VMCI Sockets Programming Guide

VMware ESX/ESXi 4.x

VMware Workstation 7.x

VMware Server 2.0

This document supports the version of each product listed and supports all subsequent versions until the document is replaced by a new edition. To check for more recent editions of this document, see http://www.vmware.com/support/pubs.

EN-000292-02

You can find the most up-to-date technical documentation on the VMware Web site at:

http://www.vmware.com/support/

The VMware Web site also provides the latest product updates.

If you have comments about this documentation, submit your feedback to:

docfeedback@vmware.com

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Contents

About This Book

The VMware® *VMCI Sockets Programming Guide* describes how to program virtual machine communications interface (VMCI) sockets. The VMCI sockets API facilitates fast and efficient communication between virtual machines, or between guest virtual machines and their host.

Revision History

VMware revises this guide with each release of the product or when necessary. A revised version can contain minor or major changes. Table 1 summarizes the significant changes in each version of this guide.

Table 1. Revision History

Revision	Description
20100521	Manual revised for the Workstation 7.1 release and for ESX/ESXi 4.x releases.
20091020	Manual revised slightly for the Workstation 7.0 release.
20090515	Revised manual, including host-to-guest stream socket support, for the ESX/ESXi 4.0 release.
20080815	Released manual, with socket options, for VMware Workstation 6.5 and VMware Server 2.0 products.
20080620	Draft of this manual for the VMware Workstation 6.5 Beta 2 and VMware Server 2.0 RC1 releases.

Intended Audience

This manual is intended for programmers who are developing applications with VMCI sockets to create C or C++ networking applications for guest operating systems running on VMware hosts, or applications that communicate between virtual machines. VMCI sockets are based on TCP sockets.

This guide assumes that you are familiar with Berkeley sockets or Winsock, the Windows implementation of sockets. If you are not familiar with sockets, "Appendix: Learning More About Sockets" on page 23 provides pointers to learning resources.

Document Feedback

VMware welcomes your suggestions for improving our documentation. Send your feedback to docfeedback@vmware.com.

VMware Technical Publications Glossary

VMware Technical Publications provides a glossary of terms that might be unfamiliar to you. For definitions of terms as they are used in VMware technical documentation go to http://www.vmware.com/support/pubs.

Technical Support and Education Resources

The following sections describe the technical support resources available to you. To access the current versions of other VMware books, go to http://www.vmware.com/support/pubs.

Online and Telephone Support

To use online support to submit technical support requests, view your product and contract information, and register your products, go to http://www.vmware.com/support.

Support Offerings

To find out how VMware support offerings can help meet your business needs, go to http://www.vmware.com/support/services.

VMware Professional Services

VMware Education Services courses offer extensive hands-on labs, case study examples, and course materials designed to be used as on-the-job reference tools. Courses are available onsite, in the classroom, and live online. For onsite pilot programs and implementation best practices, VMware Consulting Services provides offerings to help you assess, plan, build, and manage your virtual environment. To access information about education classes, certification programs, and consulting services, go to http://www.vmware.com/services.

About VMCI Sockets 1

This chapter includes the following topics:

This guide assumes that you know about either Berkeley sockets or Winsock, the Windows implementation. If you are new to sockets, see "Appendix: Learning More About Sockets" on page 23.

Introduction to VMCI Sockets

The VMware VMCI sockets library offers an API that is similar to the Berkeley UNIX socket interface and the Windows socket interface, two industry standards. VMCI sockets support fast and efficient communication between a virtual machine and its host, or between guest virtual machines on the same host.

Previous VMCI Releases

The original VMCI library was released as an experimental C language interface with Workstation 6.0. VMCI included a datagram API and a shared memory API. Both these interfaces were deprecated.

The VMCI sockets library was first released with Workstation 6.5 and Server 2.0. It was a supported interface, not experimental. The VMCI sockets library had more flexible algorithms, wrapped in a stream sockets API for external presentation. Stream socket support was improved for ESX/ESXi hosts when VMware vSphere™ and VMware vCenter™ were released.

How VMCI Sockets Work

VMCI sockets are similar to other socket types. Like local UNIX sockets, VMCI sockets work on an individual physical machine, and can perform interprocess communication on the local system. With Internet sockets, communicating processes usually reside on different systems across the network. Similarly, VMCI sockets allow different virtual machines to communicate with each other, if they reside on the same VMware host.

The VMCI sockets library supports both connection-oriented stream sockets like TCP, and connectionless datagram sockets like UDP. However, with VMCI sockets, a virtual socket can have only two endpoints and unlike TCP sockets, the server cannot initiate a connection to the client.

VMCI sockets support data transfer among processes on the same system (interprocess communication). They also allow communication among processes on different systems, even ones running different versions and types of operating systems. VMCI sockets comprise a single protocol family.

Sockets require active processes, so communicating guest virtual machines must be running, not powered off.

VMCI sockets are available only at the user level. Kernel APIs are not supported.

Persistence of Sockets

VMCI sockets lose connection after suspend and resume of a virtual machine.

In VMware vSphere with ESX/ESXi hosts and vCenter Server, VMCI sockets do not survive live migration with VMware VMotion™ from source to destination host. In VMware vSphere with ESX/ESXi hosts, VMCI stream socket connections are dropped when a virtual machine is put into fault tolerance (FT) mode. No new VMCI stream socket connections can be established while a virtual machine is in FT mode.

Socket Programming

If you have existing socket-based applications, only a few code changes are required for VMCI sockets. If you do not have socket-based applications, you can easily find public-domain code on the Web. For example, Apache and Firefox, as shown in Figure 1-1, "VMware Hosts with Stream VMCI Sockets in Guests," on page 9, use stream sockets and are open source.

Repurposing a networking program to use VMCI sockets requires minimal effort, because VMCI sockets behave like traditional Internet sockets on a given platform. However, some socket options do not make sense for communication across the VMCI device, so they are silently ignored to promote program portability.

Modification is straightforward. You include a header file, change the protocol address family, and allocate a new data structure. Otherwise VMCI sockets use the same API as Berkeley sockets or Windows sockets. See "Porting Existing Socket Applications" on page 11 for a description of the modifications needed.

Features in Specific VMware Releases

VMCI sockets communicate from guest to guest, or guest to host, on one VMware host. You can also use VMCI sockets for interprocess communications on a single guest. However, you cannot use VMCI sockets between virtual machines running on two separate physical machines, or from one host to another across a network.

In the VMware Workstation 6.5 releases VMware Server 2.0 and, stream sockets are not supported between host and guest, so you must use datagram sockets instead. Stream sockets work from guest to guest only. Datagram sockets work from guest to guest, host to guest, and guest to host.

As of the VMware Server 2.0 RC2 and Workstation 6.5 RC releases, you can set the minimum, maximum, and default size of communicating stream buffers. See "Set and Get Socket Options" on page 14.

The ESX/ESXi 4.0 (vSphere 4) release has complete user-level support for VMCI sockets. Datagram and stream sockets are supported between host and guests, and from guest to guest on both Linux and Windows.

In the Workstation 7.0 release, both datagram and stream sockets are supported on Linux hosts, Linux guests, and Windows guests. On Windows hosts, only datagram sockets are supported.

IMPORTANT Upgrade any virtual machines that use VMCI sockets to VMware virtual hardware version 7, which was introduced in VMware Workstation 6.5 and incorporated into ESX/ESXi 4.0.

Finding and Enabling VMCI Sockets

You must explicitly enable VMCI sockets on a virtual machine.

Location of Include File

The `vmci_sockets.h` include file is located in the following directories:

- Windows guest – `C:\Program Files\VMware\VMware Tools\VSock SDK\include`
- Linux guest – `/usr/lib/vmware-tools/include/vmci`
- Windows host – `C:\Program Files\VMware\VMware Workstation`
- Linux host – `/usr/lib/vmware/include/vmci`
- ESX/ESXi host – Not installed on the system

Enabling VMCI Communications

For two virtual machines to communicate, VMCI communication between them must be enabled for both virtual machines. You can enable VMCI from the user interface.

- For VMware Workstation, select **VM > Settings > Options > Guest Isolation > Enable VMCI**.
- From the vSphere Client on ESX/ESXi, select the VMCI device property **Enable VMCI Between VMs**. This is the same as setting the virtual machine VMCI device to `allowUnrestrictedCommunication` in the vSphere API. Currently this setting does not take effect until a virtual machine is restarted.

Use Cases for VMCI Sockets

VMCI sockets can help with the following solutions:

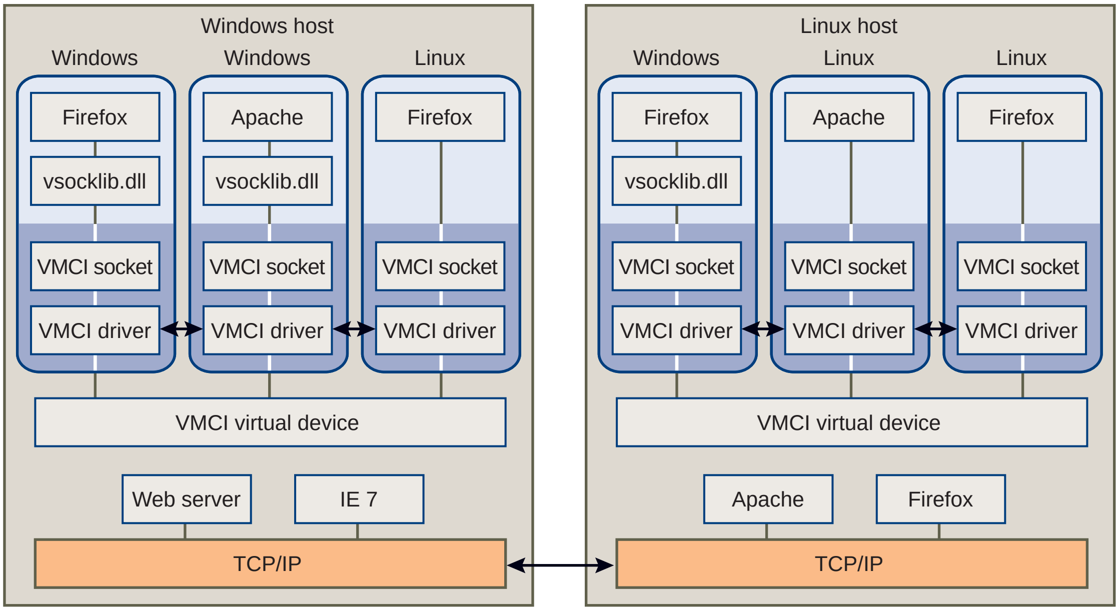
- Implement network-based communication for off-the-network virtual machines
- Improve the privacy of data transmission on hosted virtual machines
- Increase intra-host performance for socket-modified applications
- Provide an alternative data path for management of guest virtual machines
- Improve efficiency of database-backed applications seeking guest-to-guest for data
- Implement a fast host-guest file system

Web Access with Stream VMCI Sockets

Figure 1-1 shows an example of two VMware Workstation hosts, one Windows based and the other Linux based. On each host, modified Firefox browsers on Windows and Linux virtual machines are communicating with a modified Apache server on a separate virtual machine through VMCI sockets. Meanwhile, a Web browser on each host is communicating with a Web server on the other host using standard networking through TCP/IP sockets.

VMware does not provide modified versions of the third-party applications shown in these diagrams. However, open source versions of Firefox and Apache are available.

Figure 1-1. VMware Hosts with Stream VMCI Sockets in Guests



When the Firefox browsers on Linux and Windows request a connection to the Apache Web server, the VMCI sockets layer creates a socket endpoint and establishes a connection through the VMCI driver and virtual device. The VMCI sockets layer on the system with Apache receives the connection and provides an accepted socket through the socket on which Apache was listening.

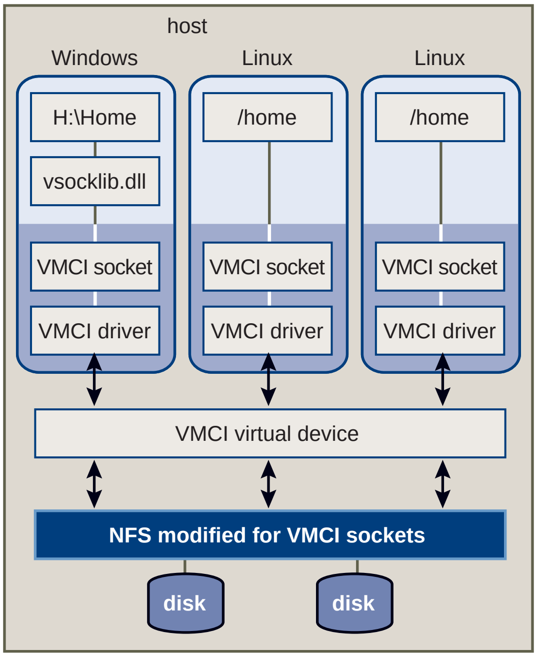
Meanwhile, unmodified Web browsers on the physical machines (Windows host and Linux host) are sending requests to each other's Web servers over a standard TCP/IP network connection. If guest operating systems needed to access the Web outside the physical machine, they must use different (unmodified) Web browsers or have a fallback capability outside of VMCI sockets.

Network Storage with Datagram VMCI Sockets

Figure 1-2 shows an example of a VMware host acting as the NFS server for the home directories of its three clients: a Windows guest and two Linux guests. NFS uses datagram sockets for file I/O. The NFS code on the VMware host must be slightly modified to use VMCI sockets instead of UDP datagrams.

VMware does not provide modified versions of the third-party applications shown in these diagrams. However, an open source version of NFS is available.

Figure 1-2. VMware Host with Datagram VMCI Sockets for NFS in Guests



Porting to VMCI Sockets 2

This chapter includes the following topics:

- "Porting Existing Socket Applications" on page 11
- "Communicating Between Guests" on page 12
- "Communicating Between Guest and Host" on page 12

Porting Existing Socket Applications

Modifying existing socket implementations is straightforward. This chapter describes the lines of code you must change.

Include a New Header File

To obtain the definitions for VMCI sockets, include the `vmci_sockets.h` header file.

```
#include "vmci_sockets.h"
```

Change AF_INET to VMCI Sockets

Call `VMCISock_GetAFValue()` to obtain the VMCI address family. Declare structure `sockaddr_vm` instead of `sockaddr_in`. In the `socket()` call, replace the `AF_INET` address family with the VMCI address family.

When the client creates a connection, instead of providing an IP address to choose its server, the client must provide the context ID (CID) of a virtual machine or host. An application running on a virtual machine uses the local context ID for `bind()` and a remote context ID for `connect()`.

Obtain the CID

In virtual hardware version 6 (Workstation 6.0.x releases), the VMCI virtual device is not present by default. After you upgrade a virtual machine's virtual hardware to version 7, the following line appears in the `.vmx` configuration file, and when the virtual machine powers on, a new `vmci0.id` line also appears there.

```
vmci0.present = "TRUE"
```

In virtual hardware version 7 (Workstation 6.5 releases), the VMCI virtual device is present by default. When you create a virtual machine, the `.vmx` configuration file contains lines specifying PCI slot number and the ID of the VMCI device. On the `vmci0.id` line, CID is the number in double quotes.

```
vmci0.pciSlotNumber = "36"
vmci0.id = "1066538581"
```

The VMCISock_GetLocalCID() Function

For convenience, you can call the `VMCISock_GetLocalCID()` function to obtain the local system's CID. This function works on both host server and guest virtual machines. The VMware host usually has CID = 2.

Connection-Oriented Stream Socket

To establish a stream socket, include these declarations and calls, and replace AF_INET with afVMCI, as set by VMCISock_GetAFValue().

```
int sockfd_stream;
int afVMCI = VMCISock_GetAFValue();
if ((sockfd_stream = socket(afVMCI, SOCK_STREAM, 0)) == -1) {
    perror(“Socket stream”);
}
```

Connectionless Datagram Socket

To establish a datagram socket, include these declarations and calls:

```
int sockfd_dgram;
int afVMCI = VMCISock_GetAFValue();
if ((sockfd_dgram = socket(afVMCI, SOCK_DGRAM, 0)) == -1) {
    perror(“Socket datagram”);
}
```

Initializing the Address Structure

To initialize the address structure passed to bind(), insert these source code statements, where sockaddr_vm for VMCI sockets replaces sockaddr_in for network sockets.

```
struct sockaddr_vm my_addr = {0};
my_addr.svm_family = afVMCI;
my_addr.svm_cid = VMADDR_CID_ANY;
my_addr.svm_port = VMADDR_PORT_ANY;
```

The first line declares my_addr as a sockaddr_vm structure and initializes it with zeroes. AF_INET replaces afVMCI. Both VMADDR_CID_ANY and VMADDR_PORT_ANY are predefined so that at runtime, the server can fill in the appropriate CID and port values during a bind operation. The initiating side of the connection, the client, must provide the CID and port, instead of VMADDR_CID_ANY and VMADDR_PORT_ANY.

Communicating Between Guests

To communicate between two guest virtual machines on the same host, you can establish a VMCI sockets connection of either the SOCK_STREAM or the SOCK_DGRAM socket type.

VMCI Sockets and Networking

If limited network access is sufficient for a virtual machine, you could replace TCP networking with VMCI sockets, thereby saving memory and processor bandwidth by disabling the network stack. If networking is enabled, as it typically is, VMCI sockets can still make some operations run faster.

Setting Up a Networkless Guest

You can install a virtual machine without any networking packages, so it cannot connect to the network. The system image of a network-free operating system is likely to be small, and isolation is a security advantage, at the expense of convenience. Install network-free systems as a networkless guest. After you upgrade VMware Tools, you can use VMCI sockets to communicate with the networkless guest.

You create a networkless guest with the option “Do not use a network connection” in the Workstation wizard. Alternatively, you can transform a network-capable guest into a networkless guest by removing all its virtual networking devices in the Workstation UI.

Communicating Between Guest and Host

To communicate between a guest virtual machine and its host, establish a VMCI sockets connection using the SOCK_DGRAM socket type, or on product platforms that support it (most do), the SOCK_STREAM socket type.

Creating Stream VMCI Sockets 3

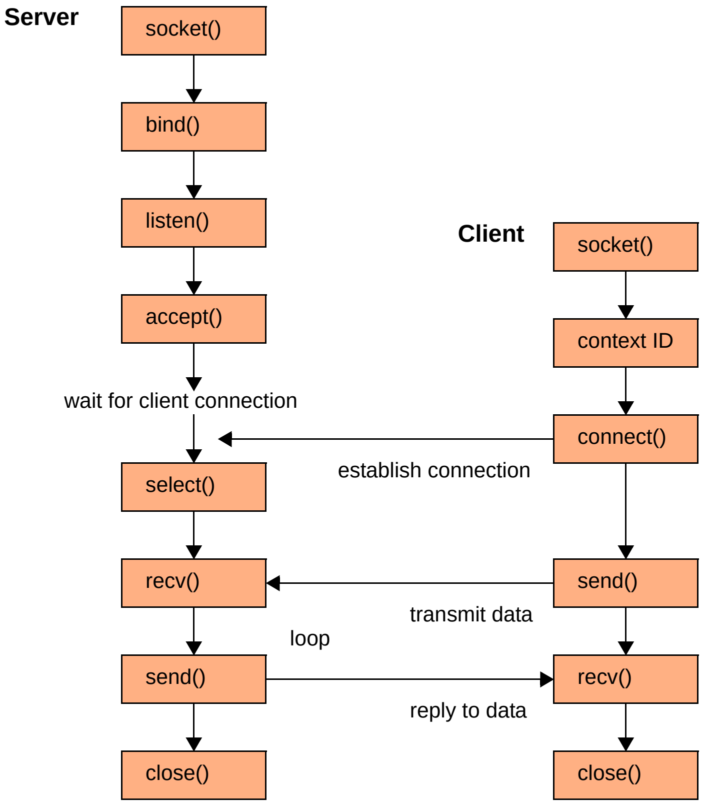
This chapter describes the details of creating VMCI sockets to replace TCP stream sockets.

- "Preparing the Server for a Connection" on page 14
- "Having the Client Request a Connection" on page 17

Stream VMCI Sockets

The flowchart in Figure 3-1 shows how to establish connection-oriented sockets on the server and client.

Figure 3-1. Connection-Oriented Stream Sockets



With VMCI sockets and TCP sockets, the server waits for the client to establish a connection. After connecting, the server and client communicate through the attached socket. In VMCI sockets, a virtual socket can have only two endpoints, and the server cannot initiate a connection to the client. In TCP sockets, more than two endpoints are possible, though rare, and the server can initiate connections. Otherwise, the protocols are identical.

Preparing the Server for a Connection

At the top of your application, include `vmci_sockets.h` and declare a constant for the socket buffer size. In the example below, `BUFSIZE` defines the socket buffer size. The number 4096 is a good choice for efficiency on multiple platforms. It is not based on the size of a TCP packet, which is usually smaller.

```
#include "vmci_sockets.h"
#define BUFSIZE 4096
```

To compile on Windows, you must also call the Winsock `WSAStartup()` function.

```
err = WSAStartup(versionRequested, &wsaData);
if (err != 0) {
    printf(stderr, "Could not register with Winsock DLL.\n");
    goto cleanup;
}
```

This is not necessary on non-Windows systems.

Socket() Function

In a VMCI sockets application, obtain the new address family (domain) to replace `AF_INET`.

```
int afVMCI = VMCISock_GetAFValue();
if ((sockfd = socket(afVMCI, SOCK_STREAM, 0)) == -1) {
      perror("socket");
      goto cleanup;
}
```

`VMCISock_GetAFValue()` returns a descriptor for the VMCI sockets address family if available.

Set and Get Socket Options

VMCI sockets allows you to set the minimum, maximum, and default size of communicating stream buffers. Names for the three options are:

- `SO_VMCI_BUFFER_SIZE` – Default size of communicating buffers; 65536 bytes if not set.
- `SO_VMCI_BUFFER_MIN_SIZE` – Minimum size of communicating buffers; defaults to 128 bytes.
- `SO_VMCI_BUFFER_MAX_SIZE` – Maximum size of communicating buffers; defaults to 262144 bytes.

To set a new value for a socket option, call the `setsockopt()` function. To get a value, call `getsockopt()`.

For example, to halve the size of the communications buffers from 65536 to 32768, and verify that the setting took effect, insert the following code:

```
uint64 setBuf = 32768, getBuf;
/* reduce buffer to above size and check */
if (setsockopt(sockfd, afVMCI, SO_VMCI_BUFFER_SIZE, (void *)&setBuf, sizeof setBuf) == -1) {
    perror(“setsockopt”);
    goto close;
}
if (getsockopt(sockfd, afVMCI, SO_VMCI_BUFFER_SIZE, (void *)&getBuf, sizeof getBuf) == -1) {
    perror(“getsockopt”);
    goto close;
}
if (getBuf != setBuf) {
    printf(stderr, “SO_VMCI_BUFFER_SIZE not set to size requested.\n”);
    goto close;
}
```

Parameters `setBuf` and `getBuf` must be declared 64 bit, even on 32-bit systems.

To have an effect, socket options must be set before establishing a connection. The buffer size is negotiated before the connection is established and stays consistent until the connection is closed. For a server socket, set options before any client establishes a connection. To be sure that this applies to all sockets, set options before calling `listen()`. For a client socket, set options before calling `connect()`.

Bind() Function

This bind() call associates the stream socket with the network settings in the sockaddr_vm structure, instead of the sockaddr_in structure.

```
struct sockaddr_vm my_addr = {0};
my_addr.svm_family = afVMCI;
my_addr.svm_cid = VMADDR_CID_ANY;
my_addr.svm_port = VMADDR_PORT_ANY;
if (bind(sockfd, (struct sockaddr *) &my_addr, sizeof my_addr) == -1) {
      perror("bind");
      goto close;
}
```

The sockaddr_vm structure contains an element for the context ID (CID), which specifies the virtual machine. For the client this is the local CID. For the server (listener), this could be any connecting virtual machine. Both VMADDR_CID_ANY and VMADDR_PORT_ANY are predefined so that at bind or connection time, the appropriate CID and port number are filled in from the client. VMADDR_CID_ANY is replaced with the CID of the virtual machine and VMADDR_PORT_ANY provides an ephemeral port from the nonreserved range (>= 1024).

The client (connector) can obtain its local CID by calling VMCISock_GetLocalCID().

The bind() function is the same as for a regular TCP sockets application.

Listen() Function

The listen() call prepares to accept incoming client connections. The BACKLOG macro predefines the number of incoming connection requests that the system accepts before rejecting new ones. This function is the same as listen() in a regular TCP sockets application.

```
if (listen(sockfd, BACKLOG) == -1) {
      perror("listen");
      goto close;
}
```

Accept() Function

The accept() call waits indefinitely for an incoming connection to arrive, creating a new socket (and stream descriptor newfd) when it does. The structure their_addr gets filled with connection information.

```
struct sockaddr_vm their_addr;
socklen_t their_addr_len = sizeof their_addr;
if ((newfd = accept(sockfd, (struct sockaddr *) &their_addr, &their_addr_len)) == -1) {
      perror("accept");
      goto close;
}
```

Select() Function

The select() call enables a process to wait for events on multiple file descriptors simultaneously. This function hibernates, waking up the process when an event occurs. You can specify a timeout in seconds or microseconds. After timeout, the function returns zero. You can specify the read, write, and exception file descriptors as NULL if the program can safely ignore them.

```
if ((select(nfds, &readfd, &writefds, &exceptfds, &timeout) == -1) {
      perror("select");
      goto close;
}
```

Recv() Function

The recv() call reads data from the client application. The server and client can communicate the length of data transmitted, or the server can terminate its recv() loop when the client closes its connection.

```
char recv_buf[BUFSIZE];
if ((numbytes = recv(sockfd, recv_buf, sizeof recv_buf, 0)) == -1) {
      perror("recv");
      goto close;
}
```

Send() Function

The send() call writes data to the client application. Server and client must communicate the length of data transmitted, or agree beforehand on a size. Often the server sends only flow control information to the client.

```
char send_buf[BUFSIZE];
if ((numbytes = send(newfd, send_buf, sizeof send_buf, 0)) == -1) {
      perror("send");
      goto close;
}
```

Close() Function

Given the original socket descriptor obtained from the socket() call, the close() call closes the socket and terminates the connection if it is still open. Some server applications close immediately after receiving client data, while others wait for additional connections. To compile on Windows, you must call the Winsock closesocket() instead of close().

```
#ifdef _WIN32
    return closesocket(sockfd);
#else
    return close(sockfd);
#endif
```

The shutdown() function is like close(), but shuts down the connection.

Poll() Information

Not all socket-based networking programs use poll(), but if they do, no changes are required. The poll() function is like select(). See "Select() Function" on page 15 for related information.

Read() and Write()

The read() and write() socket calls are provided for convenience. They provide the same functionality as recv() and send().

Getsockname() Function

The getsockname() function retrieves the local address associated with a socket.

```
my_addr_size = sizeof my_addr;
if (getsockname(sockfd, (struct sockaddr *) &my_addr, &my_addr_size) == -1) {
    perror("getsockname");
    goto close;
}
```

Having the Client Request a Connection

At the top of your application, include vmci_sockets.h and declare a constant for the socket buffer size. In the example below, BUFSIZE defines the socket buffer size. It is not based on the size of a TCP packet.

```
#include "vmci_sockets.h"
#define BUFSIZE 4096
```

To compile on Windows, you must call the Winsock WSAStartup() function. See "Preparing the Server for a Connection" on page 14 for sample code.

Socket() Function

In a VMCI sockets application, obtain the new address family (domain) to replace AF_INET.

```
int afVMCI = VMCISock_GetAFValue();
if ((sockfd = socket(afVMCI, SOCK_STREAM, 0)) == -1) {
      perror("socket");
      goto exit;
}
```

VMCISock_GetAFValue() returns a descriptor for the VMCI sockets address family if available.

Connect() Function

The connect() call requests a socket connection to the server specified by CID in the sockaddr_vm structure, instead of by the IP address in the sockaddr_in structure.

```
struct sockaddr_vm their_addr = {0};
their_addr.svm_family = afVMCI;
their_addr.svm_cid = SERVER_CID;
their_addr.svm_port = SERVER_PORT;
if ((connect(sockfd, (struct sockaddr *) &their_addr, sizeof their_addr)) == -1) {
      perror("connect");
      goto close;
}
```

The sockaddr_vm structure contains an element for the context ID (CID) to specify the virtual machine or host. The client making a connection should provide the CID of a remote virtual machine or host.

The port number is arbitrary, although server (listener) and client (connector) must use the same number, which must designate a port not already in use. Only privileged processes can use ports < 1024.

The connect() call allows you to use send() and recv() functions instead of sendto() and recvfrom(). The connect() call is not necessary for datagram sockets.

Send() Function

The send() call writes data to the server application. The client and server can communicate the length of data transmitted, or the server can terminate its recv() loop when the client closes its connection.

```
char send_buf[BUFSIZE];
/* Initialize send_buf with your data. */
if ((numbytes = send(sockfd, send_buf, sizeof send_buf, 0)) == -1) {
      perror("send");
      goto close;
}
```

Recv() Function

The `recv()` call reads data from the server application. Sometimes the server sends flow control information, so the client must be prepared to receive it. Use the same socket descriptor as for `send()`.

```
char recv_buf[BUFSIZE];
if ((numbytes = recv(sockfd, recv_buf, sizeof recv_buf, 0)) == -1) {
      perror("recv");
      goto close;
}
```

Close() Function

The `close()` call shuts down a connection, given the original socket descriptor obtained from the `socket()` function. To compile on Windows, you must call the Winsock `closesocket()` instead of `close()`.

```
#ifdef _WIN32
    return closesocket(sockfd);
#else
    return close(sockfd);
#endif
```

Poll() Information

Not all socket-based networking programs use `poll()`, but if they do, no changes are required.

Read() and Write()

The `read()` and `write()` socket calls are provided for convenience. They provide the same functionality as `recv()` and `send()`.

Creating Datagram VMCI Sockets 4

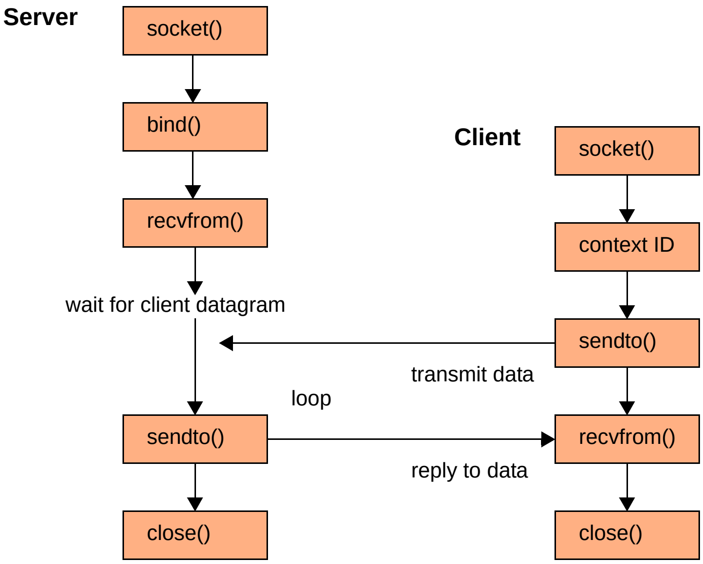
This chapter describes the details of creating VMCI sockets to replace UDP sockets.

- "Preparing the Server for a Connection" on page 20
- "Having the Client Request a Connection" on page 21

Datagram VMCI Sockets

The flowchart in Figure 4-1 shows how to establish connectionless sockets on the server and client.

Figure 4-1. Connectionless Datagram Sockets



In UDP sockets, the server waits for the client to transmit, and accepts datagrams. In VMCI sockets, the server and client communicate similarly with datagrams.

Preparing the Server for a Connection

At the top of your application, include vmci_sockets.h and declare a constant for the socket buffer size. In the example below, BUFSIZE defines the socket buffer size. The number 4096 is a good choice for efficiency on multiple platforms. It is not based on the size of a UDP datagram.

```
#include "vmci_sockets.h"
#define BUFSIZE 4096
```

To compile on Windows, you must call the Winsock WSAStartup() function.

```
err = WSAStartup(versionRequested, &wsaData);
if (err != 0) {
    printf(stderr, "Could not register with Winsock DLL.\n");
    goto exit;
}
```

This is not necessary on non-Windows systems.

Socket() Function

To alter a UDP socket program for VMCI sockets, obtain the new address family to replace AF_INET.

```
int afVMCI = VMCISock_GetAFValue();
if ((sockfd_dgram = socket(afVMCI, SOCK_DGRAM, 0)) == -1) {
    perror("socket");
    goto exit;
}
```

VMCISock_GetAFValue() returns a descriptor for the VMCI sockets address family if available.

This call is similar to the one for stream sockets, but has SOCK_DGRAM instead of SOCK_STREAM.

Socket Options

Currently VMCI sockets offers no options for datagram connections.

Bind() Function

The bind() call associates the datagram socket with the network settings in the sockaddr_vm structure, instead of the sockaddr_in structure.

```
struct sockaddr_vm my_addr = {0};
my_addr.svm_family = afVMCI;
my_addr.svm_cid = VMADDR_CID_ANY;
my_addr.svm_port = VMADDR_PORT_ANY;
if (bind(sockfd, (struct sockaddr *) &my_addr, sizeof my_addr) == -1) {
      perror("bind");
      goto close;
}
```

The sockaddr_vm structure contains an element for the context ID (CID) to specify the virtual machine. For the client (connector) this is the local CID. For the server (listener), it could be any connecting virtual machine. VMADDR_CID_ANY and VMADDR_PORT_ANY are predefined so that at bind or connection time, the appropriate CID and port number are filled in from the client. VMADDR_CID_ANY is replaced with the CID of the virtual machine and VMADDR_PORT_ANY provides an ephemeral port from the nonreserved range (>= 1024).

The client (connector) can obtain its local CID by calling VMCISock_GetLocalCID().

The VMCI sockets bind() function is the same as for a UDP datagram application.

Getsockname() Function

The `getsockname()` function retrieves the local address associated with a socket.

```
my_addr_size = sizeof my_addr;
if (getsockname(sockfd, (struct sockaddr *) &my_addr, &my_addr_size) == -1) {
    perror("getsockname");
    goto close;
}
```

Recvfrom() Function

The `recvfrom()` call reads data from the client application. Server and client can communicate the length of data transmitted, or the server can terminate its `recvfrom()` loop when the client closes its connection.

```
if ((numbytes = recvfrom(sockfd, buf, sizeof buf, 0,
        (struct sockaddr *) &their_addr, &my_addr_size)) == -1) {
    perror("recvfrom");
    goto close;
}
```

Sendto() Function

The `sendto()` call optionally writes data back to the client application. See "Sendto() Function" on page 22.

Close() Function

The `close()` call shuts down transmission, given the original socket descriptor obtained from the `socket()` call. Some server applications close immediately after receiving client data, while others wait for additional connections. To compile on Windows, you must call the Winsock `closesocket()` instead of `close()`.

```
#ifdef _WIN32
    return closesocket(sockfd);
#else
    return close(sockfd);
#endif
```

Having the Client Request a Connection

At the top of your application, include `vmci_sockets.h` and declare a constant for buffer size. This does not have to be based on the size of a UDP datagram.

```
#include "vmci_sockets.h"
#define BUFSIZE 4096
```

To compile on Windows, you must call the Winsock `WSAStartup()` function. See "Preparing the Server for a Connection" on page 20 for sample code.

Socket() Function

To alter a UDP socket program for VMCI sockets, obtain the new address family to replace `AF_INET`.

```
int afVMCI = VMCISock_GetAFValue();
if ((sockfd = socket(afVMCI, SOCK_DGRAM, 0)) == -1) {
    perror("socket");
    goto exit;
}
```

Sendto() Function

Because this is a connectionless protocol, you pass the socket address structure their_addr as a parameter to the sendto() call.

```
struct sockaddr_vm their_addr = {0};
their_addr.svm_family = afVMCI;
their_addr.svm_cid = SERVER_CID;
their_addr.svm_port = SERVER_PORT;
if ((numbytes = sendto(sockfd, buf, BUFIZE, 0,
        (struct sockaddr *) &their_addr, sizeof their_addr)) == -1) {
    perror("sendto");
    goto close;
}
```

The sockaddr_vm structure contains an element for the CID to specify the virtual machine. For the client making a connection, the VMCISock_GetLocalCID() function returns the CID of the virtual machine.

The port number is arbitrary, although the server (listener) and client (connector) must use the same number, which must designate a port not already in use. Only privileged processes can use ports < 1024.

Connect() and Send()

Even with this connectionless protocol, applications can call the connect() function once to set the address, and call the send() function repeatedly without having to specify the sendto() address each time.

```
if ((connect(sockfd, (struct sockaddr *) &their_addr, sizeof their_addr)) == -1) {
    perror("connect");
    goto close;
}
if ((numbytes = send(sockfd, send_buf, BUFSIZE, 0)) == -1) {
    perror("send");
    goto close;
}
```

Recvfrom() Function

The recvfrom() call optionally reads data from the server application. See “Recvfrom() Function” on page 21.

Close() Function

The close() call shuts down a connection, given the original socket descriptor obtained from the socket() function. To compile on Windows, call the Winsock closesocket(), as shown in “Close() Function” on page 21.

Appendix: Learning More About Sockets

This appendix introduces Internet sockets and provides pointers to further information.

- "About Berkeley Sockets and Winsock" on page 23
- "Short Introduction to Sockets" on page 24

About Berkeley Sockets and Winsock

A socket is a communications endpoint with a name and address in a network. Sockets were made famous by their implementation in Berkeley UNIX, and made universal by their incorporation into Windows.

Most socket-based applications employ a client-server approach to communications. Rather than trying to start two network applications simultaneously, one application tries to make itself always available (the server or the provider) while another requests services as needed (the client or the consumer).

VMCI sockets are designed to use the client-server approach but, unlike TCP sockets, they do not support multiple endpoints simultaneously initiating connections with one another.

Data going over a socket can be in any format, and travel in either direction.

Many people are confused by `AF_INET` as opposed to `PF_INET`. Linux defines them as identical. This manual uses AF only. AF means address family, while PF means protocol family. As designed, a single protocol family could support multiple address families. However as implemented, no protocol family ever supported more than one address family. For Internet Protocol version 6 (IPv6), `AF_INET6` is synonymous with `PF_INET6`.

WinSock includes virtually all of the Berkeley sockets API, as well as additional WSA functions to cope with cooperative multitasking and the event-driven programming model of Windows.

Programmers use stream sockets for their high reliability, and datagram sockets for speed and low overhead.

Trade Press Books

Internetworking with TCP/IP, Volume 3: Client-Server Programming and Applications, Linux/Posix Sockets Version, by Douglas E. Comer and David L. Stevens, 601 pages, Prentice-Hall, 2000.

UNIX Network Programming, Volume 1: The Sockets Networking API, Third Edition, by W. Richard Stevens (RIP), Bill Fenner, and Andrew M. Rudoff, 1024 pages, Addison-Wesley, 2003.

Berkeley Sockets

Wikipedia offers an excellent overview of the history and design of Berkeley sockets.

For reference information about Berkeley sockets, locate a Linux system with manual pages installed, and type **`man socket`**. You should be able to find both socket(2) and socket(7) reference pages.

Microsoft Winsock

The *Winsock Programmer's FAQ* is an excellent introduction to Windows sockets. Currently it is hosted by the http://tangentsoft.net Web site.

For complete reference information about Winsock, refer to the public MSDN Web site.

Short Introduction to Sockets

Network I/O is similar to file I/O, although network I/O requires not only a file descriptor sufficient for identifying a file, but also sufficient information for network communication.

Berkeley sockets support both UNIX domain sockets (on the same system) and Internet domain sockets, also called TCP/IP (transmission control protocol) or UDP/IP (user datagram protocol).

Socket Addresses

The socket address specifies the communication family. UNIX domain sockets are defined as sockaddr_un. Internet domain sockets are defined as sockaddr_in or sockaddr_in6 for IPv6.

```
struct sockaddr_in {
    short          sin_family;    /* AF_INET */
    u_short        sin_port;      /* port number */
    struct in_addr sin_addr;      /* Internet address */
    char           sin_zero[8];   /* unused */
};
```

Socket() System Call

The socket() system call creates one end of the socket.

```
int socket(int <family>, int <type>, int <protocol>);
```

- The first parameter specifies the communication family, AF_UNIX or AF_INET.
- The second parameter specifies the socket type, SOCK_STREAM or SOCK_DGRAM.
- The third parameter is usually zero because communication families usually have only one protocol.

The socket() system call returns the socket descriptor, a small integer that is similar to the file descriptor used in other system calls. For example:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
    int sockfd;
    sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
```

Bind() System Call

The bind() system call associates an address with the socket descriptor.

```
int bind(int sockfd, struct sockaddr *myaddr, int addrlen);
```

- The first parameter is the socket descriptor from the socket() call, sockfd.
- The second parameter is a pointer to the socket address structure, which is generalized for different protocols. The sockaddr structure is defined in <sys/socket.h>.
- The third parameter is the length of the sockaddr structure, because it can vary.

In the sockaddr structure for IPv4 sockets, the first field specifies AF_INET. The second field sin_port can be any integer > 5000. Lower port numbers are reserved for specific services. The third field in_addr is the Internet address in dotted-quad notation. For the server, you can use the constant INADDR_ANY to tell the system to accept a connection on any Internet interface for the system. Conversion functions htons() and htonl() are for hardware independence. For example:

```
#define SERV_PORT 5432
    struct sockaddr_in serv_addr;
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(SERV_PORT);
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
```

Listen() System Call

The listen() system call prepares a connection-oriented server to accept client connections.

```
int listen(int sockfd, struct <backlog>);
```

- The first parameter is the socket descriptor from the socket() call, sockfd.
- The second parameter specifies the number of requests that the system queues before it executes the accept() system call. Higher and lower values of <backlog> trade off high efficiency for low latency.

For example:

```
    listen(sockfd, 5);
```

Accept() System Call

The accept() system call initiates communications between a connection-oriented server and the client.

```
int accept(int sockfd, struct sockaddr *cli_addr, int addrlen);
```

- The first parameter is the socket descriptor from the socket() call, sockfd.
- The second parameter is the client's sockaddr address, to be filled in.
- The third parameter is the length of the client's sockaddr structure.

Generally programs call accept() inside an infinite loop, forking a new process for each accepted connection. After accept() returns with client address, the server is ready to accept data.

For example:

```
for( ; ; ) {
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, sizeof(cli_addr));
    if (fork() = 0)  {
        close(sockfd);
        /*
        * read and write data over the network
        * (code missing)
        */
        exit (0);
    }
    close(newsockfd);
}
```

Connect() System Call

On the client, the connect() system call establishes a connection to the server.

```
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
```

- The first parameter is the socket descriptor from the socket() call, sockfd.
- The second parameter is the server's sockaddr address, to be filled in.
- The third parameter is the length of the server's sockaddr structure.

This is similar to the `accept()` system call, except that the client does not have to bind a local address to the socket descriptor before calling `connect()`. The server address pointed to by `srv_addr` must exist.

For example:

```
#define SERV_PORT 5432
    unsigned long inet_addr(char *ptr);
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(SERV_PORT):
    serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
    connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr));
```

Socket Read and Write

Sockets use the same read and write system calls as for file I/O.

- The first parameter is the socket descriptor from the `socket()` call, `sockfd`.
- The second parameter is the read or write buffer.
- The third parameter is the number of bytes to read.

Unlike file I/O, a read or write system call on a stream socket may result in fewer bytes than requested. It is the programmer's responsibility to account for varying number of bytes read or written on the socket.

For example:

```
nleft = nbytes;
while (nleft > 0) {
    if ((nread = read(sockfd, buf, nleft)) < 0)
        return(nread); /* error */
    else if (nread == 0)
        break; /* EOF */
    /* nread > 0. update nleft and buf pointer */
    nleft - = nread;
    buf += nread;
}
```

Index