POLITECNICO di TORINO
Dipartimento di Elettronica e Telecomunicazioni

# COURSE SYLLABUS - NETWORK LABORATORY
# SIMPLE SCRIPTING AND PLOTTING

Marco Mellia
E-mail: mellia@polito.it
February, 2021

## How to make the plots from traces using Bash

It is often required to extract specific data from trace files collected with wireshark or other tools. Here, we briefly describe how to write simple scripts that automatize this process. We'll be using the bash shell scripting language, but most of this processing could be generalized with other scripting languages, from awk to python. This is not supposed to be a complete guide to bash programming.

### Extracting data from Wireshark

Once you have collected a packet trace with wireshark the easiest way is to save the trace for later postprocessing is to write it on a `.txt` file. Later, some simple Bash scripts allows you to easily extract the desired fields you are interested into. You can follow these steps for this:

1. **Select the set of packets** you want to consider by using a proper visualization filter in wireshark. Often you need to separate client-to-server and server-to-client packets, remove undesired packets, etc. For this, write the most specific visualization filter.
2. **Export the selected packets** in ascii format. Use the menu `File->Export Packet Dissection -> as plain text`. In the option panel, select to export only the `Displayed` packets, and to write only the `summary line`. This will save a file where each row corresponds to one packet, with exactly the same information that is displayed in the summary line only.

### Useful Bash commands

Once you have saved the list of packets you are interested into, you can use Bash scripting tools to process the content and extract those values that you are interested into. The idea is to "cut" the selected field, and "paste" it is a file so to be ready for plotting. Below, a list of useful bash commands for this. You can consult the `man` page of each command to get more details and options.

### Extracting specific columns from files

- **`echo STRING`**: Echo the `STRING`(s) to standard output
  **`-n`** do not output the trailing newline
  `echo "hello word"` : print hello word
  `echo hello word` : same as above
  `echo $myVar` : prints the value of the variable `$myVar`
- **`cat[OPTION]... [FILE]...`**: concatenate files and print on the standard output. prints the [FILE] to the screen. Useful to feed the content of [FILE] to another command using a pipe.
- **`cut OPTION... [FILE]...`**: Print selected parts of lines from each FILE to standard output.
  `-d 'DELIM'`: use DELIM instead of TAB for field delimiter
  (e.g., -d ' ' uses space, or –d '=' uses the = char as delimiter)
  `-f` : select only these fields, e.g., -f 1,3-5 prints first, third, forth and fifth fields (notice the different meaning of , and -).
  `cut -d '=' -f 1 data.txt`: chop each line into fields using the '=' as delimiter, then print only the first field.
- **`paste [FILE…]`**: merge lines of files. Write lines consisting of the sequentially corresponding lines from each FILE, separated by TABs, to standard output.
  `paste file1.txt file2.txt` prints each lines of file1.txt and file2.txt on the same line.
- **`grep[OPTIONS] PATTERN [FILE...]`**: searches the FILE for lines containing a match to the given PATTERN. Prints lines of FILE matching the PATTERN
  `grep ACK data.dat`: prints only lines where ACK is present in data.dat file

`-v`: Invert the sense of matching, to select non-matching lines.

- **`tr[OPTION]... SET1 [SET2]`**: Translate, squeeze, and/or delete characters from standard input, writing to standard output
  `tr 'c' 'C'`: replace every occurrence of `c` with a `C`
  `-s`: replace each input sequence of a repeated character that is listed in SET1 with a single occurrence of that character
  `tr -s ' '`: replace multiple spaces with a single space
- **`seq OPTION]... FIRST INCREMENT LAST`**: Print numbers from `FIRST` to `LAST`, in steps of `INCREMENT`. If `FIRST` or `INCREMENT` is omitted, it defaults to 1. Numbers are printed one per line or separated by space.
  **`seq 10 :`** prints number from 1 to 10 – i.e., 1 2 3 4 5 6 7 8 9 10
  **`seq 5 10 :`** prints number from 5 to 10 – i.e., 5 6 7 8 9 10
  **`seq 1 3 10 :`** print numbers from 1 to 10 with increments of 3– i.e., 1 4 7 10

## Bash Pipes

In Bash, it is possible to concatenate commands creating a pipeline, i.e., the ouput of the previous command is used as input of the following one. The pipeline is a sequence of one or more commands separated by the control operator '|'. The format for a pipeline is:

```
command | command2 …
```

The standard output of `command` is connected via a pipe to the standard input of `command2`. Similarly, it is possible to redirect the output of a command to a file using '> OUTFILE'. If `OUTFILE` already exists, it will be overwritten. To append the results to a file which is already existing simply use '>> OUTFILE'. If outfile does not exists, it will be created. If it exists, the output will be appended to it.

For example

```
grep ACK data.dat | tr -s ' ' | cut -d ' ' -f 2,4 > results.txt
```

would look for rows in the data.dat file that contains the `ACK` pattern. For each matching line, multiple occurrences of the space character will be replaced with a single space. Then, each line is split into fields by using the space as delimiter, and only the second field will be printed. The result is saved on the `results.txt` file.

Assuming the file data.dat looks like

```
1   pippo    pluto paperino,nonna papera
2     qui        quo,qua    ACK
3 zio paperone ACK pluto      topolino
```

The output of the above command will be save in the results.txt file as

```
qui ACK
zio ACK
```

In the following, you find a sample script that can be used as example and must be properly modified to extract useful values from txt files exported from wireshark.

```
# simple script that extracts useful information from a wireshark exported txt file
# WARNING : THIS HAS TO BE CUSTOMIZED AND VERIFIED!

# extract the time information
# I need to squeeze multiple spaces, then cut on space and get the time column
# I use grep to avoid the first line.
# from the grep man page:
# -v Invert the sense of matching, to select non-matching lines.
# i.e., do NOT print lines that match ...

cat client.data.txt | grep -v Source | tr -s ' ' | cut -d' ' -f 3 > client.time.txt

# extract the seqno
cat client.data.txt | grep -v Source | cut -d'=' -f 2 | cut -d' ' -f 1 > client.seqno.txt
# extract the ackno
cat client.data.txt | grep -v Source | cut -d'=' -f 3 | cut -d' ' -f 1 > client.ackno.txt
# extract the rwin
cat client.data.txt | grep -v Source | cut -d'=' -f 4 | cut -d' ' -f 1 > client.rwin.txt

# paste all columns together
paste client.time.txt client.seqno.txt client.ackno.txt client.rwin.txt > client.dat

#plot it using gnuplot
gnuplot plot.gnu
```

Extend the previous script to generate all data that are needed to produce the required plots, for client and server traffic. Suggestion: check the output of the pipeline piece by piece to verify that there are no errors.

## Variables and loops

In bash, you can use **variables** as in any programming languages. There are **no data types**. A variable in bash can contain a number, a character, a string of characters. You have no need to declare a variable, just assigning a value to its reference will create it. A variable is simply created by assigning it. Prepending a $ sign allows to access the variable value.

```
myVar=”Hello Word”
echo $myVar                    # Prints Hello word
echo $doesNotExist             # Prints an empty line
```

A **‘for loop’** is a bash programming language statement which allows code to be repeatedly executed. A for loop is classified as an iteration statement i.e. it is the repetition of a process within a bash script. For example, you can run UNIX command or task 5 times or read and process list of files using a for loop.

The **for** loop is a little bit different from other programming languages. Basically, it let's you **iterate over a series of 'words'** within a string. The set of instructions to be executed for each item is included in the `do` – `done` keywords (these are the equivalent of a { } in C for instance).

```
for word in myVar; do
   echo $word
done
```

prints each word in a separate line.

**C-like for** can be obtained using the `seq` command.

```
for i in `seq 1 10`; do
   echo $i
done
```

prints each number from 1 to 10 in a separate line.

Note the `` `seq 1 10` `` uses the special **backtick** syntax – or **command substitution**. **A backtick is not a quotation sign.** It has a very special meaning. Everything you type between backticks is evaluated (executed) by the shell before the main command (like `seq` in our examples), and the *output* of that execution is used by that command, just as if you'd type that output at that place in the command line. That can be used to initialize variable too. For instance, the following code is equivalent to the one above:

```
myVar=`seq 1 10`
for i in $myVar; do
   echo $i
done
```

An alternate form of the for command is also supported:

```
for (( expr1 ; expr2 ; expr3 )); do
   commands
done
```

First, the arithmetic expression *expr1* is evaluated. The arithmetic expression *expr2* is then evaluated repeatedly until it evaluates to zero. Each time *expr2* evaluates to a non-zero value, *commands* are executed and the arithmetic expression *expr3* is evaluated. So you can write

```
for (( i=1 ; i<=1; i++)); do
   echo $i
done
```

## Plotting with Gnuplot

Gnuplot is a powerful tool to generate professional looking plots. It is particularly interesting for our scope since it allows to easily plot data sequences from files. Gnuplot is open source and is available for many platforms (see http://www.gnuplot.info for more details).
In this section, you can find some quick examples that can be useful to produce plots to be included in the laboratory report. For detailed instructions, and advanced capabilities offered by gnuplot, you can refer to some tutorials that can be found on the internet, on gnuplot manual (available on the CD itself), and the inline help pages that can be invoked within gnuplot itself by the help command.
Gnuplot can be used both typing command in an interactive manner, or by writing the description of the plot in a file, then tell gnuplot to load it and produce the plot. Considering plotting data from files, any file in columnar format can be used to produce plot via the following command, in which points (X,Y) are plotted and connected by lines:

```
plot ”file.dat” using X:Y title ”title” with linespoint
file.dat is the name of the file to be read as input.
```

X,Y tells which column has to be used for values of the X and Y variable. For example, using 2:3 plot the point found in the second column as X value, while values in the third column will be used as Y value.
with linespoint: specifies that the data have to be represented using points joined by lines. See help plot with for all possibilities.
Multiple curves can be plotted on the same plot by separating them with a comma

```
plot sin(x), ”file.dat” using X:Y title ”title” with linespoint
```

Some useful commands:

```
set xlabel “text” : set the text label on the X axis to “text”
set ylabel “text” : set the text label on the Y axis to “text”
```

To produce a png file, simply choose the png terminal, and direct the output to a file

```
set terminal png
set output “file.png”
```

Given this, it should be straight forward to understand what plot will be produced by the following script:

```
# Simple gnuplot file
# lines starting with # are ignored...

# set the label on the x and y axis
set xlabel "time [s]"
set ylabel "bytes"

# plot the data from the "data.txt" file
# using values appearing on the X column as x values
# and values appearing on the Y as column y values
# X and Y have to be choosen properly...

plot "data.txt" using X:Y title "title" with linespoint

# if you want to plot several datasets/lines, simply separate them using
a ',' (comma)

#plot "data.txt" using X:Y title "first line" with linespoint, "data2.txt"
using X2:Y2 title "second line" with linespoint

pause 10 # to see the plot on the screen for 10 seconds

# save the same on plot a png file to be imported later on

set term png
set output "plot.png"

replot # to simply reproduce the same plot as before…
```

## How to plot a probability density function using gnuplot

[most of this comes from http://psy.swansea.ac.uk/staff/carter/gnuplot/gnuplot_frequency.htm ]

The gnuplot commands to plot this are shown below. The principle of operation is to count the number of values that fall into specific ranges (or bins). The first thing we need to do is group the numbers into collections of similar value; this is done by the functions `rounded()` and `bin_number()`.

`rounded(x)` takes the `x` parameter and rounds it to its nearest multiple of `bin_width`. This way, the fixed values returned can be counted into bins. However, the actual value returned by `rounded()` is first shifted up or down by an amount equal to the `bin_width` multiplied by the offset. Typically, the offset will be 0.5 since this would position the bar at the centre of its range on the X axis.

The `bin_number()` function uses `bin_width` to compute how many multiples the `X` parameter is of the range (and, therefore, the particular bin into which the `X` parameter should be counted. `bin_number()` does this by converting values into grouped values (the bins). For instance, with `bin_width`=0.1, values between 0.0 and 0.09 will be rounded down to 0, 0.10 ... 0.19 will be rounded down to 0.1, 0.20 to 0.29 will be 0.20, and so on.

Once we have arranged the disparate values of the data file into a selection of predefined values, we can then count the instances of those values, which we then plot. Since we don't want to plot the actual `X` values, but the number of them in certain bins, instead of the typical using command, we use gnuplot's `smooth` command, with the `frequency` option:

Note that the plot command includes brackets around the using values. The brackets tell gnuplot to interpret these values as numerical expressions, rather than column numbers (although a dollar followed by a number explicitly references a column when inside brackets). Thus, `(rounded($1))` means "pass the value in column one to the `rounded()` function, and use the value returned as the X-axis coordinate. And (1) means to increment the bin value (i.e. Y) by one for each corresponding X value.

```
# Each bar is half the (visual) width of its x-range.
bin_width = 0.1; set boxwidth bin_width/2 absolute
set style fill solid 1.0 noborder
bin_number(x) = floor(x/bin_width)
rounded(x) = bin_width * ( bin_number(x) + 0.5 )
plot "Data.dat" using (rounded($1)):(1) smooth frequency with
boxes
```

The `smooth frequency` option makes the data monotonic in x; points with the same x-value are replaced by a single point having the **summed y-values**.

## How to plot a Cumulative Density Function using gnuplot

If you are interested into the CDF of a dataset, then you can use the `smooth cumulative` capability of gnuplot. The `cumulative` option makes the data monotonic in x; points with the same x-value are replaced by a single point containing the **cumulative sum of y-values** of all data points with lower x-values (i.e. to the left of the current data point).

```
# Count the number of elements in the dataset.
# Use the stats helper for this
stats(Data.dat)
N= floor(STATS_records)
plot "Data.dat" using ($1):(1/N) smooth cumulative with boxes
```

For more information, you could refer to the smooth.dem tutorial available at http://www.gnuplot.info/demo/smooth.html

## How to make a heatmap with gnuplot

Assume you have a data file `data.dat` which is a matrix, in which each cell represents the value to plot, i.e., *z=f(x,y)*. The variable *z* depends on two other variables *x* and *y*. A convenient way to plot this it to use a heatmap, to show the effects rather than plotting multiple lines on a regular line-graph.

Assume the file `data.dat` has a a tuple (x,y,z) in each row. Then the `plot.gnu` file below would do the magic.

Data.dat

```
# x y z
  0 0 5
  0 1 4
  0 2 3
  0 3 1
  0 4 0

  1 0 2
  1 1 2
  1 2 0
  1 3 0
  1 4 1

  2 0 0
  2 1 0
  2 2 0
  2 3 1
  2 4 0

  3 0 0
  3 1 0
  3 2 0
  3 3 2
  3 4 3

  4 0 0
  4 1 1
  4 2 2
  4 3 4
```

Plot.gnu

```
#set the terminal as we like

set term png size 1024,768
set out "heatmap.png"

# set the viewing angle for a map
set view map

# eventually choose the interpolation
# increase the values for higher interpolation
# 1,1 for no interpolation, 0,0 for max interpolation
set pm3d interpolate 0,0

# do the heatmap
splot "data2.dat" using 1:2:3 with pm3d
```