# M3-IPC 1.6 Programmer's Guide

22th of April 2015 [software version: M3-IPC 3.0-for Linux kernel 2.6.32 ]

---

Table of Contents

# 1 Introduction

This document is intended to assist software developers who are writing applications that use M3-IPC to communicate trasparently local and remote Linux processes.

# 2 M3-IPC Fundamentals

M3-IPC are InterProcess Communications APIs that emulates MINIX 3 IPC ones. A Linux process can use these primitives as it was a MINIX process.

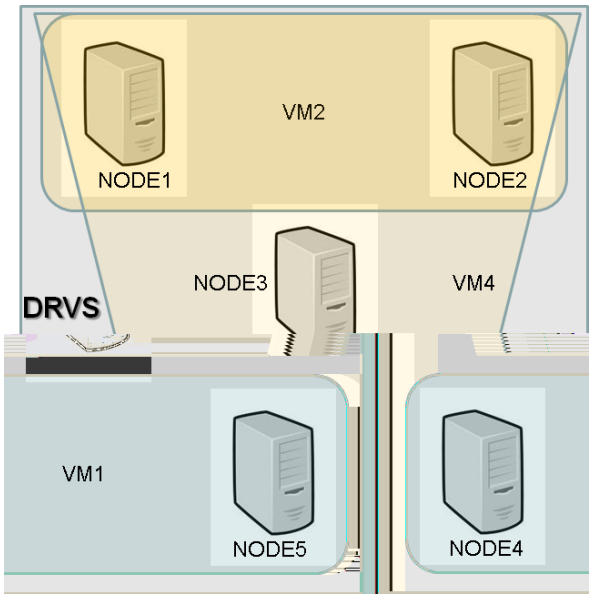
Some extra functionality were added:

- It supports different virtual environments or Virtual Machines (VM) where Linux process can communicate between them using M3-IPC without interferences with other Linux processes of other VMs. In the future, VM will be integrated with Linux Containers (LXC).
- It supports message and block data transfers among processes residing in different nodes of a cluster in a transparent way.

A brief summary of the major concepts used in M3-IPC is provided in the following sections.

## 2.1 Architecture

As M3-IPC is the foundation for a Distributed Resources Virtualization System (DRVS). A DRVS is a kind of SSI-DOS, that support operating system level VMs. In a DRVS the processes of a VM can expand in multiple computers or network ***nodes***, and multiple processes of different VMs can reside on one node.



Before the M3-IPC facilities can be used, the DRVS must be started in each node using specifying some parameters as the number of VMs that can be configured inside the DRVS.

## 2.2 Messaging Overview

M3-IPC applications typically communicate each other by exchanging data units, known as , between registered processes identified by .

From an application's perspective, a is a fixed byte string with a structure defined in " " but the use of the fields is determined by the applications.

A Virtual Machine (VM) is an execution environment for a set of Linux processes that can exchange messages. Each VM is identified by a . An identifies an entity that can send and receive messages inside a VM, therefore the same can exist in different VMs.

A Linux processes can only use M3-IPC primitives if it registers to the M3-IPC kernel using the kernel call. As a result it returns the process (or thread) that identifies it to other process that run inside the same VM.

IPCs are performed using the rendezvous message passing. Rendezvous means that the sender is blocked until the receiver actually receives the message, or it blocks the receiver until the sender actually sends to it the desired message.

### *2.2.1 Special M3-IPC endpoints*

As in MINIX, M3-IPC has special endpoints.

- : used to indicate 'any process' (can only be used in )
- : used to indicate 'no process at all'
- : used to indicate 'own process'

### *2.2.2 Endpoint Types*

As M3-IPC can transfer message and data among processes on different nodes of a network in a transparent way, endpoint can be:

- : Processes or Threads running in the local node. They have their PIDs in the local Linux.
- : Processes or Threads running in a remote node. They have not a Linux PID, but a local process can receive or send messages/data to a remote endpoint in the same way as if the endpoint were local.

### *2.2.3 Endpoint Resolution*

Whenever an application needs the of a local Linux process, it can use . It gets the process Linux PID as the parameter and returns the endpoint only if it is registered and if it is alive, otherwise returns an error. Only registered processes (in a VM) can use .

# 3 MINIX emulated IPC APIs

## 3.1 MINIX IPC functions

The following utility functions are available to programmers:

mnx_send() – sends a message to a process identified by an endpoint on the same VM. While the destination does not make the sender is blocked.

mnx_receive() – receives a message from a process identified by an endpoint on the same VM. While the sender does not make or or the caller is blocked.

mnx_sendrec() – sends a message to a process identified by an endpoint on the same VM and then waits for the reply message from the same process. The caller is blocked until it receives the reply.

mnx_notify() - sends a **notify** message to a process identified by an endpoint on the same VM. If the receiver does not make the sender is not blocked. Instead a bit is set into the destination process descriptor that signals the notification message and the source of it. The notify message only carries the sender's endpoint, the sender's process number and a sending timestamp.

mnx_vcopy() – It allows a process (the requester) to copy a block of data from the address space of a source process to the address space of a destination process.

Other functions were added for optimization:

mnx_reply() – sends a message to a process identified by an endpoint on the same VM. The destination must be blocked waiting a message from the caller of the function, otherwise returns an error. ***mnx_reply()*** never blocks except when the destination is remote and it must wait for the message acknowledge.

mnx_rcvrqst() – receives a message from ANY process on the same VM. While it does not receive any message, it is blocked.

Further information about each of these functions is provided in the following sections.

### *3.1.1 Message types*

Minix has a hard-code predefined message types. Their formats are in

```
typedef struct {
  int m_source;               /* who sent the message */
  int m_type;                 /* what kind of message is it */
  union {
      mess_1 m_m1;
      mess_2 m_m2;
```

```
        mess_3 m_m3;
        mess_4 m_m4;
        mess_5 m_m5;
        mess_6 m_m6;
        mess_7 m_m7;
        mess_8 m_m8;
        mess_9 m_m9;
    } m_u;
} message;
```

If the application programmer needs another type, it can add the new type in the payload union ( ) and must recompile the kernel. Be careful because if the message size change, all M3-IPC utilities and libraries must be recompiled.

### *3.1.2 mnx_send(), mnx_reply()*

```
long mnx_send(int dst_ep, message* m_ptr)
long mnx_send_T(int dst_ep, message* m_ptr, long to_ms)
long mnx_reply(int dst_ep, message* m_ptr)
long mnx_reply_T(int dst_ep, message* m_ptr, long to_ms)
```

The function sends a message pointed by to a process identified by on the same VM. While the destination does not make or it is waiting in the receiving half of the sender is blocked. If the destination is remote, it is blocked until it receives the remote acknowledge. The timed version waits until a timeout (to_ms in miliseconds) expires.

The function is a specialized version of intended to be used to send the reply from a server to a client process that it is waiting for this message in a This function never blocks for a local destination client, and blocks until it receives the acknowledge from the remote destination's client. The timed version waits until a timeout (to_ms in miliseconds) expires.

The values TIMEOUT_NOWAIT means that the function does not wait if it cannot complete its work.

The values TIMEOUT_FOREVER means that the function waits until it can complete its work. It has the same behavior than the not timed version.

Return codes:

- OK: The message was sent without error.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLNOTBIND: The caller is not binded to the M3-IPC system.
- EMOLBADPROC: An inconsistency exists between Linux kernel and M3-IPC.
- EMOLBADPID: An inconsistency exists between Linux kernel and M3-IPC.
- EMOLBADVMID: An inconsistency exists in M3-IPC.
- EMOLVMNOTRUN: The VM of the endpoint is not running.
- EMOLENDPOINT: The destination's endpoint is the same as the caller's endpoint.
- EMOLDSTDIED: The destination process has died or it is un-binded.

- EMOLNOTREADY: The caller is not in READY state for M3-IPC.
- EMOLBADNODEID: The destination's node is out of range.
- EMOLNOVMNODE: The destination's VM is not running in the destination's node.
- EMOLNOPROXY: The local proxy for the destination node is not running.
- EMOLNOTCONN: The local proxy for the destination node is running but not connected to the remote node.
- EMOLTIMEDOUT: The timeout has expired (for the timed version).
- (-ERESTARTSYS): If the process has received a Linux Signal while waiting.

### *3.1.3 mnx_receive(), mnx_rcvrqst()*

```
long mnx_receive(int src_ep, message* m_ptr)
long mnx_receive_T(int src_ep, message* m_ptr, long to_ms)
long mnx_rcvrqst(message* m_ptr)
long mnx_rcvrqst_T(message* m_ptr, long to_ms)
```

The function receives a message into the buffer pointed by from a process identified by an on the same VM or by process. While the sender does not make or or the receiver is blocked. The timed version waits until a timeout ( in miliseconds) expires.

The receives a message (request) into the buffer pointed by from a process that sent the message using It is a specialized and optimized function to be used in servers. The timed version waits until a timeout ( in miliseconds) expires.

Return codes:

- OK: The message was sent without error.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLNOTBIND: The caller is not binded to the M3-IPC system.
- EMOLBADPROC: An inconsistency exists between Linux kernel and M3-IPC.
- EMOLBADPID: An inconsistency exists between Linux kernel and M3-IPC.
- EMOLBADVMID: An inconsistency exists in M3-IPC.
- EMOLVMNOTRUN: The VM of the endpoint is not running.
- EMOLRANGE: The source's process number is out of range.
- EMOLENDPOINT: The source's endpoint is the same as the caller's endpoint.
- EMOLBADNODEID: The source's node ID is out of range.
- EMOLNOVMNODE: The destination's VM is not running in the destination's node.
- EMOLNOPROXY: The local proxy for the destination node is not running.
- EMOLNOTCONN: The local proxy for the destination node is running but not connected to the remote node.
- EMOLNOTREADY: The caller is not in READY state for M3-IPC.
- EMOLMSGSIZE: An error occurs trying to copy the message.
- EMOLSRCDIED: The message's source is not registered.

- EMOLPROCSTS: The message's source is not in sending state.
- EMOLTIMEDOUT: The timeout has expired (for the timed version).
- (-ERESTARTSYS): If the process has received a Linux Signal while waiting.

### *3.1.4 mnx_sendrec()*

```
long mnx_sendrec(int srcdst_ep, message* m_ptr)
long mnx_sendrec_T(int srcdst_ep, message* m_ptr, long to_ms)
```

This function sends a message pointed by to a process identified by on the same VM. Then it waits for the reply message from the same process into the same message buffer. The caller is blocked until it receives the reply. The timed version waits until a timeout ( in miliseconds) expires.

Return codes:

- OK: The message was sent without error.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLNOTBIND: The caller is not binded to the M3-IPC system.
- EMOLBADPROC: An inconsistency exists between Linux kernel and M3-IPC.
- EMOLBADPID: An inconsistency exists between Linux kernel and M3-IPC.
- EMOLBADVMID: An inconsistency exists in M3-IPC.
- EMOLVMNOTRUN: The VM of the endpoint is not running.
- EMOLDSTDIED: The message's destination is not registered.
- EMOLENDPOINT: The destination's endpoint is the same as the caller's endpoint.
- EMOLNOTREADY: The caller is not in READY state for M3-IPC.
- EMOLBADNODEID: The destination's node ID is out of range.
- EMOLNOVMNODE: The destination's VM is not running in the destination's node.
- EMOLNOPROXY: The local proxy for the destination node is not running.
- EMOLNOTCONN: The local proxy for the destination node is running but not connected to the remote node.
- EMOLTIMEDOUT: The timeout has expired (for the timed version).
- (-ERESTARTSYS): If the process has received a Linux Signal while waiting.

### *3.1.5 mnx_notify()*

```
long mnx_notify(int dst_ep)
```

This function sends a **notify** message to a process identified by on the same VM. If the receiver does not make the sender is not blocked. Instead a bit is set into the destination process descriptor that signals the notification message and the source of it. The notify message only carries the

sender's endpoint (`m_source`), the sender's process number (`m_type`) and the sending timestamp (`m2_l2`). See

Return codes:

- OK: The message was sent without error.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLNOTBIND: The caller is not binded to the M3-IPC system.
- EMOLBADPROC: An inconsistency exists between Linux kernel and M3-IPC.
- EMOLBADPID: An inconsistency exists between Linux kernel and M3-IPC.
- EMOLBADVMID: An inconsistency exists in M3-IPC.
- EMOLVMNOTRUN: The VM of the endpoint is not running.
- EMOLENDPOINT: The destination's endpoint is the same as the caller's endpoint.
- EMOLDSTDIED: The destination process has died or it is not registered.
- EMOLNOTREADY: The caller is not in READY state for M3-IPC.
- EMOLBADNODEID: The destination's node is out of range.
- EMOLNOVMNODE: The destination's VM is not running in the destination's node.
- EMOLNOPROXY: The local proxy for the destination node is not running.
- EMOLNOTCONN: The local proxy for the destination node is running but not connected to the remote node.

#### *3.1.6 mnx_vcopy()*

```
mnx_vcopy(int src_ep, (void*) src_addr, int dst_ep, (void*) dst_addr, bytes)
```

This function allows copy a block of data (until MAXCOPYLEN bytes) from one process user's address ( ) space to the other user's process address space ( ).

This function has three parties:

1. The Requester: The process that invokes the function.
2. The Sender: The process origin of the data to transfer.
3. The Destination: The process origin of the data to transfer.

The requester can be the same as Sender or Destination but must be a process with root's privileges (euid = 0).

If the copy is between local processes, the requester does not block, but for remote parties the requester blocks until the copy ends or an error is returned.

Return codes:

- OK: The message was sent without error.
- (-EPERM): The caller's **euid** not have the required permissions.

- EMOLRANGE: The number of bytes to copy is erroneous.
- EMOLENDPOINT: The source process is the same as the destination process or an invalid endpoint (ANY, NONE).
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLNOTBIND: The caller is not registered to the M3-IPC system.
- EMOLBADPROC: An inconsistency exists between Linux kernel and M3-IPC.
- EMOLBADPID: An inconsistency exists between Linux kernel and M3-IPC.
- EMOLBADVMID: An inconsistency exists in M3-IPC.
- EMOLVMNOTRUN: The VM of the endpoint is not running.
- EMOLNOTREADY: The caller is not in READY state for M3-IPC.
- EMOLSRCDIED: The source process has died or it is not registered.
- EMOLDSTDIED: The destination process has died or it is not registered.
- EMOLBUSY: A remote process descriptor is used for a remote operation. Try later.
- EMOLPROCSTS: The source/destination must be in RECEIVING state.
- EMOLNOVMNODE: The source/destination's VM is not running in the source/destination's node.
- EMOLNOPROXY: The local proxy for the source/destination node is not running.
- EMOLNOTCONN: The local proxy for the source/destination node is running but not connected to the remote node.
- EMOLBADNODEID: The destination's node is out of range.

# 4 M3-IPC Auxiliary Functions

## 4.1 Overview

The following auxiliary functions are available to programmers:

mnx_drvs_init() – Initialize the DRVS with the specified parameters.
mnx_drvs_end() – End the DRVS.
mnx_getdrvsinfo() – Gets the DRVS status and parameters information.
mnx_vm_init() – Initialize a VM and all its M3-IPC process descriptors.
mnx_vm_end() – End a VM and all its M3-IPC process descriptors.
mmx_getvminfo() - Gets the VM status and parameters information.
mnx_bind() – Registers a local Linux process into the M3-IPC kernel.
mnx_unbind() – De-registers a local Linux process from the M3-IPC kernel and release allocated resources.
mnx_rmtbind() – Registers a Remote Linux process into the M3-IPC kernel.
mnx_unbind() – De-registers a Remote Linux process from the M3-IPC kernel and release allocated resources.
mnx_getep() – Gets the endpoint of a registered (local) Linux process.
mnx_getprocinfo()-Gets the process status and parameters information.
mnx_setpriv() – Sets the process' privilege number (TEMPORARY).
mnx_vm_dump() – Logs the status of all VMs to the Linux circular buffer (dmesg).
mnx_vm_proc() – Logs the status of all processes of a specified VM to the Linux circular buffer (dmesg).

Further information about each of these functions is provided in the following sections.

### *4.1.1 mnx_drvs_init()*

```
long mnx_drvs_init(int nodeid, drvs_usr_t *du_addr)
```

This function starts the DRVS system setting the node ID and the maximum parameters for all the VMs in the local node. This function must be called with root's privileges.

The is defined as:

```
struct drvs_usr {
   int   d_nr_vms;   /* maximum number of VMs that can run in this node   */
   int   d_nr_nodes; /* maximum number of nodes for the DRVS              */
   int   d_nr_procs; /* maximum number of processes for each VM           */
   int   d_nr_tasks; /* maximum number of tasks  for each VM              */
   int   d_nr_sysprocs; /* maximum number of system processes for each VM */
   unsigned long int d_dbglvl; /* debub level mask                        */
   int   d_version;             /* version READ ONLY                     */
   int   d_subver;              /* subversion READ ONLY                  */
```

```
};
typedef struct drvs_usr drvs_usr_t;
```

The and fields inform about the M3-IPC version and subversion.

The allow to select the level of debugging messages to the kernel buffer. It is a bitmask that is an OR of the following debugging selection flags. A 0x00000000 means no debug any, and 0xFFFFFFFF means debug all.

| | |
|---|---|
| GENERIC | Some text about actions |
| INTERNAL | Some internal status variables |
| DBGPROCLOCK | Process Locking changes |
| DBGVMLOCK | VM Locking changes |
| DBGNODELOCK | Node Locking changes |
| DBGTASKLOCK | Linux Tasks Locking changes |
| DBGMESSAGE | Message contents |
| DBGCMD | Command contents |
| DBGVCOPY | Virtual Copy subcommands contents |
| DBGPARAMS | Function parameters and related values |
| DBGPROC | Process atributes and status |
| DBGPRIV | Privileges atributes and status |
| DBGPROCSEM | Process semaphore (to sleep and wakeup processes) |

Return codes:

- OK: The message was sent without error.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSBUSY: The DRVS is already running.
- EMOLRANGE: Some `drvs_usr_t` member has an invalid value.
- EMOLNOTDIR: A problem trying to create the /proc/drvs directory.
- EMOLBADF: A problem trying to create a file under the /proc/drvs directory.

### *4.1.2 mnx_drvs_end()*

```
long mnx_drvs_end()
```

This function ends the DRVS systems that it means:

- Ends all VMs.
- Ends all M3 processes.
- Ends all proxies.

Return codes:

- OK: The message was sent without error.
- (-EPERM): The caller's **euid** not have the required permissions.

- EMOLDRVSINIT: The is not running

### *4.1.3 mnx_getdrvsinfo()*

```
long mnx_getdrvsinfo(drvs_usr_t *drvs_usr_ptr)
```

This function gets DRVS status and parameter information.

Return codes:

- OK: The message was sent without error.
- (-EPERM): The caller's **euid** not have the required permissions.

### *4.1.4 mnx_vm_init*

```
long mnx_vm_init(VM_usr_t *vmu_addr)
```

This function initializes and starts a VM into the local node of the DRVS system.

This function must be called with root's privileges.

The is defined as:

```
struct VM_usr {
   int   vm_vmid;          /* The VM ID                                    */
   int   vm_flags;         /* VM's status flags                            */
   int   vm_nr_procs;      /* maximum number of processes for the VM       */
   int   vm_nr_tasks;      /* maximum number of tasks  for the VM          */
   int   vm_nr_sysprocs; /* maximum number of system processes for the VM */
   unsigned long int vm_nodes;   /* BITMAP of nodes where the VM runs      */
   char  vm_name[MAXVMNAME];     /* VM name                                */
};
typedef struct VM_usr VM_usr_t;
```

Return codes:

- OK:  The process VM has been initialized without errors.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLRANGE: Some `VM_usr_t` member has an invalid value.
- EMOLNAMETOOLONG: The VM name is too long.
- EMOLEXIST: Another VM with the same name is running.
- EMOLVMRUN: The VM is already running.
- EMOLNOMEM: The system has not enough memory to create the VM.
- EMOLNOTDIR: A problem trying to create the /proc/drvs/<vm_name> directory.
- EMOLBADF: A problem trying to create a file under the /proc/drvs/<vm_name>  directory.

#### *4.1.5 mnx_vm_end*

```
long mnx_  _end(long  vmid)
```

This function ends the specified VM and unbinds all its processes sending them a Linux SIGPIPE signal. It also de-registers the VM from the local node but no action to the remotes nodes is made.

This function must be called with root's privileges.

Return codes:

- OK: The process VM has been initialized without errors.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLNOTVMRUN: The VM is not running.

#### *4.1.6 mnx_getvminfo()*

```
long mnx_getvminfo(int vmid, VM_usr_t *vm_usr_ptr)
```

This function gets VM status and parameter information.

Return codes:

- OK: The process VM has been initialized without errors.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.

#### *4.1.7 mnx_bind()*

```
long mnx_bind(int vmid, int p_nr)
long mnx_lclbind(int vmid,int pid, int p_nr)
long mnx_rmtbind(int vmid, char name[], int endpoint, int nodeid)
long mnx_bkupbind(int vmid, int pid, int endpoint, int nodeid)
```

A Linux ordinary process cannot use communication APIs directly, it must be previously registered at the kernel to be considered as a process belonging to a specified PG using Since a process does not need to register by itself, it does not need to know which VM it belongs to. Another unregistered process with management privileges could register co-located processes by using to pass the process VMID and PID as arguments. These binding APIs return the process endpoint allocated to the process or an error code.

Every process bound to a VM will have a unique endpoint number (processes with allocated backup endpoints are exceptions).

An is an address used in M3-IPC APIs as source or destination of messages. M3-IPC kernel distinguishes among Local and Remote endpoints. Local endpoints are those allocated to processes running on the local node; and Remote endpoints are those allocated to processes running on remote nodes. Upon registration, a local process establishes a one to one relationship between the PID assigned by Linux and the endpoint within a VM (PID↔{VM, endpoint}). Afterwards, the process is enabled to use M3-IPC communication APIs addressing the other processes by their endpoints.

Two kinds of endpoint can be specified: and . Privileged endpoints will be usually allocated to server processes and non-privileged endpoints to client processes. Endpoint privileges establish capabilities which limit the communication APIs that can be used by the process and the endpoints a process can communicate with. They are granted by using API function.

The registers the caller into the VM with of M3-IPC kernel. It requests the M3-IPC kernel to assign it the process slot number. It returns the process endpoint or an error code (< ). The process that calls must have root's privileges.

The function registers a local Linux process with PID=pid into the VM with of M3-IPC kernel. It is intended to be used by a local process with root's privileges to bind a child process without those privileges. It requests the M3-IPC kernel to assign it the process slot number. It returns the process endpoint or an error code (< ). The process that calls must have root's privileges.

The rule to local bind is: if the the assigned , otherwise the kernel assigns a new process endpoint.

Every remote server process which will be referred to in communication APIs should be first registered in the local node. It must be done by using with the VM it belongs to ( ), its , and the node where it resides ( ) as arguments. Once a remote endpoint is registered by the local M3-IPC kernel, all local registered processes within the same VM are able to refer to that endpoint in communication APIs.

The function registers a remote process endpoint that runs on a remote node into the VM with of M3-IPC kernel. The remote node proxies must be registered, but message and data transfers only can be made to this endpoint if the node is in PROXY_CONNECTED state. The kernel assumes that a remote kernel in the specified , has assigned this endpoint to a Linux process running on it. It is intended to be used by a local process with root's privileges to bind a remote process.

Remote client binding is easier because M3-IPC can automatically bind remote unprivileged endpoints when their messages arrive to the local node.. If a message is received by Receiver Proxy with a source endpoint that correspond to a (unpriviledged endpoint) the endpoint is automatically bound as remote process running on the source node. There is no need to call to communicate with it.

The function registers a local process with PID= as a backup of a remote process with the same endpoint into the VM with of M3-IPC kernel. Every message sent by local processes to that endpoint will be sent to the remote (primary) process. The local process does not receive any message and cannot use any M3-IPC function until it will be promoted as a Primary endpoint. It has the same behavior as a endpoint registered as a Remote ( ). If a management application detects a fault in the Primary endpoint it could invoke on every node to stop communications to the failing Primary. Afterwards, it could call to restart communications, but sending messages to the former Backup endpoint. There can be several Backup endpoints in the same VM, each one registered on different nodes.

Threads are often used to increase parallelism and/or to facilitate programming. M3-IPC provides two ways of dealing with threads:

- With this approach, all of the process' threads can use M3-IPC communication APIs. To avoid race conditions, the application must use semaphores, mutexes or other synchronization method provided by Linux to synchronize the use of M3-IPC system calls because the same endpoint is shared by all threads.
- : With this approach, each bound thread allocates an endpoint within the same VM. There is no need to synchronize the use of M3-IPC system calls with the other bound threads.

Return codes:

- : If the return value is greater than it is the endpoint assigned to the process.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLRANGE: Some `VM_usr_t` member has an invalid value.
- EMOLVMNOTRUN: The specified VM is not running.
- EMOLNOVMNODE: The specified node does not run the specified VM.
- EMOLBUSY: The **p_nr** slot is busy.
- EMOLBADPROC: The specified PID is not running.
- EMOLNOPROXY: The node's proxies is not running.
- EMOLINVAL: The of is out of range (0≤vmid<NR_VMS)

### *4.1.8 mnx_wait4bind()*

```
int mnx_wait4bind()
```

Sometimes a non privileged process needs to wait that a privileged process binds it. The function mnx_wait4bind() blocks the caller until it is bound.

Return codes:

- : If the return value is greater than it is the endpoint assigned to the process.
- EMOLDRVSINIT: The DRVS was not initialized.

#### *4.1.9 mnx_unbind()*

```
long mnx_unbind(int vmid, int endpoint)
```

De-register a Linux process from the M3-IPC kernel and release allocated resources. It removes any pending message that the process wished to send to other process. It returns and error code to those processed that wished to send to the caller. It removes any pending notification.

Every registered local process that exits, by its own or killed by a signal, is automatically de-registered by M3-IPC kernel before exit. Therefore it is not necessary for a process to call to de-register by itself.

Return codes:

- OK: The process unbind without errors.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLVMNOTRUN: The specified VM is not running.
- EMOLNOTBIND: The specified endpoint is not binded to the specified VM.
- EMOLENDPOINT: Bad endpoint.
- EMOLINVAL: The of is out of range (0≤vmid<NR_VMS)

#### *4.1.10 mnx_getep()*

```
long mnx_getep(int pid)
```

Get the endpoint of a registered local Linux process. The process must belong to the same VM as the caller.

Return codes:

- **endpoint**: If the return value is greater than 0, it is the endpoint requested process.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLRANGE: The is out of range (0 < PID ≤ MAXPID).
- EMOLNOTBIND: The requested process is unregistered.
- EMOLBADVMID: The requested process is in other VM than the caller.
- EMOLPROXY: The process is a proxy.

#### *4.1.11 mnx_getprocinfo()*

```
long mmx_getprocinfo(int vmid, int p_nr, struct proc_usr *proc_usr_ptr)
```

Get the status and parameter information about a process in a VM.

Return codes:

- OK: The process unbind without errors.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLINVAL: The of is out of range (0≤vmid<NR_VMS)
- EMOLVMNOTRUN: The specified VM is not running.
- EMOLBADPROC: Bad process number.

#### *4.1.12 mnx_vm_dump()*

```
long mnx_vm_dump(void)
```

Logs the status and some attributes of all VMs to the Linux circular buffer (dmesg).

Sample output:

```
VMID      FLAGS    NAME
0         0        VM0
1         0        VM1
2         0        VM2
3         0        VM3
4         0        VM4
```

Return codes:

- OK: The process unbind without errors.
- EMOLDRVSINIT: The DRVS was not initialized.

#### *4.1.13 mnx_proc_dump()*

```
long mnx_proc_dump(int vmid)
```

This function logs the status of all processes of a specified to the Linux circular buffer (dmesg).

Return codes:

- OK: The process are dumped without errors.
- EMOLBADVMID: The of is out of range (0≤vmid<NR_VMS)
- EMOLVMNOTRUN: The specified VM is not running.

Sample output:

```
VMID      NR      ENDP    LPID    NODE    FLAGS   GETF    SNDT   WITM
0         1       71079   1945    0       0       27342   27342  27342
0         11      71089   1946    0       4       27342   71079  27342
0         12      71090   1947    0       4       27342   71079  27342
0         13      71091   1948    0       4       27342   71079  27342
0         14      71092   1949    0       4       27342   71079  27342
0         15      71093   1950    0       4       27342   71079  27342
```

#### *4.1.14 mnx_setpriv()*

```
long mnx_setpriv(int vmid, int proc_ep, priv_usr_t *u_priv)
```

This function sets the privileges of a process. This function must be called with root's privileges.

The is defined as:

```
struct priv_usr {
  sys_id_t s_id;               /* index of this system structure */
  int   s_warn;                /* process to warn when the process exit/fork */
  int   s_level;               /* privilege level            */
  short s_trap_mask;           /* allowed system call traps */
  sys_map_t s_ipc_from;        /* allowed callers to receive from */
  sys_map_t s_ipc_to;          /* allowed destination processes */
  long s_call_mask;            /* allowed kernel calls */
  moltimer_t s_alarm_timer;    /* synchronous alarm timer */
};
typedef struct priv_usr priv_usr_t;
```

Return codes:

- OK: The process unbind without errors.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLBADPROC: Bad process number.
- EMOLVMNOTRUN: The specified VM is not running.

#### *4.1.15 mnx_getpriv()*

```
long mnx_getpriv(int vmid, int proc_ep, priv_t *u_priv)
```

It gets the privileges of a process. This function must be called with root's privileges.

Return codes:

- OK: The process unbind without errors.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.

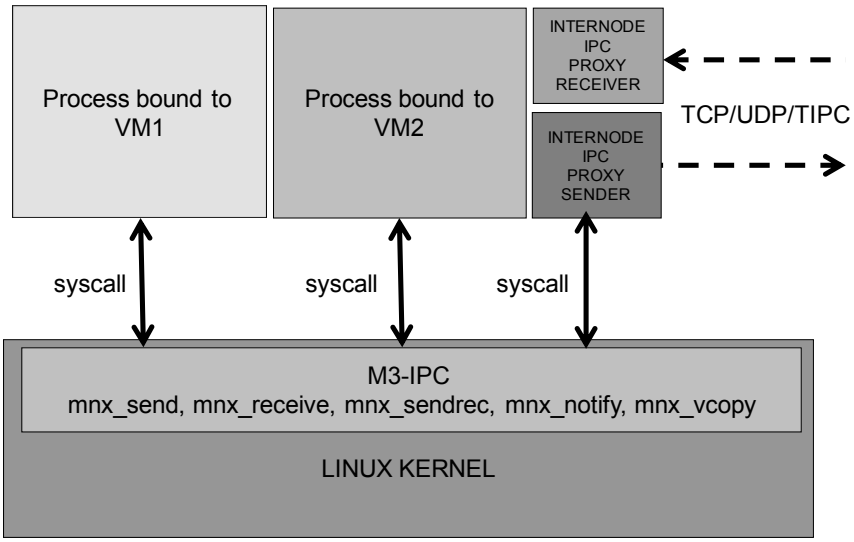
- EMOLBADPROC: Bad process number.
- EMOLDSTDIED: The process is de-registered.
- EMOLVMNOTRUN: The specified VM is not running.

# 5 Proxies and Remote Nodes

M3-IPC allows sending messages and data blocks between M3-IPC registered process in the same computing node or in different nodes in a transparent way.

The masking of remote transfers is done by proxies. There is one sender and one receiver proxy for each remote node. A node it says to be active if it has its proxies registered to the M3-IPC kernel through the , but the message and data transfers can only be made if the node status is PROXY_CONNECTED that it is set when the local and remote node has established sender and receiver connections.

**Note: The proxies supplied by the standard M3-IPC distribution are program samples. They use TCP, UDP, TIPC as protocols for remote transfers, but anyone can change those programs and protocols for other equivalent ones.**




## 5.1 Overview

The following auxiliary functions are available to programmers:

Registers a remote node sender's and receiver's proxies.
Change the status of nodes' proxies.
De-registers a remote node sender's and receiver's proxies.
Gets status and parameter information of a pair of proxies.
Relate which VM's endpoint could reside on a remote node.
Remove a node from the VM.
Gets status and parameter information of a node.

### *5.1.1 mnx_proxies_bind()*

```
int mnx_proxies_bind(name, pxid, int spid, int rpid)
```

This function registers a remote node sender's and receiver's proxies. The initial node status is PROXY_NOT_CONNECTED, therefore any communication can be done until it is connected. This function must be called with root's privileges.

This function must be called with root's privileges.

is the text that identifies the proxy pair.

: is an identifier for the proxy pair.

: the PID of the process or thread that will acts as the Sender Proxy.

: the PID of the process or thread that will acts as the Receiver Proxy.

Return codes:

- OK: The process unbind without errors.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLBADNODEID: Invalid number.
- EMOLNODEBUSY: The node already has two proxies registered.
- EMOLBADPID: The supplied PID process is not running.
- EMOLNOTDIR: A problem trying to create the /proc/drvs/<node_name> directory.
- EMOLBADF: A problem trying to create a file under the /proc/drvs/<node_name> directory.

### *5.1.2 mnx_proxy_conn()*

```
int mnx_proxy_conn(int pxid, int status)
```

This function allows to sets the proxy pair status as:

- CONNECT_SPROXY
- CONNECT_RPROXY
- DISCONNECT_SPROXY
- DISCONNECT_RPROXY

Informing M3-IPC kernel about proxies communications status. This function must be called with root's privileges.

If the proxies status is set to CONNECT_SPROXY and CONNECT_RPROXY, data or message transfer to any remote endpoint of every VM residing in that node could be made.

If the proxies status is set to DISCONNECT_SPROXY and DISCONNECT_RPROXY, every data or message transfer that involves any remote endpoint of every VM residing in that node ends returning an error.

Return codes:

- OK: The process unbind without errors.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLBADNODEID: Invalid number.
- EMOLINVAL: The status is not PROXY_NOT_CONNECTED or PROXY_CONNECTED.
- EMOLNODEFREE: The node is free.
- EMOLNOPROXY: The proxies of the node are not running or registered.

### *5.1.3 mnx_proxy_unbind()*

```
int mnx_proxy_unbind(int pxid)
```

It function de-register a pair of proxies. Every data or message transfer that involves any remote endpoint of every VM residing on a node reached by this proxy pair will return an error code.

This function must be called with root's privileges.

Every registered proxy that exits by its own or killed by a signal is de-registered by the kernel before exits. It also signals with SIGPIPE the other peer proxy. Therefore it is not necessary for a process to call to de-register itself.

Return codes:

- OK: The process unbind without errors.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLBADNODEID: Invalid number.

### *5.1.4 mnx_getproxyinfo*

```
long mnx_getproxyinfo(int pxid, proc_usr_t *sproc_usr, proc_usr_t *rproc_usr)
```

This function gets status and configuration parameters of the processes registered as a proxy pair.

This function must be called with root's privileges.

The is defined as:

```
/* ================================*/
/* PROCESS DESCRIPTOR              */
/* ================================*/
struct proc_usr {

  int p_nr;                          /* process number                              */
  int p_endpoint;                    /* process endpoint                            */
  int p_vmid;                        /* process VMID                                */
  volatile unsigned long p_rts_flags; /* process is runnable only if zero           */
  int p_lpid;                        /* local LINUX PID                             */
  int p_nodeid;                      /* Node ID where the process PRIMARY replica is running  */
  unsigned long p_nodemap;           /* bitmap of nodes where replicas of this process exists */
  unsigned long  p_misc_flags;       /* miselaneous flags                                     */
  int p_getfrom;                     /* from whom does process want to receive?     */
  int p_sendto;                      /* to whom does process want to send?          */
  int p_waitmigr;                    /* waiting the migration of a process          */
  int p_proxy;                       /* the descriptor is enqueued on a proxy       */
  unsigned long p_lclsent;           /* counter of LOCAL messages sent     */
  unsigned long p_rmtsent;           /* counter of REMOTE messages sent    */
  char p_name[MAXPROCNAME];
#ifdef MOL_USERSPACE
         cpu_set_t p_cpumask;
#else
         cpumask_t p_cpumask;
#endif
};
typedef struct proc_usr proc_usr_t;
```

### *5.1.5 mnx_add_node()*

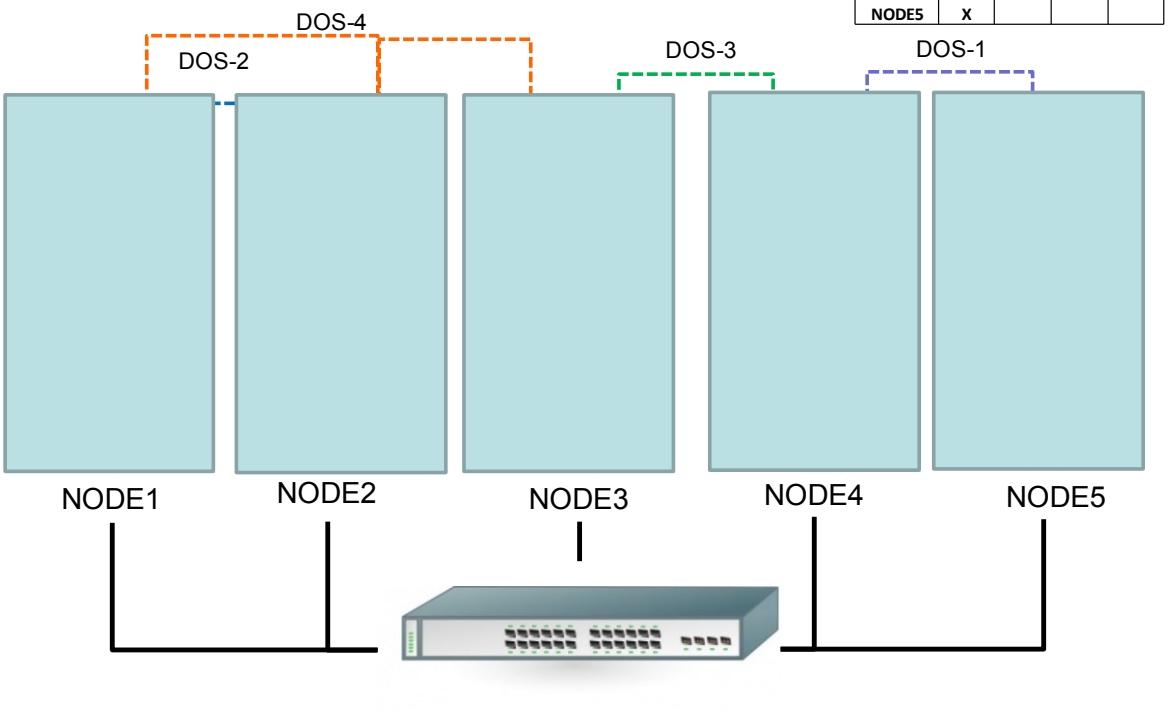
```
int mnx_add_node(int vmid, int nodeid)
```

This function registers a remote node to a VM telling that some process of that VM could resides in that node. The node's proxies must be registered before adding the node to a VM. This function must be called with root's privileges.

Return codes:

- OK: The process unbind without errors.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLBADNODEID: Invalid number.
- EMOLNOPROXY: The node's proxies have not been registered.
- EMOLVMNOTRUN: The VM is not running (init).
- EMOLVMNODE: The node is already included into the VM list of nodes.

| | VM1 | VM2 | VM3 | VM4 |
|---|---|---|---|---|
| NODE1 | | X | | X |
| NODE2 | | X | | X |
| NODE3 | | | X | X |
| NODE4 | X | | X | |
| NODE5 | X | | | |




### *5.1.6 mnx_ getnodeinfo()*

```
long mnx_getnodeinfo(int nodeid, node_usr_t *node_usr_ptr)
```

This function gets status and configuration parameters of a node.

This function must be called with root's privileges.

The is defined as:

```
struct node_usr {
        int                     n_nodeid;
        volatile unsigned long  n_flags;
        int                     n_proxies;/* proxy pair ID for this node          */
        unsigned long int       n_vms;    /* VM BITMAP                            */
        struct timespec  n_stimestamp;    /* timestamp of the last sent  msg      */
        struct timespec  n_rtimestamp;    /* timestamp of the last received msg   */
        char                    n_name[MAXNODENAME];
}; typedef struct node_usr node_usr_t;
```

The field is a bitmap telling which VMs resides on the node.

Return codes:

- OK: The process unbind without errors.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLBADNODEID: Invalid number.

### *5.1.7 mnx_del_node()*

```
int mnx_del_node(int vmid, int nodeid)
```

This function de-registers a remote node from a VM. Every pending communication with a VM's endpoint that resides on that node is finished returning an error.

This function must be called with root's privileges.

Return codes:

- OK: The process unbind without errors.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLBADNODEID: Invalid number.
- EMOLBADVMID: The of is out of range (0≤vmid<NR_VMS).
- EMOLNOPROXY: The node's proxies have not been registered.
- EMOLVMNOTRUN: The VM is not running (init).
- EMOLVMNODE: The node is NOT included into the VM list of nodes.

### *5.1.8 mnx_node_up()*

```
int mnx_node_up(char name[], int nodeid, int pxid)
```

This function links a node to the proxy pair. Every local data or message transfer to that node will be made through the mentioned proxy pair.

This function must be called with root's privileges.

Return codes:

- OK: The process unbind without errors.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLBADNODEID: Invalid number.
- EMOLPROXYFREE: The proxy pair is unused.

### *5.1.9 mnx_node_down()*

```
int mnx_node_down(int nodeid)
```

This function unlinks a node from a proxy pair.

This function must be called with root's privileges.

Return codes:

- OK: The process unbind without errors.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLBADNODEID: Invalid number.
- EMOLPROXYFREE: The proxy pair is unused.

### *5.1.10 mnx_get2rmt()*

```
int mnx_get2rmt(cmd_t *header, proxy_payload_t *payload)
```

This function can only be used by a registered Sender Proxy. It is used to get a data block or a message from M3-IPC kernel to be sent to one of the remote nodes this proxy is linked to. It must be called with root's privileges.

In the header structure there is information for the Sender Proxy about the kind of , and its payload.

```
struct cmd_s {
        int     c_cmd;
        int     c_vmid;         /* VM ID                                */
        int     c_src;          /* source endpoint                      */
        int     c_dst;          /* destination endpoint                 */
        int     c_snode;        /* source node                          */
        int     c_dnode;        /* destination node                     */
        int     c_rcode;        /* return code                          */
        int     c_len;          /* payload len                          */
struct timespec c_timestamp;    /* timestamp                            */
        vcopy_t c_vcopy;        /* struct used only for remote vcopy    */
};
typedef struct cmd_s cmd_t;
```

The field coud be:

```
CMD_NONE                NO COMMAND
CMD_SEND_MSG,           Send a message to a process
CMD_NTFY_MSG,           Send a NOTIFY message to remote process
CMD_SNDREC_MSG,         Send a message to a process and wait for reply
CMD_REPLY_MSG,          Send a REPLY message to a process
CMD_COPYIN_DATA,        Request to copy data to remote process
CMD_COPYOUT_RQST,       The remote process send to local process the data requested
CMD_COPYLCL_RQST,
CMD_COPYRMT_RQST,       REQUESTER to SENDER to copy data out to RECEIVER
CMD_COPYIN_RQST,        SENDER to RECEIVER
CMD_HELL0,              HELLO COMMAND used by proxies
CMD_SHUTDOWN,           Exit the waiting loop with error
```

Useful field for the Sender proxy will be:

- The VMID the message or data belongs to. The Sender proxy could use this information to apply QoS or filtering based on the VMID.
- : The Sender proxy could use this information to apply QoS or filtering based on source and/or destination endpoint.
- : The Sender proxy could use this information to apply QoS or filtering or protocol selection, routing, etc. based on the destination node.
- : Needed by the Sender proxy for message transfer.
- : The Sender proxy could use this information to apply QoS, filtering, delaying, ordering, etc. based on this field.

The payload could be a message, a block of data or none.

```
typedef union {
        message pay_msg;                /* Minix message          */
        char    pay_data[MAXCOPYBUF];   /* buffer space to copy data      */
}proxy_payload_t;
```

### *5.1.11 mnx_put2lcl()*

```
int mnx_put2lcl(cmd_t *header, proxy_payload_t *payload)
```

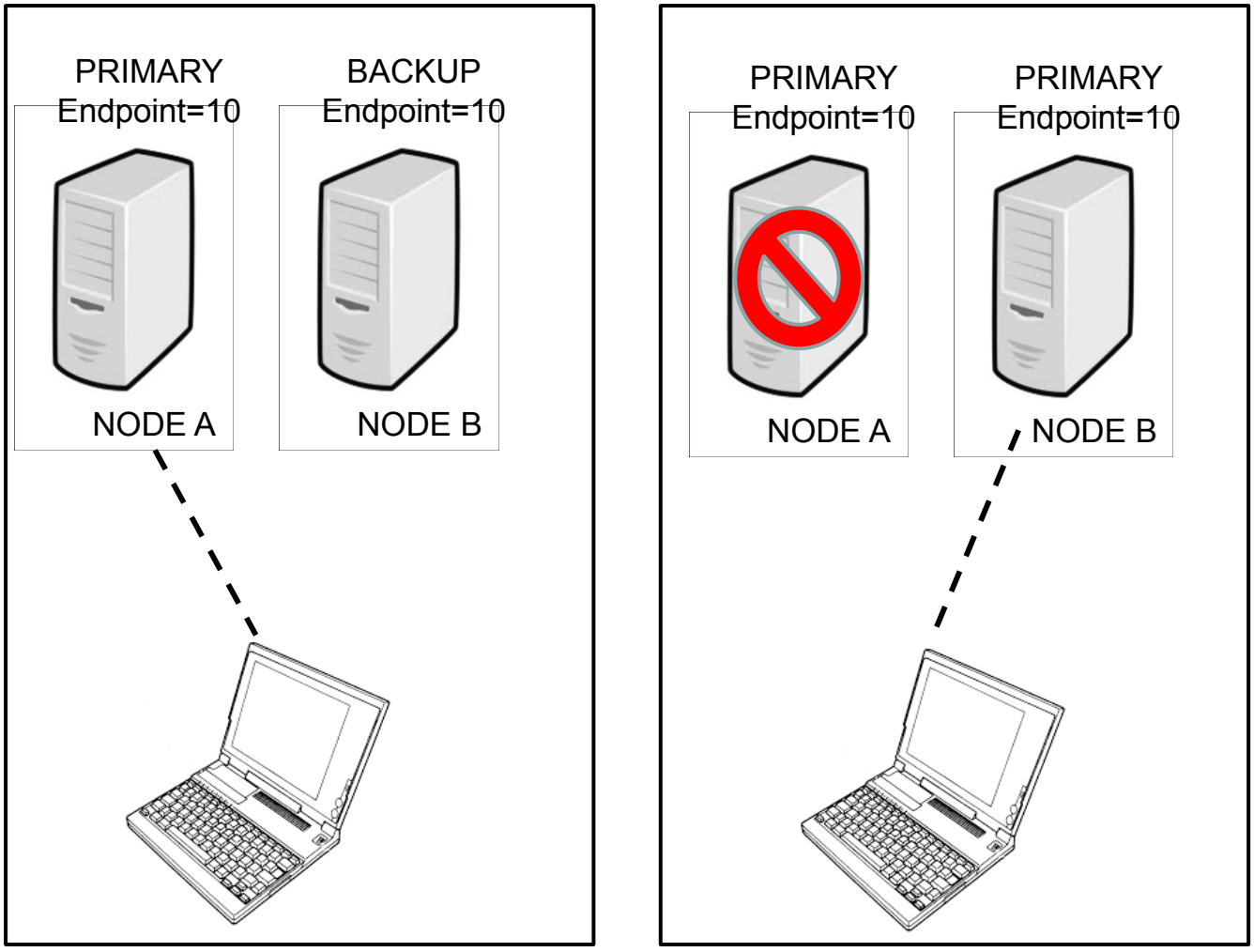
This function can only be used by a registered Receiver Proxy. It is used to put a data block or send a message to a local process. It must be called with root's privileges.

In the header structure is used by the local M3-IPC to correct deliver. It should not be modified by the proxy, but it can change the delivery order or filter based of some of the header fields.

# 6 Fault Tolerance and Process Migration

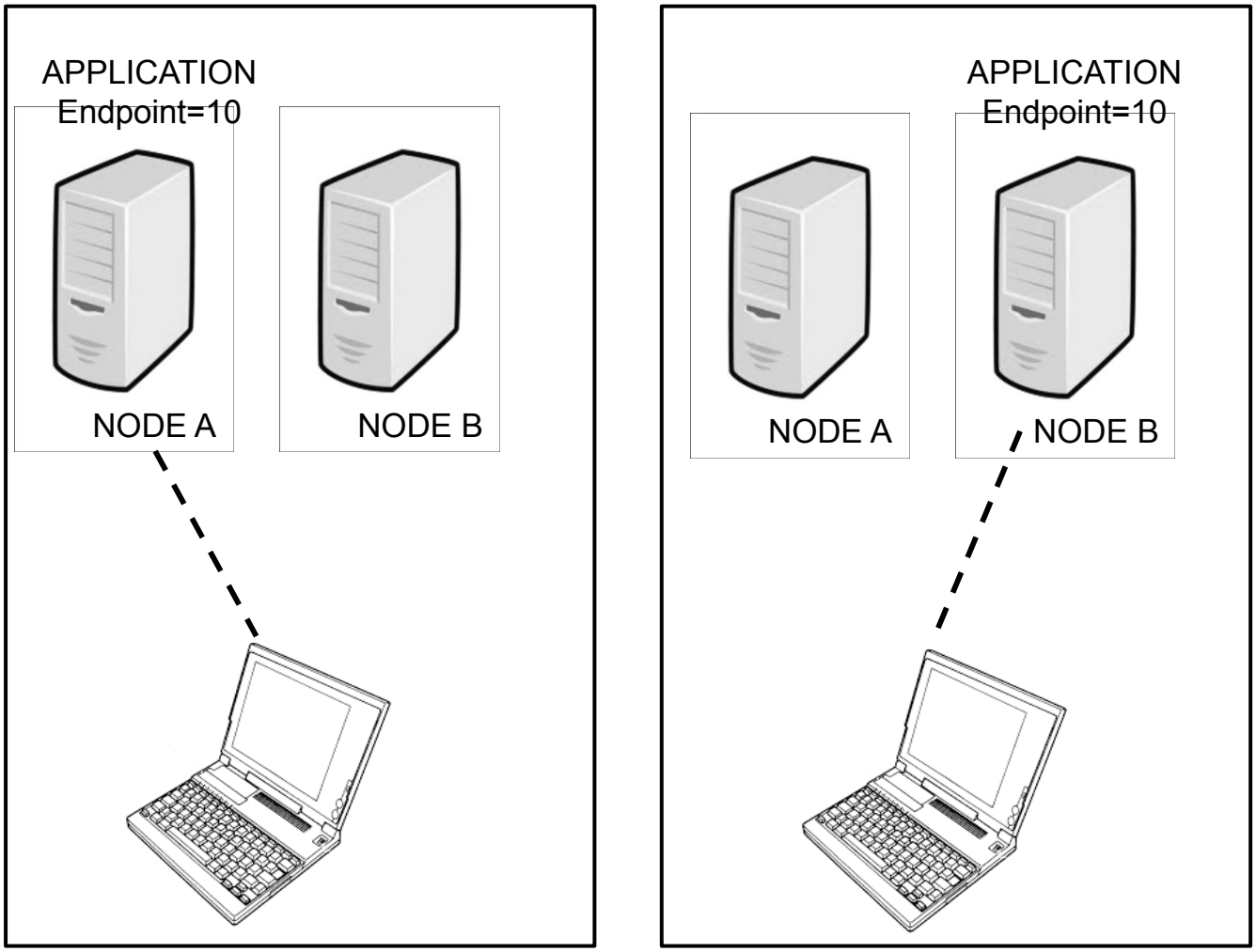
Some fault tolerant applications use the Primary-Backup approach. M3-IPC supports IPC redirection when the Primary fails and one of the Backup processes becomes the new Primary. Data and Messages will be redirected by all nodes to the node of the new Primary.

M3-IPC supports message redirection on node switching when a Primary process (on node A) is replaced by a Backup process (on node B). The local M3-IPC kernel automatically unlinks the endpoint from the previous primary process (on node A) and links it to the new primary process (on node B). M3-IPC supports that processes belonging to the same VM, but residing on different nodes of the cluster, can have the same endpoint numbers. Only one of that processes will be the primary endpoint and the others will be backup endpoints. If the primary process crashes, a high level application that detects the failure could designate one of the backup endpoints as the new primary. The switching of the primary process should be masked for those processes that were communicating with the former primary. M3-IPC kernels on the surviving nodes automatically unlink the endpoint from its previous node and then link it again, but to the new node, including if it is the own node.



A local process can be registered as a Backup of a remote Primary process by using
with the same endpoint in the same VM as the Primary. All messages sent to that endpoint are sent to the Primary process on the remote node. The Backup process could not communicate with any other process using M3-IPC until it is converted into the Primary process. A privileged management application must use followed by to convert a Backup process into a Primary one. Eventually, if a Backup process terminates its process descriptor is converted automatically into a Remote process descriptor referring to the location of the Primary endpoint. Message replication among Primary and Backup endpoints does not concern M3-IPC. For co-located processes a Backup endpoint has the same behavior as a Remote endpoint. Therefore, every message sent by a local process addressed to that endpoint will be send to the Primary process.

M3-IPC can keep communications in operational state and can automatically redirect messages on live process migration. M3-IPC APIs must transparently support timeframes in which a process resides in a source node, timeframes in which the process is migrating, and timeframes in which the process resides in the destination node.



Before a process begins to migrate, a management application must call At that moment, all messages addressed to the migrating endpoint will be queued up. Once the process has successfully migrated, the management application must call All queued messages will be sent to the process on its new location. If migration fails, the management application must call then the process can resume its execution as if nothing had happened by receiving the queued messages.

### *6.1.1 mnx_migr_start()*

```
int mnx_migr_start(int vmid, int endpoint)
```

This function stops all messages and data transfers addressed to the mentioned on the VM with .

This function must be called with root's privileges.

Return codes:

- OK: Return without errors.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLBADVMID: The of is out of range (0≤vmid<NR_VMS).
- EMOLRANGE: invalid endpoint.
- EMOLDSTDIED: The mentioned endpoint died.
- EMOLONCOPY: The mentioned endpoint is making a copy data and cannot be interrupted.
- EMOLBUSY: The mentioned endpoint is making a remote IPC and cannot be interrupted.
- EMOLENDPOINT: Bad endpoint.
- EMOLMIGRATE: The endpoint is already migrating.
- EMOLGRPLEADER: The endpoint is not registered by the main thread group leader.

### *6.1.2 mnx_migr_commit()*

```
int mnx_migr_commit(int pid, int vmid, int endpoint, int node)
```

This function restarts all messages and data transfers addressed to the mentioned redirecting to the new . If the new node for endpoint is the local node, the argument is the PID of the process on local node.

This function must be called with root's privileges.

Return codes:

- OK: Return without errors.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.

- EMOLBADVMID: The of is out of range (0≤vmid<NR_VMS).
- EMOLRANGE: invalid endpoint.
- EMOLNOPROXY: There is not a registered proxy pair for the mentioned node.
- EMOLVMNODE: The mentioned node is not valid for the VM.
- EMOLBADPID: The PID is invalid.

### *6.1.3 mnx_migr_rollback()*

```
int mnx_migr_rollback(int vmid, int endpoint)
```

This function restarts all messages and data transfers addressed to the mentioned to the same node.

This function must be called with root's privileges.

Return codes:

- OK: Return without errors.
- (-EPERM): The caller's **euid** not have the required permissions.
- EMOLDRVSINIT: The DRVS was not initialized.
- EMOLBADVMID: The of is out of range (0≤vmid<NR_VMS).
- EMOLRANGE: invalid endpoint.
- EMOLONCOPY: The mentioned endpoint is making a copy data and cannot be interrupted.
- EMOLBUSY: The mentioned endpoint is making a remote IPC and cannot be interrupted.
- EMOLENDPOINT: Bad endpoint.
- EMOLMIGRATE: The endpoint is already migrating.
- EMOLGRPLEADER: The endpoint is not registered by the main thread group leader.

***Backup endpoint use case : sample pseudocode***

| Primary Server on NODE A | Backup Server on NODE B | Client on NODE C |
|---|---|---|
| Before Failure and switching | | |
| `svr_ep = mnx_bind(vmid,SVR_NR);` | `bk_ep = mnx_bkupbind(vmid,getpid(),SVR_NR ,node_A);` | `clt_ep = mnx_bind(vmid,CLT_NR);`<br><br>`svr_ep = mnx_rmtbind(vmid,"SERVER", SVR_NR,node_A)` |
| After Failure and switching | | |
| | `mnx_migr_start(vmid, bk_ep);`<br><br>`mnx_migr_commit(getpid(), vmid, bk_ep, node_B);` | `mnx_migr_start(vmid, svr_ep);`<br><br>`mnx_migr_commit( (-1), vmid, srv_ep, node_B);` |

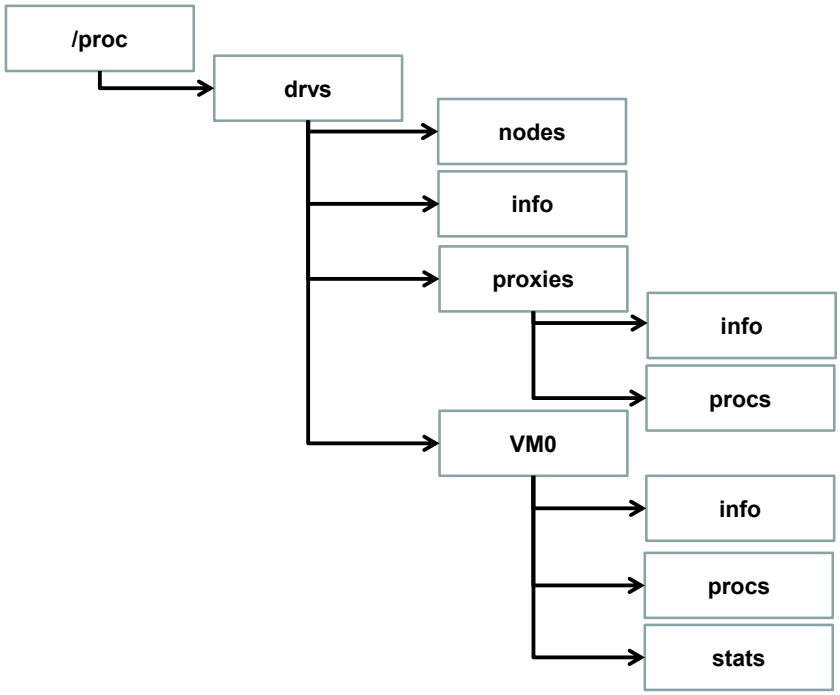
***Process Migration use case: sample pseudocode***

| Primary Server on NODE A | Backup Server on NODE B | Client on NODE C |
|---|---|---|
| Before Migration | | |
| svr_ep = mnx_bind(vmid,SVR_NR);<br><br>mnx_migr_start(vmid, svr_ep); | mnx_migr_start(vmid, bk_ep); | clt_ep = mnx_bind(vmid,CLT_NR);<br><br>svr_ep = mnx_rmtbind(vmid,"SERVER", SVR_NR,node_A)<br><br>mnx_migr_start(vmid, svr_ep); |
| After successful Migration | | |
| | mnx_migr_commit(getpid(), vmid, bk ep, node B); | mnx_migr_commit( (-1), vmid, srv ep, node B); |
| After failed Migration | | |
| mnx_migr_rollback(vmid, svr_ep); | mnx_migr_rollback(vmid, svr_ep); | mnx_migr_rollback(vmid, svr_ep); |

# 7 Integration with Linux /proc filesystem

M3-IPC kernel is integrated into the Linux kernel and with its /proc filesystem. Main M3-IPC abstractions are represented by directories and files in /proc.

The M3-IPC directories have the following structure.




The directory is created by It creates the and files.

```
root@node0:/proc/drvs# ls -l
total 0
-r--r--r-- 1 root root 0 Apr 27 03:30 info
-r--r--r-- 1 root root 0 Apr 27 03:30 nodes
dr-xr-xr-x 2 root root 0 Apr 27 03:30 proxies
dr-xr-xr-x 2 root root 0 Apr 27 03:30 VM0
```

The file shows the parameter and status of the DRVS.

```
root@node0:/proc/drvs# cat info
nodeid=0
nr_vms=32
nr_nodes=32
max_nr_procs=189
max_nr_tasks=67
max_sys_procs=96
```

```
max_copy_buf=32768
max_copy_len=524288
dbglvl=0
version=2.1
sizeof(proc)=340
sizeof(proc) aligned=512
sizeof(vm)=96
sizeof(node)=80
```

The file shows the parameter and status of the nodes and their proxies PIDs, which VMs resides on those nodes and other status flags.

```
root@node0:/proc/drvs# cat nodes
Node Flags Proxies 10987654321098765432109876543210 Name
   0     6      -1 -------------------------------X node0
   1     2       1 -------------------------------X node1
   2     2       2 -------------------------------- node2
```

The function binds a pair of processes to be used as the Sender and Receiver processes to the M3-IPC kernel.

```
root@node0:/proc/drvs/proxies# more procs
PXID -Type- -lpid- -flag- -misc- -getf- -sendt -wmig- name
   1 sender   2292      0      1  27342  27342  27342 tproxy
   1 recver   2293      0      1  27342  27342  27342 tproxy
   2 sender   2290      0      1  27342  27342  27342 tproxy
   2 recver   2291      0      1  27342  27342  27342 tproxy
```

The function associates a pair of proxies to a node.

```
root@node0:/proc/drvs/proxies# more info
Proxies Flags Sender Receiver --Proxies_Name- 10987654321098765432109876543210
      1     1   2292     2293           node1 -----------------------------X-
      2     1   2290     2291           node2 -----------------------------X--
```

The creates a directory with the VM name and the and files under it.

```
root@node0:/proc/drvs/VM0# ls -l
total 0
-r--r--r-- 1 root root 0 Apr 27 03:38 info
-r--r--r-- 1 root root 0 Apr 27 03:38 procs
-r--r--r-- 1 root root 0 Apr 27 03:38 stats
```

The file shows the parameter and status of the VM, and in which nodes the VM resides.

```
root@node0:/proc/drvs/VM0# cat info
vmid=0
flags=0
nr_procs=189
nr_tasks=67
nr_sysprocs=96
```

```
nr_nodes=32
vm_nodes=3
vm_name=VM0
nodes 33222222222211111111110000000000
      10987654321098765432109876543210
      -------------------------------XX
cpumask=ff
```

The file shows the parameter and status of the process endpoints of the VM and in which nodes they reside. If the endpoint resides in the local node, its PID is shown.

```
root@node0:/proc/drvs/VM0# more procs
VMID p_nr -endp- -lpid- node -flag- -getf- -sndt- -wmig- -prxy- name
   0    1      1     -1    1   1000  27342  27342  27342  27342 SoyRemoto
```

The register a node and creates a directory with the name of the node and the info and stats files under it.

The file shows the parameter and status of the node, the proxies PIDs and which VMs resides on that node.

```
root@node0:/proc/drvs/node1# more info
Node Flags Sender Receiver 10987654321098765432109876543210 Name
   1     0   5260     5261 -------------------------------X node1
```

The file shows status of the node´s proxies and statistics of sent and received remote commands. More communication statistics data could be captured in the proxies.

```
root@node0:/proc/drvs/VM0# more stats
VMID p_nr -endp- -lpid- node --lsnt-- --rsnt--
   0    1      1     -1    1        0        0
```

Once the proxies end, VM ends, or DRVS ends, the related directories and files are removed.