

VMCI Sockets Programming Guide

VMware ESXi 5.5

VMware Workstation 9.x

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-001166-00

vmware[®]

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

Copyright © 2008–2013 VMware, Inc. All rights reserved. This product is protected by U.S. and international copyright and intellectual property laws. VMware products are covered by one or more patents listed at <http://www.vmware.com/go/patents>.

VMware is a registered trademark or trademark of VMware, Inc. in the United States and/or other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies.

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

Contents

About This Book	5
1 About VMCI Sockets	7
Introduction to VMCI Sockets	7
Previous VMCI Releases	7
How VMCI Sockets Work	7
Persistence of Sockets	8
Socket Programming	8
Features in Specific VMware Releases	8
Enabling and Finding VMCI Sockets	8
Location of Include File for C Programs	8
Security of VMCI Sockets	9
Use Cases for VMCI Sockets	9
RabbitMQ with Stream VMCI Sockets	9
Network Storage with Datagram VMCI Sockets	10
2 Porting to VMCI Sockets	11
Porting Existing Socket Applications	11
Include a New Header File	11
Change AF_INET to VMCI Sockets	11
Obtain the CID	11
The VMCIsock_GetLocalCID() Function	11
Connection-Oriented Stream Socket	12
Connectionless Datagram Socket	12
Initializing the Address Structure	12
Communicating Between Host and Guest	12
VMCI Sockets and Networking	12
Setting Up a Networkless Guest	12
3 Creating Stream VMCI Sockets	13
Stream VMCI Sockets	13
Preparing the Server for a Connection	14
Socket() Function	14
Set and Get Socket Options	14
Bind() Function	15
Listen() Function	15
Accept() Function	15
Select() Function	15
Recv() Function	16
Send() Function	16
Close() Function	16
Poll() Information	16
Read() and Write()	16
Getsockname() Function	16
Having the Client Request a Connection	17
Socket() Function	17
Connect() Function	17
Send() Function	17

Recv() Function	18
Close() Function	18
Poll() Information	18
Read() and Write()	18
4	Creating Datagram VMCI Sockets
Datagram VMCI Sockets	19
Preparing the Server for a Connection	20
Socket() Function	20
Socket Options	20
Bind() Function	20
Getsockname() Function	21
Recvfrom() Function	21
Sendto() Function	21
Close() Function	21
Having the Client Request a Connection	21
Socket() Function	21
Sendto() Function	22
Connect() and Send()	22
Recvfrom() Function	22
Close() Function	22
5	Security of the VMCI Device
Interfaces for VMCI Settings	23
VMCI Device Always Enabled	23
VMCI and Hardware Version	23
Authentication	23
Isolation Options in VMX	24
Isolation of Virtual Machines	24
Isolation in Workstation	24
Isolation in ESX/ESXi	24
Trusted VMCI Sockets	24
	Appendix: Learning More About Sockets
About Berkeley Sockets and Winsock	25
Trade Press Books	25
Berkeley Sockets	25
Microsoft Winsock	26
Short Introduction to Sockets	26
Socket Addresses	26
Socket() System Call	26
Bind() System Call	26
Listen() System Call	27
Accept() System Call	27
Connect() System Call	27
Socket Read and Write	28
	Index
	29

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 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
20130830	Manual slightly revised for the ESXi 5.5 release.
20120719	Guest-to-guest communication dedocumented for the ESXi 5.1 release.
20120105	Windows header file now in Program Files\Common Files\VMware\Drivers\vmci\sockets\include.
20110720	Manual revised for the Workstation 8.0 release and for the ESXi 5.0 release.
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 using VMCI sockets to create C or C++ networking applications for guest operating systems running on VMware hosts. 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 25 provides pointers to learning resources.

Document Feedback

VMware welcomes your suggestions for improving our documentation and search tools. 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>.

About VMCI Sockets

This chapter includes the following topics:

- [“Introduction to VMCI Sockets”](#) on page 7
- [“Features in Specific VMware Releases”](#) on page 8
- [“Enabling and Finding VMCI Sockets”](#) on page 8
- [“Use Cases for VMCI Sockets”](#) on page 9

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 25.

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 guest virtual machines and their 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 interfaces were discontinued in Workstation 6.5.

The VMCI sockets library was first released with Workstation 6.5 and Server 2.0 as a supported interface. 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™ 4 and vCenter™ Server 4 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 guest virtual machines to communicate the host on which they reside.

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 to processes on different systems, including 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, “ESXi host with Stream VMCI Sockets and RabbitMQ”](#) 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 between the host and a guest on VMware platform products. You could also use VMCI sockets for interprocess communications on a guest. You cannot use VMCI sockets between the host and a virtual machine running on a different host.

IMPORTANT To use VMCI sockets, virtual machines must be upgraded to VMware compatibility 7 (virtual hardware version 7), which was introduced in VMware Workstation 6.5 and supported in ESX/ESXi 4.0.

As of 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.

ESX/ESXi 4.x (vSphere 4) releases and later have complete user-level support for VMCI sockets. Datagram and stream sockets are supported between host and guests on both Linux and Windows. In the Workstation 7.x releases running on Windows hosts, only datagram sockets were supported.

In the ESXi 5.0 and Workstation 8.0 releases, it was announced that the guest to guest VMCI sockets feature would be discontinued. As of the ESXi 5.1 release, only host to guest VMCI sockets are allowed.

Enabling and Finding VMCI Sockets

For host to guest communication, VMCI is enabled on virtual machines with version 7 compatibility and later.

Location of Include File for C Programs

VMware Tools or another installer places the `vmci_sockets.h` include file in one of the following locations:

- Windows guests on Workstation 8.0 or later, and Windows hosts of Workstation 8.0 or later – `C:\Program Files\Common Files\VMware\Drivers\vmci\sockets\include`
- earlier Windows guests – `C:\Program Files\VMware\VMware Tools\VSock SDK\include`
- earlier Windows hosts – `C:\Program Files\VMware\VMware Workstation`
- Linux guests – `/usr/lib/vmware-tools/include/vmci`
- Linux hosts – `/usr/lib/vmware/include/vmci`
- ESX/ESXi hosts – Not installed on the system.

Security of VMCI Sockets

VMCI Sockets are more secure after elimination of guest to guest communications. For an overview of VMCI security, see [Chapter 5, “Security of the VMCI Device,”](#) on page 23.

Use Cases for VMCI Sockets

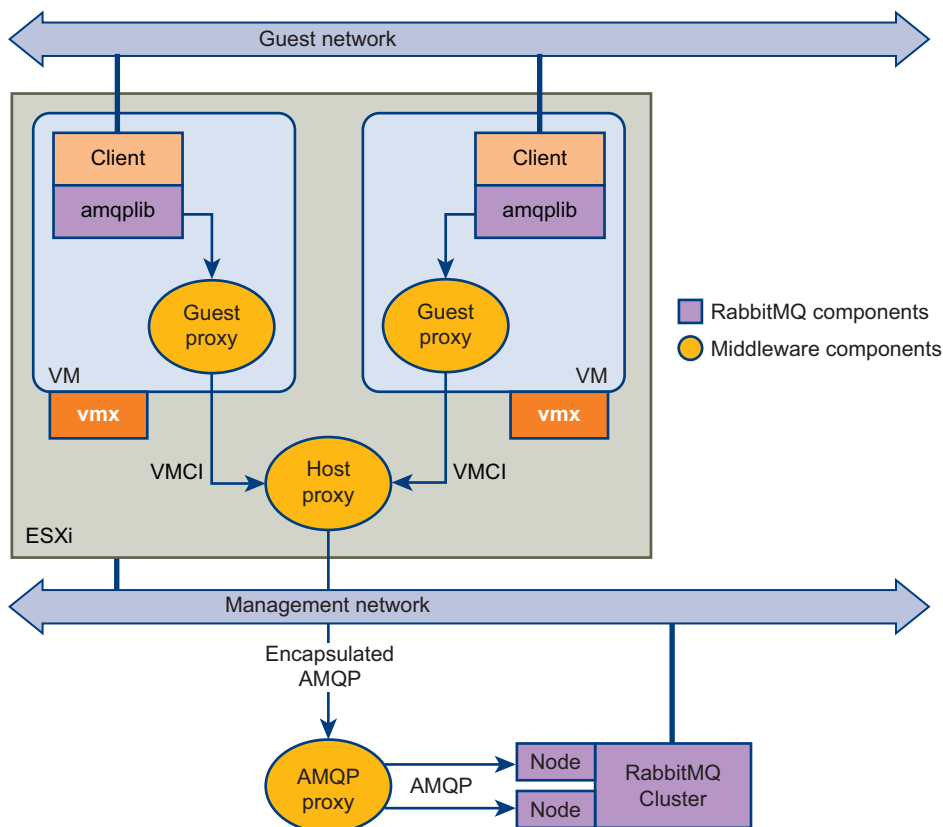
VMCI sockets can help with the following solutions:

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

RabbitMQ with Stream VMCI Sockets

On the ESXi host in [Figure 1-1](#), two virtual machines contain a message queuing client that communicates with a guest proxy through `amqp-lib`. Guest proxies communicate with the host proxy over VMCI Sockets, which treat each guest connection as a separate session. The host proxy multiplexes these sessions and communicates with the RabbitMQ proxy over a single TCP/IP socket, passing encapsulated AMQP. A RabbitMQ node runs on a virtual machine. The RabbitMQ cluster is a collection of RabbitMQ nodes that are assembled for reliability and scaling. The AMQP proxy splits-out multiplexed sessions of individual connections to a RabbitMQ node. All this takes place on the management network, reducing traffic on the guest network.

Figure 1-1. ESXi host with Stream VMCI Sockets and RabbitMQ



Originated in 2003, AMQP (advanced message queuing protocol) is an industry standard for passing business messages between applications and organizations. It is popular in the banking and finance industries.

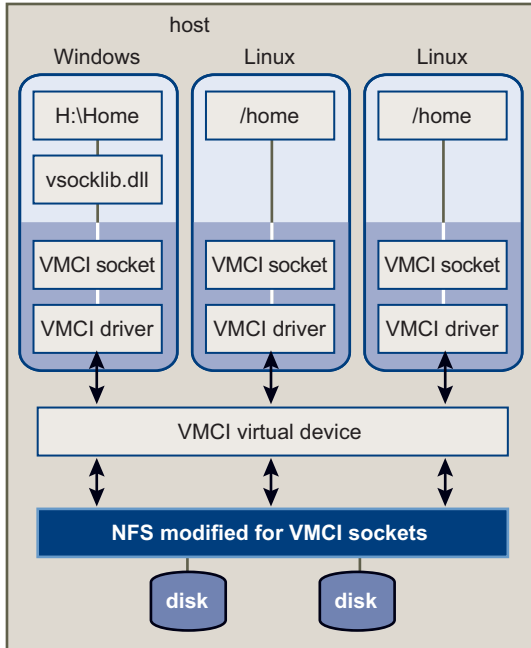
RabbitMQ software is an open-source message broker (or message-oriented middleware) that implements the AMQP standard. SpringSource currently develops and supports RabbitMQ.

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

This chapter includes the following topics:

- [“Porting Existing Socket Applications”](#) on page 11
- [“Communicating Between Host and Guest”](#) 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 the ESXi host and guest virtual machines, although the ESXi host always has `CID = 2`, even in a nested virtual machine (VM running in a VM).

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 Host and Guest

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.

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 installing VMware Tools, the host 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.

Creating Stream VMCI Sockets

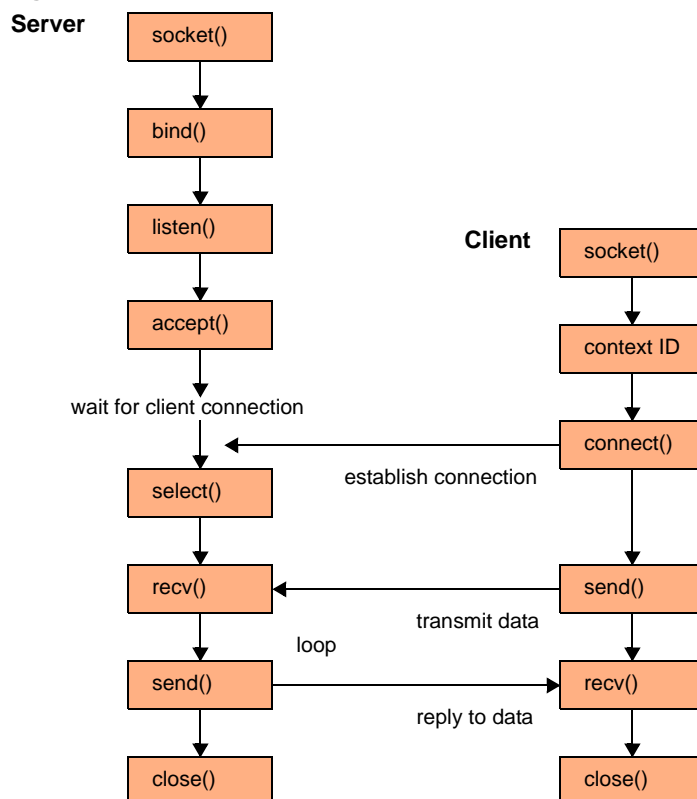
This chapter describes the details of creating VMCI sockets to replace TCP stream sockets.

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

`VMCSock_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 = AF_VMCI;
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

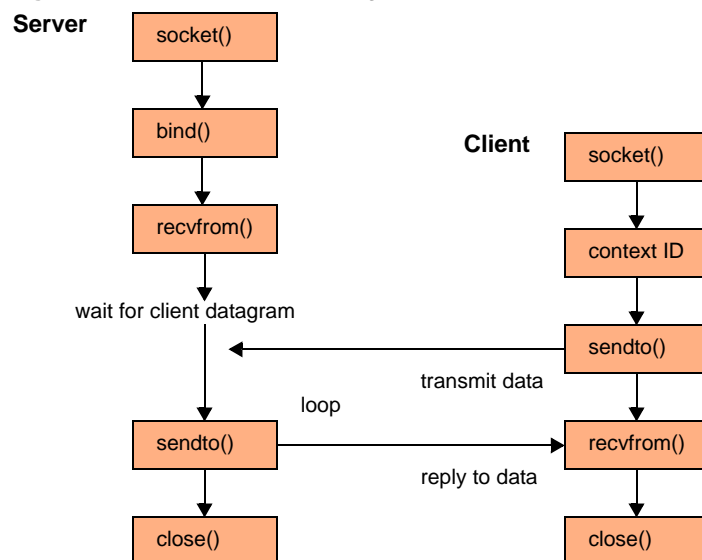
This chapter describes the details of creating VMCI sockets to replace UDP sockets.

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

`VMCSock_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 `VMCSock_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 = VMCSock_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 = AF_VMCI;
their_addr.svm_cid = SERVER_CID;
their_addr.svm_port = SERVER_PORT;
if ((numbytes = sendto(sockfd, buf, BUFSIZE, 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.

Security of the VMCI Device

This chapter provides background information about security of the VMCI device, especially about interfaces that are not part of the public VMCI Sockets API. VMCI Sockets are implemented on top of the VMCI device.

- [“Interfaces for VMCI Settings”](#) on page 23
- [“VMCI Device Always Enabled”](#) on page 23
- [“Isolation of Virtual Machines”](#) on page 24

Interfaces for VMCI Settings

VMCI is used primarily for communication between virtual machines and the hypervisor. Communication between virtual machines is not supported, but can be accomplished by use of normal TCP or UDP sockets.

On ESXi 5.1 and later, the **VMCI device > Enable VMCI between VMs** setting (if present) has no effect.

After Workstation 8.x, **Settings > Options > Guest Isolation > Enable VMCI** will be discontinued.

For older virtual hardware versions without VMCI, the hypervisor reverts to a “backdoor” mechanism for communication. However VMware services introduced in new products may have no backdoor fallback, so some services may require VMCI to work correctly.

VMCI Device Always Enabled

The VMCI device is always present in recently created VMware virtual machines, raising questions about the security implications of having a VMCI device.

VMCI and Hardware Version

Starting with VMware virtual hardware version 7, the VMCI device is enabled by default. Virtual machines upgraded from older hardware versions to version 7 acquire the VMCI device even if it was not present before upgrading the virtual hardware. The VMCI device cannot be removed. On most guest operating systems, VMware Tools should be installed to provide a VMCI device driver.

To address security concerns, VMware provided a method to restrict VMCI-based services that are available to a virtual machine. Services were restricted to a trusted subset of only the hypervisor-related services needed to run a virtual machine in isolation. Restricted was the default, as is now the only configuration.

Authentication

All VMCI communications are authenticated. The source (context ID) may not be spoofed. The VMCI facility implicitly authenticates any hypervisor service as being part of the trusted code base. VMCI does not provide fine grained authentication of communication endpoints, so applications must deal with fine grained authentication as a separate issue. It is the responsibility of applications running on top of VMCI to implement their own authentication mechanisms if necessary. VMCI ensures only that malicious software cannot spoof the source field in VMCI datagrams identifying the sending virtual machine.

Isolation Options in VMX

ESX/ESXi 4.0 to ESXi 5.0 provide `.vmx` options for VMCI isolation. As of ESXi 5.1, these options have no effect.

```
[vmci0.unrestricted = FALSE|TRUE]
```

When its `vmci.unrestricted` option was set `TRUE`, a virtual machine could communicate with all host endpoints and other virtual machines that had `vmci0.unrestricted` set `TRUE`.

```
[vmci0.domain = <domainName>]
```

(ESX/ESXi only) All virtual machines and host applications were members of the default domain (""), null string, by default. If the `vmci0.domain` option specified a non-default domain, then the virtual machine could communicate only with the hypervisor and other virtual machines in the same domain. This was to organize virtual machines into groups that could communicate with each other.

As of ESXi 5.1, or earlier when configured for restricted communication, the VMCI device has a security profile similar to any other device such as keyboard, video monitor, mouse, or motherboard. Guest communications depend on the VMCI applications running on the host. VMCI in itself does not expose any guest information.

Isolation of Virtual Machines

This section describes VMCI isolation mechanisms as they apply to VMware Workstation and ESXi hosts.

Isolation in Workstation

After Workstation 8.x, or earlier with it a marked isolated, virtual machine is allowed to interact only with hypervisor services (context ID = 0). This allows use of VMware Tools without any problems even for an isolated virtual machine. An isolated virtual machine is not allowed to interact with other virtual machines.

A virtual machine is isolated by default, but Workstation 8.x and earlier had a check box to remove its isolation.

Isolation in ESX/ESXi

ESX/ESXi 4.0 until ESXi 5.0 supported the ability to have several groups of virtual machines per physical host, where a virtual machine could see only the virtual machines that were a member of the same group. Groups were not hierarchical and could not overlap. Each host could belong to one or more VMCI domains, and guest virtual machines could see other virtual machines in the same domain, and the hypervisor context. Context IDs had to be unique across domains on the host. VMCI domains were specified in a virtual machine's `.vmx` file – no user interface was provided to manage VMCI domains.

As of ESXi 5.1, and earlier if marked isolated, a virtual machine has the same restrictions as for Workstation.

Trusted VMCI Sockets

VMCI device interfaces are not available to user-level processes, which must access it using VMCI Sockets.

The VMCI Sockets API permits some host applications to create trusted VMCI Sockets, which may be used for communication with isolated guest virtual machines. The mechanism for deciding whether a host application creates a trusted VMCI socket depends on the host operating system:

- Linux – A process with the capability `CAP_NET_ADMIN` can create trusted endpoints.
- ESXi – A system process with access privileges `dgram_vsocket_trusted` or `stream_vsocket_trusted` can create trusted datagram or stream sockets, respectively.
- Creation of trusted endpoints is not allowed on other host operating systems.

On Workstation 8 and Fusion 4, a host application running with the same user ID as the virtual machine is considered trusted.

The VMCI Sockets API also supports the notion of reserved ports (ports numbers under 1024), where a process must have capability `CAP_NET_BIND_SERVICE` so it can bind to a port within the reserved < 1024 port range. On Windows, only members of the Administrator group are allowed to bind to ports under 1024.

Appendix: Learning More About Sockets

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

- [“About Berkeley Sockets and Winsock”](#) on page 25
- [“Short Introduction to Sockets”](#) on page 26

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

A

about VMCI sockets 7
accept() 15
address structure for sockets 12
AF_INET and PF_INET 25
AF_INET and VMCI sockets 11
afVMCI from VMCI_Sock_GetAFValue() 12, 14, 17
AMQP (advanced message queuing protocol) 9
Apache and Firefox VMCI sockets 9

B

bind() 12, 15, 20
books about sockets 25

C

close() 16, 18, 21
connect() 17
connectionless socket 12
connection-oriented socket 12
context ID (CID) summary 11
context ID, VMADDR_CID_ANY 12, 15

D

datagram VMCI sockets 19

E

ESX/ESXi support for VMCI 8

F

Firefox and Apache VMCI sockets 9
flowchart of datagram sockets 19
flowchart of stream sockets 13

G

getsockname() 16, 21
getsockopt() 14
guest to host communications 8, 12

H

hidden information about sockets 25
host to guest communications 8, 12

L

listen() 15

N

NFS and datagram VMCI sockets 10

P

PF_INET and AF_INET 25
poll() 16, 18
port number, VMADDR_PORT_ANY 12, 15
porting sockets applications 11

R

RabbitMQ 9, 10
read() 16, 18
recv() 16, 18
recvfrom() 21
release contents 8

S

select() 15
send() 16, 17
sendto() 21, 22
setsockopt() 14
SO_VMCI_BUFFER_SIZE 14
SOCK_DGRAM 12
socket() 12, 14, 17, 20, 21
stream VMCI sockets 13

U

use cases for VMCI sockets 9

V

VMCI library deprecated 7
VMCI_Sock_GetAFValue() 14, 17, 20
VMCI_Sock_GetLocalCID() 11, 15, 20

W

Web resources about sockets 25
write() 16, 18
WSAStartup() 14, 17, 20

