ELEC-E7130 Network capture tutorial

Markus Peuhkuri Tran Thien Thi

This documents provides you some helpful instructions capturing packets, tools to analyse them and to construct flow data out of them. Finally there will discussion about data structures that may be needed for large data.

Packet capture

Capture tools

The first task for a network data analysis is to capture network packets. Refer the first exercise for usage of **tcpdump** and **dumpcap**.

When networks are using the shared medium like Ethernet or WLAN, everyone is able to capture the traffic by just putting the network card into promiscuous mode and starts listening to the link. In normal operations NICs will filters out all the packets not destined to their own MAC address except the broadcast or any multicast messages (if the host has joined any multicast group) but in *promiscuous* mode, all packets are accepted and will delivered to operating system.

These days almost all local area networks (LAN) use switches with point-to-point links where each NIC is directly connected to a separate switch port and thus a computer connected to wired network can only see its own, multicast, and broadcast traffic mostly. Within certain situation, the switches may also broadcast other packets.

There are several solutions to capture traffic from switched network links such as link taps or more advanced method of port monitoring and mirroring (where switch will replicate a copy of each frame to the mirrored link). Capturing network traffic usually requires administrative permissions from root or Administrator. Linux systems can be configured to allow any user in `wireshark` group to be able to capture network traffic. It is actually recommended **not to run Wireshark as a root** but as a user who is in wireshark group.

Before capturing packets, you need to specify one interface which you will be using for sniffing packets. Network interface is basically the thing that connects your device to the Internet, i.e., enables Internet connections. If you are unsure about your currently available interfaces, check them easily with `ifconfig` (old

school) or `ip link` (modern) command. Choose the interface that connects to the Internet (i.e., not loopback).

You can also use `dumpcap -D` option to see interfaces your present user is authorized to capture from.

For example following command has been used to capture traffic from eth0 interface using **dumpcap** tool for duration of one hour and saving first 96 bytes from each packet into in `capture.pcap` file:

```
dumpcap -a duration:3600 -i eth0 -s 96 ´-P -w capture.pcap
```

In addition to a single file, both **tcpdump** and **dumpcap** do support capturing into multiple files in rotation. Check tcpdump options `-C`, `-G`, and `-W`; dumppcap option `-b`.

Packet data analysis

Wireshark, **tshark** and **tcpdump** are good tools to make interactive analysis and quick checks. See *Reasonably short introduction to using Linux/Unix and network measurement tools* for details. Also check Extracting packet data below.

Sustained analysis of terabytes of data does need tools optimised for this usage. There are many tools originating from research or information security communities available for massive data analysis while no single analysis tool can answer all the requirements imposed by different scenarios.

Following tools are tested and believed to be useful tools for data analysis. These are available either as standard tools in Aalto Linux environment or installed in course directory. You can have them into use by sourcing familar `use.sh` script.

Table 1: Mass analysis tools

Software	GUI/CLI	Description
CoralReef	CLI	A set of tools (more important tools in the set are `crl_flow`) for flow based analysis. However, CoralReef is not maintained anymore and not all features work on recent Linux distributions. Documentation
NetMate	CLI	Flow based analysis. Outputs more features than CoralReef which can be helpful for ML classification purposes. Documentation
TStat	CLI	Stateful analysis tool. Produces TCP and UDP connection data and also histograms. Documentation
tcptrace	CLI	Provides TCP connection analysis. Documentation
tcpstat	CLI	Provides statistics for each intervals. Documentation

Software	GUI/CLI	Description
Wireshark	GUI	Industry standard for inspecting packet captures. Includes protocol-specific decoding and analysis tools, e.g. for VoIP calls and other RTP traffic.
TShark	CLI	Command line version of Wireshark. Can be used from scripts to analyse captures.
Capinfos	CLI	A program tool that reads one or more capture files (supported by Wireshark) and returns some or all available statistics (info) in one of two types of output formats: long or table. Documentation
tcpslice	CLI	Allows combining multiple pcap files and extracting specified time frame from those.

Next we will take a look at their basic usage. Let's assume that the captured file is named as `capture.pcap`.

NetMate

NetMate is another CoralReef-like software for producing flow information from the captured packets. For each flows, it will be able to produce 44 different features.

NetMate is installed on Aalto computers in the course folder. Source the `use.sh` script and you are able to use it. You need to run in such directory where you have write access. It will overwrite any `netmate.pid` and `netmate.log` files in that directory.

> If you want to install NetMate by yourself to your Linux system, first you need to git clone https://github.com/danielarndt/netmate-flowcalc and then you can follow installation instructions[1]. Some of the libraries need to be installed for the successful installation. By default, it will install to `/usr/local` if you install it with root privileges. You can install it also into another location but you need to modify `PATH` environment variable.

The NetMate is configured with rule file. Basic example can be found from `/work/courses/unix/T/ELEC/E7130/general/netmate/` as `netAI-e7130.xml`. You **need to copy it into your directory** and modify the `Filename` preference to point into your directory. You can specify filename as `/proc/self/cwd/netmate.out` in which case it will generate `netmate.out` file into current diretory.

Then run the following command to get the flow files:

```
netmate -r netAI-e7130.xml -f capture.pcap -l capture.log
```

[1] The site has an expired TLS certificate late August 2019; follow appropriate caution.

The NetMate does not support reading from named pipes or from compressed capture files.

CoralReef

CoralReef is a software to produce flow level information from the captured packets. Flow is a group of packets with the similar properties, especially source IP, destination IP, source port, destination port, and protocol. When determining some traffic statistics, analyzing flows can be better idea than analyzing packets.

The next commands can generate flow information from `capture.pcap` file:

1. To convert pcap file into flows generating multiple files:

   ```
   crl_flow -Ci=3600 -cl -Tf60 -O %i.t2 -Cai=1 capture.pcap
   ```

   Where some of the options are as follows:

   - Period defined is one hour `-Ci=3600`, this affects to the total number of output files
   - Counters cover whole lifetime of flow `-cl`
   - Flows expire after 60-second inactivity `-Tf60`, this affects to the total number of flow instances
   - Output to sequentially numbered files `-O %i.t2`, just for formatting file names
   - Intervals aligned round intervals `-Cai=1`. In this case interval starts at each hour wall clock time.

2. To convert pcap file into flows generating only one file:

   ```
   crl_flow -I -Ci=172800 -cl -Tf60 -o output-all-ended.t2 -Cai=1 $TRACE/capture/flow.pcap
   ```

   Where the option `-o` *`outfile`* specifies the (non-rotating) output file to write to (default: stdout), in other words, store the flow data in only one file.

Created file above (`1.t2` or `output-all-ended.t2`) can be then sorted e.g. by bytes (`-Sb`) to show only the 10 largest flows (`-n 10`):

```
t2_top -Sb -n 10 < 1.t2
```

Tstat

The `tstat` is designed to be run over long period of times. It has also large set of options. At the minimum, the network and histogram options should be defined.

- `-N net.conf`

  This defines "internal" network. If capture is done in single machine, there should be just one file containing host IP address with 32-bit prefix like: `192.0.2.5/32`.

- `-H histo.conf`

  This defines included histograms. If you want all, the file contains just line `include ALL`.

Example:

```
tstat -N net.conf -H histo.conf capture.pcap
```

Above command will read trace from file `capture.pcap` and outputs results to file `capture.pcap.out` directory. You can specify output directory with `-s outdir` option. You can supply also multiple trace files, but those must be in chronological order.

This output directory consists lots of trace data which could be used for, e.g., plotting some useful graphs. In order to plot such graphs, you need to download Perl scripts `plot_cum.pl` and `plot_time.pl` from http://tstat.polito.it/viewvc/software/tstat/trunk/scripts/ . After you have downloaded them, give them execution rights with `chmod +x [SCRIPT]` and run them on the output directory. You might need to install some required Perl modules (such as `Date::Manip`) which will require superuser privileges.

Tcptrace

Tcptrace is for analysing TCP connections from the captured file. When given some specific options, tcptrace can also output some `.xpl` files that could be used for shell plotting with `xplot` command.

By default it will print one line per TCP connection listing number of segments in each direction. Output can be modified with options, some of importat options include:

- `-n`: do not resolve host names (runs much faster)
- `-o5`: print only connection 5 (can also be range `-o5-14`)
- `-l`: prints long output for each connection.
- `--csv`: provide long output in CSV format.
- `-r` provide RTT statistics for each connection (analysis is slower).

Running tcptrace on the captured file to produce long statistics to a CSV file:

```
tcptrace -n -l --csv capture.pcap > capture-tcp.csv
```

Tcpstat

Command `tcpstat` can be used to check interface or read captured packet files to display some statistics for each interval. Interval will be 5 seconds if not specified further.

```
tcpstat -r capture.pcap
```

By default, `tcpstat` will output timestamp, number of passed packets, average size of packets, standard deviation of packets, and bandwidth as bits per second (from left to right). Furthermore, `tcpstat` can be configured to produce different kind of output using format string provided by `-o` option. For example, `%S\t%p\t%b\n` will print a line for each interval including UNIX timestamp, packets per second and bits per second over that interval.

It also supports BPF to include only a part of traffic in statistics.

Tcpslice

Another useful tool is **tcpslice** that will combine number of PCAP files optionally extracting only range of packets. For example, if have large number of PCAP files (long capture that has been split by hour for example) in `cap` diretory and you want to extract all ICMP traffic for a week starting from 15th September 2019. The results will be saved to `icmp15+7.pcap` file.

```
tcpslice -w - 2019y9m15d +7d cap/* | tcpdump -r - -w icmp15+7.pcap icmp
```

Extracting packet data

In many cases you need to make analysis on individual packets. With **tcpdump** you can parse textual output with regular expressions. You can control tcpdump output using following options:

- `-n`: do not resolve names. If you want to work with IP addresses, you do not want them to be resolved. Also speeds processing significantly.
- `-tt`: print timestamp as seconds from epoch. Other number of `t`s modify output.
- `-q`: quick/quiet output prints less than default
- `-v`: verbose output. Use two or three `v` for even more verbose output. Verbose output results multi-line output where the first line includes timestamp and continuation lines are indented with spaces.

Let's consider for example that you want to analyse delays and packet loss from ICMP echo messages. By running `/usr/sbin/tcpdump -ntt -r capture.pcap icmp` we can extract all ICMP messages from the capture. The output would be something like:

```
reading from file tmp/test.pcap, link-type EN10MB (Ethernet)
1569215008.517695 IP 192.0.2.155 > 8.8.8.8: ICMP echo request, id 24357, seq 1, length 64
1569215008.541868 IP 8.8.8.8 > 192.0.2.155: ICMP echo reply, id 24357, seq 1, length 64
```

From the output we can see it includes all required fields, no need for `-v` option. To parse each line, we can use again regular expressions. With following line we capture all fields:

```
^(\d+\.\d+) IP ([0-9.]+) > ([0-9.]+): ICMP echo (request|reply), id (\d+), seq (\d+), length
```

We can then use it in perl script:

```
#!/usr/bin/perl -w
use strict;

while(<>) {
  if (m/^(\d+\.\d+) IP ([0-9.]+) > ([0-9.]+): ICMP echo (request|reply), id (\d+), seq (\d+)
    my @res = @{^CAPTURE};  # same as ($1, $2, $3, ...)
    if ($res[3] eq "request") {
      $res[3] = 0;
    } else {            # guaranteed to be reply
      $res[3] = 8;
    }
    print join(",", @res), "\n";
  }
}
```

or equivalent python:

```
#!/usr/bin/env python3


import sys
import re

if (len(sys.argv)<2):
  print("Give input file name (/dev/stdin to read standard input)")
  sys.exit()

r=re.compile('(\d+\.\d+) IP ([0-9.]+) > ([0-9.]+): ICMP echo (request|reply), id (\d+), seq

for line in open(sys.argv[1]):
  m=r.match(line)
  if m is None:
    continue
  if m[4] == 'request':
    rq = '0'
  elif m[4] == 'reply':
    rq = '8'
  print(','.join([m[1], m[2], m[3], rq, m[5], m[6], m[7]]))
```

Pipe tcpdump output to a script and direct script output to a CSV file that can then be further analysed with suitable tools.

```
/usr/sbin/tcpdump -n -r capture.pcap icmp | ./icmp2csv.pl > echos.csv
```

While tcpdump do have the benefit of being readily available with moderate memory requirements, parsing textual output can be cumbersome. Converting data into textual format is also slow. There exists PCAP library interfaces for

perl (`Net::Pcap` and `Net::Pcap::Easy`) that provide helper functions to decode basic protocols with one packet a time. For python there exists PyPCAP and Scapy of which the latter will read files as whole.

The other alternative would be to use **tshark**. It can decode packet data and output desired information as CSV and JSON formats. The drawback being that it will consume large amounts of memory for large captures. This can be helped with pre-filtering data with tcpdump, for example.

```
#!/bin/sh
export LANG=C
tshark -n -r ${1:-capture.pcap} -Y icmp.code==0 \
  -T fields -E header=y -E separator=, -E occurrence=f \
  -e frame.time_epoch -e ip.src -e ip.dst -e icmp.type \
  -e icmp.ident -e icmp.seq -e ip.len -e ip.hdr_len -e icmp.resptime
```

The output above script produced was not exactly identical but had two minor differences.

1. ICMP packet length was not printed as a single value but as two values: IP packet length (`ip.len`) and IP packet header length (`ip.hdr_len` in bytes). The length would be `ip.len - ip.hdr_len`. You can find all variables from Display Filter Reference. Quite often it is a good idea to test variables with Wireshark as it provides easy help and suggests extensions.
2. Another difference is that we got also the response time without calculating it. The tshark runs multiple analysis on data and results can be extracted. Also check `-z` option that will run additional statistical modules, for example `-z icmp,srt`. The same statistical modules are also available in Wireshark, so you can try out them easily.

And yes, it also had a header line. In addition to CSV, you can output in different formats, including JSON. Some formats produce huge amounts of data, test first with small captures. You can control JSON output with `-j` and `-J` options. Check wireshark documentation for details.

Network interface statistics

Operating systems also provide statistics of network traffic (By default Windows provide only small amount of information about network traffic, counters and more advanced configurations are needed to extract those information from a Windows machine). By checking interface properties there are information of sent packets, bytes or both. In Linux systems, for example, `ip` reports following information:

```
ip -s link show dev wlp2s0
3: wlp2s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DORMANT group d
    link/ether e4:b3:de:ad:be:ef brd ff:ff:ff:ff:ff:ff
    RX: bytes  packets  errors  dropped overrun mcast
```

```
1773715758 1758985  0       0       0       0
TX: bytes  packets  errors  dropped carrier collsns
251208218  1175505  0       0       0       0
```

In other Unix-like systems and older Linux systems there has been `ifconfig` and `netstat` commands but but those are depracticed on Linux. Most ot their functionality is replaced with `ip` and `ss` commands. Many statistics can be found under `/proc/net/` directory path..

Large data analysis

There are cases where a simple approach to store information is not possible. In networking, data structures can be quite sparse. If one wants to count how many times each IPv4 addresses exists as a source address, even this simple can take lot of memory. Using 4 bytes per packet, it will use $4*2^{32}$ bytes, about 17.2 GB. While this is possible with modern computer, there will be only a fraction of those values in trace.[2] With IPv6 and 128 bit addresses, this is not possible.

Following we will cover some examples how you could encode values to fit into minimal address space. If you would use python or perl, using dictionaries (`dict()`) or hash (`%var`) will be often sufficient.

This can be helpful, since in some future tasks, you will be dealing with pairs of things. The things have different values but the similar pairs need to be dealt accordingly. For instance, src-dst pair A-B is same as pair B-A, so when we are calculating total amount of values for pair A-B, values from both pairs need to be summed. With big amount of dataset, dealing with each pairs separately can be time consuming, so one way to speed up the process is to encode similar pairs into one.

Next examples are given using Perl, but as usual, you can use any programing or scripting language or tool as long as the results are clear.

Let us next consider a measurement of relatively large number of flows. Example trace file is in `sample-flows.txt`, which consists of about 2M flows observed in one hour time interval. Each line in the trace file has the following information:

- **SRC**: Source address (64-bit, hexadecimal number)
- **DST**: Destination address (64-bit, hexadecimal number)
- **PKT**: Number of packets exchanged

The source address is the one initiate the connection and send the first packet. Task is to determine the most active pairs of hosts, i.e., the pair that has most number of flows exchanged in total.

[2] At one gigabit per second link, it will take about 36 minutes to send minimum sized packet using all possible source addresses.

Preliminaries

A bitwise XOR operation is denoted by $\oplus$.

$$1011 \oplus 0110 = 1101 \tag{1}$$

Hash function $h(x)$ in general maps the variable x to an integer number in some finite interval. Here we settle with simple hash functions based on XOR operation. For example, for 32-bit IPv4 addresses of form $a.b.c.d$ we could use $(a|b \oplus c|d)$ to get a 16-bit integer number. This is like a lightweight secure hash algorithm, i.e., the basic purpose of this XOR encoding is to ensure that two similar pairs produce same encoded result. And two different pairs will not likely produce same encoded result (but there is still small probability).

Solution 1: straightforward data structure (not using hash function)

The first solution to solve this problem is using the already existing hash function in library of Perl:[3]

```
#!/usr/bin/perl
my %F; # hash for flow counts

# read flows in and store to hash %F

while(<>) {
  my ($src,$dst,$pkt) = split(' ',$_);
  my $key = ($src lt $dst ? $src.' '.$dst : $dst.' '.$src);
  $F{$key}++;
}

# sort hash and print the counts
foreach my $i (sort { $F{$b} <=> $F{$a} } (keys %F)) {
  printf "%s %dn", $i, $F{$i};
}
```

The first part reads the data file one line at a time, forms an unique string by combining the source and destination address in unambiguous way, and then increases the value of hash at the corresponding memory location by one. Once the file has been parsed, we sort the obtained values and print the result.

However, there is one potential issue with this approach: if the number of SRC/DST pairs is huge, the size of the hash can become too large (in theory, the size of the key space is 2^{255}, which is an enormous number!).

[3] Source codes can be found from `sampling-code.zip` archive.

Solution 2: using hash function

In order to save memory, we can aggregate the flows. If we do this randomly using an appropriate hash function, there is a high chance that the most common pairs of hosts do not get combined to same entries.

Example:

```
#!/usr/bin/perl

use strict;
my @A; # fixed array for flow counts, size 65536

sub simple_hash {
  my $a = shift; # 64-bit address
  my ($h,$l)=split(':',$a);

  return hex($h)^hex($l);
}

# read flows in and store to array @A
while(<>) {
  my ($src,$dst,$pkt) = split(' ',$_);
  my $key = simple_hash($src)^simple_hash($dst);

  $key = ($key & 0xFFFF) ^ ($key << 16); # 16-bit value

  $A[$key]++;
}

# sort hash and print the counts

foreach my $i (sort { $b <=> $a } @A) {
  printf "%dn", $i;
}
```

Here we utilize a simple hash function that takes XOR of all 4 32-bit numbers forming the address pair:

$$x = src.high \oplus src.low \oplus dst.high \oplus dst.low \tag{2}$$

Then to further reduce the memory consumption, the 32-bit number is compressed to half by:

$$x = x.high \oplus x.low \tag{3}$$

Hence, $0 \leq x2^{16}$ then the corresponding entry in the fixed size array A (with size 2^{16}) is increased by one:

$$A[x] = A[x] + 1 \tag{4}$$

The final part is to sort array A and print out the results. The memory consumption of this approach is clearly small. Also the sort operation is fast when the size of the array is moderate as in here. The downside is that different flows unavoidably do get aggregated, and therefore the results are only approximation of the real one.

Solution 3: using command line tools

For small enough trace files, we can also utilize the standard Unix tools including "sed", "awk" and pipes as follows:

```
cat sample-flows.txt | awk '$1<$2{print $1,$2;next}{print $2,$1;}' |
sort | uniq -c | sort -r -n > sample-flows.cnt
```

Let's now describe the command above. First, we read whole file contents with `cat` command. Then we use `awk` text processing command to only deal with first two columns and "encode" them. Next we use `sort` command to put similar pairs nexts to each other. Then we use `uniq -c` to remove duplicate rows and also to display their total amount of counts. Finally, we sort according to the counts and output whole thing to another file. So, the desired result is in file `sample-flows.cnt` and the most active flows are at the head of the file.

Measuring resource usage

Operating systems collect quite lot of resource consumption information from each process. Easy way to know how much a program has been consumed resources, is to use `time` command on Linux. It will print out how much CPU time the process has used in user space (i.e. the program code) and how much in system space (kernel, e.g. networking, file processing).

The shell built-in `time` will provide this basic information. If you want more details from e.g. memory usage, an alternative is to use `/bin/time` command to get more detailed resource consumption of a command, use `-v` for more verbose.

Another difference is that built-in `time` will report all CPU usage of pipeline; `time grep -v ERR * | sort -o output` will include both `grep` and `sort` resource usage while `/bin/time grep -v ERR * | sort -o output` will only report what was consumed by `grep`.

Useful tools or tips for completing this final assignment

Pre-installed software in school

Some of the tools such as *Coral Reef* or *TStat* which might help you for this final assignment are already provided and installed in Aalto Linux workstations. They are accessible after using one these commands:

1. `source /work/courses/unix/T/ELEC/E7130/general/use.sh`
2. `source /work/courses/unix/T/ELEC/E7130/general/use.csh`

As you already know first command is used if you using any Bourne Shell compatible (like *bash* or *zsh*) and second one is used when you are using C Shell compatibles shells.

CoralReef software package provides several tools which can be useful in analyzing the captured trace files including:

- `crl_to_pcap`: for converting and anonymizing packet traces (in case you want to do it for your own captured data)
- `crl_flow`: for summarising packet data to flows

For example, flow files used in Final Assignment Task 1 was generated with the following command:

```
crl_to_pcap -r "[" pcap/*.pcap "]" |
crl_flow -Ci=3600 -Cai=1 -Tf60 -O flow-continue/%V-%u-%H%M.t2 pcap:-
```

In command above, first the captured packets were anonymized and then they were converted into flows. Output file name is generated with "strftime" formatting. Name components are week number, day of week and time. Using reverse name lookups, *whois* databases or *geoip* databases will lead to random results.

And files of tstat for Task 1 were generated using command below.

```
tstat -H histo-all.conf -N net.ten -s tstat-log -c ../pcap/*
```

The `histo-all.conf` content is `include ALL` which results it to produce all histograms.

Finding geographical locations based on IP address

To convert IP addresses to (approximate) country listings one can use geoiplookup command or python GeoIP library. For the latter simple example is below program `geoip.py` that prints all IP addressed given as command line arguments to stdout with two letter country code (or None if not known).

```
#!/usr/bin/env python3
import sys
```

```
import GeoIP
gi = GeoIP.new(GeoIP.GEOIP_MEMORY_CACHE)
for n in sys.argv[1:]:
  print("%s %s" % (n, gi.country_code_by_addr(n)))
```

For Ubuntu the GeoIP python library can be installed from `python3-geoip` package. Example usage of the program when using on Google's popular name-server. Remember to `chmod +x geoip.py` before running the command.

```
./geoip.py 8.8.8.8
8.8.8.8 US
```

TCP connection stats

In some tasks you will be dealing with TCP connections only. Use `tcptrace` for handling such situations.

Traffic volume in certain interval

From pcap files it is easy to generate traffic volume by second - just calculate sum of packet sizes captured within a second. Tool like tcpstat can be used for that.

From flow data files (outputted by CoralReef) getting traffic volume is more difficult because only start time, end time and flow size are known. Reasonable approximation is distribute whole flow volume over lifetime of flow. So if flow 100 000 byte size flow starts at 3.4 and ends at 7.8 we can account 20 000 bytes (160 kbit) for each second 3-7.

If flow is has duration of 0 seconds (i.e. just one packet) it would be counted only towards that second it was calculated. A function `update()` in a program below would be called for each flow record with following arguments:

- `table` an array large enough to hold bw information for each second. If measurement period is one hour, it must have 3600 elements.
- `t0` is start timestamp of measurements
- `start` and `end`are flow start and end times respectively
- `bytes` count of bytes in flow.

```
import sys

def update(table, t0, start, end, bytes):
  # compute how many bits per second there are for each second
  bps=8 * bytes / (int(end)-int(start) + 1)
  try:
    for sec in range(int(start), int(end) + 1):
      table[sec-int(t0)] += bps
  except IndexError:
    print('Seconds was not within table bounds (0-%d):\n sec=%d\n  t0=%d'%(len(table), sec,
```

```
    raise
  return

# Simple manual test function to play around
# In reality would read flow records from a file in a loop

def test():
  # minimum timestamp (here 1970-01-01T00:01:40 but you would set it to start of capture
  t0=100
  # our analysis only 20 seconds long
  tbl=[0.0]*20
  # first flow, over 10 seconds, 600 bytes (4800 bits) per second
  update(tbl, t0, 100.2, 109.8, 6000)
  print(tbl) # print array content
  # second flow over 8 seconds, partly overlaps with one earlier
  update(tbl, t0, 108.9, 115.5, 12000)
  # a 0 s length flow
  update(tbl, t0, 119.4, 119.4, 40)
  print(tbl)
  # uncomment following lines to trigger an error
  update(tbl, t0, 118.9, 125.5, 12000)
  print(tbl)

test()
```

Further, it could be defined many more accurately by taking fractional seconds into account (3rd second would get 12636 bytes, 4th to 6th 22 727 bytes each and 7th 18 181 bytes). In table below that would be *flow1*.

Table 2: Example of distibuting flow over partial seconds

sec	1	2	3	4	5	6	7	8
flow1			.4←	—	—	—	→.8	
flow2	.1							
bytes	40	0	12636	22727	22727	22727	18181	0

If flow is has duration of 0 seconds (i.e. just one packet), one could make an artificial length of one microsecond: `td=max(1e-6, end-start)` or if one works with start and end values, `end=max(start + 1e-6, end)`. This can make computation easier as there would be no need to handle special case (`td=0`) dependig how computation is organized. In table above *flow2* would be single 40-byte packet that was observed at 1.1 seconds.