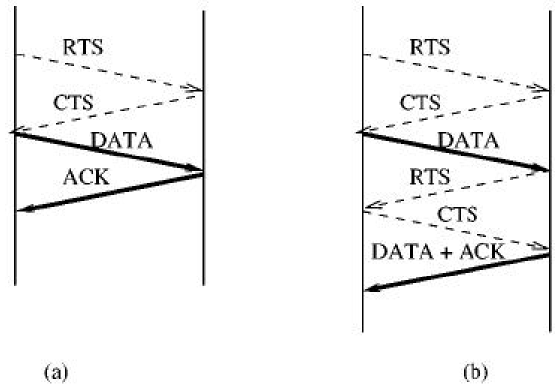


Figure 4. (a) *MAC Acceleration:* FAMA transmits TULIP data packet and returns ACK without another RTS/CTS exchange. Returning TULIP ACK may contain a TCP ACK. (b) FAMA exchange with large ACK packet (encapsulated data packet) requires another RTS/CTS exchange.

MAC-level Acceleration

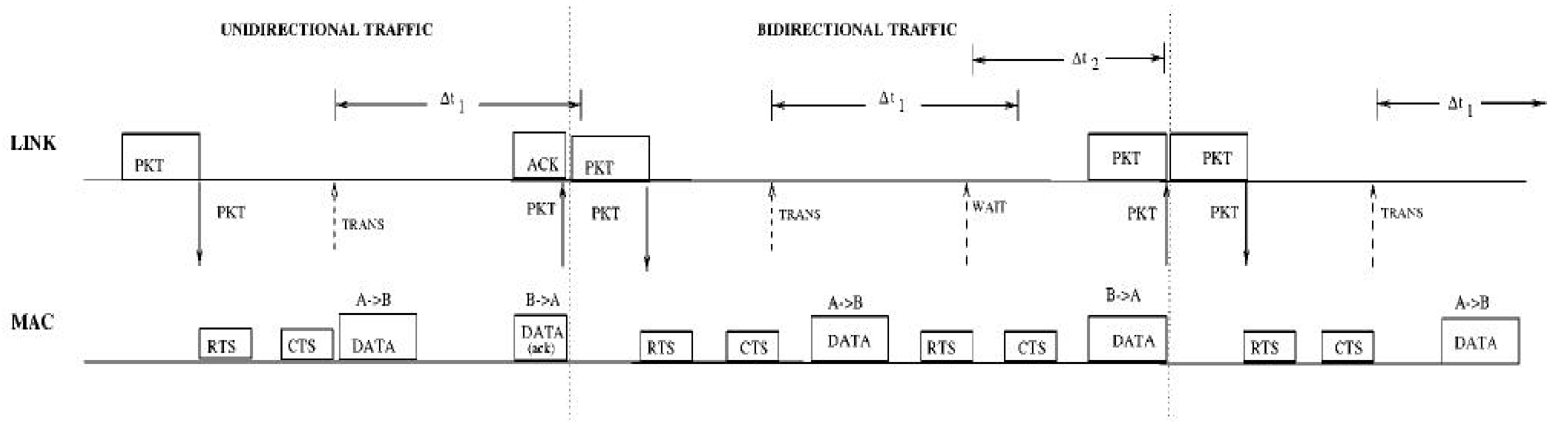


Figure 5. Unidirectional and bidirectional Traffic from the perspective of Node A in a logical link with Node B.

- TRANS: acquired channel, data packet about to be transmitted
- WAIT: received RTS (sends source address, packet size to link-layer)

Implementation

- Implemented TULIP, Snoop in C++ Protocol Toolkit
- Simulation based on same source code as WING prototypes
- IEEE 802.11 physical layer emulation

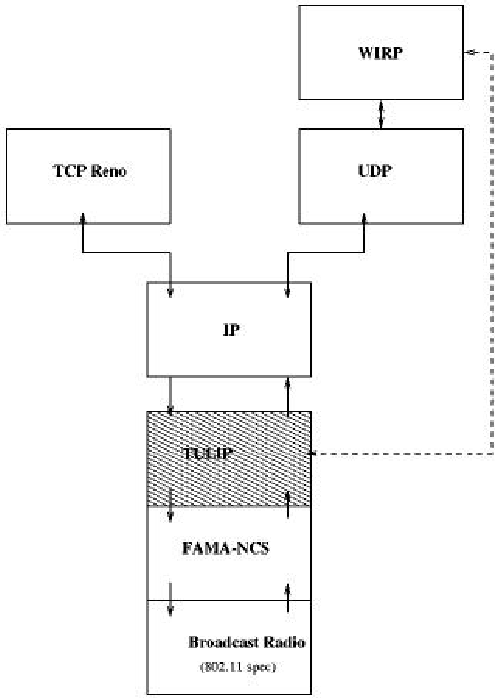


Figure 7. Protocol stack of wireless node.

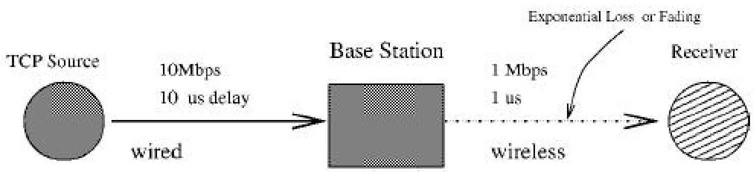


Figure 8. Topology for Experiments 1–3.

Experiment 1: Throughput

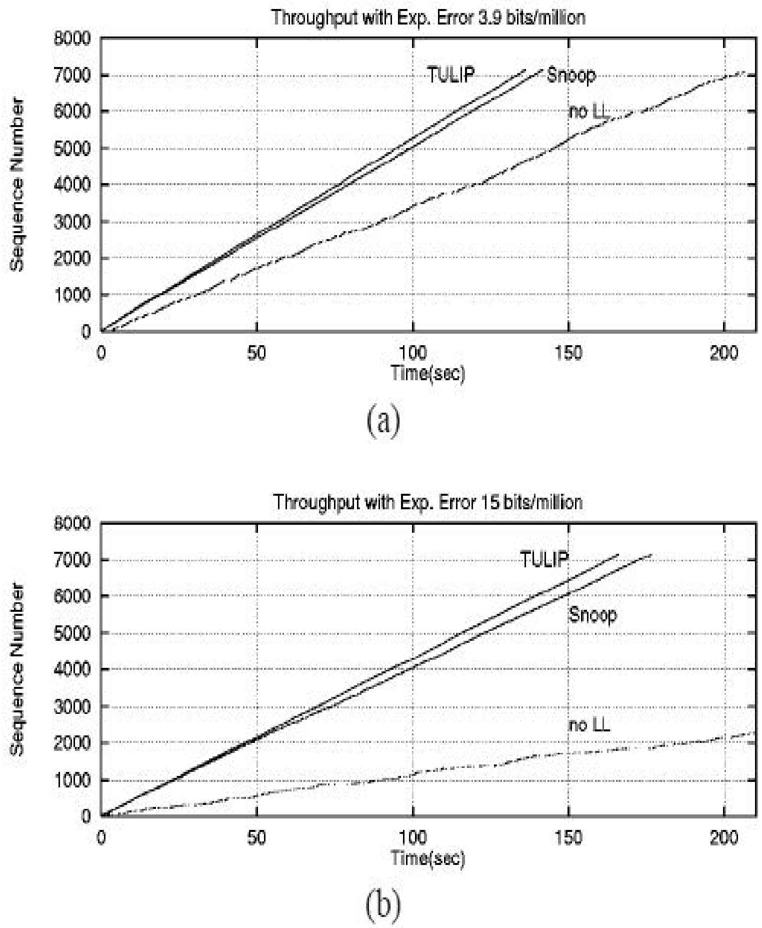


Figure 9. Experiment 1: TCP Sequence number growth. (a) BER = 3.9 bits/million = 1/256 Kbytes, (b) BER = 15 bits/million = 1/64 Kbytes. Receiver window 42 Kbytes.

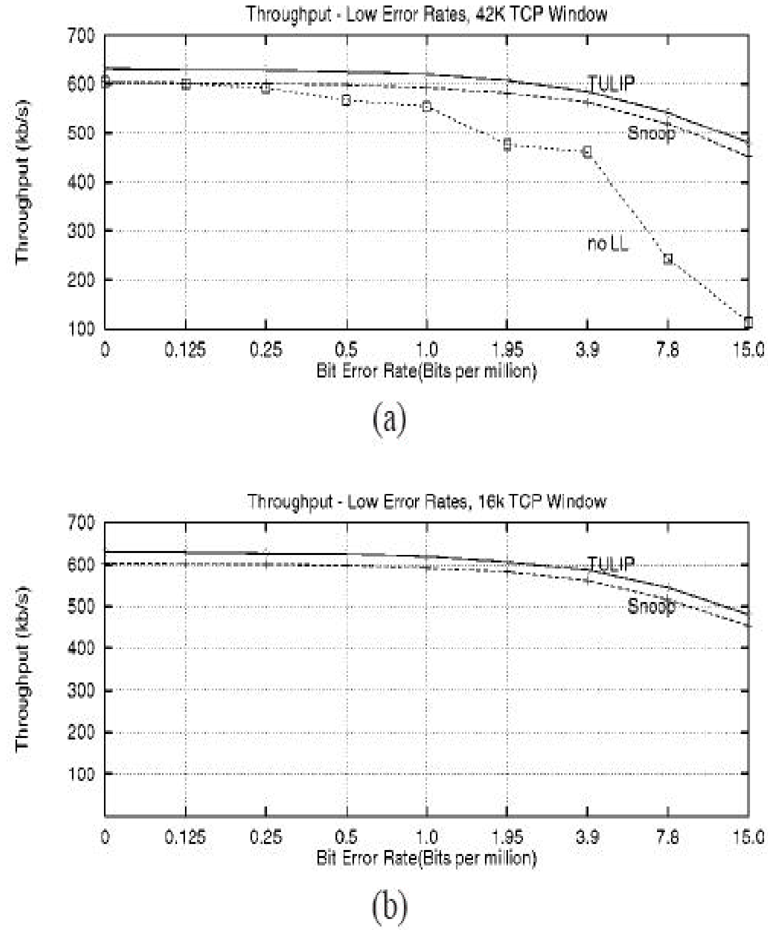


Figure 10. Experiment 1: Average throughput for all three protocols with varying BER. (a) 42 Kbytes receiver window, (b) 16 Kbytes receiver window.

Experiment 1: Goodput, Retransmissions

Table 1

Experiment 1: Ideal and achieved goodput, number of TCP timeouts and redundant packets transmitted over the wireless link for a BER of 15 bits per million and a receiver window of 42 Kbytes.

Comparison of goodput for TULIP protocol, Snoop and no LL

Protocol	BER (bits/million)	Packet loss (percent)	Ideal goodput	Achieved goodput	#TCP timeouts	#redundant packets
TULIP	15.1	0.159	0.841	0.840	0	0
Snoop	15.5	0.169	0.831	0.829	0	0
No LL	15.2	0.166	0.834	0.814	732	158

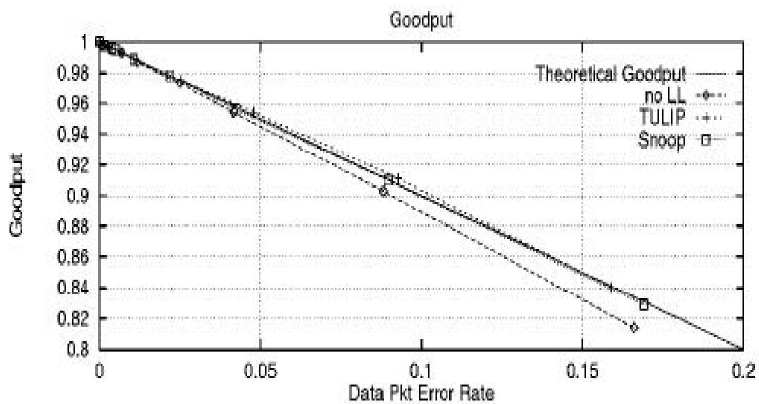


Figure 11. Experiment 1: Goodput for all three protocols with varying packet error rates and a receiver window of 42 Kbytes.

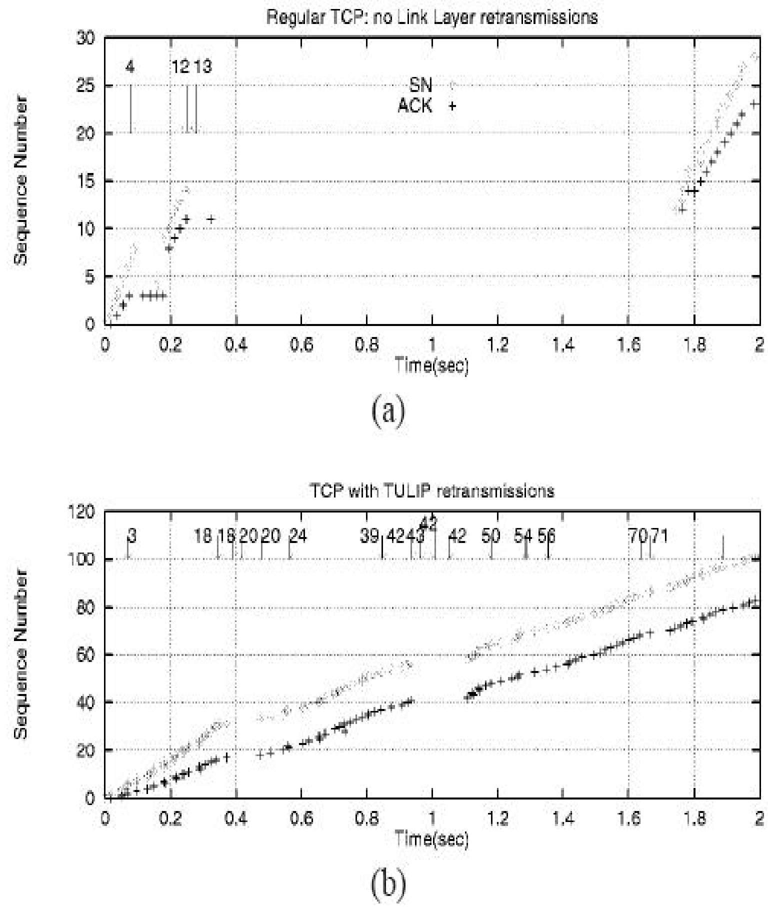


Figure 12. Experiment 1: Sequence number and ACKs at the source for the first 2 seconds of the TCP transfer. Packets dropped on the channel are shown with arrows. BER = 15 bits per million. (a) No link retransmissions, (b) TULIP protocol.

Experiment 1: RTT & Delay

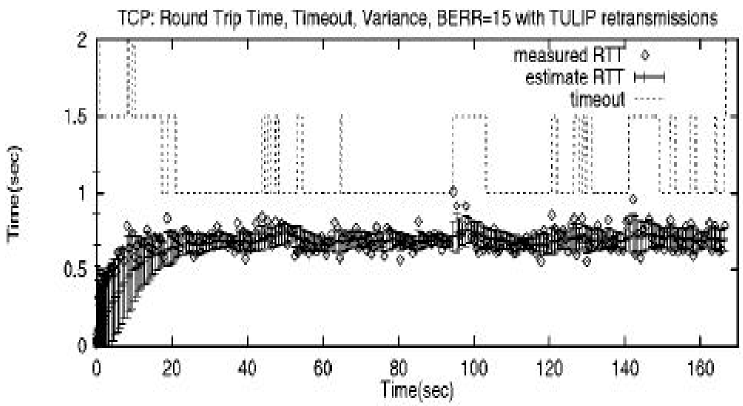


Figure 13. Experiment 1: RTT measurement and estimate for TULIP with a BER of 15 bits/million with a receiver window of 42 Kbytes.

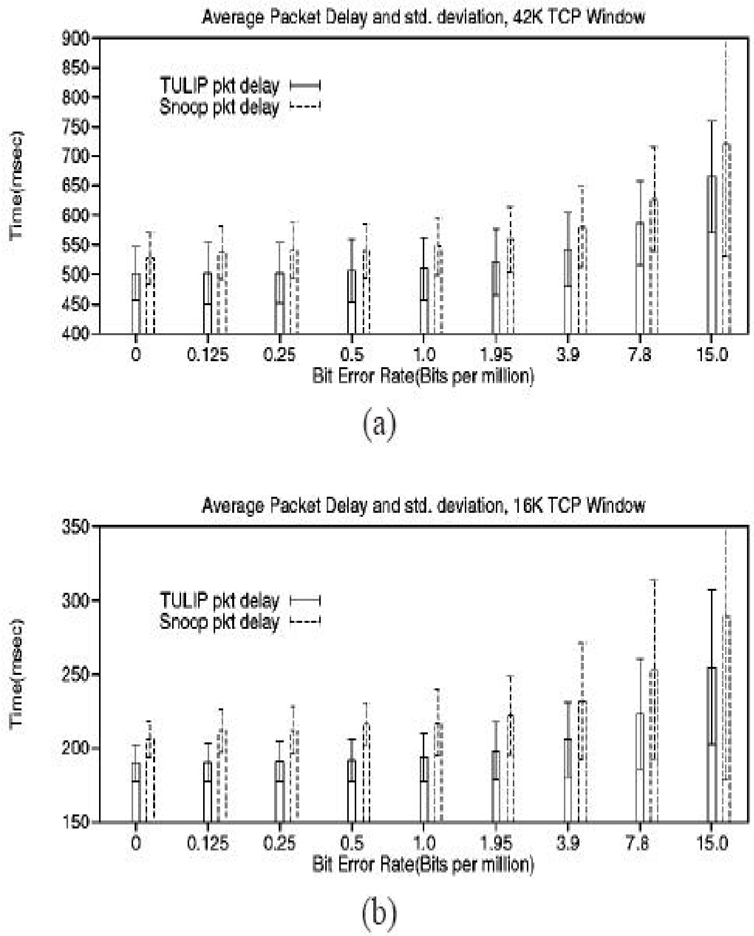


Figure 14. Experiment 1: Average packet delay and std. deviation for TULIP and Snoop protocols. (a) Receiver window is 42 Kbytes. (b) Receiver window is 16 Kbytes.

Experiment 2: Throughput & Delay

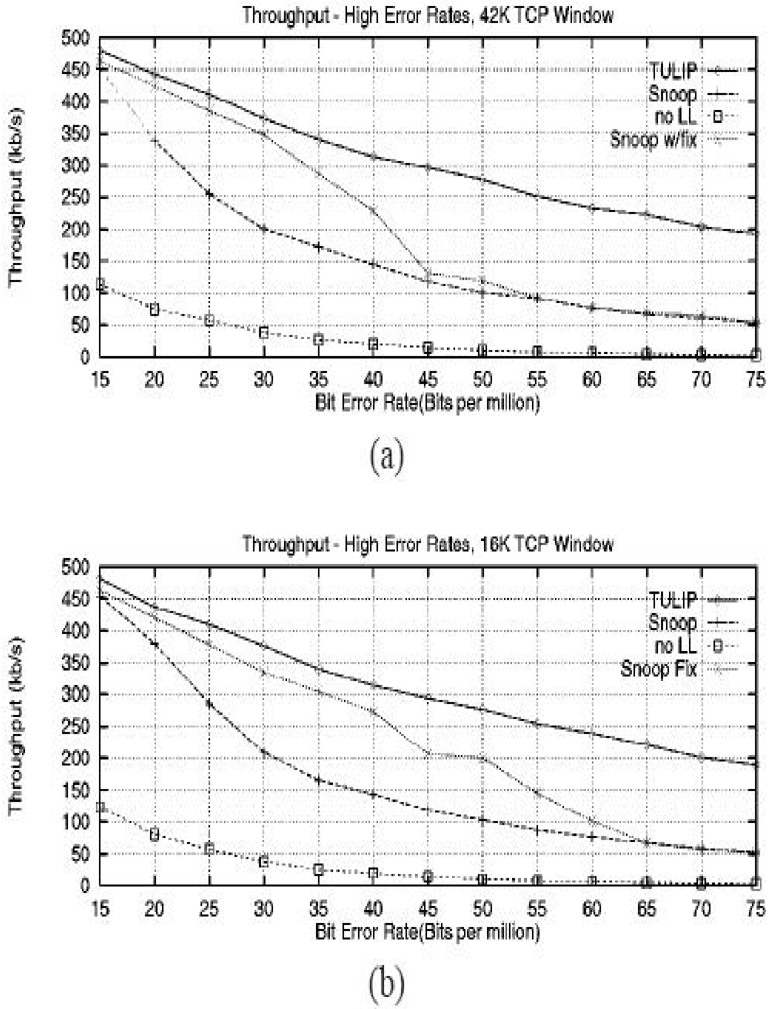


Figure 15. Average throughput for all three protocols with high error rates. (a) 42 Kbyte window, (b) 16 Kbyte window.

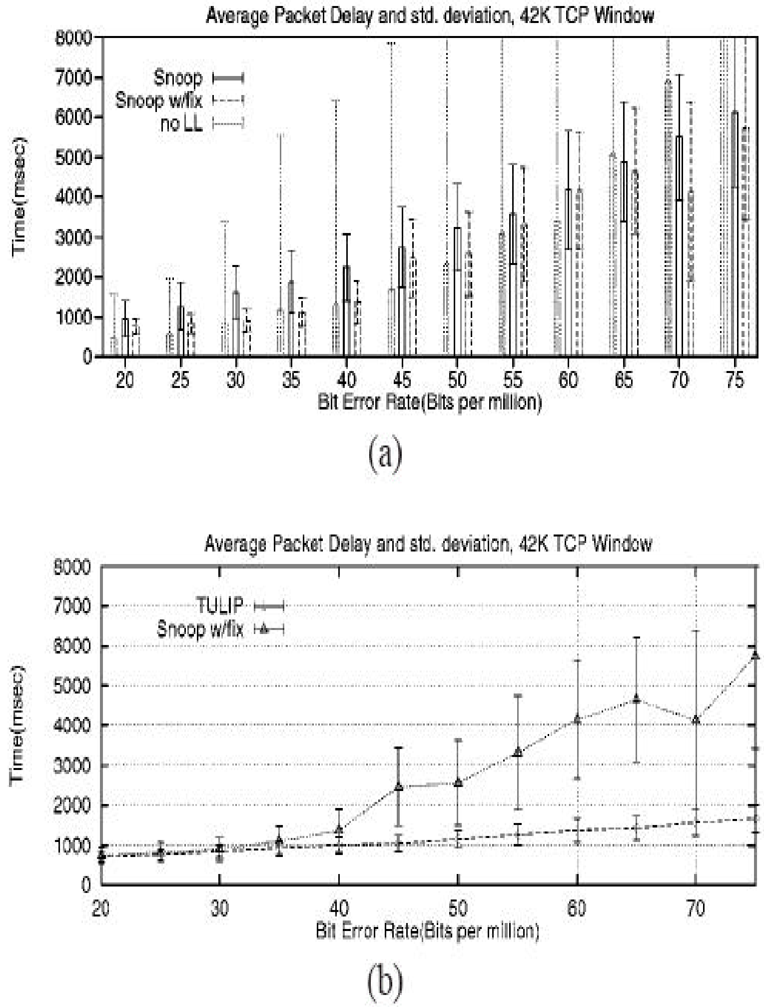
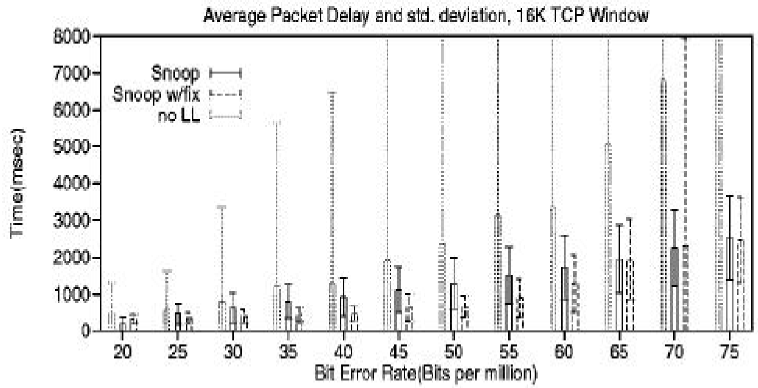
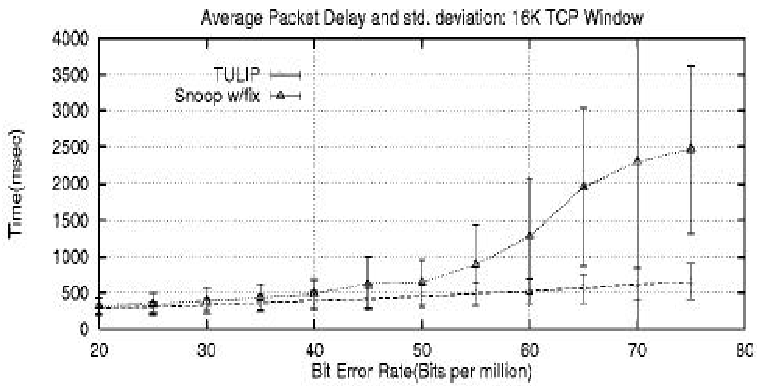


Figure 16. Experiment 2: Average packet end-to-end delay and std. deviation with high error rates and 42 Kbyte receiver window. (a) Snoop, Snoop w/fix, and no LL, (b) Snoop w/fix and TULIP.

Experiment 2: Delay



(a)



(b)

Figure 17. Experiment 2: Average end-to-end packet delay and std. deviation with high error rates and 16 Kbyte receiver window. (a) Snoop, Snoop w/fix and no LL, (b) Snoop w/fix and TULIP.

Experiment 3: Fading & Burst Losses

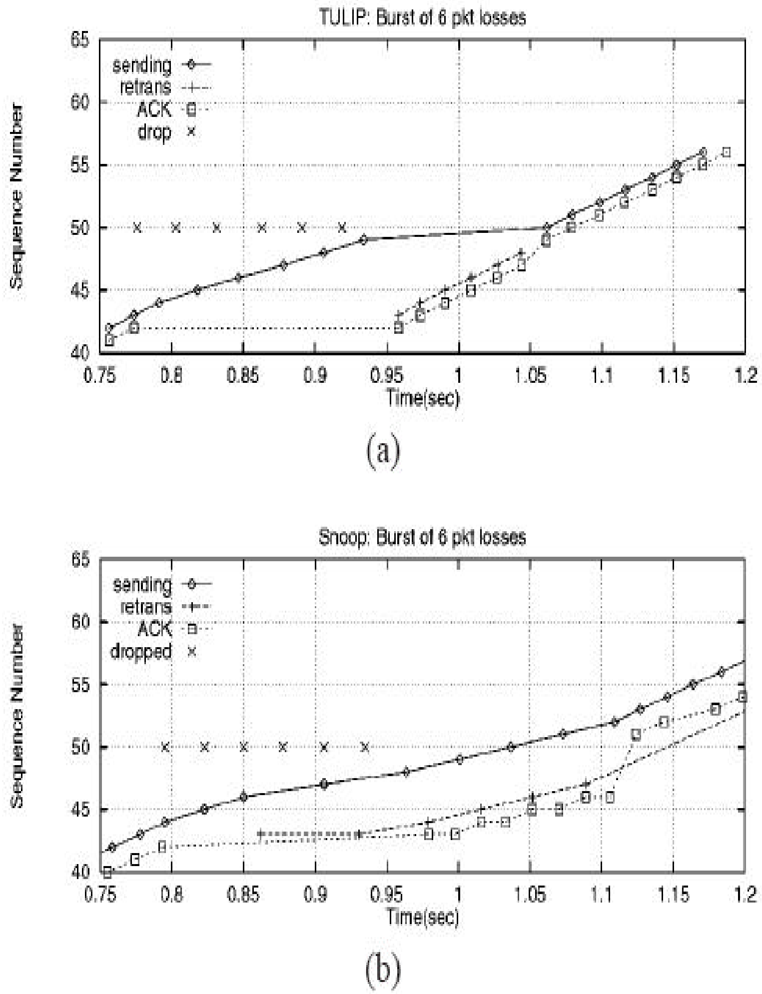


Figure 18. Experiment 5a: Burst loss of 6 packets. (a) TULIP, (b) Snoop.

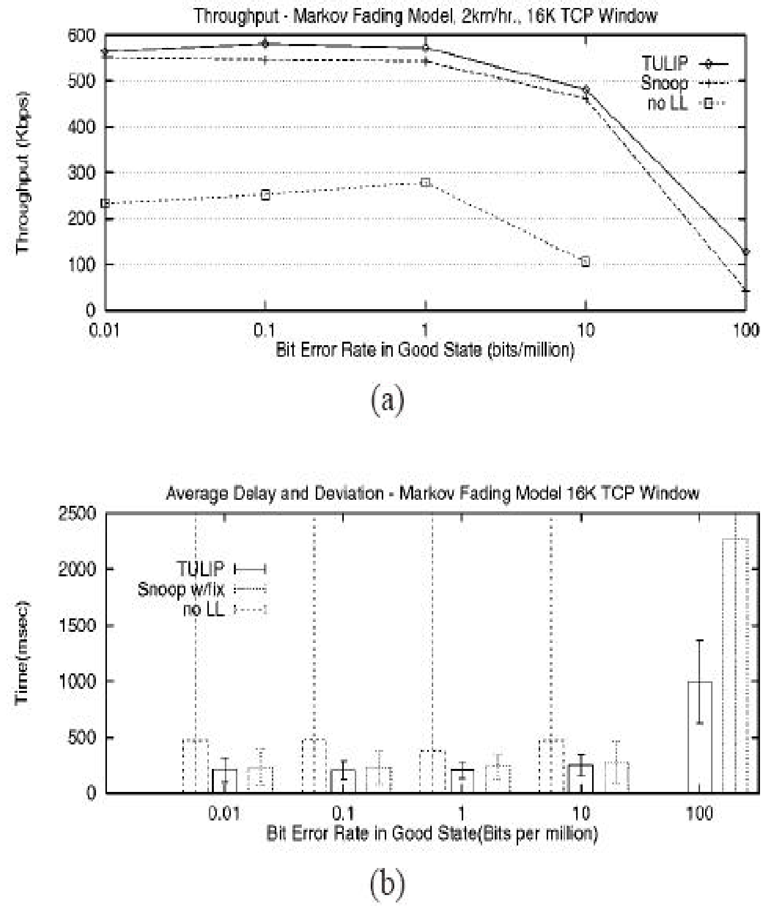


Figure 19. Experiment 5b: TULIP and Snoop protocols during Markov fading model. Loss probability in bad state is 50% and BER in good state is varied. Pedestrian speed 2 km/h. (a) Throughput, (b) end-to-end delay and standard deviation.

Experiment 3: Fading & Burst Losses

Experiment 5a: Throughput of TULIP and Snoop in the presence of bursts of length 2, 4 and 6 packets. Burst periods are distributed every 64 Kbytes of data. Receiver window is 42 Kbytes.

Bursts distributed every 64 Kbytes					
Burst size #packets	TULIP throughput (Kbps)	Snoop throughput (Kbps)	Δ (Kbps)	TULIP delay $\pm$ dev. (ms)	Snoop delay $\pm$ dev. (ms)
2	587.3	562.6	24.7	540 ± 56	582 ± 60 1
4	550.0	527.6	22.4	579 ± 74	621 ± 84 1
6	516.1	496.4	19.7	618 ± 98	660 ± 114

Conclusions

- TULIP successfully hides packet loss from TCP
- TULIP proves to be more successful at reducing timeouts due to varying BERs than Snoop
- Exploits normal link-MAC layer interaction
  - Reduces bandwidth consumption, etc.
- Last but not least, **STATELESS!!!**
  - Lends itself to be extremely scalable, since it is essentially TCP-version independent