Hands-On Tutorial with OpenThread

This lab procedure walks through the steps to create a Thread network and an OpenThread Border Router. The first part reviews how to create an OpenThread project in Simplicity Studio v5. The second part shows how to create a network with three nodes, and how to use some basic commands to explore the feature of each node. The final part introduces how to remove and add a node, and analyze the status of each node. A demonstration of commissioning ends this session.

KEY POINTS

- Create OpenThread project for EFR32MG12
- Create a Thread network
- Add and remove a node and analyze the network topology change
- Create an OpenThread Border Router
- Use commissioning with Thread
- Use Silicon Labs tools to analyze the network

1 Prerequisites

For this lab you will need the following:

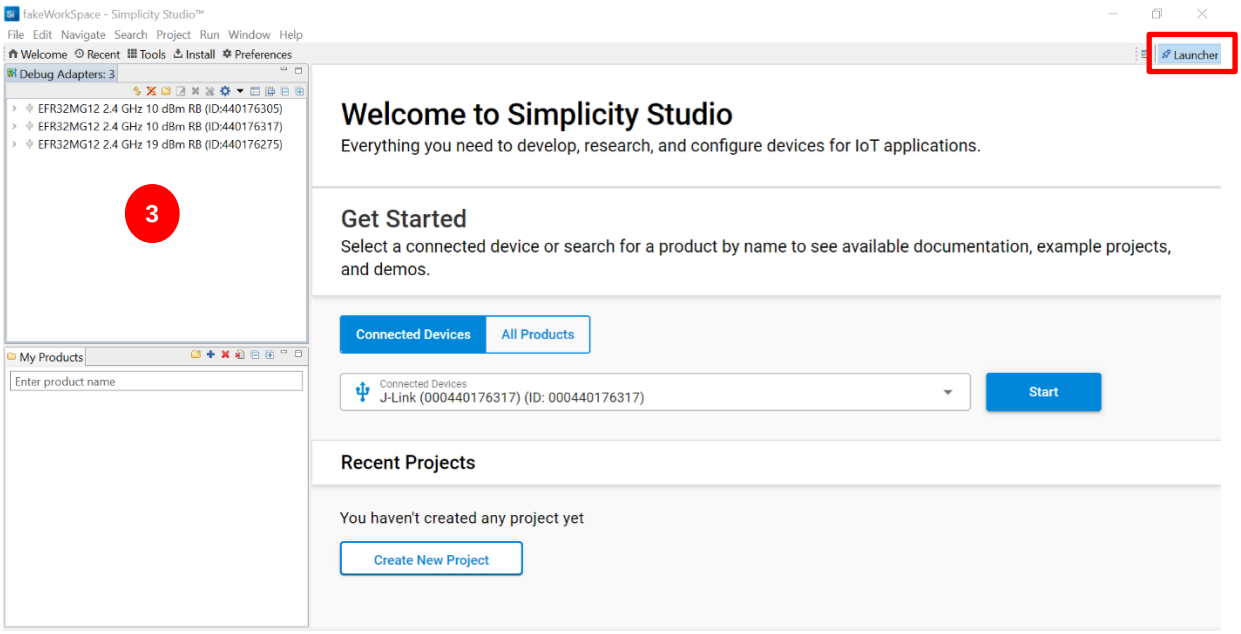
- Three EFR32MG12 radio boards - this tutorial assumes BRD4170A, although other EFR32MG12 radio boards would work too with possible minor adaptation in instructions
- Three Mini-USB Type-B to USB Type-A cable supplied with any Wireless Gecko starter kit
- Simplicity Studio v5
    - GNU ARM V7
    - Gecko SDK Suite 3.1.0 or later
- (Optional) a serial terminal such as Tera Term

2 Create an OpenThread Project

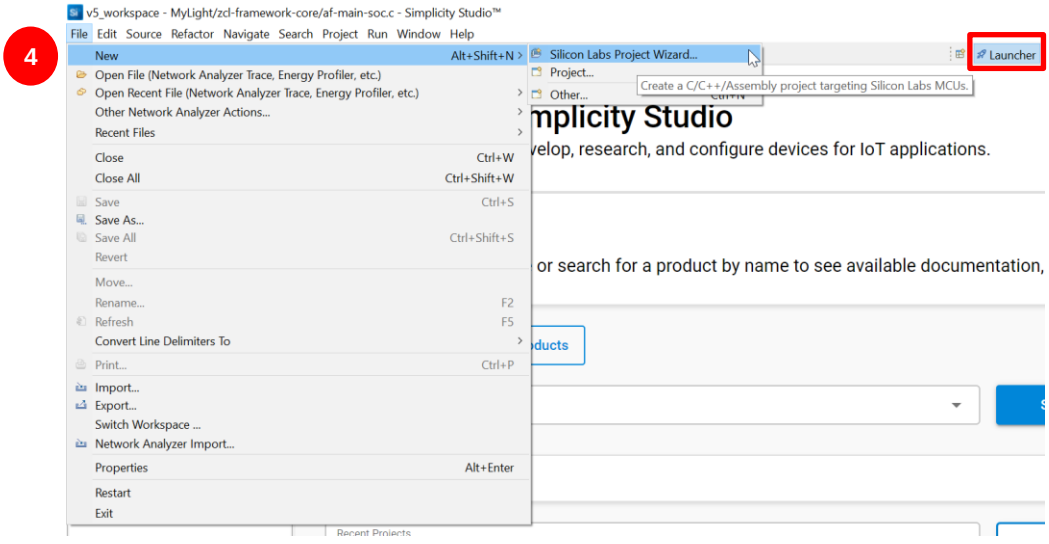
We want to create three OpenThread projects and then create a network:

- ✓ child project
- ✓ router_eligible_1 project
- ✓ router_eligible_2 project

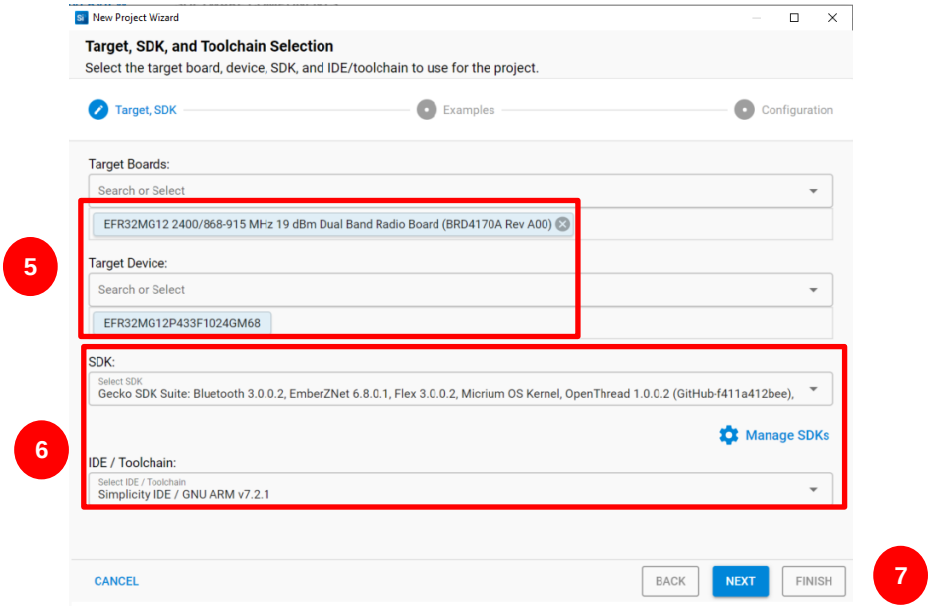
1. Launch Simplicity Studio from your desktop
2. Connect the **4170A** radio board with **EFR32MG12** to your PC using the USB cable
3. When the device is connected to your PC, you should see it listed in the **Debug Adapters** window in Simplicity Studio



4. File >> New >> Silicon Labs Project Wizard



5. If your radio board and WSTK are detected automatically, a new window shows up
6. Make sure you use Gecko SDK 3.1.0 or later and GNU ARM v7.X.X compiler
7. Click on **NEXT**



2.1 Minimal Thread Device project

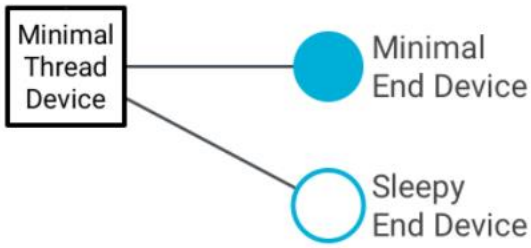
The node **child** project is created with *ot-cli-mtd* example. The child is linked with a parent device (leader, router). In a network the child can act in two different ways:

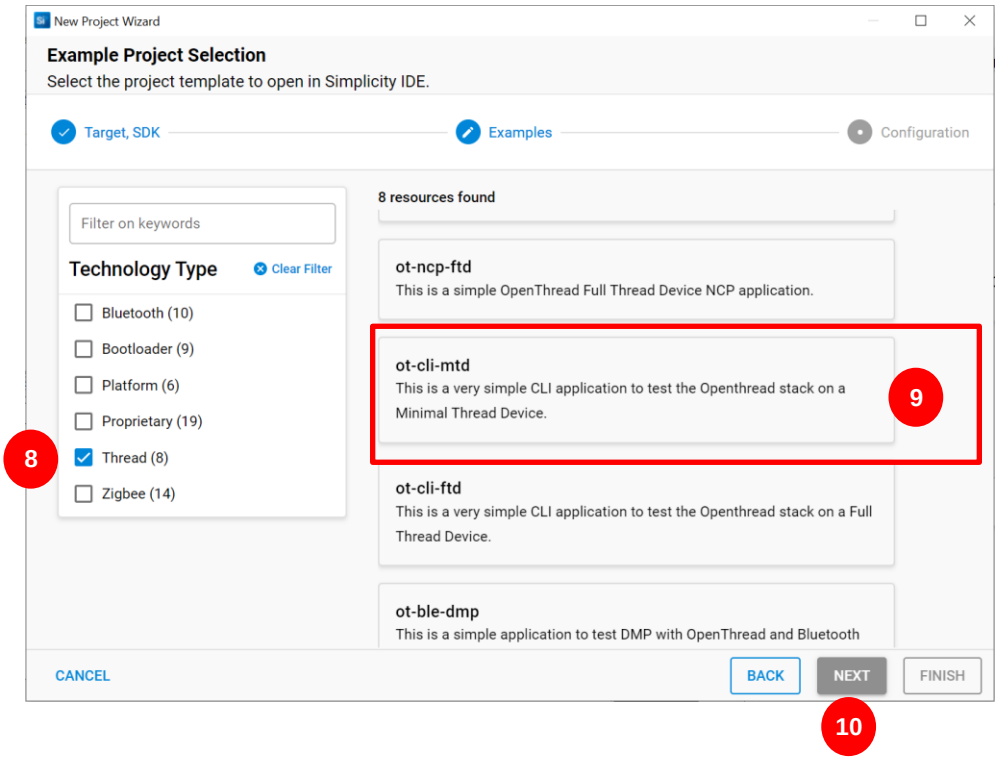
-As a **S**leepy **E**nd **D**evice (SED), in the software, we need to wake it up to check if there is a new message

-As a **M**inimal **E**nd **D**evice (MED), its radio device is always on, we do not need to poll to have an update.

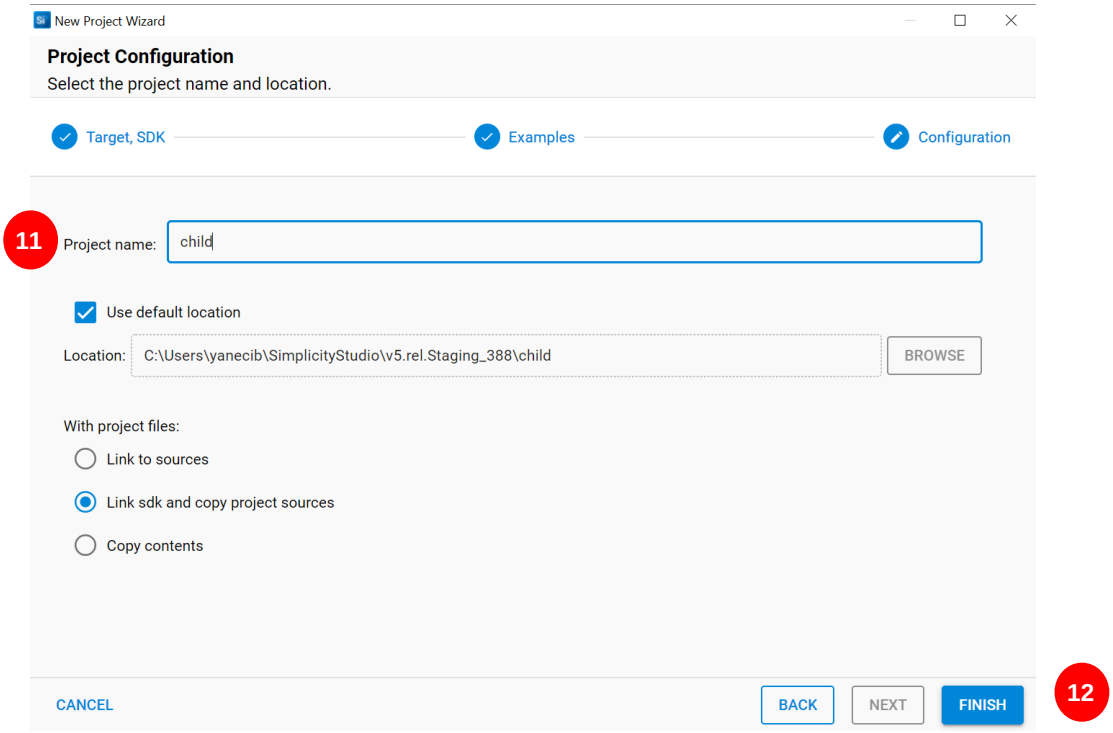
In our case, we will use it as a **MED**.

8. Now, we can tick the thread check box,
9. Select the *ot-cli-mtd* example only for the first node and click on next. The other two nodes *use ot-cli-ftd* example
10. Then click on **NEXT**



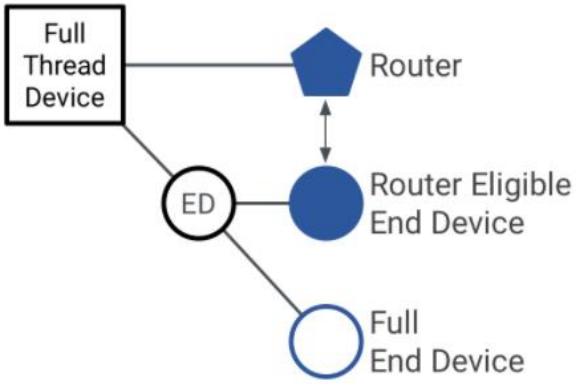


11. For *ot-cli-mtd*, enter the name “child”.
12. Click on **FINISH**

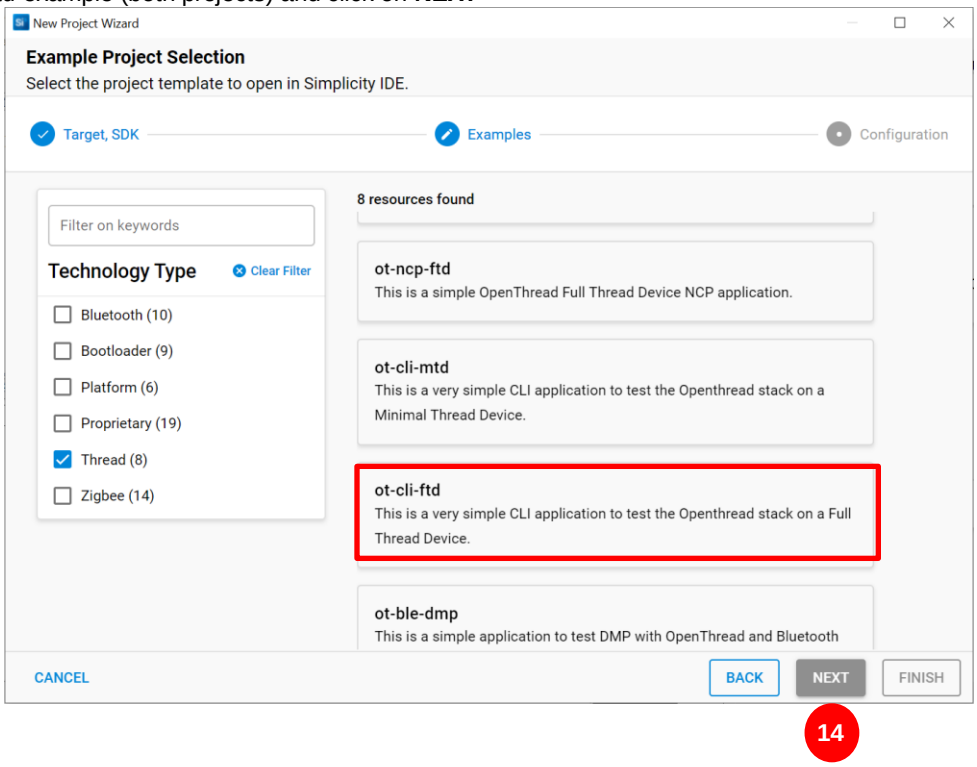


2.2 Full Thread Device Project

The **router_eligible_1** project is created with *ot-cli-ftd* example. The **router_eligible_1** will be the *leader* when the network first forms. The **router_eligible_1** can act as a router, **R**outer **E**ligible **E**nd **D**evice, or **F**ull **E**nd **D**evice.

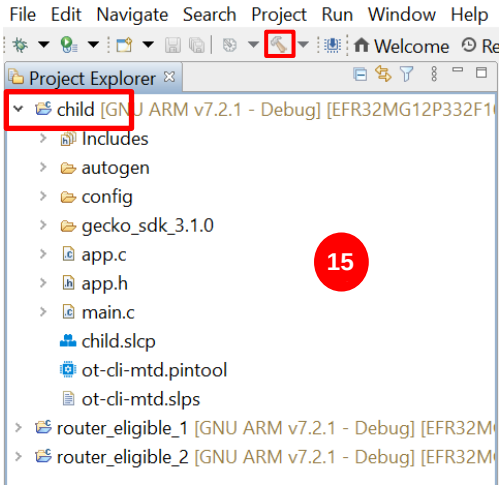


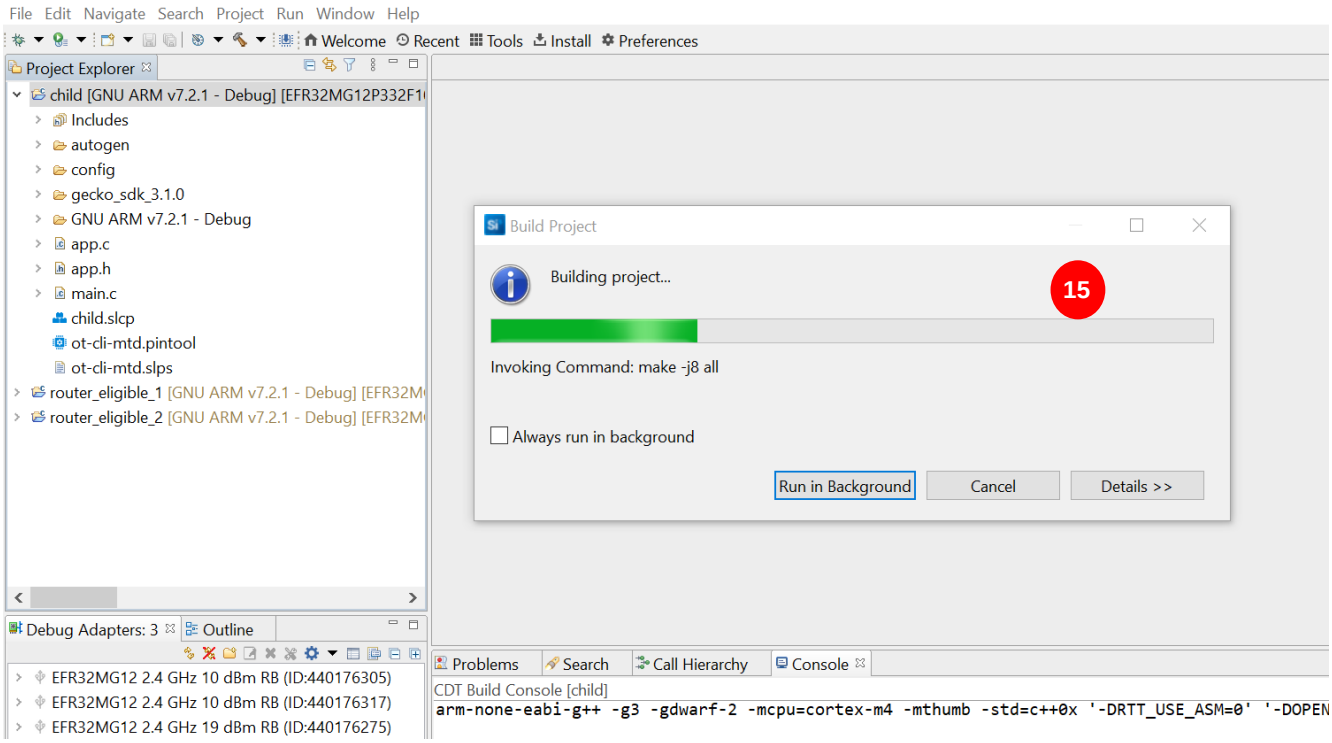
13. Repeat step_4_to_8_Create_an_OpenThread Project to create two projects**,** name the first one **router_eligible_1** and the other one **router_eligible_2**
14. Select the *ot-cli-ftd* example (both projects) and click on **NEXT**



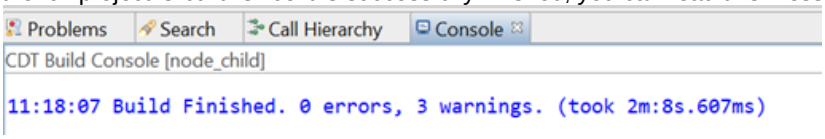
3 Compile and Flash Device Firmware

15. Select your **child** project and click on the **hammer** to compile your project. Repeat this step for the **router_eligible_1** and **router_eligible_2**





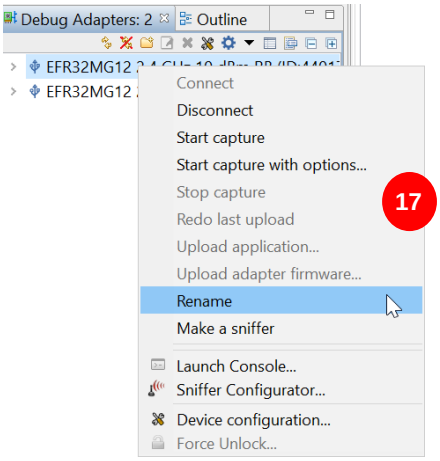
16. Wait a few minutes until the full project is built. Once it is successfully finished, you can read this message in the **Console** tab:

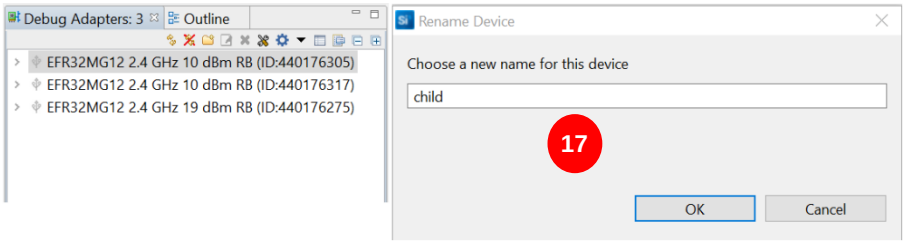


4 Node Debug setup

4.1 Rename Your Board

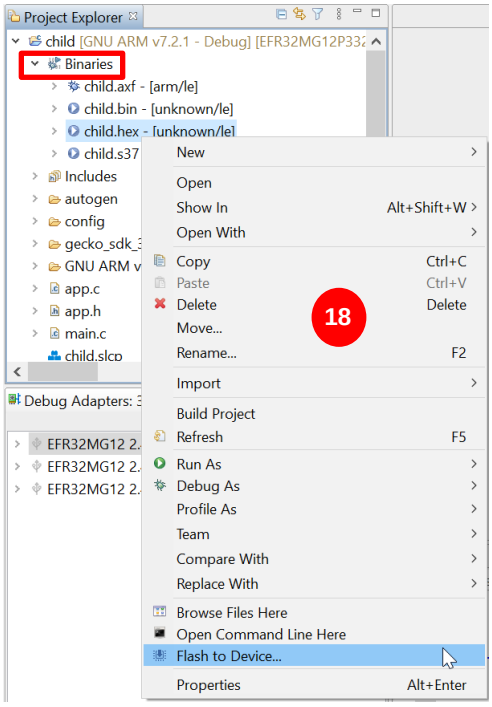
17. Rename your device to display each device role
    In **Debug Adapter review,** right-click on your device and select **Rename,** then a “Rename Device” window opens.



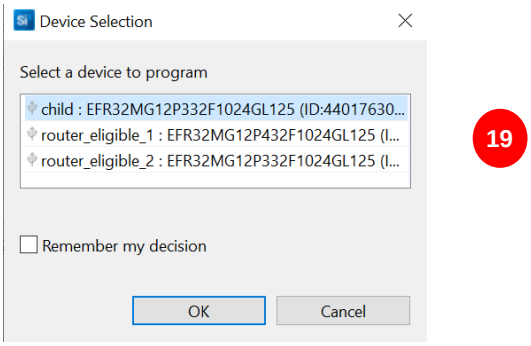


18. For each project, to flash the corresponding .hex file, expand the **Binaries folder** under your project**,** right-click on the **.hex,** and select **Flash to Device…**

Node Debug setup

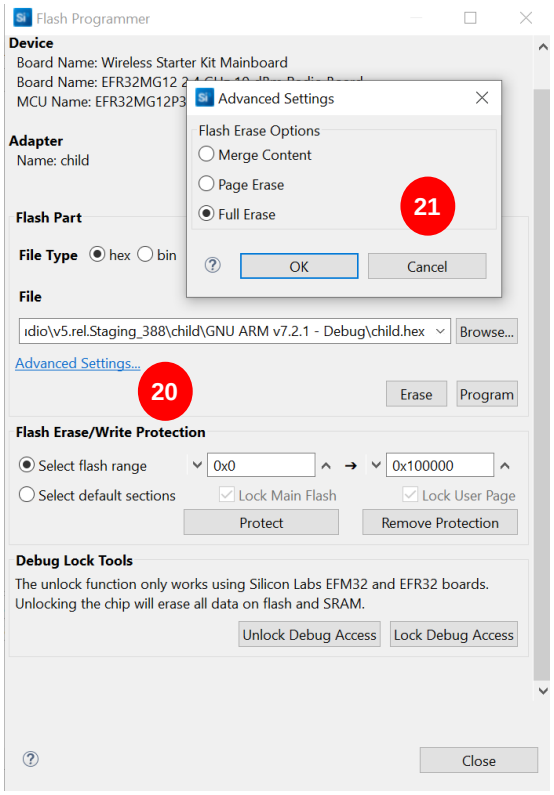


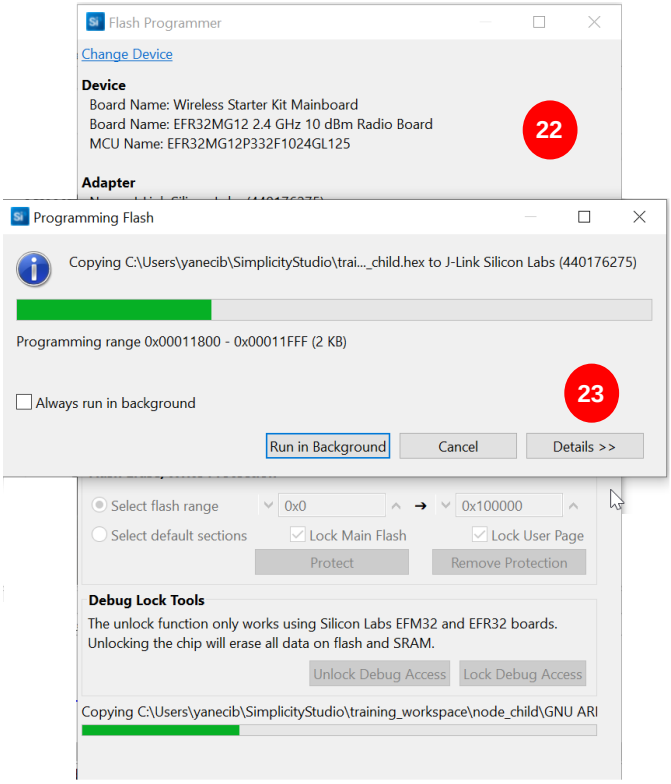
19. Select your device in the window and click **OK**



20. Click on **Advanced Settings**
21. **Advanced settings** window opens, click on **Full Erase** and **OK**
22. Then on the **Flash Programmer** window click on **Program**
23. A **Programming Flash** window will be opened, the firmware is being flashed.

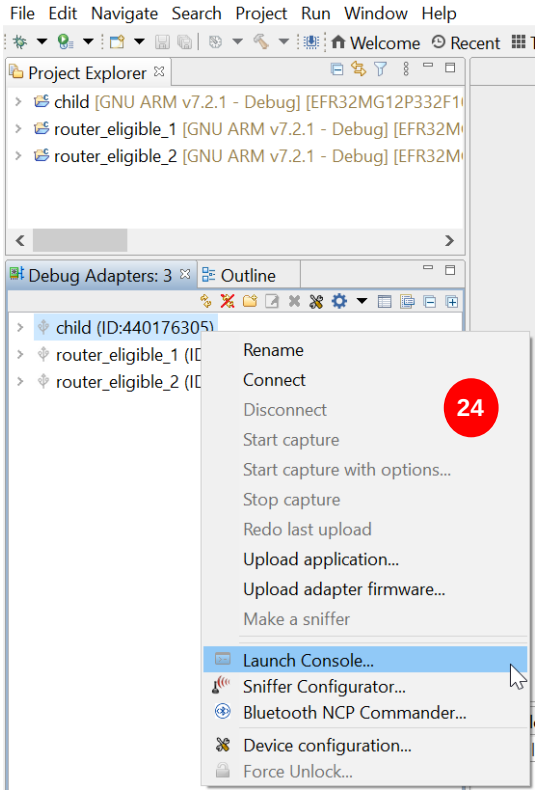
Node Debug setup



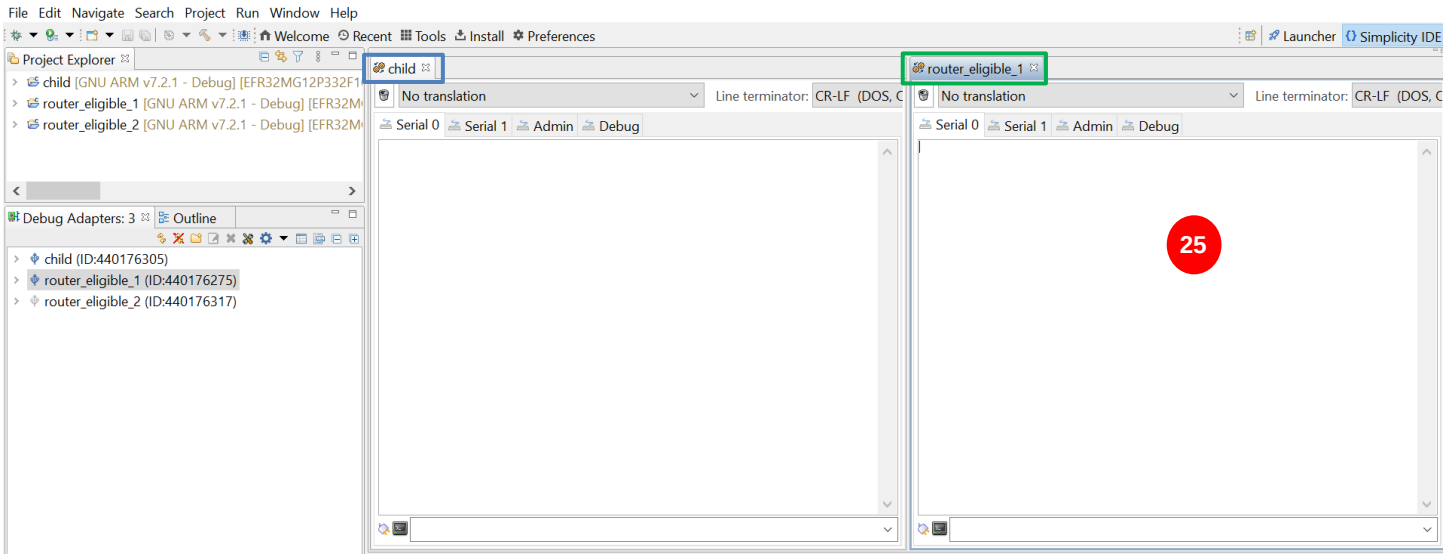


4.2 Open Console to Communicate with Your Board

24. Open a console for each project in the **Debug Adapter** window:
Right-click on the device and select **Launch Console…**

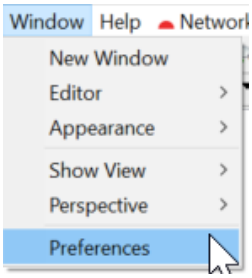


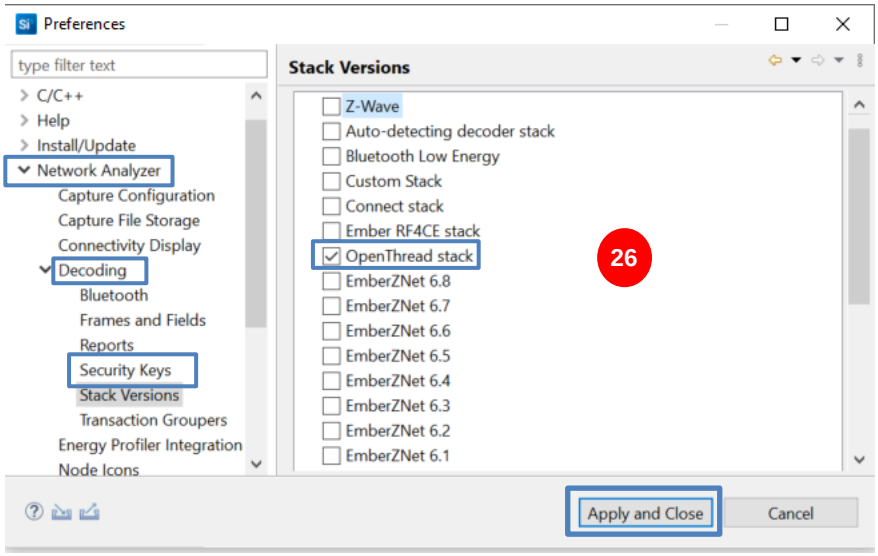
25. Placing the two console windows side by side, as shown in the picture below, can make it easier to switch your active console window from one node to another. We use **serial 1** to communicate with the device.



26. From the top bar menu, click on window and select **Preferences** >> **Network Analyzer** >> **Decoding** >> **Stack Versions**

Node Debug setup





5 Lab1: Create a Thread network

5.1 Create the Network

To create the network, we start with the **router_eligible_1.** From its console enter the commands lines below:

Router_eligible_1 console		
Command-line	Expected Response	Description
`> dataset init new`	`Done`	Create a new network configuration
`> dataset commit active`	`Done`	Commit new dataset to the Active Operational Dataset in non-volatile storage.
`> ifconfig up`	`Done`	Enable Thread interface
`> thread start`	`Done`	Enable and attach Thread protocol operation.
Wait about 10 seconds ….		
`> state`	`leader`	Read the current status : offline, disabled, detached, child, router, or leader

`> dataset`	`> dataset`
	`Active Timestamp: 1`
	`Channel: 15`
	`Channel Mask: 0x07fff800`
	`Ext PAN ID: fa8f8d5f1cd2b5e9`
	`Mesh Local Prefix:`
	`fd6e:c30c:24e9:e3d9::/64`
	`Master Key:`
	`97e05cb1f2df6f0d9af3026448f4b834`
	`Network Name: OpenThread-2136`
	`PAN ID: 0x2136`
	`PSKc: 47c8cd2bed340bab86c47929cf0e99db`
	`Security Policy: 0, onrcb`
	
	`Done`
		View network configuration

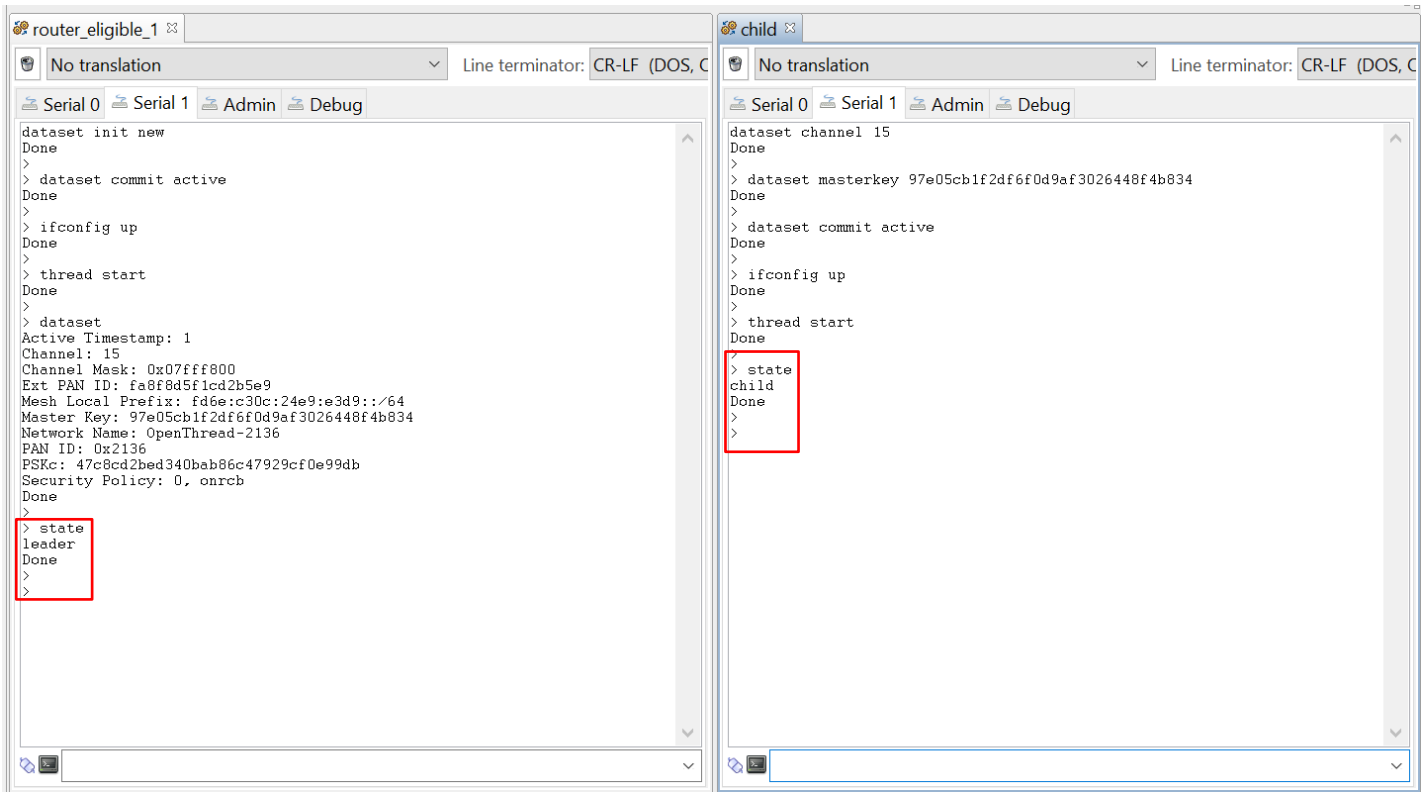
According to the response of the command *state* and *dataset*, the **leader** is created.

5.2 Add a Node to Our Network

To add a node to our network, enter the commands lines below in the **child** console:

Child console		
Command-line	Expected Response	Description
`> dataset channel 15`	`Done`	To decrease latency, the node needs to use the channel used in the alive network.
`> dataset masterkey` `97e05cb1f2df6f0d9af3026448f4b834`	`Done`	**Only the Master Key is required for a device to attach to a Thread network**
`> dataset commit active`	`Done`	Commit new dataset to the Active Operational Dataset in non-volatile storage.

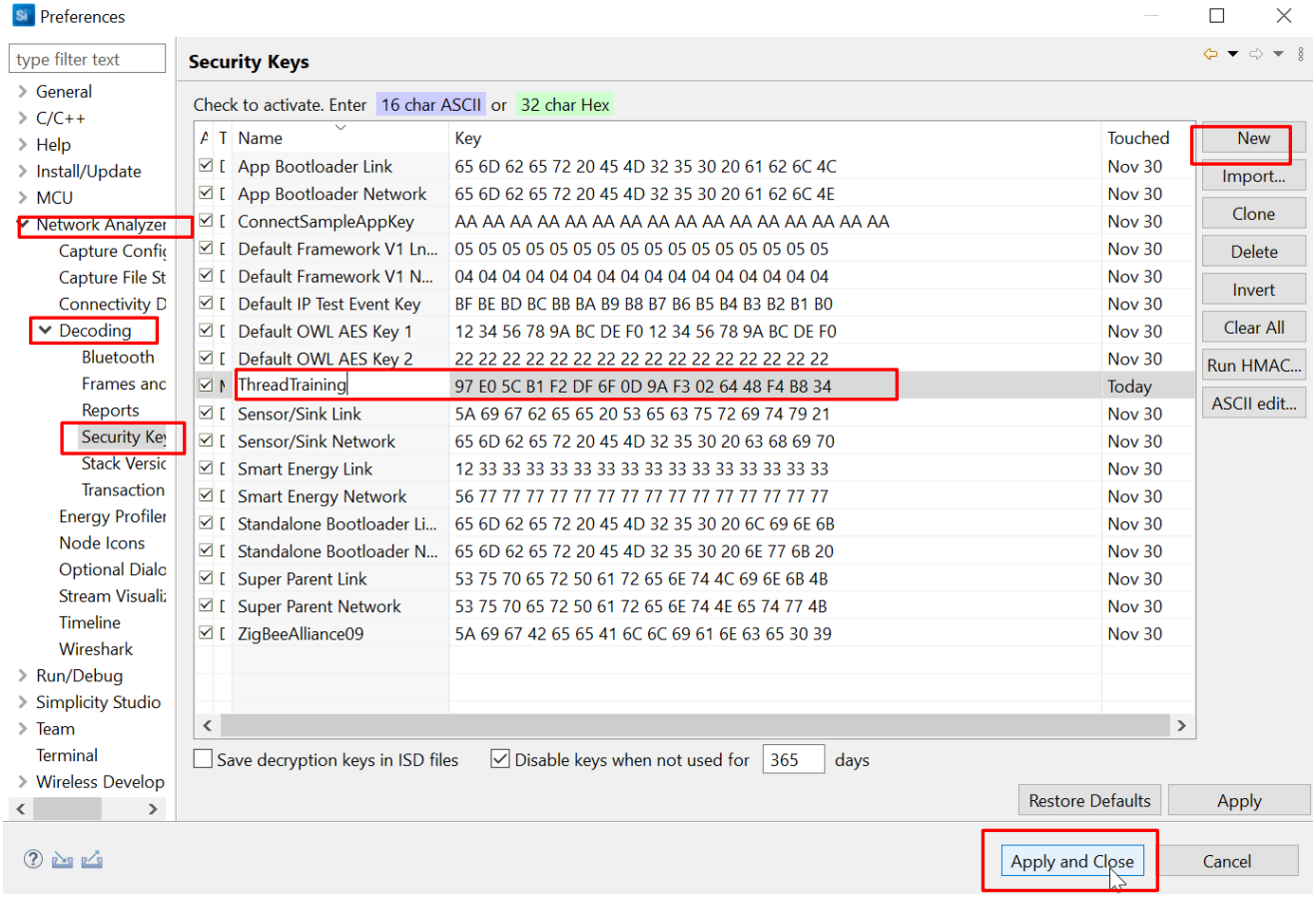
`> ifconfig up`	`Done`	Enable Thread interface
`> thread start`	`Done`	Enable and attach the Thread protocol operation.
Wait about 20 seconds ….		
`> state`	`child`	Read the current status: offline, disabled, detached, child, router, or leader



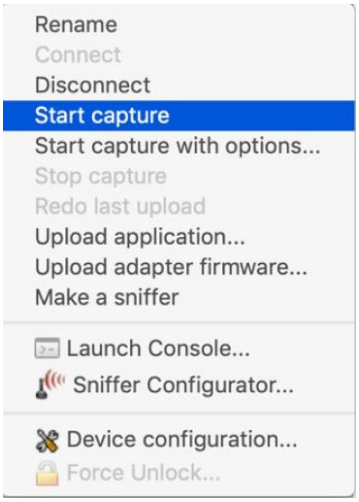
5.3 Explore Deeper in Our Network

After our network is created, we want to check the environment and use the Network Analyzer in Simplicity Studio

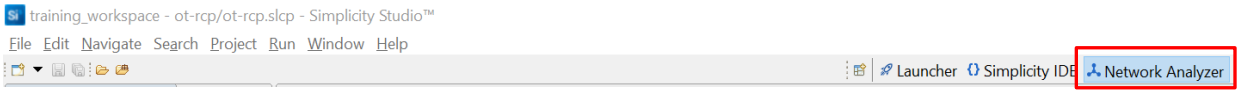
1. Enter the security key: Window >> Preferences



2. In the Debug adapter, right-click on **router_eligible_1** and select **Start capture,** repeat this step for the **child**.

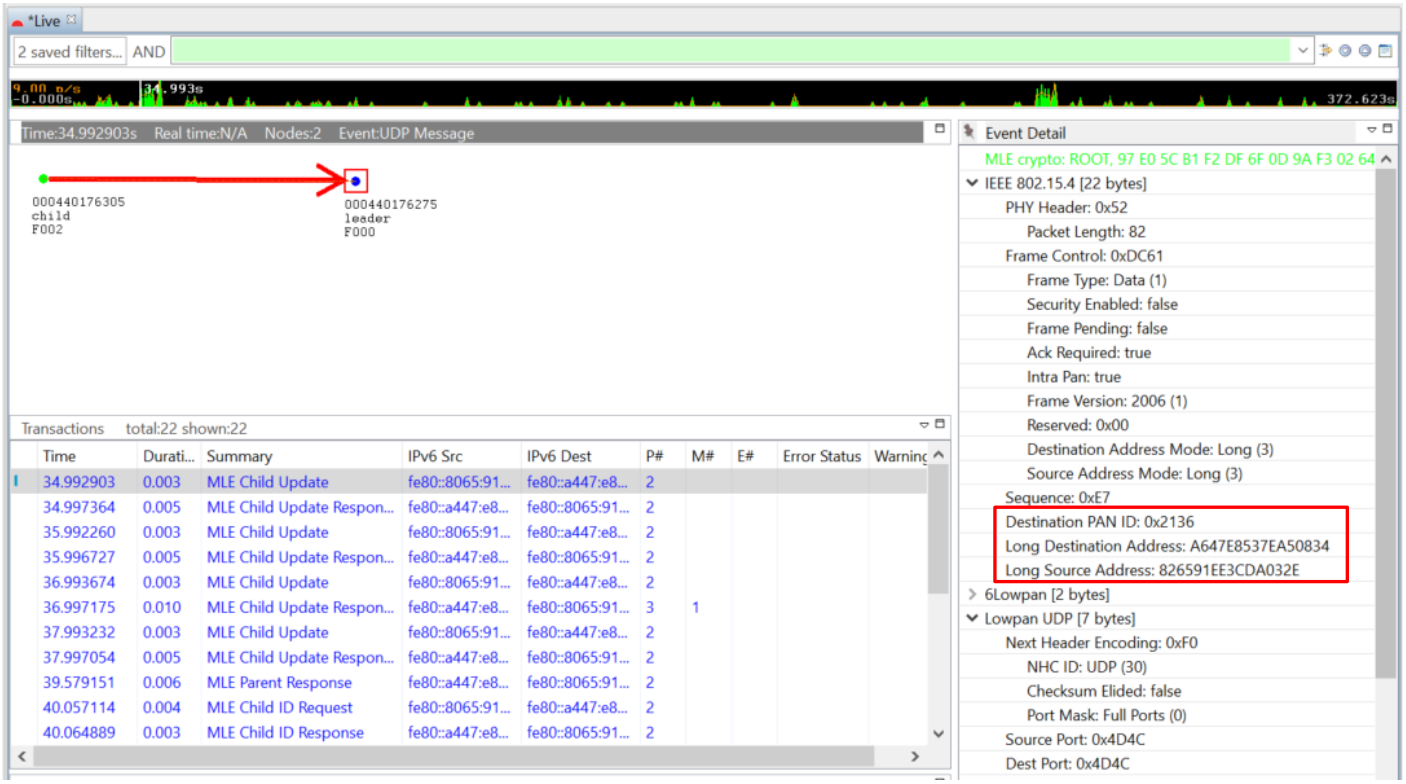


3. Click on **Network Analyzer** button to switch the view

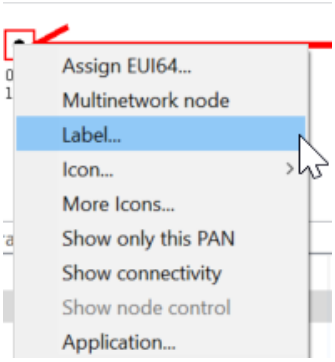


The network analyzer opens, feel free to set up your consoles as below to have a panoramic view.

-Highlight a network transaction such as the “MLE Child Update” shown in the example below, and in the “Event Detail” window you can find the PAN ID of your network. In my example, the PAN ID is 0x2136. Yours may be different.



You can also rename your device represented in the upper view: right-click on it and choose **label**, then enter the desired name



5.4 A Quick Look at Our IP Addresses

Router_eligible_1 console		
Command line	Response	Description

`> ipaddr`	`> ipaddr`
	`fd6e:c30c:24e9:e3d9:0:ff:fe00:fc00`
	`fd6e:c30c:24e9:e3d9:0:ff:fe00:f000`
	`fd6e:c30c:24e9:e3d9:fb6d:7770:b132:5823`
	`fe80:0:0:0:a447:e853:7ea5:834`
	`Done`
		Display the ipv6

Child console		
Command line	Response	Description
`> ipaddr`	`> ipaddr`	Display the ipv6

	fd6e:c30c:24e9:e3d9:0:ff:fe00:f001 fd6e:c30c:24e9:e3d9:6fa:ecc5:34b1:27c5 fe80:0:0:0:8065:91ee:3cda:32e Done	

5.4.1 Link-Local addresses (LLA)

This address starts with fe80::/16 prefix (for instance `fe80:0:0:0:a447:e853:7ea5:834`), it is created with the MAC address. It is not used to communicate between nodes. We can still use them between two nodes if there is only a link, one radio transmission, not more than one cable to retransmit the message.

5.4.2 Example of building LLA (router_eligible_1):

Mac address:	A647:e853:7ea5:834
Flip the seventh bit	A6 = 1010 0110 A4 = 1010 0100
LLA	FE80::A447:e853:7ea5:834

5.4.3 Routing Locator Address (RLOC)

`fd6e:c30c:24e9:e3d9:0:ff:fe00:f000` (**Router_eligible_1**), this address is created when the device is attached to the network, and is generally not used by applications. The blue part is the mesh prefix. This address changes if the topology changes, in other words, if you remove/add a device.

5.4.4 Mesh Local Address (ML-EID)

`fd6e:c30c:24e9:e3d9:fb6d:7770:b132:5823` (**Router_eligible_1**), this address is independent of the network topology, and is used to communicate with the other interface in the same thread network.

5.4.5 Anycast (only the leader)

`fd6e:c30c:24e9:e3d9:0:ff:fe00:fc00` (**router_eligible_1**), this is anycast address, it is used to route traffic to a Thread interface when the RLOC of a destination is not known. An Anycast Locator (ALOC) identifies the location of multiple interfaces within a Thread partition. The last 16 bits of an ALOC, called the ALOC16, is in the format of 0xfcXX, which represents the type of ALOC.

ALOC16	Type
0xfc00	Leader

To sum up, only the leader has an anycast address, the LLA is built from the MAC address, the ML-EID is created when the network is up and does not change, the RLOC changes if the topology of the network changes as well. To communicate between the interface of the same mesh network, we use the ML-EID. To identify an interface we use RLOC. Here is a link to study deeper IPV6 addressing: https://openthread.io/guides/thread-primer/ipv6-addressing

Nature	Communication	Description	Comments
LLA	A----------B	Yes, both direction	Point to point
	A----------B----------C	A to B yes, both direction B to C yes, both direction A to C no, any direction	Only one radio transmission, useful to discover the neighbor or routing.
RLOC		A to B yes, both direction B to C yes, both direction A to C yes, any direction	If you change the topology, the address change.
ML-EID			Any interface of the same mesh network, so here my address which starts with `fd6e:c30c:24e9:e3d9`

5.4.6 Zoom in on the "child table" command line

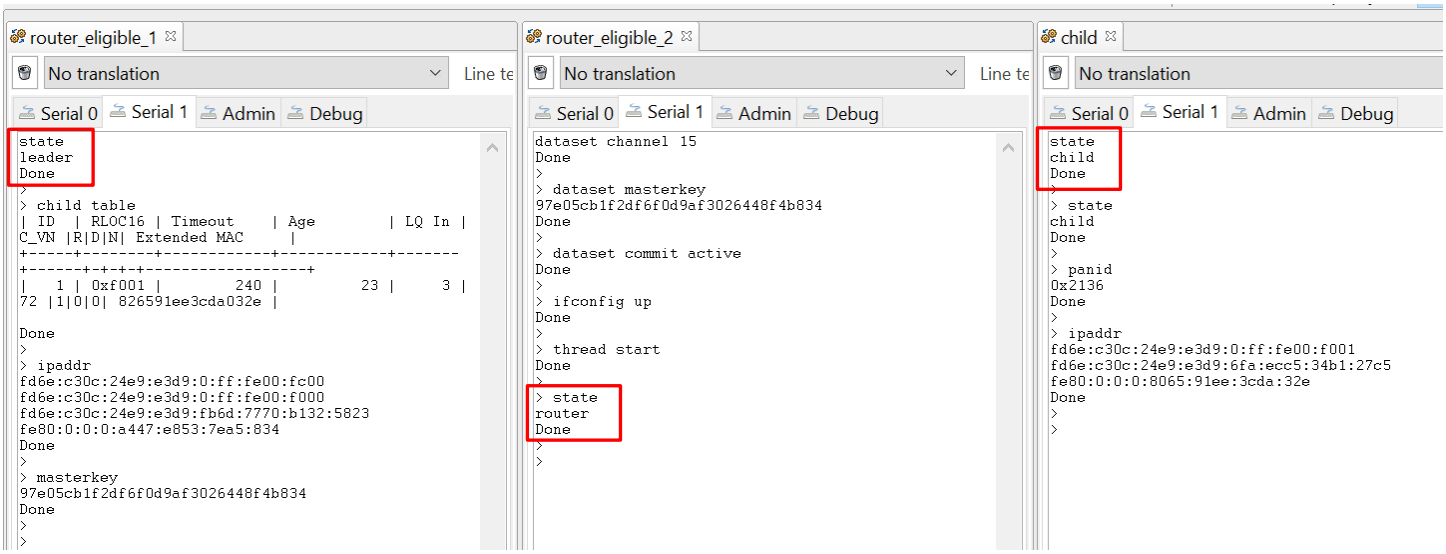
Router_eligible_1 console		
Command-line	Example of an Expected Response	Description

`> child table`	`| ID | RLOC16 | Timeout| Age | LQ In| C_VN|R|S|D|N| Extended MAC |`
	`+-----+--------+--------+-----+------+-----+-+-+-+-+------------------+`
	`| 1 | 0xf001 | 240 | 38 | 3 | 47|1|1|0|0| 826591ee3cda032e |`
	`Done`
		Attached child info

5.5 Understanding Leader, Router, and Child Roles

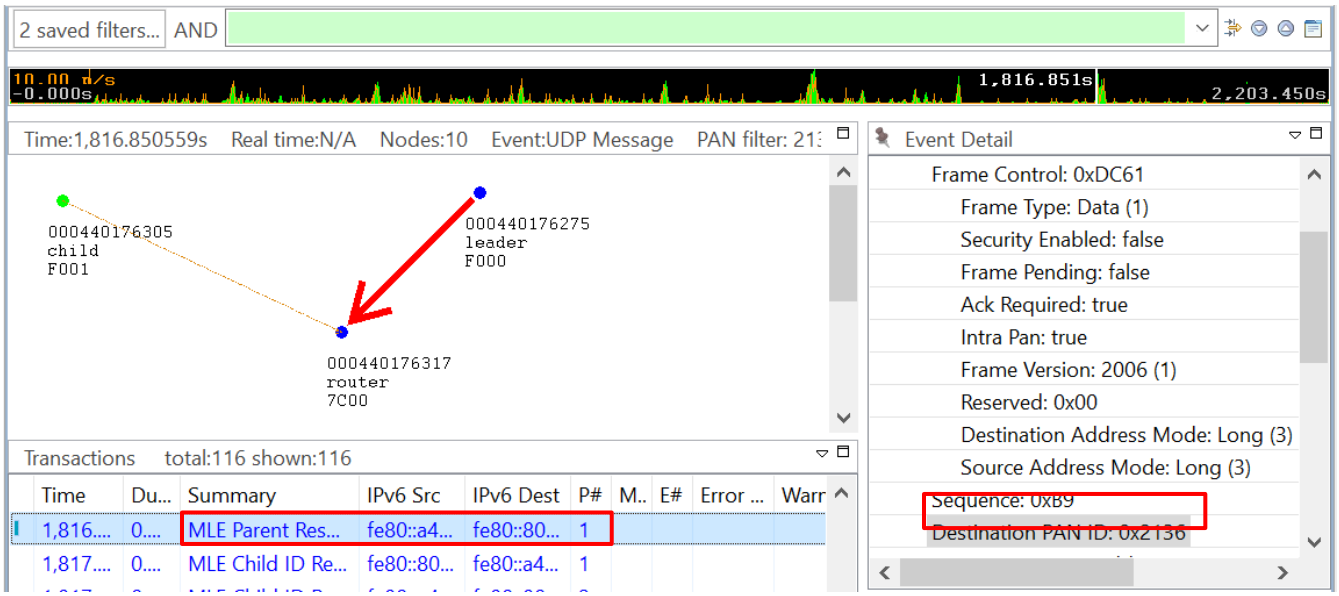
We want to add a device and then remove the current leader to understand some of the healing properties of Thread networks.

Launch the console of the **router_eligible_2** and repeat command lines in the section Add a Node to Our Network**.**



router_eligible_2 starts as a child and becomes quickly a router.

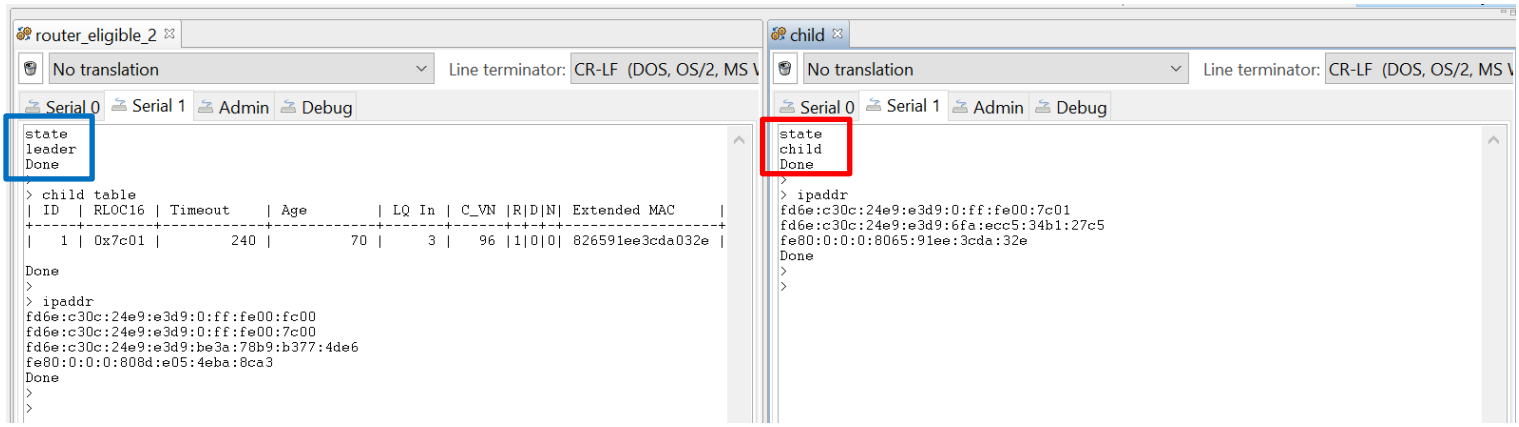
Then we can check the capture in the **Network Analyzer** view.



The image above shows communication between the **leader (router_eligible_1)** and the **router** (**router_eligible_2**)

5.6 Remove a leader

We want to discover what happens if we remove the current leader. Unplug the **router_eligible_1** node, wait three minutes and, from each remains nodes enter this command: *state*



Once the **router_eligible_1** is removed, the **child** has a new parent, the router (**router_eligible_2**) became the new **leader**, as we can see, in the child table of **router_eligible_2**, there is the extended mac address of the **child**. We can notice some differences for the **child**:

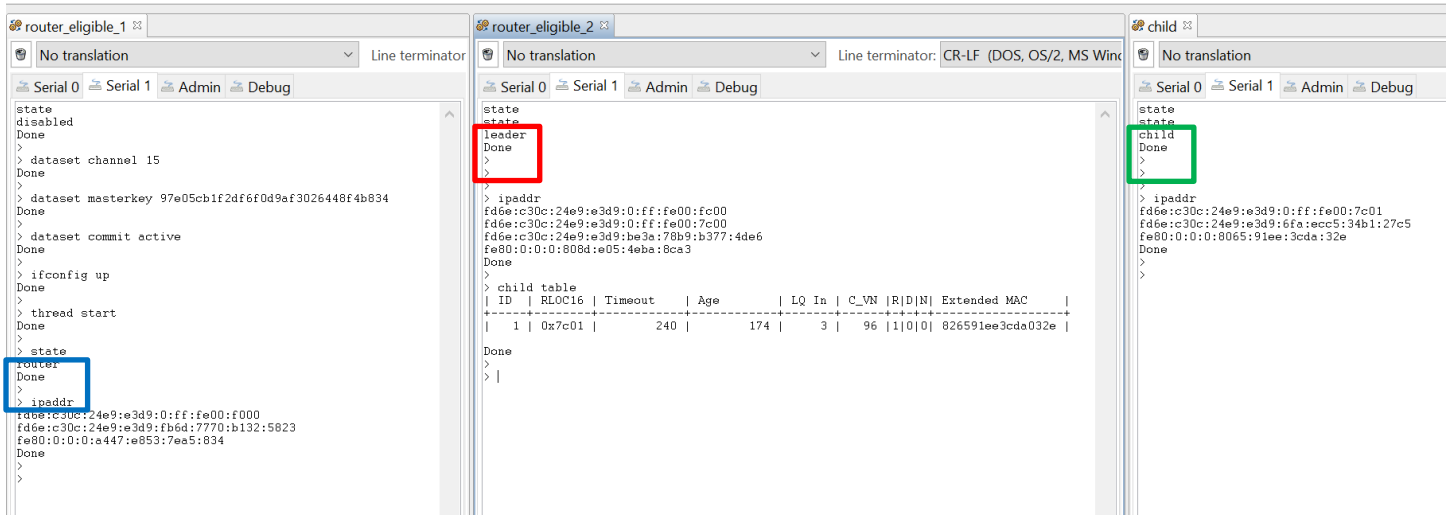
-Only the local address did not change

-Its **RLOC16** value changed: 0x7c01

5.7 Put router_eligible_1 back

Plug it back in, open its console and display its state

We need to add it to our network, repeat command lines in the section Add a Node to Our Network**.**



The **router_eligible_1** became a **router**; the **child** did not change and **router_eligible_2** is still the leader. We notice that **Router_eligible_1** does not have anycast address anymore.

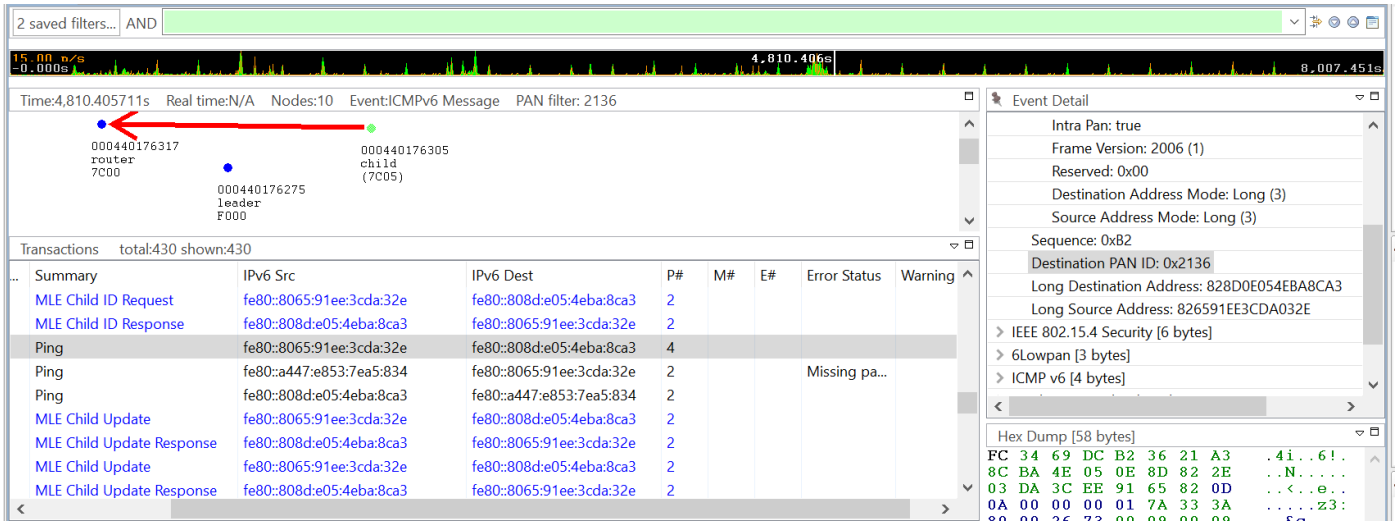
5.8 Communication between Nodes

The ping command enables to send a request and check if the communication works, let's have a ping between the leader and the child:

5.8.1 Leader (router_eligible_2) and child ping each other

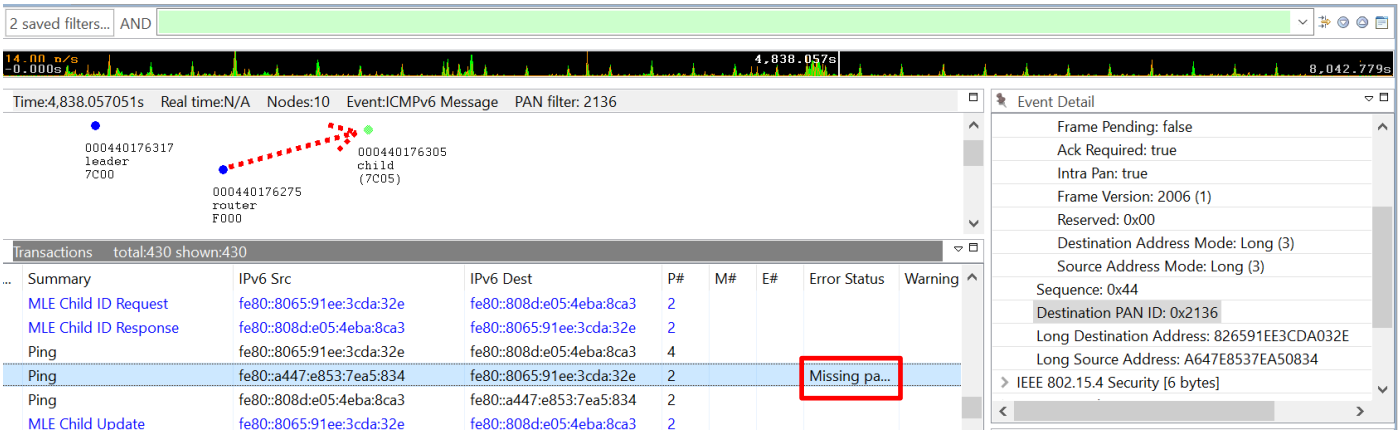
Router_eligible_2 console (ping local address)		
Command line	Response	Description

`> ping`	`ping fe80:0:0:0:8065:91ee:3cda:32e`
	`Done`
	`> 16 bytes from fe80:0:0:0:8065:91ee:3cda:32e: icmp_seq=2 hlim=64 time=11ms`
	`ping fe80:0:0:0:a447:e853:7ea5:834`
		Send an ICMPv6 Echo Request



5.8.2 The child and the router (router_eligible_1) cannot ping each other by their local addresses

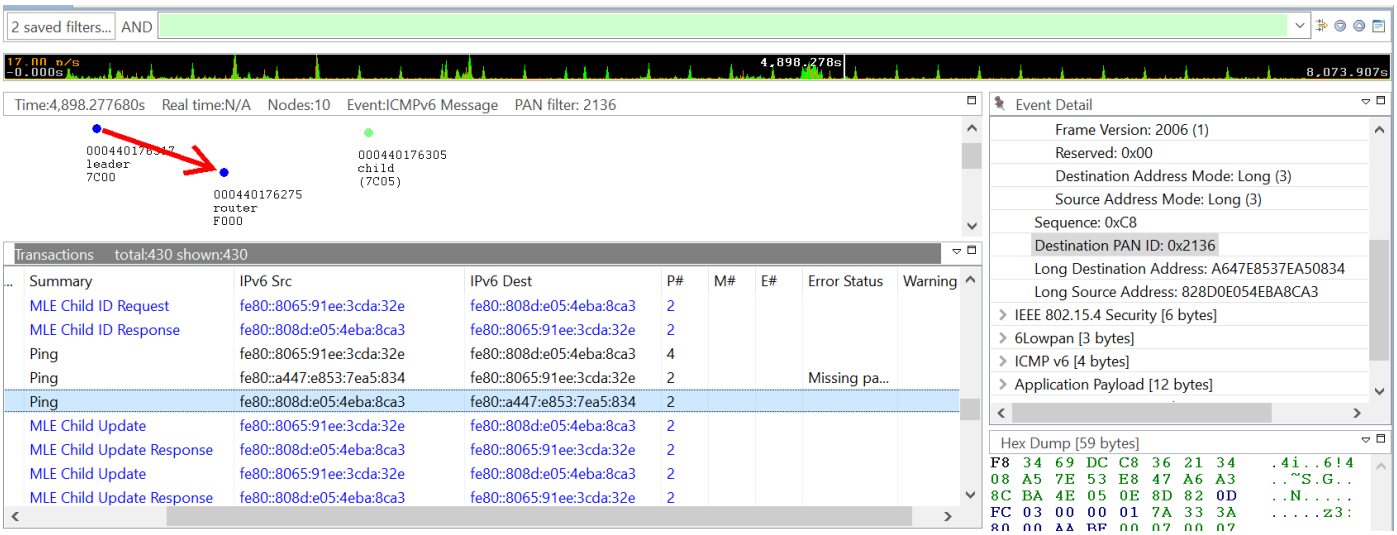
child console (ping local address)		
Command line	Response	Description
`> ping`	`ping fe80:0:0:0:a447:e853:7ea5:834` `Done`	Send an ICMPv6 Echo Request



5.8.3 Leader (router_eligible_2) and the router (router_eligible_1) can ping each other by their local addresses

Child console (ping local address)		
Command line	Response	Description

`> ping`	`ping fe80:0:0:0:a447:e853:7ea5:834`
	`Done`
	`>`
	`> 16 bytes from fe80:0:0:0:a447:e853:7ea5:834: icmp_seq=6 hlim=64 time=12m`
		Send an ICMPv6 Echo Request

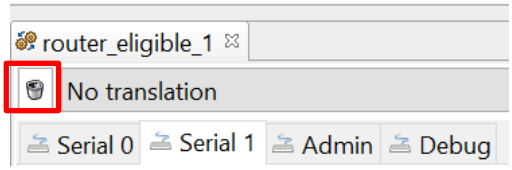


The **ping** request from the **child** to **leader** or leader to the **router** works with the local IPV6 address since there is only one hop!

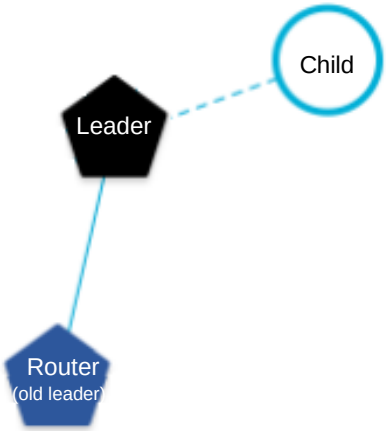
Let's take a look at the network analyzer trace., There is a "*Missing packet"* which means a message is missing.

The prefix has a 64 bits length, thread uses IPV6 native address. The MSB 64 bits for the network ID and the LSB 64 bits for the ID device.

We added back **router_eligible_1**, so the topology changed as the RLOC16 of the child too. Once **router_eligible_1** is attached to the network, refresh the console (click on the bin icon) and enter the **rloc16** command to see: 0x7C05.



According to the result of the ping response, we can draw our network topology:



6 Lab2: OpenThread Border Router (Raspberry Pi 3b/b+)

6.1 Prepare your Raspberry Pi

Once you get the image OT_FAE_Training_2020_Labs.img, you have to burn it on an SD card.

I recommend using this software to flash your SD card *balenaEtcher-Setup-1.5.100*.

6.2 Prepare 4170A radio board

The OTBR is built with two hardware platforms:

- A host processor hosts the core of OpenThread and the applications, in our case, this is Linux with OpenThread core

- A device controller with the minimal MAC layer: in our case 4170A board with *ot-rcp* runs on it.

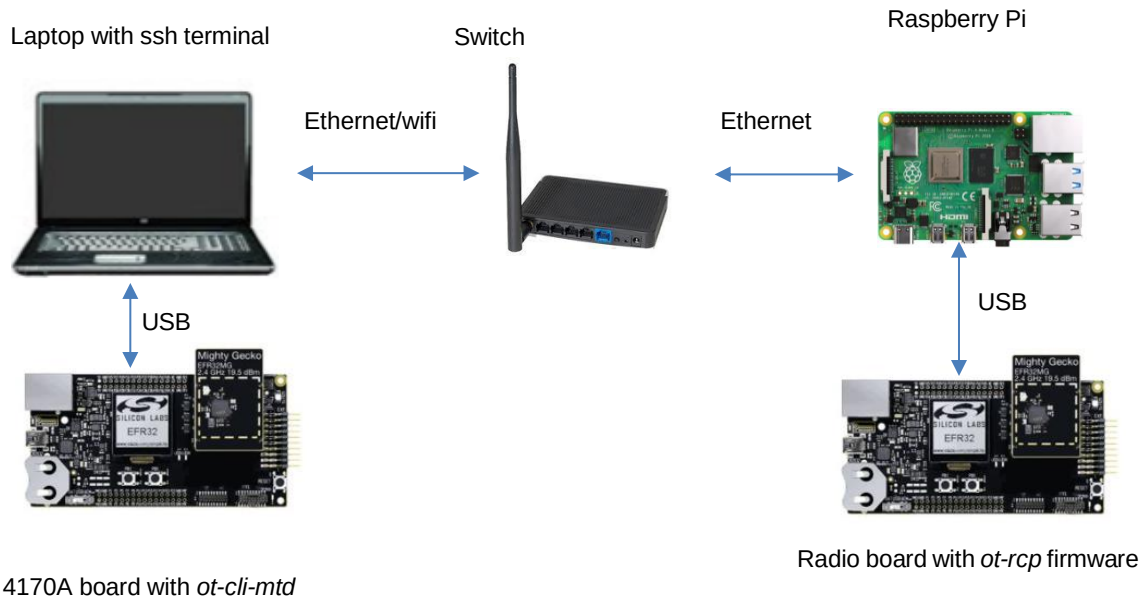
Create an OpenThread project from Simplicity Studio v5 (section_4_to_8_Create_an_OpenThread Project) and choose *ot-rcp* example, compile it and flash **router_eligible_1**.

6.3 Prepare a node device

Use your **child**, it has already the corresponding firmware.

6.3.1 OpenThread Border Router setup

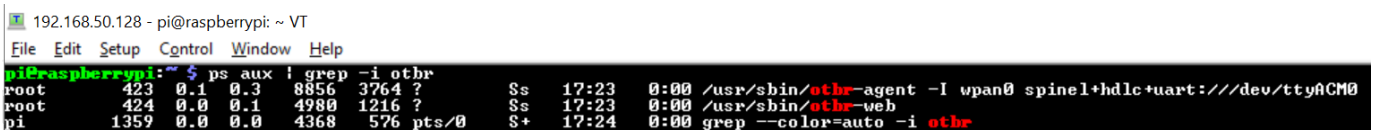
At this step you should have this configuration:



6.4 Start the OTBR (OpenThread Border Router)

Once your Raspberry Pi boots,

1. The command line below shows if **otbr-agent** and **otbr-web** processes are running. The expected result is below.



2. Go to *ot-br-posix* directory: *cd ot-br-posix*
3. Run the script `sudo./startOTBR_Lab2.sh`

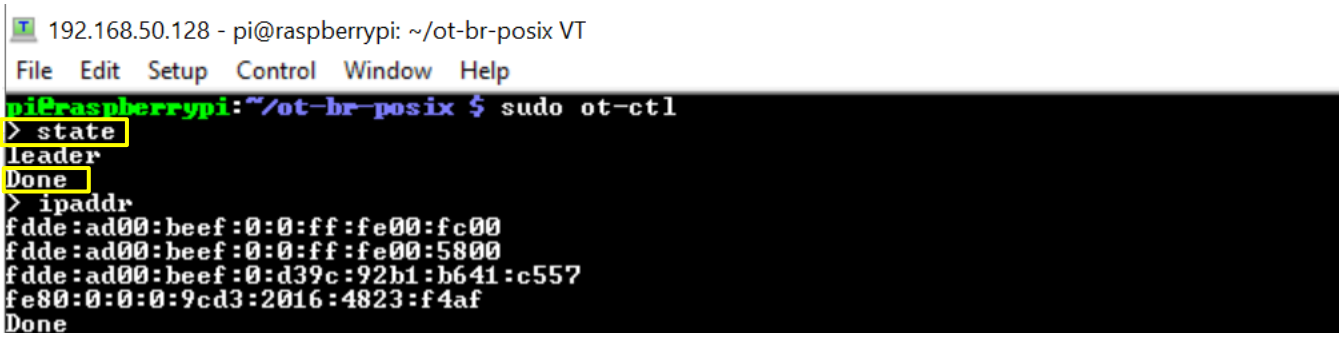
The table below sums up some command lines to setup the OTBR.

Leader console (startOTBR_Lab2.sh**)**		
Command-line	Expected Response	Description
`> factoryreset`		Delete all stored settings, and signal a platform reset
`> thread stop`	`> Done`	Disable Thread protocol operation and detach from a Thread network.
`> networkname SL-OpenThread`	`> Done`	Set the Thread Network Name.

6.4.1 OTBR info

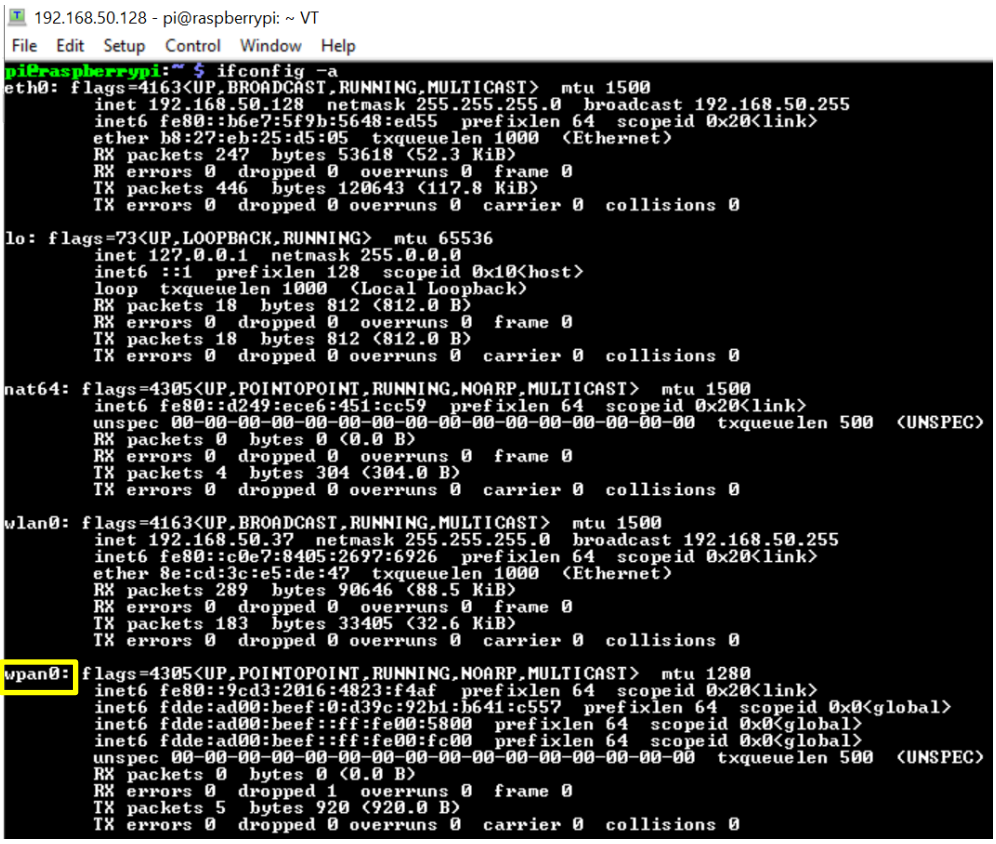
We can enter commands at the Raspberry Pi terminal:

Leader console from Raspberry Pi (sudo ot-ctl)		
Command line	Response	Description
`> sudo ot-ctl`	`>`	Enter in thread terminal



To leave the Thread host terminal hit Ctrl+c keys. Now have a look at our network interface.

The *ifconfig* Linux command line displays the network interface available.



wpan0 interface was added once the script *startOTBR_Lab2.sh* was executed successfully

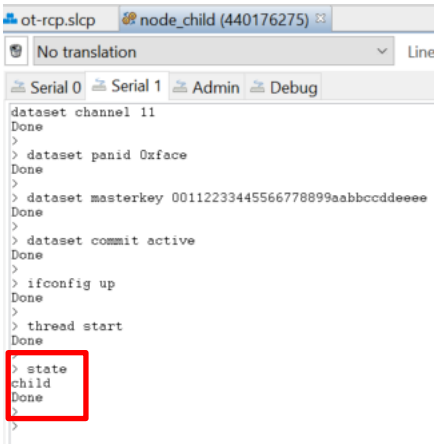
6.4.2 Add a node to our network (out-of-band method)

With the out-of-band method, we know all security information and add the node manually. In the “real world”, we use a user interface via the web or mobile app to add the node automatically.

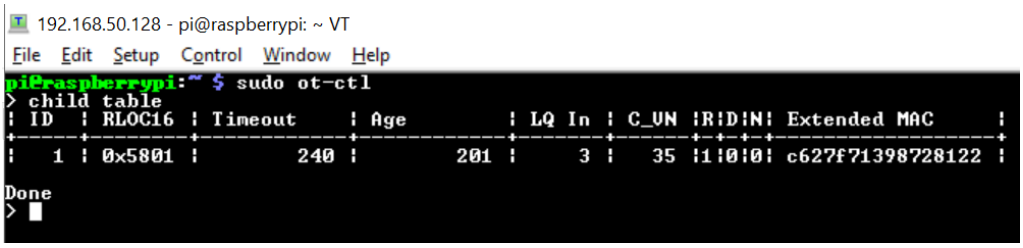
At the end of the script, you can read all the necessary data to add an end device.

```
On the client, you will need these commands:
dataset channel 11
dataset panid 0xface
dataset masterkey 00112233445566778899aabbccddeeee
dataset commit active
ifconfig up
thread start
```

From the node, console Add a Node to Our Network the node should join the network properly.



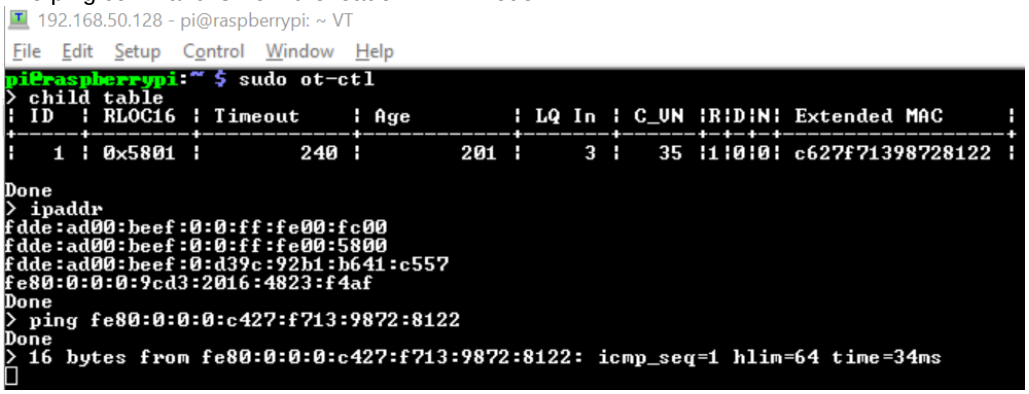
Go back to Raspberry Pi Thread host terminal and check if the **child** is present in the child table:



So, the child was added successfully, let’s ping each other.

6.4.3 Communication between OTBR and the node

The ping command is from the leader ===➔ node



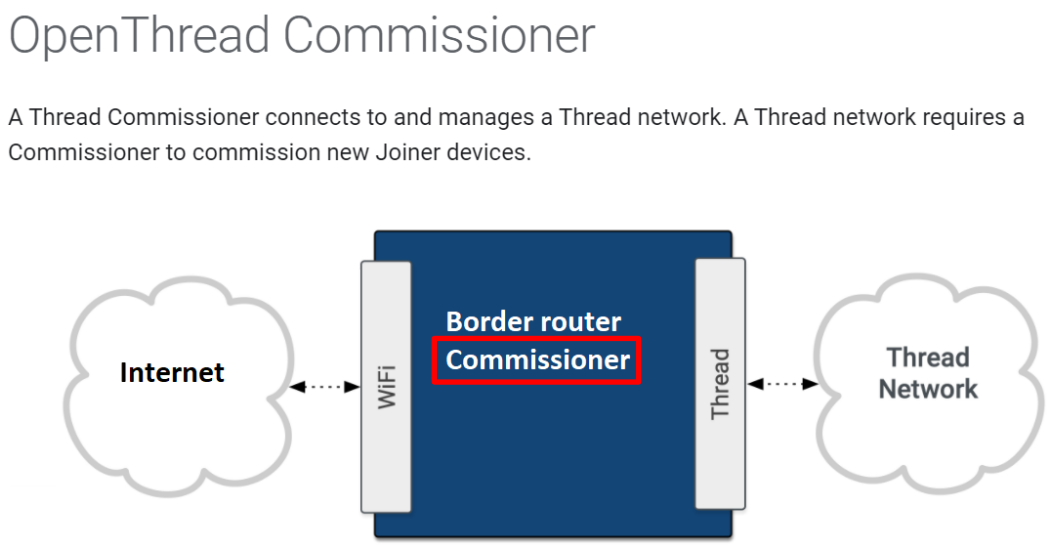
Go back to your child console, ping from the node ===➔ leader

```
> ipaddr
fdde:ad00:beef:0:0:ff:fe00:5801
fdde:ad00:beef:0:452e:1161:341a:fca4
fe80:0:0:0:c427:f713:9872:8122
Done
>
> ping fe80:0:0:0:9cd3:2016:4823:f4af
Done
>
> 16 bytes from fe80:0:0:0:9cd3:2016:4823:f4af: icmp_seq=1 hlim=64 time=32ms
```

Communication through the local address works on both sides.

7 Demo

In this demonstration, the user will discover the commissioning with a user interface command line. The commissioner application is embedded in the Raspberry Pi contrary to the external commissioner, which is a web or mobile application. All steps will be described, I will show you how to automatize all steps.



We need an OTBR and another node, the software/hardware configuration is the same as the Lab2.

Restart your Raspberry Pi, make sure the node has *ot-cli-mtd* firmware.

Go to ot-commissioner folder: *cd ot-commissioner*

Execute the following script: *sudo ./scriptCommissionDemo.sh*

All-new command lines are described below (all characters between “${}” are script bash variable):

Demo

```
pi@raspberrypi:~/ot-commissioner $ sudo ./scriptCommissionDemo.sh
ot-ctl factoryreset

sleep 3
ot-ctl thread stop
Done

sleep 1

ot-ctl ifconfig down
Done

sleep 1

ot-ctl panid 0xface
Done

ot-ctl extpanid dead00beef00cafe
Done

ot-ctl masterkey 00112233445566778899aabbccddeeee
Done
ot-ctl channel 11
Done

ot-ctl networkname SL-OpenThread
Done
****SL-OpenThread
****dead00beef00cafe
 SL-OpenThread J01NME dead00beef00cafe
Done
****pskc is dbc1dc72993de43bfb2bd5f853d3ec63

ot-ctl ifconfig up
Done
ot-ctl prefix add 2001:dce:1:ffff::/64 pasor
Done
ot-ctl thread start
Done
ot-ctl state
leader
Done
dbc1dc72993de43bfb2bd5f853d3ec63
Done
 /usr/local/etc/commissioner/non-ccm-config.json dbc1dc72993de43bfb2bd5f853d3ec63
File name modified is /usr/local/etc/commissioner/non-ccm-config.json
New PSKc is dbc1dc72993de43bfb2bd5f853d3ec63
```

Leader console from Raspberry Pi (sudo ot-ctl)		
Command-line	Expected Response	Description
`> ot-ctl extpanid dead00beef00cafe`	`> Done`	Set the Thread Extended PAN ID value.
`> ot-ctl pskc -p J01NME ${EXTPANID} ${NETWORKNAME}`	`> Done`	Generate a hex-encoded PSKc by using a Passphrase (Commissioner Credential), the extpanid, and the Network Name with the PSKc Generator tool on the OTBR. **Make sure to use the same Extended PAN ID and Network Name that was used in the operational dataset:**
`ot-ctl prefix add 2001:dce:1:ffff::/64 pasor`		Add a valid prefix to the Network Data.

		In other words, the OTBR and the node bust have the same network ID, to be on the same network.
`ot-ctl pskc`	`> Done` `dbc1dc72993de43bfb2bd5f853d3ec63`	Display security key
`./updatePSKc.sh ${PATH_PSKC} ${PSKC}`		Replace PSKC in `/usr/local/etc/commissioner/non-ccm-config.json` and change the `PSKc`

We can see our new pskc in `/usr/local/etc/commissioner/non-ccm-config.json` the command line below can show quicker the result:

```
cat /usr/local/etc/commissioner/non-ccm-config.json | grep -i pskc
```

```
pi@raspberrypi:~/ot-commissioner $ cat  /usr/local/etc/commissioner/non-ccm-config.json | grep -i pskc
    "PSKc" : "dbc1dc72993de43bfb2bd5f853d3ec63"
```

Now we can start our commissioner application:

Open a new terminal from your Raspberry Pi and enter the followings commands below:

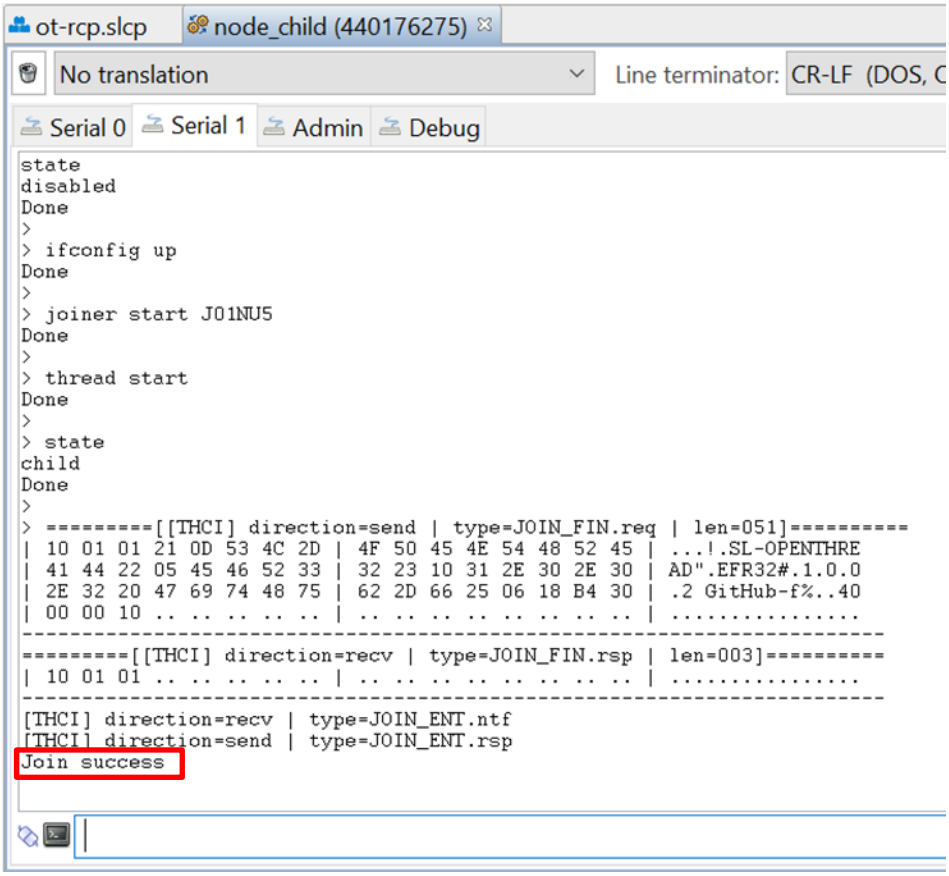
1. `commissioner-cli /usr/local/etc/commissioner/non-ccm-config.json`
2. `start :: 49191`
3. `active`
4. `joiner enableall meshcop J01NU5`

Leader console from Raspberry Pi (sudo ot-ctl)		
Command line	Response	Description
`commissioner-cli /usr/local/etc/commissioner/non-ccm-config.json`	`A commissioner shell opens`	Start the OT Commissioner CLI with the Non-CCM configuration:
`> start :: 49191`	`[done]`	Connect to OTBR
`> active`	`true` `[done]`	Verify that the Commissioner is active
`> joiner enableall meshcop J01NU5`	`> Done`	In OT Commissioner, enable Thread 1.1 MeshCoP joiner for all Joiners with a password of `J01NU5`:

7.1.1 Join the network

From the joiner (node device), enter the following commands lines (image below), it should take 2 minutes:

As we can see, at the beginning the device is not in the network.



We do not need to enter channel, masterkey, panid and so on, all steps were automatized in bash script.

In case you have a fail message, repeat `joiner start J01NU5` command line.

8 Communicate with the external world

Previously we added a prefix to communicate locally with IPV6 addresses:

OTBR pings child (ping local address)		
Command line	Response	Description

`> ping`	`sudo ot-ctl`
	`> ping fe80:0:0:0:584b:fe7e:55d3:61fd`
	`Done`
	`> 16 bytes from fe80:0:0:0:584b:fe7e:55d3:61fd: icmp_seq=1 hlim=64 time=37ms`
		Send an ICMPv6 Echo Reques

child pings OTBR (ping local address)		
Command line	Response	Description

`> ping`	`> ping fe80::e01c:c24:102:dc9`
	`Done`
	`>`
	`> 16 bytes from fe80:0:0:0:e01c:c24:102:dc9: icmp_seq=4 hlim=64 time=32ms`
		Send an ICMPv6 Echo Reques

In both directions, the local address works, since there is only one hop.

If I want to communicate with my IPV6 *eth0* interface, it will not work, the mesh prefix was not created for it. The interface *eth0* uses ipv4 address, to communicate between IPV6 and IPV4, we need a translater: NAT64.

```
pi@raspberrypi:~/ot-commissioner $ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.50.128  netmask 255.255.255.0  broadcast 192.168.50.255
        inet6 fe80::b6e7:5f9b:5648:ed55  prefixlen 64  scopeid 0x20<link>
        ether b8:27:eb:25:d5:05  txqueuelen 1000  (Ethernet)
        RX packets 1075  bytes 129605 (126.5 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1190  bytes 248899 (243.0 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Contrary to *eth0*, *wpan0* interface has the mesh prefix

```
pi@raspberrypi:~/ot-commissioner $ ifconfig wpan0
wpan0: flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST>  mtu 1280
        inet6 fdde:ad00:beef::ff:fe00:4c00  prefixlen 64  scopeid 0x0<global>
        inet6 fdde:ad00:beef:0:5fc9:3d24:caa8:f611  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::e01c:c24:102:dc9  prefixlen 64  scopeid 0x20<link>
        inet6 fdde:ad00:beef::ff:fe00:fc00  prefixlen 64  scopeid 0x0<global>
        inet6 2001:dce:1:ffff:60de:a6a9:1019:7812  prefixlen 64  scopeid 0x0<global>
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen 500  (UNSPEC)
        RX packets 8  bytes 448 (448.0 B)
        RX errors 0  dropped 1  overruns 0  frame 0
        TX packets 21  bytes 2936 (2.8 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

In `/etc/tayga.conf`, we have the prefix to communicate between IPV4 and IPV6,.

The prefix used in NAT64 is 2001:db8:1:ffff::/96, you can retrieve it with the following command line:

```
cat /etc/tayga.conf | grep -i prefix.*96$
```

Do not mix NAT64 prefix with the mesh prefix, the mesh prefix 2001:dce:1:ffff is used to communicate in mesh network and the prefix NAT64 2001:db8:1:ffff::/96 is used to communicate between IPV4 and IPV6 addresses.

```
192.168.50.128 - pi@raspberrypi: ~/ot-commissioner VT
File  Edit  Setup  Control  Window  Help
pi@raspberrypi:~/ot-commissioner $ cat /etc/tayga.conf | grep -i prefix.*96$
prefix 2001:db8:1:ffff::/96
# prefix 64:ff9b::/96
pi@raspberrypi:~/ot-commissioner $
```

To ping an IPV4, just use the NAT64 prefix and add the IPV4 address in hex format, here we have `192.168.50.128`

Decimal	Hexadecimal	Prefix::IPV4	Description
192.168.50.128	C0a8:3280	2001:db8:1:ffff::C0a8:3280	Communicate with eth0 interface

Child pings (eth0 interface)		
command	Response	Description
ping 2001:db8:1:ffff::C0a8:3280	> 16 bytes from 2001:db8:1:ffff:0:0:c0a8:3280: icmp_seq=8 hlim=62 time=42ms Done	ping eth0 address
Child pings google		
ping 2001:db8:1:ffff::808:808 Done	> 16 bytes from 2001:db8:1:ffff:0:0:808:808: icmp_seq=9 hlim=116 time=50ms	ping google

Here is the result of the external communication of the child and the external world, you can get IPV4 of Google from this link https://fr.wikipedia.org/wiki/Google_Public_DNS

Google Public DNS

Google Public DNS is a service from Google that provides recursive DNS servers to Internet users. It was announced onDecember 9, 2009[1] .

The anycast IP addresses of the servers are as follows:

- IPv4: 8.8.8.8 and 8.8.4.4
- IPv6: 2001: 4860: 4860 :: 8888 and 2001: 4860: 4860 :: 8844 [2]