**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 estándar.



1. UML Switch estándar.

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 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 y de las interfaces TAP locales asociadas al DVS UML Switch remoto y de enviarlos al thread Receiver del switch del nodo remoto correspondiente.

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 DVS UML 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 RTAP2 {

tap "TAP1";

nodeid 2;

};

rmttap LTAP1 {

tap "TAP1";

nodeid 0;

};

En este archivo de ejemplo se establece la configuración correspondiente al Switch del NODO0 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 nodo 1.
* Luego, se especifica una interface TAP remota (RTAP2) que se asociará a la interface TAP1 del nodo 2.
* Finalmente se especifica la interface local LTAP1 asociada a la interface TAP1 del Host 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 FUNCIONAL EL 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 FUNCIONANA *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 del poll() que utilizará el hilo Sender.

Cada uno de los nodos remotos se registra en el DVS para poder utilizar M3IPC (*dvk\_rmtbind()*) entre el nodo local y el remoto.

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

* **SENDER**: Monitorea con *poll()* todos los FD 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\_idx*). 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 número de interface TAP (*rt\_rmttap\_idx*) 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 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 paquetes de WRITE de frames 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 da setea como inválido el *rt\_rmttap\_idx* hasta tanto se vuelva a conectar el nodo remoto y el OPEN sea exitoso*.*

# 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

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

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

char \*sw\_data\_path; // switch data 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\_rmttap\_idx; // rmttap index in the remote node

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\_proc; // remote switch server process descriptor

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

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;

# MIGRACION DE PROCESOS: ESCENARIO 1

**Escenario PREVIO**

**migr\_server: NODE0**

**migr\_client: NODE1**

**Escenario FINAL:**

**migr\_server: NODE1**

**migr\_client: NODE1**

A los efectos de mostrar el funcionamiento se utilizarán múltiples terminales en cada uno de los Nodos.

|  |  |
| --- | --- |
| **NODE0** | **NODE1** |
| * /usr/src/lkl/dmtcp/bin/dmtcp\_coordinator -p 1234 –-daemon -q |  |
| * cd /usr/src/dvs/dvk-tests/ * ./tests.sh 0 0 * . /dev/shm/DC0.sh * nsenter -p -t$DC0 ./test\_rmtbind 0 11 1 migr\_client * cd /usr/src/dvs/dvs-apps/dvs\_run * rm \*.dmtcp | * cd /usr/src/dvs/dvk-tests/ * ./tests.sh 1 0 * . /dev/shm/DC0.sh * nsenter -p -t$DC0 ./test\_rmtbind 0 10 0 migr\_server * cd /usr/src/dvs/dvs-apps/dvs\_run * rm \*.dmtcp |
| root@node0:/usr/src/dvs/dvs-apps/dvs\_run# cat /proc/dvs/DC0/procs  DC pnr -endp -lpid/vpid- nd flag misc -getf -sndt -wmig -prxy name  0 11 11 -1/-1 1 1000 0 27342 27342 27342 27342 migr\_client | root@node1:/usr/src/dvs/dvs-apps/dvs\_run# cat /proc/dvs/DC0/procs  DC pnr -endp -lpid/vpid- nd flag misc -getf -sndt -wmig -prxy name  0 10 10 -1/-1 0 1000 0 27342 27342 27342 27342 migr\_server |
| * /usr/src/lkl/dmtcp/bin/dmtcp\_launch -h node0 -p 1234 ./migr\_server 0 10 & |  |
|  | * nsenter -p -t$DC0 ./migr\_client 0 11 10 4096 100 1 & |
|  | * ./dvs\_migrate -s 0 10 |
| * ./dvs\_migrate -s 0 10 |  |
| root@node0:/usr/src/dvs/dvs-apps/dvs\_run# cat /proc/dvs/DC0/procs  DC pnr -endp -lpid/vpid- nd flag misc -getf -sndt -wmig -prxy name  0 10 10 939/939 0 800 20 27342 27342 27342 27342 migr\_server  0 11 11 -1/-1 1 1000 0 27342 27342 27342 27342 migr\_client | root@node1:/usr/src/dvs/dvs-apps/dvs\_run# cat /proc/dvs/DC0/procs  DC pnr -endp -lpid/vpid- nd flag misc -getf -sndt -wmig -prxy name  0 10 10 -1/-1 0 1800 0 27342 27342 27342 27342 migr\_server  0 11 11 867/3 1 8000 20 27342 27342 10 27342 migr\_client |
| * /usr/src/lkl/dmtcp/bin/dmtcp\_command -h node0 -p 1234 -bc * mv ckpt\_\*.dmtcp checkpoint\_migr\_server.dmtcp |  |
| * sshpass -p "root" scp -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -r checkpoint\_migr\_server.dmtcp root@node1:/usr/src/dvs/dvs-apps/dvs\_run |  |
|  | * dmtcp\_restart checkpoint\_migr\_server.dmtcp & |
| root@node0:~# cat /proc/dvs/DC0/procs  DC pnr -endp -lpid/vpid- nd flag misc -getf -sndt -wmig -prxy name  0 11 11 -1/-1 1 1000 0 27342 27342 27342 27342 migr\_client | root@node1:~# cat /proc/dvs/DC0/procs  DC pnr -endp -lpid/vpid- nd flag misc -getf -sndt -wmig -prxy name  0 10 10 889/-1 1 8 0 31438 27342 27342 27342 DMTCP:migr\_serv  0 11 11 867/3 1 0 20 27342 27342 27342 27342 migr\_client |
|  | * ./dvs\_migrate -s 0 10 |
|  | root@node1:/usr/src/dvs/dvs-apps/dvs\_run# cat /proc/dvs/DC0/procs  DC pnr -endp -lpid/vpid- nd flag misc -getf -sndt -wmig -prxy name  0 10 10 889/-1 1 800 0 27342 27342 27342 27342 DMTCP:migr\_serv  0 11 11 867/3 1 8000 20 27342 27342 10 27342 migr\_client |
|  | * ./dvs\_migrate -r 0 10 |
|  | root@node1:/usr/src/dvs/dvs-apps/dvs\_run# cat /proc/dvs/DC0/procs  DC pnr -endp -lpid/vpid- nd flag misc -getf -sndt -wmig -prxy name  0 10 10 889/-1 1 8 0 31438 27342 27342 27342 DMTCP:migr\_serv  0 11 11 867/3 1 0 20 27342 27342 27342 27342 migr\_client |

1. En NODE0 se arranca el coordinador de DMTCP porque allí se encuentra el servidor. Este coordinador se podría arrancar en todos los nodos del DVS.
2. En NODE0 se inicial el DVS, se configura y se inicia el DC0 en donde van a ejecutar los procesos de pruebas. Se arrancan los proxies contra el NODE1 y se adiciona el NODE1 al DC0. Esto quiere decir que desde el NODE0 se pueden referencia procesos del DC0 que se encuentren en el NODE1. Luego se setea la variable de entorno $DC0 que contiene el PID del proceso asociado al DC0. También se establece que el proceso ***migr\_client*** es un proceso REMOTO (flag=1000) y se encuentra en el NODE1 (nd=1). Luego, se borran del directorio de trabajo todos los archivos de checkpoint del DMTCP.
3. En NODE1, se inicializan el DVS, el DC0, se arrancan los proxies contra NODE0 y se establece que en NODE0 pueden ejecutar procesos del DC0. También se establece que el proceso ***migr\_server*** es un proceso REMOTO (flag=1000) y se encuentra en el NODE0 (nd=0). Luego, se borran del directorio de trabajo todos los archivos de checkpoint del DMTCP.
4. En NODE0, se arranca a través del lanzador de DMTCP la ejecución del ***migr\_server***.
5. En NODE1, se arranca utilizando ***nsenter*** la ejecución del ***migr\_client***. En este momento comienzan a transferir datos entre ambos procesos localizados en diferentes nodos.
6. En NODE1, se le notifica al DVK que el proceso ***migr\_server*** (dcid=0, endpoint=10) va a iniciar (start) un proceso de migración (flag=1800). En este momento se detienen las comunicaciones y el ***migr\_client*** queda en espera de la finalización de la migración (flag=8000). **ATENCION: Este comando de migración SIEMPRE debe ejecutarse primero en los nodos en donde se encuentran los procesos Client, de lo contrario, al no responder el Server, el Client puede dar por finalizada la comunicación**.
7. En NODE0, se le notifica al DVK que el proceso ***migr\_server*** (dcid=0, endpoint=10) va a iniciar (start) un proceso de migración.
8. En NODE0, se le solicita al coordinador de DMTCP realizar un Checkpoint bloqueante (bc) lo que provoca que se genere el archivo de checkpoint. Luego, se normaliza el nombre del archivo de checkpoint.
9. En NODE0, se transfiere el archivo de Checkpoint al NODO1 (nodo destino) utilizando ***ssh***. Para automatizar la transferencia se utiliza ***sshpass*** de tal forma de que no solicite la password del usuario de NODE1.
10. En NODE1, se rearranca el proceso ***migr\_server*** a partir de la imagen del DMTCP. El ***migr\_server*** está preparado para la migración que le solicita al DVK que lo considere migrado en el NODO1 (*dvk\_migr\_commit()*). Esta operación reinicia automáticamente las comunicaciones de todos los procesos que se estaban comunicando con el ***migr\_server*** y que fueron detenidos al realizar *dvk\_migr\_start().* En este momento se puede ver que las comunicaciones continúan entre Client y Server.
11. En NODE1. Al solo efecto de probar que ocurriría ante una migración fallida, se notifica al DVK que se inicia una migración del proceso de Endpoint=10. En este momento se vuelven a detener las comunicaciones.
12. En NODE1, se simula una migración fallida y que el proceso Server continuará ejecutando en su nodo origen (ahora NODE1). Para ello se le comunica el DVK que realice un Rollback (*dvk\_migr\_rollback()*) y reinicie las comunicaciones.

# MIGRACION DE PROCESOS: ESCENARIO 2

**Escenario PREVIO**

**migr\_server: NODE1**

**migr\_client: NODE1**

**Escenario FINAL:**

**migr\_server: NODE0**

**migr\_client: NODE1**

A los efectos de mostrar el funcionamiento se utilizarán múltiples terminales en cada uno de los Nodos.

|  |  |
| --- | --- |
| **NODE0** | **NODE1** |
|  | * /usr/src/lkl/dmtcp/bin/dmtcp\_coordinator -p 1234 –-daemon -q |
| * cd /usr/src/dvs/dvk-tests/ * ./tests.sh 0 0 * . /dev/shm/DC0.sh * nsenter -p -t$DC0 ./test\_rmtbind 0 11 1 migr\_client * nsenter -p -t$DC0 ./test\_rmtbind 0 10 1 migr\_server * cd /usr/src/dvs/dvs-apps/dvs\_run * rm \*.dmtcp | * cd /usr/src/dvs/dvk-tests/ * ./tests.sh 1 0 * . /dev/shm/DC0.sh * cd /usr/src/dvs/dvs-apps/dvs\_run * rm \*.dmtcp |
| root@node0:/usr/src/dvs/dvs-apps/dvs\_run# cat /proc/dvs/DC0/procs  DC pnr -endp -lpid/vpid- nd flag misc -getf -sndt -wmig -prxy name  0 10 10 -1/-1 1 1000 0 27342 27342 27342 27342 migr\_server  0 11 11 -1/-1 1 1000 0 27342 27342 27342 27342 migr\_client |  |
|  | * /usr/src/lkl/dmtcp/bin/dmtcp\_launch -h node1 -p 1234 ./migr\_server 0 10 |
|  | * nsenter -p -t$DC0 ./migr\_client 0 11 10 4096 100 1 & |
|  | * ./dvs\_migrate -s 0 10 |
| * ./dvs\_migrate -s 0 10 |  |
| root@node0:/usr/src/dvs/dvs-apps/dvs\_run# cat /proc/dvs/DC0/procs  DC pnr -endp -lpid/vpid- nd flag misc -getf -sndt -wmig -prxy name  0 10 10 -1/-1 1 1800 0 27342 27342 27342 27342 migr\_server  0 11 11 -1/-1 1 1000 0 27342 27342 27342 27342 migr\_client | root@node1:/usr/src/dvs/dvs-apps/dvs\_run# cat /proc/dvs/DC0/procs  DC pnr -endp -lpid/vpid- nd flag misc -getf -sndt -wmig -prxy name  0 10 10 801/801 1 800 20 27342 27342 27342 27342 migr\_server  0 11 11 824/2 1 8000 20 27342 27342 10 27342 migr\_client |
|  | * /usr/src/lkl/dmtcp/bin/dmtcp\_command -h node1 -p 1234 –bc * mv ckpt\_\*.dmtcp checkpoint\_migr\_server.dmtcp |
|  | * sshpass -p "root" scp -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -r checkpoint\_migr\_server.dmtcp root@node0:/usr/src/dvs/dvs-apps/dvs\_run |
| * dmtcp\_restart checkpoint\_migr\_server.dmtcp |  |
| root@node0:~# cat /proc/dvs/DC0/procs  DC pnr -endp -lpid/vpid- nd flag misc -getf -sndt -wmig -prxy name  0 10 10 722/-1 0 8 0 31438 27342 27342 27342 DMTCP:migr\_serv  0 11 11 -1/-1 1 1000 0 27342 27342 27342 27342 migr\_client |  |
|  | * ./dvs\_migrate -c 0 10 0 |
|  | root@node1:~# cat /proc/dvs/DC0/procs  DC pnr -endp -lpid/vpid- nd flag misc -getf -sndt -wmig -prxy name  0 10 10 -1/-1 0 1000 0 27342 27342 27342 27342 migr\_server  0 11 11 824/2 1 0 20 27342 27342 27342 27342 migr\_client |

1. En NODE1 se arranca el coordinador de DMTCP porque allí se encuentra el servidor. Este coordinador se podría arrancar en todos los nodos del DVS.
2. En NODE1 se inicial el DVS, se configura y se inicia el DC0 en donde van a ejecutar los procesos de pruebas. Se arrancan los proxies contra el NODE0 y se adiciona el NODE0 al DC0. Esto quiere decir que desde el NODE1 se pueden referencia procesos del DC0 que se encuentren en el NODE0. Luego se setea la variable de entorno $DC0 que contiene el PID del proceso asociado al DC0. Se borran del directorio de trabajo todos los archivos de checkpoint del DMTCP.
3. En NODE0, se inicializan el DVS, el DC0, se arrancan los proxies contra NODE1 y se establece que en NODE1 pueden ejecutar procesos del DC0. También se establece que los procesos ***migr\_server*** y ***migr\_clientr*** son procesos REMOTOS (flag=1000) y se encuentran ejecutando en el NODE1 (nd=1). Luego, se borran del directorio de trabajo todos los archivos de checkpoint del DMTCP.
4. En NODE1, se arranca a través del lanzador de DMTCP la ejecución del ***migr\_server***. También se arranca utilizando ***nsenter*** la ejecución del ***migr\_client***. En este momento comienzan a transferir datos entre ambos procesos localizados en el mismo nodo.
5. En NODE1, se le notifica al DVK que el proceso ***migr\_server*** (dcid=0, endpoint=10) va a iniciar (start) un proceso de migración (flag=1800). En este momento se detienen las comunicaciones y el ***migr\_client*** queda en espera de la finalización de la migración (flag=8000). **ATENCION: Este comando de migración SIEMPRE debe ejecutarse primero en los nodos en donde se encuentran los procesos Client, de lo contrario, al no responder el Server, el Client puede dar por finalizada la comunicación**.
6. En NODE0, se le notifica al DVK que el proceso ***migr\_server*** (dcid=0, endpoint=10) va a iniciar (start) un proceso de migración.
7. En NODE1, se le solicita al coordinador de DMTCP realizar un Checkpoint bloqueante (bc) lo que provoca que se genere el archivo de checkpoint. Luego, se normaliza el nombre del archivo de checkpoint.
8. En NODE1, se transfiere el archivo de Checkpoint al NODO0 (nodo destino) utilizando ***ssh***. Para automatizar la transferencia se utiliza ***sshpass*** de tal forma de que no solicite la password del usuario de NODE0.
9. En NODE0, se rearranca el proceso ***migr\_server*** a partir de la imagen del DMTCP. El ***migr\_server*** está preparado para la migración que le solicita al DVK que lo considere migrado en el NODO1 (*dvk\_migr\_commit()*). Esta operación reinicia automáticamente las comunicaciones de todos los procesos que se estaban comunicando con el ***migr\_server*** y que fueron detenidos al realizar *dvk\_migr\_start().* Ahora, ***migr\_server*** es un proceso local del NODE0 (nd=0) que está detenido esperando recibir un mensaje de algún cliente (flag=8).
10. En NODE1, se le notifica al DVK que la migración ha resultado exitosa (*dvk\_migr\_commit()*). Ahora el ***migr\_server*** es considerado un proceso REMOTO (flag=1000) localizado en el NODE0 (nd=0). En este momento las comunicaciones continúan entre Client y Server. **ATENCION: Este comando de migración SIEMPRE debe ejecutarse primero en el nodo en donde se encuentra el procesos Server, de lo contrario, si se arranca primero en los Client el server no responderá y el Client puede dar por finalizada la comunicación.**