A Simple Manual for NetBench

Zhiyan Liu

1. Introduction

NetBench is a packet-level simulator aimed at evaluating various network architectures under different traffic traces. This document will provide a quick-start to the simulator, explain the modifications I have made, and provide solutions to some problems you may encounter. I suggest reading Section IV of the NetBench thesis as well. Feel free to contact me on cnrglzy@gmail.com if you have any further issues.

2. Installation

The simulator is written in Java and packed in one single .jar file. All you need is to prepare the JRE environment to execute the NetBench.jar file. Version 8 of Java is needed and both Oracle JDK and OpenJDK are supported.

I used it in Ubuntu 18.04, where Java 8 can be installed with:

```
sudo apt-get update
sudo apt-get install openjdk-8-jre
```

Refer to https://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html to install Oracle JRE 8 on Windows.

Also please make sure you can run Python from the command line, because the script for analysis is written in Python.

Optionally, to re-compile the .jar file from the source code, you will need the Apache Maven software project management and comprehension tool. Refer to https://maven.apache.org/ for the installation guide. After that, you can execute

```
mvn clean compile assembly:single
```

to compile the executable NetBench.jar.

3. Simulation Setting

A typical simulation run requires three main parts, namely the topology, the running settings and the traffic trace. I will introduce the three parts as follows.

(1) Topology

The topology is defined in a text file. All switches and servers are considered as nodes in a graph, while all links are considered as edges. For example, the following topology determines a simple topology where server node 0 and server node 2 communicate through an intermediate switch node 1.

```
# Details

|V|=3 # The number of nodes

|E|=4 # The number of edges (one bi-directional link = two edges)
```

```
ToRs=set(0,2)
Servers=set(0,2)
Switches=set(1)

# Links
0 1
1 0
1 2
2 1
```

Note that we do not specify the link capacity in the topology file, so the capacity would be homogeneous as determined in the .properties file. To implement reconfiguration, we can assign different capacity to different links. To do this, simply add an integer next to the link (the number is in Gbps). For example:

```
# Links
0 1 10
1 0 10
1 2 20
2 1 20
```

In this case there is actually no ToR. If we add the following code in the .properties file

```
scenario_topology_extend_with_servers=regular
scenario_topology_extend_servers_per_tl_node=2
```

then each ToR will be extended with two servers, yielding an extended topology as below:

```
# Extension with 2 servers/TL node of topology file
"example/topologies/simple/simple_n2.topology"

# Extended details
|V|=7
|E|=12
AutoExtended=true
Servers=incl_range(3, 6)
Switches=set(1)
ToRs=set(0,2)

# Original 4 links:
0 1 10
```

```
1 0 10
1 2 10
2 1 10


# Extended 8 links:
3 0
0 3
4 0
0 4
5 2
2 5
6 2
2 6
```

(2) Running Settings

The .properties file provides many settings, and consists of 5 parts. Most of them are self-explanatory. For some properties, if you are not sure what options are valid, you can look for source code files with the name ‘Selector’ and with the suffix ‘.java’. Some items are explained as below:

`run_time_ns (run_time_s):` The simulated time in nanoseconds (seconds).

`transport_layer`: The protocol used in transport layer. Supported options include dctcp, tcp, simple_tcp, simple_dctcp… (please refer to InfrastructureSelector.java and the 4.1 Section in the NetBench thesis)

network device & network device: Determines the routing scheme.

Supported routing scheme include equal-cost multi-path (ECMP), k-shortest-path (KSP), valiant load balancing (VLB) and their hybrid. Some routing schemes will need extra settings (such as the k value in KSP). Please refer to the examples in ./private/runs.

Since the original version does not allow specifying the capacity of each link, I cannot guarantee all routing schemes work correctly if you specify the capacity of each link. ECMP works well, but it simply ignores the capacity difference and makes the same decision as when all links are homogeneous.

`network_device_intermediary:` When using routing scheme which utilizes multi-path, how does a switch decide which port to send a packet? It calculates a hash based on (source, destination, flow id, flowlet id). Setting this option to ‘identity’ gives the same flowlet id to packets from the same flow. Splitting flows will be disallowed, which means that packets from a single flow will always go through the same route (because all 4 values of (source, destination, flow id, flowlet id) will be the same). Setting this option to ‘uniform’ will change the flowlet id over time. The time interval will be determined by FLOWLET_GAP_NS. So every FLOWLET_GAP_NS nanoseconds, packets from a single flow will be routed to a potentially different path.

`output_port:` The type of output port model. The tail drop queue accepts packet into its FIFO queue until the maximum buffer queue size is reached, at which point it simply drops packets. It marks packets with the ECN flag that arrive when the current buffer queue size exceeds the ECN threshold.

`link_bandwidth_bit_per_ns:` The global link bandwidth. It can be overridden by specify capacity in the topology file.

All the properties can be overridden when calling the executable .jar file in the command line. For example,

```
java -ea -jar test.properties run_folder_name=work
```

will override the run_folder_name in the .properties file.

(3) Traffic

The simulator provides two ways to customize traffic: (1) using an arrival list; (2) generate traffic from a traffic heatmap. To use an arrival list you might use the following code in the .properties file:

```
# Traffic

traffic=traffic_arrivals_string

traffic_arrivals_list=(0, 0, 1, 3000);(0, 0, 1, 3000);(0, 0, 1, 3000)
#(start_time, src_id, dst_id, flow_size_byte)
```

To generate traffic from a traffic heatmap, you will need a pair probability distribution. You can find some examples in ./private/data/pair_distributions. Or you can use synthesized distributions such as all-to-all and all-to-all with hotspots. You can find some examples in ./examples and ./private. The flow arrival will follow a Poisson process with parameter \lambda, namely the flow arrival rate. It can be defined by

```
traffic_lambda_flow_starts_per_s=3000
```

The simulator provides some flow size distributions generated from real applications. These options include (check ./src/main/java/ch/ethz/systems/netbench/ext/poissontraffic/flowsize and ./src/main/java/ch/ethz/systems/netbench/core/run/TrafficSelector.java)

```
traffic_flow_size_dist=pfabric_web_search_upper_bound/
pfabric_web_search_lower_bound/pfabric_data_mining_upper_bound/
pfabric_data_mining_lower_bound
```

4. Starting the simulation.

Execute `java -ea -jar test.properties` to start simulation. As mentioned above, you can add some properties in the command line such as

```
java -ea -jar test.properties run_time_ns=100000 run_folder_name=work
```

These properties will be effective whether or not they already exist in the .properties file.

5. Analyzing the results

All statistics will be saved in the specified run folder. The packet latency will be recorded in statistics.log. Remember that this data will only record packets that are not dropped. You may need

to consider the dropped packet as well when the drop rate is high. I used ‘server_port_mean_utilization’ in analysis/port_utilization.statistics as the injection rate. If you don’t see this file, make sure that Python is installed correctly.

6. Some tools

I wrote some Python scripts to generate the topology file, generate the pair probability file and do reconfigurations. If you are still using the HyperLION architecture, they may be helpful. We can talk about them in our future meetings.