Sting: a TCP-based network measurement tool

Stefan Savage

Department of Computer Science and Engineering
University of Washington
Simple problem

- Can we measure one-way packet loss rates to and from unmodified hosts?

- ICMP-based tools (e.g. ping)
  - Can’t measure one-way loss
  - Must cope with degraded service for ICMP

- Measurement infrastructures (e.g. NIMI)
  - Require cooperation from remote endpoints
Mind-expanding moment…

Stop thinking of TCP as a transport protocol

Think of it… as an opportunity
Overview

- Sting’s key features
  - Measures one-way packet loss rates
  - Does not require remote cooperation
  - TCP-based measurement traffic (not filtered)

- Basic approach
  - Send selected TCP packets to remote host
  - Leverage TCP behavior to deduce which packets were lost in each direction
Deducing losses in a TCP transfer

- **What we know**
  - How many data packets we sent
  - How many acknowledgements we received

- **What we need to know**
  - How many data packets were received?
    - Remote host’s TCP MUST know
  - How many acknowledgements were sent?
    - Easy, if one ACK is sent for each data packet
      \((ACK\ parity)\)
How TCP reveals packet loss

- Data packets ordered by seq#
- ACK packets specify next seq# expected

Nothing lost

Data lost

ACK lost
Basic loss deduction algorithm

- **Data seeding phase**
  - Send $n$ packets (dataSent)
  - Count ACKs received (ackReceived)

- **Hole filling phase**
  - Send new packet; next ACK points to first loss
  - Reliably retransmit lost packet and increment count of lost data (dataLost)
  - Repeat until all packets delivered
Example

Data seeding

- dataSent = 3
- ackReceived = 1
- ackSent = dataReceived = 2

\[ Loss_{fwd} = 1 - \frac{dataReceived}{dataSent} = 33\% \]

Hole filling

- dataLost = 1

\[ Loss_{rev} = 1 - \frac{ackReceived}{ackSent} = 50\% \]
Guaranteeing ACK parity

- How do we know one ACK is sent for each data packet received?

- Exploit TCP’s fast retransmit algorithm
  - *TCP must send an immediate ACK for each out-of-order packet it receives*

- Send all data packets out-of-order
  - Skip first sequence number
  - Don’t count first “hole” in *hole filling* phase
Managing limited receiver buffers

- Large packets can overflow receiver buffer
- Mitigate by overlapping sequence numbers

Sequence space

5 packets sent (7500 bytes)

1504 bytes of buffer used
Delaying connection termination

- Some Web servers/firewalls terminate connections abruptly by sending RST
- Solutions:
  - Format data packets as valid HTTP request
  - Set advertised receiver window to 0 bytes
    - TCP flow control prevents server from sending
    - HTTP response, hence RST, trapped at server
Sting implementation details

- Raw sockets to send TCP datagrams
- Packet filter (libpcap) to get responses
- **Problems** with packet filters
  - Very easy to race with native TCP stack
  - Fragile; next version will use OS-specific firewall interfaces
- Currently runs on Tru64 and FreeBSD
Last-generation user interface

# sting -c 100 -f poisson -m 0.500 -p 80 www.audiofind.com

Source = 128.95.2.93
Target = 207.138.37.3:80
dataSent = 100
dataReceived = 98
acksSent = 98
acksReceived = 97
Forward drop rate = 0.020000
Reverse drop rate = 0.010204
Preliminary experimental results

○ Anecdotally
  ■ Works great for debugging operational problems

○ Real data
  ■ Measured loss rates from UW to 50 web servers (25 random, 25 popular)
  ■ Significant loss rate asymmetry
Distribution of losses to Web servers

25 Popular servers

25 Random servers

Cumulative fraction

Forward loss rate

Reverse loss rate

Loss rate
Conclusions

- TCP protocol features can be leveraged for non-standard purposes
- Packet loss is highly asymmetric
- Ongoing work:
  - Using TCP to estimate one-way queuing delays, bottleneck bandwidths, propagation delay and server load
Loss rate as ping sees it

- Reported loss = 1 – \((1 - \text{loss}_{\text{forw}})(1 - \text{loss}_{\text{rev}})\)
- Both of these cases look the same:

![Diagram showing the process of an EchoReq and EchoReply]

- You
  - Target
  - EchoReq
  - EchoReply
User/kernel races with packet filters
Example: diurnal effects

Reverse path loss rate to www.idg.net
ICMP rate limiting

Client

Server

ICMP EchoReq

ICMP EchoReq

ICMP EchoReq

ICMP EchoRep

Time
ICMP spoofing

Client | Firewall | Server
---|---|---
ICMP EchoReq | | TCP ACK
ICMP EchoRep | | TCP DATA
TCP ACK | | TCP DATA
TCP DATA | | TCP DATA

Time
Mapping seq#'s to packets

Sequence space

1

1500

1504

5 packets sent (7500 bytes)

1500 bytes

1500 bytes

1500 bytes

1500 bytes

1500 bytes

1504 bytes of buffer used
Data Seeding phase

for i = 1 to n
    send packet w/seq #i
    dataSent++
    wait for long time

for each ack received
    ackReceived++
Hole Filling Phase

lastACK := 0
while lastAck = 0
    send packet w/seq # n+1
while lastAck < n + 1
    dataLost++
    retransPkt := lastAck
while lastAck = retransPkt
    send packet w/seq# retransPkt
dataReceived := dataSent – dataLost
ackSent := dataReceived

for each ack received w/ack #j
    lastAck = MAX(lastAck,j)
Distribution of losses to 25 popular Web servers

![Graph showing cumulative fraction vs. loss rate for forward and reverse loss rates.](image)
Distribution of losses to 25 random Web servers