VMMC Communication Model

Windows NT User's Guide
and
API Reference

Version 2.0

The Shrimp Project

Department of Computer Science
Princeton University

February 1999

About this Document

Welcome to VMMC! The goals of this document are threefold:

1. An introduction to the VMMC model.

2. How to use and manage VMMC on a Windows NT cluster.

3. A description of the VMMC programming model as well as a programmer's reference for the VMMC API.

Users of VMMC are encouraged to join the email list used for VMMC announcements. Join the list to keep up with the latest VMMC developments. To join the email list, send a message to majordomo@cs.princeton.edu with "subscribe vmmc-announce youremail@yourcompany.com" in the body of the message. Email traffic is minimal.

We welcome input about functionality, bugs, and possible extensions to the API. Please send your comments or questions to vmmc-help@cs.princeton.edu. For more information on The SHRIMP Project (including technical papers) please visit our web site at http://www.cs.princeton.edu/shrimp.

Contents

Introduction

The SHRIMP Project at Princeton University studies ways to integrate commodity desktop computers, such as PCs and workstations, into inexpensive, high-performance multicomputers. The goal is to build an inexpensive system from off-the-shelf components with minimal custom-designed hardware. Ideally, such a system would offer a performance competitive with, or better than, the performance of specially designed multicomputers for both message passing and shared-memory programming models.

During the course of our research we found that the network interfaces (NI) of existing multi-computers and workstation networks introduce large software overheads for communication. The main reason for this overhead is that these network interfaces require a significant number of instructions at the operating system and user levels to provide protection and buffer management. Motivated by this fact, we designed and built two custom network interfaces (SHRIMP[1]-I and SHRIMP-II) for low-latency, high-bandwidth user-to-user communication. These network interfaces implement our model of user-level communication called VMMC (virtual memory-mapped communication) which provides direct data transfer between the sender's and receiver's virtual address spaces. Additionally, this model eliminates operating system involvement in communication, provides full protection, supports user-level buffer management, zero-copy protocols, and minimizes software communication overhead.

We also wanted VMMC to enable the creation of libraries implementing a variety of new and old APIs for parallel and distributed programming. Examples of such libraries include message-passing APIs like PVM, NX/2, distributed shared-memory, and client-server APIs like RPC and stream sockets.

The introduction of commodity programmable network interfaces enabled us to transfer most of the functionality of our custom network interface to off-the-shelf NI hardware. We achieved this by implementing support for our user-level communication model in the NI firmware. The firmware together with a device driver and a user-level library implement the VMMC communication model.

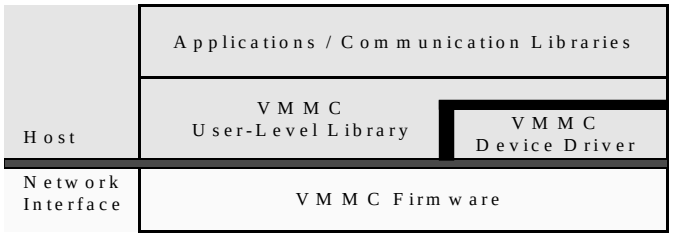


Figure 1. The components of VMMC

[1] SHRIMP is an acronym for **S**calable **H**igh-performance **R**eally **I**nexpensive **M**ulti-**P**rocessor

Changes in Version 2.0

Version 2.0 of VMMC includes many improvements to the original VMMC release. Briefly, the improvements are:

- The API functions are renamed to begin with "`vmmc_`".
- The VMMC data types were changed and renamed for consistency.
- A transfer redirection mechanism was added to enable true zero-copy protocols.
- The API now supports remote data fetch.
- The network interface firmware was extended to support reliable data transfer using link-level retransmission.

Administrator's Guide

This section describes the VMMC system file layout on each cluster node, VMMC cluster services, and how to start and stop VMMC. To run the commands described in this section the administrator must log in as VMMC. Note that the VMMC account has administrative privileges.

1. VMMC System Root Directory

The VMMC system installation sets up a VMMC system root directory on each cluster node. The VMMC root is `%SystemRoot%\vmmc` by default. On most Windows NT, the `%SystemRoot%` is `C:\WinNT`. The VMMC system root directory contains all necessary files for running VMMC services. The directory structure is as follows:

`%SystemRoot%\vmmc`:

driver\	contains `vmmcdrv.sys` (VMMC device driver)
server\	contains `vmmcsvr.exe` (VMMC Cluster Server)
bin\	contains utilities and the firmware loader `Mload.exe`, `MCPStart.bat`, `MCPStop.bat`
mcp\	contains Myrinet firmware code: mcp4.dat
logs\	system log files: server.log and driver.log
vmmc-hosts.txt	a text file that lists all VMMC nodes in the cluster

During the VMMC system software installation the VMMC root directory is exported as a network share with a share name `VMMC`. This is done to simplify the remote administration of the VMMC cluster. On a Windows PC, the administrator can update the VMMC system files or check the VMMC logs by simply accessing a node's exported VMMC folder. For example,

```
NET use \\node1\vmmc vmmc /user:vmmc    # login as vmmc
TYPE \\node1\vmmc\logs\server.log       # view server log file
```

2. VMMC System Account and Network Share

During VMMC system installation, a local account **`vmmc`** is created on each cluster node. We have chosen **vmmc** as the password for this account. An administrator can modify the password using Windows NT's User Manager. However, the administrator must make sure that the `VMMCSVR` service can still run under the local vmmc account. The Service Control Manager (invoked from the Control Panel) can be used for this purpose.

The home directory for the vmmc account is `%SystemRoot%\vmmc`. If UWIN is installed on the cluster node, the **.rhosts** file must be located in this directory in order for the rsh service to work properly.

3. Starting and Stopping VMMC

The low-level components of VMMC include the device driver `VMMCDRV` and the Myrinet firmware mcp4.dat. To enable VMMC, the administrator must perform the following two tasks in order: (1) start the VMMC device driver and, (2) load the Myrinet firmware.

To start the VMMC device driver, the administrator can invoke the NET command from a command shell (DOS shell) as follows:

```
NET start   VMMCDRV
```

After the driver is successfully started, the administrator can load the VMMC firmware into the Myrinet network interface, using the MPCSTART script. The MCPSTART script is located in the **bin**/ folder under VMMC system directory. In the same folder, there exists an MCPSTOP script that the administrator can use to disable the Myrinet network interface.

```
C:\> CD %SystemRoot%\VMMC
C:\WINNT\VMMC> CD bin
C:\WINNT\VMMC\bin> MCPStart.BAT
```

4. VMMC Cluster Service and Utilities

Once the VMMC system is successfully started on each cluster node, the user can run VMMC programs using Myrinet. An integral part of VMMC is its API for remote process creation. This API includes `vmmc_Spawn()`, `vmmc_Parent()`, and others. The remote process creation is implemented by the VMMC Cluster Service, `VMMCSVR`. The VMMCSVR also provides basic support for process management and output logging. An instance of `VMMCSVR` runs as a Windows NT service on each VMMC Cluster node. It is automatically started during the system boot time.

5. Enabling Interactive Jobs

The user must log on to each VMMC cluster node using the **`vmmc`** account, before he or she can run any interactive jobs. Otherwise, processes created by `VMMCSVR` will not be visible on the NT desktop. There is a way to enable automatic logging into a local account on a Windows NT workstation. The administrator should consult Microsoft Developer's Network documentation for this feature.

6. System Log Files

On each VMMC cluster node, the VMMC system produces two log files, server.log and driver.log. Both reside in `%SystemRoot%\vmmc\logs` directory. The server.log file contains information and error logs from VMMCSVR, mostly regarding session and process management. The driver.log file contains logs for the VMMCDRV device driver and the Myrinet network interface.

User's Guide

In this section we give an overview of the components involved in the Windows NT implementation of VMMC. We also describe how to run VMMC applications.

1. Installing the VMMC SDK

The VMMC SDK is distributed in zip file a named `vmmc_sdk_2_0.zip`. The zip contains the following directory structure:

- `include:` header files needed to compile VMMC programs
- `example:` sample vmmc test programs
- `example\bin:` pre-compiled test programs
- `lib:` user-level library `vmmc.lib` and the debug version `vmmcd.lib`
- `utils:` various utilities including `CFGVMMC.exe`

2. VMMC Session Creation and Deletion

Prior to running VMMC programs, the user must create an active ***session*** on all VMMC cluster nodes. A VMMC session consists of a *session name*, a *user name*, a *user password*, a *local drive*, a *network share*, and an *output logging directory* (see Figure 2). The user must specify a non-empty string for the session name. The string can be of any characters. The session name is unique on any VMMC cluster node.

Each session requires a file system drive on which the `VMMCSVR` sets a new process' current working directory[2]. The current working directory is a full path relative to the drive, without the drive letter attached. The drive can be either a local disk drive (e.g., C:) on the node, or represent a mounted network share. In the latter case, the user must specify a network share using Uniform Naming Convention (e.g., `\\FileServer\vmmc`) during session creation. However, the user need not specify a drive letter for mounting a network share. `VMMCSVR` can select an available drive letter to use. Furthermore, the drive letter for a given session need not be the same on all nodes. `VMMCSVR` keeps track of the drive letter for each session and sets a process' working directory accordingly when creating processes.

When a user deletes a session, all processes created within the session are killed automatically by `VMMCSVR`. The network share is also dismounted during session destruction.

[2] Mounting the network share on the local drive is necessary because the NT operating system cannot use a UNC (Uniform Naming Convention) path as a process' working directory.

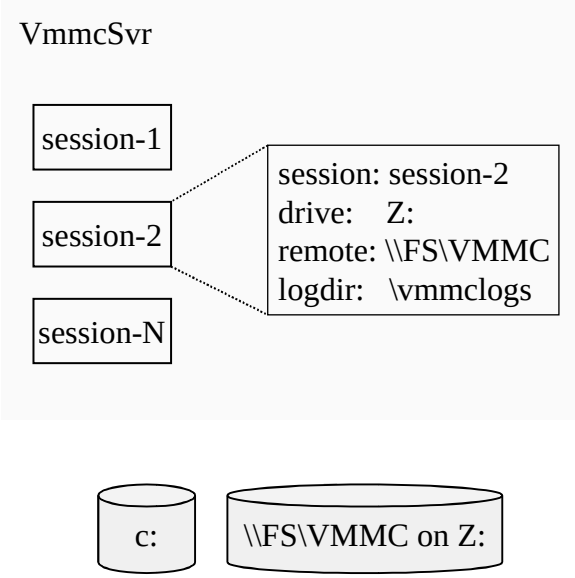


Figure 2. An example VMMC session

Using CFGVMMC.EXE

The VMMC SDK distribution supplies a utility program, `CFGVMMC.EXE` (in the `sdk\utils` directory), for creating and deleting sessions. `CFGVMMC` runs on any Windows NT workstation, including non-VMMC cluster nodes. It communicates with VMMCSVR services using Windows NT RPC mechanism.

Syntax:

```
CFGVMMC hostname add session user password drive netShare outDir
CFGVMMC hostname del session
```

Where:

`session`	a non-empty string of characters
`user`	a string, it can be empty, e.g., `vmmc` or `""`
`password`	a string, and could be empty, e.g., `vmmcrulz` or `""`
`drive`	a two-letter drive name (e.g., `Z:`) or the character `'-'` in which case `VMMCSVR` allocates the drive
`netShare`	a UNC path for a network share, e.g., `\\fs\test` or a null string `""`
`outDir`	a string representing the absolute path for the Output logging directory, e.g., `\vmmcout`. It cannot be an null string.

Examples:

```
CFGVMMC node1 add TestSession guest passwd K: \\FS\Guest \tmp
```

mounts `\\FS\Guest` on `K:` using `passwd` for account `guest`

```
CFGVMMC node1 add TestSession guest passwd - \\FS\Guest \tmp
```
VMMCSVR selects a drive to mount \\FS\Guest

```
CFGVMMC node1 add TestSession “”  “” F: “” \tmp
```
F: is either a local disk drive or an already-mounted drive

```
CFGVMMC node1 del TestSession
```

3. Running VMMC Programs

Once an active VMMC session is established, the user can run VMMC programs either directly on the VMMC nodes or create them remotely from a non-VMMC PC using the CFGVMMC utility program. But the first thing the user needs to do is set two environment variables: VMMCSESSION and VMMCHOSTS. This can be done with the SET command in a Windows command shell:

```
Z:\> Set VMMCHOSTS=node1 node2 node 3 node4
Z:\> Set VMMCSESSION=TestSession
```

VMMCSESSION informs the VMMC program of the session it belongs to. VMMCHOSTS tells the VMMC program how many nodes are available. The list of available nodes can be retrieved by a call to `vmmc_SessionHosts()`.

Running programs on a VMMC Node

On a VMMC node, the user can then run VMMC programs directly from the command shell. The user should have already created an active session on all the nodes where the user intends to create VMMC processes. Assuming the user uses drive X as the session drive, the following steps are required to run a VMMC program:

```
C:\>X:
X:\>Set VMMCHOSTS= u2 ledzep sade
X:\>SET VMMCSESSION=BLAH
X:\>CD  mytestdir
X:\mytestdir>testvmmc.exe
```

Running programs using CFGVMMC

The user can also create VMMC processes remotely from a non-VMMC PC, using the CFGVMMC utility. This program can be found in the sdk/utils directory of the VMMC-SDK distribution. `CFGVMMC` uses RPC to communicate with the `VMMCSVR` service running on each VMMC node for process creation and destruction.

Syntax:

```
CFGVMMC hostname run  session workingDir programPath [args]
CFGVMMC hostname kill session [pid]
```

Where:

`session`	the name of an active session
`workingDir`	the absolute path name of the working directory,e.g., `\mytest\bin`

`programPath` the path name for the executable program, it can be either relative to the working directory, e.g., `test.exe`, or absolute, e.g., `\mytest\bin\test.exe`. The suffix, `.exe` or `.bat`, cannot be omitted.

Examples:

Suppose under the network share `\\FileServer\vmmctest`, there are two directories:

- `bin\` which contains executables
- `tmp\` for output logging

Under `bin`, there is a VMMC program, `TEST.EXE` with syntax:

```
TEST.EXE arg1 arg2
```

```
CFGVMMC node1 run TestSession \ \bin\test.exe arg1 arg2
```
run `test.exe` with `\` as the working directory

```
CFGVMMC node1 run TestSession \bin test.exe arg1 arg2
```
run `test.exe` with `\bin` as the working directory

```
CFGVMMC node1 kill TestSession
```
kill all programs created in `TestSession`

```
CFGVMMC node1 kill TestSession 123
```
kill process with Pid=123 in `TestSession`

4. Output Logging

The output of VMMC processes that are created by VMMCSVR can be redirected to text files. To enable this feature, the user must specify a directory (the *output logging directory*) to store the redirected output logs, when creating a session. The output of each VMMC process is recorded in a file in the logging directory. The file name is a concatenation of the fixed string "VMMC", the name of the node on which the process is created, and the process' logical PID. The logical PID of a process is allocated by VMMCSVR. For example, the process with a logical PID `xyz` on `node1` will have an output logging file: `VMMC_node1.xyz`.

Processes spawned by a common parent (or grandparent) process are grouped into a *run group*. The run group ID is assigned by VMMCSVR when creating the first process (the *lead*) in the group. This lead process is the one that user starts from a command shell on a VMMC node, or the one that user creates remotely using the CFGVMMC utility. Subsequent processes *spawned* by this lead process inherit the run group ID.

The names for all output logging within a run group are recorded in an output control file. Its file name is a concatenation of the session name and the run group ID, e.g., TestSession_11. An example of an output control file is as follows:

```
#
# VMMC Output Control For Session [test]  Run #11  User []
#
VMMC_NODE1  11    # HOST=NODE1 REALPID=66  TIME=1999/ 2/14-15:25
VMMC_NODE2  12    # HOST=NODE2 REALPID=337 TIME=1999/ 2/14-13: 6
```

Each process in the run group is represented on a line after the comments. The syntax for each line is:

```
VMMC_nodename     LogicalPid  # comments
```

In the above example, there are two processes created in this group. One on `node2`, with logical PID 11 and real PID 233. The output logging file for this process is `VMMC_node2.11` in the logging directory.

5. Sample VMMC Programs

This section describes the example programs that come with the VMMC release. The pre-compiled sample programs can be found in SDK/example/bin directory.

Basics

The programs we provide all manage process creation themselves. The user only needs to run the program on a single VMMC node. The program will create the appropriate number of processes on remote VMMC nodes, using `vmmc_Spawn()`.

Most programs only spawn remote processes on nodes that are listed in the `VMMCHOSTS` environment variable. The user needs to set this environment variable before running a program:

```
SET VMMCHOSTS = node1 node2 node3
```

Simple Examples

- `spawn.exe [level]`

  This program spawns itself on a remote node with `level-1`. The default level is 1. The recursion terminates when level reaches 0. The program tests the VMMC cluster server's spawn facility.

- `latency.exe numberofwords numberofiterations`

  This program measures the single-message Ping-Pong latency between two VMMC nodes.

- `all2all.exe method numberofnodes numberofwords numberofiterations`
  where: method = `pairwise` or `random`

  This program measures the all-to-all communication bandwidth among a number of VMMC nodes. At each iteration, every process on each node sends a number of words to the other processes in either random or pairwise order.

Two Node Bandwidth Curve Examples

This family of programs shares the same command line syntax:

```
<progname> <iterations> <start_nwords> <end_nwords>
```

These programs report the bandwidth of a particular communication pattern. The initial message size is `<start_nwords>` which doubles until it reaches the final size `<end_nwords>`. Each message size test

is repeated for a number of <iterations>. For example, to test the message size sequence 1, 2, 4, ..., 1024, with 10,000 iterations at each message size, use the command: <progname> 10000 1 1024

- PPBandwidth: average bandwidth of ping-pong communication between 2 nodes.
- OnewayBandwidth: average bandwidth of one node continuously pumping data to the other.
- BidirBandwidth: average bandwidth for two nodes sending data the each other simultaneously.
- FetchBandwidth: average bandwidth of one node fetch data from the other continuously.

Multi-node Examples

This family of examples share the same command line syntax:

```
<prog_name> <num_nodes> <nwords> <iterations>
```

These programs' communication pattern involves communication between all node involved. The program runs on <num_nodes> contiguous nodes started from the master node, from which the command is issued, according to the order of hosts returned by vmmc_SessionHosts(). Each message involves <nwords> words. They can be used to test the connectivity among node.

- RandomChain: The pattern is A->B, B->C, C->D, ..., X->A
  all nodes are chosen and repeated randomly except that A is the master node.
- RandomOne2All: The pattern is A->others, others->B, B->others, others->C, ..., others->A
  all nodes are chosen and repeated randomly except that A is the master node.
- RandomAll2All: The pattern is All->All, All->All, ... until <iterations> times

Programming Model

1. VMMC Overview

Virtual memory-mapped communication is a model for protected user-level data transfer from the sender's virtual address space to the receiver's virtual address space. Communication is protected because data transfer can take place only after the receiver gives the sender permission to transfer data to a given area of the receiver's virtual address space. The receiving process expresses this permission by *exporting* areas of its address space as *receive buffers* where it is willing to accept incoming data. A sending process must *import* remote buffers (that are used as handles) which specify the available remote destinations. An exporter can restrict which processes and hosts can importer a buffer (i.e. send it data). VMMC enforces the restrictions when a process attempts to import a buffer (i.e., acquire a handle to the remote process's memory). After a successful import, the sender can transfer data from its virtual memory to the imported receive buffer. VMMC makes sure that transferred data does not overwrite addresses outside the buffer specified by the receiver.

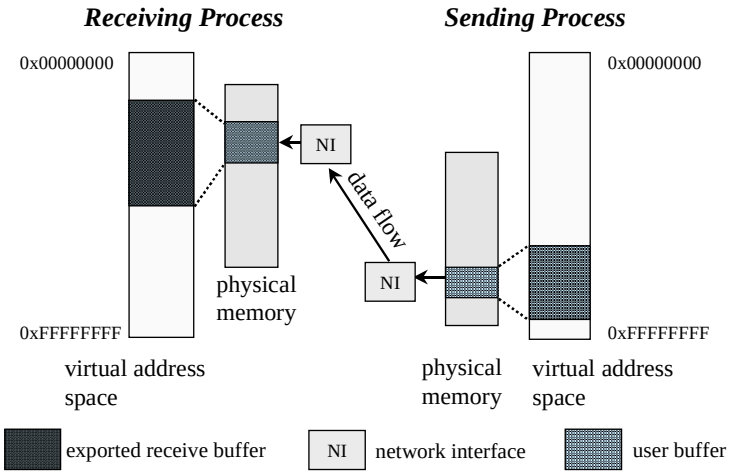


Figure 3. Virtual Memory Mapped Communication

VMMC supports two data transfer modes: *deliberate update* and *automatic update*. Automatic update is only available on SHRIMP network interfaces and will not be discussed here. Deliberate update is an explicitly initiated transfer of a contiguous block of data from any (readable) virtual address of the sending process to a previously imported receive buffer that represents a virtual address range of the receiving process (see Figure 3). The `vmmc_SendData*()` family of functions are used transfer data in this fashion. VMMC also allows data to flow in the opposite direction, this is accomplished using the `vmmc_GetData*()` family of functions. Note that VMMC guarantees in-order, reliable delivery of deliberate update messages.

When a message arrives at its destination, it is transferred directly into the memory of the receiving process, without interrupting the receiver's CPU. Thus *there is no explicit receive operation in VMMC.*

VMMC supports user-level buffer management because data transfer is performed between user-level memory locations. Buffer management is divorced from the data movement mechanism and becomes the responsibility of the communicating parties. Zero-copy protocols are possible because data transfers occur directly between applications.

The CPU overhead to send data is very small, only a few user-level instructions are needed for deliberate update. The model does not impose any CPU overhead to receive data, as there is no explicit receive operation. CPU involvement in receiving data can be as little as checking a flag; moreover program logic can be used to reason about what data has already arrived (since messages are delivered in order).

The model as described in this section can be applied to communication between processes executing on one uniprocessor machine, on separate processors of a shared memory multiprocessor, or between processes executing on different nodes in a local area network. In the former two cases, VMMC is a special restricted case of shared memory communication with deliberate update added for bulk transfer. The LAN case is discussed in the next section.

2. Cluster Organization

The VMMC cluster is a set of nodes. Each node is identified with a unique node identifier of type `vmmc_node_t`. The calls `vmmc_SessionHosts()` and `vmmc_AllHosts()` are used to obtain the identification of nodes in the VMMC cluster. While `vmmc_MyNode()` returns the id for the node of the calling process. `vmmc_MyPid()` returns the pid of a calling process. Note that full process identification is a `(vmmc_node_t, vmmc_pid_t)` pair.

User address space is divided in VMMC pages, which size is returned by `vmmc_PageSize()` (currently 4096 bytes). Each VMMC page contains integer number of VMMC words. The size of a VMMC word is returned by `vmmc_WordSize()`(currently 4 bytes).

3. Receive Buffers

Communication in the VMMC model is based on *receive buffers*. A receive buffer is a contiguous region of process virtual memory. Other processes can directly send data to, or fetch data from, a receive buffer. Each receive buffer is identified with user-selected buffer id (`uint32`). A receiver process makes a receive buffer available to senders with the `vmmc_ExportRecvBuf()` call. The buffer id must be unique among all ids of receive buffers exported by a given process. Receive buffers cannot overlap.

4. Importing Receive Buffers

A sender process has to import a given receive buffer before it can send any data. The import operation is implemented with the `vmmc_ImportRecvBuf()` call. Import succeeds only after the corresponding export call has been completed for this receive buffer (on the other node). There is also an asynchronous version of the import call, `vmmc_ImportRecvBufAsync()`, which issues only an import request and returns immediately returning a handle to the outstanding request. The progress of an asynchronous import request is verified with `vmmc_RecvBufStatus()`.

For a given process, ids of exported and imported buffers belong to two disjoint name spaces. As a result, one buffer id can be used in both export and import calls. However, for a given process, ids of exported buffers have to be unique. The full identification of an imported buffer is not its buffer id, but a triple

`(uint32:bufid, vmmc_node_t, vmmc_pid_t)`. As a result it is possible to import two buffers with the same buffer id, if two different processes exported them.

With one export call, a process can export exactly one receive buffer to potentially unlimited number of processes. With one import call, a process can import only one receive buffer.

We say that a successful import establishes an *import-export link* between a remote receive buffer and the local *DestSpace*. Establishing an import-export mapping requires a trusted third party (operating system kernel or daemon process) to verify protection. Therefore, creating mappings can be relatively expensive. However, it should occur infrequently, as import is required only once for a given receive buffer; afterward messages can be sent directly from user-level.

5. Destination Proxy Space (DestSpace)

Destination proxy space (*DestSpace* for short) is an *imaginary* address space in each sender process; it is used to represent remote imported receive buffers. An imported remote receive buffer is represented by an equal size contiguous region in *DestSpace*. In particular, an application can use pointer arithmetic on address in *DestSpace*. However, *DestSpace* is not backed by local memory and its addresses do not refer to local data or code. Instead, they are only used to specify destinations for remote data transfers (i.e. they act as handles to destination buffers). An address in *DestSpace* is translated by VMMC into a destination machine, process, and virtual address.

6. Data Transfer

Data transfer can occur from anywhere in a sender's virtual memory to a previously imported receive buffer. The `vmmc_SendData*()` family of functions are used to initiate communication on the sender's side. `vmmc_SendData()` takes as arguments an address in *DestSpace*, which identifies receive buffer to be used, a local address which identifies data to be send, and `nwords` which gives the size of the message. If no error occurs, `vmmc_SendData()` returns after all data has been sent out to the network.

VMMC also provides a non-blocking variant of `vmmc_SendData()` called `vmmc_SendDataAsync()`. The non-blocking send is designed to minimize the CPU overhead required to start a data transfer. The progress of an asynchronous send request is verified with `vmmc_AsyncStatus()`.

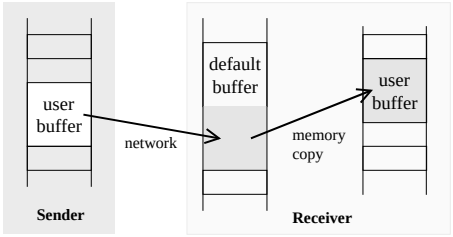
`vmmc_GetData()` and `vmmc_GetDataAsync()` are similar except that the direction of data flow is reversed. That is, a process can fetch data from a remote node.

7. Transfer Redirection

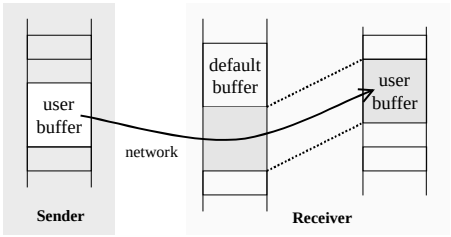
The basic virtual memory-mapped communication model provides protected, user-level communication to move data directly from a send buffer to a receive buffer without any copying. Since it requires the sender to know the receiver buffer address, it does not provide good support for connection-oriented high-level communication APIs.

VMMC includes a mechanism called *transfer redirection*. The basic idea is to use a default, *redirectable* receive buffer when a sender does not know the final receive buffer addresses.

Redirection is a local operation affecting only the receiving process. The sender does not have to be aware of a redirection and always sends data to the default buffer. When the data arrives at the receive side, the redirection mechanism checks to see whether a redirection address has been posted. If no redirection address has been posted, the data will be moved to the default buffer. Later, when the receiver posts the receive buffer address, the data will be copied from the default buffer to the receive buffer, as shown in Figure 4a.



(a) A copy takes place when the receiver posts its buffer address too late



(b) Transfer redirection moves data directly from the network to the user buffer

Figure 4. Transfer Redirection

If the receiver posts its buffer address before the message arrives, the message will be put into the user buffer directly from the network without any copying, as shown in Figure 4b. If the receiver posts its buffer address during message arrival, the message will be partially placed in the default buffer and partially placed in the posted receive buffer. The redirection mechanism tells the receiver exactly how much and what part of a message is redirected. When partial redirection occurs, this information allows the receiver to copy the part of the message that is placed in the default buffer.

There are two calls a receiver uses for transfer redirection on a redirectable buffer. The first is:

```
(vmmc_result_t) res = vmmc_PostRedir( vmmc_exphandle_t  bufHandle,
                                      uint32            redirOffset,
                                      uint32            numWords,
                                      int32             *userBuf)
```

where `bufHandle` is a handle to an exported buffer, `redirOffset` is the buffer's word offset from which to initiate redirection, `numWords` is the number of words to redirect, and `userBuf` is the destination for the redirected data. This call posts a transfer redirection at user level and currently at most one redirection can be posted for a receive buffer.

The second call is:

```
(vmmc_result_t) res= vmmc_EndRedir(vmmc_exphandle_t bufHandle,
                                   uint32           *numWordsPtr,
                                   uint32           *firstWordOffsetPtr)
```

where `bufHandle` is a handle identifying an exported *redirectable* buffer, `numWordsPtr` is the number of words that were redirected to the user buffer, and `firstWordOffsetPtr` is the index of the first word that was redirected to the user buffer. This call terminates the transfer redirection posted to the redirectable buffer. If no data has been redirected, the number of redirected words is zero. If a redirection did happen, the redirected data is in the contiguous user address-space area that starts at address `userBuf + *firstWordOffsetPtr` and continues for `*numWordsPtr` words (see Figure 5).

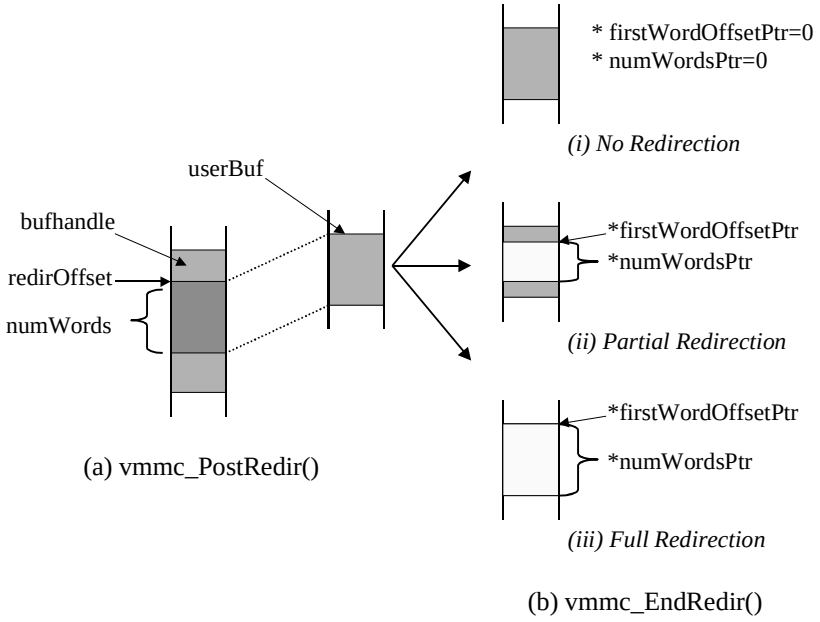


Figure 5. vmmc_PostRedir() and vmmc_EndRedir()

The transfer redirection mechanism naturally extends the basic communication model and it is very simple. It is controlled entirely by the receiver side; the sender does not need to know what the actual destination address is, nor whether a transfer redirection takes place.

Our implementation allows redirection to happen at most once for each `vmmc_PostRedir()`. As soon as a message is redirected from a redirectable buffer to a receive buffer, it ends the redirection automatically. We choose this design to simplify `vmmc_EndRedir()`. Otherwise, it would have to return multiple redirection regions.

Transfer redirection is designed for a single pair of sender and receiver processes. This design decision is based on the fact that most connection-based high-level APIs deal with a single pair of sender and receiver *per connection*. To avoid interleaving messages from different senders, a redirectable receive buffer should be imported by just one sender which is easy to enforce when the buffer is exported. If one really needs to support multiple senders and the communicating processes trust each other, a high-level protocol can be used to ensure that only one sender sends a message to a redirected buffer at a time.

How much data is actually redirected depends on the relative timing of `vmmc_PostRedir()` call and data arrival time, assuming `vmmc_EndRedir()` does not interfere with data redirection. If a redirection is posted before message arrival, the data will be redirected. If a redirection is posted after data starts arriving but before its transfer finishes, the remainder of arriving message will be redirected.

For redirectable receive buffers, VMMC provides additional functionality to help user processes figure out data arrival. When last chunk of a message arrives, a receive-buffer specific memory location in user space is updated with the receive buffer offset corresponding to the last word transferred. This word index is available to users via the `vmmc_DataEnd()` call.

8. Notifications

When sending a message we have the choice of transferring data only or data and control. The mechanism we use to transfer control is called a *notification*. Notifications are similar to UNIX signals. When a message with a notification attached arrives at destination receive buffer a user-level notification handler is invoked after the message data in user memory.

Handlers can be associated with receive buffers during the export operation. Each receive buffer can have zero or one handler. If a message with a notification arrives at receive buffer with no handler attached, this notification has no effect. `vmmc_SendDataNotify()` sends message with notification in deliberate update mode. This call takes the same arguments as `vmmc_SendData()`.

Each handler has the same function signature (i.e. number and type of arguments). The first argument is the address of the last word of data transferred by the message that generated this notification, the second argument is the value of this word. Since VMMC continues to receive incoming messages between the arrival of a message with notification and the call to the associated user handler, the data of a notification message can be overwritten by subsequent messages even before the handler is called. However, VMMC makes sure that the handler is called and passed the virtual address and value of the last word in the message that invokes the notification.

VMMC provides two calls: `vmmc_BlockNotifications()` and `vmmc_UnblockNotifications()` to control the delivery of notifications. Blocking notifications is useful to ensure consistency of data structures modified by both user-level handlers and main thread of execution. Blocking notifications is global, i.e. it affects all receive buffers of a given process. When notifications are blocked, they are queued by the system. After they are unblocked, they are delivered in-order to the appropriate user-level handlers. Since there is limited space to store queued notifications, they should not be blocked for too long.

For each `vmmc_BlockNotifications()` there should be a call to `vmmc_UnblockNotifications()`. Pairs of these calls can be nested. `vmmc_UnblockNotifications()` unblocks notifications only if it is called at the first nesting level. In this case, the call returns a positive integer, otherwise it returns zero. To make sure that notifications are unblocked unconditionally, one can call `vmmc_UnblockNotifications()` in the loop until it returns positive integer. If notifications are unblocked, further calls to `vmmc_UnblockNotifications()`have no effect.

While user-level notification handler is executing, notifications remain blocked. Notification handler should not block or wait spinning. Not all VMMC calls can be used from within notification handler. Both `vmmc_BlockNotifications()` and `vmmc_UnblockNotifications()` can be called from within the handler, provided they are paired. However, any attempt to unblock notifications

unconditionally by repeated calls to `vmmc_UnblockNotifications()` will eventually return an error (as notifications must remain blocked within a handler).

9. Removal of Import-Export Link

There are two calls provided to undo export and import operations. The importer of a given buffer executes `vmmc_UnimportRecvBuf()`. This call undoes a previous `vmmc_ImportRecvBuf()` call by breaking the connection to the remote receive buffer, and de-allocating the local *DestSpace* mappinsg.

`vmmc_UnexportRecvBuf()` undoes a previous call to `vmmc_ExportRecvBuf()`. All existing connections to the buffer are forcibly broken, and the buffer is made unavailable for further connections. In particular all importers of this buffer can no longer send any data to this buffer after this call completes.

`NOTE: these calls are not implemented.`

VMMC API Reference

This section describes all of the VMMC calls that are available to user applications.

VMMC Data Types

The section gives a brief overview of the various data types needed by the VMMC programmer.

- **int32**: 32-bit integer
- **uint**: unsigned integer
- **uint32**: unsigned 32-bit integer
- **ulong**: unsigned long integer
- **vmmc_callback_t**: specifies the type for a user-level function that can be invoked when data arrives. The function must be defined as
  `void (*vmmc_callback_t) (int32*, int32)`
- **vmmc_exphandle_t**: a handle type to keep track of exported buffers
- **vmmc_imphandle_t**: a handle type to keep track on imported buffers
- **vmmc_perm_t**: a type that specifies an export buffer's permission
- **vmmc_node_t**: a type used to store node identification
- **vmmc_pid_t**: a type used to store VMMC PIDs
- **vmmc_async_handle_t**: a type of handle used to keep track of outstanding asynchronous requests.

VMMC Return Values

Most VMMC calls return the type vmmc_result_t. Negative integer indicates an error, while zero means no error occurred. Errors can be reported with `vmmc_Error()`.

The following return values are possible:

- **vmmc_Success**: the call completed successfully (the constant 0)
- **vmmcErr_BadAlignment**: newly created buffer is not aligned to VMMC word boundary; or send or destination address is not aligned to such boundary.
- **vmmcErr_BufAlreadyExists**: receive buffer with a given buffer id already as been exported
- **vmmcErr_BadIbufState**: imported receive buffer is in bad state
- **vmmcErr_BadRbufState**: exported receive buffer is in bad state.
- **vmmcErr_BadProxyExtension**: exported receive buffer is in bad state.
- **vmmcErr_BadProxyAddr**: proxy address does not belong to imported receive buffer (or first and last proxy addresses specified by `vmmc_SendData()` do not belong to the same imported receive buffer).
- **vmmcErr_BadSize**: number of words to be sent is not positive
- **vmmcErr_BadArg**: number of
- **vmmcErr_NotImplemented**: number of words to be sent is not positive
- **vmmcErr_UnblockedNotifications**: block
- **vmmcErr_BlockedNotifications**: blocked notifications lead to deadlock. This happens when we want to unexport a buffer with pending notifications that are blocked.
- **vmmcErr_NoPhysDestSpace**: no more destination space
- **vmmcErr_NoVirtDestSpace**: no more destination space
- **vmmcErr_NoMem**: VMMC cannot allocate memory

- **vmmcErr_NoSuchNode**: bad node id
- **vmmcErr_NoSuchProcess**: bad process id
- **vmmcErr_NotInHandler**: this call cannot be used from within notification handler
- **vmmcErr_OverlappedBufs**: new receive buffer would overlap with existing buffer
- **vmmcErr_InProgress**: an asynchronous send or get request is pending

vmmc_AllHosts()

Returns the node information of all hosts in the VMMC cluster. This includes hosts that may not be part of the user's current session (see `vmmc_SessionHosts()`).

Synopsis:

(vmmc_result_t) res = vmmc_AllHosts (uint32 *nhosts, vmmc_node_t **hosts)

Parameters:

OUT uint32 *nhosts: returns the number of hosts

OUT vmmc_node_t **hostIds: returns an array of `nhosts` host ids

Description:

`vmmc_AllHosts()` returns *all* the nodes of the VMMC cluster, where `hosts` points to an array of `vmmc_node_t`. The size of the array is `nhosts`. Once a user program is started, nodes should not be added or deleted. The hostIds array should not be freed or written by user program.

Returns:

- `vmmc_Success`
- negative error code on failure

vmmc_AsyncStatus()

Returns the status of an asynchronous request.

Synopsis:

(vmmc_result_t) res = vmmc_AsyncStatus (vmmc_async_handle_t *handle)

Parameters:

IN vmmc_async_handle_t *handle: a handle identifying an outstanding asynchronous request

Description:

`vmmc_AsyncStatus()` returns the status of an asynchronous request initiated with either `vmcc_SendDataAsync()`, `vmcc_SendDataAsyncNotify()` or `vmmc_GetDataAsync()`.

Returns:

- `vmmc_Success` if the request has completed
- `vmmcErr_InProgress` if the request is still in progress
- other negative error code on failure

vmmc_BlockNotifications()

Blocks delivery of notifications.

Synopsis:

(void) vmmc_BlockNotifications()

Parameters:

none

Description:

`vmmc_BlockNotifications()` blocks delivery of notifications to all receive buffers of a calling process. Blocked notifications are queued and will be delivered after notifications are unblocked by a call to `vmmc_UnblockNotifications()`. Each call to `vmmc_BlockNotifications()` should be paired with a call to `vmmc_UnblockNotifications()`. These pairs can be nested, in which case the last `vmmc_UnblockNotifications()` (at nesting level one) will unblock notifications.

Notifications are automatically blocked when notification handler is called. `vmmc_BlockNotifications()` can be called from within a notification handler.

Returns:

no return value

vmmc_ClearDataEnd()

Resets the value of the “end-of-data” word index associated with a exported *redirectable* buffer.

Synopsis:

(void) vmmc_ClearDataEnd(vmmc_exphandle_t bufHandle)

Parameters:

IN vmmc_exphandle_t bufHandle: a handle identifying an exported *redirectable* buffer

Description:

VMMC keeps track of the index of the last word sent a redirectable buffer. This word index is relative to the start of a buffer and can range from zero to one less than the buffer size (in words). `vmmc_ClearDataEnd()` resets this index to `vmmcErr_NoSuchError`. The index can read with the function `vmmc_DataEnd()`.

Returns:

no return value

vmmc_DataEnd()

Returns the value of the "end-of-data" word index associated with a *redirectable* exported buffer.

Synopsis:

(int) res = vmmc_DataEnd(vmmc_exphandle_t bufHandle)

Parameters:

IN vmmc_exphandle_t bufHandle: handle identifying an exported *redirectable* buffer

Description:

VMMC keeps track of the index of the last word sent to redirectable buffers. This word index is relative to the start of a buffer. `vmmc_DataEnd()` returns this index for a specified buffer. If the buffer has received no data, `vmmcErr_NoSuchError` is returned. The user can reset the index to `vmmcErr_NoSuchError` by using `vmmc_ClearDataEnd()`.

Returns:

- `vmmcErr_NoSuchError` when no data has arrived in the buffer or if the user has reset the index with a call to `vmmc_ClearDataEnd()`
- a positive integer specifying the *word index* of the buffer where the last word was written to

vmmc_EndRedir()

Terminates redirection on a specified redirectable buffer.

Synopsis:

```
(vmmc_result_t) res = vmmc_EndRedir( vmmc_exphandle_t  bufHandle,
                                     uint32            *numWordsPtr,
                                     uint32            *firstWordOffsetPtr)
```

Parameters:

IN vmmc_exphandle_t bufHandle:	handle identifying an exported *redirectable* buffer
OUT uint32 *numWordsPtr:	number of words that were redirected to the user buffer
OUT uint32 *firstWordOffsetPtr	index of the first word that was redirected to the user buffer

Description:

Terminates redirection on a specified buffer. The last two arguments return the status of the redirection. It could be that all, some, or none, of the incoming data was redirected. Use `*numWordsPtr` and `*firstWordOffsetPtr` to determine exactly how much data was redirected.

Returns:

- `vmmc_Success` (the constant 0)
- negative error code on failure

vmmc_EqualNodes()

Determines if two VMMC nodes are the same.

Synopsis:

(int) res = vmmc_EqualNodes(vmmc_node_t node1, vmmc_node_t node2)

Parameters:

IN vmmc_node_t node1: a VMMC node

IN vmmc_node_t node2: a VMMC node

Description:

Determines if two VMMC nodes are the same. We supply this function because `vmmc_node_t` is an opaque type. The user should not access any members of `vmmc_node_t`.

Returns:

- 1 if the nodes are equal
- 0 if the nodes are different

vmmc_ErrorStr()

Returns a string describing a VMMC error code.

Synopsis:

(char *) str = vmmc_ErrorStr (int errCode)

Parameters:

IN int errCode: error code (negative integer), as returned by other VMMC calls

Description:

Returns a null terminated ASCII string describing a VMMC error code. If the string is needed longer than the execution of the calling function it should be copied to a user character array. The user must never free the string returned by this call.

Returns:

a character string describing the VMMC error

vmmc_ExportRecvBuf()

Exports a receive buffer.

Synopsis:

```
(vmmc_result_t) res = vmmc_ExportRecvBuf ( int32              *buf,
                                           uint32             nwords,
                                           uint32             rbufid,
                                           vmmc_callback_t    proc,
                                           uint32             flags,
                                           vmmc_perm_t        perm,
                                           vmmc_exphandle_t   *bufHandle)
```

Parameters:

IN int32 *buf : address of the receive buffer

IN uint32 nwords: size of the receive buffer in words

IN uint32 rbufid: receive buffer id

IN vmmc_callback_t proc: notification handler, user-level function
or `NULL` for no handler

IN uint32 flags: specify export characteristics
one of:

`BUF_WRITE_ONLY`	buffer can only be written
`BUF_READ_ONLY`	buffer can only be read
`BUF_READWRITE`	buffer can be read and written

optionally OR-ed with one of:

`BUF_REDIRECTABLE`	buffer can redirected
`BUF_GLOBAL_SPACE`	buffer is allocated from the global buffer space

IN vmmc_perm_t perm: export permissions (not used yet)

OUT vmmc_exphandle_t *bufHandle: handle that will identify the exported buffer

Description:

Exports receive buffer for importing by senders. Note that this call *does not* allocate any memory. If there is no memory allocated to back this buffer, page faults may occur.
The newly created receive buffer is identified with `rbufid`, and occupies memory between `buf` and `buf + vmmc_WordSize()*nwords - 1`.
`buf` should be word aligned. Receive buffers cannot overlap.

Returns:

- `vmmc_Success`
- negative error code on failure

vmmc_GetData()

Fetches data from a remote exported buffer.

Synopsis:

```
(vmmc_result_t) res = vmmc_GetData (  int32   *localDstBuf,
                                      int32   *remoteSrcProxyAddr,
                                      uint32  nwords)
```

Parameters:

IN int32 *localDstBuf:	address of user buffer to place fetched data. buffer memory must be allocated.
IN int32 *remoteSrcProxyAddr:	address in local *DestSpace* corresponding to an imported remote receive buffer
IN int nwords:	size of the data in words

Description:

`vmmc_GetData()` fetches `nwords` from address `remoteSrcProxyAddr` and into the local buffer specified by `localDstBuf`. `vmmc_GetData()` returns after the message has arrived in the user buffer. `vmmc_GetData()` is the compliment of `vmmc_SendData()`.

`remoteSrcProxyAddr` determines the source of the data. This is the receive buffer which proxy in local *DestSpace* contains `localDstBuf` address. Since each *DestSpace* address belongs to no more than one proxy, this identification is unique. The data will be put in the receive buffer on destination node starting from the offset equal to the offset of `localDstBuf` from the beginning of this buffer proxy in local *DestSpace*.

Both `remoteSrcProxyAddr` and `localDstBuf` should be VMMC word aligned. Both `remoteSrcProxyAddr` (source address of the first word) and `remoteSrcProxyAddr+4*nwords-1` (source address of the last word) should belong to the same imported receive buffer.

`vmmc_GetData()` can be used from within notification handler.

Note: Page faults are possible if `remoteSrcProxyAddr` or `localDstBuf` do not correspond to valid addresses.

Returns:

- `vmmc_Success`
- negative error code on failure

vmmc_GetDataAsync()

Asynchronously gets data from a remote buffer.

Synopsis:

```
(vmmc_result_t) res = vmmc_GetData (  int32                 *localDstBuf,
                                      int32                 *remoteSrcProxyAddr,
                                      uint32                nwords,
                                      vmmc_async_handle_t   *handle)
```

Parameters:

IN int32 *localDstBuf:	address of user buffer to place fetched data, buffer memory must be allocated
IN int32 *remoteSrcProxyAddr:	address in local *DestSpace* corresponding to an imported remote receive buffer
IN int nwords:	size of the message in words
OUT vmmc_async_handle_t *handle:	handle to the outstanding asynchronous request

Description:

`vmmc_GetDataAsync()` initiates an asynchronous request to retrieve `nwords` from address `remoteSrcProxyAddr` and copies them to the local buffer specified by `localDstBuf`. `vmmc_GetData()` returns as soon as the request is initiated.

`remoteSrcProxyAddr` determines the source of the data. This is the receive buffer which proxy in local *DestSpace* contains `localDstBuf` address. Since each *DestSpace* address belongs to no more than one proxy, this identification is unique. The data will be put in the receive buffer on destination node starting from the offset equal to the offset of `localDstBuf` from the beginning of this buffer proxy in local *DestSpace*.

Both `remoteSrcProxyAddr` and `localDstBuf` should be VMMC word aligned. Both `remoteSrcProxyAddr` (source address of the first word) and `remoteSrcProxyAddr+4*nwords-1` (source address of the last word) should belong to the same imported receive buffer.

`vmmc_GetDataAsync()` can be used from within notification handler.

Note: Page faults are possible if `remoteSrcProxyAddr` or `localDstBuf` do not correspond to valid addresses.

Returns:

- `vmmc_Success`
- negative error code on failure.

vmmc_ImportRecvBuf()

Imports a receive buffer.

Synopsis:

```
(vmmc_result_t) res = vmmc_ImportRecvBuf ( vmmc_node_t        node,
                                           vmmc_pid_t         pid,
                                           uint32             rbufid,
                                           vmmc_imphandle_t   *handle,
                                           int32              **proxyBuf,
                                           uint32             *nwords)
```

Parameters:

IN vmmc_node_t node:	address of a node on which exported the receive buffer
IN vmmc_pid_t pid:	process id that exported the receive buffer
IN uint32 rbufid:	receive buffer id
OUT vmmc_imphandle_t *handle:	handle to the imported buffer
OUT int32 **proxyBuf:	local address in *DestSpace* which corresponds to the imported receive buffer
OUT uint32 *nwords:	size of the imported buffer in VMMC words.

Description:

Imports a receive buffer named `rbufid` which has been exported by process `pid` running on `node`.

The imported receive buffer is allocated in a region of destination address space between `proxyBuf` and `proxyBuf + vmmc_WordSize()*nwords -1`.

Returns:

- `vmmc_Success`
- negative error code on failure

vmmc_ImportRecvBufAsync()

Issues an asynchronous request to import a receive buffer.

Synopsis:

```
(vmmc_result_t) res = vmmc_ImportRecvBufAsync (  vmmc_node_t        node,
                                                 vmmc_pid_t         pid,
                                                 uint32             rbufid,
                                                 vmmc_imphandle_t   *handle)
```

Parameters:

IN vmmc_node_t node:	address of a node on which exported the receive buffer
IN vmmc_pid_t pid:	process id that exported the receive buffer
IN uint32 rbufid:	receive buffer id
OUT vmmc_imphandle_t *handle:	handle to the import buffer request

Description:

Issues an asynchronous request to import a receive buffer named `rbufid` which has been exported by process `pid` running on `node`. The status (and completion) of the request is verified using the returned `handle` and the call `vmmc_ImportRecvBufStatus()`.

The imported receive buffer is allocated in a region of destination address space between `proxyBuf` and `proxyBuf + vmmc_WordSize()*nwords -1`.

Returns:

- `vmmc_Success`
- negative error code on failure.

vmmc_ImportRecvBufStatus()

Checks the status of an outstanding asynchronous request to import a receive buffer.

Synopsis:

```
(vmmc_result_t) res = vmmc_ImportRecvBufStatus ( vmmc_imphandle_t  *handle,
                                                 int32             **proxyBuf,
                                                 uint32            *nwords)
```

Parameters:

IN/OUT vmmc_imphandle_t *handle:	handle to the import-buffer request
OUT int32 **proxyBuf:	local address in *DestSpace* which corresponds to the imported receive buffer
OUT uint32 *nwords:	size of the imported buffer in VMMC words.

Description:

Checks the status of an outstanding asynchronous request to import a receive buffer. `handle` identifies the outstanding request. The return result is `vmmc_Success` if the request completes successfully. In this case, `*proxyBuf` and `*nwords` represent the appropriate values for a buffer import.

If the asynchronous is not yet complete, the return value is `vmmcErr_InProgress`. `vmmcErr_BadAsyncHandle` is returned for invalid handles. While `vmmcErr_StateHandle` indicates that the application has already verified the success of this particular request.

The imported receive buffer is allocated in a region of destination address space between `proxyBuf` and `proxyBuf + vmmc_WordSize()*nwords -1`.

Returns:

- `vmmc_Success`
- `vmmcErr_InProgress` if the asynchronous request is not yet complete
- `vmmcErr_BadAsyncHandle` if the `handle` is not valid
- `vmmcErr_StaleHandle` if the `handle` has already been checked after completion
- negative error code on failure

vmmc_MyHostName()

Return the hostname of the machine.

Synopsis:

(char*) res = vmmc_MyHostName()

Parameters:

none

Description:

Returns the hostname of the machine.

Returns:

(char*) hostname string

vmmc_MyNode()

Returns the node id of the machine.

Synopsis:

(vmmc_node_t) res = vmmc_MyNode()

Parameters:

none

Description:

Returns the node id of the machine.

Returns:

(vmmc_node_t) a node Id

vmmc_MyPid()

Returns the pid of the calling process.

Synopsis:

(vmmc_pid_t) res = vmmc_MyPid()

Parameters:

none

Description:

Returns the pid of the calling process.

Returns:

(vmmc_pid_t) the pid of the calling process

vmmc_NameToNode()

Returns the node Id of a given hostname.

Synopsis:

(vmmc_node_t) res = vmmc_NameToNode(char *hostname)

Parameters:

IN char *hostname: name of host to convert

Description:

Returns the node Id of a given hostname.

Returns:

(vmmc_node_t) a node Id

vmmc_NodeToName()

Returns the name of a given node.

Synopsis:

(char *) res = vmmc_NodeToName (vmmc_node_t node)

Parameters:

IN vmmc_node_t node: the node Id to convert

Description:

Returns the name of a given node.

Returns:

(char *) a hostname

vmmc_PageSize()

Returns the size of a VMMC page in bytes.

Synopsis:

(int) res = vmmc_PageSize()

Parameters:

none

Description:

Returns the size of a VMMC page in bytes.

Returns:

(int) the size of a VMMC page in bytes

vmmc_Parent()

Returns the pid and node id of the parent process.

Synopsis:

(vmmc_result_t) res = vmmc_Parent (vmmc_node_t *parentNode, vmmc_pid_t *parentPid)

Parameters:

OUT vmmc_node_t *parentNode: returns node of parent process

OUT vmmc_pid_t *parentPid: returns the pid of a parent process

Description:

vmmc_Parent() returns node and process id of the parent process. This call is guaranteed to work only if the caller process has been started with `vmmc_Spawn()`. In this case, the parent is the process which executed `vmmc_Spawn()`.

Returns:

- `vmmc_Success`
- `vmmcErr_NoParent`

vmmc_PostRedir()

Activates transfer redirection for an exported buffer.

Synopsis:

```
(vmmc_result_t) res = vmmc_PostRedir(  vmmc_exphandle_t   bufHandle,
                                       uint32             redirOffset,
                                       uint32             numWords,
                                       int32              *userBuf)
```

Parameters:

IN vmmc_exphandle_t bufHandle:	handle to exported buffer
IN uint32 redirOffset:	buffer's word offset to initiate redirection
IN uint32 numWords:	number of words to redirect
IN int32 *userBuf:	destination for the redirected data

Description:

Activates transfer redirection for an exported buffer at the specified `redirOffset` for `numWords`. There is no guarantee that redirection will take place. VMMC will try to redirect as much of the data as possible, but how much was actually redirected is reported by the call to `vmmc_EndRedir()`.

There can be only one outstanding redirection request per buffer.

Returns:

- `vmmc_Success`
- negative error code on failure.

vmmc_SendData()

Sends a message.

Synopsis:

```
(vmmc_result_t) res = vmmc_SendData (  int32   *localSrcAddr,
                                       int32   *remoteDestProxyAddr,
                                       uint32  nwords)
```

Parameters:

IN int32 *localSrcAddr:	address of data to send. It can be any address corresponding to allocated data.
IN int32 *remoteDestProxyAddr:	address in local *DestSpace* corresponding to an imported receive buffer.
IN int nwords:	size of the message in VMMC words

Description:

Sends a message of `nwords` taken from address `localSrcAddr` to remote memory specified by `remoteDestProxyAddr` (obtained by importing a buffer). `vmmc_SendData()` returns after the message has been transferred to the network.

Both `localSrcAddr` and `remoteDestProxyAddr` must be VMMC word aligned.
Both `remoteDestProxyAddr` (first word of destination address)
and `remoteDestProxyAddr+4*nwords-1` (last word of destination address) must belong to the same imported receive buffer.

vmmc_SendData() can be used from within notification handler.

Note: Page faults are possible if `localSrcAddr` or `remoteDestProxyAddr` do not correspond to valid addresses.

Returns:

- `vmmc_Success`
- negative error code on failure

vmmc_SendDataAsync()

Asynchronously sends a message.

Synopsis:

```
(vmmc_result_t) res = vmmc_SendDataAsync (  int32                 *localSrcAddr,
                                            int32                 *remoteDestProxyAddr,
                                            uint32                nwords,
                                            vmmc_async_handle_t   *handle)
```

Parameters:

IN int32 *localSrcAddr:	address of data to send. It can be any address corresponding to allocated data.
IN int32 *remoteDestProxyAddr:	address in local *DestSpace* corresponding to an imported receive buffer.
IN int nwords:	size of the message in VMMC words
OUT vmmc_async_handle_t handle:	handle to keep track of the asynchronous send request

Description:

Sends a message of `nwords` taken from address `localSrcAddr` to remote memory specified by `remoteDestProxyAddr` (obtained by importing a buffer). `vmmc_SendDataAsync()` returns immediately upon issuing the data transfer request without waiting for transmission to the network.. `vmmc_AsyncStatus()` is used to verify the progress of the data transfer request.

Both `localSrcAddr` and `remoteDestProxyAddr` must be VMMC word aligned.
Both `remoteDestProxyAddr` (first word of destination address)
and `remoteDestProxyAddr+4*nwords-1` (last word of destination address) must belong to the same imported receive buffer.

vmmc_SendDataAsync() can be used from within notification handler.

Note: Page faults are possible if `localSrcAddr` or `remoteDestProxyAddr` do not correspond to valid addresses.

Returns:

- `vmmc_Success`
- negative error code on failure

vmmc_SendDataAsyncNotify()

Asynchronously sends a message with a notification.

Synopsis:

```
(vmmc_result_t) res = vmmc_SendDataAsyncNotify(int32                 *localSrcAddr,
                                               int32                 *remoteDestProxyAddr,
                                               uint32                nwords,
                                               vmmc_async_handle_t   *handle)
```

Parameters:

IN int32 *localSrcAddr:	address of data to send. It can be any address corresponding to allocated data.
IN int32 *remoteDestProxyAddr:	address in local *DestSpace* corresponding to an imported receive buffer.
IN int nwords:	size of the message in VMMC words
OUT vmmc_async_handle_t handle:	handle to keep track of the asynchronous send request

Description:

Sends a message of `nwords` taken from address `localSrcAddr` to remote memory specified by `remoteDestProxyAddr` (obtained by importing a buffer). A notification is invoked when the data is received. `vmmc_SendDataAsyncNotify()` returns immediately upon issuing the data transfer request without waiting for transmission to the network.. `vmmc_AsyncStatus()` is used to verify the progress of the data transfer request.

Both `localSrcAddr` and `remoteDestProxyAddr` must be VMMC word aligned.
Both `remoteDestProxyAddr` (first word of destination address)
and `remoteDestProxyAddr+4*nwords-1` (last word of destination address) must belong to the same imported receive buffer.

vmmc_SendDataAsyncNotify() can be used from within notification handler.

Note: Page faults are possible if `localSrcAddr` or `remoteDestProxyAddr` do not correspond to valid addresses.

Returns:

- `vmmc_Success`
- negative error code on failure

vmmc_SendDataNotify()

Sends a message with a notification.

Synopsis:

```
(vmmc_result_t) res = vmmc_SendData (  int32   *localSrcAddr,
                                       int32   *remoteDestProxyAddr,
                                       uint32  nwords)
```

Parameters:

IN int32 *localSrcAddr:	address of data to send. It can be any address corresponding to allocated data.
IN int32 *remoteDestProxyAddr:	address in local *DestSpace* corresponding to an imported receive buffer.
IN int nwords:	size of the message in VMMC words

Description:

Sends a message of `nwords` taken from address `localSrcAddr` to remote memory specified by `remoteDestProxyAddr` (obtained by importing a buffer). A notification is invoked when the data is received. `vmmc_SendData()` returns after the message has been transferred to the network.

Both `localSrcAddr` and `remoteDestProxyAddr` must be VMMC word aligned.
Both `remoteDestProxyAddr` (first word of destination address)
and `remoteDestProxyAddr+4*nwords-1` (last word of destination address) must belong to the same imported receive buffer.

vmmc_SendDataNotify() can be used from within notification handler.

Note: Page faults are possible if `localSrcAddr` or `remoteDestProxyAddr` do not correspond to valid addresses.

Returns:

- `vmmc_Success`
- negative error code on failure

vmmc_SessionHosts()

Returns the names of the hosts that are part of the current user's session.

Synopsis:

(vmmc_result_t) res = vmmc_SessionHosts (uint32 *nhosts, vmmc_node_t **hostIds)

Parameters:

OUT uint32 *nhosts: returns the number of hosts

OUT vmmc_node_t **hostIds: returns nhosts host ids

Description:

`vmmc_SessionHosts()` returns *only* the nodes that are part of the current user's session. Once a user program is started, nodes should not be added or deleted. The hostIds array should not be freed or written by user program.

Returns:

- `vmmc_Success` (the constant 0)
- negative error code on failure

vmmc_SetDebugLevel()

Sets the amount of debug information that VMMC outputs.

Synopsis:

(void) vmmc_SetDebugLevel (int level)

Parameters:

IN int level: the debugging level

Description:

Sets the amount of debug information that VMMC outputs. Zero (default) turns off all information while a positive integer produces lots of fun VMMC debug messages.

Returns:

(void) nothing

vmmc_Spawn()

Spawns a process.

Synopsis:

```
(vmmc_result_t) res = vmmc_Spawn ( char          *execfile,
                                   char          **argv,
                                   vmmc_node_t   node,
                                   vmmc_pid_t    *pid)
```

Parameters:

IN char *execfile:	new process will execute the filename. If full path is not given, the path will be relative to the current working directory of the calling process.
IN char **argv:	list of arguments for new process, starting from `argv[1]`. Last element of this list must be NULL.
IN vmmc_node_t node:	identifies remote machine on which to start new process
OUT vmmc_pid_t *pid:	returns the pid of the new process.

Description:

`vmmc_Spawn()` starts a new process.

Returns:

- `vmmc_Success`
- negative error code on failure

vmmc_UnblockNotifications()

Conditionally unblocks delivery of notifications.

Synopsis:

(int) res = vmmc_UnblockNotifications()

Parameters:

none

Description:

`vmmc_UnblockNotifications()` conditionally unblocks delivery of notifications to all receive buffers of a calling process. VMMC maintains internal counter which counts blocking level of notifications. This counter is incremented each time `vmmc_BlockNotifications()` is called. When `vmmc_UnblockNotifications()` is called, and this counter is positive, it is decremented. When this counter is zero, a call to `vmmc_UnblockNotifications()` has no effect. Notifications are unblocked only if this counter reaches zero.

Notifications are automatically blocked when notification handler is called. `vmmc_UnblockNotifications()` can be called from within notification handler, but it cannot actually unblock notifications in such case. If the internal counter is one, and `vmmc_UnblockNotifications()` is called from the handler, this call returns the error `vmmc_ENotInHandler` and the counter remains unchanged.

Returns:

- a positive integer indicating the remaining number blocking levels. Notifications are still blocked.
- zero if notifications were successfully unblocked
- negative error code

Example:

The following loop unconditionally unblocks notifications (with error set if it is called inside handler):

```
int status;
while (status = vmmc_UnblockNotifications()) == 0)
      ;
if (status < 0)
      vmmc_Error(status, "unconditional unblocking");
```

vmmc_UnexportRecvBuf()

Unexports a receive buffer.

Synopsis:

(vmmc_result_t) res = vmmc_UnexportRecvBuf(vmmc_exphandle_t handle)

Parameters:

IN vmmc_exphandle_t handle: handle that corresponds to an exported receive buffer

Description:

Unexports the receive buffer specified by `handle`.

Returns:

- `vmmc_Success`
- negative error code on failure

vmmc_UnimportRecvBuf()

Unimports a receive buffer.

Synopsis:

(vmmc_result_t) res = vmmc_UnimportRecvBuf (vmmc_imphandle_t handle)

Parameters:

IN vmmc_imphandle_t handle: handle that corresponds to an imported receive buffer

Description:

Unimports the receive buffer specified by `handle`.

Returns:

- `vmmc_Success`
- negative error code on failure

vmmc_Version()

Returns the major and minor VMMC version numbers.

Synopsis:

(void) vmmc_Version (uint32 *major, uint32 *minor)

Parameters:

OUT uint32 *major: returns the VMMC version major number

OUT uint32 *minor: returns the VMMC version minor number

Description:

Returns the major and minor VMMC version numbers.

Returns:

(void) nothing

vmmc_WordSize()

Returns the number of bytes in a VMMC word.

Synopsis:

(int) res = vmmc_WordSize()

Parameters:

none

Description:

Returns the number of bytes in a VMMC word.

Returns:

(int) the number of bytes in a VMMC word

Index