****

**DMM Developer Manual**

[**1** **Introduction** 3](#_Toc507521014)

[**2** **DMM Overall Architecture** 3](#_Toc507521015)

[**3** **Core Components** 3](#_Toc507521016)

[**3.1** **nSocket** 3](#_Toc507521017)

[**3.2** **Framework** 4](#_Toc507521018)

[**3.3** **Adapters** 4](#_Toc507521019)

[**3.4** **RD** 4](#_Toc507521020)

[**3.5** **HAL** 4](#_Toc507521021)

[**3.6** **OMC** 5](#_Toc507521022)

[**4** **Plug-in Architectures** 5](#_Toc507521023)

[**4.1** **Overview** 5](#_Toc507521024)

[**4.2** **Plug-in interface** 7](#_Toc507521025)

[**4.2.1** **Interface of protocol stack adapter APIs** 7](#_Toc507521026)

[**4.2.1.1** **nStack\_module\_info** 7](#_Toc507521027)

[**4.2.1.2** **ep\_free\_ref** 8](#_Toc507521028)

[**4.2.1.3** **nstack\_stack\_register\_fn** 9](#_Toc507521029)

[**4.2.1.4** **nstack\_proc\_cb** 9](#_Toc507521030)

[**4.2.1.5** **DMM-adapter APIs** 10](#_Toc507521031)

[**4.2.1.6** **nstack\_adpt\_init** 10](#_Toc507521032)

[**4.2.2** **epoll Architecture** 11](#_Toc507521033)

[**4.2.2.1** **ep\_ctl** 13](#_Toc507521034)

[**4.2.2.2** **ep\_getevt** 13](#_Toc507521035)

[**4.2.2.3** **nstack\_event\_callback** 14](#_Toc507521036)

[**4.2.2.4** **nsep\_force\_epinfo\_free** 14](#_Toc507521037)

[**4.2.3** **Select** 15](#_Toc507521038)

[**4.2.3.1** **pfselect** 15](#_Toc507521039)

[**4.2.4** **fork** 16](#_Toc507521040)

[**4.2.4.1** **fork\_init\_child** 17](#_Toc507521041)

[**4.2.4.2** **fork\_parent\_fd** 18](#_Toc507521042)

[**4.2.4.3** **fork\_child\_fd** 18](#_Toc507521043)

[**4.2.4.4** **fork\_free\_fd** 18](#_Toc507521044)

[**4.2.5** **Multithreading** 19](#_Toc507521045)

[**4.2.6** **Resource recovery** 19](#_Toc507521046)

[**4.2.6.1** **obj\_recycle\_reg** 20](#_Toc507521047)

[**4.2.6.2** **obj\_recycle\_fun** 20](#_Toc507521048)

[**4.2.7** **LRD** 20](#_Toc507521049)

[**4.2.7.1** **nstack\_rd\_init** 21](#_Toc507521050)

[**4.2.7.2** **nstack\_get\_route\_data** 21](#_Toc507521051)

[**4.2.7.3** **nstack\_rd\_get\_stackid** 22](#_Toc507521052)

[**5** **Release file** 23](#_Toc507521053)

[**5.1** **libs** 23](#_Toc507521054)

[**5.2** **Posix API of nSocket** 23](#_Toc507521055)

# 

# **Introduction**

This document is used to guide the user-mode protocol stack developers to use the DMM framework. The document defines the interface that need to be registered to the DMM when the protocol stack is integrated into the DMM and the interface that the DMM provides to the protocol stack. In the document as well as in the code, DMM is also called nStack. The two names are interchangeable.

# **DMM Overall Architecture**

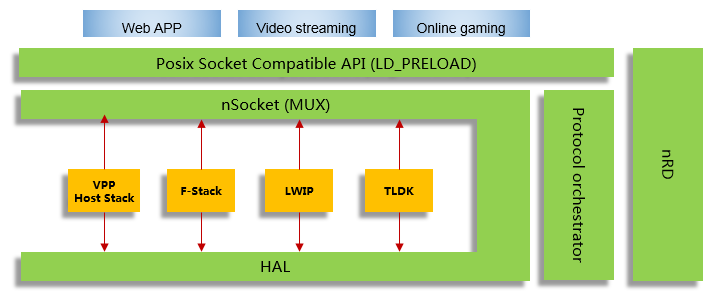


Figure1. The DMM software architecture.

The DMM framework provides posix socket APIs to the application. A protocol stack could be plugged into the DMM. DMM will choose the most suitable stack according to the policy of nRD to application.

nRD is a strategy system which is used to choose a protocol stack for application based on certain rules, such as network information, SLA, security and so on.

# **Core Components**

Figure1 shows an overview of the architecture design for DMM. DMM can be divided into six main components: nSocket, Framework (FW), adapters, RD, HAL and OMC (orchestration\manage\control).

## **nSocket**

“nSocket” provides user-friendly POSIX Compatible API of nStack, it intercepts the socket API calls from the user application via system call hijacking or the LD\_PRELOAD mechanism.

## **Framework**

Framework is a public framework module that includes shared-memory management, IPC-management, initialization framework, basic data structures and timer.

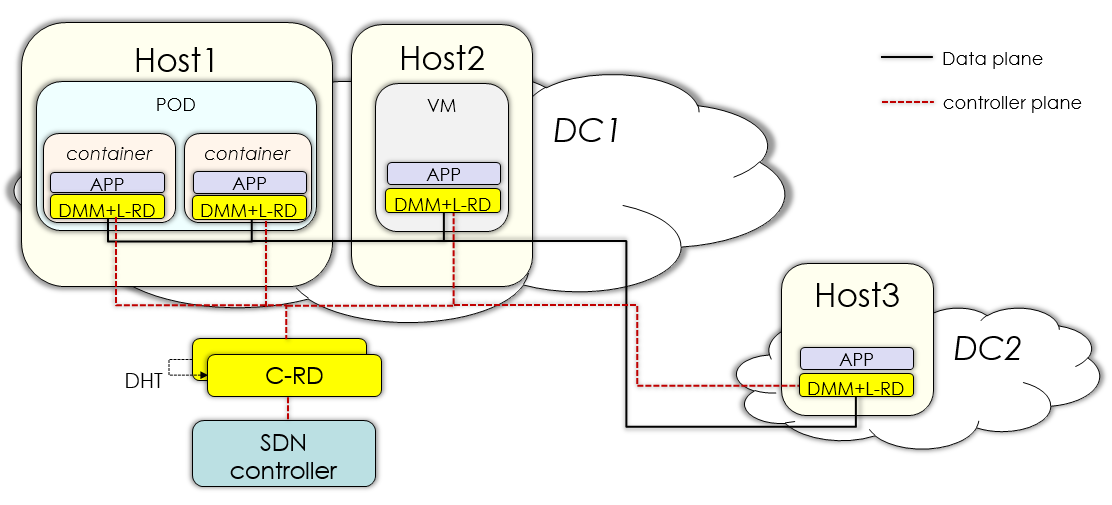
## **Adapters**

**DMM-adapter:** Used by protocol stack and implemented by DMM. Provides interfaces towards DMM.

**Protocol-stack-adapter:** Used by DMM and implemented by protocol stack. Provides interfaces towards protocol stack.

**Kernel-adapter:** Used and implemented by DMM. Help interact with kernel stack (if supported).

## **RD**

 Figure2. RD topology

Resource Discovery subsystem is responsible for “routing/rule” discovery, negotiation and decision-making (decision to select protocol stack).

## **HAL**

HAL (Hardware Abstraction Layer) layer provides the following functions for upper-layer protocol stack:

A unified user plane IO interface, shields the lower-level driver implementation differences. Mask the port implementation differences between IO layer and Protocol stack.

Initialize the network interface card.

## **OMC**

OMC (operation, Monitor and Control) is responsible for the whole protocol stack configuration management, monitoring and control.

It provides functionalities like query statistics, monitoring process status etc.

# **Plug-in Architectures**

## **Overview**

DMM-Plug-in architecture is an attractive solution for developers seeking to build protocol stacks that are multi-functional, customizable, and easily extensible.



Figure3. The Integration scheme

Figure 3 shows a typical deployment mode that protocol stack and application process are deployed in a pipeline mode (DMM also support the run-to-completion mode) across processes .IPC is used for communication between them.

1. DMM need to integrate the protocol stack adapter-library. The library should implement the interfaces defined in the list1. The protocol stack adapter is used for communicating with protocol stack process. In addition to the function of the protocol stack, this module must implement the interfaces defined by DMM, including the socket APIs, the epoll APIs, the fork APIs and the resource recycle APIs.

|  |  |  |
| --- | --- | --- |
| Interface |  | detail |
| ep\_ctl |  | Section 4.2.2.1 |
| ep\_getevt |  | Section 4. 2.2.2 |
| ep\_create |  | Section 4. 2.2.2 |
| fork\_init\_child |  | Section 4. 2.4.1 |
| fork\_parent\_fd |  | Section 4. 2.4.2 |
| fork\_child\_fd |  | Section 4. 2.4.3 |
| fork\_free\_fd |  | Section 4. 2.4.4 |
| module\_init |  | Section 4.3.5.5 |

List1. The functions provided by protocol stack adapter.

1. Protocol stack needs to integrate the DMM adapter-library. The library which is used to create and initialize shared-memory. The shared-memory is the highest performance method of IPC between nSocket and protocol stack. The library utilizes the plug-in interface to provide rich features to the protocol stack, such as resource recycle and event management.
   1. Protocol stack needs to register the recycle object and functions (See section 4.2.7 for details) to the DMM adapter. When the OMC receive the process exit signal, DMM adapter processes the exit signal, and trigger resource recovery action. Protocol stack does not need to focus on the exit of the application.
   2. Protocol stack needs to call the nstack\_adpt\_init (See section 4.2.1 for details) defined in the DMM adapter to create and initialize the shared memory object used by DMM.The init function returns the nStack\_adpt\_cb (See section 4.2.1.1 for details) object to the protocol stack. The protocol stack calls the callback functions when processing packet.

Then DMM adapt-library should implement the interface defined in the list2.

|  |  |  |
| --- | --- | --- |
| Interface |  | detail |
| nstack\_event\_callback |  | Section 4.2.2.3 |
| nsep\_force\_epinfo\_free |  | Section 4.2.2.4 |
| obj\_recycle\_reg |  | Section 4.2.6.1 |
| obj\_recycle\_fun |  | Section 4.2.6.2 |

List2. The function provided by DMM-adapter. This will be used by protocol Stack

* 1. In the pipeline mode, initialization of nSocket does not create shared memory; instead it looks up and attaches the shared memory created by the DMM adapter.
  2. nRD reads the routing information from the configuration file to make a decision as nSocket selects the protocol stack.

With these modules, DMM provides a unified socket interface, and support epoll, select, fork, zero copy, resource recovery features.

## **Plug-in interface**

### **Interface of protocol stack adapter APIs**

The following APIs must be implemented by the protocol stack adapter. They are used by nSocket module to do its work.

### **nStack\_module\_info**

When the protocol stack is integrated into DMM, the first step is adding new protocol stack module information.

|  |
| --- |
| typedef struct \_ nstack\_module\_keys {  const ns\_char\* modName; /\*stack name\*/  const ns\_char\* registe\_fn\_name; /\*stack register func name\*/  const ns\_char\* libPath; /\*if libtype is dynamic, it is the path of lib\*/  ns\_char deploytype; /\*depoly model type: model type1, model type2, model type3\*/  ns\_char libtype; /\*dynamic lib or static lib\*/  ns\_char ep\_free\_ref; /\*when epoll information free, need to wait that stack would not notify event \*/  ns\_char default\_stack; /\*whether is default stack: when don't know how to choose stack, just use default stack firstly \*/  ns\_int32 priority; /\*reserv\*/  ns\_int32 maxfdid; /\* the max fd id, just used to check\*/  ns\_int32 minfdid; /\* the min fd id, just used to check \*/  ns\_int32 modInx; /\* This is alloced by nStack , not from configuration \*/  } nstack\_module\_keys;    nstack\_module\_keys g\_nstack\_module\_desc[] ={};  typedef enum {  NSTACK\_MODEL\_TYPE1, /\*nSocket and stack belong to the same process\*/  NSTACK\_MODEL\_TYPE2, /\*nSocket and stack belong to different processes,  \*and nStack don't take care the communication between stack and stack adpt\*/  NSTACK\_MODEL\_TYPE3, /\*nSocket and stack belong to different processes, and stax-x-adapter was  spplied to communicate whit stack\*/  NSTACK\_MODEL\_INVALID,  } nstack\_model\_deploy\_type; |

DMM uses g\_nstack\_module\_desc to manage the stack initialization information, e.g. registration function name, lib path, etc.

When the nSocket calls the nstack\_stack\_registe\_fn() (See section 4.2.1.3 for details) , the information returned by the protocol stack adapter is stored in nstack\_module\_info.

|  |
| --- |
| typedef struct \_\_NSTACK\_MODULE {  char modulename[NSTACK\_PLUGIN\_NAME\_LEN];  ns\_int32 priority;  void\* handle;  nstack\_proc\_cb mops;  ns\_int32 ep\_free\_ref; //ep information need free with ref  ns\_int32 modInx; // The index of module  ns\_int32 maxfdid; //the max fd id  ns\_int32 minfdid; //the min fd id  } nstack\_module;  typedef struct{  ns\_int32 modNum; // Number of modules registed  ns\_int32 linuxmid;  ns\_int32 spmid; /\* Stackx module index. added by zengyi 00353502 \*/  nstack\_module \*defMod; // The default module  nstack\_module modules[NSTACK\_MAX\_MODULE\_NUM];  } nstack\_module\_info;  nstack\_module\_info g\_nstack\_modules = {  .modNum = 0,  .linuxmid = -1,  .modules = {{{0}}},  .defMod = NULL,  } |

CS2. The information of nstack\_module\_info

### **ep\_free\_ref**

This field in data structure nstack\_module\_keys is used in pipe-line mode. When the application process exits, the socket resources in the application process are released, but the resources in the protocol stack are not released. At this time, the protocol stack needs to continue accessing the epoll shared resource to send events.

So, you cannot free the resources of epoll immediately after the application exits. Only when the resources of the protocol stack are completely released, the epoll share resource can be released and it is done by the interface provided by the DMM adapter.

This field marks whether the stack needs to release the epoll resources. If it is 1, DMM will not release resources when close is called, and the stack MUST calls the interface provided by DMM adapter to release after releasing its own resources. If it is 0, DMM (nSocket) released the source directly.

### **nstack\_stack\_register\_fn**

Function prototypes:

|  |
| --- |
| typedef int (\*nstack\_stack\_register\_fn)(nstack\_proc\_cb \*proc\_fun, nstack\_event\_cb \*event\_ops); |

nstack\_socket\_ops: Defined in declare\_syscalls.h.tmpl file and is same as in Posix.

nstack\_event\_ops**:**

|  |
| --- |
| typedef struct \_\_nstack\_event\_cb {  void\* handle; /\*current so file handler\*/  int type; /\*nstack is assigned to the protocol stack and needs to be passed  to nstack when the event is reported\*/  int (\*event\_cb)(int protoFD, int type, int events);  } nstack\_event\_cb; |

Detail in the Section 4.2.3

### **nstack\_proc\_cb**

|  |
| --- |
| typedef struct \_\_nstack\_proc\_ops{  nstack\_socket\_ops socket\_ops; /\*posix socket api\*/  nstack\_extern\_ops extern\_ops; /\*other proc callback\*/  } nstack\_proc\_cb;  typedef struct \_\_nstack\_extern\_ops {  int (\*module\_init) (void); /\*stack module init \*/  int (\*fork\_init\_child) (pid\_t p, pid\_t c); /\*after fork, stack child process init again if needed. \*/  void (\*fork\_parent\_fd) (int s, pid\_t p); /\*after fork, stack parent process proc again if needed. \*/  void (\*fork\_child\_fd) (int s, pid\_t p, pid\_t c); /\*after fork, child record pid for recycle if needed. \*/  void (\*fork\_free\_fd) (int s, pid\_t p, pid\_t c); /\*for SOCK\_CLOEXEC when fork if needed. \*/  unsigned int (\*ep\_ctl) (int epFD, int proFD, int ctl\_ops, struct epoll\_event \* event, void \*pdata); /\*when fd add to epoll fd, triggle stack to proc if need \*/  unsigned int (\*ep\_getevt) (int epFD, int profd, unsigned int events); /\*check whether some events exist really \*/  int (\*ep\_prewait\_proc) (int epfd); /\* The pretreatment before epwait by stack-x \*/  int (\*stack\_fd\_check) (int s, int flag); /\* check whether fd belong to stack, if belong, return 1, else return 0 \*/  int (\*stack\_alloc\_fd) (); /\*alloc a fd id for epoll \*/  int (\*peak) (int s); /\*used for stack-x , isource maybe no need \*/  } nstack\_extern\_ops; |

### **DMM-adapter APIs**

For the multi-process model, DMM provide a library with resource initialization, resource recycling, event notification and other functions to the protocol stack.

nstack\_adpt\_fun is the core object between protocol stack and DMM. It defines some pointers which can be used by protocol stacks.

|  |
| --- |
| typedef struct nsfw\_com\_attr  {  int policy;  int pri;  } nsfw\_com\_attr;  typedef struct \_\_nstack\_dmm\_para  {  nstack\_model\_deploy\_type deploy\_type;  int proc\_type;  nsfw\_com\_attr attr;  int argc;  char \*\*argv;  } nstack\_dmm\_para;  int nstack\_adpt\_init (nstack\_dmm\_para \* para) |
|  |

### **nstack\_adpt\_init**

**Declaration**:

|  |
| --- |
| int nstack\_adpt\_init (nstack\_dmm\_para \* para) |

**Description**:

The protocol stack calls this function to initialize the DMM framework and get the callback function from DMM.

**Parameter:**

**nstack\_dmm\_para:** Input arguments described below.

***deploy\_type***: Decides whether protocol stack is deployed in same process along with DMM or in a separate process.

***proc\_type*** : Type of process.

***attr***: Thread policy and thread priority.

***argv*** : Arguments to initialize memory.

### **epoll Architecture**

The function of these data structures is similar to the one in linux.

|  |
| --- |
| struct eventpoll {  sys\_sem\_st lock; /\*eventpoll lock\*/  sys\_sem\_st sem; /\* rdlist lock \*/  sem\_t waitSem; /\*sem for epoll\_wait \*/  struct ep\_hlist rdlist; /\*ready epitem list\*/  struct ep\_hlist txlist; /\*pad\*/  struct ep\_rb\_root rbr; /\* rbtree\*/  int epfd; /\*epoll fd\*/  u32 pid; /\* Pid of the process who create the structure, the resources used to release \*/  nsfw\_res res\_chk; /\* Verify whether resources are repeatedly released \*/  }; |

|  |
| --- |
| typedef struct{  int iindex;  int iNext;  int fd;  i32 fdtype; /\* 0: socket fd, 1: epoll fd \*/  i32 rlfd; /\* copy of fdInf->rlfd \*/  i32 rmidx; /\* copy of fdInf->rmidx \*/  i32 protoFD[NSEP\_SMOD\_MAX];  i32 epaddflag[NSEP\_SMOD\_MAX];  struct eventpoll \*ep;  sys\_sem\_st epiLock;  sys\_sem\_st freeLock;  struct ep\_list epiList; /\* This restore the epitem of this file descriptor \*/  u32 sleepTime; /\* add for NSTACK\_SEM\_SLEEP \*/  nsep\_pidinfo pidinfo;  nsfw\_res res\_chk;  void\* private\_data;  i32 reserv[4];  } nsep\_epollInfo\_t; |

|  |
| --- |
| struct epitem {  struct ep\_rb\_node rbn;  struct ep\_hlist\_node rdllink;  struct ep\_hlist\_node lkFDllink;  int nwait;  struct eventpoll \*ep;  nsep\_epollInfo\_t \*epInfo;  struct epoll\_event event;  struct list\_node fllink;  struct ep\_hlist\_node txlink;  unsigned int revents;  int fd;  u32 pid;  void\* private\_data;  nsfw\_res res\_chk;  }; |

When the application calls epoll\_create(), nSocket first creates socket or epoll object in the data type of nsep\_epollInfo\_t, then creates an eventpoll object. For the created nsep\_epollInfo\_t object, when the file descriptor is an epoll socket, ep points to eventpoll and the eplist is not used. When the file descriptor is a normal socket, the epiList restores the fd and the ep is not used.

When the application calls epoll\_ctl() to add a socket to the epoll, nSocket allocates an epitem object. The epitem object is added to a rbtree pointed by epList in nsep\_epollInfo\_t, which is the epoll created in the previous step. In the epitem object just allocated, nSocket uses ep to point to eventpoll, revents to store event that is produced by the protocol stack, epinfo to point to the epoll object in the data type of nsep\_epollInfo\_t. This is how DMM connects the epoll, event, and socket together.

When the application calls epoll\_wait(), nSocket scans the rdlist in eventpoll to get events and return them to the application.

Figure 3 is Runtime Graphs of nSocket in epoll function:



Figure 4: flow chart of various epoll functions:



Figure5: flow chart of nstack\_event\_callback and nsep\_force\_epinfo\_free in protocol Stack

### **ep\_ctl**

|  |
| --- |
| int (\*ep\_ctl)(int proFD, int ctl\_ops, struct epoll\_event \*event, void \*pdata); |

**Functional description:**

1. Used by the application to inform protocol stack to add/modify/delete file descriptor (FD) to/from epoll and monitor events.

2. When the application adds new events or modifies existing events, it gets prompt notification.

3. pdata stores the nsep\_epollInfo\_t corresponding to the FD. It is used to pass the value to the protocol stack event notification API when an event occurs.

**Parameters:**

***proFD***: sock file descriptor created by the protocol stack.

***ctl\_ops***: 0 add, 1 modify, 2 delete。

***event***: similar to Linux “struct epoll\_event”。

***pdata***: Private data that DMM passes to the protocol stack.

**Interface implemented by:** Protocol stack.

### **ep\_getevt**

|  |
| --- |
| void (\*ep\_getevt)(int epFD, int profd, unsigned int events); |

**Description:**

To check if an event in protocol stack exist. Due to multi-threading or other reasons, it’s possible that protocol stack is busy and causes delay in event notification. Or event may be cleared by some application operation. Application may use the API to check whether the event still exists.

**Parameter:**

***epFD***: epoll FD.

***proFD***: sock file descriptor created by the protocol stack.

***event******s***: Events which need to check.

**Interface implemented by:** Protocol stack.

### **nstack\_event\_callback**

|  |
| --- |
| int (\*nstack\_event\_callback)(void \*pdata, int events) |

**Description:**

Called by the protocol stack when there is an event occurs.

**Parameter Description:**

***pdata***: pdata is the private data passed in ep\_ctl().

***events***: Whatever event occurred

**Interface implemented by:** nStack

### **nsep\_force\_epinfo\_free**

|  |
| --- |
| int (\*nsep\_force\_epinfo\_free)(void \*pdata) |

**Description:**

When closes fd, the protocol stack frees the resources related to fd. This API is called by the protocol stack to release private data.

**Precautions for use:**

This API should only be called when ep\_free\_ref is configured 1 in nstack\_module\_keys.

**Parameter:**

***pdata***: pdata is the private data in ep\_ctl().

**Interface implemented by:** DMM

### **Select**

Select requires the protocol stack adapter module to provide a select API in order to query for existing events. When the application calls select(), DMM starts a separate thread to fetch the select event for each stack and sends it back to the application.



Figure 6: flow chart of select function

### **pfselect**

|  |
| --- |
| int \*pfselect(int s, fd\_set\* readfds, fd\_set\* writefds, fd\_set\* exceptfds, struct timeval\* timeout) |

**Description:**

Query whether there is a corresponding protocol stack event.

**Parameter description:**

***s***: The maximum file descriptor value in the collection

***readfds***: file descriptor set will be watched for read event

***writefds***: file descriptor set will be watched for write event

***exceptfds***: file descriptor set will be watched for exception cases

**Interface implemented by:** Protocol Stack

### **fork**

fork() creates a child process by making an exact duplicate of the calling process in a separate address spaces. Each process, parent and child, has its own process address space where memory segments, such as code segment, data segment, stack segment, etc., are placed.

There are a few exceptions.

1. The fork system call duplicates all the file descriptors of the parent into the child, so that the file descriptors in the parent and child point to the same files.
2. The shared memory is still shared between parent and child process after fork ().
3. Child process clones only thread, which executed it.

When parent process or child process releases the shared resources on process quit or kill, it affects the other process.

Also if the child process wants to inherit other threads in the parent process, the child process needs to pull or create those threads after the fork.

Both parent and child process can access the shared memory, it create conflicts.

Because of these issues, DMM provides the fork APIs to take care of processing before and after calling the Linux fork interface. It includes interface lock and shared resource lock, reference count incrimination, saving the PID of the subprocesses, reinitialization and SOCK\_CLOEXEC option in socket special handlings.

Each shared memory data structure in DMM contains a reference count and a list of process pids, so that the shared memory can be managed between parent and child processes. On each fork, the reference count is increased by 1 and the child process id is added to the pid array.

|  |
| --- |
| struct xxx{  xxx，  int ref； / \* Reference count \* /  pid\_t pid\_array[];  / \* for father and son after the process list \* /  }; |

DMM needs assistance from protocol stack to release or hold the shared resources. Protocol stack is to provide stack specific implementation on these DMM APIs.

The following flow chart shows how fork process is handled:



#### **fork\_init\_child**

|  |
| --- |
| int (\* fork\_init\_child) (pid\_t p, pid\_t c) |

**Description:**

After fork, the API is used to add child process pid into the pid list in the share socket data structure. When the child process exits, only the resources in the child process are released unless the shared resource reference count reaches 0.

**Parameter Description:**

***p***: parent process pid

***c***: child process pid

**Interface Implemented by:** Protocol stack.

#### **fork\_parent\_fd**

void (\* fork\_parent\_fd)(int s, pid\_t p)

**Description:**

For the socket with the SOCK\_CLOEXEC flag on, when it is closed after fork, only the file descriptor is recycled, no other resource is released. It is a special treatment for SOCK\_CLOEXEC.

**Parameter:**

***s***: socket fd

***p***: parent process pid

**Interface Implemented by:** Protocol stack.

#### **fork\_child\_fd**

void (\* fork\_child\_fd)(int s, pid\_t p, pid\_t c)

**Description:**

After fork, the API is used to add child process pid into the pid list in the share socket data structure. When the child process exits, only the resources in the child process are released unless the shared resource reference count reaches 0.

**Parameters:**

***s***: file descriptor of the socket to be operated

***p***: parent process pid

***c***: child process pid

**Interface Implemented by:** Protocol stack.

#### **fork\_free\_fd**

void (\* fork\_free\_fd)(int s, pid\_t p, pid\_t c);

**Description:**

For the socket with the SOCK\_CLOEXEC flag on, when it is closed after fork, only the file descriptor is recycled, no other resource is released. It is a special treatment for SOCK\_CLOEXEC.

**Parameters:**

***s***: socket fd.

***p***: parent process pid.

***c***: child process pid.

**Interface Implemented by:** Protocol stack

### **Multithreading**

nSocket supports multi-thread. nSocket uses a ref and state to support the feature.

### **Resource recovery**

Protocol stack must implement resource recovery. When the application crashes or exits, the resources allocated by protocol stack must be released, otherwise it causes resources leaking and can crash the system.

DMM provides the following mechanism to support the feature.

There is certain relationship between resource recovery and fork. Each fork increases the reference count by 1 and saves the child process PID to the corresponding shared resource. On exiting of the process, decreases the reference count by 1 if the exiting process PID equals to one of the saved PIDs and releases the shared resource if the reference count is 0, do not release the shared resource if the reference count in not zero.

1. There are two kinds of resource recovery, normal exit and abnormal process exit. Resources are freed on normal exit operation. But on the abnormal process exits, there is no one to clean up the shared resources or decrease the reference count, so we rely on the separate process to monitor the processes and notify the stack process to release the resources.
2. To support resource recovery, the protocol stack needs to define a resource object type in responding to DMM request and registers the recycle function at initialization time.

#### **obj\_recycle\_reg**

TO DO

#### **obj\_recycle\_fun**

TO DO

### **LRD**

LRD function is mainly the protocol stack routing function, the internal storage of a protocol stack selection policy table. When multiple protocol stacks are integrated, according to the incoming IP address, protocol stack type and other information, you can select a suitable protocol stack from the protocol Stack selection policy table. During nSocket module initialization, it invokes the LRD module initialization API to register the protocol stack information and the protocol stack selection policy gets the interface.

The initialization function for the LRD module is defined as follows:

|  |
| --- |
| int nstack\_rd\_init (nstack\_stack\_info \*pstack, int num, nstack\_get\_route\_data \*pfun, int fun\_num)  typedef struct \_\_nstack\_rd\_stack\_info  {  /\*stack name \*/  rd\_stack\_plane\_map stack;  /\*stack id \*/  int stack\_id;  /\*when route info not found, high priority stack was chose, same priority chose fist input one \*/  int priority; /\*0: highest: route info not found choose first \*/  } nstack\_rd\_stack\_info;  /\*rd local data\*/  typedef struct \_\_rd\_local\_data {  nstack\_rd\_stack\_info \*pstack\_info;  int stack\_num;  nstack\_rd\_list route\_list[RD\_DATA\_TYPE\_MAX]; /\*route table \*/  nstack\_get\_route\_data sys\_fun[NSTACK\_SYS\_FUN\_MAX]; /\*rd data sys proc function list \*/  int fun\_num;  } rd\_local\_data;  rd\_local\_data \*g\_rd\_local\_data = NULL;  /\*get rd info. if return ok, data callee alloc memory, caller free, else caller don't free\*/  int (\*nstack\_get\_route\_data)(rd\_route\_data \*\*data, int \*num); |

When a new protocol stack is integrated into DMM, the protocol stack selection configuration file will not directly match with the protocol stack name, but with the plane name, which is more descriptive about stack and other interface technology, therefore the RD has a map between the protocols stack and plane name.

|  |
| --- |
| typedef struct \_\_rd\_stack\_plane\_map{  char stackname[STACK\_NAME\_MAX]; /\* Stack name \*/  char planename[RD\_PLANE\_NAMELEN]; /\* Plane name \*/  int stackid; /\*stack id \*/  } rd\_stack\_plane\_map;  rd\_stack\_plane\_map g\_nstack\_plane\_info[] = {  {{“stackx”}, {“nstack-dpdk”} , -1},  {{“kernel”}, {“nstack-linux”} , -1},  }; |

When application calls the DMM socket, connect, sendto, sendmsg APIS, it triggers the nSocket module to invoke the nstack\_rd\_get\_stackid() API to get the route information. The protocol stack is then selected based on the returned protocol stack ID.

#### **nstack\_rd\_init**

|  |
| --- |
| int nstack\_rd\_init(nstack\_stack\_info \*pstack, int num, nstack\_get\_route\_data pfun, int fun\_num) |

**Description:**

Initializes the RD module g\_rd\_local\_data to receive and save the protocol stack information provided by nSocket. Also provides the API to get protocol stack selection information.

When initialized, nSocket passes all the integrated protocol stacks information to the RD module. The information includes the protocol stack name, nSocket assigned stack id after integration and selects priority.

**Parameters:**

***pStack***: a list of the protocol stack info

***num***: the number of protocol stack info passed in.

***pfun***: function to get the protocol stack selection information

***fun\_num***: the number of pfun passed in

**Interface implemented by:** DMM

#### **nstack\_get\_route\_data**

|  |
| --- |
| int (\*nstack\_get\_route\_data)(char \*planename, rd\_route\_data \*\*data, int \*num); |

**Description:**

Obtain the protocol stack selection information according to the protocol stack plane name.

**Parameter Description:**

***planename***: description on stack and other interface technology. nSocket provides the protocol stack name. The configuration file saves an outbound interface type, which is corresponding to a protocol stack.

***data***: protocol stack selection information.

|  |
| --- |
| typedef enum \_\_rd\_data\_type{  RD\_DATA\_TYPE\_IP, / \* According to the ip address to select the protocol stack \* /  RD\_DATA\_TYPE\_PROTO, / \* Select protocol stack by protocol type \* /  RD\_DATA\_TYPE\_MAX,  }rd\_data\_type;  typedef struct \_\_rd\_route\_data{  rd\_data\_type type; / \* Stored protocol stack selection type \* /  char stack\_name[RD\_PLANE\_NAMELEN]; / \* Stack plane name \*/  union {  rd\_ip\_data ipdata; / \* Select the IP address of the protocol stack type \* /  unsigned int proto\_type; / \* Select protocol information for this stack \* /  };  } rd\_route\_data; |

***num***: The number of routing information returned.

**Interface implemented by:** DMM

#### **nstack\_rd\_get\_stackid**

|  |
| --- |
| int nstack\_rd\_get\_stackid(nstack\_rd\_key\* pkey, int \*stackid) |

**Description:**

Select parameters according to the protocol stack to obtain the protocol stack id corresponding to the parameter.

**Parameters:**

***pkey***: protocol stack selection key.

|  |
| --- |
| typedef struct \_\_nstack\_rd\_key{  rd\_data\_type type; / \* Select protocol type parameters \* /  union {  unsigned int ip\_addr; / \* This field is valid if the type is RD\_DATA\_TYPE\_IP, which is  the IP address of the connection to be initiated \* /  unsigned int proto\_type; / \* This field is valid if the type is RD\_DATA\_TYPE\_PROTO, which  is the protocol type of the socket created at the time \* /  };  } nstack\_rd\_key; |

***stackid***: output parameter; nSocket internal number of the protocol stack id.

**Interface implemented by:** DMM

# **Release file**

## **libs**

**libnStackAPI.so**: The API library that DMM provides to the application.

**libnStackAdapt.so**: DMM provides an adapter library for cross process protocol stack connections.

## **Posix API of nSocket**

DMM provides socket APIs and integrates various protocol stacks. DMM selects and invokes a specific protocol stack based on the information, such as IP and protocol type, from the application. The protocol stacks do not interact with the application. After the selection, what features an API supports are determined by the implementation provided by the protocol stack itself.

The standard socket APIs defined by DMM are as follows:

|  |  |
| --- | --- |
| **Interface definition** | **Interface Description** |
| int socket(int, int, int) | same as corresponding POSIX apis |
| int bind(int, const struct sockaddr\*, socklen\_t) | same as corresponding POSIX apis |
| int listen(int, int) | same as corresponding POSIX apis |
| int shutdown(int, int) | same as corresponding POSIX apis |
| int getsockname(int, struct sockaddr\*, socklen\_t\*) | same as corresponding POSIX apis |
| int getpeername(int, struct sockaddr\*, socklen\_t\*) | same as corresponding POSIX apis |
| int getsockopt(int, int, int, void\*, socklen\_t\*) | same as corresponding POSIX apis |
| int setsockopt(int, int, int, const void\*, socklen\_t) | same as corresponding POSIX apis |
| int accept(int, struct sockaddr\*, socklen\_t\*) | same as corresponding POSIX apis |
| int accept4(int, struct sockaddr\*, socklen\_t\*, int flags) | same as corresponding POSIX apis |
| int connect(int, const struct sockaddr\*, socklen\_t) | same as corresponding POSIX apis |
| ssize\_t recv(int , void\*, size\_t, int) | same as corresponding POSIX apis |
| ssize\_t send(int, const void\*, size\_t, int) | same as corresponding POSIX apis |
| ssize\_t read(int, void\*, size\_t) | same as corresponding POSIX apis |
| ssize\_t write(int, const void\*, size\_t) | same as corresponding POSIX apis |
| ssize\_t writev(int, const struct iovec \*, int) | same as corresponding POSIX apis |
| ssize\_t readv(int, const struct iovec \*, int) | same as corresponding POSIX apis |
| ssize\_t sendto(int, const void \*, size\_t, int, const struct sockaddr \*, socklen\_t) | same as corresponding POSIX apis |
| ssize\_t recvfrom(int, void \*, size\_t, int, struct sockaddr \*, socklen\_t \*) | same as corresponding POSIX apis |
| ssize\_t sendmsg(int, const struct msghdr \*, int flags) | same as corresponding POSIX apis |
| ssize\_t recvmsg(int, struct msghdr \*, int flags) | same as corresponding POSIX apis |
| int close(int) | same as corresponding POSIX apis |
| int select(int, fd\_set\*, fd\_set\*, fd\_set\*, struct timeval\*) | same as corresponding POSIX apis |
| int ioctl(int, unsigned long, unsigned long) | same as corresponding POSIX apis |
| int fcntl(int, int, unsigned long) | same as corresponding POSIX apis |
| int epoll\_create(int) | same as corresponding POSIX apis |
| int epoll\_ctl(int, int, int, struct epoll\_event \*) | same as corresponding POSIX apis |
| int epoll\_wait(int, struct epoll\_event \*, int, int) | same as corresponding POSIX apis |
| pid\_t fork(void) | Before/after calling base fork() need to handle other processing like ref\_count etc. |