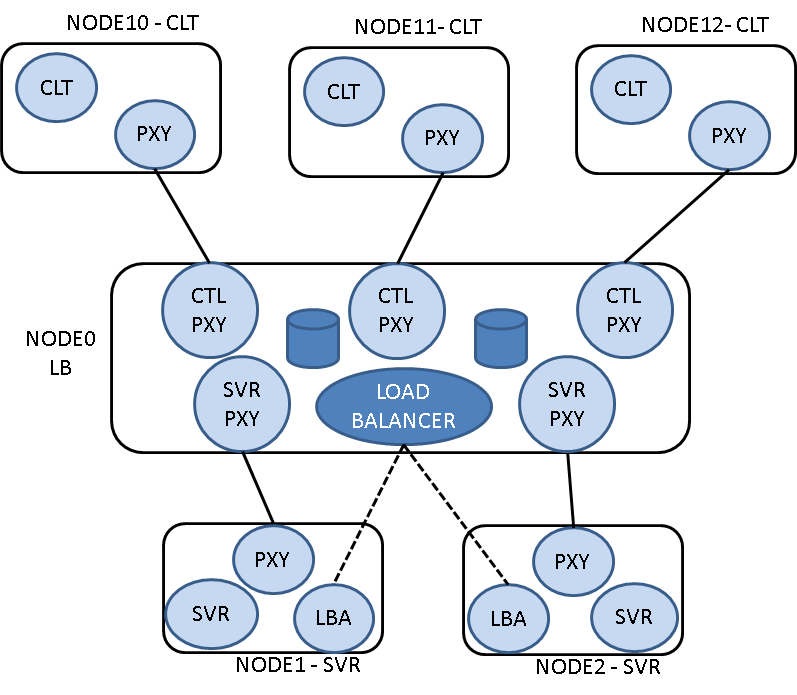
**ESPECIFICACIONES LOAD BALANCER – VERSION 2**

**Objetivo:** que el LB distribuya las sesiones M3-IPC iniciadas por Clients (CLT) entre diferentes Servers (SVR).Que realice autoscaling y auto-running. Con autoscaling levanta y baja VMs, con autorunning arranca y detiene procesos servidores en el nodo servidor, algo similar a function as a service.

**TOPOLOGIA**



En el DC utilizado hay:

* *NR\_SVR\_NODES*: Que van desde (1, (*MAX\_SVR\_NODES*))
* *LB\_NODEID*: Nodo del LB=NODE0
* *NR\_CLT\_NODES*: Que van desde ((*MAX\_SVR\_NODES* +1), (DC\_NRNODES-1))

**CLIENTES:**

Cada uno de los CLT registra (*lclbind*) su propio Endpoint *clt\_endpoint* en el mismo DC, estando en nodos diferentes y tienen registrado el Endpoint *svr\_ endpoint* del SVR en el LB\_NODE.

// Da de alta el nodo del LB como parte del DC

Ret = dvk\_add\_node(dcid, lb\_nodeid);

// BIND del CLIENT

Ret = dvk\_bind(dcid, clt\_endpoint);

// BIND del SERVER (apuntando al nodo del LB)

Ret = dvk\_rmtbind(dcid, “rmt\_server”, svr\_endpoint, lb\_nodeid);

De esta manera se podrían conformar múltiples pares idénticos {*clt\_endpoint, svr\_endpoint*}, uno por cada CLT\_NODE.

**SERVERS:**

Cada uno de los SVR registra (*lclbind*) su propio Endpoint SVR\_ENDPOINT en el mismo DC, estando en nodos diferentes (tipo Replica REPLICA\_BIND) y tienen registrados TODOS los CLT Endpoints en el LB\_NODE como remotos (*rmtbind*). Esto último puede no ser necesario si se habilita el AUTOBINDING del proxy del multi\_proxy de los SVR nodes.

En primer término se registran manualmente los Endpoints de clientes en los SVRs:

// Da de alta el nodo del LB como parte del DC

Ret = dvk\_add\_node(dcid, lb\_nodeid);

// BIND del SERVER

Ret = dvk\_bind(dcid, svr\_endpoint);

// registra todos los endpoint CLT en el nodo del LB

for( clt\_ep = dc\_nr\_sysprocs; clt\_ep < (dc\_nr\_procs-dc\_nr\_task); clt\_ep++){

sprint(clt\_name;”client\_%0.3d”, clt\_ep);

ret = dvk\_bind(dcid, clt\_name; clt\_ep, lb\_nodeid);

}

**LOAD BALANCER:**

En principio se creo el multi\_proxy en modificando *lz4tcp\_proxy\_bat*  para que utilice threads en lugar de procesos. NOTA: La funcionalidad de BATCH de mensajes no se puede usar en la versión actual del LB porque si el LB recibe un batch de mensajes no necesariamente tiene que pasarlo a un único servidor. Quizas en el batch de mensajes hay mensajes para multiples Endpoints que son mapeados por el LB en diferentes servidores.

El LB levanta un thread por cada CLT node y un thread por cada SVR node.

* Proxies hacia SVRs
* Proxies hacia CLTs
* Failure Detector: En la versión anterior se utilizó spread como mecanismo de detección de fallos y comunicaciones multicast. En esta versión tanto el LB como el multiproxy funcionan como Failure Detectors y utilizan el mismo canal de comunicación de M3-IPC para comunicarse entre los proxies (a través del cmd NONE) Además funcionan como agentes para realizar diferentes tareas tales como:
  + Ejecutar un comando en el nodo local
  + Detener un proceso
  + Hacer el rmtbind de un Endpoint
  + Obtener información del estado de un proceso o Endpoint (getprocsts)
  + Reporta el estado del NODO (%CPU por ejemplo)
* Load Balancer: Periódicamente, el LB verifica el estado de cada proxy para saber si está conectado o no. Si un SVR node está activo, entonces le puede derivarle sesiones. Además controla la carga de los diferentes nodos servidores recibiendo información perioódica del estado de estos. Si todos los nodos servidores están sobrecargados y la configuración lo permite, arranca nuevos nodos. Si existen nodos servidores ociosos, puede terminar las VMs hasta un mínimo configurado.

Cuando un CLT inicia una sesión contra el LB:

1. Se consulta si el CLT Endpoint ya está siendo usado. Si lo está, verifica que el SVR node esté activo. Si lo está no hace nada ya que la sesión está creada.
2. Si no hay sesión creada se consulta la DB para ver cual SVR node está más descargado y busca cuales son los Endpoint clients que están libres.

Problemas:

* Como distinguir entre un inicio de una nueva sesión (previamente utilizada por otro proceso del mismo node) y una continuación de la misma sesión?
  + Solución: El agente del multi\_proxy informe acerca del PID de cada proceso que emite un comando.

El agente del proxy en los CLT, verifica si un Endpoint de un DC dado cambio de PID.

Se modificó el header de los proxies incluyendo el campo c\_pid para indicar el PID del proceso correspondiente al Endpoint.

También se modifico el header para que en el mismo comando puedan ir mensajes M3-IPC conjutamente con datos para una futura implementación de una librería de RPC en la cual el cliente o el servidor puedan incluir ambas cosas en el mismo mensaje.

COMUNICACIÓN ENTRE PROXIES DEL LB

Cada Proxy Sender tiene asociada una message queue (mqueue) cuyo tipo de registro permite almacenar el *header.*  El payload alloc() lo hace el Proxy Client, y en el header pone la dirección del payload. El Proxy Sender, toma el *payload* de ese buffer y una vez enviado el nodo destino hace free(payload\_buffer). De esta manera, proxy client y proxy server pueden trabajar asincrónicamente.

La DB del LB se protege con un semáforo lb\_mtx.

Comunicación en sentido CLT->SVR (primer opción con message queue)

* Cuando el Proxy Receiver de un CLTPXY recibe un mensaje, hace un down(lb\_mtx) y lee la DB para ver si existe o no la sesión y eventualmente modificar los campos del *header*. Luego hace up(lb\_mtx)
* Inserta el *header* en la mqueue del SVRPXY correspondiente al nodo del SVR.
* El header contiene la dirección del buffer del payload

Comunicación en sentido SVR->CLT

* Cuando el Proxy Receiver de un SVRPXY recibe un mensaje, hace un down(lb\_mtx) y lee la DB y modifica los campos del *header*.
* Inserta el *header* en la mqueue del CLTPXY correspondiente al nodo del CLT.
* El header contiene la dirección del buffer del payload

BASE DE DATOS

Los campos de la DB de sesiones son son:

* se\_dcid: DC ID de la sesion
* se\_clt\_nodeid: ID del nodo del Client
* se\_clt\_ep: Endpoint del Client
* se\_clt\_PID: PID del Client
* se\_lbclt\_ep: Endpoint presentado en el LB hacia lado Client
* se\_lbsvr\_ep: Endpoint presentado en el LB hacia lado Server
* se\_svr\_nodeid: ID del server
* se\_svr\_ep: Endpoint del Server
* se\_svr\_PID: PID del Server
* \*se\_rmtcmd: puntero a un string del comando remoto a ejecutar
* \*se\_service: puntero al servicio ejecutado en el server (del archivo de configuración)

En un flujo CLT->SVR, el LB busca una entrada en la DB basándose en:

* { se\_clt\_nodeid, se\_clt\_ep, se\_lbclt\_ep, se\_clt\_PID }

Si encuentra una sesión activa entonces:

* Si el campo del header, (c\_pid == ss\_clt\_PID) es igual entonces quiere decir que es una misma sesión.
* Si (c\_pids != ss\_clt\_PID) quiere decir que es una sesión nueva por lo que suprime la sesión anterior y crea una nueva.
* Envia el paquete modificado hacia el SVR modificando los campos del header recibido {c\_snode=lb\_nodeid, c\_dnode=lb\_svr\_nodeid, c\_src=lb\_lbsvr\_ep, c\_pid=lb\_pid }
* Cuando recibe desde el SVR el header, modifica {c\_snode=lb\_nodeid, c\_dnode=lb\_clt\_nodeid, c\_src=ss\_lbclt\_ep, c\_pid=lb\_pid}

Si no encuentra una sesión activa:

1. Busca el server menos cargado
2. Busca para ese server un Endpoint CLT libre en la DB del Server.
3. Crea una nueva sesion

El LB tiene otra DB de los Endpoints clients del LB utilizados para comunicarse con cada SVR utilizando un bitmap.

NODE FAILURE DETECTOR

Se deben distinguir entre distintos tipos de fallos benignos:

1. ***Fallos de Conexión (Network Partition):*** El LB o alguno o todos los nodos del cluster perdieron conectividad o se produjo una partición de red.
2. ***Fallos de Nodo***: son indistinguibles respecto a los fallos de Conexión.
3. ***Fallos del Agente/Proxy***: Existe conectividad entre el LB y el NODO pero el agente/proxy del nodo servidor no tiene una sesión activa con su correspondiente par de proxies en el LB.
4. ***Fallos de un proceso:*** Algún proceso del Nodo Servidor no responde porque, o no se arrancó o finalizó (normalmente o con error) o no realizó el bind en el nodo servidor.

Para realizar la detección de fallos de tipo 1 y 2 se utiliza un thread monitor en el LB que basa sus decisiones en el protocolo ICMP.

Debería quedar claro que ante un fallo de tipo 1-2 automáticamente se producen fallos de tipo 3 y 4.

Los nodos activos se encuentran mapeados en el bitmap lb\_bm\_active.

El monitor realiza un ciclo infinito, que a su vez realiza otro ciclo interno de lb\_period en segundos.

Los ciclos de testeo de cada Nodo se definen en función del ID del nodo. Supongamos que se tienen 3 nodos configurados (que se encuentran marcados en el bitmap lb\_bm\_svrpxy), los mismos (NODE1, NODE3, NODE7) darían como resultado el siguiente bitmap 0000 0000 1000 1010. Si el período establecido lb\_period es MAYOR o IGUAL a la cantidad de nodos configurados, entonces por cada ciclo se chequeará un nodo o ninguno, como muestra la siguiente tabla.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| NODE1 |  | X |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| NODE3 |  |  |  | X |  |  |  |  |  |  |  |  |  |  |  |  |
| NODE7 |  |  |  |  |  |  |  | X |  |  |  |  |  |  |  |  |

En cambio, si la cantidad de nodos configurados es MAYOR que el período lb\_period, entonces en cada ciclo se podrán testear 1 o mas nodos de acuerdo a su ID (se testean los nodos donde resto de la división NODEID/lb\_period = Nro\_Ciclo). En este caso, donde se configuraron 13 nodos y el período es 8, entonces, en el ciclo 0 se testean al NODE0 y NODE8, dado que mod(0/8)=0 y mod(8/8)=0

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| NODE0 | X |  |  |  |  |  |  |  |
| NODE1 |  | X |  |  |  |  |  |  |
| NODE2 |  |  | X |  |  |  |  |  |
| NODE3 |  |  |  | X |  |  |  |  |
| NODE4 |  |  |  |  | X |  |  |  |
| NODE5 |  |  |  |  |  | X |  |  |
| NODE6 |  |  |  |  |  |  | X |  |
| NODE7 |  |  |  |  |  |  |  | X |
| NODE8 | X |  |  |  |  |  |  |  |
| NODE9 |  | X |  |  |  |  |  |  |
| NODE10 |  |  | X |  |  |  |  |  |
| NODE11 |  |  |  | X |  |  |  |  |
| NODE12 |  |  |  |  | X |  |  |  |

Cuando corresponde testear a un nodo se envía un paquete ICMP ECHO REQUEST y se marca en el bitmap lb\_bm\_echo que para ese nodo debe esperarse una respuesta ICMP ECHO REPLY.

Luego se espera 1 segundo para realizar las eventuales respuestas de los nodos.

Si el ciclo está vacio porque no se envio a ningún nodo el ECHO REQUEST, no se hace nada y pasa al siguiente ciclo.

Por cada bit encendido en el bitmap lb\_bm\_echo se realiza la recepción del ECHO REPLY.

Si el nodo testeado forma parte de los nodos actualmente activos, si hay algún tipo de error se va llevando la cuenta de cuantos errores se van detectando en el contador svr\_icmp\_retry. En el primer error que se detecta, el nodo pasa a estado sospechoso en el bitmap lb\_bm\_suspect y se decrementa el contador y se espera hasta el próximo ciclo (1 segundo) pero sin enviarle un nuevo ECHO REQUEST, manteniéndose el bit correspondiente en el bitmap lb\_bm\_echo. Cuando el contador llega a cero, se elimina al nodo del bitmap lb\_bm\_active. A partir de este momento, el nodo será testeado en el período establecido lb\_period.

Si después de uno o varios errores, un marcado como activo responde correctamente, entonces se lo quita del bitmap de sospechosos lb\_bm\_suspect y se vuelve el contador de reintentos a FD\_MAXRETRIES.

Cada vez que un nodo responde correctamente se incrementa el contador svr\_icmp\_rcvd de tal forma que el próximo ECHO REQUEST se cree con un numero de secuencia (svr\_icmp\_send = svr\_icmp\_rcvd+1)

Cada vez que se sale por error, no se incrementa el contador svr\_icmp\_rcvd.

La estructura del paquete ICMP se puede consultar de:

<https://raw.githubusercontent.com/afabbro/netinet/master/ip_icmp.h>

Para distinguir un nodo que falló de un nodo que se desconectó, cuando el nodo levanta el proxy debe enviar un mensaje tipo MT\_JOIN al LB con número de secuencia 1. Si no se recibe este mensaje y el número de secuencia es mayor que 1 es porque el nodo estuvo desconectado de la red del LB.

Cuando el proxy del LB detecta una caída de sesión, pone como sospechoso al nodo y lo remueve de los inicializados lb\_bm\_init y lb\_bm\_proxy.

Cuando el proxy del LB detecta que se levanta una sesión, pone como activo al nodo (lb\_bm\_active) y cuando recibe un MT\_JOIN lo incluye en los inicializados lb\_bm\_init. Si no recibe un JOIN y el número de secuencia es mayor que 1 entonces quiere decir que era un nodo desconectado por lo que también lo incluye en lb\_bm\_init pero era un NETWORK PARTITION.

**bachmanm\_failure-detectors\_ Agreement in Asynchronous Distributed Systems**

LOAD MONITORING

El LB es encargado de recolectar información sobre el estado de carga (solo CPU por ahora) de cada uno de los nodos SVRs.

Lleva un registro de:

* Nodos Activos
* Endpoint SVR Activos (utilizando un bitmap)
* Carga de cada nodo {CPU}

Para monitorear el estado de carga de cada nodo, se ejecuta en cada multi\_proxy de nodo SVR un agente que evalúa carga de cada nodo {CPU} e informa al LB mediante un mensaje de tipo MT\_LOAD\_LEVEL.

El LB y los multi\_proxy se transfieren info de carga periódicamente.

MT\_LOAD\_THRESHOLDS : Donde el LB le indica al nodo servidor cuales son los limites de carga y el período de medición.

MT\_LOAD\_LEVEL: Donde el agente del multi\_proxy del nodo server le informa al LB el estado de carga del nodo.

Por cada JOIN, el *lb\_monitor*:

* Verifica el estado del proxy correspondiente para ver si lo tiene conectado.
* Si está conectado lo incluye en la lista de nodos activos y le envía un mensaje LB\_ACTIVE, para que el *lb\_agent* sepa que ahora puede operar.

*El lb\_agent* arranca con un estado STS\_NONE y hace el JOIN del nodo al grupo del LB.

* Cambia su estado a STS\_JOINT y queda a la espera de un mensaje LB\_ACTIVE. Hasta que no recibe un mensaje este mensaje permanece en este estado.
* Cuando recibe el mensaje LB\_ACTIVE, cambia su estado a STS\_ACTIVE.
* Si se recibe un mensaje DISCONNECT o NETWORK que indica que el LB se cayo o su nodo, cambia al estado STS\_NONE nuevamente, hasta recibir un JOIN del LB.

Por cada DISCONNECT de un SVR, el LB:

* Busca en la DB todos las entradas que tienen al nodo en el campo *ss\_svr\_nodeid* y por cada una de ellas,
  + Envía al proxy del nodo CLT correspondiente una orden de CMD\_PROCUNBIND. Esto hace que el cliente reciba un mensaje de EDVSNOTBIND.
  + Elimina al nodo de la lista de nodos activos
  + Borra la entrada del nodo en la DB

Por cada mensaje LB\_SVRSTSINFO enviado por el LB, el lb\_agent responde a mensajes de tipo {dcid, endpoint} con el estado del proceso (proc\_usr\_t) utilizando

long dvk\_getprocinfo(int dcid, int p\_nr, proc\_usr\_t \*proc\_usr\_ptr)

Periodicamente (LBA\_PERIOD) el *lb\_agent*, verifica la carga {CPU} del nodo. Para ello se establecen:

* dos limites (thresholds LBA\_LOWWATER y LBA\_HIGHWATER)
* 4 niveles (LVL\_NOTINIT no iniciado, LVL\_IDLE, LVL\_LOADED, LVL\_SATURATED)

Int node\_loadlvl = LVL\_NOTINIT; // nivel anterior de carga del nodo local

Int cur\_loadlvl = LVL\_NOTINIT; // nivel de carga actual del nodo local

Int cur\_load = 0; // valor de carga del nodo local

Periódicamente() {

cur\_load = get\_cpu\_usage();

If ( cur\_load < LBA\_LOWWATER) {

cur\_loadlvl = LVL\_IDLE;

}else if (If ( cur\_load > LBA\_HIGHWATER)

cur\_loadlvl = LVL\_SATURATED;

}else {

cur\_loadlvl = LVL\_LOADED;

}

// si se cambio de nivel, hace un unicast al LB informandolo

If( node\_loadlvl != cur\_loadlvl) {

Node\_loadlvl = cur\_loadlvl;

unicast(lb\_node, node\_loadlvl);

}

}

**NOTA1**: Los procesos SVR podrían no existir en principio, de tal forma que cuando un CLT inicia una sesión, se evalúa la carga del nodo y en forma remota se ejecuta un nuevo proceso server (mediante el agente LB via spread LB\_SVREXEC) en un Endpoint server libre.

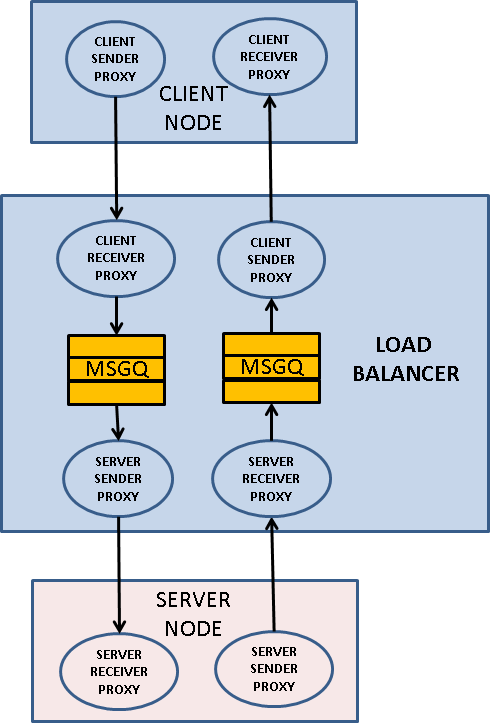
**NOTA2**: El lb\_agent por cada proceso SVR creado (fork, exec) crearía un thread que haga un *dvk\_wait4unbind()* esperando que el SVR finalice. Una vez finalizado le hace un unicast al LB de tipo LB\_SVREXIT. De esta forma el LB puede remover de su DB la sesión establecida.

ETAPAS:

* Crear el grupo SPREAD (“DVS\_LB”) para que el LB reciba la carga de los diferentes nodos. Cada nodo debe distinguirse si es LB o es SVR.
* Probar la ejecución de un nuevo server
* Si una sesión se cae, se puede matar al server via el agente LB.

***PROGRAMAS EN C***

* Dvs\_lb.c: Aquí estará el main el que creará los diferentes threads que lo componen.
* Glo.h: Variables Globales
* Lb\_cfg: lee la configuración del archivo xxxx.cfg mencionado como argumento de main
* Lb\_cltpxy: Threads de los proxies que miran hacia los clients
* Lb\_svrpxy: Threads de los proxies que miran hacia los servers
* Lb\_queue: Funciones que leen de una cola y pone en header y escriben en una cola {header}. Mientras tanto el payload va en un buffer dinamico. Cuando el Proxy Receiver encola el header con la dirección del buffer, se desprende del buffer y adquiere con malloc o posix\_malign un nuevo buffer para el payload. Cuando el Proxy Sender envió el mensaje con su payload, liberar (free) el payload.
* Lb\_monitor: Monitor de Estado y Carga de los Servers
* Dvs\_lba.c: load balancing agent



## Archivo de Configuración

Cuando arranca el LB el único argumento es el nombre del archivo de configuración. En este archivo se describen el Load Balancer (lb), los Servers (server), los servicios (service) y los Clients (client). Por ahora, tanto el lb, server y client, solo tienen como parámetro de configuración al nodeid. En un futuro, quizás no sea necesario crear los servers y que éstos se creen en forma automática al hacer el JOIN al grupo “DVS\_LB”.

Los servicios requieren una mayor cantidad de parámetros tales como el Endpoint que presentarán a los Clients (ext\_ep), el rango de Endpoints que pueden utilizar los servidores para brindar esos servicios (min\_ep, max\_ep), el tipo de registración que se hará en los DVKs de los Servers (bind) al arrancar los programas servidores (prog) y el DC al que pertenece ese servicio (dcid)

# this is a comment

lb LB\_NAME {

nodeid 0;

};

server node1 {

nodeid 1;

};

#dcid puede ser un numero 0-(NR\_DCS-1)

#si no se menciona dcid, entonces es ANY

#ext\_ep es el endpoint presentado a los clientes.

#min\_ep y max\_ep es el rango de endpoints en los que se pueden arrancar los services

# el rango máximo es {0, (NR\_SYS\_PROCS-NRTASKS)} actualmente limitado a 32 (0-31)

service m3ftp {

dcid 0;

ext\_ep 10;

min\_ep 10;

max\_ep 20;

bind replica;

prog "/usr/src/dvs/dvs-apps/dvs\_run/ftpd.sh"

};

service m3web {

ext\_ep 21;

min\_ep 21;

max\_ep 30;

bind replica;

prog "/usr/src/dvs/dvs-apps/dvs\_run/httpd.sh"

};

client node2 {

nodeid 2;

};

client node3 {

nodeid 3;

};

client node4 {

nodeid 4;

};

# Estructuras de Datos

## Server

Cada Server tiene un nombre (svr\_name) el cual debe estar dado de alta en el /etc/hosts del LB o bien en un DNS. Se especifica también el número de nodo (svr\_nodeid) y luego tiene una estructura de datos para su Sender proxy (svr\_spx) y otra para su Receiver Proxy (svr\_rpx).

Además posee un semáforo mutex (svr\_mutex) para proteger los campos compartidos con otros threads.

En cada server se podrá ejecutar servicios diferentes y multiples instancias de un mismo servicio, pero cada una de ellas registrada en un Endpoint diferente. El registro de los Endpoints utilizados por los servicios en cada servidor se mantiene con un bitmap (svr\_bm\_svc).

El monitor de carga del LB mantiene un registro porcentual entero de la carga (svr\_load) de un server que le es informada por el agente del server y un nivel de carga (svr\_level) que puede ser LVL\_NOTINIT, LVL\_IDLE, LVL\_LOADED, LVL\_SATURATED.

typedef struct {

char \*svr\_name; // server name from configuration file

int svr\_nodeid; // server nodeid

int svr\_level; // Load LEVEL

int svr\_load; // CPU Load (0-100) . Value (-1) implies INVALID

unsigned long int svr\_bm\_svc; // bitmap of service endpoint used

lbpx\_desc\_t svr\_spx; // Server sender proxy

lbpx\_desc\_t svr\_rpx; // Server receiver proxy

pthread\_mutex\_t svr\_mutex; // protect on change of status and load level.

}server\_t;

## Client

Cada Client tiene un nombre (clt\_name) el cual debe estar dado de alta en el /etc/hosts del LB o bien en un DNS. Se especifica también el número de nodo (clt\_nodeid) y luego tiene una estructura de datos para su Sender proxy (clt\_spx) y otra para su Receiver Proxy (clt\_rpx).

typedef struct {

char \*clt\_name;

int clt\_nodeid;

lbpx\_desc\_t clt\_spx;

lbpx\_desc\_t clt\_rpx;

} client\_t;

## Services

Describe las características de un servicio cuyo nombre se especifica en el archivo de configuración (svc\_name).

El LB presenta en su proxy contra cada Client un Endpoint (svc\_extep) que es visto como Servidor para cada DC (svc\_dcid). Este Endpoint, no necesariamente será el que utilizará el servicio al ser arrancado en un nodo servidor. Los servicios pueden utilizar un Endpoint dentro de un rango {svc\_minep, svc\_maxep}. Además, por cada servicio se especifica el tipo de bind (replica, local, backup) y el path (en el nodo destino) del proceso que debe ejecutarse al iniciarse una sesión.

typedef struct {

char \*svc\_name; // server name from configuration file

int svc\_dcid; // server dcid

int svc\_extep; // server external endpoint

int svc\_minep; // server lower endpoint when execute the server

int svc\_maxep; // server higher endpoint when execute the server

int svc\_bind; // Which kind of bind must be done on server creation

char \*svc\_prog; // Program to execute on server

}service\_t;

## Load Balancer Proxy

Cada Client y cada Server tienen asociado un par de proxies (Sender y Receiver).

Por cada proxy se crea un thread.

El proxy Receiver de un Server recibe los mensajes de su Server correspondiente, los modifica y los reenvia al Client de la Sesion Activa a través de su Client Sender utilizando una cola de mensajes. De igual forma, el proxy Receiver de un Client recibe los mensajes de su Client correspondiente, los modifica y los reenvia al Server de la Sesión Activa o de una nueva Sesión a través de su Client Sender utilizando una cola de mensajes.

La cola de mensaje de la estructura de datos LB Proxy Descritor es solo utilizada por los Proxy Senders.

Cada Proxy tiene asociado un puntero a un buffer para el header (lbp\_header) y otro para el payload (lbp\_payload) de los mensajes recibidos (Receiver) o enviados (Sender) a través de los sockets TCP.

También contiene información de la sesiones TCP (sockets, direcciones IP, ports, etc) y contadores de mensajes Enviados/Recibidos en forma Correcta/Errónea.

// Load Balancer Proxy Descriptor

struct lbpx\_desc\_s {

pthread\_t lbp\_thread;

proxy\_hdr\_t \*lbp\_header;

proxy\_payload\_t \*lbp\_payload; /\* uncompressed payload \*/

pthread\_mutex\_t lbp\_mtx; //mutex & condition to allow main thread to

//wait for the new thread to set its TID

pthread\_cond\_t lbp\_cond;

pid\_t lbp\_tid; /\* to hold new thread's TID \*/

msgq\_buf\_t \*lbp\_mqbuf; // Server proxy sender message queue

int lbp\_mqid;

int lbp\_mqkey;

struct msqid\_ds lbp\_mqds;

int lbp\_sd; // Sender TCP socket descriptor

int lbp\_lsd; // Receiver TCP LISTEN socket descriptor

int lbp\_csd; // Sender TCP CONNECTION socket descriptor

int lbp\_port;

struct sockaddr\_in lpb\_rmtclt\_addr;

struct sockaddr\_in lpb\_rmtsvr\_addr;

#define lpb\_lclsvr\_addr lpb\_rmtsvr\_addr

struct hostent \*lbp\_rmthost;

long lbp\_msg\_ok;

long lbp\_msg\_err;

};

typedef struct lbpx\_desc\_s lbpx\_desc\_t;

## Session

Cuando un cliente inicia una comunicación contra el Client Proxy del LB se verifica si es Client tiene una sesión iniciada con un servidor designado. Para ello se comparan un conjunto de parámetros tales como el DC, sender Endpoint, destination Endpoint, etc. Además se compara el PID del Client, porque si este difiere solo en este parámetro de una sesión existente en la BD del LB, se elimina la sesión existente y se crea una nueva. Para ello se busca en el LB si hay una entrada libre en la entrada de Sesiones. Si no encuentra, retorna al cliente un mensaje EDVSNOSPC.

Si no existe una sesión, se debe asignar un Server. Esto se realiza en base a la información que el LB dispone de la carga de cada Server obtenida a partir del Monitor. Una vez seleccionado el Server, se busca en el bitmap del Server svr\_bm\_svc, un bit en 0, que significa un Endpoint libre. Los Endpoints pueden tener un valor de 0 a 32 (solo para simplificar la implementación del bitmap). Si no encuentra Endpoint libre, seguirá buscando por el server siguiente de menor carga hasta terminar la lista de Endpoints y Servers. Si no encuentra ningún Endpoint libre, entonces retorna al Client un mensaje con error EDVSNOTBIND.

Una vez asignada la Sesion, se completan sus campos en base a la información del header (incluido el subheader de vcopy) recibido desde el Client y de la información del Server obtenida de la DB del Monitor (svr\_nodeid, svr\_endpoint). Con la sesión nueva o con la previamente existente, se reenvia el mensaje recibido desde el Client (ahora con su header modificado) al Server Sender a través de su cola de mensajes.

typedef struct {

int se\_dcid;

int se\_clt\_nodeid;

int se\_clt\_ep;

int se\_clt\_PID;

int se\_lbclt\_ep;

int se\_lbsvr\_ep;

int se\_svr\_nodeid;

int se\_svr\_ep;

int se\_svr\_PID;

} sess\_entry\_t;

typedef struct {

int st\_nr\_sess; // # of active sessions

pthread\_mutex\_t st\_mutex; // per Table Mutex

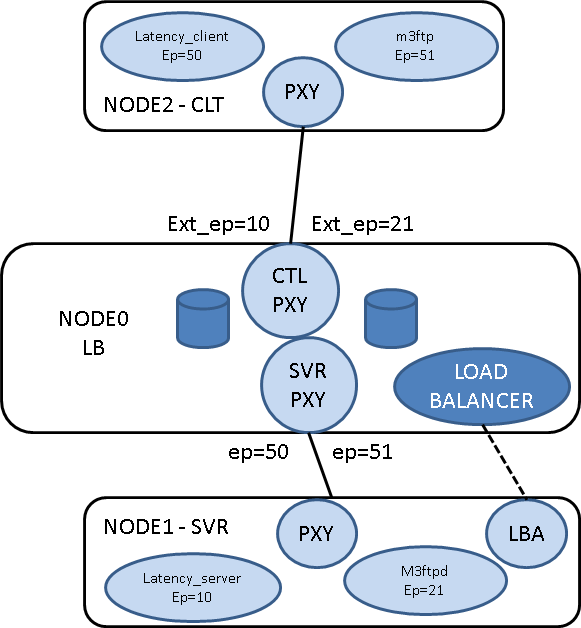
sess\_entry\_t \*st\_tab\_ptr; // pointer to the first table element

} sess\_tab\_t;

Cuando el Server Receiver Proxy recibe un mensaje de un Server, verifica si hay una sesión existente. Si no existe sesion, y el mensaje NO ES un ACKNOWLEGE le devuelve al Server un mensaje con error EDVSCONNREFUSED.

Si existe la sesión, pero solo difiere en el PID del Server, esto indica que es otra instancia del Server y que hay que eliminar la sesión existente.

# PRUEBAS



ARCHIVO DE CONFIGURACION EN NODE0

# this is a comment

# this is a comment

lb node0 {

nodeid 0;

lowwater 30;

highwater 70;

period 30;

};

server node1 {

nodeid 1;

};

#dcid puede ser un numero 0-(NR\_DCS-1)

#si no se menciona dcid, entonces es ANY

#ext\_ep es el endpoint presentado a los clientes.

#min\_ep y max\_ep es el rango de endpoints en los que se pueden arrancar los services

service latency {

dcid 0;

**ext\_ep 10**;

min\_ep 10;

max\_ep 19;

};

service m3ftp {

dcid 0;

**ext\_ep 21**;

min\_ep 20;

max\_ep 29;

};

client node2 {

nodeid 2;

};

ATENCION, UTILIZAR PROXIES LZ4 TCP BAT

* SIN AUTO BIND
* SIN COMPRESION
* SIN BATCH
* CON PID AGREGADO

/usr/src/dvs/dvk-proxies/lz4tcp\_proxy\_bat **-P** -n node$rmtA -i $rmtA > /dev/shm/node$rmtA.txt 2> /dev/shm/error$rmtA.txt &

/usr/src/dvs/dvk-proxies/lz4tcp\_proxy\_bat **-P** -n node$rmtB -i $rmtB > /dev/shm/node$rmtB.txt 2> /dev/shm/error$rmtB.txt &

PARA LAS PRUEBA REMOVER:

* DEBUG DEL DVK
* DEBUG DE LA LIB
* DEBUG DE LOS PROXIES
* DEBUG DEL LB
* DEBUG DE LATENCY\_CLIENT Y LATENCY\_SERVER
* DEBUG DE FTPD Y FTP

Crear un archivo de 1 MB, otro de 10 MB y otro de 100M

dd bs=1024 count=1024 </dev/urandom > file-1M.dat

dd bs=1024 count=10240 </dev/urandom > file-10M.dat

dd bs=1024 count=102400 </dev/urandom > file-100M.dat

***TEST LOCALES***

|  |  |
| --- | --- |
| **SEQ** | **NODE1** |
| 1 | cd /usr/src/dvs/dvk-tests/ |
| 2 | ./tests.sh 1 0 |
|  | . /dev/shm/DC0.sh |
| 3 | ./latency\_server 0 10 > latency0.out 2> latency0.err &  ./latency\_server 0 11 > latency1.out 2> latency1.err &  ./latency\_server 0 12 > latency2.out 2> latency2.err & |
| 4 | cd /usr/src/dvs/dvs-apps/m3ftp  ./m3ftpd 0 21 > m3ftpd0.out 2> m3ftpd0.err &  cd /usr/src/dvs/dvs-apps/m3ftp  ./m3ftpd 0 22 > m3ftpd1.out 2> m3ftpd1.err &  cd /usr/src/dvs/dvs-apps/m3ftp  ./m3ftpd 0 23 > m3ftpd2.out 2> m3ftpd2.err &  cat /proc/dvs/DC0/procs |
| 5 | cd /usr/src/dvs/dvk-tests/ |
| 6 | ./latency\_client 0 50 10 1000 > lat0.out &  ./latency\_client 0 51 10 1000 > lat1.out &  ./latency\_client 0 52 10 1000 > lat2.out & |
| 7 | cd /usr/src/dvs/dvs-apps/m3ftp/  ./m3ftp -g 0 60 21 file-1M.dat file0-1M.dat > gftp0.out &  ./m3ftp -g 0 61 22 file-1M.dat file1-1M.dat > gftp1.out &  ./m3ftp -g 0 62 23 file-1M.dat file2-1M.dat > gftp2.out & |
|  | ./m3ftp -p 0 60 21 file0a-1M.dat file0-1M.dat > pftp0.out &  ./m3ftp -p 0 61 22 file1a-1M.dat file1-1M.dat > pftp1.out &  ./m3ftp -p 0 62 23 file2a-1M.dat file2-1M.dat > pftp2.out & |
|  | ./m3ftp -g 0 60 21 file-10M.dat file0-10M.dat > gftp0.out &  ./m3ftp -g 0 61 22 file-10M.dat file1-10M.dat > gftp1.out &  ./m3ftp -g 0 62 23 file-10M.dat file2-10M.dat > gftp2.out & |
|  | ./m3ftp -p 0 60 21 file0a-10M.dat file0-10M.dat > pftp0.out &  ./m3ftp -p 0 61 22 file1a-10M.dat file1-10M.dat > pftp1.out &  ./m3ftp -p 0 62 23 file2a-10M.dat file2-10M.dat > pftp2.out & |
|  | ./m3ftp -g 0 60 21 file-100M.dat file0-100M.dat > gftp0.out & |
|  | ./m3ftp -p 0 60 21 file0a-100M.dat file0-100M.dat > pftp0.out & |

***TEST REMOTOS***

|  |  |  |
| --- | --- | --- |
| **SEQ** | **NODE1(SERVER)** | **NODE2(CLIENT)** |
|  | cd /usr/src/dvs/dvk-tests/ | cd /usr/src/dvs/dvk-tests/ |
|  | ./tests.sh 1 0 | ./tests.sh 2 0 |
|  | **ESPERAR QUE TERMINE** | **ESPERAR QUE TERMINE** |
|  | netstat –nat  . /dev/shm/DC0.sh | netstat –nat  . /dev/shm/DC0.sh |
|  | ./test\_rmtbind 0 **50** 2 latclient0  ./test\_rmtbind 0 **51** 2 latclient1  ./test\_rmtbind 0 **52** 2 latclient2  ./test\_rmtbind 0 **60** 2 m3ftp0  ./test\_rmtbind 0 **61** 2 m3ftp1  ./test\_rmtbind 0 **62** 2 m3ftp2  cat /proc/dvs/DC0/procs | ./test\_rmtbind 0 **10** 1 latency\_server  ./test\_rmtbind 0 **11** 1 latency\_server  ./test\_rmtbind 0 **12** 1 latency\_server  ./test\_rmtbind 0 **21** 1 m3ftpd  ./test\_rmtbind 0 **22** 1 m3ftpd  ./test\_rmtbind 0 **23** 1 m3ftpd  cat /proc/dvs/DC0/procs |
|  | ./latency\_server 0 10 > latency0.out 2> latency0.err &  ./latency\_server 0 11 > latency1.out 2> latency1.err & ./latency\_server 0 12 > latency2.out 2> latency2.err & |  |
|  | cd /usr/src/dvs/dvs-apps/m3ftp  ./m3ftpd 0 21 > m3ftpd0.out 2> m3ftpd0.err &  cd /usr/src/dvs/dvs-apps/m3ftp  ./m3ftpd 0 22 > m3ftpd1.out 2> m3ftpd1.err &  cd /usr/src/dvs/dvs-apps/m3ftp  ./m3ftpd 0 23 > m3ftpd2.out 2> m3ftpd2.err &  cat /proc/dvs/DC0/procs |  |
|  |  | ./latency\_client 0 50 10 100 > lat0.out &  ./latency\_client 0 51 10 100 > lat1.out &  ./latency\_client 0 52 10 100 > lat2.out & |
|  |  | cd /usr/src/dvs/dvs-apps/m3ftp/  ./m3ftp -g 0 60 21 file-1M.dat file0-1M.dat > gftp0.out &  ./m3ftp -g 0 61 22 file-1M.dat file1-1M.dat > gftp1.out &  ./m3ftp -g 0 62 23 file-1M.dat file2-1M.dat > gftp2.out & |
|  |  | ./m3ftp -p 0 60 21 file0-1M.dat file0-1M.dat > pftp0.out &  ./m3ftp -p 0 61 22 file1-1M.dat file1-1M.dat > pftp1.out &  ./m3ftp -p 0 62 23 file2-1M.dat file2-1M.dat > pftp2.out & |
|  |  | ./m3ftp -g 0 60 21 file-10M.dat file0-10M.dat > gftp0.out &  ./m3ftp -g 0 61 22 file-10M.dat file1-10M.dat > gftp1.out &  ./m3ftp -g 0 62 23 file-10M.dat file2-10M.dat > gftp2.out & |
|  |  | ./m3ftp -p 0 60 21 file0-10M.dat file0-10M.dat > pftp0.out &  ./m3ftp -p 0 61 22 file1-10M.dat file1-10M.dat > pftp1.out &  ./m3ftp -p 0 62 23 file2-10M.dat file2-10M.dat > pftp2.out & |
|  |  | ./m3ftp -g 0 60 21 file-100M.dat file0-100M.dat > gftp0.out & |
|  |  | ./m3ftp -p 0 60 21 file0-100M.dat file0-100M.dat > pftp0.out & |

***TEST REMOTOS CON LOAD BALANCER SINGLE SERVER***

|  |  |  |  |
| --- | --- | --- | --- |
| **SEQ** | **NODE0 (LB)** | **NODE1(SERVER)** | **NODE2(CLIENT)** |
|  | mkdir /var/run/spread  /usr/local/sbin/spread -c /etc/spread.conf > /dev/shm/spread.txt & |  |  |
| 1 | cd /usr/src/dvs/dvs-apps/dvs\_lb/ |  |  |
| 2 | ./lb\_dvs lb\_dvs.cfg > lb\_dvs.out 2> lb\_dvs.err & |  |  |
| 3 |  | cd /usr/src/dvs/dvk-tests/ | cd /usr/src/dvs/dvk-tests/ |
| 4 |  | ./tests.sh 1 0 | ./tests.sh 2 0 |
|  |  | **ESPERAR QUE TERMINE** | **ESPERAR QUE TERMINE** |
| 5 |  | netstat –nat  . /dev/shm/DC0.sh | netstat –nat  . /dev/shm/DC0.sh |
| 6 |  | ./test\_rmtbind 0 **50** 0 latclient0  ./test\_rmtbind 0 **51** 0 latclient1  ./test\_rmtbind 0 **52** 0 latclient2  ./test\_rmtbind 0 **60** 0 m3ftp0  ./test\_rmtbind 0 **61** 0 m3ftp1  ./test\_rmtbind 0 **62** 0 m3ftp2  cat /proc/dvs/DC0/procs | ./test\_rmtbind 0 **10** 0 latency\_server  ./test\_rmtbind 0 **11** 0 latency\_server  ./test\_rmtbind 0 **12** 0 latency\_server  ./test\_rmtbind 0 **20** 0 m3ftpd  ./test\_rmtbind 0 **21** 0 m3ftpd  ./test\_rmtbind 0 **22** 0 m3ftpd  cat /proc/dvs/DC0/procs |
| 7 |  | ./latency\_server 0 **10** > latency0.out 2> latency0.err &  ./latency\_server 0 **11** > latency1.out 2> latency1.err & ./latency\_server 0 **12** > latency2.out 2> latency2.err & |  |
| 8 |  | cd /usr/src/dvs/dvs-apps/m3ftp  ./m3ftpd 0 **20** > m3ftpd0.out 2> m3ftpd0.err &  cd /usr/src/dvs/dvs-apps/m3ftp  ./m3ftpd 0 **21** > m3ftpd1.out 2> m3ftpd1.err &  cd /usr/src/dvs/dvs-apps/m3ftp  ./m3ftpd 0 **22** > m3ftpd2.out 2> m3ftpd2.err &  cat /proc/dvs/DC0/procs |  |
| 9 |  |  | ./latency\_client 0 **50 10** 100 > lat0.out &  ./latency\_client 0 **51 11** 100 > lat1.out &  ./latency\_client 0 **52 12** 100 > lat2.out & |
|  |  |  | cd /usr/src/dvs/dvs-apps/m3ftp/  ./m3ftp -g 0 **60 20** file-1M.dat file0-1M.dat > gftp0.out &  ./m3ftp -g 0 **61 21** file-1M.dat file1-1M.dat > gftp1.out &  ./m3ftp -g 0 **62 22** file-1M.dat file2-1M.dat > gftp2.out & |
|  |  |  | ./m3ftp -p 0 **60 20** file0-1M.dat file0-1M.dat > pftp0.out &  ./m3ftp -p 0 **61 21** file1-1M.dat file1-1M.dat > pftp1.out &  ./m3ftp -p 0 **62 22** file2-1M.dat file2-1M.dat > pftp2.out & |
|  |  |  | ./m3ftp -g 0 **60 20** file-10M.dat file0-10M.dat > gftp0.out &  ./m3ftp -g 0 **61 21** file-10M.dat file1-10M.dat > gftp1.out &  ./m3ftp -g 0 **62 22** file-10M.dat file2-10M.dat > gftp2.out & |
|  |  |  | ./m3ftp -p 0 **60 20** file0-10M.dat file0-10M.dat > pftp0.out &  ./m3ftp -p 0 **61 21** file1-10M.dat file1-10M.dat > pftp1.out &  ./m3ftp -p 0 **62 22** file2-10M.dat file2-10M.dat > pftp2.out & |
|  |  |  | ./m3ftp -g 0 60 20 file-100M.dat file0-100M.dat > gftp0.out & |
|  |  |  | ./m3ftp -p 0 60 20 file0-100M.dat file0-100M.dat > pftp0.out & |