



Intel[®] Simics[®] Simulator New User Training

Lab 007 – Networking and checkpoints

Copyright © Intel Corporation

Lab 007 – Networking and checkpoints

This lab covers networking in the Intel® Simics® Simulator, as well as advanced checkpointing.

A. Test networking with the service node

In the first network lab, we will try networking between a single target system and the service node. The setup you have been using until now always contained a network, with a service node as the only other machine in the network.

1. Start a simulation session with the checkpoint **AfterDriver.ckpt**, the second one you saved in lab 001:

```
[C:]> simics[.bat] AfterDriver.ckpt
```

2. The machine is on a network. To see this, print the target info in CLI:

```
simics> print-target-info
```

Which should say:

```
QSP x86 with Linux - Simics Training Setup
```

Namespace	machine
System	a QSP x86 chassis
Processors	2 QSP X86-64, 2000.0 MHz
Memory	16 GiB
Ethernet	1 of 1 connected
Storage	2 disks (208 GiB)

It says **Ethernet: 1 of 1 connected**. This indicates that the target machine believes that it is connected to a virtual network.

3. Check what top-level components you have.

```
simics> list-components
```

You see an Ethernet switch there:

Component	Class
ethernet_switch0	ethernet_switch
machine	chassis_qsp_x86
service_node_cmp0	service_node_comp

4. Inspect the switch further using its **status** command to see what is on the network.

```
simics> ethernet_switch.status
```

Note that it is connected to both the target machine and to something known as the service node. This will also print other information about the network. Check the **goal_latency** of the network link to see the configured network latency:

```
Status of ethernet_switch [class ethernet_switch]
=====

Setup:
    Top component : machine
    Instantiated  : True

Attributes:
    global_id    : none
    goal_latency  : 1e-05
    immediate_delivery : False

Connections:
    device0 : machine.mb.sb:eth_slot
    device1 : service_node:connector_link0
```

5. To see the configuration of the service code, use the **info** command on the **sn** object inside the **service_node_cmp0** component.

```
simics> service_node.sn.info
```

Note that it is configured to use the IP address **10.10.0.1**, which is the typical IP assigned to the gateway in an Ethernet network in Intel Simics Simulator models.

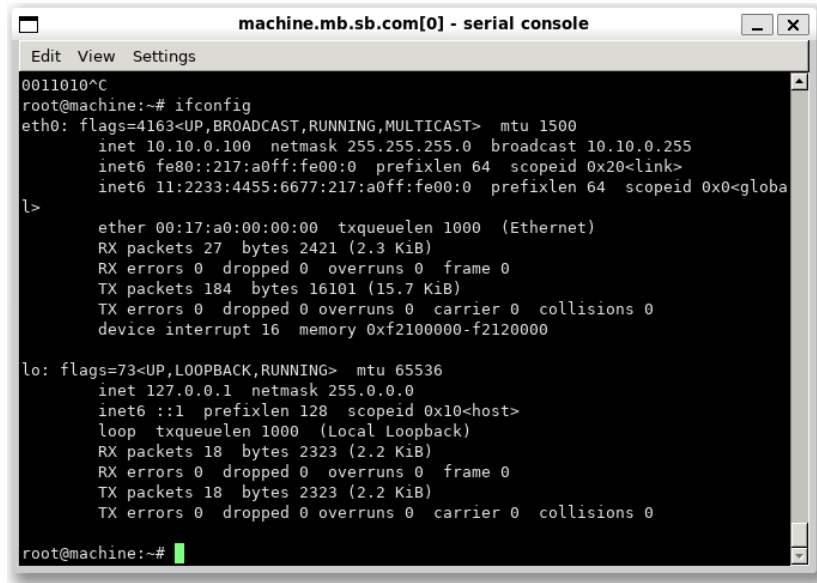
6. **Run** the simulation forward:

```
simics> r
```

7. Go to the target **Serial Console** window and use Linux commands to check the IP address that the target machine has been assigned on the virtual network inside of the simulation, using the **ifconfig** command on the target.

```
# ifconfig
```

Make a note of the IP (it is expected to be 10.10.0.100).



```
machine.mb.sb.com[0] - serial console
Edit View Settings
0011010^C
root@machine:~# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.10.0.100 netmask 255.255.255.0 broadcast 10.10.0.255
    inet6 fe80::217:a0ff:fe00:0 prefixlen 64 scopeid 0x20<link>
    inet6 11:2233:4455:6677:217:a0ff:fe00:0 prefixlen 64 scopeid 0x0<global>
    ether 00:17:a0:00:00:00 txqueuelen 1000 (Ethernet)
    RX packets 27 bytes 2421 (2.3 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 184 bytes 16101 (15.7 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
    device interrupt 16 memory 0xf2100000-f2120000

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 18 bytes 2323 (2.2 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 18 bytes 2323 (2.2 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

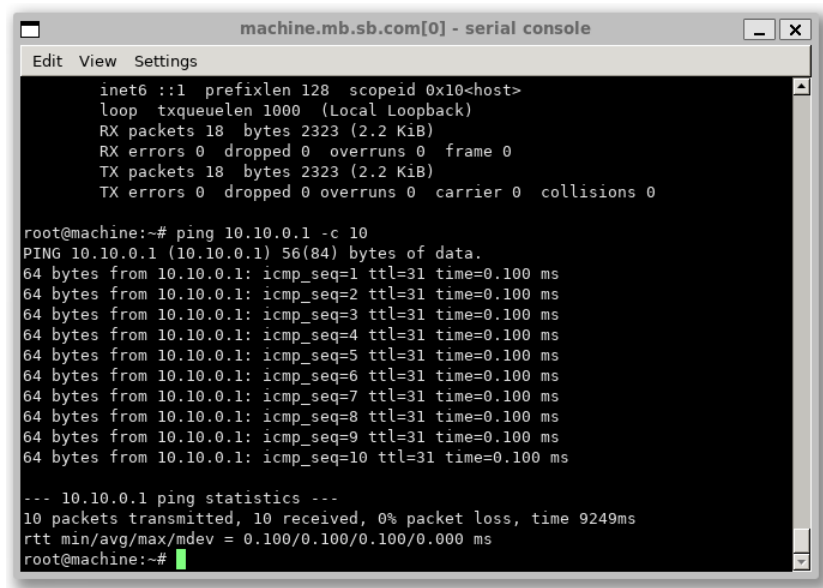
root@machine:~#
```

If no IP address is shown, just issue the command again. Sometimes, the network interface can take a while to get configured after boot.

8. Enter a **ping** command on the target command line. Target the IP **10.10.0.1**, the service node. Send 10 packages:

```
# ping 10.10.0.1 -c 10
```

The target will print 10 lines, showing that it did get replies from the service node.



```
machine.mb.sb.com[0] - serial console
Edit View Settings

inet6 ::1 prefixlen 128 scopeid 0x10<host>
loop txqueuelen 1000 (Local Loopback)
RX packets 18 bytes 2323 (2.2 KiB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 18 bytes 2323 (2.2 KiB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@machine:~# ping 10.10.0.1 -c 10
PING 10.10.0.1 (10.10.0.1) 56(84) bytes of data:
64 bytes from 10.10.0.1: icmp_seq=1 ttl=31 time=0.100 ms
64 bytes from 10.10.0.1: icmp_seq=2 ttl=31 time=0.100 ms
64 bytes from 10.10.0.1: icmp_seq=3 ttl=31 time=0.100 ms
64 bytes from 10.10.0.1: icmp_seq=4 ttl=31 time=0.100 ms
64 bytes from 10.10.0.1: icmp_seq=5 ttl=31 time=0.100 ms
64 bytes from 10.10.0.1: icmp_seq=6 ttl=31 time=0.100 ms
64 bytes from 10.10.0.1: icmp_seq=7 ttl=31 time=0.100 ms
64 bytes from 10.10.0.1: icmp_seq=8 ttl=31 time=0.100 ms
64 bytes from 10.10.0.1: icmp_seq=9 ttl=31 time=0.100 ms
64 bytes from 10.10.0.1: icmp_seq=10 ttl=31 time=0.100 ms
--- 10.10.0.1 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9249ms
rtt min/avg/max/mdev = 0.100/0.100/0.100/0.000 ms
root@machine:~#
```

Increase the network latency

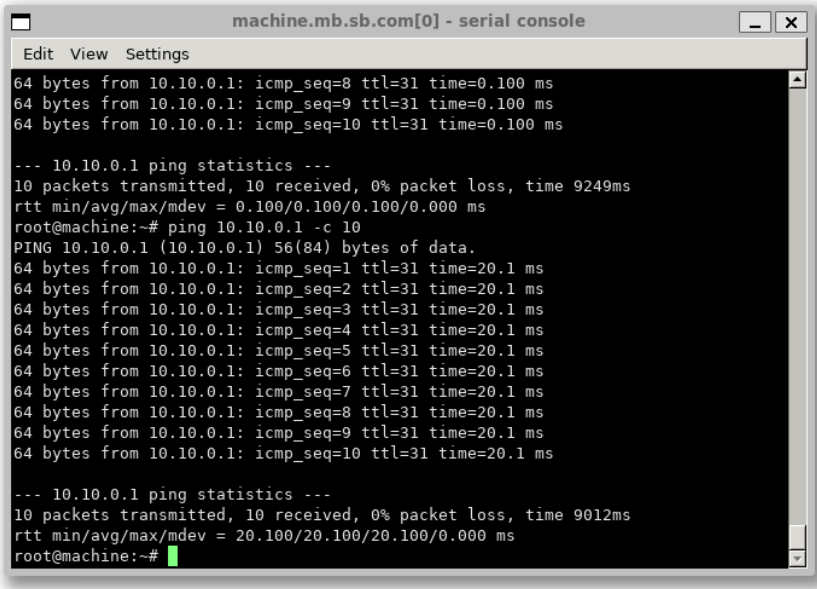
- To see the effect of the network latency on the target software, increase the network latency set on the Ethernet link in the simulation. To set it to 10 ms, use the following command on the simulator command line:

```
running> ethernet_switch.set-goal-latency latency = 0.010
```

- Go back to the target **Serial Console** window and repeat the ping command.

```
# ping 10.10.0.1 -c 10
```

You should now see a much higher latency reported by **ping**. Note that the latency is twice the network goal latency you configured, since each ping requires one packet from the target to the service node and one packet back.



```
machine.mb.sb.com[0] - serial console
Edit View Settings
64 bytes from 10.10.0.1: icmp_seq=8 ttl=31 time=0.100 ms
64 bytes from 10.10.0.1: icmp_seq=9 ttl=31 time=0.100 ms
64 bytes from 10.10.0.1: icmp_seq=10 ttl=31 time=0.100 ms

--- 10.10.0.1 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9249ms
rtt min/avg/max/mdev = 0.100/0.100/0.100/0.000 ms
root@machine:~# ping 10.10.0.1 -c 10
PING 10.10.0.1 (10.10.0.1) 56(84) bytes of data.
64 bytes from 10.10.0.1: icmp_seq=1 ttl=31 time=20.1 ms
64 bytes from 10.10.0.1: icmp_seq=2 ttl=31 time=20.1 ms
64 bytes from 10.10.0.1: icmp_seq=3 ttl=31 time=20.1 ms
64 bytes from 10.10.0.1: icmp_seq=4 ttl=31 time=20.1 ms
64 bytes from 10.10.0.1: icmp_seq=5 ttl=31 time=20.1 ms
64 bytes from 10.10.0.1: icmp_seq=6 ttl=31 time=20.1 ms
64 bytes from 10.10.0.1: icmp_seq=7 ttl=31 time=20.1 ms
64 bytes from 10.10.0.1: icmp_seq=8 ttl=31 time=20.1 ms
64 bytes from 10.10.0.1: icmp_seq=9 ttl=31 time=20.1 ms
64 bytes from 10.10.0.1: icmp_seq=10 ttl=31 time=20.1 ms

--- 10.10.0.1 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9012ms
rtt min/avg/max/mdev = 20.100/20.100/20.100/0.000 ms
root@machine:~#
```

11. Quit the simulation session.

```
running> quit
```

B. Connect to the simulation from the real world

In this section, you will connect the target system to the outside world and connect to the simulation from the outside.

1. Start a simulation session with the checkpoint "**AfterDriver.ckpt**", the second one you saved in lab 001.

```
[C:] simics[.bat] AfterDriver.ckpt
```

2. In the simulator CLI, connect the simulation to the real network using port forwarding. The **connect-real-network** command requires the IP address of the target machine in order to set up a set of default inbound port forward targets. Use the IP address you determined using **ifconfig** above.

```
simics> connect-real-network target-ip = "10.10.0.100"
```

This command will set up a number of default inbound port mappings. The expected output is something like this:

```
NAPT enabled with gateway 10.10.0.1/24 on link ethernet_switch0.link.  
NAPT enabled with gateway fe80::2220:20ff:fe20:2000/64 on link ethernet_switch0.link.  
Host TCP port 4021 -> 10.10.0.100:21  
Host TCP port 4022 -> 10.10.0.100:22  
Host TCP port 4023 -> 10.10.0.100:23  
Host TCP port 4080 -> 10.10.0.100:80  
Real DNS enabled at 10.10.0.1/24 on link ethernet_switch0.link.  
Real DNS enabled at fe80::2220:20ff:fe20:2000/64 on link ethernet_switch0.link.  
Warning: This can expose the target system on the host local network.
```

Note that the ports used on the host might be different if the default ports were already in use. In this case, the simulator finds new port numbers automatically.

3. Run the simulation forward.

```
simics> r
```

"Install" web page

To demonstrate that real network is working, you connect to a web server running on the target system. First, you need to copy a demo web page to the right directory on the target.

4. Verify we have a webserver running on the target. Go to the target **Serial Console** window and run the below command.

```
# ps -x | grep http
```

This should show that a simple Python based webserver is running. E.g.:

```
595 ?          S          0:00 python3 -m http.server -d /var/www 80
```

Note that it serves the content of the directory **/var/www**.

5. Still in the target console, verify that currently there is nothing in that directory.

```
# ls /var/www
```

This should come up empty.

6. Now copy the files for our webpage into **/var/www/** on the target. The files are already in **/home/root/web-page-example**. Run the below command in the target **Serial Console** window.

```
# cp web-page-example/* /var/www/
```

Connect to the web server on the target

7. Check the port forwarding setup using the **list-port-forwarding-setup** command:

```
running> list-port-forwarding-setup
```

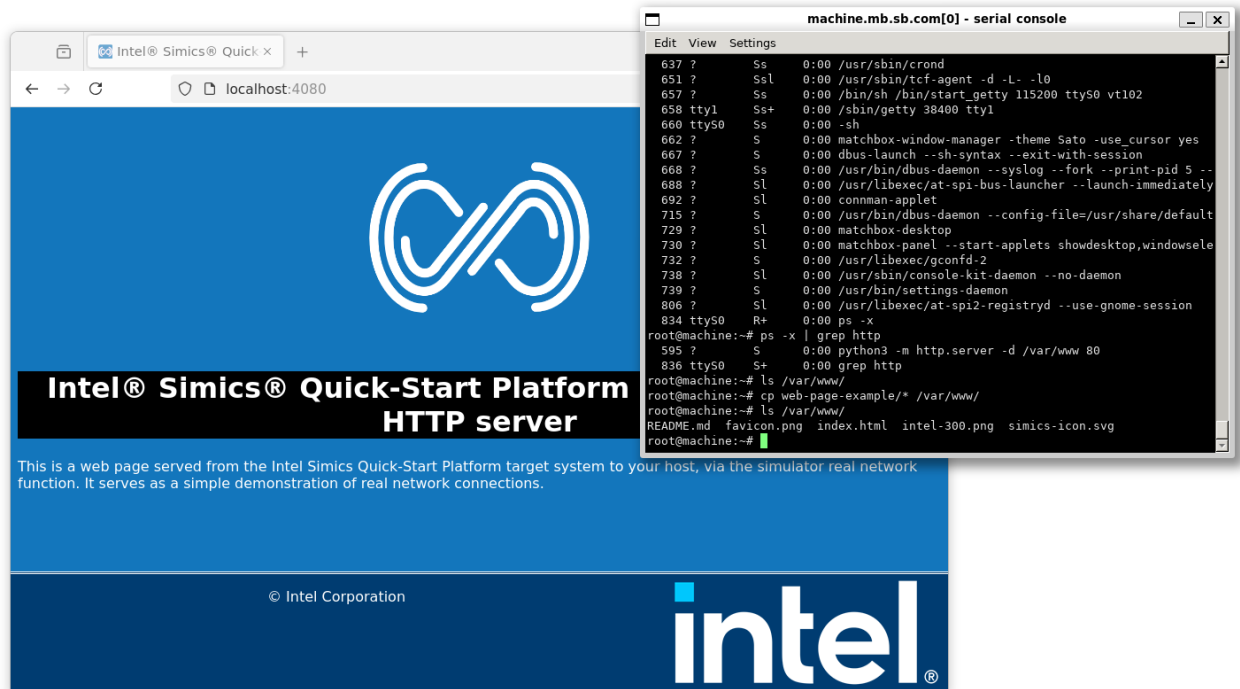
The output is something like this:

```
NAPT enabled with gateway 10.10.0.1/24 on link ethernet_switch0.link.  
NAPT enabled with gateway fe80::2220:20ff:fe20:2000/64 on link ethernet_switch0.link.  
  
Real DNS enabled at 10.10.0.1/24 on link ethernet_switch0.link.  
Real DNS enabled at fe80::2220:20ff:fe20:2000/64 on link ethernet_switch0.link.  
  
Host TCP port 4021 -> 10.10.0.100:21  
Host TCP port 4022 -> 10.10.0.100:22  
Host TCP port 4023 -> 10.10.0.100:23  
Host TCP port 4080 -> 10.10.0.100:80
```

Note which host port is used to reach **10.10.0.100:80** (the default is 4080, but if that port is in use, the simulator will find a different free port to use).

8. Open a web browser on your host and connect to **http://localhost:4080** (assuming **4080** is the port number provided by port forwarding). You should see output on the

target system showing the files being fetched, as well as a web page appearing in the web browser.



Target ssh server

To do this lab, you need an **ssh** client on the host. This is always available on Linux hosts and usually installed by default on Windows 11. Make sure you have a command-line **ssh** available.

Logging in via **ssh** to the target system is not allowed for the **root** user. Thus, you have to use the standard “**simics**” user on the target system. No password is needed, which is obviously not how a real machine should be set up. However, it is OK for a training setup confined inside of the simulation like this.

9. Login to the target system. Since **ssh** clients try to use your local user name on the target system, you need to provide the “**simics**” user name on the target. Use the port number mapped to port 22, according to **list-port-forwarding-setup**. It defaults to 4022, but might be different in case that port was in use:

```
[${C:>}] ssh simics@localhost -p 4022
```

10. The client will ask you to acknowledge the SSH key from the target system – since it has not seen that **ssh** server before. Answer “**yes**”.

```
The authenticity of host '[localhost]:4022 ([127.0.0.1]:4022)' can't be established.  
ECDSA key fingerprint is SHA256:vIdJv90KNdcaaPcgaMmoo4aZrk2VgvZ9bP10Uplar2k.  
This key is not known by any other names  
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes  
Warning: Permanently added '[localhost]:4022' (ECDSA) to the list of known hosts.
```

11. It is quite likely the connection terminates at this point with an unknown error. This is basically a time-out, due to the simulator running much faster than real time.

To avoid the problem, instruct the simulator not to run faster than real time. Go simulator CLI and issue the following command:

```
running> enable-real-time-mode
```

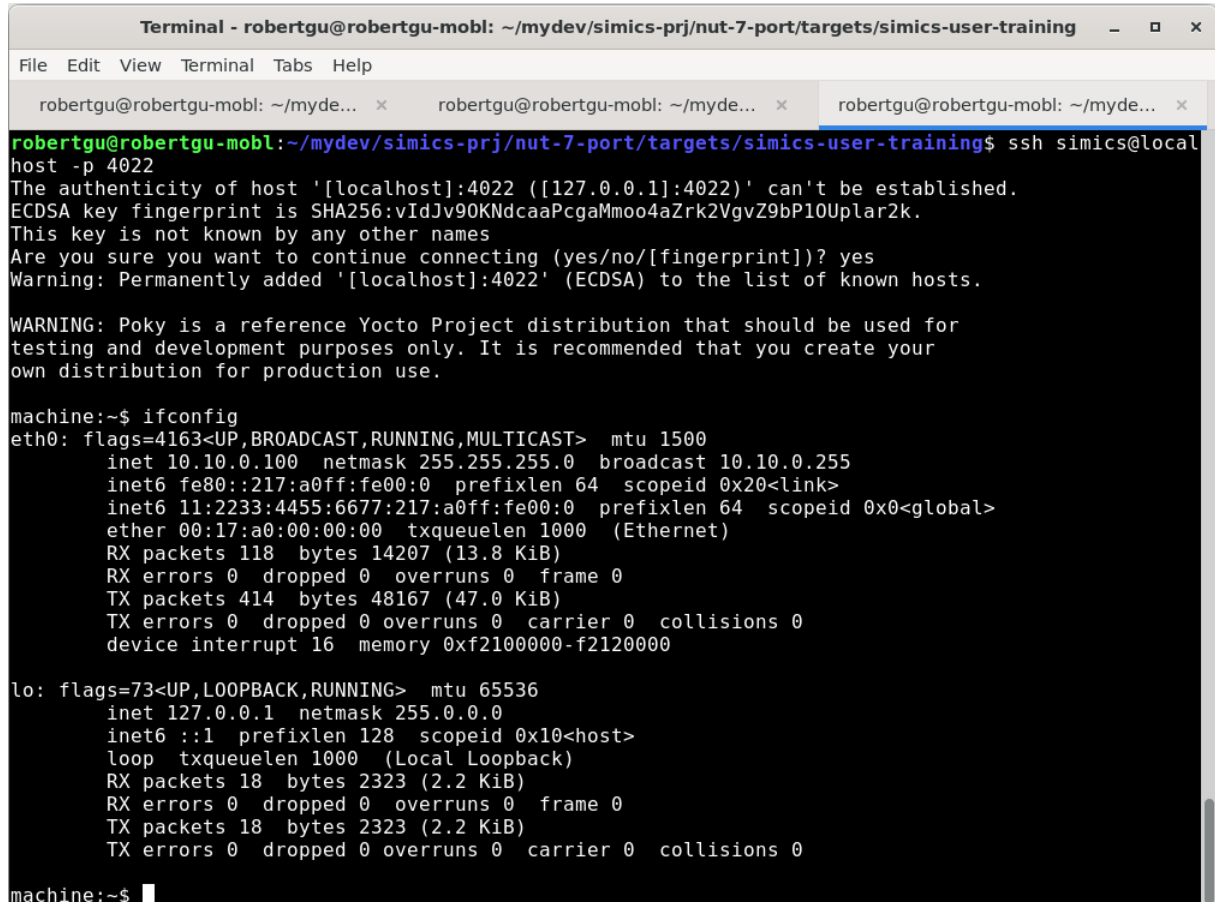
12. Try to connect again. This time it should work.

```
[$|C:>] ssh simics@localhost -p 4022
```

13. Once logged in, check that you are indeed on the target machine. Use the **ifconfig** command on the target, in the **SSH window** (not in the target serial console window).

```
$ ifconfig
```

The result should be the same as what you saw when you ran the command in the target serial console previously:



```
Terminal - robertgu@robertgu-mobl: ~/mydev/simics-prj/nut-7-port/targets/simics-user-training
File Edit View Terminal Tabs Help
robertgu@robertgu-mobl: ~/myde... x robertgu@robertgu-mobl: ~/myde... x robertgu@robertgu-mobl: ~/myde... x
robertgu@robertgu-mobl:~/mydev/simics-prj/nut-7-port/targets/simics-user-training$ ssh simics@local
host -p 4022
The authenticity of host '[localhost]:4022 ([127.0.0.1]:4022)' can't be established.
ECDSA key fingerprint is SHA256:vIdJv90KNdcaaPcgaMmoo4aZrk2VgvZ9bP10Uplar2k.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '[localhost]:4022' (ECDSA) to the list of known hosts.

WARNING: Poky is a reference Yocto Project distribution that should be used for
testing and development purposes only. It is recommended that you create your
own distribution for production use.

machine:~$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 10.10.0.100 netmask 255.255.255.0 broadcast 10.10.0.255
    inet6 fe80::217:a0ff:fe00:0 prefixlen 64 scopeid 0x20<link>
    inet6 11:2233:4455:6677:217:a0ff:fe00:0 prefixlen 64 scopeid 0x0<global>
    ether 00:17:a0:00:00:00 txqueuelen 1000 (Ethernet)
    RX packets 118 bytes 14207 (13.8 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 414 bytes 48167 (47.0 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
    device interrupt 16 memory 0xf2100000-f2120000

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (Local Loopback)
    RX packets 18 bytes 2323 (2.2 KiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 18 bytes 2323 (2.2 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

machine:~$
```

14. Check running processes to find any **sshd** daemons running:

```
$ ps -x | grep sshd
```

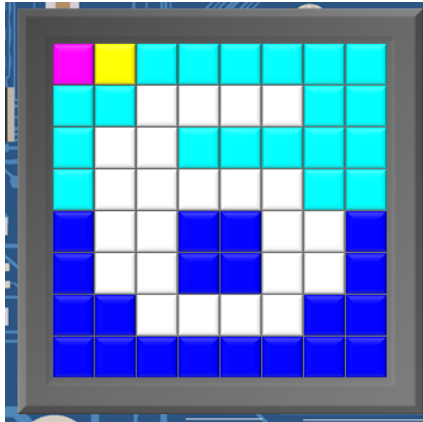
There should be one, indicating a log in as user **simics**.

```
root@cl-qsp ~ # ps -x | grep sshd
449 ?        Ss        0:00 sshd: simics [priv]
471 ttyS0    S+        0:00 grep sshd
```

15. To show that you are indeed logged in to the target, go to the **SSH window** and change the contents of the LED display. Since you are logged in as the regular user **simics** and not as root, you cannot simply use a plain redirect to the device file. Instead, you need to use **tee**...

```
$ echo "ff00ff00ffff0000" | sudo tee /dev/simics_training_pcie_driver
```

16. Go to the **LED Panel** window and check that the contents of the first few pixels of LED display did change.



17. Close the SSH connection from the client. Go to the **ssh window** and do exit:

```
$ exit
```

18. Close the web browser window connected to the target system.
19. **Quit** the simulator.

C. Run multiple machines inside one simulation session

In this exercise, you will run a network of several machines inside a single simulation session. In order to make sure that this works even on weak laptops, we will limit the number of target machines to two, but it is possible to run any number of targets inside a single simulation session – as long as there is enough memory and processor power to run them.

1. Start the simulation using the target “**simics-user-training/007-multimachine-network**”. The target will create three machines and a service node, all connected in a single virtual Ethernet networking inside the simulation.

```
$ ./simics simics-user-training/007-multimachine-network
```

2. Once the setup is completed, go print the target info. You will see three machines and a service node. Note that in this case, the script was written to make the service node a visible entity in its own right – normally, it is just attached to a particular machine and does not show in the target info.

```
simics> print-target-info
```

Which looks like this:

QSP x86 with Linux - Simics Training Setup

Namespace	tmachine
System	a QSP x86 chassis
Processors	2 QSP X86-64, 2000.0 MHz
Memory	8 GiB
Ethernet	1 of 1 connected
Storage	2 disks (6.11 GiB)

Yocto Linux with UEFI

Namespace	qsp
System	a QSP x86 chassis
Processor	1 QSP X86-64, 2000.0 MHz
Memory	8 GiB
Ethernet	1 of 1 connected
Storage	2 disks (6.11 GiB)

Yocto Linux with UEFI

Namespace	qsp_too
System	a QSP x86 chassis
Processor	1 QSP X86-64, 2000.0 MHz
Memory	8 GiB
Ethernet	1 of 1 connected
Storage	2 disks (6.11 GiB)

Pseudo Machine for Ethernet-Based Services

Namespace	servicenode
System	an Ethernet service node
Processor	No processor
Memory	0 B
Ethernet	1 of 1 connected
Storage	No disks

3. Check the top-level namespaces in the simulator configuration. Each machine should be its own top-level namespace (actually component), along with the usual infrastructure objects.

```
simics> list-objects -local
```

The output looks something like this:

Object	Component Class
ethernet_switch	<ethernet_switch>
qsp	<chassis_qsp_x86>
qsp_too	<chassis_qsp_x86>
servicenode	<service_node_comp>
tmachine	<chassis_qsp_x86>

Object	Class
bp	<bp-manager>
default_cell0	<cell>
default_cell0_rec0	<recorder>
default_sync_domain	<sync_domain>
params	<script-params>
prefs	<preferences>
sim	<sim>

4. The parameter tree can also be inspected:

```
simics> params.list -tree substr = name
```

The output shows how the parameters for each machine is slotted into the parameter tree. The parameter tree node names do not entirely align with the names in configuration, as the objects are generated from values in the parameters. Look for the **hardware:name** parameters:

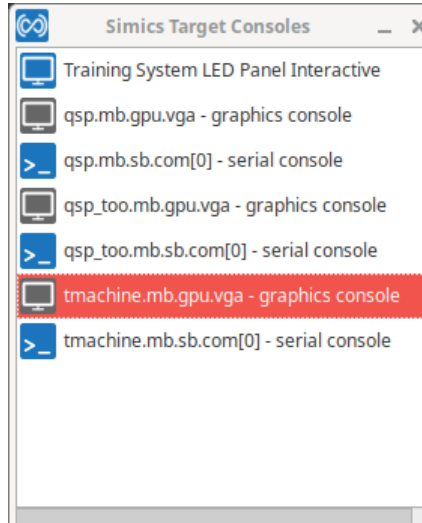
```
└─ 007_multimachine_network
   └─ num_plain_qsps = 2
      └─ training_setup
         └─ qsp[0]
            └─ hardware
               └─ name = qsp
            └─ software
               └─ name = qsp
         └─ qsp[1]
            └─ hardware
               └─ name = qsp_too
            └─ software
               └─ name = qsp_too
```

5. Check the set of processors in the simulation:

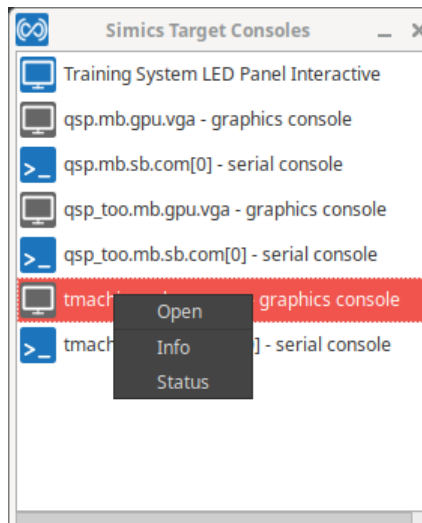
```
simics> list-processors
```

The target machines called **qsp** each have one processor, while the one called **tmachine** has two.

6. The **Simics Target Consoles** window will show all the consoles for all the machines, allowing you to hide and show individual consoles from a central location:



7. **Hide** the graphics consoles by double-clicking on each of their lines in the in the window. You can also right-click on a console to close it and check its info and status:



8. From the command-line, the connection information is found using the **status** command on the **ethernet_switch0** component:

```
simics> ethernet_switch.status
```

The output should look like this:

```
Status of ethernet_switch0 [class ethernet_switch]
=====

Setup:
  Top component : servicenode
  Instantiated  : True

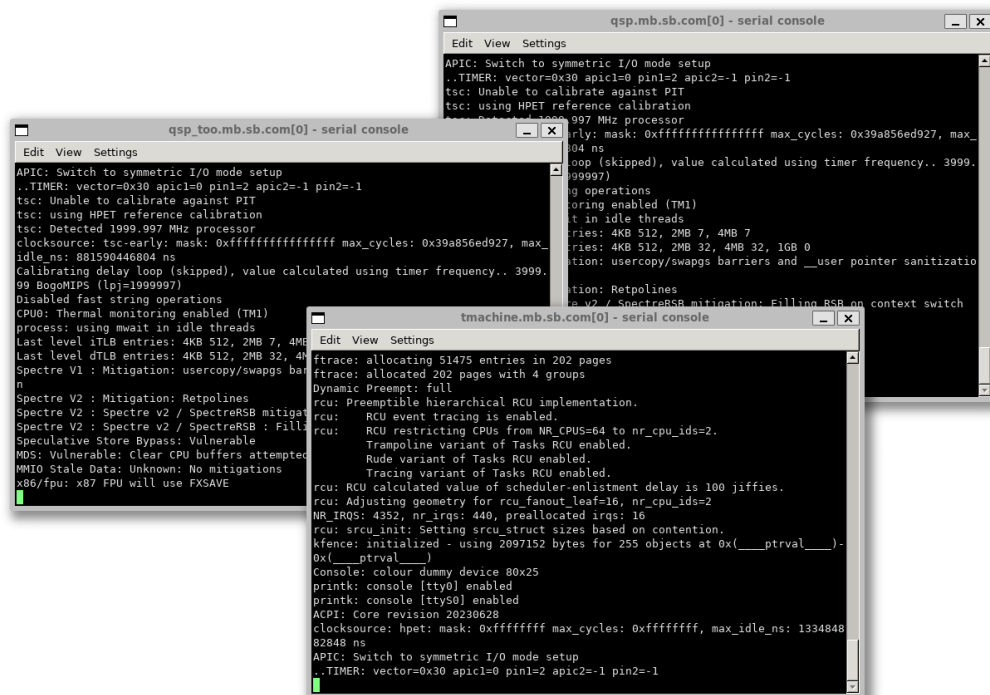
Attributes:
  global_id    : none
  goal_latency : 1e-05

Connections:
  device0 : servicenode:connector_link0
  device1 : tmachine.mb.sb:eth_slot
  device2 : qsp.mb.sb:eth_slot
  device3 : qsp_too.mb.sb:eth_slot
```

9. Start the simulation to boot the target machines.

```
simics> r
```

10. Bring all the serial consoles to front to see the machines in the network boot.



Note that you might see activity happen on one machine, and then switch to another. This happens since the multimachine multithreading and temporal decoupling can

make each machine run for up to 10ms of target time before switching to another target machine.

Investigate the network configuration as seen by target software

11. Wait for all machines to arrive at prompt.
12. Check on the network configuration of the target machines. It takes some time to get the network up and configured, as Linux queries **DHCP** for an address.

Go to the target **Serial Console** window called **qsp.mb.sb.com[0]**, and use **ifconfig** to check if the interface has an ipv4 address.

```
# ifconfig eth0
```

Repeat until it shows that an IP address of the form "10.10.0.xx" is available, where "xx" can be any number.

13. Verify in the same way that the other two targets have a valid IP address.
14. All the target machines retrieved their IP addresses from the DHCP server in the service node. Check the status of the DHCP server on the simulator command line:

```
simics> servicenode.dhcp-leases
```

15. The script also set up DNS names for the machines in the virtual network. To check the DNS state, use the list-host-info command on the service node:

```
simics> servicenode.list-host-info
```

This information is used to perform DNS in the simulated network. Note that all the machines have different Ethernet MAC addresses, which is necessary for the network to function:

IP	name.domain	MAC
10.10.0.1	simics0.network.sim	20:20:20:20:20:00
10.10.0.10	tmachine.network.sim	00:17:a0:00:00:00
10.10.0.11	qsp.network.sim	00:17:a0:00:00:01
10.10.0.12	qsp_too.network.sim	00:17:a0:00:00:02

Network operations between the machines

16. Go to the **Serial Console** called **qsp.mb.sb.com[0]**, and **ssh** over to the **tmachine** target, using the DNS information. Note that you cannot log in as root, thus you need to specify the user **simics**:

```
# ssh simics@tmachine
```

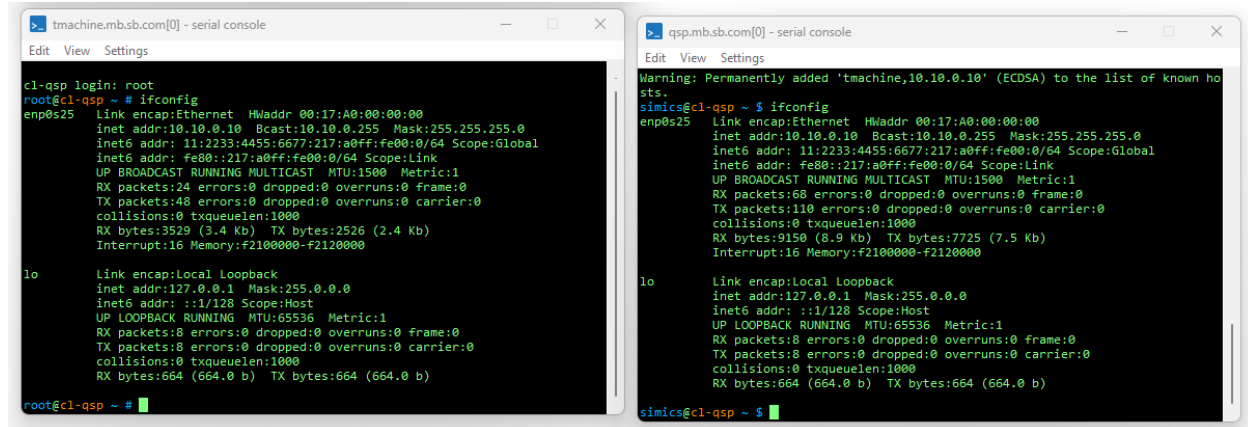
The way the QSP Yocto Linux is configured, there is no need for a password.

Note that due to how the Yocto Linux disk image is set up, the hostname of it is not set via DHCP. Thus, the hostname that Linux displays will not match the DNS name of the machine.

17. Check that the login worked by comparing the **ifconfig** output in the **ssh** login with the **ifconfig** output from the serial port of **tmachine**:

```
$ ifconfig
```

Obviously, the output should show the same IP address and MAC address. Like this:



18. Become **root** in the **ssh** login, in order to have equivalent rights to what you had in previous labs.

```
$ sudo -s
```

19. Go to the home directory for **root**:

```
# cd ~
```

20. List the available files using **ls**. You should see the contents of the files on the training machine (since that is what you logged in to):

```
# ls
```

You should see something like this:

```
simics@cl-qsp ~ $ sudo -s
root@cl-qsp /home/simics # cd ~
root@cl-qsp ~ # ls
README.md  chessboard1.txt  lines.txt          six.txt
System.map chessboard2.txt  simics-training-pcie-driver.ko  web-page-example
```

21. Go to the **Serial Console** window for the **tmachine** target. Check the network connections using **netstat** and **grep**:

```
# netstat | grep ssh
```

This should show something like this:

```
tcp      0      0 tmachine.network.si:ssh qsp.network.sim:44700 ESTABLISHED
```

Note the use of the machine names from DNS to show where the connection comes from.

Optional: Update the LED panel display

22. From the serial console that has **ssh**:ed into the **tmachine** target, do **insmod** and **cat** to activate the device, like you did above.

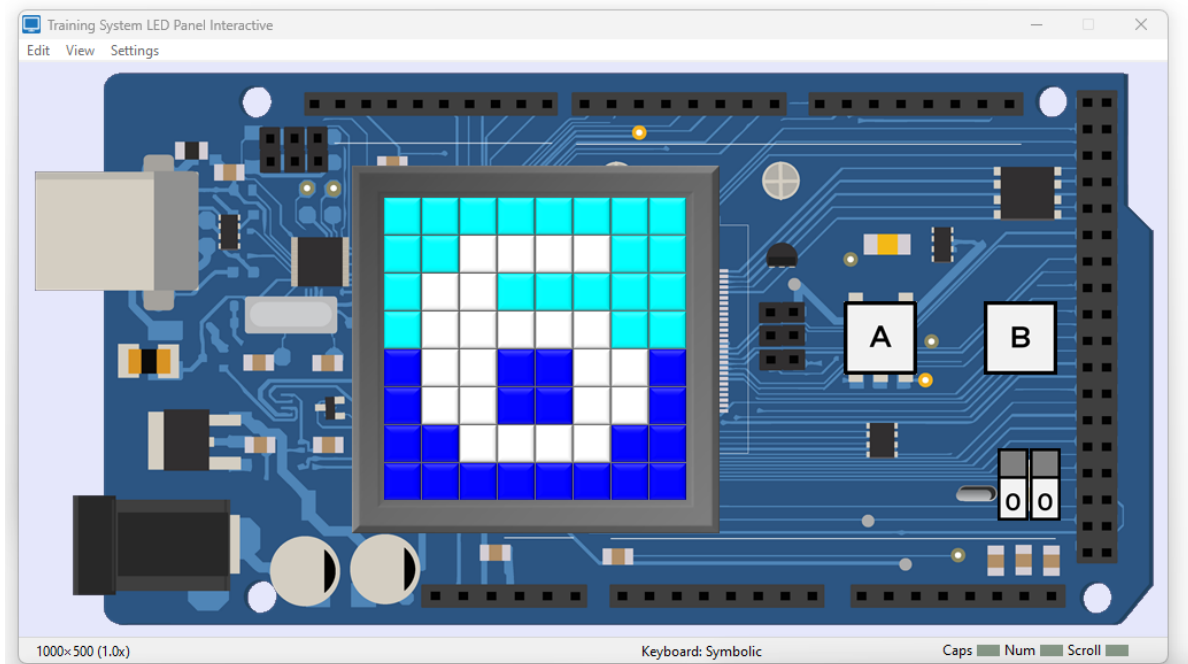
```
# /sbin/insmod simics-training-pcie-driver.ko
```

And then:

```
# cat six.txt > /dev/simics_training_pcie_driver
```

23. Check the **LED Panel** for the results.

You should see something like below.



Check the resource use of the simulation process

Simulating a network in the Intel Simics Simulator only uses the resources needed to represent the state of the machines in the network. Despite simulating three machines with a total of 32GiB of RAM and 324GiB of disks (the disks are silly small, admittedly), the process is not likely using all that much RAM.

24. Use the tech-preview Probes system to see how much memory the process is using. First, enable the probes system:

```
running> enable-probes
```

25. List the probes associated with the host memory usage of the process:

```
running> probes.list-kinds substr = sim.process.memory
```

26. Read the probe that correspond to the host memory usage of the simulation process. There are a few different ones. The “resident” probe corresponds to the actual current use of host RAM:

```
running> probes.read sim:sim.process.memory.resident
```

The result is expected to be a few gigabytes, at most.

27. Read the virtual size of the process, which could be much bigger than the resident size:

```
running> probes.read sim:sim.process.memory.virtual
```

28. Most of the memory is expected to be used by the simulator image system. Checks its current memory usage using the **print-image-memory-stats** command:

```
running> print-image-memory-stats -h
```

29. The total memory usage of the image system can be read out using probes. List probe kinds associated with images:

```
running> probes.list-kinds substr = image
```

30. And read the total image system memory usage:

```
running> probes.read sim:sim.image.memory_usage
```

Keep the simulation running

At this point, keep the simulation running as the starting point for the next section.

31. Pause the execution in the simulation. Do NOT quit the session.

D. Analyze network traffic with Wireshark

This section continues from the previous, with a booted network of three machines running in the simulation. You will learn how to use the Wireshark program to inspect network traffic. You need to have Wireshark installed on your host!

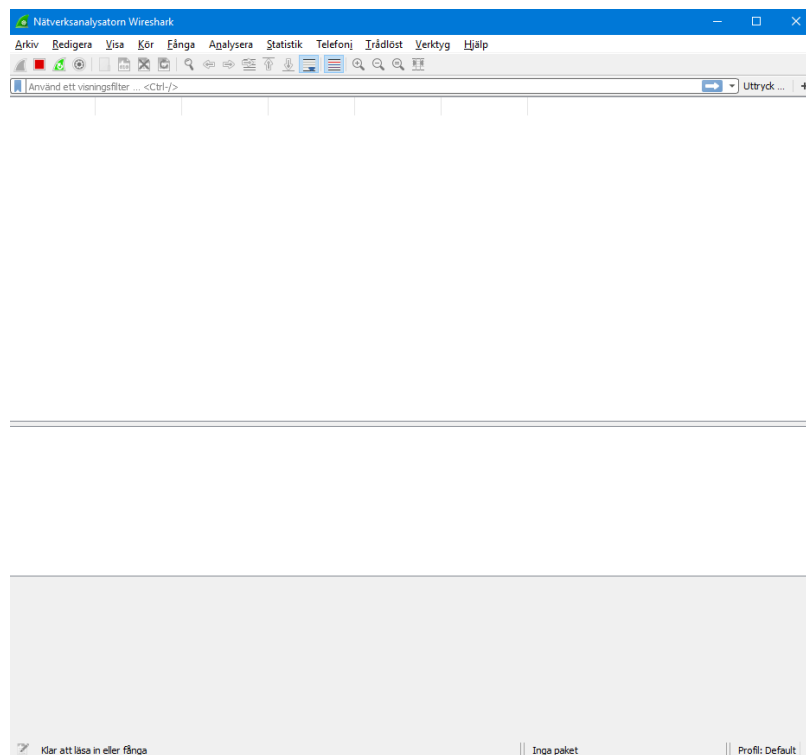
1. In the simulator CLI, tell the simulator where to find the Wireshark program executable. If you have installed Wireshark in the standard way on Windows, it should be located in **C:\Program Files\Wireshark**.

```
simics> prefs->wireshark_path = "c:\\Program Files\\Wireshark"
```

2. Start Wireshark and log network traffic using the **"wireshark"** command:

```
simics> wireshark
```

3. This should start a separate Wireshark process running on your host, connected to the network traffic in the simulation. **Wireshark** will show that it is capturing from standard input:



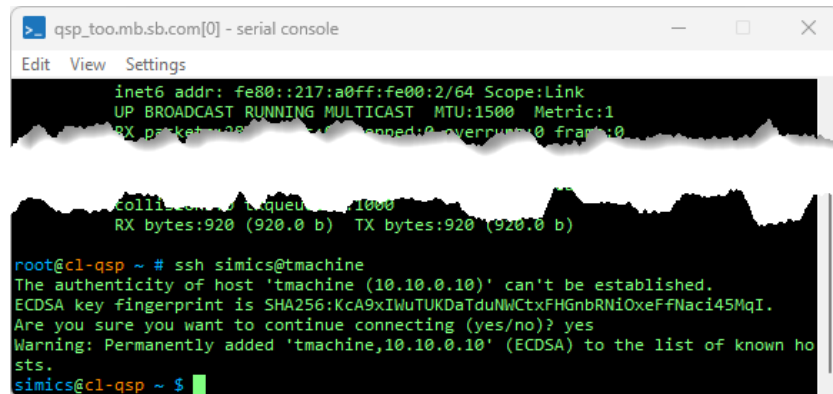
4. Run the simulation forward.

```
simics> r
```

5. Create some network traffic by going to the **Target Serial** console for the **qsp_too** target (**qsp_too.mb.sb.com[0]**). Use **ssh** to connect from this target to the **tmachine** target.

```
# ssh simics@tmachine
```

The result should look like this:



The screenshot shows a serial console window titled "qsp_too.mb.sb.com[0] - serial console". The window contains the following text:

```
inet6 addr: fe80::217:a0ff:fe00:2/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:0 (0 b) TX packets:0 (0 b)
RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)

Collision: 0 Queue: 1000
RX bytes:920 (920.0 b) TX bytes:920 (920.0 b)

root@cl-qsp ~ # ssh simics@tmachine
The authenticity of host 'tmachine (10.10.0.10)' can't be established.
ECDSA key fingerprint is SHA256:KcA9xIWuTUKDaTduNWCtxFHGnbRniOxeFFNaci45MqI.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'tmachine,10.10.0.10' (ECDSA) to the list of known ho
sts.
simics@cl-qsp ~ $
```

6. Once you have logged in to **tmachine** over **ssh**, pause the simulation (otherwise the trace will just keep piling on).

```
running> stop
```

- Go to the **Wireshark** window to review the traffic. Scroll back up to the start of the capture.

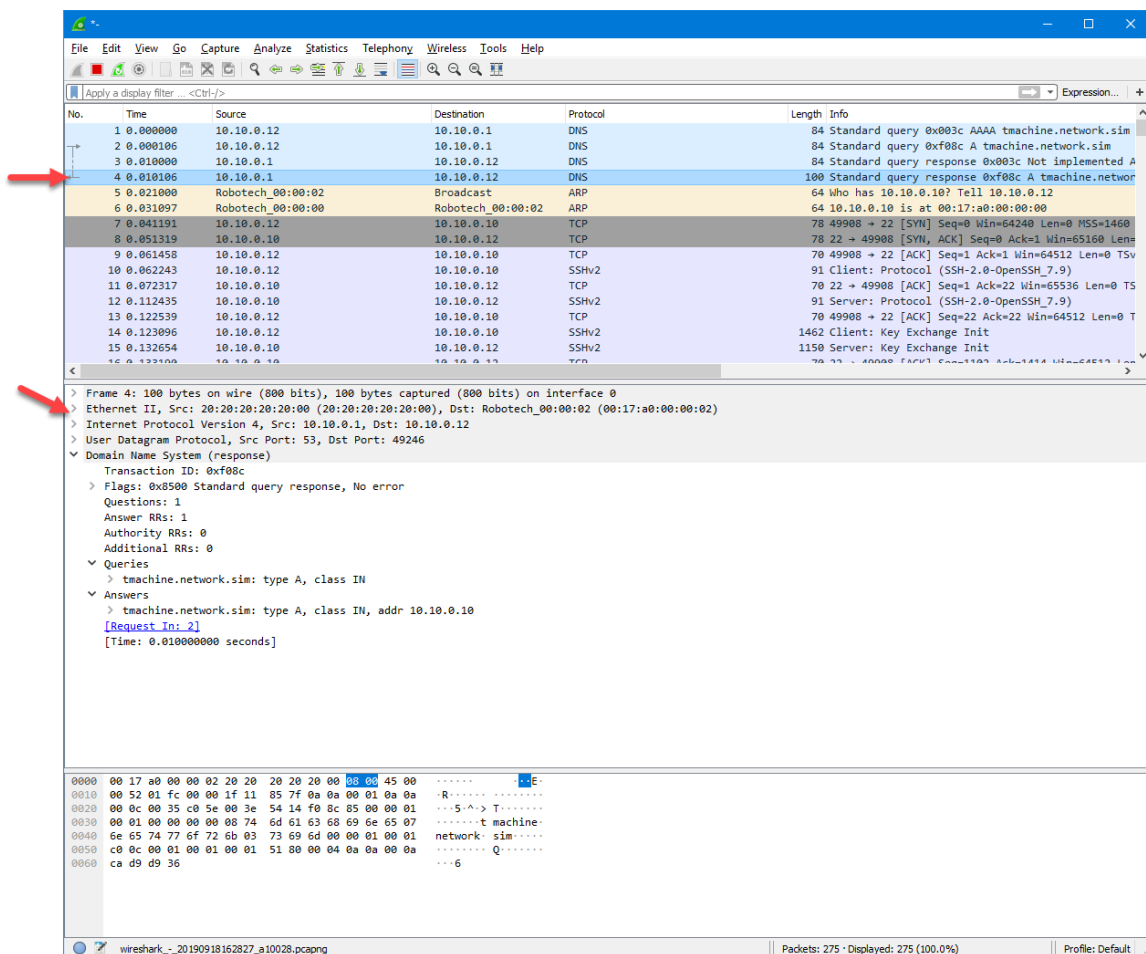
The image shows a Wireshark window displaying a network traffic capture. The main pane shows a list of 25 packets. The first packet is a DNS Standard query from 10.10.0.12 to 10.10.0.1. The second packet is a DNS Standard query response from 10.10.0.1 to 10.10.0.12. The third packet is a DNS Standard query response from 10.10.0.1 to 10.10.0.12. The fourth packet is a DNS Standard query response from 10.10.0.1 to 10.10.0.12. The fifth packet is an ARP request from Robotech_00:00:02 to Broadcast. The sixth packet is an ARP request from Robotech_00:00:02 to Robotech_00:00:02. The seventh packet is a TCP SYN from 10.10.0.12 to 10.10.0.10. The eighth packet is a TCP SYN, ACK from 10.10.0.10 to 10.10.0.12. The ninth packet is a TCP ACK from 10.10.0.10 to 10.10.0.12. The tenth packet is a TCP ACK from 10.10.0.10 to 10.10.0.12. The eleventh packet is a TCP ACK from 10.10.0.10 to 10.10.0.12. The twelfth packet is a TCP ACK from 10.10.0.10 to 10.10.0.12. The thirteenth packet is a TCP ACK from 10.10.0.10 to 10.10.0.12. The fourteenth packet is a TCP ACK from 10.10.0.10 to 10.10.0.12. The fifteenth packet is a TCP ACK from 10.10.0.10 to 10.10.0.12. The sixteenth packet is a TCP ACK from 10.10.0.10 to 10.10.0.12. The seventeenth packet is a TCP ACK from 10.10.0.10 to 10.10.0.12. The eighteenth packet is a TCP ACK from 10.10.0.10 to 10.10.0.12. The nineteenth packet is a TCP ACK from 10.10.0.10 to 10.10.0.12. The twentieth packet is a TCP ACK from 10.10.0.10 to 10.10.0.12. The twenty-first packet is a TCP ACK from 10.10.0.10 to 10.10.0.12. The twenty-second packet is a TCP ACK from 10.10.0.10 to 10.10.0.12. The twenty-third packet is a TCP ACK from 10.10.0.10 to 10.10.0.12. The twenty-fourth packet is a TCP ACK from 10.10.0.10 to 10.10.0.12. The twenty-fifth packet is a TCP ACK from 10.10.0.10 to 10.10.0.12.

Domain Name System (response)
Transaction ID: 0xf08c
Flags: 0x8500 Standard query response, No error
Questions: 1
Answer RRs: 1
Authority RRs: 0
Additional RRs: 0
Queries
 > tmachine.network.sim: type A, class IN
Answers
 > tmachine.network.sim: type A, class IN, addr 10.10.0.10
[Request in: 2]
[Time: 0.010000000 seconds]

0000 00 17 a0 00 00 02 20 20 20 20 00 08 00 45 00E
0010 00 52 01 fc 00 00 1f 11 85 7f 0a 0a 00 01 0a 0a ..R.....
0020 00 0c 00 35 c0 5e 00 3e 54 14 f0 8c 85 00 00 01 ...S.A->T.....
0030 00 01 00 00 00 00 08 74 6d 61 63 68 69 6e 65 07t machine
0040 6e 65 74 77 6f 72 6b 03 73 69 6d 00 00 01 00 01 network.sim.....
0050 c0 0c 00 01 00 01 00 01 51 80 00 04 0a 0a 00 0aQ.....
0060 ca d9 d9 366

wireshark_-_20190918162827_a10028.pcapng | Packets: 275 · Displayed: 275 (100.0%) | Profile: Default

8. Find the **DNS** exchange that signals the start of the **ssh** session from **qsp_too** to **tmachine**. Check that it indeed gets the address of **tmachine** over DNS:
 - a. DNS packets from 10.10.0.12 to 10.10.0.1, looking up the hostname **tmachine**.
 - b. Potentially an ARP asking for the MAC of 10.10.0.10, to tell 10.10.0.12 (depending on the system state, the MAC of 10.10.0.10 might already be cached by 10.10.0.12 due to other previous traffic)
 - c. TCP and then SSHv2 between 10.10.0.12 (**qsp-too**) and 10.10.0.10 (**tmachine**).
9. Click on a packet to look at the details of its content – Wireshark is really good at investigating the contents of network packets. For example, the last DNS packet before the ARP:



This shows that the virtual network looks just like a real network to the software running on top of the simulated target. Tip: check out the little arrows at the edge of the list that shows which packets are replies to which other packets. Very handy feature.

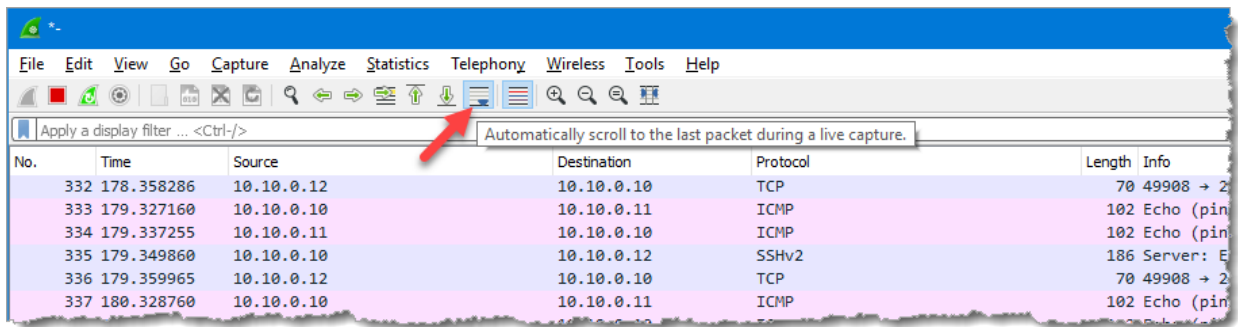
10. Run the simulation forward again.

```
simics> r
```

11. Add some more packets to the trace by going to Serial Console for the **qsp_too** target (which is logged into **tmachine**), and ping the **qsp** machine.

```
# ping qsp
```

12. Go over to **Wireshark** and see how the **ICMP** packets come in alongside **TCP** and **SSH** packets. Click the button “Automatically scroll to the last packet...” to make sure you see the packets as they come in.



13. Watch the traffic for a while.
14. **Pause** the simulation.

```
running> stop
```

Continue to the next section.

E. Checkpoint the simulated network

Checkpoints encompass entire networks – including all network activity.

1. With the simulation paused, as should be after the end of the previous section, save a checkpoint.

```
simics> write-configuration network-in-action.ckpt
```

2. **Quit** the simulation session. This will also make **Wireshark** quit since the simulation session it was attached to has been shut down.

```
simics> quit
```

Open the checkpoint

3. **Open** the checkpoint you just saved.

```
[$|C:>] simics[.bat] network-in-action.ckpt
```

The simulation should be back exactly where you left off.

4. Check the set of machines in the simulation:

```
simics> print-target-info
```

It should be the same as what you saw before.

5. Check the simulated time:

```
simics> ptime -all
```

It is possible some processor has a time that is different from the others. This is due to the simulation being stopped at a point where not all simulation cells are synchronized. The simulation will continue exactly as it did before.

6. Start the simulation running again.

```
simics> r
```

Note how the **ping** process keeps running, and that all the **ssh** sessions are still up and working. This is thanks to the simulator entirely encapsulating the state across all the target machines, including all state including the local time in each. For the networked machines, the checkpoint save and restore is completely invisible.

7. **Quit** the simulation session.

```
running> quit
```

This concludes the network checkpoint lab.

F. Use record-replay to repeat a session

Use record-replay to reproduce a session where you interact with the target system.

1. Open the second checkpoint from lab 001, with the kernel driver for the training device **insmod**-ed and the device running (usually called "**AfterDriver.ckpt**").

```
[${C:>}] simics[.bat] AfterDriver.ckpt
```

Do not start the simulation running just yet!

2. Enable real-time mode via the simulator command line.

```
simics> enable-real-time-mode
```

By using real-time mode, the replay will be somewhat shorter since there the simulator spends less virtual time between each keypress.

Record target actions

3. Start recording the current session. Note that the recording will be stored in what we refer to as a session checkpoint.

```
simics> record-session RecordedTest.ckpt
```

Showing the message:

```
Recording of asynchronous input started
```

When doing a session checkpoint, the simulator must save the initial state in addition to the recorded inputs. Thus, the first action is establishing a checkpoint even though you have not actually done anything to record yet!

4. You can verify that recording is active by inspecting one of the input records in the session. Depending on your setup you might have multiple. First, list all recorders.

```
simics> list-objects class = recorder
```

Which should find two recorders:

Object	Class
machine.cell_rec0	<recorder>
machine.training_card.panel.rec	<recorder>

5. Inspect the status of the cell recorder of **machine**:

```
simics> machine.cell_rec0.status
```

Note that it is recording into a file in the checkpoint directory.

```
Status of machine.cell_rec0 [class recorder]
=====

    Recording : yes (file: RecordedTest.ckpt/recording.machine.cell_rec0)
    Playing back : no
```

6. Check that the other recorder is also recording to a file:

```
simics> machine.training_card.panel.rec.status
```

7. **Run** the simulation forward.

```
simics> r
```

8. Go to the **Target Serial** console for the target serial port.

9. Type in a command to change the contents of the LED display. It is more interesting if you actually **type in the commands and use tab-completion**, rather than pasting from the lab instructions, since you will want to see the key-by-key playback later.

```
# cat lines.txt > /dev/simics_training_pcie_driver
```

10. Type in another command:

```
# echo "00ffff0000ffff00" > /dev/simics_training_pcie_driver
```

11. And another command:

```
# ls
```

12. **Pause** the simulation.

```
running> stop
```

13. Stop the recording in the simulator CLI.

```
simics> stop-recording
```

Which should be acknowledged:

```
Recording of session RecordedTest.ckpt stopped
```

14. **Quit** the simulation session.

```
simics> quit
```

Replay the recording

15. Start a new session using the checkpoint **RecordedTest.ckpt**.

```
[C:] simics[.bat] RecordedTest.ckpt
```

Note the message **Playback of recorded session from checkpoint started** in the CLI.

```
$ ./simics RecordedTest.ckpt
Intel Simics 6 (build 6243 win64) © 2023 Intel Corporation

Use of this software is subject to appropriate license.
Type 'copyright' for details on copyright and 'help' for on-line documentation.

Playback of recorded session from checkpoint started
```

16. Arrange your windows so that you can see both the **Target Serial Console** and the **LED Panel** at the same time.
17. Enable real-time-mode, to avoid playing back too quickly:

```
simics> enable-real-time-mode
```

18. **Run** the simulation forward.

```
simics> r
```

19. Look at how the recorded key presses are replayed in the **Target Serial Console** window, and how the display in the **LED Panel** window updates.
20. Once the recorded input has been replayed, the simulation will stop.

```
[machine.training_card.panel.rec info] End of recording
```

21. **Quit** the simulation session.

```
simics> quit
```

This concludes the lab.

G. Optional. Inspect the network setup script

Look at the network script you used in Section B to create the network. It provides a robust template for how other network scripts can be built.

1. Identify the YAML file that defines the target you just started.

```
simics> list-targets -verbose substr = network
```

The command shows the path to the YAML target definition in your Intel Simics Simulator installation.

2. Open the file target definition file (`/path/to/install/.../007-multimachine-network.target.yml`) in a text editor.

The file looks like the below. It is a YAML-format file using keywords defined by the simulator parameter system.

```
%YAML 1.2
---
description: A network of machines for the training.

params:
  num_plain_qsp:
    type: int
    default: 2

  training_setup:
    import: "%script%/001-qsp-training.target.yml"
    defaults:
      platform:
        machine:
          hardware:
            name: tmachine
        network:
          service_node:
            create: false

  qsp[num_plain_qsp]:
    import: "%simics%/targets/qsp-x86/yocto-linux-setup.yml"
    defaults:
      system_info: null
      hardware:
        name: null
    provides:
      - hardware:name
script: "%script%/007-multimachine-network.target.yml.include"
...
```

This shows how the network is built up from one training machine and a configurable number of “plain” Intel Simics Quick-Start Platforms. Note the array declaration **qsp[num_plain_qsp]**, which provides parameters for all the Intel Simics Quick-Start Platform models (as seen above when listing the parameters).

3. Open the script file that sets up the network based on the parameters: "**007-multimachine-network.target.yml.include**", found next to the YAML file you already opened.

The code is well-commented and should be self-explanatory. Here are some key points to look for:

4. Note the use of **run-script** to run the targets that are imported. Using **run-script** means that the filename used is the same as that used to import the **.yml** file into the target parameter set.
5. Note the loop used to create the plain Intel Simics Quick-Start Platform instances. This loop mirrors the array in the target declaration file. Just declaring an array of targets will not cause them to be created, the script has to create each target in turn and connect them to the network.

Test a bigger network

After looking at the script, it makes sense to see what happens if the number of machines is increased. There is no need to run anything, but it is interesting to look at the created setup.

6. Start an empty simulation session:

```
[C:] simics[.bat]
```

7. Load the network target, with five plain QSP machines:

```
simics> load-target "simics-user-training/007-multimachine-network"  
num_plain_qsps = 5
```

8. Check the set of machines:

```
simics> print-target-info
```

9. Check the top-level objects:

```
simics> list-objects -local
```

Note the top-level object names generated by the name generator in the script (**qsp**, **qsp_too**, **qsp_three**, **qsp_4**, ...).

10. **Quit** the simulation session.

```
simics> quit
```

This concludes the lab.

H. Optional: Create an independent checkpoint

Checkpoints are normally saved as diffs, in order to reduce their size (which also makes saving them faster). Checkpoints can also be made independent, in order to make them easier to transport.

1. Start a new session with the checkpoint you saved in the recording lab.

```
[C:] simics[.bat] RecordedTest.ckpt
```

2. Inspect the checkpoint dependency chain. The exact paths will differ.

```
simics> sim->checkpoint_path
```

It will show a list of paths:

```
["/home/abcdefg/simics-projects/my-simics-project-for-training-  
purposes/MyFirstCheckpoint.ckpt", "/home/abcdefg/simics-projects/my-simics-project-for-  
training-purposes/AfterDriver.ckpt", "/home/abcdefg/simics-projects/my-simics-project-  
for-training-purposes/RecordedTest.ckpt"]
```

3. The list is in order of “age”, oldest checkpoint first. So you see that **RecordedTest.ckpt** depends on **AfterDriver.ckpt**, which in turn depends on **MyFirstCheckpoint.ckpt**. Hence, to work, **RecordedTest.ckpt** needs the previous two checkpoints in the chain.

Note that even if this list only contains your checkpoint, there can still be dependencies on initial images somewhere on your file system.

4. Save the current state in a new checkpoint, but this time, pass the **-independent-checkpoint** flag.

```
simics> write-configuration -independent-checkpoint RecordedTest-  
independent.ckpt
```

Note that saving this checkpoint will take noticeably longer than saving previous checkpoints.

5. **Quit** the simulation session.

```
simics> quit
```

Test the independent checkpoint

6. Open the new checkpoint to check that it works.

```
[C:] simics[.bat] RecordedTest-independent.ckpt
```

7. Inspect the dependency chain again.

```
simics> sim->checkpoint_path
```

It should be much shorter:

```
["/home/abcdefg/simics-projects/my-simics-project-for-training-purposes/Lab-006-D-independent.ckpt"]
```

So now this checkpoint does not depend on any other checkpoint and since it was created with the **-independent-checkpoint** flag will also not depend on other files somewhere on the disk. Hence, it can be passed on to other people.

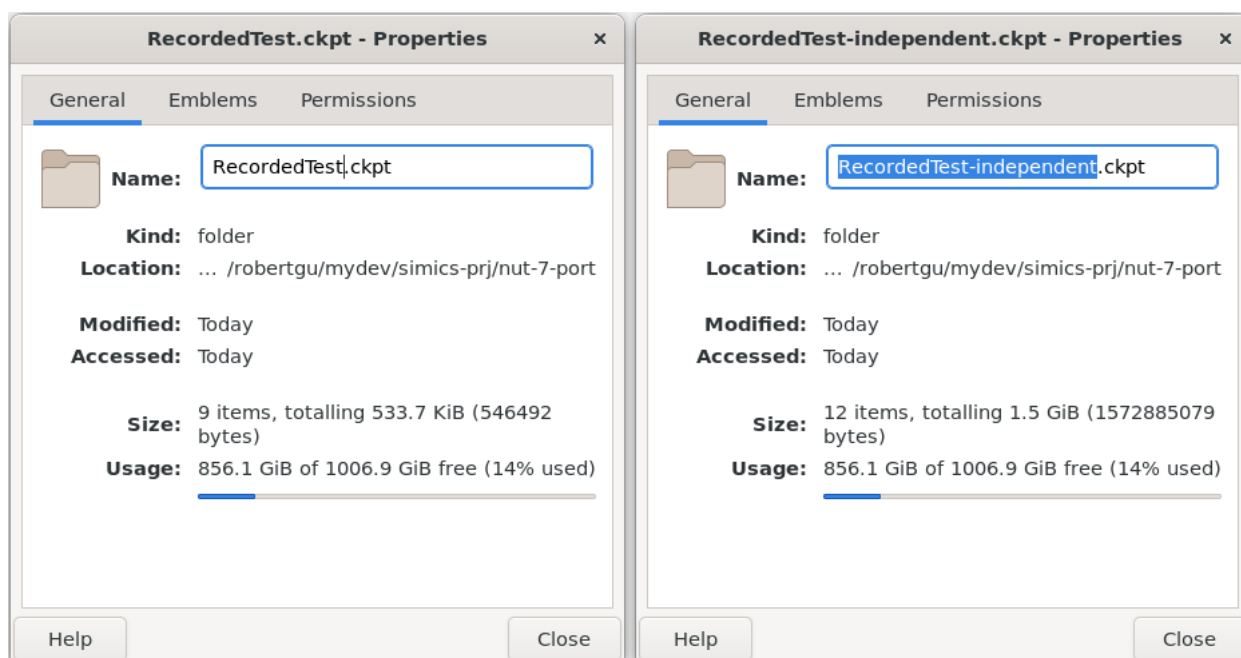
Note that the independent checkpoint save did not include the recording.

8. In principle, you could now delete the previous checkpoints to demonstrate that the new checkpoint is indeed independent. However, that makes it harder to go back and finish up any left-overs in previous labs, and it is not recommended.

Investigate checkpoint size

An independent checkpoint is convenient, but also quite a bit bigger.

9. In Windows explorer (or equivalent tool on a Linux host), go to your project directory.
10. Check the size of the two checkpoints, by getting their properties.



The independent checkpoint is much bigger since it includes the base Yocto Linux disk image in addition to all the changes to disk and memory that has happened over the course of the labs.