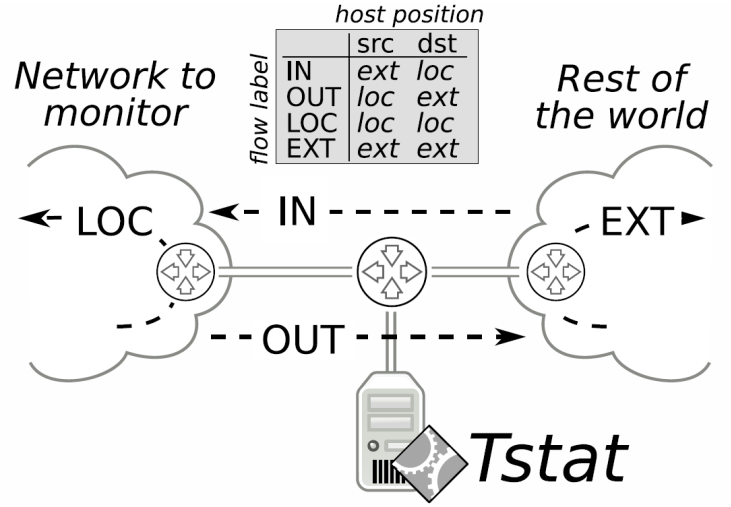
Networking Lab 9 – Passive monitoring

Network monitoring has always played a key role in understanding telecommunication networks since the pioneering time of the Internet. Today, monitoring traffic has become a key element to characterize network usage and users' activities, to understand how complex applications work, to identify anomalous or malicious behaviors. In this lab, we will start using Tstat, a free open source passive monitoring tool that has been developed in the past ten years. Started as a scalable tool to continuously monitor packets that flow on a link, Tstat has evolved into a complex application that gives to network researchers and operators the possibility to derive extended and complex measurements via advanced traffic classier.

1. Tstat

Tstat initial design objective was to automate the collection of TCP statistics of traffic aggregate, using real time monitoring features. Over the years, Tstat evolved into a more complex tool offering rich statistics and functions. Developed in ANSI C for efficiency purposes, Tstat is today an Open Source tool that allows sophisticated multi-Gigabit per second traffic analysis to be run live using common hardware. Tstat design is highly flexible, with several plug-in modules offering different capabilities that are briefly described in the following. Being a passive tool, live monitoring of Internet links, in which all transmitted packets are observed, is the typical usage scenario. Fig. 1 sketches the common setup for a probe running Tstat: on the left there is the network to monitor, e.g., a network, that is connected to the Internet through an access link that carries all packets originated from and destined to terminals in the monitored network. The Tstat probe observes the packets and extracts the desired information. Note that this scenario is common to a wide set of passive monitoring tools.



2. Monitored objects

The basic objects that passive monitoring tools considers are the *IP packets* that are transmitted on the monitored link. *Flows* are then typically defined by grouping, according to some rules, all packets identified by the same `flowID` and that have been observed in a given time interval. A common choice is to consider

```
flowID = (ipProtoType, ipSrcAddr, srcPort, ipDstAddr, dstPort)
```

so that TCP and UDP flows are automatically considered, and tracked. For example, in case of TCP, a new flow start is commonly identified when the TCP three-way handshake is observed; similarly, its end is triggered when either the proper TCP connection teardown is seen, or no packets have been observed for some *idle* time. Similarly, in case of UDP, a new flow is identified when the first packet is observed, and it is ended after an *idle* time. As Internet conversations are generally bidirectional, the two opposite unidirectional flows (i.e., having symmetric source and destination addresses and ports) are typically grouped and tracked as connections. This allows gathering separate statistics for client-to-server and server-to-client flow, e.g., the size of HTTP client request and server HTTP response. Furthermore, the origin of information can be distinguished, so that it is possible to separate local hosts from remote hosts in the big Internet. As depicted in Fig. 1, traffic is then organized in four classes:

- **incoming traffic**: the source is remote and the destination is local;
- **outgoing traffic**: the source is local and the destination is remote;
- **local traffic**: both source and destination are local;
- **external traffic**: both source and destination are remote.

This classification allows to separately collecting statistics about incoming and outgoing traffic; for example, one could be interested in knowing how much incoming traffic is due to YouTube, and how many users access Facebook from the monitored network. The local and external cases should not be considered but in some scenarios they can be present. At packet, flow and application layers, a large set of statistics can be defined and possibly customized at the user's will.

Tstat can analyze traffic traces, extracting as much information as possible from the pure passive observation of packets. It first rebuilds flows, and, for each flow, a number of statistics are collected. Once the flow is terminated, Tstat logs it in a simple text file, in which each column is an attribute of the flow. Several "*log files*" are then created, one per type of flows (e.g., TCP, UDP, Video, etc.). A *log file* is arranged as a simple table where each column is associated to specific information and each line reports the two unidirectional flows of a connection. Different flow-level logs are available, e.g., the log of all UDP flows, or the log of all VoIP calls. The log information is a summary of the connection or flow properties. For instance, the starting time of a TCP connection, its duration, the number of sent and received packets, the observed Round Trip Time are all valuable metrics that allow to monitor the TCP performance. The file containing the log of monitored TCP connections is called `log_tcp_complete`. Table 1 reports the description of each fields, while table 2 reports the classification of "Connection Type" used in field 101. More detailed statistics can be found on the Tstat web site at http://www.tstat.polito.it, in the referenced papers[1], and on the references listed in the Publications section. As it can be seen, the amount of information exposed by Tstat is large, and allows to easily computing other metrics by combining them.

See the file "*log_tcp_complete_description.pdf*" for more details.

3. Analysis of TCP logs

In this laboratory, the PC must be normally connected to the LAIB-4 network since we will use it as normal unix terminal. Most of the analysis can be also run on any PC, possibly using other software and any operating system.

Warning: if you cannot connect to the didattica.polito.it portal, it may be because the DNS is not properly configured in your system. Check the file /etc/resolv.conf. If it does not exist, try the following

```
# delete the wrong configuration
sudo rm /etc/resolv.conf
# replace with the one got from the DHCP
sudo ln –s /run/resolvconf/resolv.conf  /etc
```

We will consider TCP logs that have been collected from operative networks, and will learn how to post-process them to extract valuable information. To this goal, simple post-processing scripts will be required, implemented using any scripting languages, like `awk, perl, phyton`, etc. By combining them with some

[1] Alessandro Finamore, Marco Mellia, Michela Meo, Maurizio M. Munafò, Dario Rossi, *Experiences of Internet Traffic Monitoring with Tstat*, IEEE Network "March/April 2011", Vol.25, No.3, pp.8-14, ISSN: 0890-8044, March/April 2011

shell tools like `grep`, `sort`, `cut`, etc., it will be easy to derive aggregated statistics from the `log_tcp_complete` file. Remember that you can always check the command man page, and you can look for support in the Internet.

For this lab, we will use the file "`log_tcp_complete.xz`" that **must be downloaded** from the http://didattica.polito.it portal. This log has been collected from an operative network during November 2015 and contains all TCP flows observed considering a one-hour long time interval. The file has been compressed using `xz`, an efficient compression program, which reduced its size to 339MB from the original 2.1GB. The file can be easily accessed by using `xzcat` and using a pipe to input the uncompressed data to the post processing script/command. For instance

```
xzcat log_tcp_complete.xz | head -n 10
```

will print the first 10 lines of the file on the screen.

WARNING: Although decompressing the file every time can slow down the processing, you should consider avoiding decompress the file on the laboratory computers since the system has very limited disk space! You could consider working on a decompressed file that contains only the rows or columns that you are interested into. To test and tune the scripts, you should limit the analysis to the first few lines of the log, by using the `head -n XX` command.

The typical steps to follow to get a desired figure are

1. *Define the metric* that has to be computed, e.g., "distribution of number of transmitted packets per flow", or "average download throughput", or "number of IP addresses serving facebook.com traffic", etc.
2. *Define the set of flows* that are targeted, e.g., all HTTP traffic, or facebook.com flows coming from a given IP address, etc.
3. Compute the metric for the desired set of flows by writing a suitable script
4. Plot the results using for example gnuplot
5. Go back to 3 to refine the results
6. Go back to 1 to compute another metric and produce another plot

4. Example of scripts

For example, given the `log_tcp_complete.xz`, count

1. the number of total TCP connections
2. the number of TCP connections whose client is "local"
3. the number of TCP connections whose client is "local" and uses HTTP as L7-protocol
4. the number of TCP connections whose client is "local", uses HTTP as L7-protocol and are directed to "*facebook.com". Do the same for HTTPS flows
5. produce a histogram that counts, for each server IP address, the number of connections whose client is "local", uses HTTP as L7-protocol, are directed to "*facebook.com"
6. as above, but sort the results in increasing number of flows
7. **plot the results using gnuplot**

5. SOLUTIONS

1. Simply count the number of rows in the file:

```
#using awk to count the number of lines
xzcat log_tcp_complete.xz | awk '{tot++} END {print "total number of lines", tot}'
```

Other possibility:

```
xzcat log_tcp_complete.xz | awk 'END {print NR}' # recall that NR is the current line number …
```

or even simpler using the `wc` unix command

```
xzcat log_tcp_complete.xz | wc -l
```

2. This time we need to filter those flows whose client is marked as "internal"

```
# using awk to count the number of lines which take the value of
#    "1"    in    the    38th    column    -    client    internal
xzcat log_tcp_complete.xz | awk '{if ($38==1) tot_int++} END {print "number of flows initiated by internal clients", tot_int}'
```

3. We need to consider those lines in which the 38th field takes the value of 1 (client internal), and then 42nd connection type field takes the value of 1 (HTTP)

```
# using awk to count the number of lines which take
#  the value of "1" in the 38th column - client internal
# the value of "1" in the 42nd column - HTTP protocol
# note: the "\" - backslash - character is useful to break a single
# line into multiple lines
xzcat log_tcp_complete.xz | \
awk '{if ($38==1 && $42==1) tot_int++}\
END {\
print "number of HTTP flows initiated by internal clients", tot_int\
}'
```

4. As above, but consider only those lines that match "facebook.com".

```
#  using awk to count the number of lines which take
#  the value of "1" in the 38th column - client internal
#  the value of "1" in the 42nd column - HTTP protocol
#  or the value of "8192" in the 42nd column - HTTPS protocol
# and match facebook.com in the line.
#   let's use the "filtering" functionalities of awk!
xzcat log_tcp_complete.xz | \
awk ' /facebook.com/ {\
  if ($38==1 && $42==1) tot_int_fb_http++;\
  if ($38==1 && $42==8192) tot_int_fb_https++\
}\
END {\
print "flows to facebook initiated by internal clients - HTTP",\
tot_int_fb_http \
print "flows to facebook initiated by internal clients - HTTPS",\
tot_int_fb_https \
}'
```

5. This time we need to count the number of flows per single IP addresses. We can use an associative array to count the number of flows directed to a given IP address. The server IP address will be used as "index" of the array.

```
# using awk to count the number of lines which take
# the value of "1" in the 38th column – client internal
# the value of "1" in the 42nd column – HTTP protocol
# or the value of "8192" in the 42nd column – HTTPS protocol
# and match facebook.com in the line.
# Count per IP address the number of flows
#       so       create       and       array       this       time
xzcat log_tcp_complete.xz | \
awk '/facebook.com/ {\
  if ($38==1 && $42==1) tot_int_fb_http[$15]++;\
  if ($38==1 && $42==8192) tot_int_fb_https[$15]++\
}\
END { \
print "#IP address\t number of flows – HTTP flows";\
for (addr in tot_int_fb_http)\
   print "HTTP: " addr "\t" tot_int_fb_http[addr];\
print "#IP address\t number of flows – HTTPS flows";\
for (addr in tot_int_fb_https)\
   print "HTTPS: " addr "\t" tot_int_fb_https[addr]\
}'
```

6. As above, plus simply use the `sort` command.

```
# using awk to count the number of lines which take
# the value of "1" in the 38th column – client internal
# the value of "1" in the 42nd column – HTTP protocol
# or the value of "8192" in the 42nd column – HTTPS protocol
# and match facebook.com in the line.
# Count per IP address the number of flows
#       so       create       and       array       this       time
xzcat log_tcp_complete.xz | \
awk '/facebook.com/ {\
  if ($38==1 && $42==1) tot_int_fb_http[$15]++;\
  if ($38==1 && $42==8192) tot_int_fb_https[$15]++\
}\
END { \
print "#IP address\t number of flows – HTTP flows";\
for (addr in tot_int_fb_http)\
   print "HTTP: " addr "\t" tot_int_fb_http[addr];\
print "#IP address\t number of flows – HTTPS flows";\
for (addr in tot_int_fb_https)\
   print "HTTPS: " addr "\t" tot_int_fb_https[addr]\
}' | sort –n –k2
```

A much better way to organize the work is to create an AWK scripts instead of typing it on the command line directly. This can be done by simply creating a text file that contains the AWK commands, and then running it.

The above single-line-long-command can then be simplified as follows:

Create the file "`count_servers.awk`" that contains

```
/facebook.com/ {
  if ($38==1 && $42==1) tot_int_fb_http++;
  if ($38==1 && $42==8192) tot_int_fb_https++
}
END {
print "flows to facebook initiated by internal clients - HTTP", tot_int_fb_http
print "flows to facebook initiated by internal clients - HTTPS", tot_int_fb_https
}
```

Then, run it as before

```
xzcat log_tcp_complete.xz | awk –f count_servers.awk
```

Suggestions: when testing the scripts, you can run it on a smaller portion of the file to speed up testing, e.g., considering the first 10000 lines or so. Use the command `head –n 10000` for this.

```
xzcat log_tcp_complete.xz | head –n 1000 | awk –f count_servers.awk
```

1. Modify the previous script to find which is the facebook.com most used service. What if you check "facebook" instead of "facebook.com"? (Consider the name in column 127 as "service")
2. What are the top most used "services" in general?
3. **Write a script that counts the fraction of connections for different "connection type". Plot the results as a histogram**.
4. **Write a script that counts the fraction of HTTP flows every 10,000 connections. Plot the results over time.**
5. **Add a second line to the above plot that reports the fraction of HTTPS connections over time.**
6. **Add a third line with the number of other protocols flows over time.**
7. **Consider a given service (e.g., /facebook/, or /whatsapp/, or /google/, or /scorecardresearch/, …)**
    1. **Plot the number of flows per each sub-service**
    2. **Count the number of flows handled by each IP addresses matching the service. Plot the rank of IP server addresses, sorted from the one handling the most flows, to the least.**
    3. **Count the client IP addresses handled by each IP addresses matching the service. Plot the rank of IP server addresses, sorted from the one handling the most clients, to the least. Suggestion: use arrays of arrays, e.g., a[server][client] = 1…**
    4. **Count the total bytes handled by each IP addresses matching the service. Plot the rank of IP server addresses, sorted from the one handling the largest amount of traffic, to the least**
    5. **For each server IP address matching the service, extract the minimum and average RTT from the client to the server. Plot them, sorted from the one handling the most clients, to the least.**

8. What is the next script doing? Try to understand it

```
xzcat log_tcp_complete.xz | \
head -n 1000000| awk -f count_server_and_names.awk \
sort -n -k2
```

Content of the "count_server_and_names.awk" file

```
{
    if ($38==1 && $127 ~ /googlevideo/) {
        tot_int[$15]++;
        alreadyIn = match (name[$15],$127);
        if(alreadyIn==0)
           name[$15]=name[$15] "," $127
    }
}
END {
  print "number\tIP address\tnames";
  for (addr in tot_int) {
     print tot_int[addr] "\t" addr "\t" name[addr]
  }
}
```