vEPC 2.0 User Manual

NOTE: The instructions given in this manual work only for linux-used machines, and might not work as expected on other OSes such as Mac and Windows. Please use online references in such cases. For details on understanding the code and various procedures involved in vEPC, please look at the **`developer_manual.pdf`** under **`doc`** folder in vEPC 1.0. For details on understanding the distributed architecture, load balancing strategies and data synchronization, please look at the **developer_manual_2.0.pdf** under **doc** folder in **vEPC 2.0**.

Installation

1. Download vEPC 1.1 repository, which includes **`doc`** , **`scripts`** and **`src`** folders.
2. Navigate to **`scripts`** folder and run **`install.sh`** file.

   ```
   $ bash install.sh
   ```

   This will install all the software modules/tools required for proper compilation and working of vEPC.
3. Navigate to **`src`** folder and run **`makefile`** to obtain binary executables for the individual software modules: MME, HSS, SGW, PGW, RAN, SINK.

   ```
   $ make
   ```

4. The following binary executable files would have been generated: **`mme.out`** , **`hss.out`** , **`sgw.out`** , **`pgw.out`** , **`ransim.out`** , **`sink.out`** . With these executables, you can now proceed to setup our vEPC, where individul modules will be hosted on separate Virtual Machines (VMs).

Note: If there were any installation errors thrown by the compiler while running makefile, please install the corresponding dependencies. There should not be any other errors in this set of steps. Please note that the above procedure is provided only to understand the installation instructions, and it can be skipped to move directly to the Setup section.

Setup LTE components

1. vEPC 2.0 is a distributed design. It can contain many parallel replicas of LTE components. We describe here a three replica system. More replicas can be setup using the same procedure. Figure 1 shows virtual machine placement and interconnection of LTE components in the distributed setup. The system as per the Figure 1 contains two replicas for MME, SGW and PGW one HSS. Each of the MME replicas is identical to each other and are placed behind a load balancer. SGW and PGW also follow similar setup. A shared data store is also used for state sharing among the replicas. The setup also contains a ran-simulator which generates load for the

vEPC and a SINK module to receive the load. Therefore, to setup a two replica system you will be requiring 15 virtual machines.

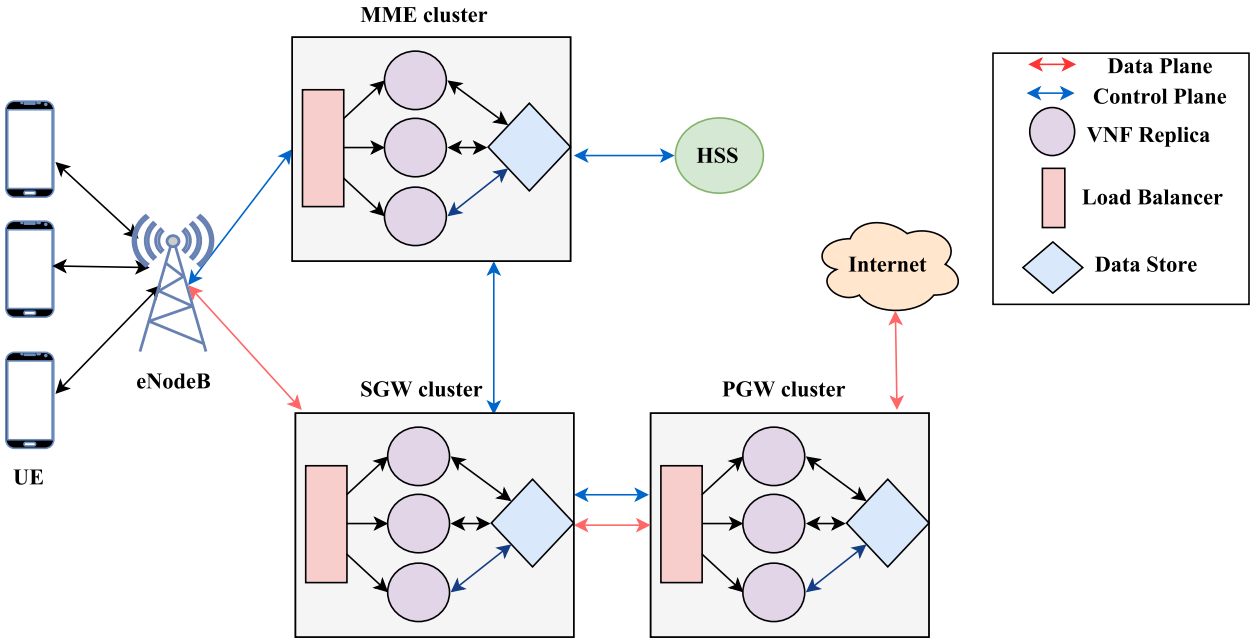

Figure 1: vEPC Setup.

2. The VM details of a standard experiment setup of our vEPC is listed in Table 1. Specification for MME, SGW and PGW are for a single instance. A two replica system having two MMEs, two SGWs and two PGWs will require each of the instances to have the mentioned specification.

Table 1: Individual component specification

COMPONENT	CPU CORES	RAM	DISK	OS
RAN, SINK	8	8 GB	10 GB	Ubuntu 14.04
MME, SGW, PGW, HSS (per instance)	1	2 GB	10 GB	Ubuntu 14.04
MME LOAD BALANCERS SGW LOAD BALANCERS PGW LOAD BALANCERS	2	4 GB	10 GB	Ubuntu 14.04
LEVELDB CLUSTER	8	8 GB	20 GB	Ubuntu14.04

Before proceeding, ensure proper communication among all VMs through the use of **`ping`** command. Also, note down the **`eth0`** IP addresses of VMs for future references.

3. Distribute the source code files into six sets according to the table 2 below. Eact set of **`.cpp/.h`** files correspond to one of the EPC modules. Note that in table 2, each file name (e.g., **`diameter`**) corresponds to

both **`.cpp`** and **`.h`** files (**`diamter.cpp`** and **`diameter.h`**).

Table 2: Setup: Source code distribution.

MODULE	MME	HSS	SGW	PGW	RAN	SINK
diameter	✓	✓	✓	✓	✓	✓
gtp	✓	✓	✓	✓	✓	✓
hss		✓				
hss_server		✓				
mme	✓					
mme_server	✓					
mysql		✓				
network	✓	✓	✓	✓	✓	✓
packet	✓	✓	✓	✓	✓	✓
pgw				✓		
pgw_server				✓		
ran					✓	
ran_simulator					✓	
s1ap	✓	✓	✓	✓	✓	✓
sctp_client	✓				✓	
sctp_server	✓	✓				
security	✓				✓	
sgw			✓			
sgw_server			✓			
sink						✓
sink_server						✓
sync	✓	✓	✓	✓	✓	
telecom	✓				✓	
tun					✓	✓
udp_client	✓		✓	✓	✓	✓
udp_server			✓	✓	✓	✓
utils	✓	✓	✓	✓	✓	✓

4. Once source code files are segregated for each module (MME, HSS, SGW, PGW, RAN, SINK), place each set of files in its corresponding VM. Perform the following steps for each VM.

5. Copy the **install.sh** file from **scripts** folder and run it to have all the required packages/tools in all the VMs that will be used for EPC operations. Install any additional tools that might be required (e.g., **gedit, emacs**)

```
$ bash install.sh
```

6. Open the utilsh file for the module.
7. Set DEBUG flag to 0/1 according to your need of debug outputs.
8. Set UE_BINDING flag to 1/2/3 for Always sync/ Session sync/ No sync for the requirement mode of operation in control plane.
9. Set NOCACHE to 1 to enable Always sync mode in data plane and 0 for Session synch mode in data plane.
10. Set RAN as the current IP of the ran simulator system. In case multiple RAN are used set the RAN2 and RAN3 to the IP address of the other RAN VMs (assuming there are 3 parallel RANs).
11. Set MMELB to the ip address of the MME load balancer.
12. Set SGWLB to the ip address of the SGW load balancer.
13. Set PGWLB to the ip address of the PGW load balancer.
14. Set SINK to the ip address of the SINK system. In case multiple SINKs set SINK2 and SINK3 to the IP address of the other SINK VMs.

- **Set up RAN Simulator**

  Note: Perform the following steps with the ran simulator code module only

  (a) Set DATA_TRANSFER flag to 0/1 for choice between control plane / data plane operations.

  (b) Set DSR = 99 for ran simulator as it does not need a data store integration. Refer to in code documentation in utils.h for more information.

  (c) compile using the command

  ```
  $ make ransim.out
  ```

  (d) run using the command for control plane operation:

  ```
  ./mme.out <num of parallel UEs> <Duration of experiments>
  ```

  (e) run using the command for data plane operation:

  ```
  ./ransim.out <num of parallel UEs> <Inpul load generated from each UE>
  ```

- **Set up MME**

  Note: Perform the following steps with the MME code module only and for each of the parallel MME replicas.

  (a) Open the utilsḣ file for the module.

  (b) set MME1 to the ip address of the current MME system. In case of multiple replicas for the MME cluster, please specify the next replicas IP address (assuming 2 more parallel replicas) as MME2 and MME3.

  (c) Set INIT_VAL TO 1000000 / 4000000 / 7000000 for different replicas to maintain a disjoint range for UE tunnel id assignment.

  (d) Set REP to 1 to enable synchronization with sibling replicas.

  (e) Set DSR to 0/1/2/3 for integration services with different data stores. Refer to in code documentation for more details.

  (f) compile using the command

  ```
  $ make mme.out
  ```

  (g) run using the command for control plane operation:

  ```
  ./mme.out <num of worker threads for control plane>
  ```

- **Set up SGW**

  Note: Perform the following steps with the SGW code module only and for each of the parallel SGW replicas.

  (a) Open the utilsḣ file for the module.

  (b) set SGW to the ip address of the current MME system. In case of multiple replicas for the SGW cluster, please specify the next replicas IP address (assuming 2 more parallel replicas) as SGW2 and SGW3.

  (c) Set REP to 1 to enable synchronization with sibling replicas.

  (d) Set DSR to 0/1/2/3 for integration services with different data stores. Refer to in code documentation for more details.

  (e) compile using the command

  ```
  $ make sgw.out
  ```

(f) run using the command for control plane operation:

```
./sgw.out <threadCount1> <threadCount2> <threadCount3>
```

threadCount# is worker count for control plane, uplink and downlink data plane

- **Set up PGW**

Note: Perform the following steps with the PGW code module only and for each of the parallel PGW replicas.

(a) Open the utilsḣ file for the module.

(b) set PGW1 to the ip address of the current PGW system. In case of multiple replicas for the PGW cluster, please specify the next replicas IP address (assuming 2 more parallel replicas) as PGW2 and PGW3.

(c) Set REP to 1 to enable synchronization with sibling replicas.

(d) Set DSR to 0/1/2/3 for integration services with different data stores. Refer to in code documentation for more details.

(e) compile using the command

```
$ make pgw.out
```

(f) run using the command for control plane operation:

```
./pgw.out <threadCount1> <threadCount2>
```

threadCount# is worker count for control plane/uplink and down link data plane

- **Set up HSS**

Note: Perform the following steps with the HSS code module only.

(a) Open the utilsḣ file for the module.

(b) set HSS to the ip address of the current HSS system.

(c) Set DSR to 0 for integration services with leveldb data store which serves as the hss internal state store for storing user authentication information. Refer to in code documentation for more details.

(d) compile using the command

```
$ make hss.out
```

(e) run using the command for control plane operation:

```
./hss.out <servingthreadCount>
```

(f) Setting up HSS state store

i. Set up a levelDB based key value store **`hss`** in the VM assigned for HSS. A levelDB server client implementation is included as part of the release in directory datastore. Any other datastore also can be used as long as it follows identical setup. We describe below the procedure to setup a levelDB datastore.

ii. Copy the setup_ds directory from the scripts directory.

iii. Place the directory in the virtual machine for HSS.

iv. cd to setup_ds and execute

```
$ bash install_server.sh
$ bash run_server.sh <ip of the vm>:8090
```

v. HSS has a details of authenticated users who can perform attach. To configure data, copy the **setup_hss** directory from **scripts** folder replace the target ip address of levelDB in **fill_hss.cpp** and run the following steps to load data.

```
$ run load_data.sh
```

The command should end with a success message if everything worked correctly.

- **Set up SINK**

Note: Perform the following steps with the SINK code module only.

(a) Open the utilsh file for the module.

(b) set SINK to the ip address of the current HSS system.

(c) Set DSR to 99 as SINK does not require any data store integration. Refer to in code documentation for more details.

(d) compile using the command

```
$ make sink.out
```

(e) run using the command for control plane operation:

```
./sink.out <servingthreadCount>
```

Setup Datastore for MME/ SGW/ PGW

Refer to instruction provided at LINK.[give link to jash's documentation]

Setup load-balancers

The system has been well tested with LVS-DR based load balancer. However, you are free to choose any other layer-4 load-balancer as long as you follow similar configuration. The following section contains configuration of an LVS-DR based load balancer.

Setup three virtual machines for MME/SGW/PGW load balancers as per the specification provided in Table 1.

1. Follow the following steps to setup each of the load-balancers:

2. Edit the ipvsadm configuration as shown below

```
$ vi /etc/default/ipvsadm
# change
AUTO="true"
# change
DAEMON="master"
# change to the interface to use
IFACE="eth0"
```

3. Create a new interface with a virtual ip that is different from the ip address of the load balancers. A different IP can serve the purpose of a virtual director to handle a master slave load balancer configuration in case of failure of the load balancer.

```
vi /etc/network/interfaces
# add to the end
auto eth0:0
iface eth0:0 inet static
address <VIP> (VIP that is exposed to clients)
network 10.0.0.0  (your network address)
netmask <subnetmask> (your subnet mask)
root@user:~# ifup eth0:0
```

4. With this procedure the laod balancer network setup is complete. LVS load balancing can be referred from ipvsadm man pages. Load balancing configuration for a certain setup with three replicas is given in the script folder. Next section shows one of the LVS load balancing configuration as an illustration. [LVS configuration of SGW]

```
ipvsadm -A -u <SGW_VIP>:7000 -s rr
ipvsadm -a -u <SGW_VIP>:7000 -r <Replica 1 IP>:7000 -g -w 1
```

```
ipvsadm -a -u <SGW_VIP>:7000 -r <Replica 2 IP>:7000 -g -w 1
ipvsadm -a -u <SGW_VIP>:7000 -r <Replica 3 IP>:7000 -g -w 1

ipvsadm -A -u <SGW_VIP>:7100 -s rr
ipvsadm -a -u <SGW_VIP>:7100 -r <Replica 1 IP>:7100 -g -w 1
ipvsadm -a -u <SGW_VIP>:7100 -r <Replica 2 IP>:7100 -g -w 1
ipvsadm -a -u <SGW_VIP>:7100 -r <Replica 3 IP>:7100 -g -w 1

ipvsadm -A -u <SGW_VIP>:7200 -s rr
ipvsadm -a -u <SGW_VIP>:7200 -r <Replica 1 IP>:7200 -g -w 1
ipvsadm -a -u <SGW_VIP>:7200 -r <Replica 2 IP>:7200 -g -w 1
ipvsadm -a -u <SGW_VIP>:7200 -r <Replica 3 IP>:7200 -g -w 1
```

5. The load balancer configuration also requires an iptables entry at the MME/SGW/PGW replicas.

   ```
   iptables -t nat -A PREROUTING -d <SGW_VIP> -j REDIRECT
   ```

   More information on LVS setup can be found here.

[NOTE: The complete script for all load balancing rules for MME, SGW, PGW and iptables rules scrripts are provided in the script directory for reference.]

The vEPC setup would be now complete and you can proceed to experimenting with our vEPC using different types of traffic. A sketch of the implemented setup is given in Figure 1. Communication among modules will take place according to the blue/red lines given in Figure 1.

Generating Control traffic

For experiments with control traffic, we simulate a number of concurrent UEs in the RAN simulator and make the UEs continuously perform attach and detach procedures with the EPC, to create a continuous stream of control traffic. We increase or decrease load on the EPC by varying the number of concurrent UE threads loading the EPC. A detailed description of the procedure is given below.

1. Once the setup is ready, run each vEPC binary executable in its own VM. Usage format of each executable is given in table 3 below. Please look at the **`developer_manual.pdf`** of vEPC 1.0 under **`doc`** folder to understand more about the command line parameters used with each executable.

Table 3: Usage format of binary executables.

MODULE	USAGE
MME	**./mme.out <#S1-MME threads>**
HSS	**./hss.out <#S6a threads>**
SGW	**./sgw.out <#S11 threads> <#S1 threads> <#S5 threads>**
PGW	**./pgw.out <#S5 threads> <#SGi threads>**

A sample run of vEPC modules is given below.

MME:

```
$ ./mme.out 50 (to be done at each mme instance)\\
```

HSS:

```
$ ./hss.out 50
```

SGW:

```
$ ./sgw.out 50 50 50 (to be done at each instance)
```

(to be done at each sgw instance)

PGW:

```
$ ./pgw.out 50 50 (to be done at each instance)
```

2. **RAN:** Open **ran_simulator.cpp** and comment out the following code sections.
   - **data_transfer** section under the **simulate** function.
   - **tun** and **traffic_monitor** sections under the **run** function.

   By doing this, you are simulating **RAN** objects that generate only control traffic.

3. **RAN:** Rerun the **makefile** for the RAN module to make the binary executable for generating only control traffic.

   ```
   $ make ransim.out
   ```

4. **RAN:** Start RAN simulator with appropriate parameters as given below. This will generate the required amount of control traffic for the given time duration.

   ./ransim.out <#RAN threads> <Time duration>

A sample run is as follows.

```
$ ./ransim 10 100
```

Fault tolerance experiments:

For fault tolerant experiments the lvs load balancer needs to be configured with the keepalived extension. A sample working script for the MME load balancer is available at the script/LBrules directory.

Enabling the keepalived extension:

1. Install keepalived by sudo apt-get install keepalived
2. Copy the keepalived script provided at the above mentioned script location.
3. Start keepalived in root with command: service keepalived start.
4. [optional] For stopping keepalived mode we can use service keepalived stop

Use the ran simulator code provided in the misc directory in the folder structure for fault tolerance operations. This RAN version has all procedure retry codes required. Start EPC in a 3 replica mode with around 30 concurrent UEs. Stop one of the replica by killing its server. The ran will list out number of registration attempts and number of successful attempts. It also informs about number of extra retry count that was required for successful registration compared to a normal case.

Generating Data traffic

For experiments with data traffic, we attach a specified number of UEs to the EPC from the RAN simulator, and pump traffic from the RAN to the sink. A traffic generating tool called **`iperf3`** is employed, using which TCP data can be sent with a given bandwidth and time duration. To generate traffic, **`iperf3`** process is started in Server mode at the Sink module and the corresponding **`iperf3`** Client process is started at the RAN simulator with the required input data rate and time duration parameters. A detailed description of the procedure is given below.

1. Begin vEPC modules (MME, HSS, SGW, PGW) as explained before.
2. **SINK:** Start the Sink module with the required number of **`iperf3`** servers.

   `./sink.out <#iperf3 servers>`

   A sample run is as follows.

   ```
   $ ./sink 10
   ```

3. **RAN:** Open **ran_simulator_data.cpp** and rename it to **ran_simulator.cpp**.

4. Open utils.h and set DATA_TRANSFER to 1 to change ran simulator mode.

5. **RAN:** Open **ran.cpp** and set the duration of **iperf3** data transfer - **dur** under the **transfer_data** function. Note that this duration has to be smaller than the overall duration given as command line parameter while running RAN simulator. Input data rate is currently fixed at 1 Mbps. It can be modified as per requirements.

6. **RAN:** Rerun the **makefile** for the RAN module to reflect the updates in the binary executable.

   ```
   $ make ransim.out
   ```

7. **RAN:** Now begin the RAN simulator as explained before. This will simulate the required number of **RAN** objects, generating both control traffic and data traffic in sequence, for the given time duration. Note that the number of **RAN** objects created has to be equal or greater than the number of **iperf3** servers started at Sink module.

Performance results

Control traffic

Two performance metrics would be reported at the end of control traffic experimentation. These parameters are given below. No additional scripts/coding would be required to produce these parameters as they are computed along with the generation of control traffic.

1. **Throughput**, the number of registration requests successfully completed by the EPC per second

2. **Latency**, the time taken by a registration to complete

Data traffic

While experimenting with data traffic, the following steps need to be followed to obtain performance results.

1. **SINK:** Copy the **find_bw.sh** file from **scripts** folder. Just before starting the RAN simulator, run this script at the Sink module. This will measure the required throughput value. A file named **bw.txt** will be created, which contains the various throughput values (in bytes/second): Uplink, Downlink, Total (Uplink + Downlink).

   Format:

   bash find_bw.sh -i <interface> -s <duration> -c <count> -n <#UEs>

   Sample run:

```
$ bash find_bw.sh -i eth0 -s 100 -c 1 -n 1
```

[Note: find_bw.sh can be found in the scripts folder]

2. **RAN:** Before starting the RAN simulator, open **`ran_simulator.cpp`** and uncomment the **`RTT`** code section under the **`simulate`** function. This will create a **`ping.txt`** file, which contains the Round Trip Time (RTT) latency values for the data transfer in milliseconds. Note that the time duration under **`ping`** function need to be set according to the time duration of the user data transfer.

With the procedure explained above, two performance metrics would be obtained at the end of data traffic experimentation. These parameters are given below.

1. **Throughput**, the amount of data traffic successfully forwarded by the EPC gateways to the Sink per second
2. **RTT Latency**, the amount of processing overhead added by the EPC gateways in the data plane, as measured by a ping command during load test

Our results

On a succesful setup of a 3 replica distributed vEPC 2.0 the following throughput results may be observed.

Control plane throughput results:

No sync mode evaluated with 1, 2 and 3 replicas configuration of MME / SGW / PGW. The graph captures number of registration/second vs number of concurrent UEs. The saturation throughput for 1, 2, 3 replicas respectively are 4283, 6917, 10215.6 registrations/second.

Session-sync mode evaluated with 1, 2 and 3 replicas configuration of MME / SGW / PGW. The saturation throughput for 1, 2, 3 replicas respectively are 2016, 3752, 5004 registrations/second.

Always-sync mode evaluated with 1, 2 and 3 replicas configuration of MME / SGW / PGW. The saturation throughput for 1, 2, 3 replicas respectively are 1249, 2195, 2885 registrations/second.

Data plane throughput results:

Session-sync mode shows EPC data plane scaling in Session sync mode evaluated with 1, 2 and 3 replicas configuration of MME / SGW / PGW. The saturation data plane throughput for 1,2,3 replicas respectively are 233, 405, 504 Mbps.

Always-sync mode shows EPC data plane scaling in Always sync mode evaluated with 1, 2 and 3 replicas configuration of MME / SGW / PGW. The saturation data plane throughput for 1,2,3 replicas respectively are 43, 83, 121

Mbps.

Tips to avoid common mistakes in setup:

- Check enabling of virtual interfaces at all load balancers while setting up VIP.
- Check consistency between ip address mentioned in utils.h for all modules and their real ip addresses
- Check iptables rules set installation in all worker replicas of MME, SGW and PGW.
- Check correctness the UE INIT_VALUE range for all MME replicas.
- Enable ip fowrading in all Load balancers.
- Check that the load balancers has been set with source_hash_ipvs3.conf for No sync mode.
- Check use of proper DSR setting.