**UML Switch con interfaces TAP Remotas en un DVS**

El User Mode Linux (UML) es un Linux que ejecuta en modo Usuario y que utiliza técnicas y recursos de la virtualización para ejecutar. Una de esos recursos son las interfaces de red virtuales (vNIC) que se asocian a una interface TAP del Host donde reside el UML.

Otra forma de conectar el UML a una red en la que reside el Host es a través de un Switch virtual (uml\_switch) que es un proceso demonio que, entre sus interfaces cuenta con una interface TAP del Host. El switch virtual dispone de varios puertos virtuales que son simplemente conexiones entre procesos a través de Unix Sockets. Cada puerto tiene un socket de control (por donde intercambia requerimientos con el Switch) y un puerto de datos por donde se transfieren paquetes (concretamente frames) entre el Switch y el proceso daemon en cuestión.

Un DVS es un Sistema de Virtualización Distribuido que permite gestionar múltiples Contenedores Distribuidos (DC) los cuales pueden abarcar más de un nodo del cluster de virtualización y los procesos que se ejecutan dentro del DC se pueden comunicar entre sí sin considerar el nodo en el que se encuentran ejecutando ni direcciones IP que lo relacionan con una localización o interface determinada. Además, las comunicaciones en un DVS soportan la replicación y migración de servicios los cuales proveen una solución para aquellos sistemas que requieren alta disponibilidad.

El DVS UML SWITCH es una modificación del UML Switch al cual se le han incorporado interfaces TAP localizadas en nodos diferentes al Host donde reside el UML. Si bien esta extensión en su funcionamiento puede realizarse utilizando protocolos más estándares tales como TCP o UDP o TIPC, la utilización de M3IPC (mecanismo de IPC del DVS) tiene el valor agregado de soportar facilidades de replicación y migración que no están disponibles en los otros protocolos, o eventualmente, su uso hace más compleja la solución.

En la Fig. 1 se muestra cómo funciona un UML Switch original.



1. UML Switch original.

Los UMLs se unen al switch habilitando la interface virtual denominada “*daemon*” que es un cliente del server que ejecuta en el Switch. Como se mencionó anteriormente, la comunicación se realiza utilizando Unix sockets. Los puertos virtuales adicionales al Switch se crean en forma automática al conectarse nuevos UMLs al Switch.

El objetivo del DVS UML Switch es incorporarle la posibilidad de utilizar interfaces TAPs pero de Hosts remotos, tal como se muestra en la Fig. 2.



1. Esquema lógico del DVS UML Switch

La implementación del DVS UML Switch se realiza utilizando dos hilos (threads) que son los que se comunican con los nodos remotos y se usa un par de Unix sockets (Control y Datos) por cada interface TAP Remota (***rmttap***) (Fig. 3).



1. Esquema operacional del DVS UML Switch

En este esquema de ejemplo, se puede implementar la siguiente configuración:

* Las interfaces tap0 de cada nodo se dedican a conectarse al Switch del Host Local.
* Las interfaces tap1 se dedican a conectarse (a través del DVS) a un Switch Remoto.

El thread Sender se encarga de tomar frames de los Unix Sockets de cada uno de los ports virtuales del switch enviarlos al thread Receiver del switch del nodo remoto correspondiente De la misma forma lo hace con las interfaces TAP locales asociadas a un Switch remoto.

El thread Receiver se encarga de recibir las peticiones y datos procedentes de los thread Senders de los nodos remotos e insertarlos en los Unix Sockets del switch local o en la interface TAP del Host local asociada a un Switch remoto.

Cuando se arranca el DVS UML Switch se requiere indicarle un archivo de configuración, como por ejemplo:

# this is a comment

switch SW\_NAME {

path "/tmp/dvs\_uml.ctl";

hub NO;

daemon NO;

tap "tap0";

dcid 0;

};

rmttap RTAP1 {

tap "tap1";

nodeid 1;

};

rmttap LTAP1 {

tap "tap1";

nodeid 0;

};

En este archivo de ejemplo se establece la configuración correspondiente al Switch del NODE0 del esquema de la Fig. 3.

* Se especifica el path del unix socket de control del switch.
* Se indica si se va a comportar como hub (Broadcast de todos los frames por todos los puertos) o como switch.
* Se indica si se va a ejecutar en modo interactivo (generalmente cuando se realiza depuración) o como demonio.
* Se indica cual es el nombre de la interface TAP local asociada al switch Local.
* Se indica el identificador del DC (DCID) en donde se ejecutará el switch y todos sus componentes.
* Luego, se especifica una interface TAP remota (RTAP1) que se asociará a la interface tap1 del NODE1.
* Finalmente se especifica la interface local LTAP1 asociada a la interface tap1 del nodo Local. Esta interface no está conectada al switch Local sino que está asociada a un switch remoto. Esto se distingue porque el parámetro *nodeid* se corresponde con el identificador del nodo local.

# COMO FUNCIONA UML SWITCH

El *uml\_switch* es un demonio (en general) que inicializa las ports virtuales consistentes en unix sockets, y la interface TAP del Host.

En el loop infinito utiliza la system call *poll()* que monitorea varios file descriptors (fds[nfds]) al mismo tiempo. Puede recibir peticiones desde la STDIN (fd=0) o en su *connect\_fd* o en su *data\_fd*.

* Cuando el *connect\_fd* recibe una conexión, crea un nuevo *fd* de control que será monitoreado por el *poll()*. Luego, el *poll()* informa de un evento en ese *fd*, se hace una búsqueda entre todos los ports de control (*handle\_port()*) para ver si alguno coincide con ese *fd*, si no coincide lo devuelve como *new*. Lo toma *new\_port()*, que recibe el requerimiento de conexión (REQ\_NEW\_CONTROL) desde el UML junto con el socket de datos que va a utilizar y le retorna el *data* file descriptor del socket para ese *port*.
* Cuando un evento en el *data\_fd* es el que disparó el *poll()*, *handle\_sock\_data()* lee los datos que viene en el socket y le da tratamiento con *handle\_data()*.

Cuando uno de los UMLs envía un frame (denominado erróneamente packet) al switch, este verifica (*match\_sock()*) que el *fd* de datos del que proviene está en la lista de puertos habilitados. Si está todo bien, busca en una lista HASH de MAC addresses (*update\_src()*) para saber cuál puerto virtual debe utilizar para enviar el frame. Si no lo encuentra, deberá realizar un Broadcast a todos los puertos y cuando el destinatario responda con algún frame, entonces la MAC address hasta ahora desconocida será incorporada a la lista HASH para próximos frames. Si en la configuración se indicó un comportamiento de tipo HUB entonces también todos los frames son enviados a todos los ports virtuales excepto el del emisor. La lista de HASH puede ir actualizándose si es que no coincide con los datos contenidos en la misma.

Finalizado este procesamiento, se utiliza *send\_dst()* para enviar a cada destino el frame recibido mediante *send\_sock()*.

Un tratamiento similar se da a los frames recibidos desde el dispositivo TAP local, pero dispone de su propio mecanismo de envío de frames hacia la TAP local. En este caso, *send\_dst()* utiliza *send\_tap()* para enviar a la interface TAP.

# COMO FUNCIONAN *DVS\_UML\_SWITCH*

A los efectos de mantener lo más parecido el funcionamiento al *uml\_switch* original, la comunicación del Switch con los RTAP se hace utilizando Unix sockets, de tal forma que no hace ningún tratamiento diferente entre lo que sería un UML en el HOST que un RTAP.

Se inicializan las estructuras de datos de los Remote TAPs (*init\_rt\_sockets()*) y se conecta cada uno de ellos al switch (*connect\_to\_switch()*), y se dan de alta en el array de *fds* que será monitoreado por el *poll()* que utilizará el hilo Sender.

Cada uno de los nodos remotos se registra en el DVK para poder utilizar M3IPC (*dvk\_rmtbind()*) entre el nodo local y el remoto. Esto se hace buscando en el campo *dc\_nodes* de la estructura *dc\_usr\_t*. Cada bit de ese bitmap representa un nodo asociado al DC. La posición del bit determina el identificador del nodo (nodeid=0 => bit 0).

Esto se puede ver cuando se lista información del DC. Los nodos marcados con “X” en el listado, son los nodos en los que se puede expandir el DC0.

# cat /proc/dvs/DC0/info

dcid=0

flags=0

nr\_procs=221

nr\_tasks=34

nr\_sysprocs=64

nr\_nodes=32

dc\_nodes=7

dc\_pid=729

warn2proc=27342

warnmsg=-1

**dc\_name=DC0**

**nodes 33222222222211111111110000000000**

**10987654321098765432109876543210**

**-----------------------------XXX**

Finalmente, se crean los hilos SENDER y RECEIVER que operan de la siguiente forma.

* **SENDER**: Monitorea con *poll()* todos los *fds* de los Unix sockets que lo unen al switch para ver si recibe datos de ellos. Cuando por uno de los *fds* recibe un frame, ese *fd* está asociado a un nodo remoto (*rt\_nodeid*) y un TAP remoto (*rt\_rmttap\_fd*). El SENDER realiza un *dvk\_sendrec()* para enviar la petición de WRITE de frame hacia el TAP remoto (REQ\_RT\_WRITE) indicando la dirección en memoria del frame (*packet*) y el *fd* en el nodo remoto de la interface TAP (*rt\_rmttap\_fd*) exportada de ese nodo remoto. El server TAP remoto, realiza la transferencia del frame utilizando *dvk\_vcopy()* desde el nodo del switch hacia un buffer propio. Luego hace el *write()* a su interface local TAP exportada y en función del resultado, le retorna la respuesta al SENDER, el cual luego sigue con su loop de *poll()*.
* **RECEIVER:** El hilo RECEIVER tiene 2 funciones. 1) recibir paquetes desde las interfaces TAP remotas e insertarlas en los sockets de los puertos virtuales correspondientes del switch 2) recibir peticiones de OPEN, WRITE y CLOSE para realizar sobre las interfaces TAP locales exportadas a switches remotos.

Si durante el inicio del *dvs\_uml\_switch* algunos de los nodos remotos que exportan interfaces TAP al switch no hubiesen estado conectados, el SENDER, cuando está ocioso, reintenta periódicamente el OPEN de la TAP remota. También hace el reintento de OPEN cuando hubo un fallo de envío de paquete a esa TAP remota (posiblemente por fallo del nodo remoto, del proceso remoto o por un rearranque). En caso de fallo, se setea como inválido el *rt\_rmttap\_idx* hasta tanto se vuelva a conectar el nodo remoto y el OPEN sea exitoso*.*

Al finalizar con CTRL-C (*SIGINT*) se ejecuta una rutina de *cleanup()* que elimina los archivos de sockets control y datos, hace el *dvk\_unbind()* de todos los Endpoints remotos, y hace *close()* de las interfaces TAP locales exportadas y abiertas.

# ESTRUCTURAS DE DATOS UTILIZADAS

Los parámetros, datos de configuración y estado del switch se almacenan en una estructura.

struct switch\_s{

int sw\_hub; // hub mode ?

int sw\_daemon; // deamon mode ?

int sw\_recv\_ep; // receiver thread endpoint (server)

int sw\_send\_ep; // sender thread endpoint (client)

int sw\_dcid; // DC ID

int sw\_tap\_fd; // tap FD

int sw\_conn\_fd; // connect FD

int sw\_data\_fd; // data FD

char \*sw\_tap; // switch TAP name for the switch

char \*sw\_ctrl\_path; // switch control socket path

char \*sw\_name; // switch name

pid\_t sw\_recv\_tid; // receiver thread TID

pid\_t sw\_send\_tid; // sender thread TID

proc\_usr\_t sw\_recv\_proc; // receiver thread process descriptor

proc\_usr\_t sw\_send\_proc; // sender thread process descriptor

pthread\_t sw\_recv\_thread; // receiver thread

pthread\_t sw\_send\_thread; // sender thread

};

typedef struct switch\_s switch\_t;

En tanto que para los dispositivos virtuales TAP remotos se usa un array de estructuras de tipo:

struct rmttap\_s{

int rt\_index; // index in the array of local structures

int rt\_nodeid; // in which node is this TAP

int rt\_ctrl\_fd; // control FD to the local switch

int rt\_data\_fd; // data FD to the local switch

int rt\_poll\_idx; // index in the fds array when polling

int rt\_rmt\_idx; // index of the rmttap in the remote node

int rt\_rmt\_ep; // remote endpoint who opens this TAP

int rt\_rmttap\_fd; // rmttap fd in the remote node

// or local fd for remote switch

char \*rt\_tap; // name of the remote TAP device

char \*rt\_name; // reference name in the local switch

// of the TAP device

struct sockaddr\_un rt\_ctrl\_sun; // control unix socket to the switch

struct sockaddr\_un rt\_data\_sun; // data unix socket to the switch

proc\_usr\_t rt\_svr\_proc; // remote switch server process descriptor

proc\_usr\_t rt\_clt\_proc; // remote switch client process descriptor

struct request\_v3 rt\_req; // request to the local switch

daemon\_data\_t rt\_dd; //

char rt\_sock\_type[MAXSOCKNAME]; // socket type "unix"

char rt\_ctrl\_path[MAXSOCKPATH]; // control path name

char rt\_data\_path[MAXSOCKPATH]; // data path name

};

typedef struct rmttap\_s rmttap\_t;

struct daemon\_data {

char \*sock\_type;

char \*ctl\_sock;

void \*ctl\_addr;

void \*data\_addr;

void \*local\_addr;

int fd;

int control;

void \*dev;

};

typedef struct daemon\_data daemon\_data\_t;

# Asignación de Endpoints

A los efectos de no adicionar nuevos parámetros a la configuración se hace una asignación fija de los Endpoints a utilizar por el DVS Switch. Esto obliga a que prácticamente se dedique un DC en forma exclusiva por cada conjunto de DVS Switches (un dvs\_uml\_switch por nodo del DC).

El Receiver Thread es de tipo Server, por lo que su Endpoint es igual al ***local\_nodeid***.-

svr\_ep = local\_nodeid;

El Server Thread es de tipo Client, y su Endpoint se calcula:

clt\_ep = (dc\_ptr->dc\_nr\_sysprocs - dc\_ptr->dc\_nr\_tasks) + local\_nodeid;

Por ejemplo, si dc\_nr\_sysprocs = 64, dc\_nr\_tasks=34 y local\_nodeid = 0, entonces

clt\_ep = 64 – 34 + 0 = 30

Es decir, los Endpoints de los Receiver Threads de los diferentes nodos serán: 0, 1, 2… y los endpoints de los Senders Threads para los diferentes nodos serán 30, 31,…

Una vez arrancado el DVS, el DC, los proxies y el dvs\_uml\_switch, en la salida por pantalla siguiente se pueden ver:

root@**node0**:~# cat /proc/dvs/DC0/procs

DC pnr **-endp** -lpid/vpid- **nd** flag misc -getf -sndt -wmig -prxy name

0 0  **0** 1058/258  **0** 8 20 31438 27342 27342 27342 dvs\_uml\_switch

0 1  **1** -1/-1  **1** 1000 1000 27342 27342 27342 27342 TAPserver01

0 2  **2** -1/-1  **2** 1000 0 27342 27342 27342 27342 TAPserver02

0 30  **30** 1057/257  **0** 8 2000 1 27342 27342 27342 dvs\_uml\_switch

0 31  **31** -1/-1  **1** 1000 0 27342 27342 27342 27342 TAPclient01

0 32  **32** -1/-1  **2** 1000 0 27342 27342 27342 27342 TAPclient02

Como el listado es en NODE0 => ***local\_nodeid=0*** su Receiver tiene Endpoint=0, y su Sender Endpoint=30. Ambos son los únicos procesos locales en el DC0, el resto son los descriptores de los procesos remotos (***flag=1000***) de NODE1 (***nd=1***) y NODE02 (***nd=2***).

# DIAGRAMA DE FLUJO DE DVS\_UML\_SWITCH



* Se inicializan variables y estructuras de datos
* Se lee la configuración desde el archivo de configuración y se completan algunos campos de las estructuras de datos.
* Se cargan en una estructura interna los parámetros del DVS.
* Se cargan en una estructura interna los parámetros del DC.
* Se crean los unix sockets de conexión y de datos del switch
* Se establece el signal handler de SIGINT para realizar una limpieza antes de terminar
* Si se tiene un dispositivo TAP asignado al switch, se realiza el OPEN.
* Se arranca un thread para inicializar todo lo referido a rmttap
* Por cada nodo perteneciente al DC se realiza el binding de los Endpoints remotos, estén estos activos o no.
* Se inicializan las estructuras de datos de las rmttaps (RTAPs y LTAPs)
* Se crean los threads Sender y Receiver



* Se inicia el ciclo de monitoreo de FDs del switch mediante poll
* Si sale por EINTR o TIMEOUT, se ignora y vuelve al poll
* Si se activó el FD connect, indica que hay alguien que se quiere conectar al switch y se acepta la conexión.
* Si se activó un FD correspondiente a un unix socket (puede ser un UML o un RTAP) se da tratamiento al paquete que llega por estos medios.
* Si se activó un FD correspondiente al TAP del switch local, se da el tratamiento al paquete que llegó por este medio.
* Si no era ninguno de estos FDs se analiza si es un port nuevo y se lo incorpora al switch, de lo contrario se cierra el FD.



* Cuando se inicia el thread emisor, se hace el dvk\_bind con Endpoint Client.
* Intenta hacer el OPEN de los RTAPs especificados
* Luego hace un ciclo de polling de todos los FDs de rmttaps (LTAPs y RTAPs)
* Se hacen ciclos anidados de los FDs activados y se los compara contra los rmttaps (LTAPs y RTAPs) para identificar a cuál de ellos corresponde.
* Si el evento fue un EINTR o TIMEOUT, reintenta hacer el open de los RTAPs especificados que aún no logró abrir (quizás porque el nodo no levantó o el dvs\_uml\_switch remoto no se arrancó o porque no hay conectividad).
* Si el FD que se activó es un unix socket de la conexión al switch, entonces es de un paquete que se necesita enviar al nodo remoto donde está el RTAP (RQ\_RT\_WRITE).
* Si el FD que corresponde al LTAP que requiere enviar un paquete al switch remoto, entonces se establece el tipo de requerimiento como (RQ\_SW\_WRITE).
* Finalmente se envía mediante el DVK el paquete al nodo remoto correspondiente.
* Vuelven los ciclos de rmtaps y FDs activados hasta terminar con todos ellos para luego volver al polling.



* Cuando se inicia el thread receptor, se hace el dvk\_bind con Endpoint Server.
* Se entra en un ciclo de dvk\_receive() que espera recibir mensajes desde otros nodos. Esto pueden ser desde RTAPs de esos nodos o pueden ser de los switches remotos que tienen interfaces en el nodo local (LTAPs)
* Si en el nodo local se exportan interfaces TAPs (LTAPs), las mismas se asignan a los nodos remotos que hicieron el OPEN de las mismas. Cada OPEN es excluyente, es decir, solo un nodo remoto puede ser el propietario de esa LTAP.
* Una vez adquirida una LTAP mediante el OPEN, entonces el thread sender del nodo remoto puede solicitar la escritura sobre la LTAP o su liberación (CLOSE).
* Si ordena la escritura de una LTAP, entonces se copia el paquete desde el nodo remoto a un buffer del nodo local mediante el DVK y se envía el contenido del buffer a la TAP local correspondiente mediante send\_tap()
* Si una interface TAP remota perteneciente al switch local (RTAP) solicita la escritura de un paquete al switch local, entonces se copia el paquete desde el nodo remoto a un buffer del nodo local mediante el DVK y se envía el contenido del buffer por el unix sockets conectado al switch local mediante send\_socket().

# ESCENARIO DE PRUEBAS

# 

ARCHIVO DE ARRANQUE DE TAPs (Embebido en tests.sh)

#!/bin/bash

if [ $# -ne 1 ]

then

echo "usage: $0 <lcl\_nodeid>"

lcl=0

else

lcl=$1

fi

let rmt=(1 - $lcl)

echo lcl=$lcl rmt=$rmt

ipbr0="192.168.1.2$lcl"

iptap0="192.168.1.20$lcl"

iptap1="192.168.1.21$lcl"

echo ipbr0=$ipbr0 iptap0=$iptap0 iptap1=$iptap1

netmask="255.255.255.0"

macbr0="02:AA:BB:CC:DD:B$lcl"

mactap0="02:AA:BB:CC:DD:$lcl$lcl"

mactap1="02:AA:BB:CC:DD:$lcl$rmt"

echo netmask=$netmask mactap0=$mactap0 mactap1=$mactap1 macbr0=$macbr0

**# enable routing between interfaces**

echo 1 > /proc/sys/net/ipv4/ip\_forward

**# Bridge configuration --------------------------------------------------------**

read -p "Configuring br0. Enter para continuar... "

brctl addbr br0

ifconfig br0 $ipbr0 netmask $netmask hw ether $macbr0

ip link set dev br0 up

**# TAP0 configuration --------------------------------------------------------**

read -p "Configuring tap0. Enter para continuar... "

mknod /dev/tap0 c 36 $[ 0 + 16 ]

chmod 666 /dev/tap0

ip tuntap add dev tap0 mode tap

if [ $lcl -eq 0 ]

then

ip link set dev tap0 address $mactap0

fi

ip link set dev tap0 up

brctl addif br0 tap0

if [ $lcl -eq 0 ]

then

ifconfig tap0 $iptap0 netmask $netmask

ifconfig tap0 | grep addr

fi

**# TAP1 configuration --------------------------------------------------------**

read -p "Configuring tap1. Enter para continuar... "

mknod /dev/tap1 c 36 $[ 1 + 16 ]

chmod 666 /dev/tap1

ip tuntap add dev tap1 mode tap

~~# ip link set dev tap1 address $mactap1~~

ip link set dev tap1 up

brctl addif br0 tap1

~~#ifconfig tap1 $iptap1 netmask $netmask~~

~~#ifconfig tap1 | grep addr~~

~~# Link ETH0 to BRIDGE~~

~~#read -p "Conecting eth0 to br0. Enter para continuar... "~~

~~#brctl addif br0 eth0~~

brctl show

brctl showmacs br0

exit

|  |  |
| --- | --- |
| **NODE0 – switch0.cfg** | **NODE1 – switch1.cfg** |
| # this is a comment  switch SW\_NAME {  path "/tmp/uml.ctl";  hub NO;  daemon NO;  tap "tap0";  dcid 0;  };  rmttap RTAP1 {  tap "tap1";  nodeid 1;  }; | # this is a comment  switch SW\_NAME {  path "/tmp/uml.ctl";  hub NO;  daemon NO;  ###### tap "tap0"; ## NOT TAP  dcid 0;  };  rmttap LTAP1 {  tap "tap1";  nodeid 1;  }; |

|  |  |
| --- | --- |
| **NODE0** | **NODE1** |
| root@node0:/usr/src/dvs/dvs-apps/dvs\_uml\_switch# ifconfig  br0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500  inet 192.168.1.20 netmask 255.255.255.0 broadcast 192.168.1.255  inet6 fe80::f035:efff:fea7:438f prefixlen 64 scopeid 0x20<link>  ether 02:aa:bb:cc:dd:b0 txqueuelen 1000 (Ethernet)  RX packets 0 bytes 0 (0.0 B)  RX errors 0 dropped 0 overruns 0 frame 0  TX packets 7 bytes 610 (610.0 B)  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0  eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  inet 192.168.0.100 netmask 255.255.255.0 broadcast 192.168.0.255  inet6 fe80::20c:29ff:fe9a:7aa4 prefixlen 64 scopeid 0x20<link>  ether 00:0c:29:9a:7a:a4 txqueuelen 1000 (Ethernet)  RX packets 374 bytes 44861 (43.8 KiB)  RX errors 0 dropped 0 overruns 0 frame 0  TX packets 410 bytes 60762 (59.3 KiB)  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0  device interrupt 19 base 0x2000  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 1 (Local Loopback)  RX packets 201 bytes 15228 (14.8 KiB)  RX errors 0 dropped 0 overruns 0 frame 0  TX packets 201 bytes 15228 (14.8 KiB)  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0  tap0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500  inet 192.168.1.200 netmask 255.255.255.0 broadcast 192.168.1.255  ether 02:aa:bb:cc:dd:00 txqueuelen 1000 (Ethernet)  RX packets 0 bytes 0 (0.0 B)  RX errors 0 dropped 0 overruns 0 frame 0  TX packets 0 bytes 0 (0.0 B)  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0  tap1: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500  ether f6:83:a8:db:f3:03 txqueuelen 1000 (Ethernet)  RX packets 0 bytes 0 (0.0 B)  RX errors 0 dropped 0 overruns 0 frame 0  TX packets 0 bytes 0 (0.0 B)  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0 | root@node1:/usr/src/dvs/dvs-apps/dvs\_uml\_switch# ifconfig  br0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500  inet 192.168.1.21 netmask 255.255.255.0 broadcast 192.168.1.255  inet6 fe80::c8a1:eeff:fe68:fc77 prefixlen 64 scopeid 0x20<link>  ether 02:aa:bb:cc:dd:b1 txqueuelen 1000 (Ethernet)  RX packets 0 bytes 0 (0.0 B)  RX errors 0 dropped 0 overruns 0 frame 0  TX packets 4 bytes 340 (340.0 B)  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0  eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500  inet 192.168.0.101 netmask 255.255.255.0 broadcast 192.168.0.255  inet6 fe80::20c:29ff:fe04:5587 prefixlen 64 scopeid 0x20<link>  ether 00:0c:29:04:55:87 txqueuelen 1000 (Ethernet)  RX packets 513 bytes 44363 (43.3 KiB)  RX errors 0 dropped 0 overruns 0 frame 0  TX packets 281 bytes 31317 (30.5 KiB)  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0  device interrupt 19 base 0x2000  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 1 (Local Loopback)  RX packets 84 bytes 6972 (6.8 KiB)  RX errors 0 dropped 0 overruns 0 frame 0  TX packets 84 bytes 6972 (6.8 KiB)  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0  tap0: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500  inet 192.168.1.201 netmask 255.255.255.0 broadcast 192.168.1.255  ether 02:aa:bb:cc:dd:11 txqueuelen 1000 (Ethernet)  RX packets 0 bytes 0 (0.0 B)  RX errors 0 dropped 0 overruns 0 frame 0  TX packets 0 bytes 0 (0.0 B)  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0  tap1: flags=4099<UP,BROADCAST,MULTICAST> mtu 1500  ether 02:aa:bb:cc:dd:10 txqueuelen 1000 (Ethernet)  RX packets 0 bytes 0 (0.0 B)  RX errors 0 dropped 0 overruns 0 frame 0  TX packets 0 bytes 0 (0.0 B)  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0 |
| root@node0:/usr/src/dvs/dvs-apps/dvs\_uml\_switch# brctl show  bridge name bridge id STP enabled interfaces  br0 8000.02aabbccddb0 no tap0  tap1  root@node0:/usr/src/dvs/dvs-apps/dvs\_uml\_switch# brctl showmacs br0  port no mac addr is local? ageing timer  1 02:aa:bb:cc:dd:00 yes 0.00  1 02:aa:bb:cc:dd:00 yes 0.00  2 f6:83:a8:db:f3:03 yes 0.00  2 f6:83:a8:db:f3:03 yes 0.00 | root@node1:/usr/src/dvs/dvs-apps/dvs\_uml\_switch# brctl show  bridge name bridge id STP enabled interfaces  br0 8000.02aabbccddb1 no tap0  tap1  root@node1:/usr/src/dvs/dvs-apps/dvs\_uml\_switch# brctl showmacs br0  port no mac addr is local? ageing timer  2 02:aa:bb:cc:dd:10 yes 0.00  2 02:aa:bb:cc:dd:10 yes 0.00  1 02:aa:bb:cc:dd:11 yes 0.00  1 02:aa:bb:cc:dd:11 yes 0.00 |

Para arrancar el switch en NODE0:

root@node0:/usr/src/dvs/dvs-apps/dvs\_uml\_switch# nsenter -p -t$DC0 ./dvs\_uml\_switch switch0.cfg > borrar0.txt

Para arrancar el switch en NODE1:

root@node1:/usr/src/dvs/dvs-apps/dvs\_uml\_switch# nsenter -p -t$DC0 ./dvs\_uml\_switch switch1.cfg > borrar1.txt

Abrir otras sesiones y ejecutar UML de la siguiente forma.

En NODE0:

root@node0:~# . /dev/shm/DC0.sh

root@node0:~# cd /usr/src/linux

root@node0:/usr/src/linux# nsenter -p -t $DC0 ./uml.linux con0=null,fd:2 con1=fd:0,fd:1 con=null ssl=null umid=node00 ubda=/dev/sdb6 rw mem=1024M dcid=0 **eth0=daemon**

Luego del arranque y loggueo hacer:

root@uml0:~# ifconfig

**eth0**: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500

**inet 192.168.1.100** netmask 255.255.255.0 broadcast 192.168.2.255

ether 92:a8:4f:60:70:e8 txqueuelen 1000 (Ethernet)

RX packets 2 bytes 112 (112.0 B)

RX errors 0 dropped 0 overruns 0 frame 0

TX packets 4 bytes 234 (234.0 B)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

device interrupt 5

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536

inet 127.0.0.1 netmask 255.0.0.0

loop txqueuelen 1 (Local Loopback)

RX packets 0 bytes 0 (0.0 B)

RX errors 0 dropped 0 overruns 0 frame 0

TX packets 0 bytes 0 (0.0 B)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

En NODE1:

root@node1:~# . /dev/shm/DC0.sh

root@node1:~# cd /usr/src/linux

root@node1:/usr/src/linux# nsenter -p -t $DC0 ./uml.linux con0=null,fd:2 con1=fd:0,fd:1 con=null ssl=null umid=node1 ubda=/dev/sdb6 rw mem=1024M **eth0=tuntap,tap0,02:AA:BB:CC:DD:12** nosysemu

Luego del arranque y loggueo hacer:

root@node1:~# ifconfig

**eth0**: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500

**inet 192.168.1.101** netmask 255.255.255.0 broadcast 192.168.2.255

ether 02:aa:bb:cc:dd:12 txqueuelen 1000 (Ethernet)

RX packets 13 bytes 856 (856.0 B)

RX errors 0 dropped 11 overruns 0 frame 0

TX packets 4 bytes 285 (285.0 B)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

device interrupt 5

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536

inet 127.0.0.1 netmask 255.0.0.0

loop txqueuelen 1 (Local Loopback)

RX packets 0 bytes 0 (0.0 B)

RX errors 0 dropped 0 overruns 0 frame 0

TX packets 0 bytes 0 (0.0 B)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

Haciendo ping en el UML de NODE1 hacia el UML del NODE0

root@node1:~# ping 192.168.1.100

PING 192.168.1.100 (192.168.1.100) 56(84) bytes of data.

64 bytes from 192.168.1.100: icmp\_seq=1 ttl=64 time=26.10 ms

64 bytes from 192.168.1.100: icmp\_seq=2 ttl=64 time=41.10 ms

64 bytes from 192.168.1.100: icmp\_seq=3 ttl=64 time=26.9 ms

64 bytes from 192.168.1.100: icmp\_seq=4 ttl=64 time=39.2 ms

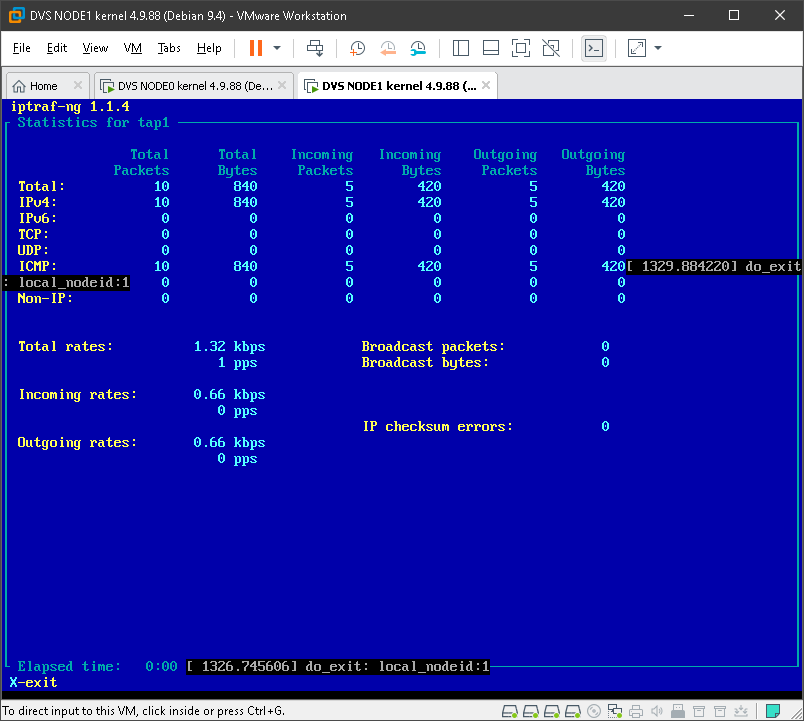
64 bytes from 192.168.1.100: icmp\_seq=5 ttl=64 time=88.6 ms

64 bytes from 192.168.1.100: icmp\_seq=6 ttl=64 time=112 ms

64 bytes from 192.168.1.100: icmp\_seq=7 ttl=64 time=43.3 ms

64 bytes from 192.168.1.100: icmp\_seq=8 ttl=64 time=22.5 ms

El iptraf-ng muestra el trafico en la interface tap1 del NODE1



Haciendo tcpdump en la interface tap1 de NODE1:

root@node1:/tmp# tcpdump -e -n -i tap1 > tcpdump1.txt

Arroja los siguientes logs:

13:39:35.751487 02:aa:bb:cc:dd:b0 > ff:ff:ff:ff:ff:ff, ethertype IPv4 (0x0800), length 98: 192.168.0.100.38650 > 192.168.1.255.4803: UDP, length 56

13:39:37.186932 02:aa:bb:cc:dd:12 > 92:a8:4f:60:70:e8, ethertype IPv4 (0x0800), length 98: 192.168.1.101 > 192.168.1.100: **ICMP echo request**, id 740, seq 1, length 64

13:39:37.223360 92:a8:4f:60:70:e8 > 02:aa:bb:cc:dd:12, ethertype IPv4 (0x0800), length 98: 192.168.1.100 > 192.168.1.101: **ICMP echo reply,** id 740, seq 1, length 64

13:39:38.229134 02:aa:bb:cc:dd:12 > 92:a8:4f:60:70:e8, ethertype IPv4 (0x0800), length 98: 192.168.1.101 > 192.168.1.100: **ICMP echo request**, id 740, seq 2, length 64

13:39:38.297852 92:a8:4f:60:70:e8 > 02:aa:bb:cc:dd:12, ethertype IPv4 (0x0800), length 98: 192.168.1.100 > 192.168.1.101: **ICMP echo reply,** id 740, seq 2, length 64

13:39:39.232622 02:aa:bb:cc:dd:12 > 92:a8:4f:60:70:e8, ethertype IPv4 (0x0800), length 98: 192.168.1.101 > 192.168.1.100: **ICMP echo request**, id 740, seq 3, length 64

13:39:39.272495 92:a8:4f:60:70:e8 > 02:aa:bb:cc:dd:12, ethertype IPv4 (0x0800), length 98: 192.168.1.100 > 192.168.1.101: **ICMP echo reply,** id 740, seq 3, length 64

13:39:40.236763 02:aa:bb:cc:dd:12 > 92:a8:4f:60:70:e8, ethertype IPv4 (0x0800), length 98: 192.168.1.101 > 192.168.1.100: **ICMP echo request**, id 740, seq 4, length 64

13:39:40.278712 92:a8:4f:60:70:e8 > 02:aa:bb:cc:dd:12, ethertype IPv4 (0x0800), length 98: 192.168.1.100 > 192.168.1.101: **ICMP echo reply,** id 740, seq 4, length 64

13:39:41.239497 02:aa:bb:cc:dd:12 > 92:a8:4f:60:70:e8, ethertype IPv4 (0x0800), length 98: 192.168.1.101 > 192.168.1.100: **ICMP echo request**, id 740, seq 5, length 64

13:39:41.348738 92:a8:4f:60:70:e8 > 02:aa:bb:cc:dd:12, ethertype IPv4 (0x0800), length 98: 192.168.1.100 > 192.168.1.101: **ICMP echo reply,** id 740, seq 5, length 64

13:39:42.244657 02:aa:bb:cc:dd:12 > 92:a8:4f:60:70:e8, ethertype IPv4 (0x0800), length 98: 192.168.1.101 > 192.168.1.100: **ICMP echo request**, id 740, seq 6, length 64

HACIENDO UN SSH EN UML NODE1 HACIA UML NODE0

root@node1:/tmp# **ssh 192.168.1.100**

The authenticity of host '192.168.1.100 (192.168.1.100)' can't be established.

ECDSA key fingerprint is SHA256:rqsNVuTbVSj/ahR0obonnY3G3thLNKtidvl+WcyXa80.

Are you sure you want to continue connecting (yes/no)? yes

Warning: Permanently added '192.168.1.100' (ECDSA) to the list of known hosts.

root@192.168.1.100's password:

Linux uml0 4.9.88-DVK #138 Wed Sep 2 01:56:05 -03 2020 i686

The programs included with the Debian GNU/Linux system are free software;

the exact distribution terms for each program are described in the

individual files in /usr/share/doc/\*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent

permitted by applicable law.

Last login: Sun Nov 1 16:28:39 2020

root@uml0:~#

root@uml0:~# hostname

uml0

root@uml0:~# uname -a

Linux uml0 4.9.88-DVK #138 Wed Sep 2 01:56:05 -03 2020 i686 GNU/Linux

root@uml0:~# exit

logout

Connection to 192.168.1.100 closed.

root@node1:/tmp# uname -a

Linux node1 4.9.88-DVK #159 SMP Thu Oct 22 10:23:49 -03 2020 i686 GNU/Linux

# ANALISIS DE LA SALIDA DE DEBUG

Al principio se analiza el archivo de configuración el cual al finalizar deja los siguientes datos importantes (según archivo de configuración ejemplo de arriba). Este ejemplo es la salida de debug en NODE0.

2:switch\_cfg.c:switch\_config:392:rt\_index=0 rt\_nodeid=1 rt\_tap=tap1 rt\_ctrl\_fd=-1 rt\_data\_fd=-1 rt\_poll\_idx=-1 rt\_rmt\_ep=0 rt\_rmt\_idx=0 rt\_rmttap\_fd=-1 rt\_name=RTAP1

2:switch\_cfg.c:switch\_config:402:nr\_switch=1 nr\_rmttap=1

En esta se ve una interface remota RTAP1, cuyo nodo es el nodo 1 y el nombre de la interface en ese nodo es “tap1”.

También hay una interface local LTAP1, cuyo nodo es 0, y el nombre de la interface es “tap1”. Otra información importante es el valor de la variable global nr\_rmttap=2 que indica que hay 2 interfaces remotas configuradas (LTAP1 y RTAP1).

2:dvs\_uml\_switch.c:main:420:sw\_hub=0 sw\_daemon=0 sw\_recv\_ep=27342 sw\_send\_ep=27342 sw\_dcid=0 sw\_ctrl\_path=/tmp/uml.ctl sw\_tap=tap0 sw\_conn\_fd=0 sw\_data\_fd=0 sw\_name=SW\_NAME

Luego se presenta información sobre los parámetros del switch, cual es el DCID, el path del socket de control, etc.

DEBUG 2:dvk\_open:108: Open dvk device file /dev/dvk

2:rmttap.c:get\_dvs\_params:246:

DEBUG 2:dvk\_getdvsinfo:265:

DEBUG 2:dvk\_getdvsinfo:274: ioctl ret=0 errno=0

DEBUG 2:dvk\_getdvsinfo:279: ioctl ret=0

2:rmttap.c:get\_dvs\_params:248:local\_nodeid=0

2:rmttap.c:get\_dvs\_params:251:d\_name=TEST\_CLUSTER d\_nr\_dcs=32 d\_nr\_nodes=32 d\_nr\_procs=221 d\_nr\_tasks=35 d\_nr\_sysprocs=64

2:rmttap.c:get\_dc\_params:261:dcid=0

DEBUG 2:dvk\_getdcinfo:366: dcid=0

DEBUG 2:dvk\_getdcinfo:378: ioctl ret=0 errno=0

DEBUG 2:dvk\_getdcinfo:383: ioctl ret=0

2:rmttap.c:get\_dc\_params:269:dc\_dcid=0 dc\_nr\_procs=221 dc\_nr\_tasks=34 dc\_nr\_sysprocs=64 dc\_nr\_nodes=32

2:rmttap.c:get\_dc\_params:270:flags=0 dc\_nodes=7 dc\_pid=694 dc\_name=DC0

Luego se realiza el OPEN del dispositivo */dev/dvk* para soportar las APIs del DVS. Se toma información del DVS (entre ellos el local\_nodeid) y los diferentes parámetros de configuración. También se obtiene información del DC donde operará el *dvs\_uml\_switch*.

2:dvs\_uml\_switch.c:main:440:sw\_ptr->sw\_conn\_fd=4

2:dvs\_uml\_switch.c:main:456:sw\_ptr->sw\_data\_fd=5

2:dvs\_uml\_switch.c:bind\_sockets:327:ctl\_fd=4 ctl\_name=/tmp/uml.ctl data\_fd=5

2:dvs\_uml\_switch.c:bind\_socket:216:fd=4 name=/tmp/uml.ctl

2:dvs\_uml\_switch.c:bind\_socket:220:sun\_path=/tmp/uml.ctl

2:dvs\_uml\_switch.c:bind\_data\_socket:308:fd=5

2:dvs\_uml\_switch.c:bind\_data\_socket:316:sun\_path=

./dvs\_uml\_switch attached to unix socket '/tmp/uml.ctl' 2:dvs\_uml\_switch.c:add\_fd:43:fd=0

2:dvs\_uml\_switch.c:add\_fd:43:fd=4

2:dvs\_uml\_switch.c:add\_fd:43:fd=5

2:dvs\_uml\_switch.c:main:487:sw\_ptr->sw\_tap=tap0

2:tuntap.c:open\_tap:21:dev=tap0

2:port.c:setup\_port:158:ctrl\_fd=6

New connection

2:tuntap.c:open\_tap:39:dev=tap0 fd=6

2:dvs\_uml\_switch.c:main:490:SWITCH TAP tap0 sw\_tap\_fd=6

2:dvs\_uml\_switch.c:add\_fd:43:fd=6

2:dvs\_uml\_switch.c:main:492:

2:dvs\_uml\_switch.c:main:499:

2:dvs\_uml\_switch.c:main:501:MTX\_LOCK sw\_mutex

2:dvs\_uml\_switch.c:main:509:

2:dvs\_uml\_switch.c:main:516:Timeout

2:dvs\_uml\_switch.c:main:519:MTX\_UNLOCK sw\_mutex

Luego se inicializa el switch sin mayores cambios que en su versión original.

3:rmttap.c:init\_rt\_sockets:677:

3:rmttap.c:init\_rt\_sockets:678:MTX\_LOCK sw\_mutex

DEBUG 2:dvk\_getprocinfo:1177: dcid=0 p\_nr=1

DEBUG 2:dvk\_getprocinfo:1190: ioctl ret=0 errno=0

DEBUG 2:dvk\_getprocinfo:1195: ioctl ret=0

DEBUG 2:dvk\_bind\_X:1221: cmd=2 dcid=0 pid=4272208 endpoint=1 nodeid=1

DEBUG 2:dvk\_bind\_X:1236: ioctl ret=1 errno=0

DEBUG 2:dvk\_bind\_X:1251: ioctl ret=1

DEBUG 2:dvk\_bind\_X:1254: ret=1

DEBUG 2:dvk\_getprocinfo:1177: dcid=0 p\_nr=1

DEBUG 2:dvk\_getprocinfo:1190: ioctl ret=0 errno=0

DEBUG 2:dvk\_getprocinfo:1195: ioctl ret=0

3:rmttap.c:init\_rt\_sockets:705:nr=1 endp=1 dcid=0 flags=1000 misc=0 lpid=-1 vpid=-1 nodeid=1 name=TAPserver01

DEBUG 2:dvk\_getprocinfo:1177: dcid=0 p\_nr=31

DEBUG 2:dvk\_getprocinfo:1190: ioctl ret=0 errno=0

DEBUG 2:dvk\_getprocinfo:1195: ioctl ret=0

DEBUG 2:dvk\_bind\_X:1221: cmd=2 dcid=0 pid=4272208 endpoint=31 nodeid=1

DEBUG 2:dvk\_bind\_X:1236: ioctl ret=31 errno=0

DEBUG 2:dvk\_bind\_X:1251: ioctl ret=31

DEBUG 2:dvk\_bind\_X:1254: ret=31

DEBUG 2:dvk\_getprocinfo:1177: dcid=0 p\_nr=31

DEBUG 2:dvk\_getprocinfo:1190: ioctl ret=0 errno=0

DEBUG 2:dvk\_getprocinfo:1195: ioctl ret=0

3:rmttap.c:init\_rt\_sockets:723:nr=31 endp=31 dcid=0 flags=1000 misc=0 lpid=-1 vpid=-1 nodeid=1 name=TAPclient01

DEBUG 2:dvk\_getprocinfo:1177: dcid=0 p\_nr=2

DEBUG 2:dvk\_getprocinfo:1190: ioctl ret=0 errno=0

DEBUG 2:dvk\_getprocinfo:1195: ioctl ret=0

DEBUG 2:dvk\_bind\_X:1221: cmd=2 dcid=0 pid=4272208 endpoint=2 nodeid=2

DEBUG 2:dvk\_bind\_X:1236: ioctl ret=2 errno=0

DEBUG 2:dvk\_bind\_X:1251: ioctl ret=2

DEBUG 2:dvk\_bind\_X:1254: ret=2

DEBUG 2:dvk\_getprocinfo:1177: dcid=0 p\_nr=2

DEBUG 2:dvk\_getprocinfo:1190: ioctl ret=0 errno=0

DEBUG 2:dvk\_getprocinfo:1195: ioctl ret=0

3:rmttap.c:init\_rt\_sockets:705:nr=2 endp=2 dcid=0 flags=1000 misc=0 lpid=-1 vpid=-1 nodeid=2 name=TAPserver02

DEBUG 2:dvk\_getprocinfo:1177: dcid=0 p\_nr=32

DEBUG 2:dvk\_getprocinfo:1190: ioctl ret=0 errno=0

DEBUG 2:dvk\_getprocinfo:1195: ioctl ret=0

DEBUG 2:dvk\_bind\_X:1221: cmd=2 dcid=0 pid=4272208 endpoint=32 nodeid=2

DEBUG 2:dvk\_bind\_X:1236: ioctl ret=32 errno=0

DEBUG 2:dvk\_bind\_X:1251: ioctl ret=32

DEBUG 2:dvk\_bind\_X:1254: ret=32

DEBUG 2:dvk\_getprocinfo:1177: dcid=0 p\_nr=32

DEBUG 2:dvk\_getprocinfo:1190: ioctl ret=0 errno=0

DEBUG 2:dvk\_getprocinfo:1195: ioctl ret=0

3:rmttap.c:init\_rt\_sockets:723:nr=32 endp=32 dcid=0 flags=1000 misc=0 lpid=-1 vpid=-1 nodeid=2 name=TAPclient02

3:rmttap.c:init\_rt\_sockets:746:rt\_index=0 rt\_nodeid=1 rt\_tap=tap1 rt\_ctrl\_fd=-1 rt\_data\_fd=-1 rt\_poll\_idx=-1 rt\_rmt\_ep=-1 rt\_rmt\_idx=-1 rt\_rmttap\_fd=-1 rt\_name=RTAP1

3:rmttap.c:rmttap\_init:17:rt\_name=RTAP1

3:rmttap.c:rmttap\_init:30:rt\_ptr->rt\_ctrl\_path=/tmp/uml.ctl

3:rmttap.c:rmttap\_init:36:rt\_ptr->rt\_dd.ctl\_sock=/tmp/uml.ctl

3:rmttap.c:connect\_to\_switch:88:rt\_name=RTAP1

2:dvs\_uml\_switch.c:accept\_connection:171:fd=4

2:dvs\_uml\_switch.c:add\_fd:43:fd=8

3:rmttap.c:connect\_to\_switch:126:write to switch RTAP1 magic=-17958194 version=3 type=0

2:port.c:handle\_port:263:fd=8

2:dvs\_uml\_switch.c:new\_port:138:ctrl\_fd=8 data\_fd=5

2:dvs\_uml\_switch.c:new\_port\_v1\_v3:108:ctrl\_fd=8 type=0 data\_fd=5

2:port.c:setup\_sock\_port:235:ctrl\_fd=8 data\_fd=5

2:port.c:setup\_port:158:ctrl\_fd=8

New connection

2:dvs\_uml\_switch.c:new\_port\_v1\_v3:114:fd=8 sun.sun\_path=

3:rmttap.c:connect\_to\_switch:135:read from switch RTAP1

3:rmttap.c:connect\_to\_switch:143:rt\_index=0 rt\_nodeid=1 rt\_tap=tap1 rt\_ctrl\_fd=-1 rt\_data\_fd=-1 rt\_poll\_idx=-1 rt\_rmt\_ep=-1 rt\_rmt\_idx=-1 rt\_rmttap\_fd=-1 rt\_name=RTAP1

3:rmttap.c:rmttap\_init:48:sock\_type=unix ctl\_sock=/tmp/uml.ctl fd=9 control=7

3:rmttap.c:init\_rt\_sockets:753:rt\_index=0 rt\_nodeid=1 rt\_tap=tap1 rt\_ctrl\_fd=7 rt\_data\_fd=9 rt\_poll\_idx=-1 rt\_rmt\_ep=-1 rt\_rmt\_idx=-1 rt\_rmttap\_fd=-1 rt\_name=RTAP1

DEBUG 2:dvk\_getprocinfo:1177: dcid=0 p\_nr=1

DEBUG 2:dvk\_getprocinfo:1190: ioctl ret=0 errno=0

DEBUG 2:dvk\_getprocinfo:1195: ioctl ret=0

DEBUG 2:dvk\_getprocinfo:1177: dcid=0 p\_nr=1

DEBUG 2:dvk\_getprocinfo:1190: ioctl ret=0 errno=0

DEBUG 2:dvk\_getprocinfo:1195: ioctl ret=0

3:rmttap.c:init\_rt\_sockets:767:rt\_index=0 rt\_nodeid=1 rt\_tap=tap1 rt\_ctrl\_fd=7 rt\_data\_fd=9 rt\_poll\_idx=0 rt\_rmt\_ep=-1 rt\_rmt\_idx=-1 rt\_rmttap\_fd=-1 rt\_name=RTAP1

3:rmttap.c:init\_rt\_sockets:774:rt\_nfds=1 nr\_lcltap=0

3:rmttap.c:init\_rt\_sockets:781:MTX\_UNLOCK sw\_mutex

Posteriormente se realiza el dvk\_bind de Senders y Receivers de los nodos remotos y cada puerto se conecta con un unix sockets al switch (*connect\_to\_switch()*).

5:rmttap.c:recv\_thread:449:RECEIVER THREAD: sw\_recv\_ep=0 sw\_recv\_tid=5

DEBUG 2:dvk\_bind\_X:1221: cmd=0 dcid=0 pid=5 endpoint=0 nodeid=-1

DEBUG 2:dvk\_bind\_X:1236: ioctl ret=0 errno=0

DEBUG 2:dvk\_bind\_X:1251: ioctl ret=0

DEBUG 2:dvk\_bind\_X:1254: ret=0

DEBUG 2:dvk\_getprocinfo:1177: dcid=0 p\_nr=0

DEBUG 2:dvk\_getprocinfo:1190: ioctl ret=0 errno=0

DEBUG 2:dvk\_getprocinfo:1195: ioctl ret=0

5:rmttap.c:recv\_thread:458:nr=0 endp=0 dcid=0 flags=0 misc=0 lpid=747 vpid=5 nodeid=0 name=dvs\_uml\_switch

DEBUG 2:dvk\_receive\_T:897: endpoint=31438 timeout=30000

Se arranca el Receiver Thread quien escucha en el endpoint Server sobre mensajes remotos hacia el switch local (recibe por M3IPC y escribe por unix socket).

4:rmttap.c:send\_thread:321:SENDER THREAD: sw\_send\_ep=30 sw\_send\_tid=4

DEBUG 2:dvk\_bind\_X:1221: cmd=0 dcid=0 pid=4 endpoint=30 nodeid=-1

DEBUG 2:dvk\_bind\_X:1236: ioctl ret=30 errno=0

DEBUG 2:dvk\_bind\_X:1251: ioctl ret=30

DEBUG 2:dvk\_bind\_X:1254: ret=30

DEBUG 2:dvk\_getprocinfo:1177: dcid=0 p\_nr=30

DEBUG 2:dvk\_getprocinfo:1190: ioctl ret=0 errno=0

DEBUG 2:dvk\_getprocinfo:1195: ioctl ret=0

4:rmttap.c:send\_thread:331:nr=30 endp=30 dcid=0 flags=0 misc=0 lpid=746 vpid=4 nodeid=0 name=dvs\_uml\_switch

4:rmttap.c:try\_rmttap\_open:281:nr\_rmttap=1

4:rmttap.c:try\_rmttap\_open:287:rt\_index=0 rt\_nodeid=1 rt\_tap=tap1 rt\_ctrl\_fd=7 rt\_data\_fd=9 rt\_poll\_idx=0 rt\_rmt\_ep=-1 rt\_rmt\_idx=-1 rt\_rmttap\_fd=-1 rt\_name=RTAP1

4:rmttap.c:rt\_open\_rmttap:211:rt\_index=0 rt\_nodeid=1 rt\_tap=tap1 rt\_ctrl\_fd=7 rt\_data\_fd=9 rt\_poll\_idx=0 rt\_rmt\_ep=-1 rt\_rmt\_idx=-1 rt\_rmttap\_fd=-1 rt\_name=RTAP1

DEBUG 2:dvk\_getprocinfo:1177: dcid=0 p\_nr=1

DEBUG 2:dvk\_getprocinfo:1190: ioctl ret=0 errno=0

DEBUG 2:dvk\_getprocinfo:1195: ioctl ret=0

4:rmttap.c:rt\_open\_rmttap:217:nr=1 endp=1 dcid=0 flags=1000 misc=0 lpid=-1 vpid=-1 nodeid=1 name=TAPserver01

4:rmttap.c:rt\_open\_rmttap:223:rt\_ptr->rt\_nodeid=1

4:rmttap.c:rt\_open\_rmttap:224:SEND REQ\_RT\_OPEN REQUEST: source=0 type=1 m3i1=0 m3i2=0 m3p1=(nil) m3ca1=[tap1]

DEBUG 2:dvk\_sendrec\_T:931: endpoint=1 timeout=30000

Se arranca el Sender Thread quien trata de hacer el OPEN de las interfaces TAP remotas.

4:rmttap.c:rt\_open\_rmttap:217:nr=1 endp=1 dcid=0 flags=1000 misc=0 lpid=-1 vpid=-1 nodeid=1 name=TAPserver01

4:rmttap.c:rt\_open\_rmttap:223:rt\_ptr->rt\_nodeid=1

4:rmttap.c:rt\_open\_rmttap:224:SEND REQ\_RT\_OPEN REQUEST: source=30 type=1 m3i1=0 m3i2=0 m3p1=(nil) m3ca1=[tap1]

……

4:rmttap.c:rt\_open\_rmttap:229:RCVD REQ\_RT\_OPEN REPLY: source=1 type=0 m3i1=0 m3i2=0 m3p1=(nil) m3ca1=[tap1]

4:rmttap.c:rt\_open\_rmttap:236:Remote TAP tap1 successfully opened on endpoint 1 with remote index 0

4:rmttap.c:try\_rmttap\_open:296:success\_open=1

Luego se trata de hacer un WRITE

4:rmttap.c:send\_thread:364:RTAP1 rt\_poll\_idx=0

4:rmttap.c:send\_thread:397:Dst: ff:ff:ff:ff:ff:ff

4:rmttap.c:send\_thread:398:Src: 92:a8:4f:60:70:e8 proto:41246c

4:rmttap.c:rt\_send\_write\_packet:166:rt\_name=RTAP1 wtype=2 len=42 rt\_rmt\_idx=0

4:rmttap.c:rt\_send\_write\_packet:177:rt\_index=0 rt\_nodeid=1 rt\_tap=tap1 rt\_ctrl\_fd=7 rt\_data\_fd=9 rt\_poll\_idx=0 rt\_rmt\_ep=1 rt\_rmt\_idx=0 rt\_rmttap\_fd=-1 rt\_name=RTAP1

4:rmttap.c:rt\_send\_write\_packet:185:REQ\_RT\_WRITE: Dst: 60:24:41:00:2a:00

4:rmttap.c:rt\_send\_write\_packet:186:REQ\_RT\_WRITE: Src: 00:00:00:00:00:00 proto:b6d2b304

4:rmttap.c:rt\_send\_write\_packet:189:SEND WRITE PACKET REQUEST: svr\_ep=1 source=1 type=2 m3i1=0 m3i2=42 m3p1=0x412460 m3ca1=[tap1]

DEBUG 2:dvk\_sendrec\_T:931: endpoint=1 timeout=30000

DEBUG 2:dvk\_sendrec\_T:944: ioctl ret=0 errno=0

DEBUG 2:dvk\_sendrec\_T:953: ioctl ret=0

4:rmttap.c:rt\_send\_write\_packet:195:RECEIVED WRITE PACKET REPLY: source=1 type=0 m3i1=0 m3i2=42 m3p1=0x412460 m3ca1=[tap1]

# OPTIMIZACIONES PENDIENTES

***OPTIMIZACION 1:*** El send\_thread(), cuando se activa un FD via poll(), hace la lectura de un **único paquete**.

Si el destinatario es un Remote TAP

len = recvfrom(rt\_ptr->rt\_data\_fd, &packet, sizeof(packet), 0, (struct sockaddr \*) &rt\_ptr->rt\_data\_sun, &socklen);

Si el destinatario es un Remote Switch

len = read(rt\_ptr->rt\_rmttap\_fd, &packet, sizeof(packet));

Habría que leer y acumular paquetes hasta el vaciar el unix socket o llegar al máximo soportado por una transferencia M3-IPC.

Dvs\_ptr->d\_max\_copybuf (tamaño máximo del buffer)

Dvs\_ptr->d\_max\_copylen (Tamaño máximo a transferir en una copia)

Los valores por default son:

#define MAXCOPYBUF 65536

#define MAXCOPYLEN (16 \* MAXCOPYBUF)

Siendo el formato de un paquete:

struct packet {

struct {

unsigned char dest[ETH\_ALEN];

unsigned char src[ETH\_ALEN];

unsigned char proto[2];

} header;

unsigned char data[1500];

};

Habría que hacer un formato tipo frame:

struct {

int fr\_len;

struct packet fr\_pkt;

} frame\_s;

typedef frame\_s frame\_t;

Para esto, quizás haya que cambiar el tipo de mensaje que se transfiere que hoy es tipo 3, por tipo 7, donde ***m\_type*** es el tipo de comando.

m7\_i1: indice de la rmttap en el nodo remoto

m7\_i2: len, packet len : deberia cambiarse a total len del buffer

m7\_i3: número de frames

m7\_i4: SIN USO

m7\_p1: packet cambiaria a buffer de frames.

m7\_p2: SIN USO

***OPTIMIZACION 2***:

En el recv\_thread, cuando llega un paquete desde el nodo remoto, se escribe en el FD local y al final se hace el REPLY .

Se podría hacer lo siguiente: una vez hecho el vcopy() que copia los datos desde el nodo remoto al nodo local, se hace el reply, y luego se escribe en el FD local, porque no es necesario retornarle el NODO remoto el resultado de la escritura en el FD local.

De esta forma se libera al sender thread remoto para que pueda enviar mensajes a este u otro nodo.